Лабораторная работа №4. Операции с ядром (Kernel)

Эта работа является четвертой в серии учебных проектов по анализу и обработке цифровых изображений. В предыдущих работах мы изменяли пиксели по отдельности (яркость, контраст) или меняли их координаты (геометрические преобразования). В данной работе мы переходим к **свертке** — фундаментальной операции, где значение пикселя зависит от его окружения.

Матричная свертка

Теоретическая информация

Основным инструментом для такой обработки является операция **свертки**.
Суть метода заключается в том, что по изображению "скользит" окно определенного размера (обычно нечетного: 3x3, 5x5 и т.д.). Значения пикселей, попавших в это окно, умножаются на соответствующие коэффициенты специальной матрицы, называемой **Ядром (Kernel)**, а затем суммируются.

Формула №1. Операция свертки для пикселя.

$$NewValue = \sum_{i=-k}^{k} \sum_{j=-k}^{k} (Pixel(x+i, y+j)) \times Kernel(i, j))$$

где, k - радиус ядра (для матрицы 3x3 радиус равен 1);
Pixel- значение интенсивности пикселя в координатах x,y;
Kernel - коэффициент веса в матрице ядра

Полученная сумма записывается в центральный пиксель выходного изображения.

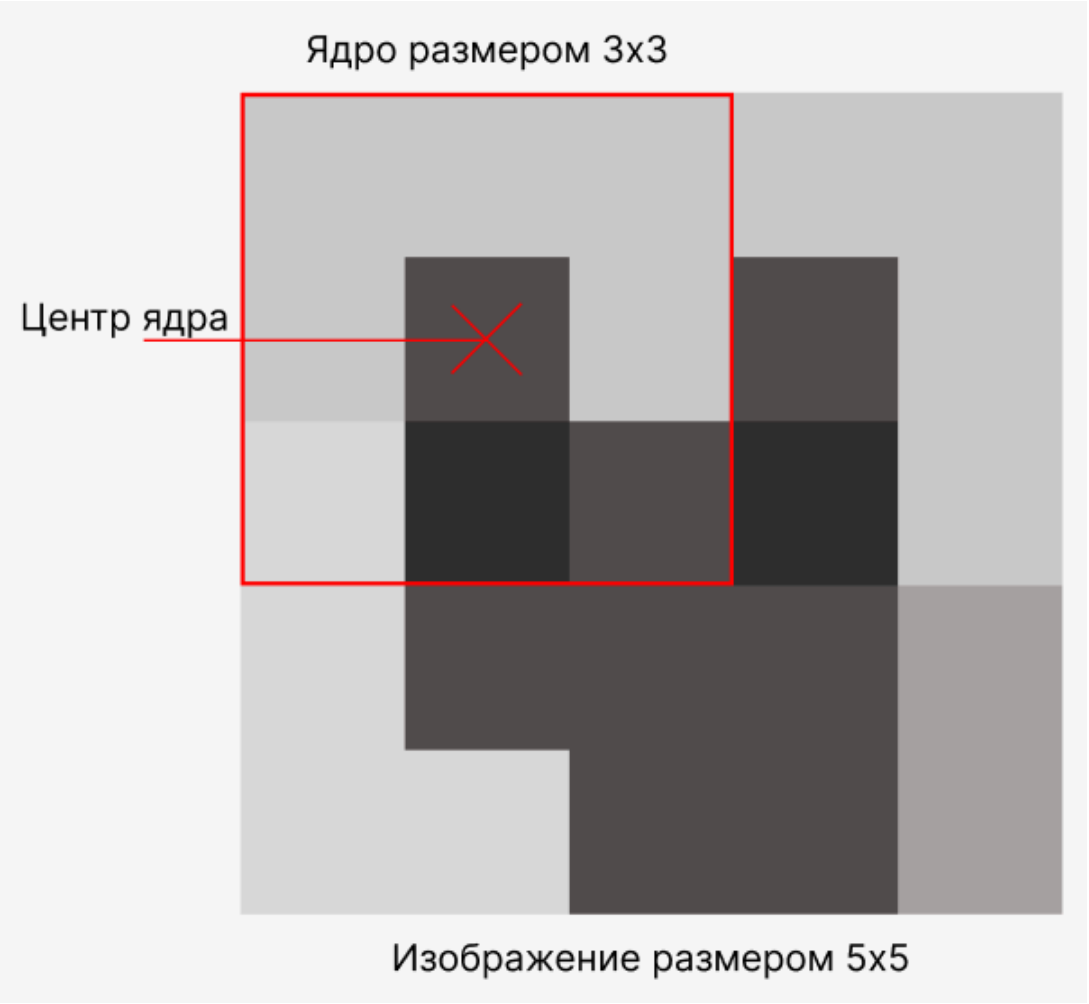

Рисунок №1. Визуализация ядра 3x3 на изображении.

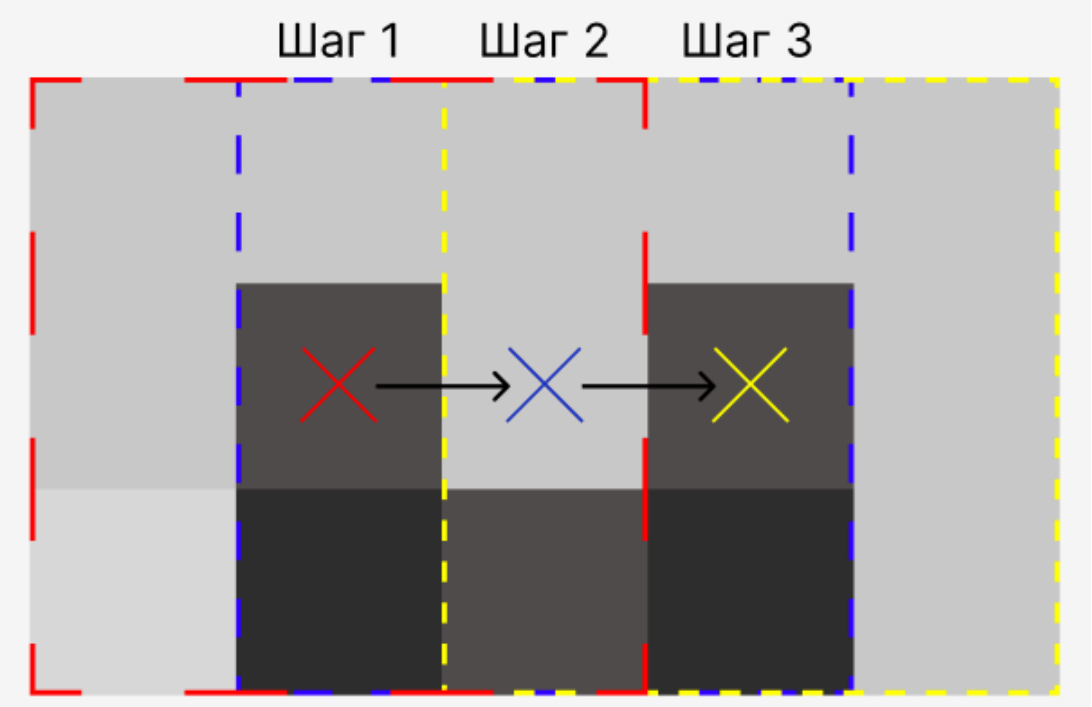

Рисунок №2. Перемещение ядра 3x3 по изображению.

Реализация универсального фильтра

Поскольку алгоритм обхода изображения одинаков для большинства фильтров, разумно реализовать одну универсальную функцию, которая принимает на вход изображение и матрицу ядра.

Блок кода №1. Универсальная функция применения свертки к черно-белому изображению.

```
//Функция применяет операцию свертки к цветному изображению.
//Свертка – это процесс, где новый цвет каждого пикселя вычисляется как взвешенная сумма цветов его соседей.
//Веса задаются ядром (матрицей (kernel)). Это основа для множества эффектов: размытия, повышения резкости, выделения границ и т.д.

private Image<Gray, byte> ApplyConvolution(Image<Gray, byte> input, double[,] kernel)
{
    int kernelSize = kernel.GetLength(0); //Размер ядра, в нашем случае 3 для матрицы 3x3.
    int kernelRadius = kernelSize / 2; //Радиус ядра, в нашем случае 1 для ядра 3x3.

    //Мы создаем копию изображения, потому что для расчета каждого нового пикселя нужны ОРИГИНАЛЬНЫЕ значения соседних пикселей
    Image<Gray, byte> output = new Image<Gray, byte>(input.Size);

    //Основной цикл проходит по всем пикселям, которые могут быть центром ядра, не выходя за границы изображения. Поэтому мы "пропускаем" края.
    for (int y = kernelRadius; y < input.Height - kernelRadius; y++)
    {
        for (int x = kernelRadius; x < input.Width - kernelRadius; x++)
        {
            double sum = 0;

            //Проход по ядру фильтра
            for (int ky = -kernelRadius; ky <= kernelRadius; ky++)
            {
                for (int kx = -kernelRadius; kx <= kernelRadius; kx++)
                {
                    //Прямой доступ к данным изображения. `[y, x, 0]` – 0 означает первый (и единственный) канал т.к. изображение в градациях серого.
                    byte neighborPixel = input.Data[y + ky, x + kx, 0];

                    //Получаем соответствующее значение (вес) из ядра.
                    double kernelValue = kernel[ky + kernelRadius, kx + kernelRadius];

                    //Умножаем цвет соседа на вес из ядра и прибавляем к общей сумме.
                    sum += neighborPixel * kernelValue;
                }
```

```
            }

            //Приводим значение к байту и записываем в выходное изображение
            output.Data[y, x, 0] = (byte)Math.Max(0, Math.Min(255, sum));
        }
    }
    return output;
}
```

Важно: это уже отмечалось ранее в предыдущих лабораторных работах, но в данной функции мы обращаемся к изображению через `Data[,,]`, что является медленным поиском в памяти. В реальности используется прямой доступ через небезопасные указатели.

Задача №1.

Реализовать в программе следующие функции:

- Универсальную функцию применения свертки к черно-белому изображению;
- Реализовать перегрузку метода `ApplyConvolution()` которая будет работать с цветными изображениями;
- Возможность применения к изображению ядра размером 3x3 введенного пользователем через интерфейс программы. Реализовать два варианта работы: применение к черно-белому или цветному изображению.

Фильтры

Фильтры размытия

Основная задача фильтров размытия (или сглаживающих) — подавление шума и устранение мелких деталей. С точки зрения частотного анализа, это фильтры низких частот (Low-pass filters): они пропускают плавные изменения цвета (низкие частоты) и глушат резкие скачки (высокие частоты).

Усредняющее размытие (Box Blur)

Это самый простой и интуитивный метод размытия. Его логика строится на принципе среднего арифметического.

Принцип работы заключается в том, что мы предполагаем, что все пиксели в окрестности имеют **равное значение** для формирования итогового цвета. Пиксель, стоящий вплотную к центру, и пиксель на краю окна влияют на результат одинаково.

Ядро фильтра (Kernel) заполняется единицами. Чтобы изображение не стало белым (не повысилась общая яркость), сумму значений всех соседей необходимо разделить на их количество (площадь окна).

Формула №2. Пример ядра 3x3 для фильтра усредняющего размытия.

$$K = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

где K - интенсивность пикселя после применения ядра

Зачастую операции которые применяются к результату работы ядра можно перенести в само ядро. Для операции усредненного размытия ядро будет иметь следующий вид:

Формула №3. Ядро 3x3 для фильтра усредняющего размытия с внесенной внутрь операцией умножения.

$$K = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

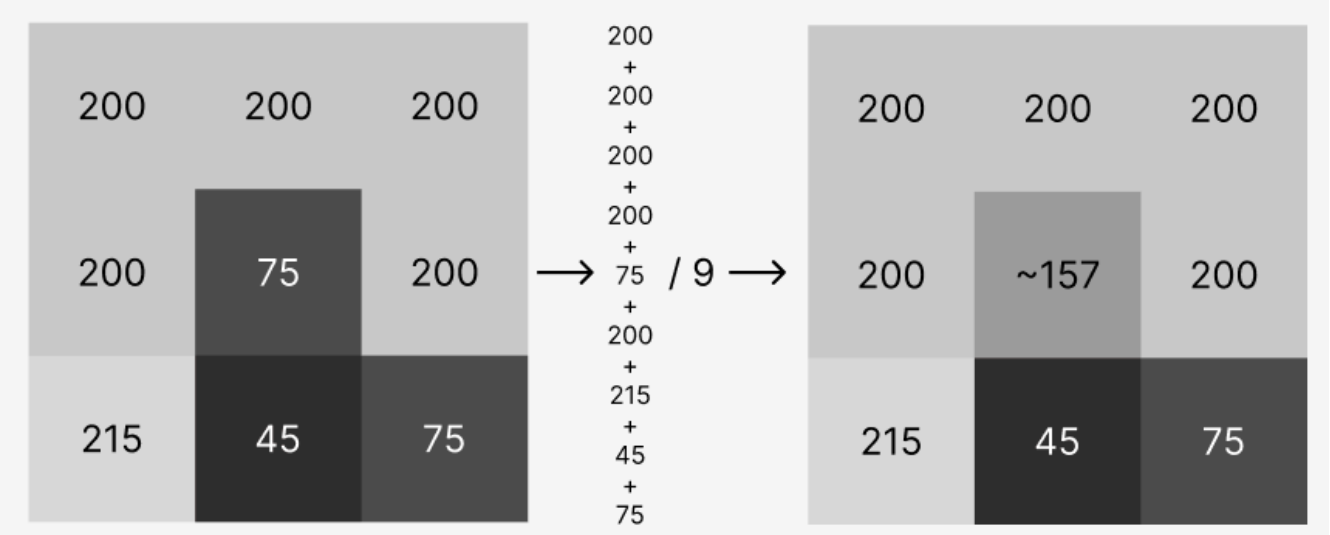

Рисунок №3. Применение усредняющего фильтра к участку изображения.

Размытие по Гауссу

Этот фильтр имитирует естественное размытие, которое мы видим в оптике (расфокусировка камеры) или в природе (рассеивание света).

Логика усредняющего размытия ошибочна с точки зрения физики. Соседи, находящиеся ближе к центру, должны оказывать большее влияние на цвет, чем те, что находятся на периферии.

Распределение весов в ядре подчиняется **нормальному распределению Гаусса:**

Формула №4. Нормальное распределение Гаусса.

$$G(x,y) = \frac{1}{2\pi\sigma^2} \times e^{-\frac{x^2+y^2}{2\sigma^2}}$$

где, x,y - расстояние от центра ядра;
σ - стандартное отклонение. Главный параметр, определяющий "силу" размытия. Чем больше сигма, тем шире "колокол" и тем сильнее влияют далекие пиксели.

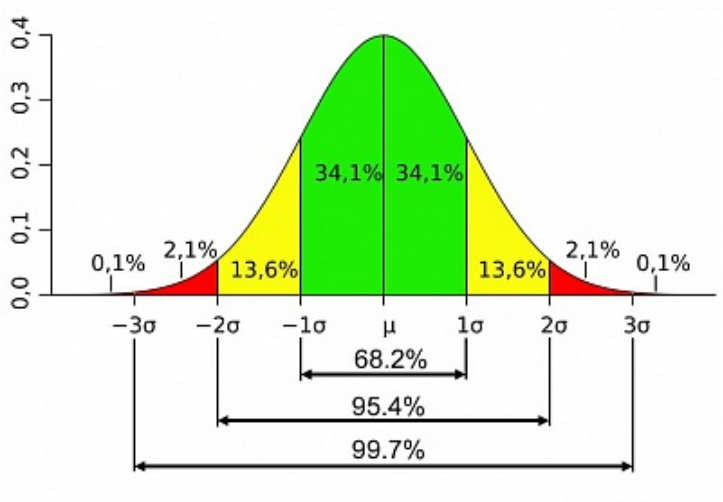

Рисунок №3. Нормальное распределение по Гауссу.

Дискретное приближение значений ядра (в случае ядра 3x3) будет иметь следующий вид:

Формула №5. Пример ядра 3x3 для фильтра размытия по Гауссу.

$$K = \frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Обратите внимание:

1. Центральный пиксель имеет самый большой вес 4.
2. Прямые соседи имеют средний вес 2.
3. Угловые соседи имеют минимальный вес 1.
4. Сумма всех коэффициентов равна 16, поэтому мы делим результат на 16.

Медианный фильтр

До этого момента мы рассматривали **линейные** фильтры. Их общий принцип: «Новый цвет — это взвешенная сумма цветов соседей». Однако у линейных фильтров есть фатальный недостаток: они «размазывают» шум по окрестности, но не удаляют его полностью.

Медианный фильтр работает по принципу статистической выборки. Он не усредняет значения, а выбирает наиболее «типичного» представителя из окрестности.

Представьте, что на изображении есть битые пиксели: случайные белые или черные точки. При использовании усредняющего размытия яркая белая точка (255) усреднится с восемью темными соседями (0). Вместо удаления точки мы получим серое грязное пятно размером 3x3. Шум не исчез, он просто расфокусировался.

Медианный фильтр также использует скользящее окно (обычно 3x3, 5x5), но вместо матрицы коэффициентов он выполняет следующие действия:

1. Значения всех пикселей, попавших в окно, записываются в одномерный массив (список). Для окна 3x3 это будет 9 значений.
2. Список сортируется по возрастанию яркости.
3. Выбирается значение, которое находится ровно посередине отсортированного списка (центральный элемент).
4. Это значение присваивается центральному пикселю.

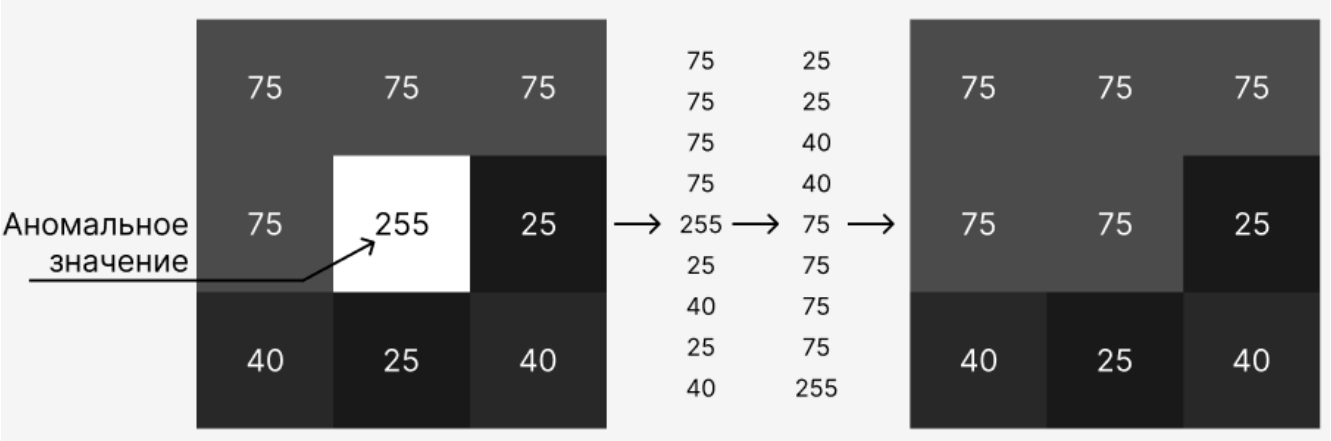

Рисунок №4. Пример работы медианного фильтра.

Фильтр повышения резкости

Если сглаживающие фильтры являются фильтрами низких частот (убирают детали), то фильтры резкости — это фильтры высоких частот (High-pass filters). Они усиливают перепады яркости, делая границы объектов более четкими и выраженными.

Чтобы усилить разницу между центральным пикселем и его соседями, мы должны:

1. Придать центральному пикселю большой **положительный** вес.
2. Придать соседям **отрицательный** вес.

Формула №6. Пример ядра 3x3 для фильтра повышения резкости.

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Так как сумма коэффициентов в конце равна единице (9 + (-1 * 8) = 1), общая яркость изображения не изменилась. Если бы сумма была равна нулю, изображение стало бы черным.

Задача №2.

Реализовать в программе следующие функции:

- Функции применения фильтров усредняющего размытия, размытия по гауссу и повышения резкости к цветным и черно-белым изображениям.
- Функция применения медианного фильтра к черно-белому изображению.

Для медианного фильтра рекомендуется написать свою функцию, в основе которой может лежать метод `ApplyConvolution()`.

Выделение границ. Оператор Собеля.

Если размытие и резкость — это инструменты художника, то **оператор Собеля** — это инструмент компьютерного зрения. Чтобы понять, что изображено на картинке, компьютер сначала должен найти контуры объектов. Контур — это место, где яркость изображения резко меняется. Оператор Собеля вычисляет этот градиент.

В математике скорость изменения функции показывает **производная**:

- На плавном градиенте производная низкая (мало изменений).
- На резкой границе (переход от черного к белому) производная огромная.

Поскольку изображение — это дискретная сетка, мы не можем взять настоящую производную. Но мы можем аппроксимировать её, вычитая значение левого соседа из правого:

Формула №7. Разница соседних пикселей.

$$\Delta X = Pixel(x+1, y) - Pixel(x-1, y)$$

Оператор Собеля делает немного больше чем находит простую разницу соседей: он комбинирует поиск разницы и сглаживание, чтобы шум не принимался за границы. Но границы бывают вертикальные, горизонтальные и диагональные. Одним ядром их не различить. Поэтому алгоритм Собеля использует **два прохода** с разными ядрами.

Формула №8. Ядро поиска вертикальных линий.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

где, Gx - результат колебаний разницы вертикальных линий.

Обратите внимание: слева отрицательные числа, справа положительные. Если слева темно, а справа светло — результат будет большим положительным. Это вертикальная граница.

Формула №9. Ядро поиска горизонтальных линий.

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

где, Gy - результат колебаний разницы горизонтальных линий.

Реализация оператора Собеля сложнее обычного фильтра из-за проблемы типов данных:

1. Исходное изображение конвертируется в оттенки серого. Цвет будет только мешает.
2. Затем, по-отдельности, применяется ядро для горизонтальных линий и для вертикальных для того чтобы найти матрицы Gx и Gy.
   - **ВАЖНО!** В данном случае результат свертки может быть отрицательным и выходить за пределы `byte`. Поэтому промежуточный результат нужно сохранять в результате `float` или `double`.
3. Чтобы получить итоговую картинку с границами, нам нужно объединить результаты. Мы вычисляем длину вектора градиента по теореме Пифагора:

Формула №10. Вычисление градиента.

$$G = \sqrt{G_x^2 + G_y^2}$$

где, G - результат вычисления градиента.

4. Полученные результаты в матрице G требуется нормализовать чтобы результаты можно было записать обратно в `byte`.

```
//Находим максимальное значение градиента на изображении.
double maxVal = 0;

for (int y = 0; y < G.Height; y++)
{
    for (int x = 0; x < G.Width; x++)
    {
        if (G.Data[y, x, 0] > maxVal)
        {
            maxVal = G.Data[y, x, 0];
        }
    }
}

//Масштабируем изображение так, чтобы maxVal стал 255.
Image<Gray, byte> normalizedImage = G.ConvertScale<byte>(255.0 / maxVal, 0);
```

Блок кода №2. Пример кода нормализации значений матрицы G.

Примечание. Для того чтобы вычислить для визуализации не полный набор границ, а лишь значения по горизонтали или вертикали вместо пункта №3 можно использовать следующую формулу:

Формула №11. Вычисление градиента для одной границы.

$$G = \left|G_{x/y}\right|$$

где, Gx/y -матрица, градиент которой требует вычисления.

Пункт №4 выполняется без изменений.

Задача №3.

Реализовать в программе следующие функции:

- Функции вычисления границ изображения: горизонтальных, вертикальных и комбинации горизонтальных и вертикальных. Визуализация результата происходит в градациях серого.

Для выполнения рекомендуется реализовать новую функцию на основе функции `ApplyConvolution()` со следующей сигнатурой:

```
private Image<Gray, float> ApplyConvolutionForSobel(Image<Gray, byte> input,
double[,] kernel)
```

Задание на лабораторную работу №4. Операции с ядром (Kernel).

В данной лабораторной работе требуется доработать редактор созданный в предыдущей лабораторной работе добавив в него следующие функции:

1. Функция ручного ввода матрицы 3x3 через интерфейс программы. Применение введенной матрицы к черно-белым и цветным изображениям.
2. Функции применения фильтров усредняющего размытия, размытия по гауссу и повышения резкости к цветным и черно-белым изображениям.
3. Функция применения медианного фильтра к черно-белым изображениям.
4. Функции вычисления границ изображения: горизонтальных, вертикальных и комбинации горизонтальных и вертикальных. Визуализация результата происходит в градациях серого.

Полезные ссылки

1. GitHub репозиторий с учебным проектом: https://github.com/TheSkyEye1/AOCI-Lab4-Kernel-Operators
2. GitHub релиз проекта демонстрирующий весь функционал реализуемый в работе: https://github.com/TheSkyEye1/AOCI-Lab4-Kernel-Operators/releases/tag/1.0.0
3. Документация Emgu.CV: https://www.emgu.com/wiki/index.php?title=Main_Page
4. Интерактивная демонстрация работы разных фильтров: https://setosa.io/ev/image-kernels/
5. Статья на википедии о ядре: https://en.wikipedia.org/wiki/Kernel_(image_processing)

Дополнительная информация

Проблема границ

В основной части лабораторной предложен простой алгоритм прохода по изображению ядром. Это значит, что у итоговой картинки будет черная рамка шириной в радиус ядра. Для ядра 3x3 — 1 пиксель. Для размытия 15x15 — 7 пикселей.

Основные методы решения этой проблемы:

1. **Копирование:** Крайний пиксель повторяется бесконечно. Самый популярный и дешевый метод.

2. **Отражение:** Изображение отражается зеркально. Дает наилучший визуальный результат, так как сохраняет непрерывность градиентов.
3. **Зацикливание:** Изображение зацикливается - левый край берется справа, верх берется снизу и т.д. Нужно только для бесшовных текстур.

Сверточные нейронные сети (CNN)

То, что вы делали в этой лабораторной — это фундамент современных нейросетей.

- В этой лабораторной вы **вручную** задавали числа в матрице ядра, чтобы найти границы или размыть шум.
- Сверточная нейросеть (CNN) занимается тем же самым, но она сама подбирает числа в этих матрицах в процессе обучения.

Первые слои любой нейросети, распознающей лица — это, по сути, набор фильтров Собеля и Гаусса, которые сеть "изобрела" сама, чтобы видеть контуры глаз и носа. Вы только что написали первый слой нейросети своими руками.