Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Отчёт

по лабораторной работе №2

Дисциплина: Техническое зрение

Тема: Фильтрация изображения с использованием библиотеки OpenCV

Студент гр. 3331506/70401 Демчева А.А.

Преподаватель Варлашин В.В.

« »\_\_\_\_\_\_\_\_2020 г.

Санкт-Петербург

2020

Задание

Написать программу, реализующую сглаживание изображения с помощью фильтра Гаусса. Размер ядра — 7x7, якорь в центре. Вариант обработки граничных условий — повторение.

При выполнении использовать средства библиотеки *OpenCV 4.0*.

Выполнение задания

При выполнении задания были использованы возможности классов языка C++. Все функции были написаны в виде методов класса *myGaussianBlur*, интерфейс которого был объявлен в заголовочном файле *myGaussianBlur.h,* а реализация описана в файле *myGaussianBlur.cpp.*

Функция *setSrcImage* задает исходное изображение, копируя его в глобальную переменную класса.

Листинг 1 — *Функция* setSrcImage\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

void myGaussianBlur::setSrcImage(Mat srcImage)

{

m\_srcImage = srcImage.clone();

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Функция *setKernel* формирует ядро в виде объекта *Mat* с учетом заданных размеров и величины среднеквадратичного отклонения. Элементы ядра вычисляются по формуле:

где и — расстояние от центра ядра, пикс; — среднеквадратичное отклонение, параметр, определяющий степень сглаживания.

Переменные и рассчитываются с учетом положения системы координат изображения в OpenCV (левый верхний угол) и положения якоря (в центре).

Результат работы функции показан на рисунке 2.

Листинг 2 — Функция setKernel\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

void myGaussianBlur::setKernel(double sigma, int kernelRows, int kernelCols)

{

m\_sigma = sigma;

m\_kernel = Mat(kernelRows, kernelCols, CV\_64F);

int i = 0;

int j = 0;

cout << "Ядро фильтра Гаусса: " << endl;

for (j = 0; j < m\_kernel.rows; j++)

{

int y = j - 3;

for (i = 0; i < m\_kernel.cols; i++)

{

int x = i - 3;

m\_kernel.at<double>(j, i) = (1 / (2 \* CV\_PI \* pow(m\_sigma, 2))) \* exp(-(pow(x, 2) + pow(y, 2)) / (2 \* pow(m\_sigma, 2)));

m\_kernelSum = m\_kernelSum + m\_kernel.at<double>(j, i);

cout << m\_kernel.at<double>(j, i) << " ";

}

cout << endl;

}

cout << endl << endl;

return;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

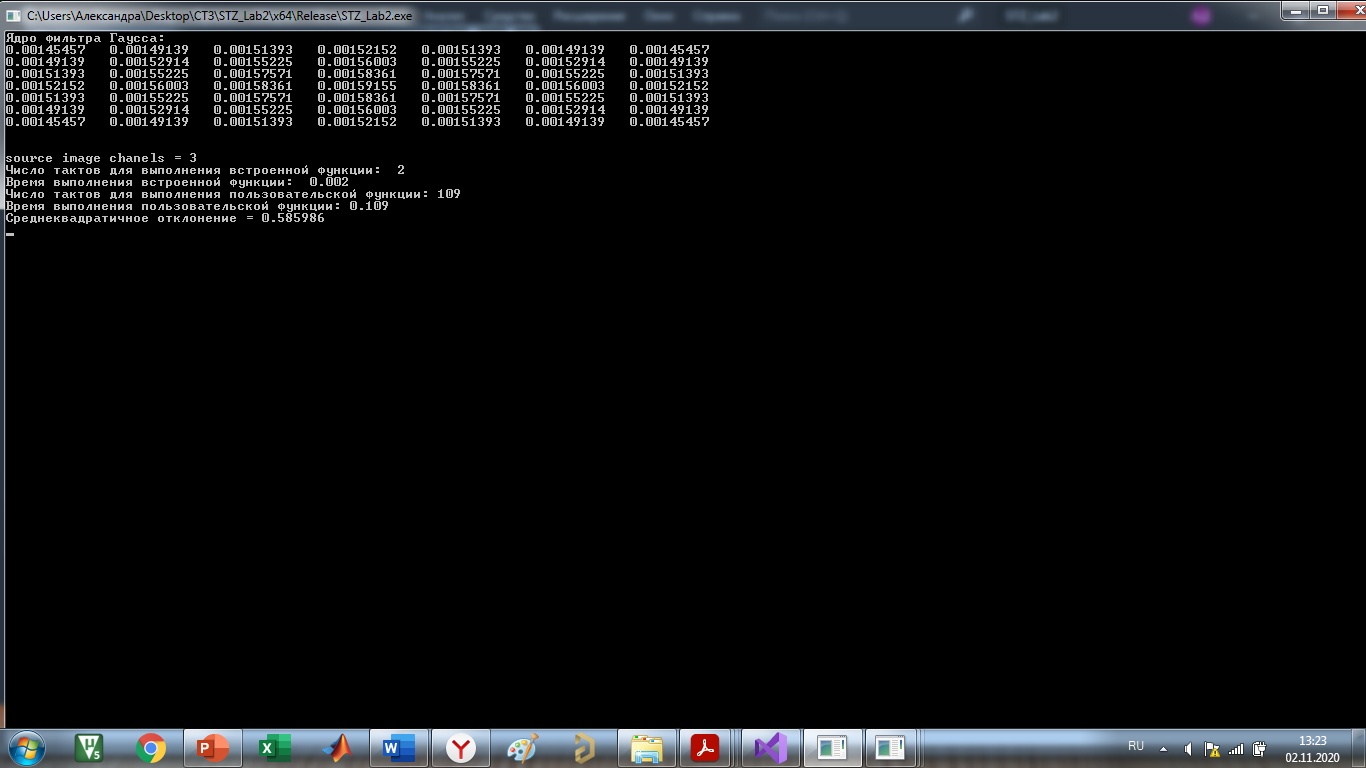


Рисунок 2 — Результат работы функции setKernel

Get-функции *getSrcImg, getExpandedImg, getResultImg, getDiffImg* осуществляют вывод изображения после каждого этапа обработки.

*Листинг 3 — Get-функции класса* myGaussianBlur\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

void myGaussianBlur::getSrcImg()

{

imshow("myGauss source image", m\_srcImage);

}

void myGaussianBlur::getExpandedImg()

{

imshow("myGauss expanded image", m\_extImage);

}

void myGaussianBlur::getResultImg()

{

imshow("myGauss result image", m\_resImage);

}

void myGaussianBlur::getDiffImg()

{

imshow("difference image", m\_diffImg);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Функция *expandImage* реализует обработку граничных условий в соответствии с правилом *border replicate*. Создается дополнительный объект типа *Mat —* расширенное изображение, в центр которого копируется исходное. Оставшиеся пустыми граничные области заполняются значением интенсивности, равным значению соответствующего крайнего пикселя исходного изображения.

*Листинг 4 — Функция expandImage\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

void myGaussianBlur::expandImage()

{

int myGaussRows = m\_srcImage.rows + m\_kernel.rows - 1;

int myGaussCols = m\_srcImage.cols + m\_kernel.cols - 1;

int myGaussType = m\_srcImage.type();

m\_extImage = Mat(myGaussRows, myGaussCols, myGaussType);

m\_Dot = Point2i((m\_kernel.rows / 2), (m\_kernel.cols / 2));

Size2i Size;

Size.width = m\_srcImage.cols;

Size.height = m\_srcImage.rows;

Rect Rectangle(m\_Dot, Size);

Mat Roi(m\_extImage(Rectangle));

m\_srcImage.copyTo(Roi);//Копируем в расширенное изображение исходное

int leftBorder = (m\_kernel.cols / 2);

int rightBorder = (m\_kernel.cols / 2) + m\_srcImage.cols - 1;

int topBorder = (m\_kernel.rows / 2);

int bottomBorder = (m\_kernel.rows / 2) + m\_srcImage.rows - 1;

//верхняя граница

for (int channel = 0; channel < m\_srcImage.channels(); channel++) //каналы пикселя

{

for (int i = leftBorder; i <= rightBorder; i++)

{

for (int j = (topBorder - 1); j >= 0; j--)

{

m\_extImage.at<Vec3b>(j, i)[channel] =

m\_extImage.at<Vec3b>(topBorder,i)[channel];

}

}

}

//нижняя граница

for (int channel = 0; channel < m\_srcImage.channels(); channel++) //каналы пикселя

{

for (int i = leftBorder; i <= rightBorder; i++)

{

for (int j = (bottomBorder + 1); j < m\_extImage.rows; j++)

{

m\_extImage.at<Vec3b>(j, i)[channel] =

m\_extImage.at<Vec3b>(bottomBorder, i)[channel];

}

}

}

//левая граница

for (int channel = 0; channel < m\_srcImage.channels(); channel++) //каналы пикселя

{

for (int j = 0; j < m\_extImage.rows; j++)

{

for (int i = (leftBorder - 1); i >= 0; i--)

{

m\_extImage.at<Vec3b>(j, i)[channel] =

m\_extImage.at<Vec3b>(j,leftBorder)[channel];

}

}

}

//правая граница

for (int channel = 0; channel < m\_srcImage.channels(); channel++) //каналы пикселя

{

for (int j = 0; j < m\_extImage.rows; j++)

{

for (int i = (rightBorder + 1); i < m\_extImage.cols; i++)

{

m\_extImage.at<Vec3b>(j, i)[channel] =

m\_extImage.at<Vec3b>(j, rightBorder)[channel];

}

}

}

return;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Функция *smooth* реализует непосредственно сглаживание — перемещение ядра по исходному изображению, умножение значений интенсивности его пикселей на соответствующий коэффициент ядра, суммирование и запись результата в центральный пиксель.

*Листинг 5 — Функция smooth\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

void myGaussianBlur::smooth()

{

m\_resImage = Mat(m\_srcImage.rows, m\_srcImage.cols, m\_srcImage.type());

double loc\_sum = 0;

int leftBorder = m\_kernel.cols / 2;

int rightBorder = m\_extImage.cols - (m\_kernel.cols / 2) - 1;

int topBorder = m\_kernel.rows / 2;

int bottomBorder = m\_extImage.rows - (m\_kernel.rows / 2) - 1;

int exImX = 0; //Координаты пикселя расширенного изображения

int exImY = 0;

int resImX = 0; //Координаты пикселя итогового изображения

int resImY = 0;

for (m\_anchor.y = topBorder; m\_anchor.y <= bottomBorder; m\_anchor.y++)

{

for (m\_anchor.x = leftBorder; m\_anchor.x <= rightBorder; m\_anchor.x++)

{

for (int channel = 0; channel < m\_resImage.channels(); channel++)

{

//Устанавливаем координаты левой верхней точки на изображении, к которой будет приложена левая верхняя точка маски

exImX = m\_anchor.x - (m\_kernel.cols / 2);

exImY = m\_anchor.y - (m\_kernel.rows / 2);

for (int kerY = 0; kerY < m\_kernel.rows; kerY++)

{

for (int kerX = 0; kerX < m\_kernel.cols; kerX++)

{

loc\_sum = loc\_sum +

(m\_kernel.at<double>(kerY,kerX) \*

\* m\_extImage.at<Vec3b>(exImY, exImX)[channel]);

exImX++;

}

//Возвращаемся в начало строки

exImX = m\_anchor.x - (m\_kernel.cols / 2);

//Переходим к следующей строке

exImY++;

}

resImX = m\_anchor.x - leftBorder;

resImY = m\_anchor.y - topBorder;

m\_resImage.at<Vec3b>(resImY, resImX)[channel] =

(uint8\_t)(loc\_sum/m\_kernelSum);

loc\_sum = 0;

}

}

}

return;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Чтобы убедиться, что написанная программа работает верно, результат применения пользовательского фильтра сравнивается со встроенной функцией *GaussianBlur* библиотеки *OpenCV* с помощью следующих методов класса.

Функция *calcDiff* реализует поэлементное сравнение полученных изображений с помощью метода *absdiff.*

*Листинг 6 — Функция calcDiff\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

void myGaussianBlur::calcDiff(Mat builtInImg)

{

int difRows = m\_srcImage.rows;

int difCols = m\_srcImage.cols;

int difType = m\_srcImage.type();

m\_diffImg = Mat(difRows, difCols, difType);

absdiff(m\_resImage, builtInImg, m\_diffImg);

return;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Функция *squareDeviation* вычисляет среднеквадратичную погрешность фильтрации для каждого канала в соответствии с формулой:

где и — ширина и высота исходного изображения соответственно, пикс; и — интенсивности пикселей изображений, полученных после применения библиотечного и пользовательского фильтров Гаусса. Разность соответствует значению интенсивности пикселей изображения, полученного с помощью функции *absdiff.*

*Листинг 7 — Функция squareDeviation\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

void myGaussianBlur::squareDeviation()

{

double diffSum = 0;

double difSize = (m\_diffImg.rows \* m\_diffImg.cols);

for (int channel = 0; channel < m\_srcImage.channels(); channel++)

{

for (int j = 0; j < m\_diffImg.rows; j++)

{

for (int i = 0; i < m\_diffImg.cols; i++)

{

diffSum = diffSum + pow((m\_diffImg.at<Vec3b>(j, i)[channel]), 2);

}

}

double squareDeviation = sqrt(diffSum / (difSize-1));

cout << "канал " << channel << endl;

cout << "Cреднеквадратичное отклонение = " << squareDeviation << endl;

}

return;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подсчитывается также число тактов и время, затраченное на выполнение пользовательской и встроенной функции фильтра Гаусса — с помощью функции *clock* и макроса *CLOCKS\_PER\_SEC.*

*Листинг 8 — Функция Main\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

int main()

{

setlocale(LC\_ALL, "Russian");

myGaussianBlur myGauss;

double sigma = 10;

int kernelRows = 7;

int kernelCols = 7;

//Ядро

myGauss.setKernel(sigma, kernelRows, kernelCols);

//Оригинальное изображение

Mat image = imread("C:/222.bmp", 1);

cout << "source image chanels = " << image.channels() << endl;

myGauss.setSrcImg(image);

myGauss.getSrcImg();

//Встроенный фильтр

Mat builtInGaussImg = image.clone();

int builtInStart = clock();

GaussianBlur(image, builtInGaussImg, Size2i(7, 7), 10, 10, 1);

int builtInEnd = clock();

imshow("builtInGauss", builtInGaussImg);

cout << "Число тактов для выполнения встроенной функции: " << (builtInEnd –

builtInStart) << endl;

cout << "Время выполнения встроенной функции: " << ((double)(builtInEnd –

builtInStart) / CLOCKS\_PER\_SEC) << endl;

//Пользовательский фильтр Гаусса

myGauss.expandImage();

myGauss.getExpandedImg();

int myGaussStart = clock();

myGauss.smooth();

int myGaussEnd = clock();

myGauss.getResultImg();

cout << "Число тактов для выполнения пользовательской функции: " << (myGaussEnd –

myGaussStart) << endl;

cout << "Время выполнения пользовательской функции: " << ((double)(myGaussEnd –

myGaussStart) / CLOCKS\_PER\_SEC) << endl;

//Вычисляем разницу

myGauss.calcDiff(builtInGaussImg);

myGauss.getDiffImg();

myGauss.squareDeviation();

while (waitKey(1) != 27)

{

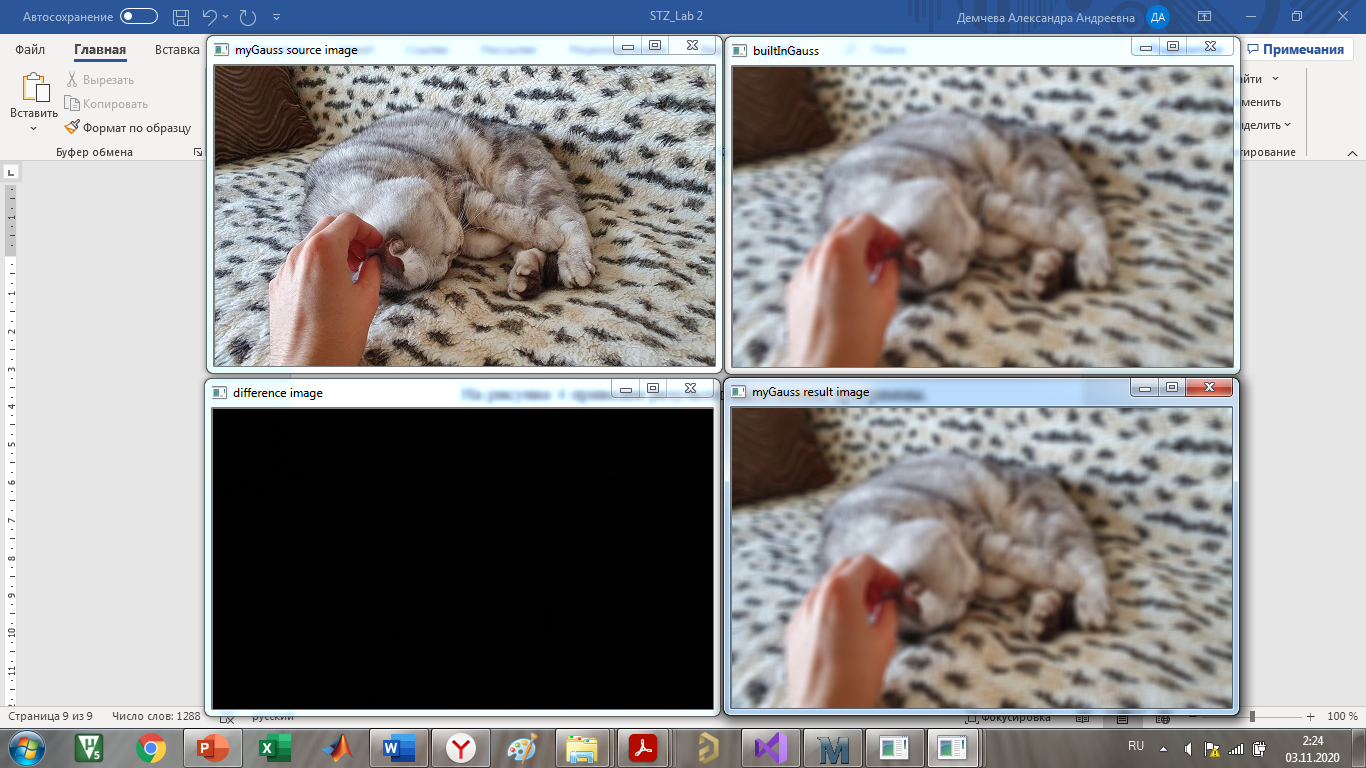
;

}

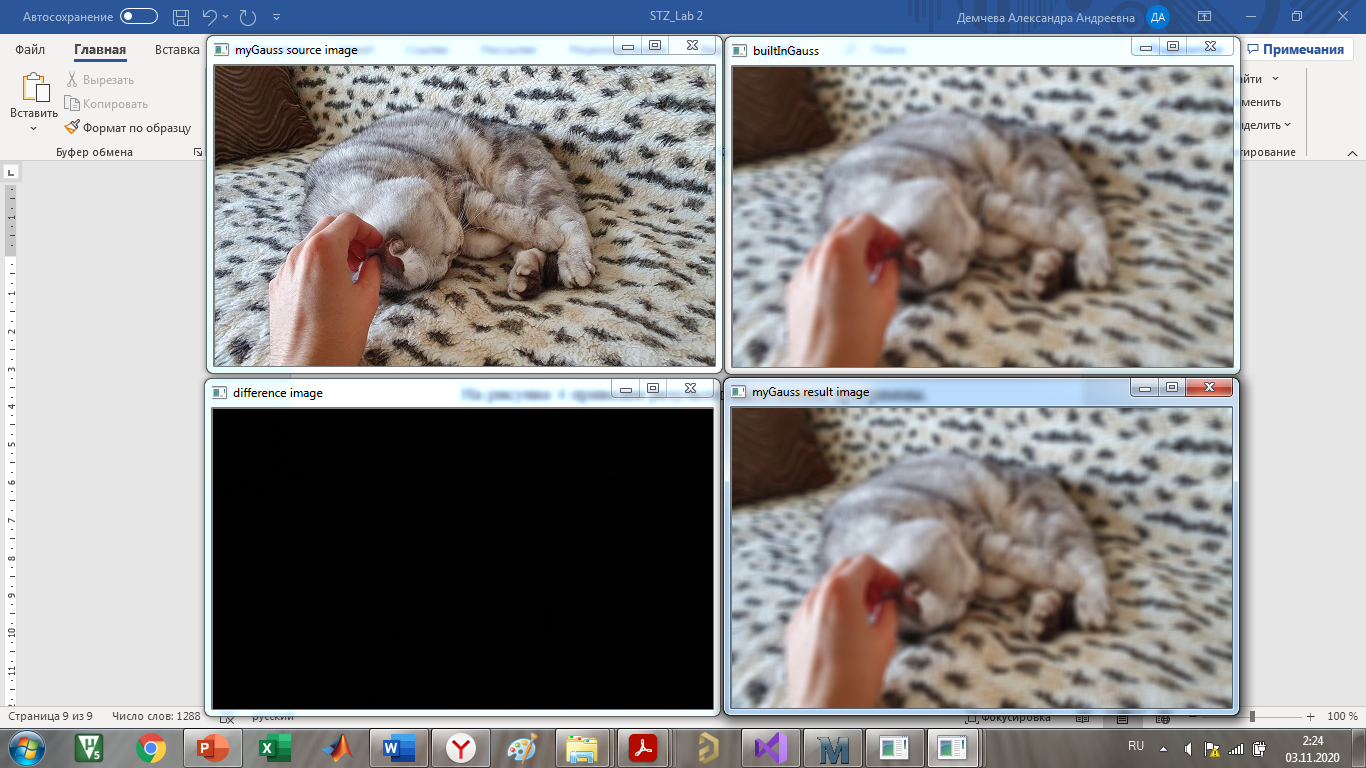
return 0;

}

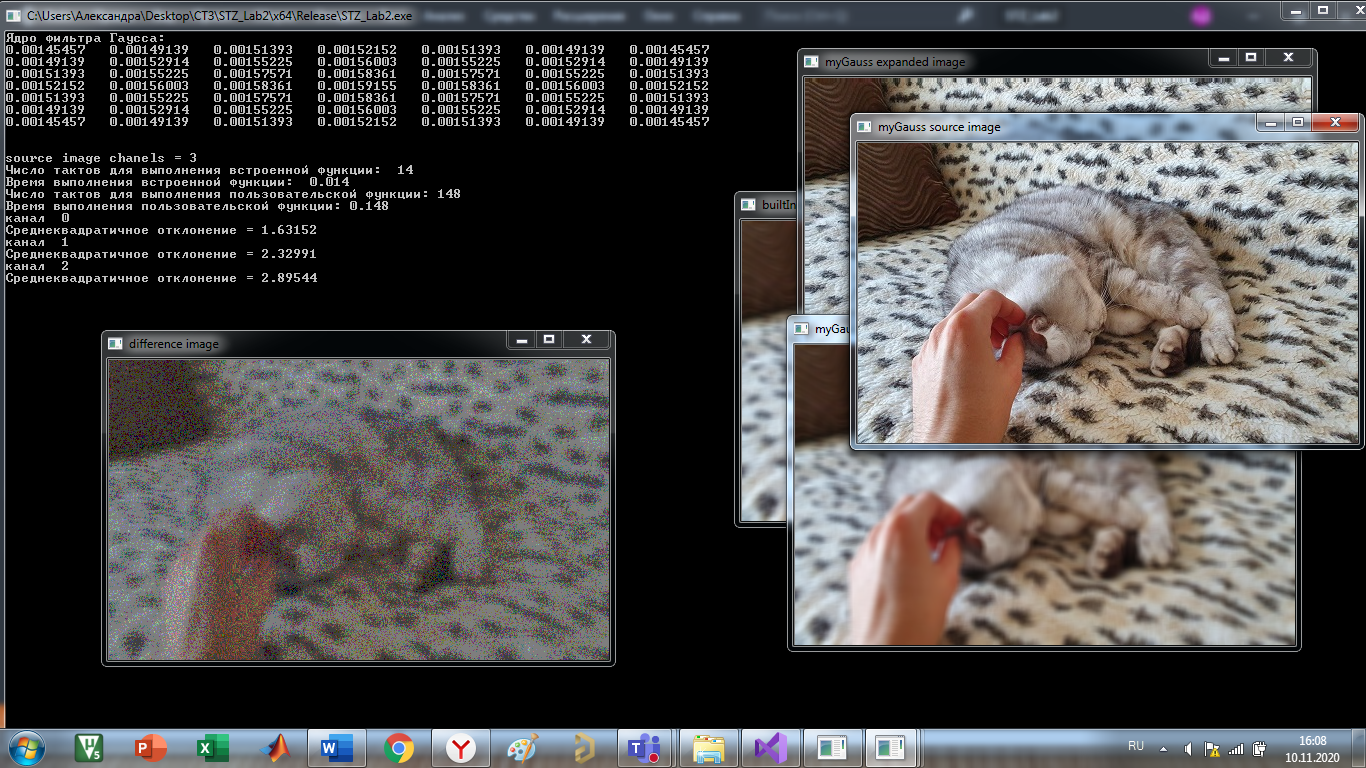
На рисунке 4 приведен результат выполнения программы: а) — исходное изображение; б), г) — результат применения встроенного и пользовательского фильтров соответственно; в) — разница между изображениями, полученными в результате применения фильтров; г) — сравнение времени выполнения функций и числа затраченных тактов.



*а) б)*



*в) г)*



*д)*

Рисунок 4 — Результат выполнения программы

В таблице 1 приведено сравнение скорости выполнения встроенной и пользовательской функций. Как можно заметить, пользовательская функция требует для выполнения почти в 10 раз большего числа тактов и времени.

Таблица 1 — Сравнение встроенного и пользовательского фильтра Гаусса

|  |  |  |
| --- | --- | --- |
| **Параметр сравнения** | **Фильтр** | |
| **Встроенный** | **Пользовательский** |
| Число затраченных тактов | 11 | 109 |
| Время выполнения, с | 0.011 | 0.109 |

Чтобы сделать разницу между примененными фильтрами более заметной, умножим значение яркости в каждом пикселе разностного изображения на некоторый масштабирующий коэффициент. Получим результат, показанный на рисунке 5.

Разницу между итоговыми изображениями можно объяснить округлением чисел типа *double*, а также различиями в реализации алгоритмов пользовательского и встроенного фильтров. Во встроенной функции нельзя задать положения якоря в ядре, кроме того, она использует разделение ядра для ускорения работы алгоритма.

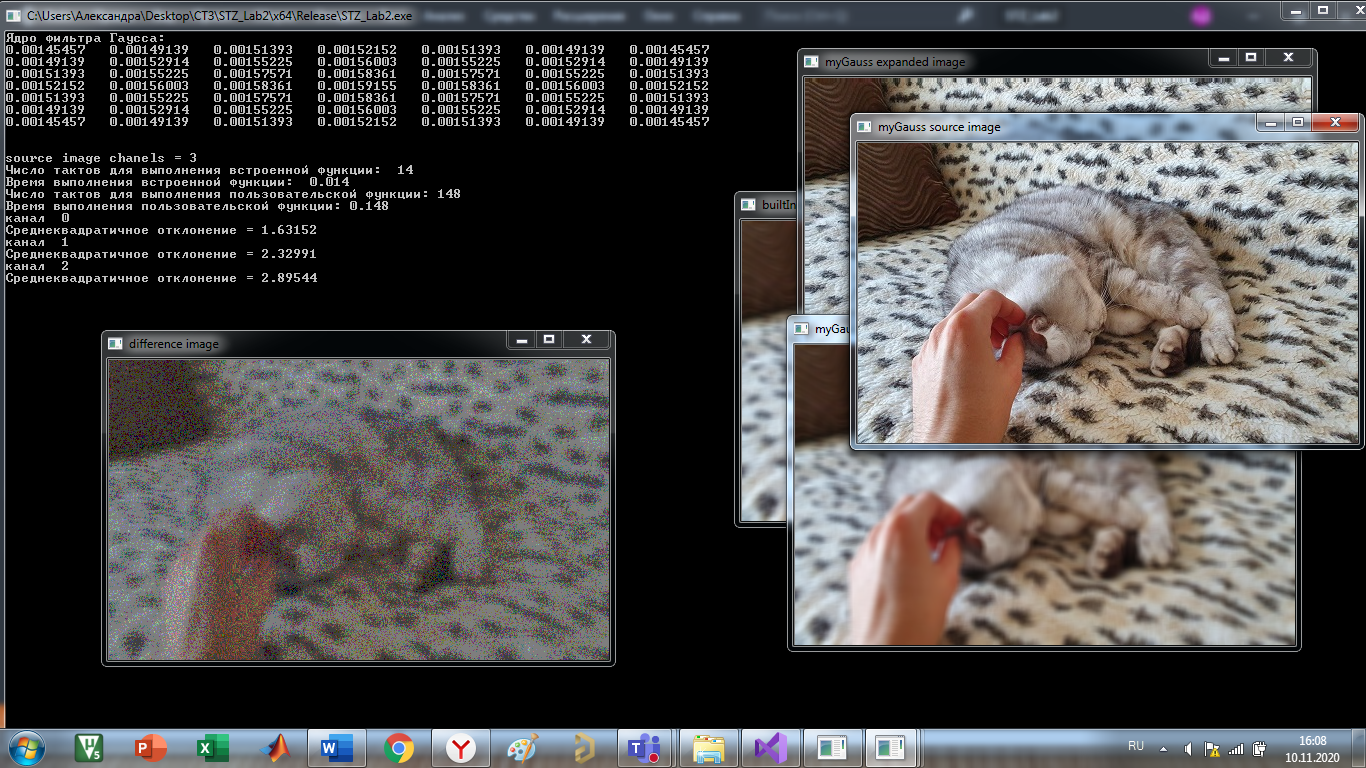


Рисунок 5 — Разница между результатами применения фильтров