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: Открывающий фильтр

Студент гр. 3331506/70401 Архипов А. Е.
Преподаватель Варлашин В. В.
« »_____2020 г.

Санкт-Петербург
2020

Задание

Пользуясь средствами языка C++ и библиотеки OpenCV, реализовать открывающий фильтр для бинарного изображения с ядром приведенном на рисунке 1. Граница имеет тип border reflect.

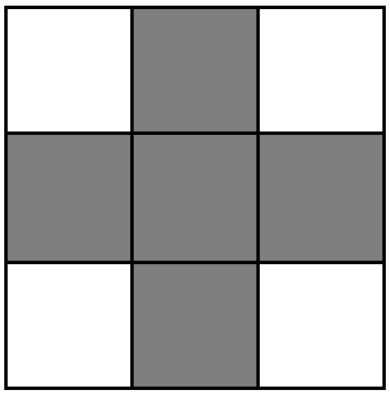

Рисунок 1 – Ядро фильтра

Ход работы

Описание алгоритма

Открывающий фильтр состоит в последовательном применении эрозии и дилатации с одинаковым структурным элементом. Его морфологический эффект заключается в удалении малых изолированных частей фигуры.

Применение эрозии сводится к проходу шаблоном по всему изображению и применению оператора поиска локального минимума к интенсивностям пикселей изображения, которые накрываются шаблоном.

Применение дилатации сводится к проходу шаблоном по всему изображению и применению оператора поиска локального максимума к интенсивностям пикселей изображения, которые накрываются шаблоном.

Реализация алгоритма

Для описания открывающего фильтра был создан класс *Filter*, изображенный на рисунке 2.

```
class Filter
{
private:
    Mat m_image;
    Mat m_imageBinar;
    int m_kernel[9];
    Mat m_imageOriginal;
    Mat m_imageWithBorder;
    Mat m_kerneMat;

public:

    Filter();
    ~Filter();
    Mat binarization(Mat inputImage);
    void padding();
    Mat getImage();
    void setImage(const Mat image);
    void erode_custom();
    void dilate_custom();
    Mat opening_custom();
    Mat comparisonWithOpencv();
    void comparisonRunTime();
};
```

Рисунок 2 – Класс *Filter*

Алгоритм реализован для бинарных изображений, перед тем как Публичный метод *opening_custom()*, код которого представлен на рисунке 3, сначала производится бинаризация изображения методом Оцу, затем последовательное применение эрозии и дилатации, расширив изображение перед морфологическими операциями.

```
Mat Filter::opening_custom()
{
    m_image = binarization(m_image);
    padding();
    erode_custom();
    padding();
    dilate_custom();
    return m_image;
}
```

Рисунок 3 – Метод *opening_custom()*

Методы, используемые в *opening_custom()*, рассматриваются ниже.

Для вычисления расширения изображения реализован метод *padding()*, код которого представлен на рисунке 4.

```
void Filter::padding()
{
    Mat m_imageWithBorder(m_image.rows + 2, m_image.cols + 2, CV_8UC1);
    for (int rows_number = 1; rows_number < m_imageWithBorder.rows - 1; rows_number++)
    {
        for (int cols_number = 1; cols_number < m_imageWithBorder.cols - 1; cols_number++)
        {
            m_imageWithBorder.at<uchar>(rows_number, cols_number) = m_image.at<uchar>(rows_number - 1,
                                                                                      cols_number - 1);
        }
    }
    for (int i = 1; i < m_imageWithBorder.cols - 1; i++)
    {
        m_imageWithBorder.at<uchar>(0, i) = m_imageWithBorder.at<uchar>(1, i);
        m_imageWithBorder.at<uchar>(m_imageWithBorder.rows - 1, i) =
            m_imageWithBorder.at<uchar>(m_imageWithBorder.rows - 2, i);
    }
    for (int i = 0; i < m_imageWithBorder.rows; i++)
    {
        m_imageWithBorder.at<uchar>(i, 0) = m_imageWithBorder.at<uchar>(i, 1);
        m_imageWithBorder.at<uchar>(i, m_imageWithBorder.cols - 1) =
            m_imageWithBorder.at<uchar>(i, m_imageWithBorder.cols - 2);
    }
    m_image = m_imageWithBorder.clone();
}
```

Рисунок 4 – Метод *padding()*

Для реализации эрозии был создан метод *erode_custom()*, код которой изображен на рисунке 5.

```
void Filter::erode_custom()
{
    Mat image(m_image.rows - 2, m_image.cols - 2, CV_8UC1);
    for (int rows_number = 1; rows_number < m_image.rows - 1; rows_number++)
    {
        for (int cols_number = 1; cols_number < m_image.cols - 1; cols_number++)
        {
            if (m_image.at<uchar>(rows_number - 1, cols_number - 1) == m_kernel[0] ||
                m_image.at<uchar>(rows_number - 1, cols_number) == m_kernel[1] ||
                m_image.at<uchar>(rows_number - 1, cols_number + 1) == m_kernel[2] ||
                m_image.at<uchar>(rows_number, cols_number - 1) == m_kernel[3] ||
                m_image.at<uchar>(rows_number, cols_number) == m_kernel[4] ||
                m_image.at<uchar>(rows_number, cols_number + 1) == m_kernel[5] ||
                m_image.at<uchar>(rows_number + 1, cols_number - 1) == m_kernel[6] ||
                m_image.at<uchar>(rows_number + 1, cols_number) == m_kernel[7] ||
                m_image.at<uchar>(rows_number + 1, cols_number + 1) == m_kernel[8])

            {
                image.at<uchar>(rows_number - 1, cols_number - 1) = 0;
            }
            else
            {
                image.at<uchar>(rows_number - 1, cols_number - 1) = 255;
            }
        }
    }
    m_image = image.clone();
}
```

Рисунок 5 – Метод *erode_custom()*

Для реализации дилатации изображения был создан метод *dilate_custom()*, код которого изображен на рисунке 6.

```
void Filter::dilate_custom()
{
    Mat image(m_image.rows - 2, m_image.cols - 2, CV_8UC1);
    for (int rows_number = 1; rows_number < m_image.rows - 1; rows_number++)
    {
        for (int cols_number = 1; cols_number < m_image.cols - 1; cols_number++)
        {
            if (m_image.at<uchar>(rows_number - 1, cols_number - 1) != m_kernel[0] ||
                m_image.at<uchar>(rows_number - 1, cols_number) != m_kernel[1] ||
                m_image.at<uchar>(rows_number - 1, cols_number + 1) != m_kernel[2] ||
                m_image.at<uchar>(rows_number, cols_number - 1) != m_kernel[3] ||
                m_image.at<uchar>(rows_number, cols_number) != m_kernel[4] ||
                m_image.at<uchar>(rows_number, cols_number + 1) != m_kernel[5] ||
                m_image.at<uchar>(rows_number + 1, cols_number - 1) != m_kernel[6] ||
                m_image.at<uchar>(rows_number + 1, cols_number) != m_kernel[7] ||
                m_image.at<uchar>(rows_number + 1, cols_number + 1) != m_kernel[8])

            {
                image.at<uchar>(rows_number - 1, cols_number - 1) = 255;
            }
            else
            {
                image.at<uchar>(rows_number - 1, cols_number - 1) = 0;
            }
        }
    }
    m_image = image.clone();
}
```

Рисунок 6 – Метод *dilate_custom()*

Сравнение с методом из OpenCV

Для сравнения использовалось монохромное изображение разрешением 1065 на 1300 пикселей (см. рисунок 7).

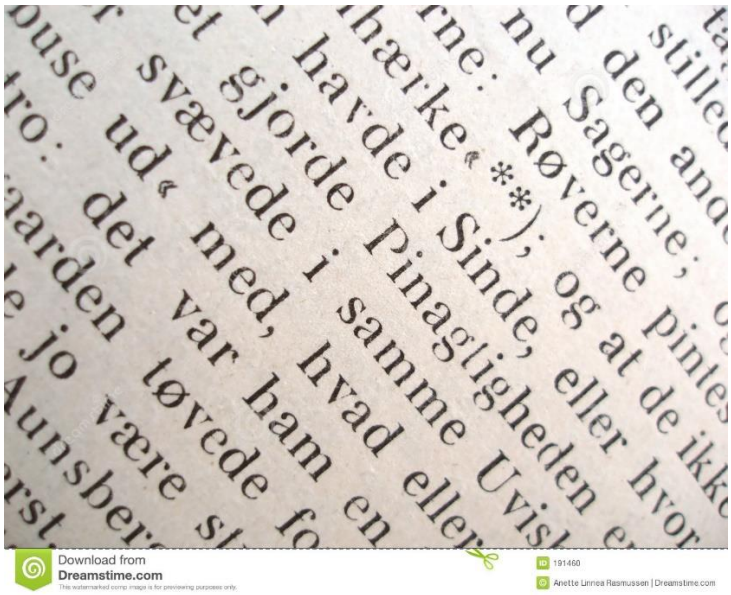

Рисунок 7 – Оригинальное изображение

Изображения, обработанные реализованным фильтром и фильтром из OpenCV, приведены ниже.

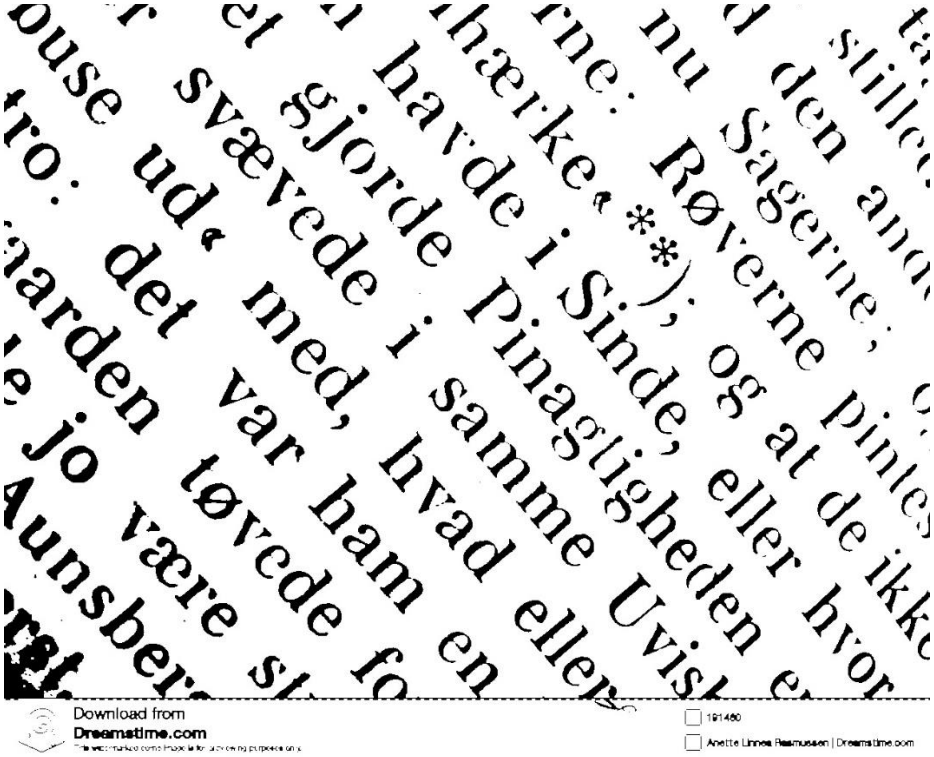

Рисунок 9 – Результат собственной реализации

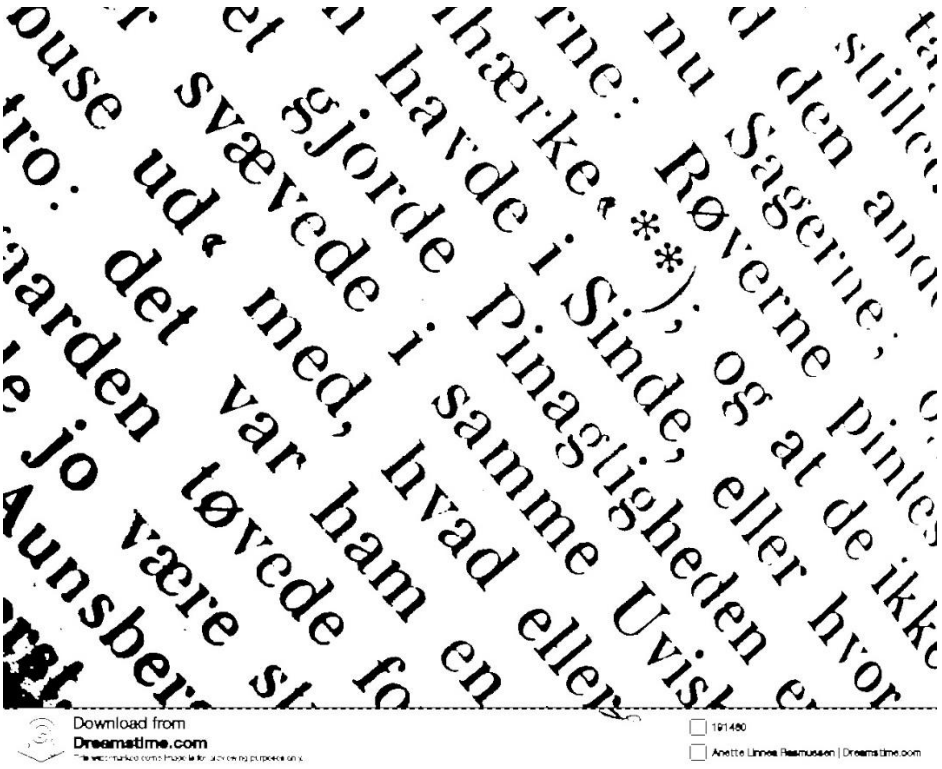

Рисунок 9 – Результат реализации OpenCV

Для точного сравнения изображений реализован вспомогательный метод *comparisonWithOpencv()*. Метод выводит в консоль отношение правильных результатов к общему значению пикселей и количество отличающихся пикселей.

При сравнении результатов обработки данный метод вывел 1 и 0, т. е. результаты обработки полностью совпадают.

Для сравнения времени выполнения обработки изображения использовался метод *comparisonRunTime(),* который выводит в консоль количество мс выполнения алгоритма в библиотеке OpenCV и собственной реализации.

Таким образом, результаты определения времени открывающего фильтра изображения разрешением 1065x1300 собственной реализации и реализацией из OpenCV представлены в таблице 1.

Таблица 1 – Сравнение времени выполнения

Конфигурация решения	Собственная реализация	OpenCV
Debug	2493 мс	18 мс
Release	26 мс	3 мс

Время выполнения обработки реализованным методом в конфигурации Release меньше, чем в Debug в ≈100 раз. Это связано с тем, что в конфигурации Release включена максимальная оптимизация (приоритет скорости) (/O2), а в Debug оптимизация отключена (/Od).

Вывод

В результате выполнения лабораторной работы был реализован открывающий фильтр. Результаты фильтра реализованным методом и методом из OpenCV полностью совпали, однако метод из OpenCV более оптимизирован по времени выполнения.