Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №2

на тему: «Методы сглаживания изображений»

Курс: «Разработка графических приложений»

Выполнил студент:

Волкова М.Д.
Группа: 13541/2

Проверил:

Абрамов Н.А.

Санкт-Петербург
2018 г.

Содержание

Лабораторная работа №2

1.1 Цель работы

Ознакомится с методами сглаживания, на таких подходах, как: Gaussian Blur; Bilateral Filter; Non – local means.

1.2 Программа работы

1. Реализовать следующие методы сглаживания изображений на языке c++:
   - Gaussian Blur
   - Bilateral Filter
   - Non – local means.
2. Сравнить результаты со стандартными функциями библиотеки opencv.
3. Сравнить работу.
4. Результаты привести в отчет.

1.3 Ход работы

1.3.1 Фильтр Гаусса

Фильтр Гаусса —фильтр, чьей импульсной переходной функцией является функция Гаусса. Фильтр Гаусса (Gaussian filter) обычно используется в цифровом виде для обработки двумерных сигналов (изображений) с целью снижения уровня шума. Однако при ресемплинге он дает сильное размытие изображения. Гауссова функция (гауссиан, гауссиана, функция Гаусса) — вещественная функция, описываемая следующей формулой:

$$g(x) = a * exp^{-\frac{(x-b)^2}{2c^2}}$$

Это наиболее часто используемый метод размытия. Мы можем использовать этот фильтр для устранения шумов на изображении. Но нам нужно быть очень осторожными в выборе размера ядра и стандартного отклонения распределения Гаусса по X и Y направлению. Они должны быть тщательно подобраны.

GaussianBlur() синтакс:

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

Параметры:

- src - входное изображение; изображение может иметь любое количество каналов, которые обрабатываются независимо друг от друга.
- dst — выходное изображение тогоже размера и типа, что и src.
- 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 тогоб чтобы полностью контролировать результат, независимо от возможных будущих модификаций, рекомендуется указать все ksize, sigmaX и sigmaY.
- borderType — пиксельный метод экстраполяции.

Для начала, возьмем изображение и добавим на него легкий шум:

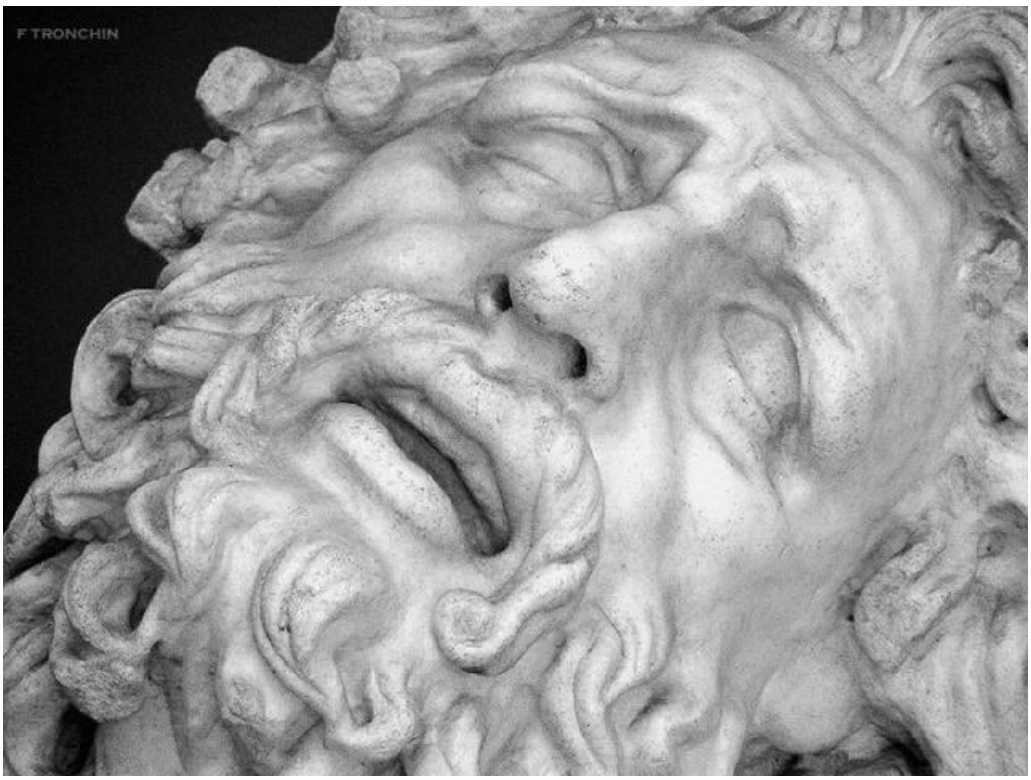
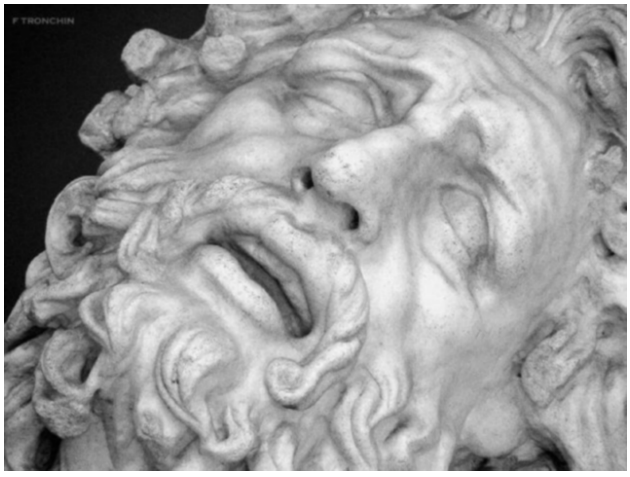
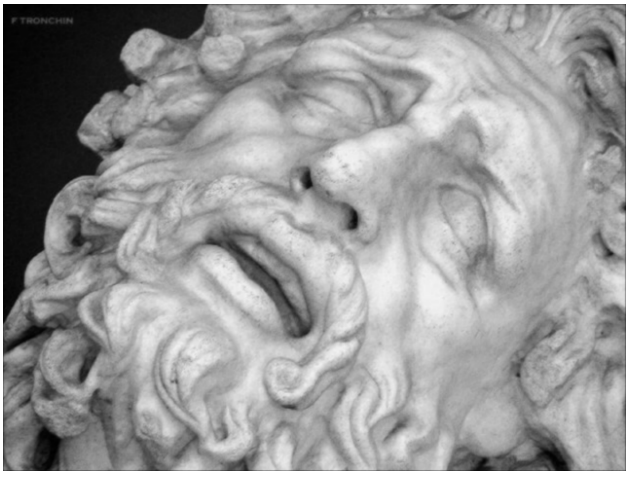


Рис. 1.1: Изображение с шумом

Полный код программы представленн в листинге. Применим фильтр для изображения и сравним с стандартной функцией opencv:



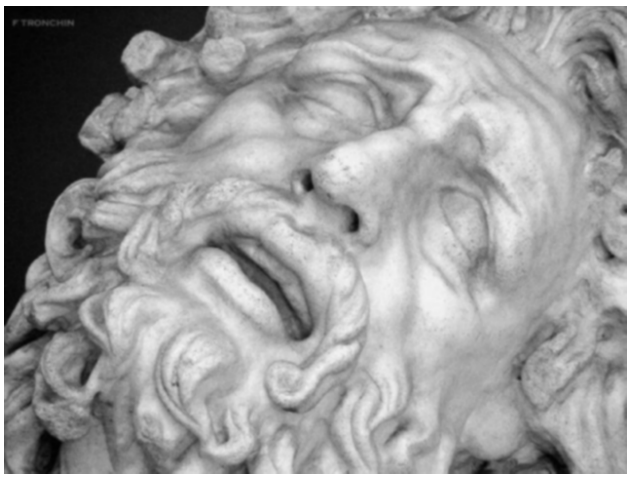
а) GaussianBlur



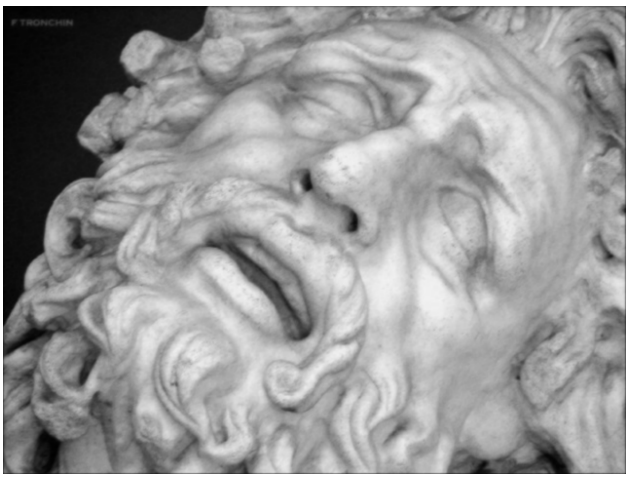
б) myGaussFilter

Рис. 1.2: Изображение после применения фильтра Гаусса (sigma = 0.8)

Для лучшего понимания работы фильтра и для более резкого результата, увеличим значение сигмы:



а) GaussianBlur



б) myGaussFilter

Рис. 1.3: Изображение после применения фильтра Гаусса (sigma = 1)

На изображениях видно, что фильтр прекрасно справился с удалением шума, без потери четкости и элементов изображений, а результат кастомного фильтра практически не различается с фильтром из библиотеки opencv.

1.3.2 Билатеральный фильтр

Введённый Tomasi и Manduchi билатеральный фильтр, сохраняющий края, нашёл широкое применение во многих задачах по обработке изображений, например, фильтрация шума , редактирование текстуры и тона, оценки оптического потока. Билатеральная фильтрация также часто используется в качестве начального этапа обработки кадров, например, для задачи распознавания объектов, где необходимо отфильтровать несущественные детали и шумы при сохранении резких краев основного изображения. Основным недостатком билатеральных фильтров являются большие вычислительные затраты. Билатеральная фильтрация (двунаправленная фильтрация) - это нелинейный и не итерационный процесс, комбинирующий пространственную (domain) и яркостную (range) фильтрацию. Таким образом, учитывается не только значения интенсивности близлежащих пикселей, но и их расстояние до текущего фильтруемого пикселя. Вклад близлежащих пикселей существенен по отношению к остальным.

bilateralFilter() синтаксис:

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

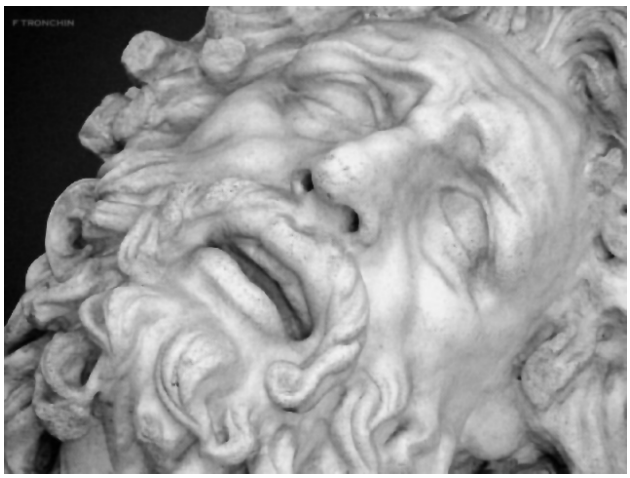
- src – входное изображение.
- dst – выходное изображение того же формата, что и src .
- d – диаметр каждой пиксельной окрестности, которая используется во время фильтрации. Если оно не положительное, оно вычисляется из sigmaSpace .
- sigmaColor – Фильтр сигма в цветовом пространстве. Большее значение параметра означает, что более дальние цвета в SigmaSpace будут смешаны вместе, что приведет к большим областям с полураствореным цветом.
- sigmaSpace – Фильтр сигма в координатном пространстве. Большее значение параметра означает, что дальнейшие пиксели будут влиять друг на друга, пока SigmaColor достаточно близки. Когда d>0, он определяет размер окрестности независимо от sigmaSpace. В противном случае, d пропорционально sigmaSpace.

Как подсказывает документация, легким способом подбора значений сигм - установить их одинаковыми. Большое значение сигмы (> 150) обещает удалить все шумы, но это означет потерю четкости и эффект "мультяшности".

Большие фильтры (d> 5) очень медленные, поэтому рекомендуется использовать $d = 5$ и, возможно, $d = 9$ для автономных приложений.

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

Посмотрим результаты:



а) bilateralFilter

б) myBilateralFilter

Рис. 1.4: Изображение после применения билатерального фильтра (d = 5, sigma = 50)

Увеличим значения параметров, для большей презентабельности:

а) bilateralFilter

б) myBilateralFilter

Рис. 1.5: Изображение после применения билатерального фильтра (d = 9, sigma = 150)

Билетеральный фильтр также не плохо справился с подавлением шума.

1.3.3 Фильтр NLMeans

Это алгоритм обработки изображений для шумоподавления. В отличие от фильтров"local mean которые принимают среднее значение группы пикселей, окружающих целевой пиксель, для сглаживания изображения, фильтрация нелокальными средствами принимает среднее значение всех пикселей в изображении, взвешенное по тому, насколько похожи эти пиксели на целевой пиксель. Это приводит к гораздо большей ясности постфильтрации и меньшей потере деталей изображения по сравнению с локальными средними алгоритмами. По сравнению с другими известными методами шумоподавления нелокальные средства добавляют "методический шум"(т. е. ошибку в процессе шумоподавления), который больше похож на белый шум, что желательно, поскольку он обычно менее тревожен в шумоизолированном продукте. Цель весовой функции-определить, насколько тесно изображение в точке p связано с изображением в точке q. Она может принимать различные формы. Весовая функция Гаусса

$$f(p,q) = exp^{-\frac{|B(q)-B(p)|^2}{h^2}}$$

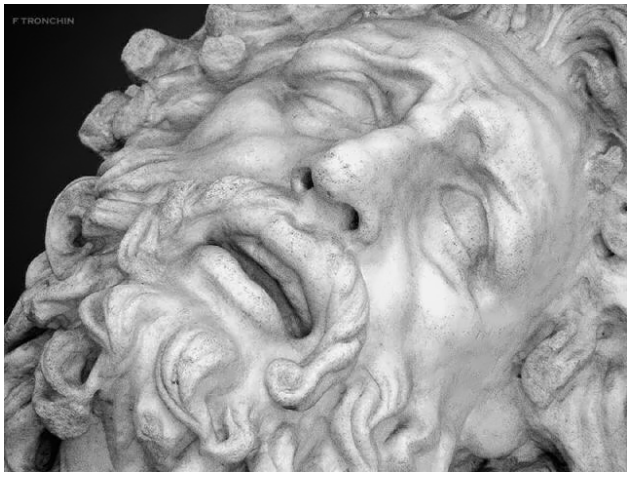
fastNlMeansDenoising() синтаксис:

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

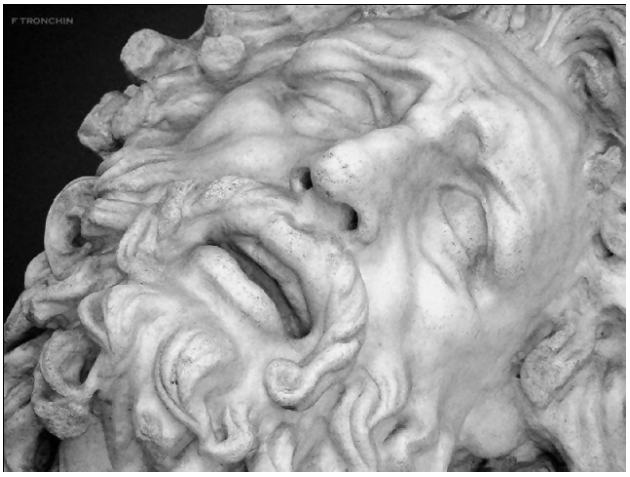
Параметры:

- src – входное изображение.
- dst – выходное изображение того же формата, что и src .
- templateWindowSize - размер в пикселях патча 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

Также есть функция fastNlMeansDenoisingColored(), которая удаляет шум на цветных изображениях. Посмотрим результаты:



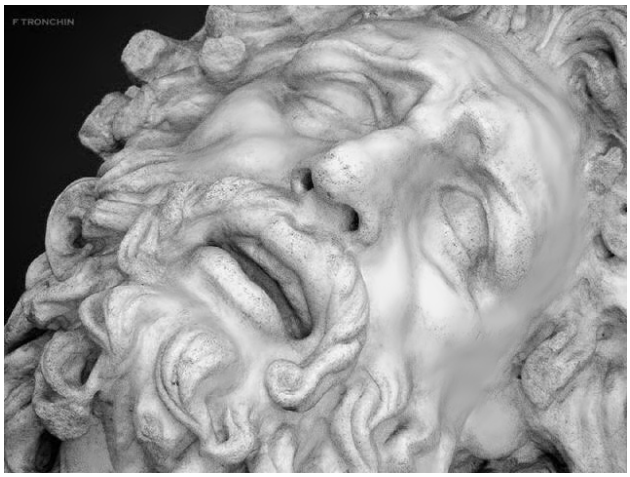
а) fastNlMeansDenoising



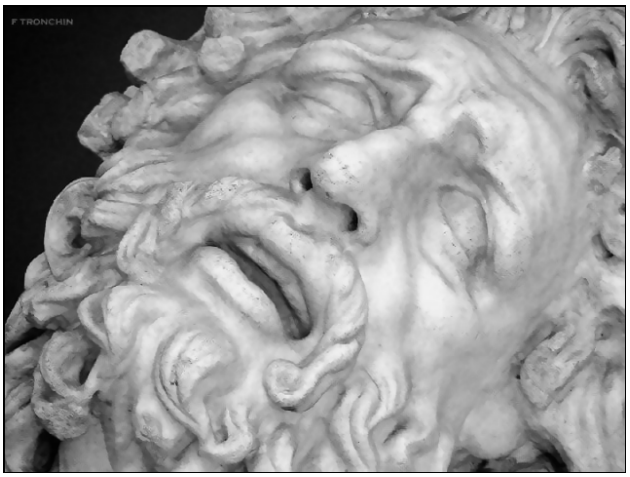
б) myNLMeansFilter

Рис. 1.6: Изображение после применения фильтра NLMeans (d = 3, sigma = 15)

Увеличим значения параметров:



а) fastNlMeansDenoising



б) myNLMeansFilter

Рис. 1.7: Изображение после применения фильтра NLMeans (d = 5, sigma = 21)

Видно, что фильтры хорошо удаляют шум с изображения и даже при больших значениях параметров, детали не теряются.

1.4 Вывод

В работе были рассмотрены следующие алгоритмы сглаживания изображений:

- Gaussian Blur
- Bilateral Filter
- Non – local means.

Фильтр Гаусса: обычно используется в цифровом виде для обработки изображений с целью снижения уровня шума. Однако при ресемплинге он дает сильное размытие изображения. Фильтр Гаусса являеться низкочастотным фильтром. Идеально подходит для бинарных изображений.

Билатеральная фильтрация: довольно медленная, 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.

Non – local means: способ имеет ряд недостатков. В частности, способ требует значительных вычислительных ресурсов. При обработке области с текстурой фильтр привносит некоторое размытие изображения, в то время как для плоских областей он работает хорошо. Поэтому для областей с текстурой необходима некоторая адаптация фильтра. Также требуется некоторая модификация способа для ускорения алгоритма.

Также для подавления шума используют: медианный фильтр, усреднение (box filter) и прочее.

1.5 Листинг

```
#include "pch.h"
#include <iostream>
#include <opencv2/opencv.hpp>

#define SIGMA 0.8

using namespace cv;
using namespace std;

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) {
  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
    sigmal, 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, sigmal);
      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 sigmal) {
  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, sigmal, height, width);
    }
  }
  return resultImage;
}

void myGaussFilter(Mat& img, int kernelSize) {
  Mat gauss;
  generateKernel(kernelSize, gauss);
  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);
    }
  }
}

```

```
124 void myBilateralFilter(Mat source, Mat filteredImage, int x, int y, int diameter, double
        sigmaI, double sigmaS, int height, int width) {
125   double iFiltered = 0;
126   double wP = 0;
127   int neighbor_x = 0;
128   int neighbor_y = 0;
129   int half = diameter / 2;
130
131   for (int i = 0; i < diameter; i++) {
132     for (int j = 0; j < diameter; j++) {
133       neighbor_x = x - (i- half);
134       neighbor_y = y - (j- half);
135       if (neighbor_x < 0) neighbor_x = 0;
136       if (neighbor_y < 0) neighbor_y = 0;
137       while (neighbor_x >= height) neighbor_x--;
138       while (neighbor_y >= width) neighbor_y--;
139       double gi = gaussian(source.at<uchar>(neighbor_x, neighbor_y) - source.at<uchar>(x,
        y), sigmaI);
140       double gs = gaussian(distance(x, y, neighbor_x, neighbor_y), sigmaS);
141       double w = gi * gs;
142       iFiltered = iFiltered + source.at<uchar>(neighbor_x, neighbor_y) * w;
143       wP = wP + w;
144     }
145   }
146   iFiltered = iFiltered / wP;
147   filteredImage.at<double>(x, y) = iFiltered;
148
149
150 }
151
152 Mat myBilateralFilter(Mat source, int diameter, double sigmaI, double sigmaS) {
153   Mat filteredImage = Mat::zeros(source.rows, source.cols, CV_64F);
154   int width = source.cols;
155   int height = source.rows;
156
157   for (int i = 2; i < height - 2; i++) {
158     for (int j = 2; j < width - 2; j++) {
159       myBilateralFilter(source, filteredImage, i, j, diameter, sigmaI, sigmaS, height,
        width);
160     }
161   }
162   return filteredImage;
163 }
164
165 int main()
166 {
167   string imgName = "masha.jpg";
168   Mat src;
169   src = imread(imgName, IMREAD_GRAYSCALE);
170   cout << "TYPE: " << typeToString(src.type());
171
172   if (!src.data)
173   {
174     printf("No image data \n");
175     return -1;
176   }
177
178   Mat gaussTest;
179   Mat gaussTestOpenCV;
180   Mat bilateralTest;
181   Mat bilateralTestOpenCV;
182   Mat NLMeansTest;
183   Mat NLMeansTestOpenCV;
184
185   src.copyTo(gaussTest);
186   src.copyTo(bilateralTest);
```

```
187     src.copyTo(NLMeansTest);
188
189       myGaussFilter(gaussTest, 7);
190       imwrite("GB_sigma_3.png", gaussTest);
191
192       GaussianBlur(src, gaussTestOpenCV, Size(7, 7), 0.8);
193       imwrite("GB_ocv_sigma_08.png", gaussTestOpenCV);
194
195
196       Mat bilateralFilteredImage = myBilateralFilter(bilateralTest, 5, 50.0, 50.0);
197       imwrite("b_5_50_50.png", bilateralFilteredImage);
198
199     bilateralFilter(src, bilateralTestOpenCV_2, 9, 150.0, 150.0);
200     imwrite("b_op_9_150_150.png", bilateralTestOpenCV_2);
201
202
203     Mat NLMeansFilteredImage = myNLMeansFilter(src, 3, 15);
204     imwrite("NLM_3_15.png", NLMeansFilteredImage);
205
206     fastNlMeansDenoising(src, NLMeansTestOpenCV_2, 5, 21, 21);
207     imwrite("NLM_op_5_21_21.png", NLMeansTestOpenCV_2);
208
209     waitKey(0);
210 }
```