САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчет
По лабораторной работе №2
Дисциплина: Разработка графических приложений

Выполнил студент группы: 13541/3: Покатило П.А.

Преподаватель: Абрамов Н.А.

Санкт-Петербург
2018

Задание

Познакомиться с фильтрами сглаживания изображений, такими как Gaussian Blur; Bilateral Filter; Non-local means Filter.

Программа работы

1. Реализовать следующие фильтры сглаживания изображений на языке C++ используя средства библиотеки OpenCV
   - Gaussian Blur
   - Bilateral Filter
   - Non – local means.
2. Сравнить результаты со стандартными функциями библиотеки OpenCV.

Ход работы

Gaussian blur

Фильтр Гаусса — цифровой фильтр размытия изображения, который использует нормальное распределение для вычисления преобразования, применяемого к каждому пикселю изображения. Пиксели, где распределение отлично от нуля используются для построения матрицы свертки, которая применяется к исходному изображению. Значение каждого пикселя становится средневзвешенным для окрестности. Исходное значение пикселя принимает наибольший вес (имеет наивысшее Гауссово значение), и соседние пиксели принимают меньшие веса, в зависимости от расстояния до них.

Сигнатура функции в OpenCV выглядит следующим образом:

```
void GaussianBlur(InputArray src, OutputArray dst, Size ksize,
double sigmaX, double sigmaY=0, int borderType=BORDER_DEFAULT)
```

src - входное изображение

- dst — выходное изображение аналогичного размера
- ksize — размер Гауссова ядра. ksize.width и ksize.height могут отличаться, но они оба должны быть положительными и нечетным.
- sigmaX — стандартное отклонение Гауссова ядра в направлении X.
- sigmaY — стандартное отклонение Гауссова ядра в Y направлении; если sigmaY равен нулю, то устанавливается равным sigmaX, если оба сигмы нули, они вычисляются из ksize.width и ksize.height, соответственно; для того чтобы полностью контролировать результат, независимо от возможных будущих модификаций, рекомендуется указать все ksize, sigmaX и sigmaY.

От выбора размера ядра и стандартного отклонения распределения Гаусса по X и Y направлению будет зависеть результат и сила размытия.

Протестируем программу на сильно зашумленном изображении

Рисунок 1 Исходное изображение

Оно довольно нечеткое, плюс имеются сильный шум, вызванный низким качеством аппарата съемки. Применим к нему стандартный и разработанный фильтры с разными параметрами. Справа – методы библиотеки OpenCV

Рисунок 2 Gaussian Blur, sigma = 0.8

Рисунок 3 Gaussian Blur, sigma=1.0

Фильтр сглаживает видимый шум, однако качество фото остается на прежнем уровне, так как изначально шум очень большой. Но при этом заметны исчезновения мелкого шума, что говорит о том, что фильтр справляется с поставленной задачей

Bilateral filter

Билатеральный фильтр - это нелинейный, сохраняющий края и уменьшающий шум фильтр сглаживания для изображений. Он заменяет интенсивность каждого пикселя средневзвешенным значением интенсивности от соседних пикселей. Этот вес может быть основан на распределении Гаусса. Важно то, что веса зависят не только от евклидова расстояния пикселей, но также и от радиометрических различий (например, различий в диапазоне, таких как интенсивность цвета, расстояние по глубине и т. д.). Это позволяет сохранять острые края. Фильтр нашёл широкое применение во многих задачах по обработке изображений, например, фильтрация шума, редактирование текстуры и тона, оценки оптического потока.

Сигнатура функции OpenCV выглядит так:

```
void bilateralFilter(InputArray src, OutputArray dst, int d,
double sigmaColor, double sigmaSpace, int
borderType=BORDER_DEFAULT)
```

- src – входное изображение.
- dst – выходное изображение того же формата, что и src .
- d – диаметр каждой пиксельной окрестности, которая используется во время фильтрации. Если оно не положительное, оно вычисляется из sigmaSpace .
- sigmaColor – Фильтр сигма в цветовом пространстве. Большее значение параметра означает, что более дальние цвета в SigmaSpace будут смешаны вместе.
- sigmaSpace – Фильтр сигма в координатном пространстве. Большее значение параметра означает дальность влияния пикселей друг на друга. Когда d>0, он

определяет размер окрестности независимо от sigmaSpace. В противном случае, d пропорционален sigmaSpace.

Исходя из документации, для подбора сигм следует устанавливать их одинаковыми. Большое значение сигмы (> 150) максимально преобразует шумы, но за это придется платить четкостью изображения.

Стремление сигмы к нулю делает билатеральный фильтр простым сглаживающим фильтром Гаусса.

Возьмем зашумленное изображения с контрастными участками:

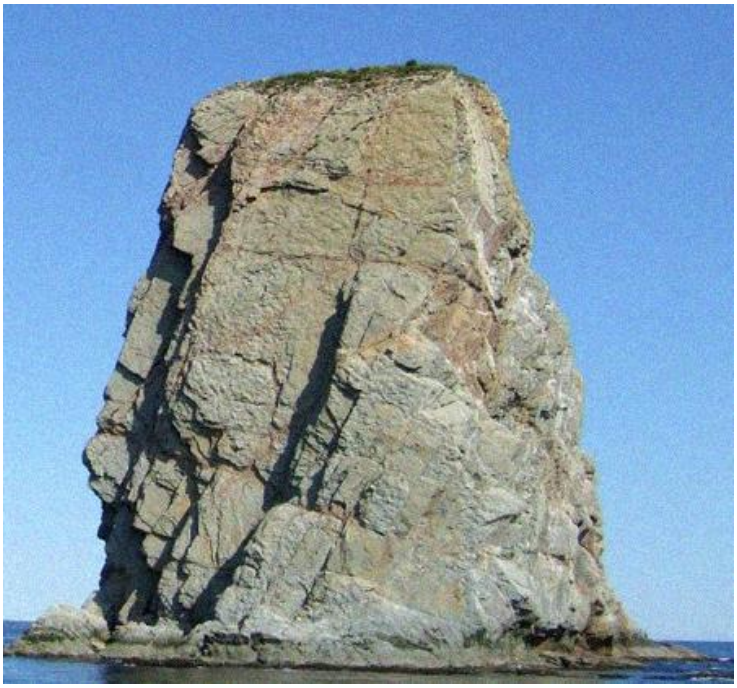

Рисунок 4 Исходное изображение 2

Рассмотрим пример, справа – функция OpenCV

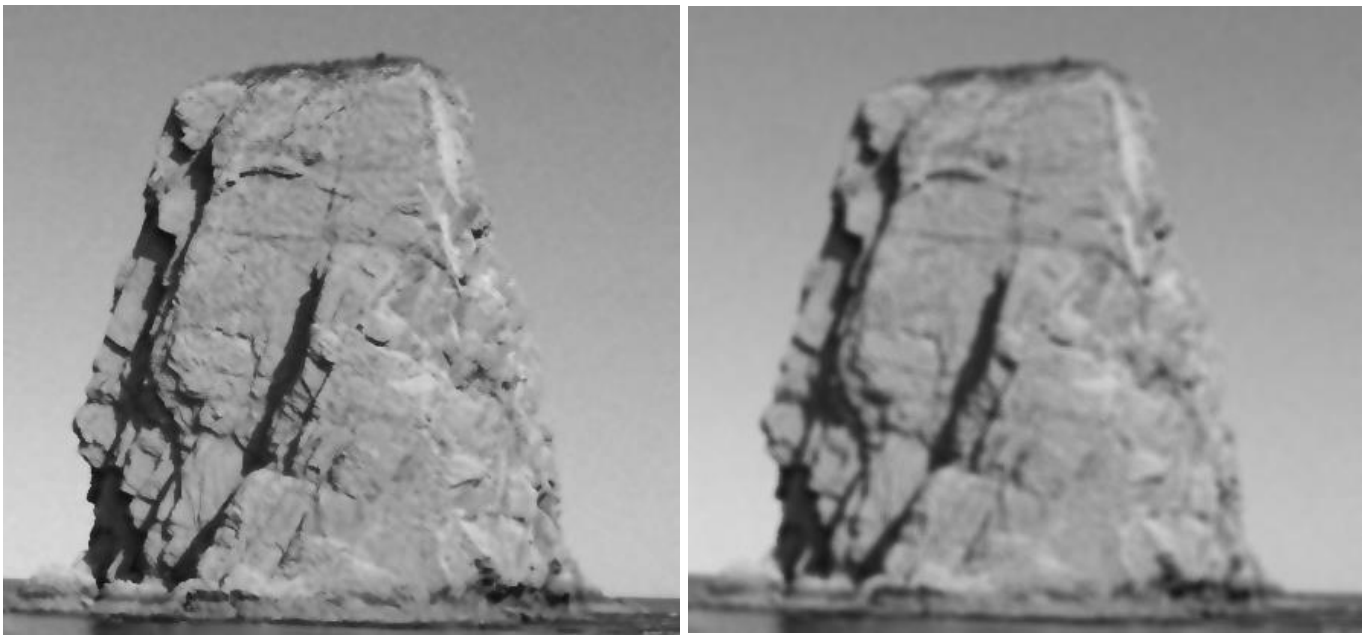

Рисунок 5 Bilateral Filter, d=5, sigma[2] = 50

Странно, но функция OpenCV при одинаковых параметрах выдает худший результат, видимо, особенности реализации. Очень хорошо заметно разделение контрастных участков и их раздельное сглаживание

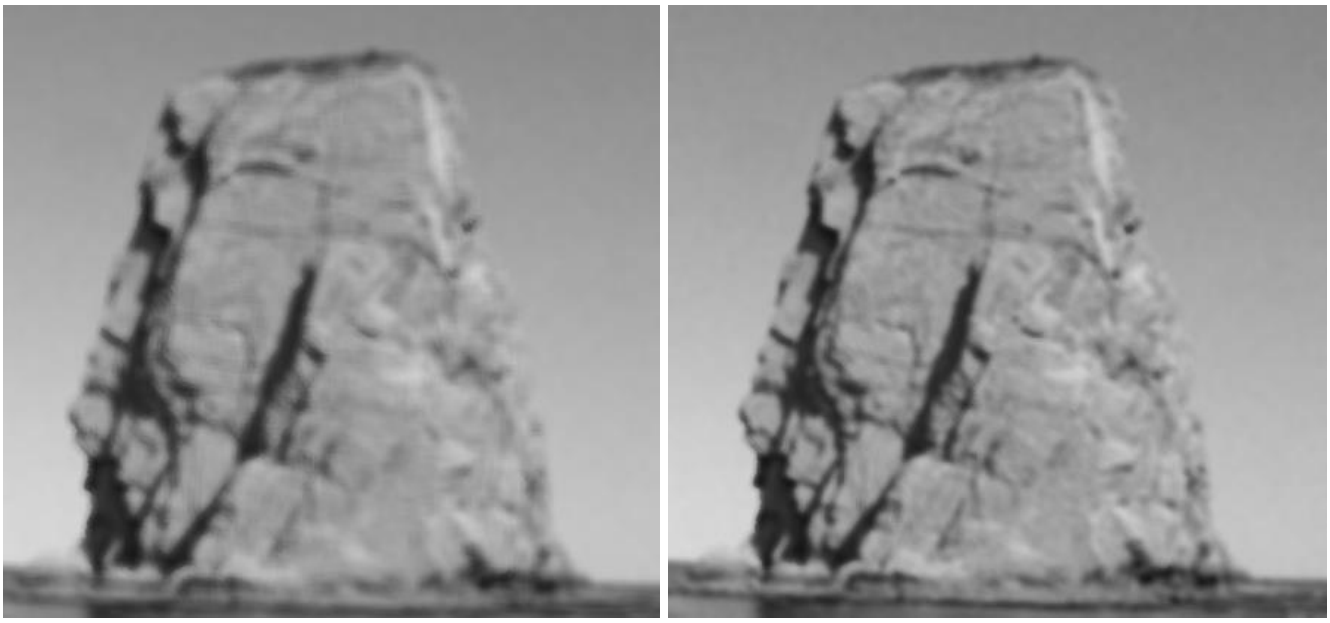

Рисунок 6 Bilateral Filter, d=9, sigma[2] = 150

При увеличении параметров, функция OpenCV начинает выигрывать в качестве. Однако обе реализации справились с устранением шума.

Non-Local Means Filter

Non-Local Means - это алгоритм обработки изображений для уменьшения шума. В отличие от «local means» фильтров, которые принимают среднее значение группы пикселей, окружающих целевой пиксель, для сглаживания изображения, нелокальная фильтрация средних значений берет среднее значение всех пикселей в изображении, взвешенное по степени сходства этих пикселей с целевым пикселем. Это приводит к гораздо большей четкости постфильтрации и меньшей потере детализации изображения по сравнению с локальными средними алгоритмами.

По сравнению с другими хорошо известными методами шумоподавления нелокальные средства добавляют «шум метода» (т.е. ошибку в процессе шумоподавления), который больше похож на белый шум, что желательно, потому что он, как правило, меньше мешает продукту с шумом. Недавно нелокальные средства были расширены для других приложений обработки изображений, таких как деинтерлейсинг и интерполяция.

Сигнатура функции OpenCV для черно-белых изображений:

```
void fastNlMeansDenoising(InputArray src, OutputArray dst,
float h, int templateWindowSize, int searchWindowSize)
```

- src – входное изображение.
- dst – выходное изображение того же размера.
- templateWindowSize - размер в пикселях шаблона, который используется для вычисления весов.

- searchWindowSize - размер в пикселях окна, который используется для вычисления средневзвешенного значения для данного пикселя. Линейно влияет на производительность. Чем больше searchWindowsSize, тем больше время удаления шума.
- h - параметр, регулирующий силу фильтра. Большое значение h идеально удаляет шум, но также удаляет детали изображения, меньшее значение h сохраняет детали, но также сохраняет шум

Исходное изображение:

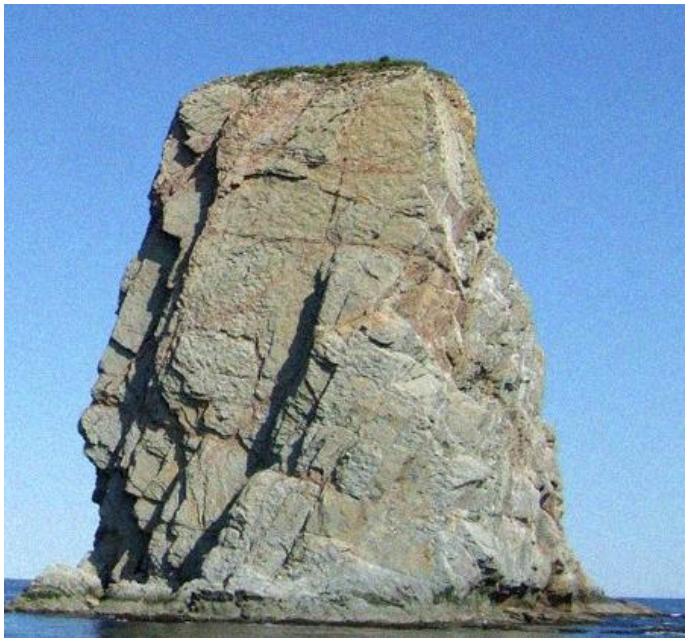

Рисунок 7 Исходное изображение 3

Справа – результат работы функции OpenCV:

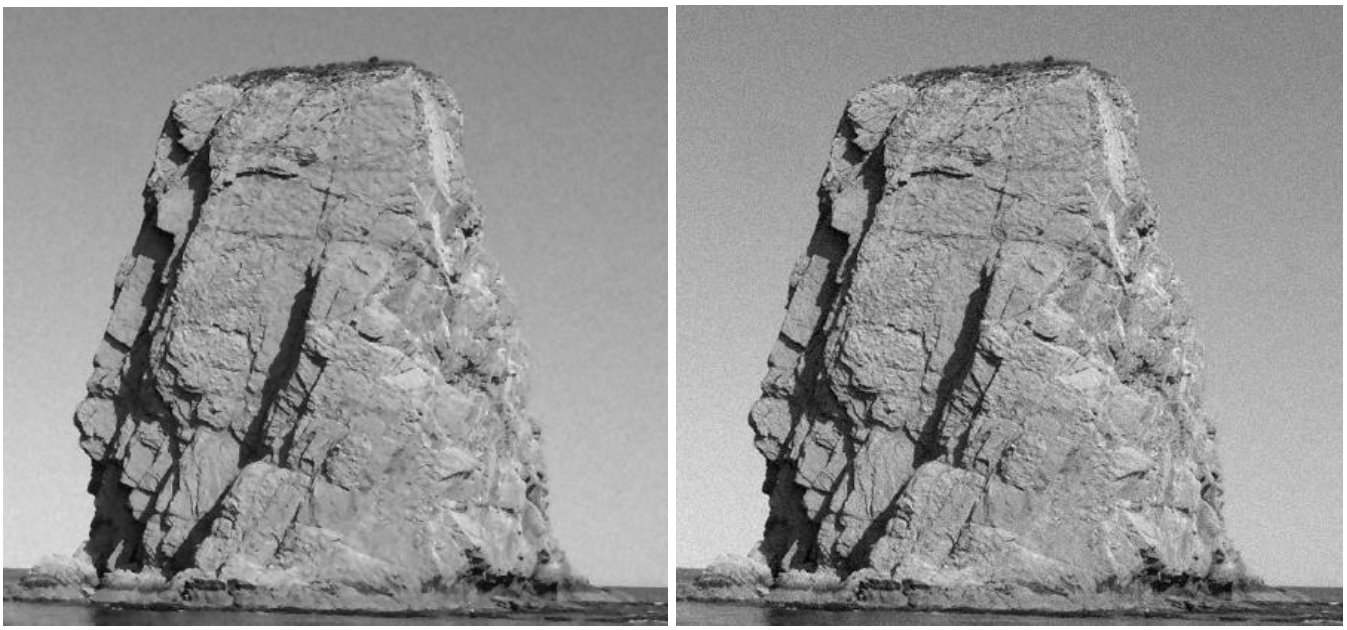

Рисунок 8 NLMeans Filter. h = 3, templateWindowSize = 15, searchWindowSize=15

При малой силе фильтра и размере окна поиска фильтр OpenCV практически не справляется с шумом.

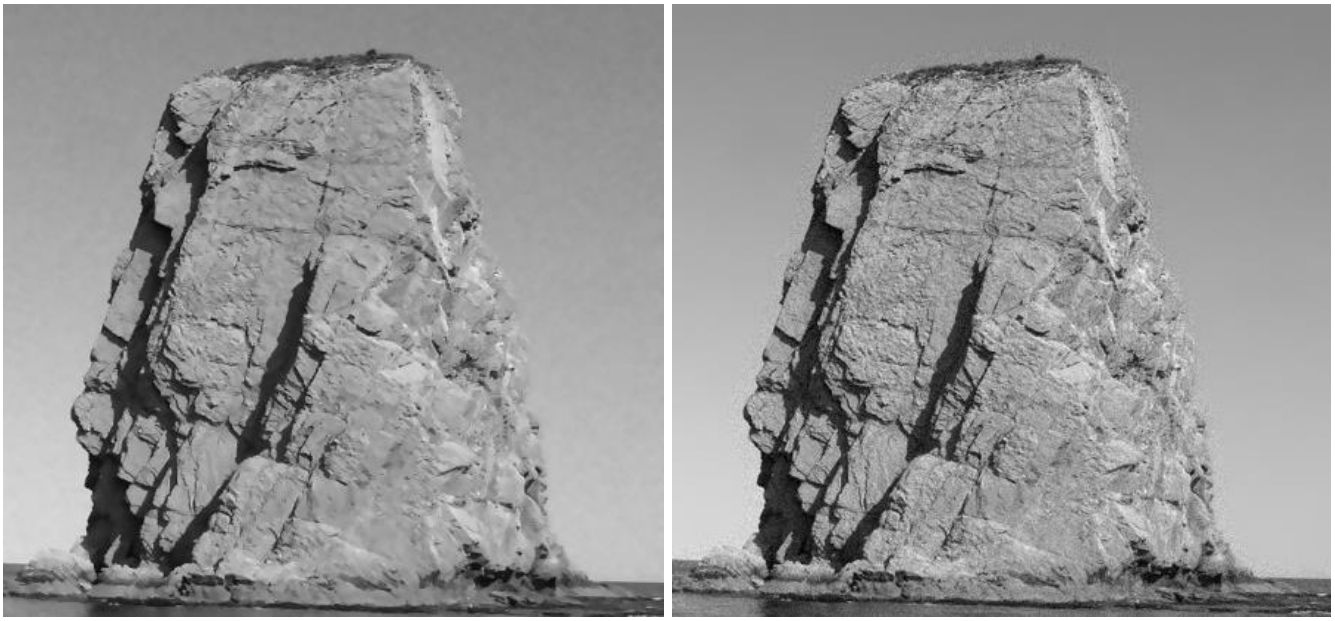

Рисунок 9 NLMeans Filter. h = 5, templateWindowSize = 22, searchWindowSize=22

Однако при увеличении этих параметров мы видим почти идеальный результат, хотя и присутствуют некоторые артефакты в виде шума по верхнему краю скалы. Чего не скажешь о разработанной функции.

Сравнение результатов

Для изображения со слабым, средним и сильным шумом применим указанные фильтры с разными параметрами и используем метрику SAD для сравнения с исходным, незашумленным изображением:

low:	Gl: 3180147	Bl: 15549722	NLl: **3025445**
	G: 3331273	B: 15132402	NL: **3321883**
medium:	Gl: **7536446**	Bl: 16242755	NLl: 7552458
	G: **7595254**	B: 15878511	NL: 7756836
high:	Gl: **10925004**	Bl: 16920905	NLl: 11010822
	G: **10939958**	B: 16616063	NL: 11163892

По итогам сравнения – фильтр гаусса лучше всех отвечает метрике SAD при средней и сильной зашумленности, но проигрывает NL фильтру при слабых шумах.

Проверим, используя фильтры OpenCV:

low:	Gl: 3131667	Bl: 3141306	NLl: **2850494**
	G: 3246953	B: 3650893	NL: **2854775**
medium:	Gl: 7520312	Bl: 7524043	NLl: **7444087**
	G: 7563376	B: 7731728	NL: **7445464**
high:	Gl: **10935680**	Bl: 10942715	NLl: 10945841
	G: 10946802	B: 11001630	NL: **10946070**

Ситуация с NL фильтром получше, он выигрывает у всех остальных фильтров, однако отрыв в сильных и средних шумах недалеко уходит от фильтра гаусса. Также метрики выглядят немного лучше, чем у реализованных вручную методов.

Выводы

В ходе данной лабораторной работы были созданы собственные реализации фильтров Гаусса, Билатерального и Нелокальных значений. Библиотека OpenCV предоставляет все эти фильтры, причем в нескольких вариациях.

По итогам сравнения результатов работы реализаций, библиотечные функции гораздо лучше справляются с задачей.

Фильтр Гаусса обычно используется в цифровом виде для обработки двумерных сигналов (изображений) с целью снижения уровня шума. Однако при ресемплинге он дает сильное размытие изображения и не справляется с мощными шумами без весомой потери качества, так как для него не важно наличие контрастных переходов между областями. Применяется быстрее остальных фильтров.

Билатеральная фильтрация: довольно медленная, на практике применяется в 4 раза дольше фильтра Гаусса. Для данного метода существуют техники ускорения фильтрации. К сожалению, эти техники используют больше памяти, чем обычная фильтрация и поэтому не могут быть напрямую применены для фильтрации цветных изображений.

Метод Нелокальных значений производит обход всех пикселей в области, удаленной от целевого пикселя, взвешивая по тому, насколько эти пиксели похожи на целевой пиксель. Это приводит к гораздо большей четкости пост-фильтрации и меньшей потере детализации изображения по сравнению с остальными алгоритмами. В частности, он требует значительных вычислительных ресурсов. При обработке области с текстурой фильтр привносит некоторое размытие изображения, в то время как для плоских областей он работает хорошо. Также хорошо проявляет себя с однородными текстурами и лучше всех показал себя в данной работе.

Данные фильтры используются для сглаживания шумов и повышения качества цифровых изображений м

Приложение

Код программы на языке C++

```
#pragma comment(lib,"I:\\Downloads\\opcv\\opencv\\build\\x64\\vc15\\lib\\opencv_world343.lib")
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

double gaussianFunc(int x, int y, double sigma);

float distance(int x, int y, int i, int j);

double gaussian(float x, double sigma);

void generateKernel(int size, Mat& kernel, float sigma);

uchar countGaussValue(int x, int y, Mat& img, Mat& kernel);

Mat neighboursValues(int area, Mat& src, int x, int y);

double normOfVector(Mat& vec1, Mat& vec2);

void myGaussFilter(Mat& img, int kernelSize, float sigma);

void myBilateralFilter(Mat source, Mat filteredImage, int x, int y, int diameter, double sigmaI,
double sigmaS, int height, int width);

Mat myBilateralFilter(Mat source, int diameter, double sigmaI, double sigmaS);

void nonLocalMeans(Mat& source, Mat& filteredImage, int x, int y, int diameter, double sigmaI, int
height, int width);

Mat myNLMeansFilter(Mat& source, int diameter, double sigmaI);


int main()
{
	string imgName = "2.jpg";
	Mat src;
	src = imread(imgName, IMREAD_GRAYSCALE);

	if (!src.data)
	{
		printf("No image\n");
		return -1;
	}

	//Gaussian Filter

	Mat gaussTest;
	Mat gaussTest2;
	Mat gaussTestCV;
	Mat gaussTestCV2;

	src.copyTo(gaussTest);
	src.copyTo(gaussTest2);

    myGaussFilter(gaussTest, 7, 0.8);
	myGaussFilter(gaussTest2, 7, 1);

	imwrite("GB_0.8.png", gaussTest);
	imwrite("GB_1.png", gaussTest2);
```

```
    GaussianBlur(src, gaussTestCV, Size(7, 7), 0.8);
    GaussianBlur(src, gaussTestCV2, Size(7, 7), 1);

       imwrite("GB_cv_08.png", gaussTestCV);
    imwrite("GB_cv_1.png", gaussTestCV2);

       //Bilateral Filter

       Mat bilateralTest;
       Mat bilateralTest2;
       Mat bilateralTestCV;
       Mat bilateralTestCV2;

       src.copyTo(bilateralTest);
       src.copyTo(bilateralTest2);

    Mat bilateralFilteredImage = myBilateralFilter(bilateralTest, 5, 50.0, 50.0);
       Mat bilateralFilteredImage_2 = myBilateralFilter(bilateralTest2, 9, 150.0, 150.0);

    imwrite("B_5_50_50.png", bilateralFilteredImage);
    imwrite("B_9_150_150.png", bilateralFilteredImage_2);

       bilateralFilter(src, bilateralTestCV2, 5, 50.0, 50.0);
       bilateralFilter(src, bilateralTestCV2, 9, 150.0, 150.0);

       imwrite("B_cv_5_50_50.png", bilateralTestCV2);
       imwrite("B_cv_9_150_150.png", bilateralTestCV2);

       // NlMeans Filter

       Mat NLMeansTest;
       Mat NLMeansTest2;
       Mat NLMeansTestCV;
       Mat NLMeansTestCV2;

       src.copyTo(NLMeansTest);
       src.copyTo(NLMeansTest2);

       Mat NLMeansFilteredImage = myNLMeansFilter(src, 3, 15);
       imwrite("NL_3_15.png", NLMeansFilteredImage);

       Mat NLMeansFilteredImage_2 = myNLMeansFilter(src, 5, 22);
       imwrite("NL_5_22.png", NLMeansFilteredImage_2);

       fastNlMeansDenoising(src, NLMeansTestCV, 3, 15, 15);
       imwrite("NL_cv_3_15.png", NLMeansTestCV);

       fastNlMeansDenoising(src, NLMeansTestCV2, 5, 22, 22);
       imwrite("NL_cv_5_22.png", NLMeansTestCV2);
}


double gaussianFunc(int x, int y, double sigma) {
       return((1 / (2 * CV_PI*sigma*sigma))*exp(-(x*x + y * y) / (2 * sigma*sigma)));
}

float distance(int x, int y, int i, int j) {
       return float(sqrt(pow(x - i, 2) + pow(y - j, 2)));
}

double gaussian(float x, double sigma) {
       return exp(-(pow(x, 2)) / (2 * pow(sigma, 2))) / (2 * CV_PI * pow(sigma, 2));
}

void generateKernel(int size, Mat& kernel, float sigma) {
       kernel = Mat(size, size, CV_32F);
       for (int i = 0; i < size; i++) {
              for (int j = 0; j < size; j++) {
```

```
                    kernel.at<float>(i, j) = gaussianFunc(i - (size - 1) / 2, j - (size - 1) / 2,
sigma);
             }
       }
}

uchar countGaussValue(int x, int y, Mat& img, Mat& kernel) {
       float acc = 0;
       for (int i = 0; i < kernel.rows; i++) {
             for (int j = 0; j < kernel.cols; j++) {
                    if (((y - (kernel.cols - 1) / 2) + i) >= 0 && ((y - (kernel.cols - 1) / 2) + i)
< img.rows && ((x - (kernel.cols - 1) / 2) + j) >= 0 && ((x - (kernel.cols - 1) / 2) + j) <
img.cols) {

                           acc += img.at<uchar>((y - (kernel.cols - 1) / 2) + i, (x - (kernel.cols -
1) / 2) + j)*kernel.at<float>(i, j);
                    }
                    else continue;
             }
       }
       return (uchar)acc;
}

Mat neighboursValues(int area, Mat& src, int x, int y) {
       Mat values = Mat::zeros(area*area, 1, CV_8U);
       for (int i = 0; i < area; i++) {
             for (int j = 0; j < area; j++) {
                    values.at<uchar>(j + i, 0) = src.at<uchar>((x - (area - 1) / 2) + j, (y - (area
- 1) / 2) + i);
             }
       }
       return values;
}

double normOfVector(Mat& vec1, Mat& vec2) {
       double norm = 0;
       for (int i = 0; i < vec1.rows; i++) {
             norm = norm + pow((vec1.at<uchar>(i, 0) - vec2.at<uchar>(i, 0)), 2);
       }
       norm = sqrt(norm);
       return norm;
}

void nonLocalMeans(Mat& source, Mat& filteredImage, int x, int y, int diameter, double sigmaI, int
height, int width) {
       double iFiltered = 0;
       double wP = 0;
       int xNeighbor = 0;
       int yNeighbor = 0;
       int half = diameter / 2;

       for (int i = 0; i < diameter; i++) {
             for (int j = 0; j < diameter; j++) {
                    xNeighbor = x - (i - half);
                    yNeighbor = y - (j - half);
                    if (xNeighbor < 0) xNeighbor = 0;
                    if (yNeighbor < 0) yNeighbor = 0;
                    while (xNeighbor >= height - half) xNeighbor--;
                    while (yNeighbor >= width - half) yNeighbor--;
                    if (x < half) x = half;
                    if (y < half) y = half;
                    if (x >= height - half)
                           x = height - half;
                    if (y >= width - half)
                           y = width - half;
                    Mat vector1 = neighboursValues(half, source, x, y);
                    Mat vector2 = neighboursValues(half, source, xNeighbor, yNeighbor);
                    double vecNorm = normOfVector(vector1, vector2);
```

```
                        double gr = gaussian(vecNorm, sigmaI);
                        double w = gr;
                        iFiltered = iFiltered + source.at<uchar>(xNeighbor, yNeighbor) * w;
                        wP = wP + w;
                }
        }
        iFiltered = iFiltered / wP;
        filteredImage.at<uchar>(x, y) = (uchar)iFiltered;
}

Mat myNLMeansFilter(Mat& source, int diameter, double sigmaI) {
        Mat resultImage = Mat::zeros(source.rows, source.cols, CV_8U);
        int width = source.cols;
        int height = source.rows;

        for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                        nonLocalMeans(source, resultImage, i, j, diameter, sigmaI, height, width);
                }
        }
        return resultImage;
}

void myGaussFilter(Mat& img, int kernelSize, float sigma) {
        Mat gauss;
        generateKernel(kernelSize, gauss, sigma);
        Size imgSize = img.size();
        for (int i = 0; i < imgSize.height; i++) {
                for (int j = 0; j < imgSize.width; j++) {
                        img.at<uchar>(i, j) = countGaussValue(j, i, img, gauss);
                }
        }
}

void myBilateralFilter(Mat source, Mat filteredImage, int x, int y, int diameter, double sigmaI,
double sigmaS, int height, int width) {
        double iFiltered = 0;
        double wP = 0;
        int neighbor_x = 0;
        int neighbor_y = 0;
        int half = diameter / 2;

        for (int i = 0; i < diameter; i++) {
                for (int j = 0; j < diameter; j++) {
                        neighbor_x = x - (i - half);
                        neighbor_y = y - (j - half);
                        if (neighbor_x < 0) neighbor_x = 0;
                        if (neighbor_y < 0) neighbor_y = 0;
                        while (neighbor_x >= height) neighbor_x--;
                        while (neighbor_y >= width) neighbor_y--;
                        double gi = gaussian(source.at<uchar>(neighbor_x, neighbor_y) -
source.at<uchar>(x, y), sigmaI);
                        double gs = gaussian(distance(x, y, neighbor_x, neighbor_y), sigmaS);
                        double w = gi * gs;
                        iFiltered = iFiltered + source.at<uchar>(neighbor_x, neighbor_y) * w;
                        wP = wP + w;
                }
        }
        iFiltered = iFiltered / wP;
        filteredImage.at<double>(x, y) = iFiltered;


}

Mat myBilateralFilter(Mat source, int diameter, double sigmaI, double sigmaS) {
        Mat filteredImage = Mat::zeros(source.rows, source.cols, CV_64F);
        int width = source.cols;
        int height = source.rows;
```

```
	for (int i = 2; i < height - 2; i++) {
		for (int j = 2; j < width - 2; j++) {
			myBilateralFilter(source, filteredImage, i, j, diameter, sigmaI, sigmaS,
height, width);
		}
	}
	return filteredImage;
}
```