



НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: ТЕХНИЧЕСКОЕ ЗРЕНИЕ

Отчет по лабораторной работе №3

“Фильтрация и выделение контуров”

Выполнили:

Новиков Дмитрий, ТЕХ.ЗРЕНИЕ 1.1

Преподаватель:

Шаветов С. В.

Санкт-Петербург, 2024

Содержание

1. Теоретическая часть	2
2. Типы шумов	3
2.1. Импульсный шум	3
2.2. Мультипликативный шум	5
2.3. Гауссов (нормальный) шум	7
2.4. Шум квантования	9
3. Низкочастотная фильтрация	11
3.1. Контргармонический усредняющий фильтр	11
3.2. Фильтр Гаусса	13
4. Нелинейная фильтрация	15
4.1. Медианная фильтрация	15
4.2. Адаптивная медианная фильтрация	17
4.3. Винеровская фильтрация	19
5. Высокочастотная фильтрация	22
5.1. Фильтр Робертса	22
5.2. Фильтр Превитта	24
5.3. Фильтр Превитта	25
5.4. Фильтр Лапласа	27
5.5. Алгоритм Кэнни	28
6. Выводы	29
7. Ответы на вопросы	29

[Исходный код на GitHub.](#)

1. Теоретическая часть

Цифровые изображения, полученные различными оптико-электронными приборами, могут содержать в себе разнообразные искажения, обусловленные разного рода помехами, которые принято называть шумом. Шум на изображении затрудняет его обработку автоматическими средствами, и, поскольку шум может иметь различную природу, для его успешного подавления необходимо определить адекватную математическую модель.

Для удобства отладки и сборки используется средство автоматизации сборки ПО CMake:

```
cmake_minimum_required(VERSION 2.8)

project( CV_LW3 )
find_package( OpenCV REQUIRED )

include_directories( ${OpenCV_INCLUDE_DIRS} )
add_executable( ${PROJECT_NAME} src/lab2.cpp )

target_link_libraries( ${PROJECT_NAME} ${OpenCV_LIBS} )
```

Листинг 1: CMakeLists.txt для сборки проекта

Возьмем одно изображение из датасета беспилотников Яндекса и наложим на него различные шумы

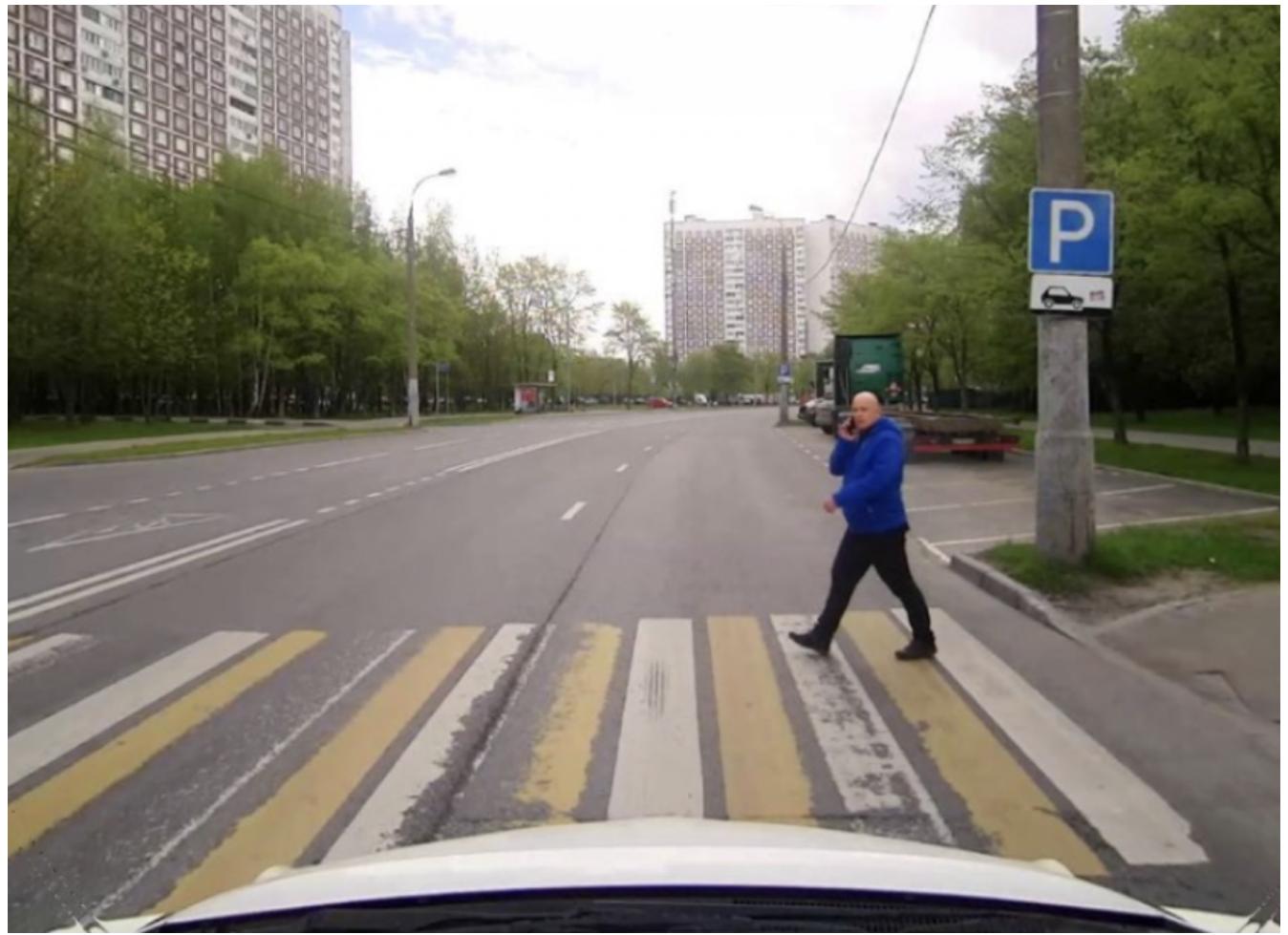


Рис. 1: Исходное изображение

2. Типы шумов

2.1. Импульсный шум

При импульсном шуме сигнал искажается выбросами с очень большими отрицательными или положительными значениями малой длительностью и может возникать, например, из-за ошибок декодирования. Такой шум приводит к появлению на изображении белых («соль») или черных («перец») точек, поэтому зачастую называется точечным шумом. Для его описания следует принять во внимание тот факт, что появление шумового выброса в каждом пикселе $I(x, y)$ не зависит ни от качества исходного изображения, ни от наличия шума в

других точках и имеет вероятность появления p , причем значение интенсивности пикселя $I(x, y)$ будет изменено на значение $d \in [0, 255]$

$$I(x, y) = \begin{cases} d, & p_d \in [0, 1] \\ s_{x,y}, & p_s = (1 - p_d), \end{cases} \quad (1)$$

где $s_{x,y}$ — интенсивность пикселя исходного изображения, I — зашумленное изображение, если $d = 0$ — шум типа «перец», если $d = 255$ — шум типа «соль».

```
double d = 0.0005;
double s_vs_p = 0.1;
std::vector<cv::Mat> image_out_BGR;

cv::split(image, image_out_BGR);

for(int i = 0; i < image_out_BGR.size(); ++i){
    cv::Mat vals(image_out_BGR[i].size(), CV_32F);
    cv::randn(vals, cv::Scalar(0), cv::Scalar(1));

    if(image_out_BGR[i].depth() == CV_8U)
        image_out_BGR[i].setTo(cv::Scalar(255), vals < d * s_vs_p);
    else
        image_out_BGR[i].setTo(cv::Scalar(1), vals < d * s_vs_p);

    image_out_BGR[i].setTo(cv::Scalar(0), (vals >= d * s_vs_p) & (vals < d));
}

cv::merge(image_out_BGR, image_out);
```

Листинг 2: Исходный код для применения импульсного шума к изображению



Рис. 2: Изображение с импульсным шумом

2.2. Мультипликативный шум

Мультипликативный шум описывается следующим выражением:

$$I_{new} = I(x, y) * \eta(x, y), \quad (2)$$

где I_{new} — зашумленное изображение, I — исходное изображение, η — не зависящий от сигнала мультипликативный шум, умножающий зарегистрированный сигнал. В качестве примера можно привести зернистость фотопленки, ультразвуковые изображения и т.д. Частным случаем мультипликативного шума является спекл-шум, который появляется на изображениях, полученных устройствами с когерентным формированием изображений, например, медицинскими сканерами или радарами. На таких изображениях можно отчетливо наблюдать светлые пятна, крапинки (спеклы), которые разделены темными участками изображения.

```
double var = 0.05;
std::vector<cv::Mat> image_out_BGR;

cv::split(image, image_out_BGR);

for(int i = 0; i < image_out_BGR.size(); ++i){

    cv::Mat gauss(image_out_BGR[i].size(), CV_32F);
    cv::randn(gauss, cv::Scalar(0), cv::Scalar(cv::sqrt(var)));

    if( image_out_BGR[i].depth() == CV_8U){
        cv::Mat image_out_BGR_f;

        image_out_BGR[i].convertTo(image_out_BGR_f, CV_32F);

        image_out_BGR_f += image_out_BGR_f.mul(gauss);
        image_out_BGR_f.convertTo(image_out_BGR[i],
        image_out_BGR[i].type());
    } else
        image_out_BGR[i] += image_out_BGR[i].mul(gauss);
}

cv::merge(image_out_BGR, image_out);
```

Листинг 3: Исходный код для применения мультипликативного шума к изображению



Рис. 3: Изображение с мультипликативным шумом

2.3. Гауссов (нормальный) шум

Гауссов шум на изображении может возникать в следствие недостатка освещенности сцены, высокой температуры и т.д. Модель шума широко распространена в задачах низкочастотной фильтрации изображений. Функция распределения плотности вероятности $p(z)$ случайной величины z описывается следующим выражением:

$$p(z) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{\frac{(-z-\mu)^2}{2\sigma^2}} \quad (3)$$

где z — интенсивность изображения (например, для полутонового изображения $z \in [0, 255]$), μ — среднее (математическое ожидание) случайной величины z , σ — среднеквадратичное отклонение, дисперсия σ^2 определяет мощность вносимого шума.

```
double mean = 0;
double var = 0.05;
std::vector<cv::Mat> image_out_BGR;

cv::split(image, image_out_BGR);

for(int i = 0; i < image_out_BGR.size(); ++i){

    cv::Mat gauss(image_out_BGR[i].size(), CV_32F);
    cv::randn(gauss, cv::Scalar(mean), cv::Scalar(cv::sqrt(var)));

    if( image_out_BGR[i].depth() == CV_8U){
        cv::Mat image_out_BGR_f;

        image_out_BGR[i].convertTo(image_out_BGR_f, CV_32F);

        image_out_BGR_f += gauss * 255;
        image_out_BGR_f.convertTo(image_out_BGR[i],
        image_out_BGR[i].type());
    } else
        image_out_BGR[i] += gauss;
}

cv::merge(image_out_BGR, image_out);
```

Листинг 4: Исходный код для применения Гауссовского (нормального) шума к изображению



Рис. 4: Изображение с Гауссовским (нормальным) шумом

2.4. Шум квантования

Зависит от выбранного шага квантования и самого сигнала. Шум квантования может приводить, например, к появлению ложных контуров вокруг объектов или убирать слабо контрастные детали на изображении. Такой шум не устраняется.

```
if(image.depth() == CV_8U)
    image.convertTo(image_out, CV_32F, 1.0 / 255);
else
    image_out = image.clone();

size_t vals = unique(image_out).size();
vals = static_cast<size_t>(cv::pow(2, ceil(log2(vals))));

int rows = image_out.rows;
int cols = image_out.cols * image_out.channels();

if(image_out.isContinuous()){
    cols *= rows;
    rows = 1;
}
```

```
using param_t = std::poisson_distribution<int>::param_type;
std::default_random_engine engine;
std::poisson_distribution<> poisson;

for(int i = 0; i < rows; ++i){
    float* ptr = image_out.ptr<float>(i);
    for(int j = 0; j < cols; ++j)
        ptr[j] = float(poisson(engine, param_t({ptr[j] * vals}))) / vals;
}

if(image.depth() == CV_8U)
    image_out.convertTo(image_out, CV_8U, 255);
```

Листинг 5: Исходный код для применения шума квантования к изображению

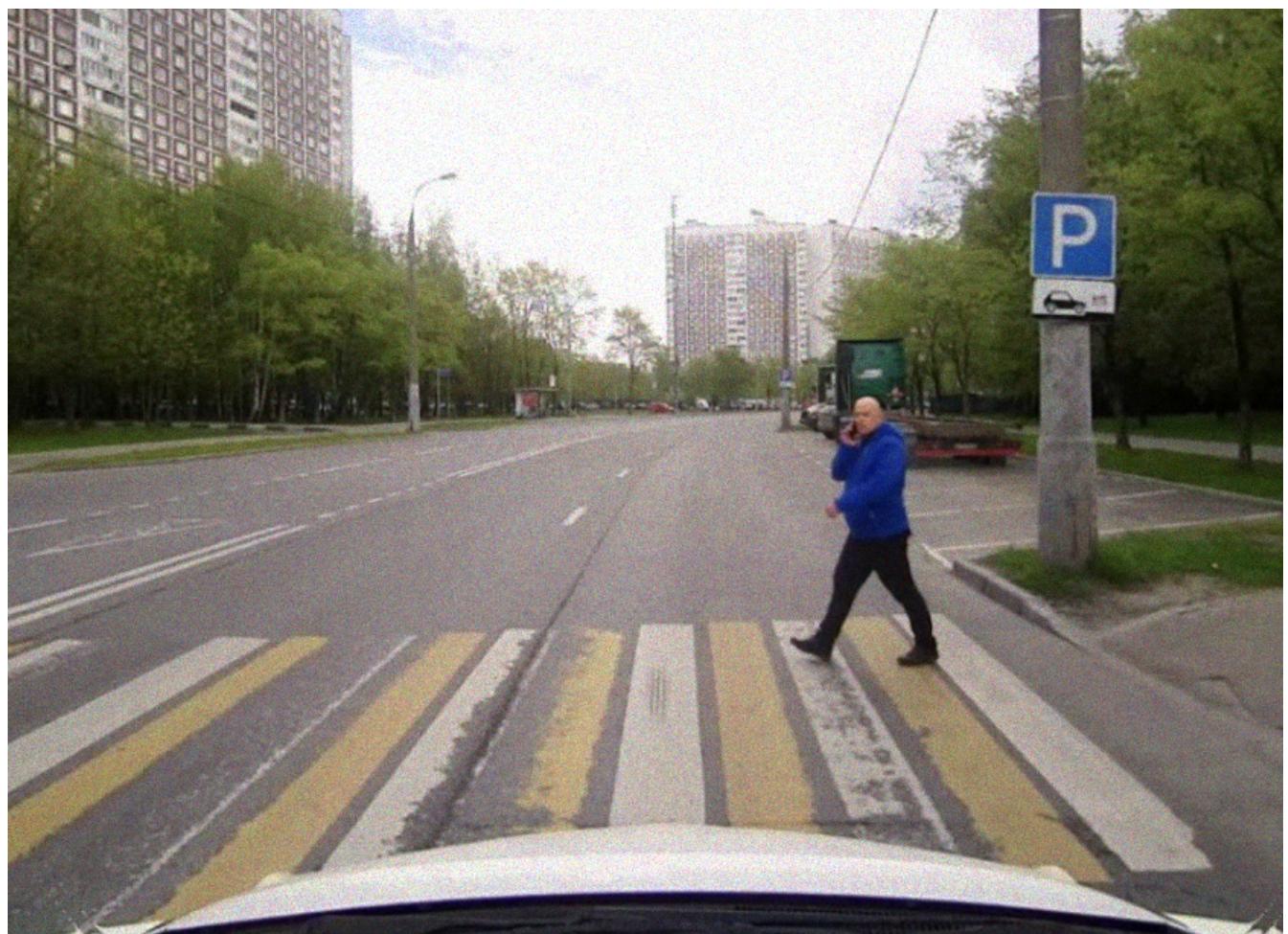


Рис. 5: Изображение с шумом квантования

3. Низкочастотная фильтрация

3.1. Контргармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x, y) = \frac{\sum_{i=0}^m \sum_{j=0}^n I(i, j)^{Q+1}}{\sum_{i=0}^m \sum_{j=0}^n I(i, j)^Q} \quad (4)$$

где Q — порядок фильтра. Контргармонический фильтр является обобщением усредняющих фильтров и при $Q > 0$ подавляет шумы типа «перец», а при $Q < 0$ — шумы типа «соль», однако одновременное удаление белых и черных точек невозможно. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ — в гармонический.

Какой-то brutforce получился... Можно оптимизировать с помощью интегральных выражений или же посмотреть более простые [оптимизации](#)

```
image = cv::imread(path + "/lab3/outputs/image_quant_filter.png", 1);

std::pair<int, int> kernel = std::make_pair(3, 3);
const int Q = -10;

image_out = image.clone();
std::cout << image.depth();

int radius_c = (kernel.first - 1) / 2; // columns offset
int radius_r = (kernel.second - 1) / 2; // rows offset
float res = 0;

for (int c = 0; c < 3; ++c)
{
    for (int i = radius_r; i < image.rows - kernel.second - 1; i++)
    {
        for (int j = radius_c; j < image.cols - kernel.first - 1; j++)
        {
            res = 0;
            for (int m = 0; m < kernel.first; ++m)
            {
                for (int n = 0; n < kernel.second; ++n)
                {
                    res += (pow(image.at<cv::Vec3f>(i + m, j + n)[c], Q +
1)) / (pow(image.at<cv::Vec3f>(i + m, j + n)[c], Q));
                }
            }
            image_out.at<cv::Vec3f>(i, j)[c] = res;
        }
    }
}
```

```
    }  
}  
}
```

Листинг 6: Исходный код для контргармонического усредняющего фильтра

Применим к изображению с импульсивным шумом фильтрацию:

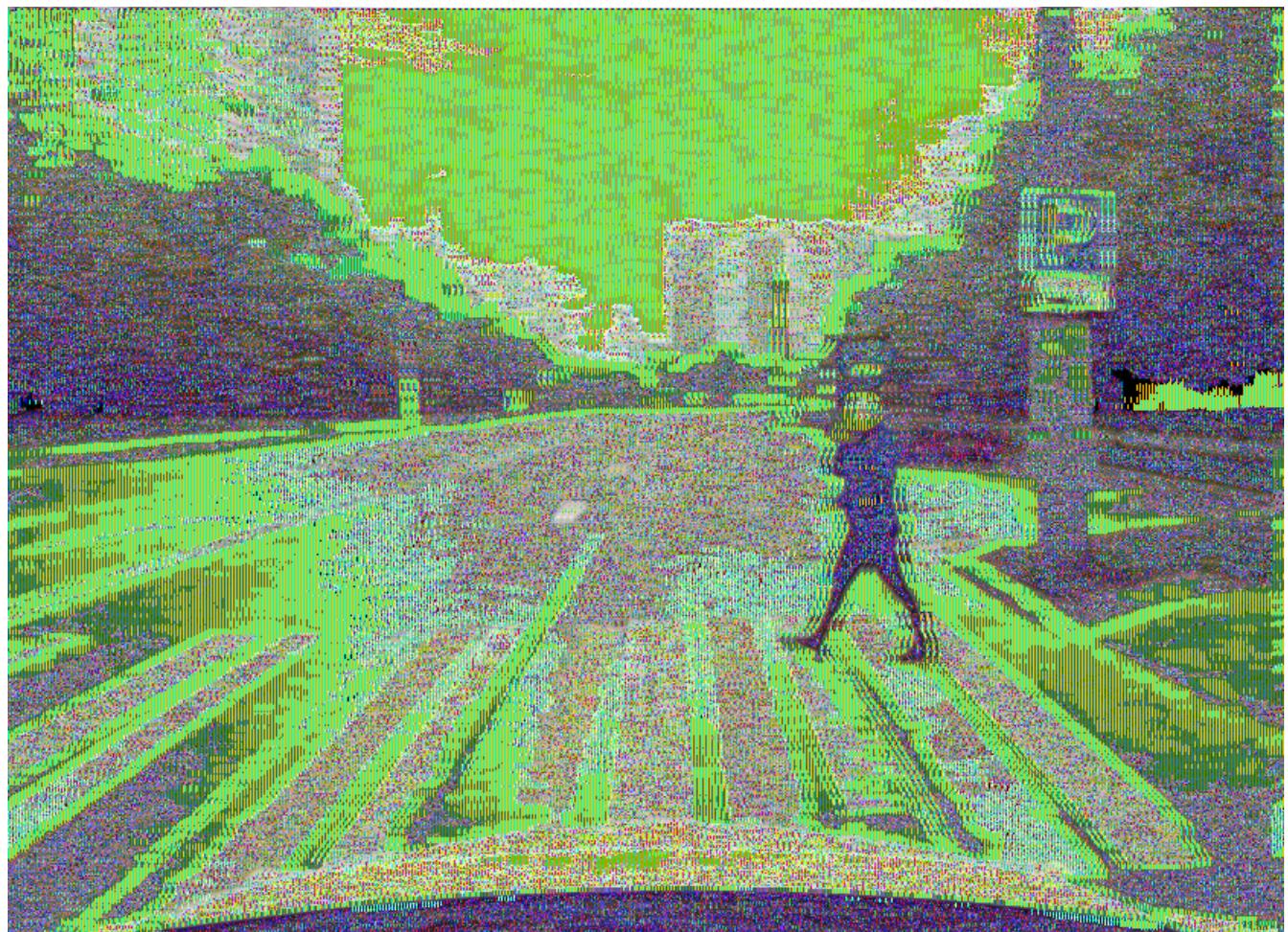


Рис. 6: Изображение после применения контргармонического усредняющего фильтра с $Q = -10$



Рис. 7: Изображение после применения контргармонического усредняющего фильтра с $Q = 5$

3.2. Фильтр Гаусса

Пиксели в скользящем окне, расположенные ближе к анализируемому пикселю, должны оказывать большее влияние на результат фильтрации, чем крайние. Поэтому коэффициенты весов маски можно описать колоколообразной функцией Гаусса (3). При фильтрации изображений используется двумерный фильтр Гаусса:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (5)$$

где Q — порядок фильтра. Контргармонический фильтр является обобщением усредняющих фильтров и при $Q > 0$ подавляет шумы типа «перец», а при $Q < 0$ — шумы типа «соль», однако одновременное удаление белых и черных точек невозможно. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ — в гармонический.

```
image = cv::imread(path + "/lab3/outputs/image_quant_noise.png", 1);
cv::GaussianBlur(image, image_out, cv::Size(5, 5), 0);
```

Листинг 7: Исходный код фильтра Гаусса



Рис. 8: Изображения для до и после применения фильтра Гаусса (сверху вниз)

4. Нелинейная фильтрация

4.1. Медианная фильтрация

В классическом медианном фильтре используется маска с единичными коэффициентами. Произвольная форма окна может задаваться при помощи нулевых коэффициентов. Значения интенсивностей пикселей в окне представляются в виде вектора-столбца и сортируются по возрастанию. Отфильтрованному пикселью присваивается медианное (среднее) в ряду значение интенсивности. Номер медианного элемента после сортировки может быть вычислен по формуле

$$n = \frac{N + 1}{2}, \quad (6)$$

где N — число пикселей, участвующих в сортировке.

```
image = cv::imread(path + "/lab3/outputs/image_gauss_noise.png", 1);
cv::medianBlur(image, image_out, 3);
```

Листинг 8: Исходный код медианного фильтра



Рис. 9: Изображения для до и после применения медианного фильтра с $\text{kernel} = 3$

4.2. Адаптивная медианная фильтрация

В данной модификации фильтра скользящее окно адаптивно увеличивается в зависимости от результата фильтрации.

Основной идеей является увеличение размера окна до тех пор, пока алгоритм не найдет медианное значение, не являющееся импульсным шумом, или пока не достигнет максимального размера окна.

```
uchar adaptiveProcess(const cv::Mat &im, int row, int col, int kernelSize,
                      int maxSize)
{
    std::vector<uchar> pixels;
    for(int a = -kernelSize / 2; a <= kernelSize / 2; a++){
        for(int b = -kernelSize / 2; b <= kernelSize / 2; b++){
            pixels.push_back(im.at<uchar>(row + a, col + b));
        }
    }
    std::sort(pixels.begin(), pixels.end());
    auto min = pixels[0];
    auto max = pixels[kernelSize * kernelSize - 1];
    auto med = pixels[kernelSize * kernelSize / 2];
    auto zxy = im.at<uchar>(row, col);
    if(med > min && med < max){
        if(zxy > min && zxy < max){
            return zxy;
        }else{
            return med;
        }
    }
    else{
        kernelSize += 2;
        if(kernelSize <= maxSize)
            return adaptiveProcess(im, row, col, kernelSize, maxSize);
        else
            return med;
    }
}

cv::Mat amf_work(cv::Mat src){
    cv::Mat dst;
    int minSize = 3;
    int maxSize = 7;
    copyMakeBorder(src, dst, maxSize / 2, maxSize / 2, maxSize / 2, maxSize / 2, cv::BORDER_REFLECT);
    int rows = dst.rows;
    int cols = dst.cols;
    for(int j = maxSize / 2; j < rows - maxSize / 2; j++){
        for(int i = maxSize / 2; i < cols * dst.channels() - maxSize / 2;
            i++) {
```

```
        dst.at<uchar>(j, i) = adaptiveProcess(dst, j, i, minSize,
maxSize);
    }
}
return dst;
}
```

Листинг 9: Функции для применения адаптивного медианного фильтра

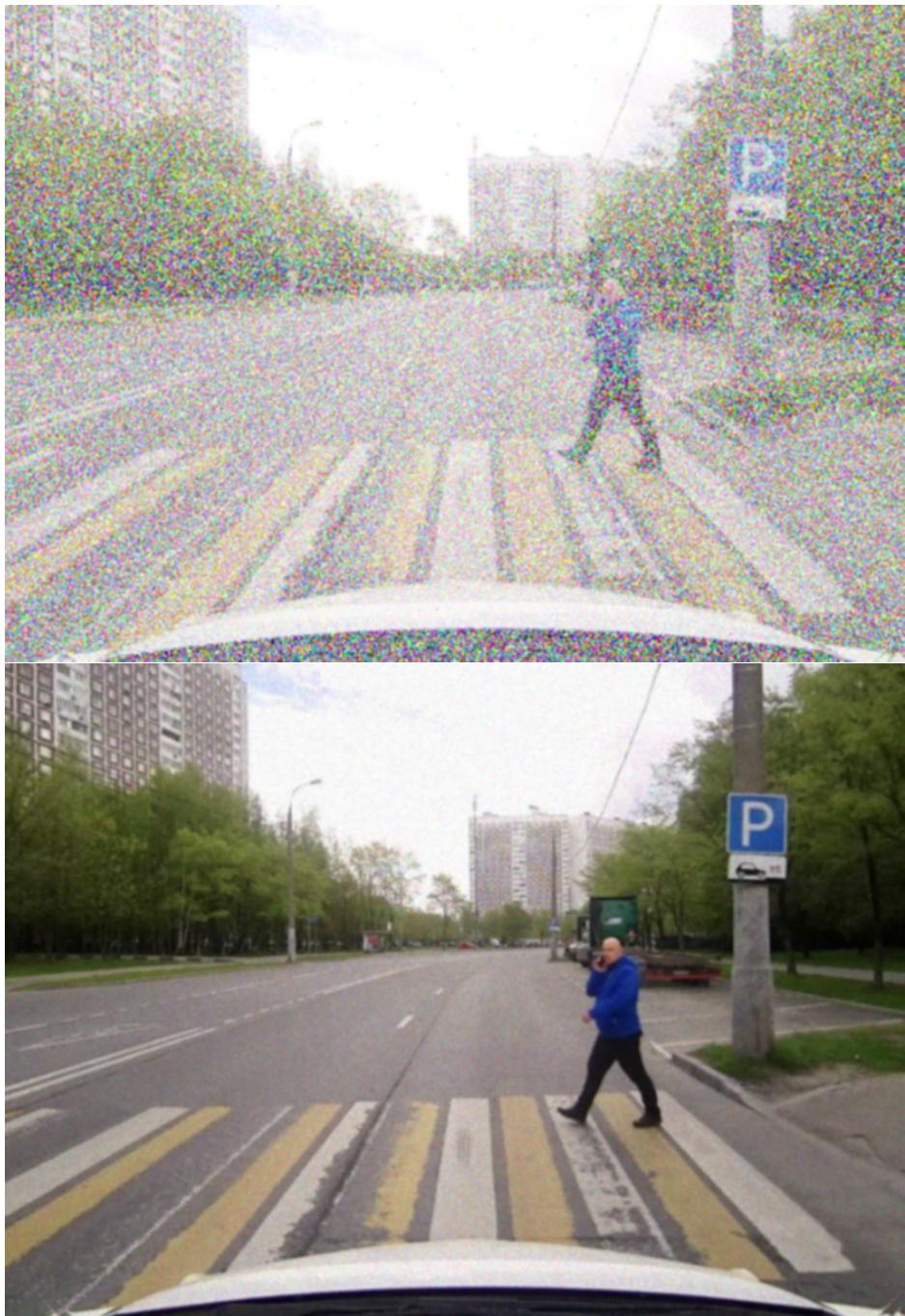


Рис. 10: Изображения до и после применения адаптивного медианного фильтра

4.3. Винеровская фильтрация

Использует пиксельно-адаптивный метод Винера, основанный на статистических данных, оцененных из локальной окрестности каждого пикселя.

```
int k_size[] = {5, 5};
cv::Mat kernel = cv::Mat::ones(k_size[0], k_size[1], CV_64F);

double k_sum = cv::sum(kernel)[0];

cv::Mat image_copy;
if(image.depth() == CV_8U)
    image.convertTo(image_copy, CV_32F, 1.0 / 255);\nelse
    image_copy = image;
cv::copyMakeBorder(image_copy, image_copy,
                  int((k_size[0] - 1) / 2),
                  int(k_size[0] / 2),
                  int((k_size[1] - 1) / 2),
                  int(k_size[1] / 2), cv::BORDER_REPLICATE);

std::vector<cv::Mat> bgr_planes;
cv::split(image_copy, bgr_planes);

for(int k = 0; k < bgr_planes.size(); k++){
    cv::Mat image_tmp = cv::Mat::zeros(image.size(), bgr_planes[k].type());
    double v(0);

    for(int i = 0; i < image.rows; i++)
        for(int j = 0; j < image.cols; j++){
            double m(0), q(0);
            for(int a = 0; a < k_size[0]; a++)
                for(int b = 0; b < k_size[1]; b++){
                    double t = bgr_planes[k].at<float>(i + a, j + b) *
kernel.at<double>(a, b);
                    m += t;
                    q += t*t;
                }

            m /= k_sum;
            q /= k_sum;
            q -= m*m;
            v += q;
        }
    v /= image.cols * image.rows;

    for(int i = 0; i < image.rows; i++)
        for(int j = 0; j < image.cols; j++){
            double m(0), q(0);
            for(int a = 0; a < k_size[0]; a++)
                for(int b = 0; b < k_size[1]; b++){
                    double t = bgr_planes[k].at<float>(i = a, j + b) *
```

```

kernel.at<double>(a, b);
    m += t;
    q += t*t;
}
m /= k_sum;
q /= k_sum;
q -= m*m;

        double im = bgr_planes[k].at<float>(i + (k_size[0] - 1) / 2, j
+ (k_size[1] - 1) / 2);
        if(q < v)
            image_tmp.at<float>(i, j) = float(m);
        else
            image.at<float>(i, j) = float((im - m) * (1 - v / q) + m);
    }
bgr_planes[k] = image_tmp;
}

cv::merge(bgr_planes, image_out);

```

Листинг 10: Исходный код Винеровского фильтра



Рис. 11: Изображения для до и после применения Винеровского фильтра

5. Высокочастотная фильтрация



Рис. 12: Исходное изображение

5.1. Фильтр Робертса

Фильтр Робертса работает с минимально допустимой для вычисления производной маской размерности 2×2 , поэтому является быстрым и довольно эффективным. Возможные варианты масок для нахождения градиента по осям Ox и Oy :

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} \quad (7)$$

```
cv::Mat G_x = (cv::Mat<double>(2, 2) << -1, 1, 0, 0);
cv::Mat G_y = (cv::Mat<double>(2, 2) << 1, 0, -1, 0);

cv::Mat I_x, I_y, I_out;

cv::filter2D(image, I_x, -1, G_x);
cv::filter2D(image, I_y, -1, G_y);
```

```
I_x.convertTo(I_x, CV_32F);
I_y.convertTo(I_y, CV_32F);

cv::magnitude(I_x, I_y, I_out);
```

Листинг 11: Исходный код фильтра Робертса



Рис. 13: Изображение после применения фильтра Робертса

5.2. Фильтр Превитта

В данном подходе используются две ортогональные маски размером 3×3 , позволяющие более точно вычислить производные по осям Ox и Oy :

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad (8)$$

```
image = cv::imread(path + "/lab3/source/SDC_Yandex.png", 1);
cv::Mat G_x = (cv::Mat<double>(3, 3) << -1, 0, 1, -1, 0, 1, -1, 0, 1);
cv::Mat G_y = (cv::Mat<double>(3, 3) << -1, -1, -1, 0, 0, 0, 1, 1, 1);

cv::Mat I_x, I_y, I_out;

cv::filter2D(image, I_x, -1, G_x);
cv::filter2D(image, I_y, -1, G_y);

I_x.convertTo(I_x, CV_32F);
I_y.convertTo(I_y, CV_32F);

cv::magnitude(I_x, I_y, I_out);
```

Листинг 12: Исходный код фильтра Превитта

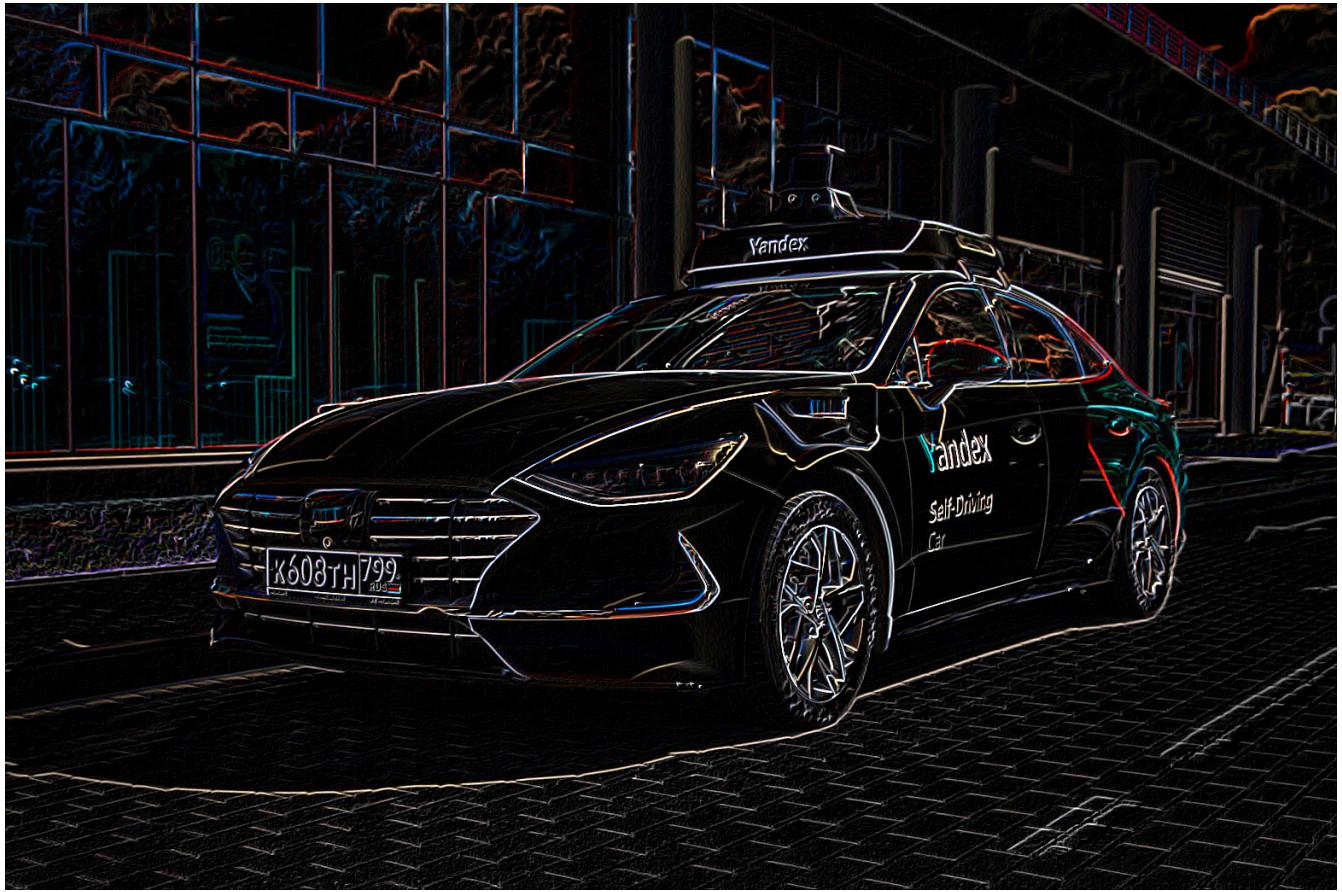


Рис. 14: Изображение после применения фильтра Превитта

5.3. Фильтр Превитта

Данный подход аналогичен фильтру Робертса, однако используются разные веса в масках.

Типичный пример фильтра Собела:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (9)$$

```
image = cv::imread(path + "/lab3/source/SDC_Yandex.png", 1);
cv::Mat G_x = (cv::Mat<double>(3, 3) << -1, 0, 1, -2, 0, 2, -1, 0, 1);
cv::Mat G_y = (cv::Mat<double>(3, 3) << 1, 2, 1, 0, 0, 0, -1, -2, -1);

cv::Mat I_x, I_y, I_out;

cv::filter2D(image, I_x, -1, G_x);
cv::filter2D(image, I_y, -1, G_y);

I_x.convertTo(I_x, CV_32F);
I_y.convertTo(I_y, CV_32F);
```

```
cv::magnitude(I_x, I_y, I_out);
```

Листинг 13: Исходный код фильтра Собела

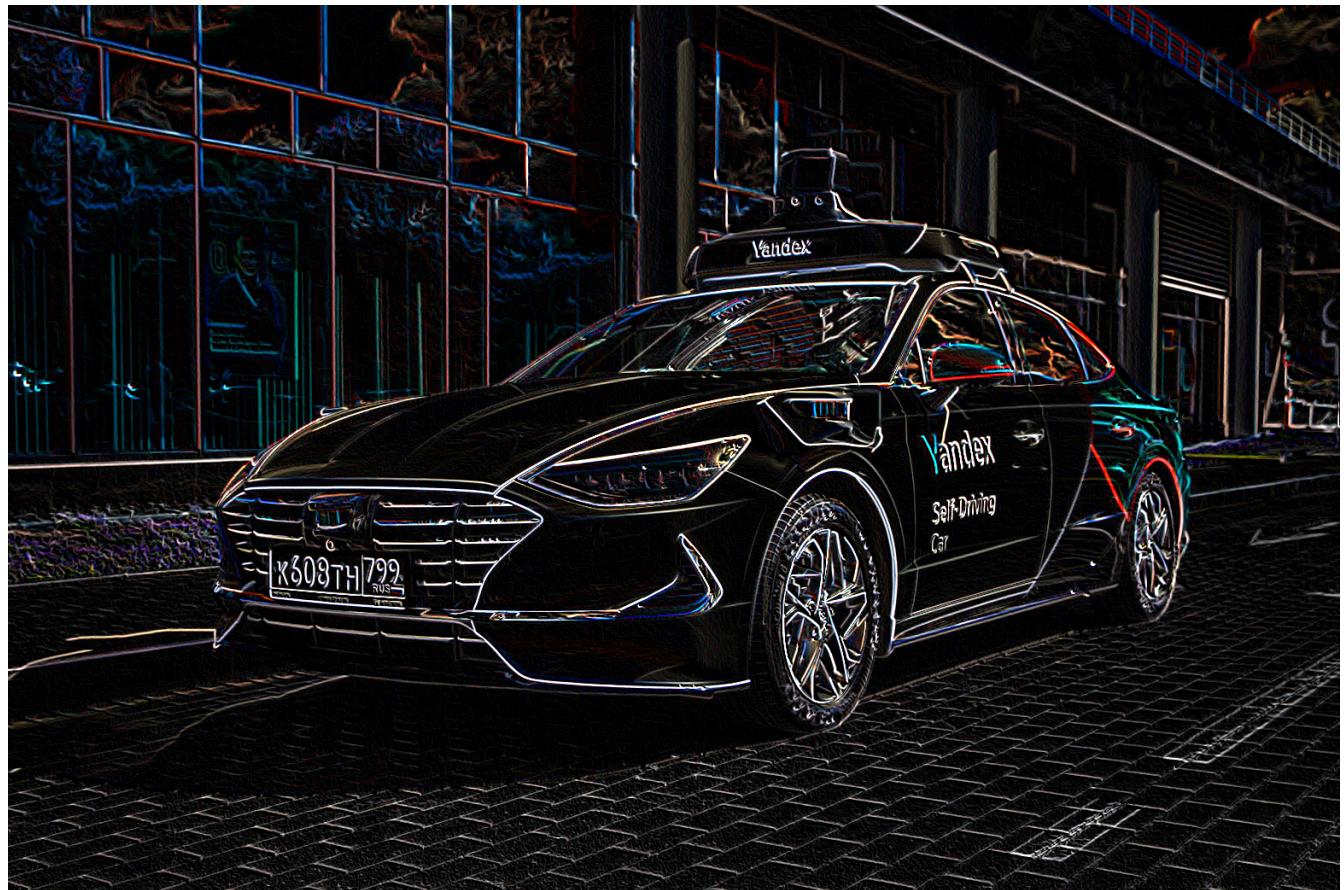


Рис. 15: Изображение после применения фильтра Собела

5.4. Фильтр Лапласа

Фильтр Лапласа использует аппроксимацию вторых производных по осям Ox и Oy , в отличие от предыдущих подходов, использующих первую производную. Формула:

$$w = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (10)$$

```
image = cv::imread(path + "/lab3/source/SDC_Yandex.png", 1);
cv::Mat G_x = (cv::Mat<double>(3, 3) << 0, -1, 0, -1, 4, -1, 0, -1, 0);
cv::Mat G_y = (cv::Mat<double>(3, 3) << 0, -1, 0, -1, 4, -1, 0, -1, 0);

cv::Mat I_x, I_y, I_out;

cv::filter2D(image, I_x, -1, G_x);
cv::filter2D(image, I_y, -1, G_y);

I_x.convertTo(I_x, CV_32F);
I_y.convertTo(I_y, CV_32F);

cv::magnitude(I_x, I_y, I_out);
```

Листинг 14: Исходный код фильтра Лапласа



Рис. 16: Изображение после применения фильтра Лапласа

5.5. Алгоритм Кэнни

Одним из самых распространенных и эффективных алгоритмов выделения контуров на изображении является алгоритм Кэнни. Данный алгоритм позволяет не только определять краевые пиксели, но и связные границы линии.

```
image = cv::imread(path + "/lab3/source/SDC_Yandex.png", 1);
cv::Canny(image, image_out, 50, 200);
```

Листинг 15: Исходный код применения алгоритма Кэнни

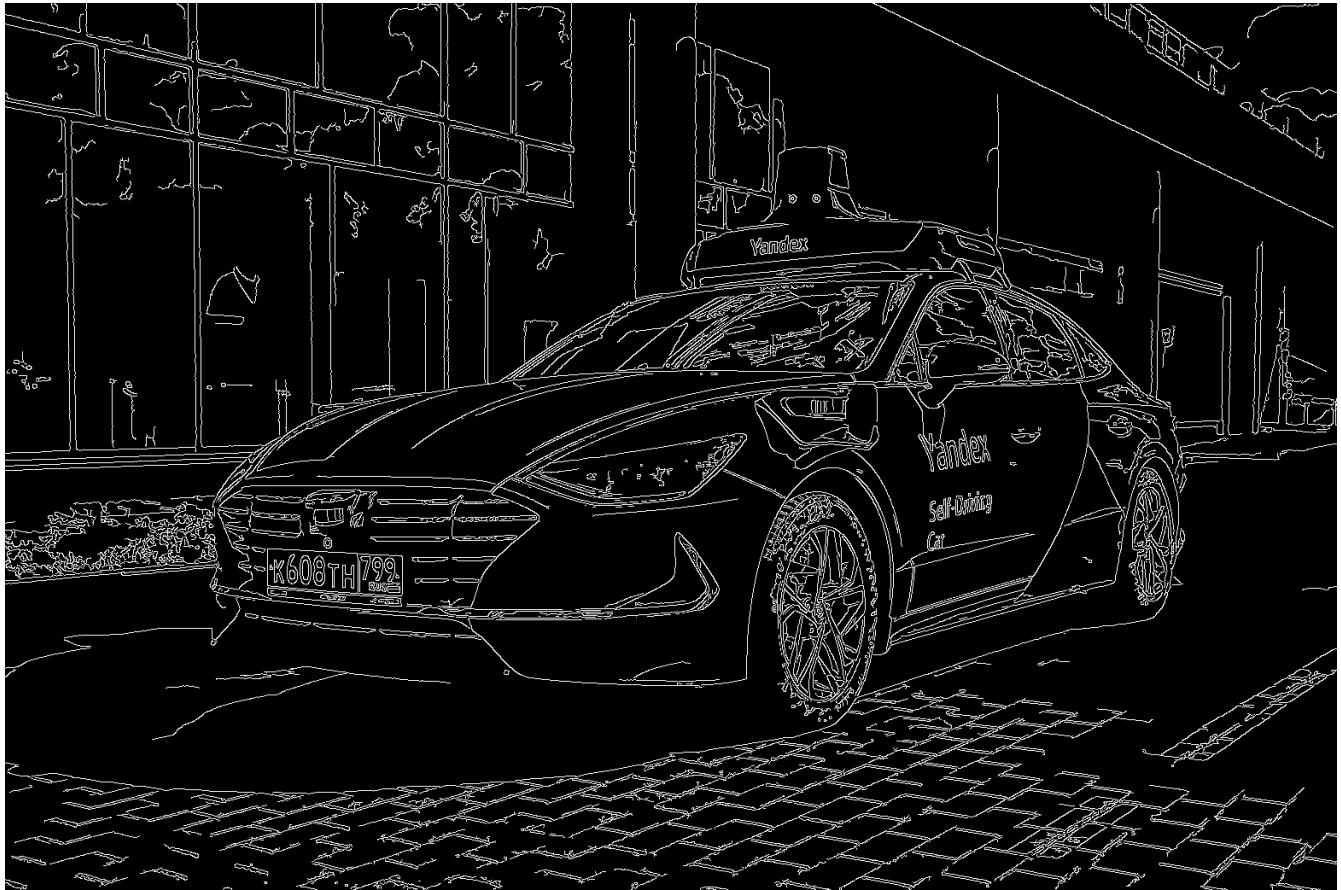


Рис. 17: Изображение после применения алгоритма Кэнни

6. Выводы

В процессе выполнения лабораторной работы мы освоили основные виды фильтраций изображений и использовали различные алгоритмы выделения границ на изображении, попробовали скорректировать шумы.

Также смогли оценить качество полученных изображений и убедиться, что фильтрация позволяет улучшать их визуальное восприятие

7. Ответы на вопросы

Q1. В чем заключаются основные недостатки адаптивных методов фильтрации изображений?

A1. Основными минусами данного метода являются: сильное размытие рисунка при недостаточном контрасте между светлыми и темными участками изображения и чувствительность к импульсивным помехам. Возможность возникновения артефактов:

в некоторых случаях адаптивные методы фильтрации могут приводить к появлению артефактов на изображениях, таких как размытие границ объектов или искажение текстур. Вычислительная сложность: адаптивные методы фильтрации могут быть более ресурсоемкими по сравнению с классическими методами, так как они требуют выполнения дополнительных вычислений для определения параметров фильтра.

Q2. При каких значениях параметра Q контргармонический фильтр будет работать как арифметический, а при каких как гармонический?

A2. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ — в гармонический.

Q3. Какими операторами можно выделить границы на изображении?

A3. На изображении можно выделить границы с помощью операторов, таких как оператор Собеля, оператор Робертса, оператор Превитта, оператор Кэнни, оператор Лапласа и т.д.

Q4. Для чего на первом шаге выделения контуров, как правило, выполняется низкочастотная фильтрация?

A4. Низкочастотная фильтрация на первом шаге выделения контуров используется для сглаживания изображения и уменьшения шумов, которые могут помешать правильному определению контуров. Фильтрация помогает убрать мелкие детали и делает контуры более четкими и выраженным, что облегчает дальнейший процесс обработки изображения.