**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ТВ**

**отчЁт**

**по лабораторной работе № 5**

**по дисциплине «Интеллектуальный анализ данных»**

**Тема: Регрессионный анализ**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9105 |  | Шаривзянов Д. Р. |
|  |  | Чугунов Р. |
| Преподаватель |  | Поздеев А. А. |

Санкт-Петербург

2024

РЕГРЕССИОННЫЙ АНАЛИЗ

**Цель работы:** моделирование алгоритма коррекции хроматических аберраций на основе регрессионного анализа.

Код программы:

#ifndef REGRESSION\_H

#define REGRESSION\_H

#include <string>

#include <vector>

#include <opencv2\opencv.hpp>

#include <io.h>

#include <iomanip> // Для std::scientific и std::setprecision

#include <direct.h>

#include "logger.hpp"

using namespace std;

using namespace cv;

string output\_path = "output/Lab 5 Regression/";

class Image5 {

public:

    Mat bgr, r, g, b, r\_cor, b\_cor, bgr\_cor;

    int blockSize, numberOfPoints; // размер блока и количество точек для формирования полинома

    vector<Point> clickedPoints; // список кликнутых координат

    struct ShiftData {

    Point point;    // Координаты точки

    Point2f shift;  // Смещение Δx, Δy

    };

    vector<ShiftData> shiftListRed; // список смещений красного канала

    vector<ShiftData> shiftListBlue; // список смещений синего канала

};

/\*\*

 \* \brief Функция для преобразования cv::Mat в строку

 \*

 \* \param[in] mat Исходный объект cv::Mat, который нужно преобразовать в строку.

 \* \return Строковое представление матрицы, которое можно использовать для логирования.

 \*

 \* Эта функция принимает объект cv::Mat и возвращает его строковое представление.

 \* В строковом представлении указываются количество строк, количество столбцов, тип матрицы,

 \* а также значения элементов матрицы.

 \*

 \*/

std::string matToString(const cv::Mat& mat) {

    std::ostringstream oss;

    // Записываем количество строк, количество столбцов и тип матрицы

    oss << "Rows: " << mat.rows << ", Cols: " << mat.cols << ", Type: " << mat.type() << "\n";

    // Устанавливаем научный формат и точность до двух знаков после запятой

    oss << std::scientific << std::setprecision(2);

    // Обходим все элементы матрицы и записываем их значения в строку

    for (int i = 0; i < mat.rows; ++i) {

        for (int j = 0; j < mat.cols; ++j) {

            // Предполагается, что матрица имеет тип CV\_64F. Замените на правильный тип при необходимости.

            oss << mat.at<double>(i, j) << " ";

        }

        oss << "\n"; // Переход на новую строку после каждой строки матрицы

    }

    return oss.str(); // Возвращаем итоговую строку

}

// Функция для логирования cv::Mat

void logMatrix(const std::string& name, const cv::Mat& mat) {

    logger.info("Matrix {}:\n{}", name, matToString(mat));

}

/\*\*

 \* \brief Разбивает исходное изображение на три цветовых канала и сохраняет их как цветные изображения.

 \*

 \* \param src Исходное изображение.

 \* \param red Канал красного цвета.

 \* \param green Канал зелёного цвета.

 \* \param blue Канал синего цвета.

 \*

 \*/

void splitChannels(const Mat& src, Mat& red, Mat& green, Mat& blue) {

    logger.info("Разбиение изображения на каналы.");

    vector<Mat> channels;

    split(src, channels); // Разбиваем изображение на отдельные каналы.

    blue = channels[0];   // Извлекаем синий канал.

    green = channels[1];  // Извлекаем зелёный канал.

    red = channels[2];    // Извлекаем красный канал.

    // Создание цветных изображений для каждого канала

    Mat blueColor, greenColor, redColor;

    // Создаём пустые матрицы того же размера, что и исходное изображение

    Mat zero = Mat::zeros(src.size(), CV\_8UC1);

    // Канал Blue (синий)

    vector<Mat> blueChannels = {blue, zero, zero}; // Формируем цветное изображение для синего канала.

    merge(blueChannels, blueColor); // Объединяем каналы в изображение.

    imshow("Синий канал (цветной)", blueColor);

    imwrite(output\_path + "blue\_color.png", blueColor); // Сохраняем изображение синего канала.

    // Канал Green (зелёный)

    vector<Mat> greenChannels = {zero, green, zero}; // Формируем цветное изображение для зелёного канала.

    merge(greenChannels, greenColor); // Объединяем каналы в изображение.

    imshow("Зелёный канал (цветной)", greenColor);

    imwrite(output\_path + "green\_color.png", greenColor); // Сохраняем изображение зелёного канала.

    // Канал Red (красный)

    vector<Mat> redChannels = {zero, zero, red}; // Формируем цветное изображение для красного канала.

    merge(redChannels, redColor); // Объединяем каналы в изображение.

    imshow("Красный канал (цветной)", redColor);

    imwrite(output\_path + "red\_color.png", redColor); // Сохраняем изображение красного канала.

}

/\*\*

 \* @brief Функция для наложения красного канала на зелёный

 \* @param green матрица зелёного канала

 \* @param red матрица красного канала

 \* @param output матрица для хранения результата

 \*

 \* Функция создает пустое изображение с тремя каналами,

 \* где синий канал (нулевой) заполняется нулями, зелёный канал (опорный) - копируется из green,

 \* а красный канал - копируется из red.

 \* Затем функция собирает изображение из трёх каналов.

 \*/

void overlayRedOnGreen(const Mat& green, const Mat& red, Mat& output) {

    // Создаём пустое изображение с тремя каналами

    vector<Mat> channels(3);

    channels[0] = Mat::zeros(green.size(), green.type()); // Синий канал (нулевой)

    channels[1] = green.clone();                          // Зелёный канал (опорный)

    channels[2] = red.clone();                            // Красный канал

    // Собираем изображение из трёх каналов

    merge(channels, output);

}

/\*\*

 \* @brief Функция для наложения синего канала на зелёный

 \* @param green матрица зелёного канала

 \* @param blue матрица синего канала

 \* @param output матрица для хранения результата

 \*

 \* Функция создает пустое изображение с тремя каналами,

 \* где синий канал (нулевой) заполняется копией синего канала blue,

 \* зелёный канал (опорный) - копируется из green,

 \* а красный канал заполняется нулями.

 \* Затем функция собирает изображение из трёх каналов.

 \*/

void overlayBlueOnGreen(const Mat& green, const Mat& blue, Mat& output) {

    // Создаём пустое изображение с тремя каналами

    vector<Mat> channels(3);

    channels[0] = blue.clone();                             // Синий канал (нулевой)

    channels[1] = green.clone();                            // Зелёный канал (опорный)

    channels[2] = Mat::zeros(green.size(), green.type());   // Красный канал

    // Собираем изображение из трёх каналов

    merge(channels, output);

}

/\*\*

 \* @brief Вычисляет смещение (Δx, Δy) между зелёным каналом и красным/синим каналом

 \*

 \* @param green матрица зелёного канала

 \* @param shiftedChannel матрица красного/синего канала

 \* @param center координаты центра блока

 \* @param blockSize размер блока

 \* @param shiftList список смещений для каждого блока

 \*

 \* Функция вычисляет смещение (Δx, Δy) между зелёным каналом и красным/синим каналом.

 \* Она создает матрицу блока зелёного канала, а затем ищет соответствующий блок в красном/синем канале,

 \* используя математическое сходство. Затем она вычисляет смещение между центром блока зелёного канала

 \* и центром соответствующего блока красного/синего канала.

 \*/

void calculateBlockShift(const Mat& green, const Mat& shiftedChannel, Point center, int blockSize, vector<Image5::ShiftData>& shiftList) {

    // Вычисляем половины блока

    int halfBlock = blockSize / 2;

    // Проверка границ

    int startX = max(center.x - halfBlock, 0);

    int startY = max(center.y - halfBlock, 0);

    int endX = min(center.x + halfBlock, green.cols - 1);

    int endY = min(center.y + halfBlock, green.rows - 1);

    // Создаём матрицу блока зелёного канала

    Mat greenBlock = green(Rect(startX, startY, endX - startX, endY - startY)).clone();

    // Вычисляем размер поиска

    int searchSize = blockSize \* 2;

    int searchStartX = max(center.x - searchSize / 2, 0);

    int searchStartY = max(center.y - searchSize / 2, 0);

    int searchEndX = min(center.x + searchSize / 2, shiftedChannel.cols - 1);

    int searchEndY = min(center.y + searchSize / 2, shiftedChannel.rows - 1);

    // Создаём матрицу поиска для красного/синего канала

    Mat redSearchArea = shiftedChannel(Rect(searchStartX, searchStartY, searchEndX - searchStartX, searchEndY - searchStartY)).clone();

    // Вычисляем корреляцию между матрицами

    Mat result;

    matchTemplate(redSearchArea, greenBlock, result, TM\_CCORR\_NORMED);

    // Ищем максимум корреляции

    Point maxLoc;

    minMaxLoc(result, nullptr, nullptr, nullptr, &maxLoc);

    // Вычисляем смещение

    Point2f shift;

    shift.x = (maxLoc.x + searchStartX) - startX;

    shift.y = (maxLoc.y + searchStartY) - startY;

    // Добавляем результат в список

    shiftList.push\_back({center, shift});

}

/\*\*

 \* @brief Функция для сборки матрицы F для вычисления коэффициентов полинома

 \*

 \* @param points список точек для которых нужно собрать матрицу F

 \*

 \* Функция собирает матрицу F, которая будет использоваться для вычисления коэффициентов

 \* корректирующего полинома. Матрица F - это матрица, состоящая из 10 столбцов,

 \* каждый из которых - это какой-то из членов полинома (1, x, y, x\*y, x^2, y^2, x^2\*y, x\*y^2, x^3, y^3).

 \*

 \* @return матрица F

 \*/

Mat buildMatrixF(const vector<Point2f>& points) {

    int N = points.size();

    Mat F = Mat::zeros(N, 10, CV\_32F);

    for (int i = 0; i < N; ++i) {

        float x = points[i].x, y = points[i].y;

        F.at<float>(i, 0) = 1.0;        // 1

        F.at<float>(i, 1) = x;          // x

        F.at<float>(i, 2) = y;          // y

        F.at<float>(i, 3) = x \* y;      // x\*y

        F.at<float>(i, 4) = x \* x;      // x^2

        F.at<float>(i, 5) = y \* y;      // y^2

        F.at<float>(i, 6) = x \* x \* y;  // x^2\*y

        F.at<float>(i, 7) = x \* y \* y;  // x\*y^2

        F.at<float>(i, 8) = x \* x \* x;  // x^3

        F.at<float>(i, 9) = y \* y \* y;  // y^3

    }

    return F;

}

/\*\*

 \* @brief Вычисляет коэффициенты полинома методом наименьших квадратов.

 \*

 \* @param F матрица F, содержащая члены полинома.

 \* @param delta вектор смещений (Δx или Δy).

 \* @return Матрица коэффициентов A.

 \*

 \* Функция использует метод наименьших квадратов для вычисления коэффициентов

 \* полинома, минимизирующего разницу между наблюдаемыми и предсказанными значениями.

 \* Для этого она вычисляет псевдообратную матрицу F и умножает её на вектор delta.

 \*/

Mat computeCoefficients(const Mat& F, const Mat& delta) {

    Mat FT, FTF\_inv, FT\_delta, A;

    transpose(F, FT);                               // Транспонирование матрицы F для получения F^T

    Mat FTF = FT \* F;                               // Умножение F^T на F

    invert(FTF, FTF\_inv, DECOMP\_SVD);               // Вычисление псевдообратной матрицы (F^T \* F)^-1 с использованием SVD

    FT\_delta = FT \* delta;                          // Умножение F^T на вектор delta

    A = FTF\_inv \* FT\_delta;                         // Вычисление коэффициентов A = (F^T \* F)^-1 \* F^T \* delta

    return A;                                       // Возвращаем матрицу коэффициентов

}

/\*\*

 \* @brief Преобразует канал изображения, используя коэффициенты полинома.

 \*

 \* @param channel канал изображения.

 \* @param Ax матрица коэффициентов для вычисления Δx.

 \* @param Ay матрица коэффициентов для вычисления Δy.

 \* @param transformedRed преобразованный канал.

 \*

 \* Функция преобразует канал изображения, используя коэффициенты полинома,

 \* вычисленные ранее. Она вычисляет смещения Δx и Δy на основе полинома,

 \* используя матрицы Ax и Ay. Затем она использует эти смещения для

 \* интерполяции соответствующих значений в канале.

 \*/

void transformChannel(const Mat& channel, const Mat& Ax, const Mat& Ay, Mat& transformedRed) {

    transformedRed = Mat::zeros(channel.size(), channel.type());

    for (int y = 0; y < channel.rows; ++y) {

        for (int x = 0; x < channel.cols; ++x) {

            // Вычисляем смещения Δx и Δy на основе полинома

            float deltaX = Ax.at<float>(0) + Ax.at<float>(1) \* x + Ax.at<float>(2) \* y +

                           Ax.at<float>(3) \* x \* y + Ax.at<float>(4) \* x \* x +

                           Ax.at<float>(5) \* y \* y + Ax.at<float>(6) \* x \* x \* y +

                           Ax.at<float>(7) \* x \* y \* y + Ax.at<float>(8) \* x \* x \* x +

                           Ax.at<float>(9) \* y \* y \* y;

            float deltaY = Ay.at<float>(0) + Ay.at<float>(1) \* x + Ay.at<float>(2) \* y +

                           Ay.at<float>(3) \* x \* y + Ay.at<float>(4) \* x \* x +

                           Ay.at<float>(5) \* y \* y + Ay.at<float>(6) \* x \* x \* y +

                           Ay.at<float>(7) \* x \* y \* y + Ay.at<float>(8) \* x \* x \* x +

                           Ay.at<float>(9) \* y \* y \* y;

            // Новые координаты пикселя

            float newX = x + deltaX;

            float newY = y + deltaY;

            // Проверка границ изображения

            if (newX >= 0 && newX < channel.cols - 1 && newY >= 0 && newY < channel.rows - 1) {

                // Билинейная интерполяция

                int x0 = static\_cast<int>(newX);

                int y0 = static\_cast<int>(newY);

                float alpha = newX - x0; // Доля по X

                float beta = newY - y0;  // Доля по Y

                // Интерполяция на основе четырёх соседних пикселей

                float value = (1 - alpha) \* (1 - beta) \* channel.at<uchar>(y0, x0) +

                              alpha \* (1 - beta) \* channel.at<uchar>(y0, x0 + 1) +

                              (1 - alpha) \* beta \* channel.at<uchar>(y0 + 1, x0) +

                              alpha \* beta \* channel.at<uchar>(y0 + 1, x0 + 1);

                transformedRed.at<uchar>(y, x) = static\_cast<uchar>(value);

            }

        }

    }

}

/\*\*

 \* @brief Функция коррекции канала изображения

 \*

 \* @param green зелёный канал изображения

 \* @param channel канал, который корректируется

 \* @param shiftList список смещений (Δx, Δy) для каждого блока

 \* @param correctedChannel матрица для хранения результата

 \* @param channelName имя канала, который корректируется

 \*

 \* Функция принимает на вход канал изображения, зелёный канал, список смещений

 \* (Δx, Δy) для каждого блока, матрицу для хранения результата и имя канала,

 \* который корректируется. Функция строит матрицу F из списка смещений,

 \* вычисляет коэффициенты полинома на основе матрицы F, трансформирует

 \* канал изображения, используя коэффициенты полинома, и отображает

 \* результат с наложенным каналом.

 \*

 \*/

void correctChannel(const Mat& green, const Mat& channel, const vector<Image5::ShiftData>& shiftList,

                    Mat& correctedChannel, const string& channelName) {

    vector<Point2f> points;

    vector<float> deltaX, deltaY;

    // Извлекаем точки и их смещения

    for (const auto& data : shiftList) {

        points.push\_back(data.point);

        deltaX.push\_back(data.shift.x);

        deltaY.push\_back(data.shift.y);

    }

    // Строим матрицу F и вычисляем коэффициенты

    Mat F = buildMatrixF(points);

    Mat deltaXMat = Mat(deltaX).reshape(1, deltaX.size());

    Mat deltaYMat = Mat(deltaY).reshape(1, deltaY.size());

    Mat Ax = computeCoefficients(F, deltaXMat);

    Mat Ay = computeCoefficients(F, deltaYMat);

    // Выводим коэффициенты

    logMatrix("F", F);

    // Трансформируем канал

    transformChannel(channel, Ax, Ay, correctedChannel);

    // Создаём изображение с наложенным каналом

    Mat correctedImage;

    if (channelName == "Red") {

        overlayRedOnGreen(green, correctedChannel, correctedImage);

    } else if (channelName == "Blue") {

        overlayBlueOnGreen(green, correctedChannel, correctedImage);

    }

    // Отображение и сохранение результата

    string windowName = "Corrected " + channelName + " Channel";

    imshow(windowName, correctedImage);

    imwrite(output\_path + "corrected\_" + channelName + "\_channel.png", correctedImage);

    logger.info("Corrected {} channel saved as corrected\_{}\_channel.png", channelName, channelName);

}

/\*\*

 \* @brief Собирает финальное изображение из каналов и отображает его

 \*

 \* Функция assembleAndDisplayResult() предназначена для сборки

 \* финального изображения из каналов, полученных после коррекции

 \* смещений. Она принимает на вход указатель на объект Image5,

 \* собирает из его каналов изображение, отображает его в окне

 \* "Result" и сохраняет в файл result.png.

 \*

 \* @param img указатель на объект Image5

 \*/

void assembleAndDisplayResult(Image5\* img) {

    // Сборка финального изображения из каналов

    // Создаём вектор из каналов изображения

    vector<Mat> channels = {img->b\_cor.clone(), img->g.clone(), img->r\_cor.clone()};

    // Собираем каналы в единое изображение

    merge(channels, img->bgr\_cor);

    // Отображение и сохранение финального изображения

    string imgCorName = "Result";

    imshow(imgCorName, img->bgr\_cor);

    // сохраняем финальное изображение в файл result.png

    imwrite(output\_path + imgCorName + ".png", img->bgr\_cor);

    // логгируем информацию о сохранении

    logger.info("Final corrected image saved as " + imgCorName + ".png");

}

/\*\*

 \* @brief Функция-обработчик события мыши

 \*

 \* @param event тип события (например, EVENT\_LBUTTONDOWN)

 \* @param x координата x клика

 \* @param y координата y клика

 \* @param flags флаги (например, EVENT\_FLAG\_LBUTTON)

 \* @param param указатель на объект Image5

 \*

 \* Функция-обработчик события мыши, вызываемая при клике мышью на

 \* изображении. Она добавляет точку в список кликов, рисует все точки

 \* поверх исходного изображения, вычисляет смещения для красного и

 \* синего каналов, логирует текущие смещения, проверяет заполнение

 \* списка и выполняет коррекцию каналов.

 \*/

static void onMouse(int event, int x, int y, int flags, void\* param) {

    if (event != EVENT\_LBUTTONDOWN) return;

    Image5\* img = (Image5\*)param;

    Point center(x, y);

    // Добавляем точку в список кликов

    img->clickedPoints.push\_back(center);

    // Рисуем все точки поверх исходного изображения

    Mat displayImage = img->bgr.clone();

    for (const auto& point : img->clickedPoints) {

        circle(displayImage, point, 5, Scalar(0, 0, 255), FILLED);

    }

    imshow("Original to click", displayImage);

    // Вычисляем смещения для красного и синего каналов

    calculateBlockShift(img->g, img->r, center, img->blockSize, img->shiftListRed);

    calculateBlockShift(img->g, img->b, center, img->blockSize, img->shiftListBlue);

    // Логируем текущие смещения

    const auto& lastShiftRed = img->shiftListRed.back().shift;

    const auto& lastShiftBlue = img->shiftListBlue.back().shift;

    logger.info("{} point out of {} (Pos=({},{}))\tshift red: Δx = {:.2f}, Δy = {:.2f}\tshift blue: Δx = {:.2f}, Δy = {:.2f}",

        img->shiftListRed.size(), img->numberOfPoints, x, y, lastShiftRed.x, lastShiftRed.y, lastShiftBlue.x, lastShiftBlue.y);

    // Проверяем заполнение списка и выполняем коррекцию каналов

    if (img->shiftListRed.size() == img->numberOfPoints) {

        correctChannel(img->g, img->r, img->shiftListRed, img->r\_cor, "Red");

        img->shiftListRed.clear();

    }

    if (img->shiftListBlue.size() == img->numberOfPoints) {

        correctChannel(img->g, img->b, img->shiftListBlue, img->b\_cor, "Blue");

        img->shiftListBlue.clear();

    }

    // Сборка итогового изображения, если оба канала готовы

    if (!img->r\_cor.empty() && !img->b\_cor.empty()) {

        assembleAndDisplayResult(img);

    }

}

/\*\*

 \* @brief Функция для выполнения лабораторной работы 5

 \*        (коррекция цветов по полиному)

 \*

 \* @param img\_bgr - входное RGB-изображение

 \* @param scaleFactor - коэффициент масштабирования

 \* @param blockSize - размер блока для сбора данных

 \* @param numberOfPoints - количество точек для сбора данных

 \*

 \* @details

 \* 1. Создаём папку для выходных изображений.

 \* 2. Масштабируем входное изображение.

 \* 3. Разделяем каналы.

 \* 4. Устанавливаем обработчик события мыши.

 \* 5. Ожидаем нажатие клавиши.

 \*/

void lab5\_Regression(const Mat& img\_bgr, int scaleFactor = 1, int blockSize = 16, int numberOfPoints = 50) {

    logger.info("Lab 5: Regression.");

    Image5 img;

    // создаём папку для выходных изображений

    if (\_access(output\_path.c\_str(), 0) != 0) {

        if (\_mkdir(output\_path.c\_str()) == -1) {

            logger.error("Failed to create directory: {}", output\_path);

            return;

        }

        logger.info("Created directory: {}", output\_path);

    }

    // масштабируем входное изображение

    resize(img\_bgr, img.bgr, Size(img\_bgr.cols \* scaleFactor, img\_bgr.rows \* scaleFactor));

    // задаём параметры

    img.blockSize = blockSize;

    img.numberOfPoints = numberOfPoints;

    // рисуем исходное изображение

    string original\_name = "Original to click";

    imshow(original\_name, img.bgr);

    // разделяем каналы

    splitChannels(img.bgr, img.r, img.g, img.b);

    // устанавливаем обработчик события мыши

    setMouseCallback(original\_name, onMouse, (void\*)&img);

    // ожидаем нажатие клавиши

    waitKey();

}

#endif REGRESSION\_H

1. Входные данные

Исходными данными является цветное изображение, содержащее хроматические аберрации.

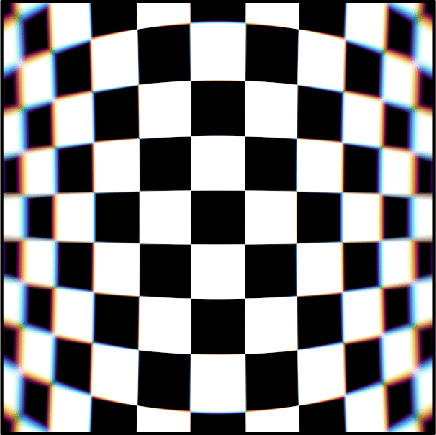


Рисунок 1 – Исходное изображение искажённое хроматическими аберрациями.

1. Разбиение изображения на каналы

После разбиения цветного изображения на отдельные каналы красного, зелёного и синего.

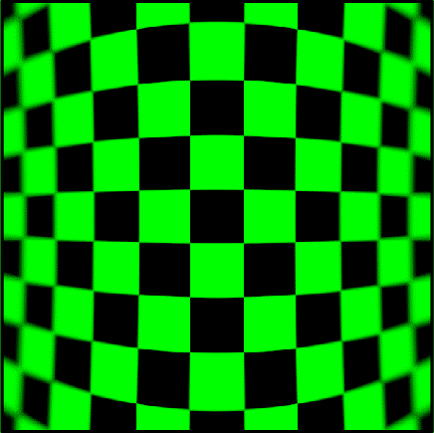
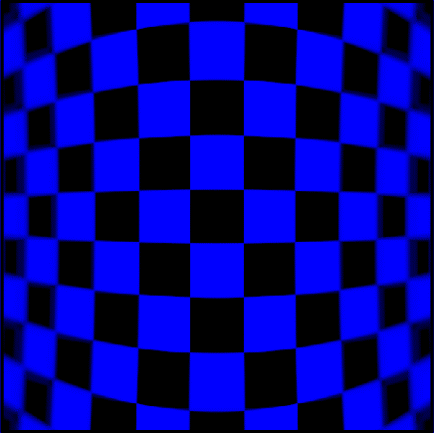
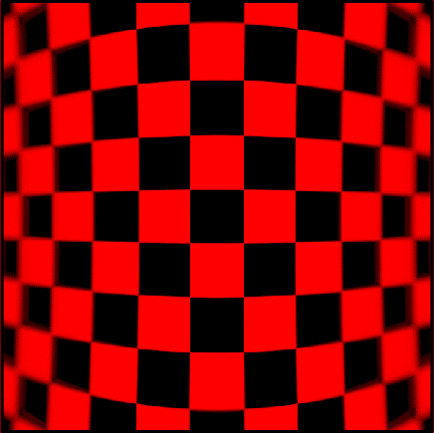


Рисунок 2 – Отдельные каналы исходного изображения.

1. Разметка ключевых точек

Для формирования корректирующего полинома необходимо разметить ключевые точки, в которых будет происходить корреляционный поиск смещения.

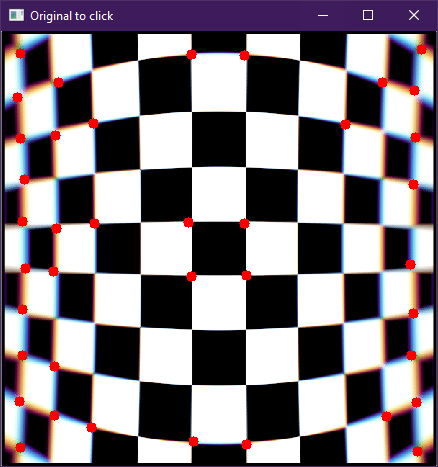


Рисунок 3 – Исходное изображение с размеченными ключевыми точками.

1. Коррекция изображения

Далее происходит формирование корректирующего полинома, заполняется матрица плана эксперимента. На заключительном этапе происходит трансформация каналов. Подставляя в модель значения «новых» координат (*xi*, *yi*), с помощью вычисляемых (Δ*xi*, Δ*yi*) определяют пиксель из исходного изображения, который следует поместить в данные координаты.

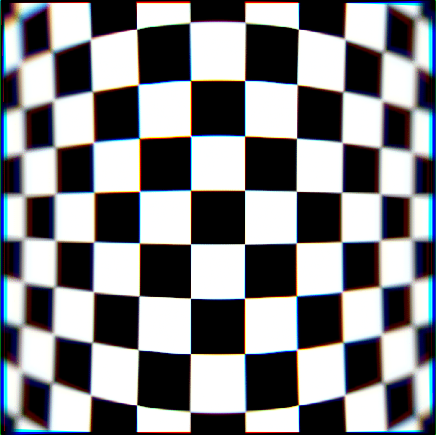


Рисунок 4 – Изображение с минимизированными аберрациями.

**Вывод:** в ходе работы был реализован алгоритм коррекции хроматических аберраций на основе регрессионного анализа.