Практическая работа 5

Материалы для выполнения практической работы могут быть получены с помощью команды:

```
$ wget https://cv-course.demo.3divi.com/download/task5/practical
$ unzip practical
```

Устройство фильтра

Для лучшего понимания принципа работы свертки и применения фильтра рассмотрим пример, содержащий наивную реализация фильтра на Python, а также применение этого фильтра к изображению размером 28x28 пикселей. В качестве примера рассмотрим простой сглаживающий фильтр, который суммирует значения всех пикселей в окне и делит его на размер окна. Как было упомянуто в лекции, для того чтобы отфильтровать изображение нужно "приложить" центр свертки (так называемый якорь - *anchor*) к каждому пикселю и "свернуть" его с соседями, попадающими в окно свертки. Но что делать с теми пикселями изображения, которые лежат на его краях? Например, рассмотрим левый верхний пиксель, находящийся в точке (0,0): если мы "совместим" его с центром матрицы свертки, то окажется, что свертка выходит за пределы изображения. Для обработки таких пикселей существует два варианта:

- Дополнить изображение (использовать *padding*)
- Не рассматривать такие пиксели вовсе

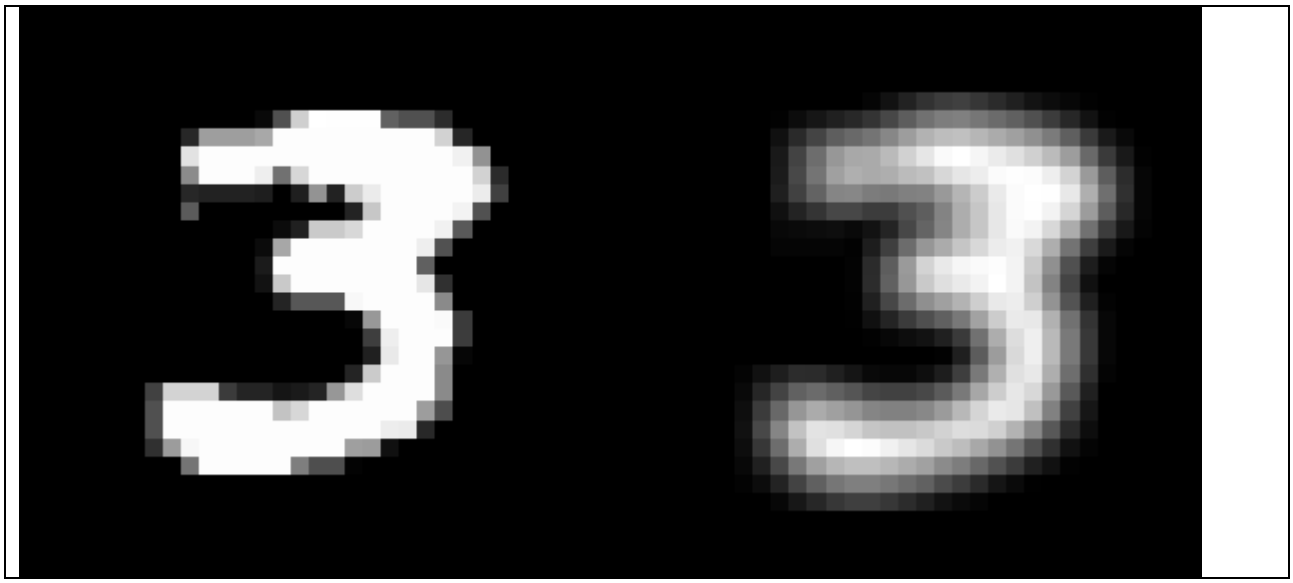
Для простоты, рамках примера, воспользуемся последним вариантом.

Код:

```
def custom_mean_filter(data, filter_size=3):
    result = np.zeros(data.shape)
    kernel_val = 1 / filter_size ** 2
    d = filter_size // 2
    # итерация по строкам изображения
```

```
    for i in range(data.shape[0]):
        if i - d < 0 or i + d > data.shape[0]:
            continue
        # итерация по столбцам изображения
        for j in range(data.shape[1]):
            if j - d < 0 or j + d > data.shape[1]:
                continue
            # определение границ окна
            left = j - d
            right = j + d
            up = i - d
            down = i + d
            # Вычисление пикселя (i,j) в отфильтрованном изображении
            current_pixel_result = 0.0
            for win_i in range(up, down):
                for win_j in range(left, right):
                    current_pixel_result += kernel_val * data[win_i,
win_j]
            result[i, j] = current_pixel_result
    return result

image = cv2.imread('mnist_3.png', cv2.IMREAD_GRAYSCALE)
filtered_image = custom_mean_filter(image, 5)

f, axarr = plt.subplots(1,2, figsize=(10,10))
plt.figure(figsize=(5,5))
axarr[0].imshow(image, cmap='gray')
axarr[1].imshow(filtered_image, cmap='gray')
plt.show()
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Что произойдёт, если в функцию *custom_mean_filter* подать параметр *filter_size*, являющийся четным числом? Какой пиксель свертки станет якорем?
2. Оцените вычислительную сложность алгоритма, реализованного в функции *custom_mean_filter*, в терминах $O(n)$.
3. Замерьте время выполнения функции *custom_mean_filter* с помощью магической команды *%timeit*. Насколько вырастет время выполнения для изображения размера 200x200? А для 1000x1000? Для получения изображения нужного размера можно применить функцию **cv2.resize** к mnist_3.png.
4. Реализуйте свой вариант функции *custom_mean_filter* используя возможности библиотеки numpy (не используя при этом **numpy.convolve**). Постарайтесь избавиться от лишних циклов. Сколько времени теперь занимает применение фильтра для изображения размера 1000x1000?

Фильтр с произвольным ядром

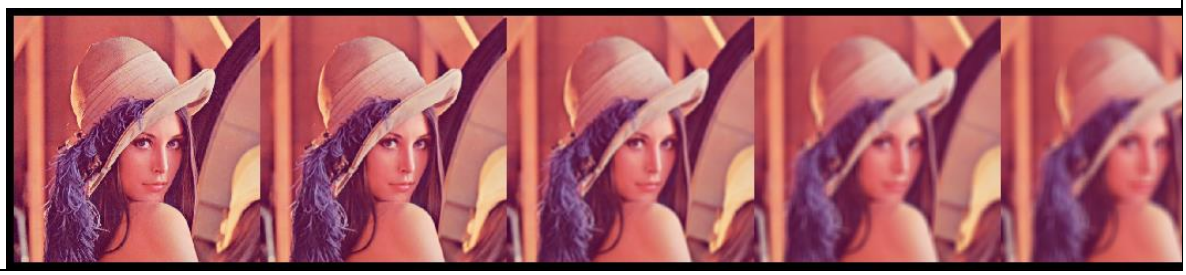
Библиотека OpenCV предоставляет пользователям возможность использовать фильтры с произвольно заданным ядром с помощью функции cv2.filter2D. Ниже представлен пример, в котором тестовое изображение обрабатывается усредняющим фильтром с окнами разного размера. Исходное изображение в цикле конкатенируется с отфильтрованными изображениями, итоговая конкатенация визуализируется. Рассмотрим аргументы функции **cv2.filter2D**:

- *image* - входное изображение, к которому будет применен фильтр;
- *ddepth* - значение, задающее тип обработанного изображения, например, **cv2.CV_8U**, **cv2.CV_16SC3**, **cv2.CV_64F**. В рассматриваемом примере, аргумент *ddepth* имеет значение -1 и это означает что тип выходного изображения будет соответствовать типу входного изображения (**cv2.CV_8UC3**);
- *kernel* - numpy-массив, описывающий ядро свертки.

Код:

```
image = cv2.imread("lenna.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

vis_image = image.copy()
ddepth = -1
for k in [3, 7, 13, 17]:
    kernel = np.ones((k, k)) / (k ** 2)
    filtered_image = cv2.filter2D(image, ddepth, kernel)
    vis_image = np.hstack((vis_image, filtered_image))

plt.figure(figsize=(20, 15))
plt.axis('off')
plt.imshow(vis_image)
plt.show()
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Замерьте время выполнения функции **cv2.filter2D** с помощью магической команды *%timeit*. Насколько оно отличается от времени исполнения вашей реализации фильтра из прошлого пункта для изображения размера 1000x1000? Сделайте предположение о причинах расхождения, в случае если таковое наблюдается.
2. При помощи **cv2.filter2D** можно создавать любые линейные фильтры, в том числе и те, что были рассмотрены в лекции. Попробуйте воссоздать *оператор Шарра* для вычисления частных производных по x и y. Совпадает ли результат работы ваших фильтров с результатом функции **cv2.Scharr**? Учитывайте, что в

OpenCV ядра для вычисления *оператора Шарра* симметрично отражены по сравнению с теми, что рассмотрены в лекции.

3. Помимо рассмотренных аргументов, у функции **cv2.filter2D** есть также:
   - *anchor* - задает позицию якоря. Якорь – это пиксель ядра свертки, относительного которого производится свертка. Почти всегда якорь находится в центре, чему соответствует значение по умолчанию аргумента *anchor* (-1, -1).
   - *delta* - опциональный аргумент, по умолчанию равен 0. Задает константное число, которое прибавляется к каждому пикселю входного изображения. Попробуйте запустить пример выше с разными значениями аргументов *anchor* и *delta*. Как меняется результат?
4. Что происходит с отфильтрованными изображениями при больших значениях аргумента *delta*, например, при 300? Отличаются ли визуализации отфильтрованных изображений при *delta* равном 300, 400, 500? Почему так происходит?

Усредняющие фильтры

OpenCV предоставляет несколько усредняющих фильтров, позволяющих сглаживать изображение, чтобы избавиться от шума. Для того, чтобы рассмотреть работу таких фильтров нужны зашумленные изображения. Для большей наглядности наложим шум на тестовое изображение самостоятельно. Под шумом будем понимать дефект изображения, вносимый фотосенсорами и компонентами устройств, используемых для фотосъемки. Причиной возникновения шума является несовершенство технологий. Синтетический шум – это дефект изображения, внесенный программно.

Равномерный (*uniform*) шум соответствует равномерному распределению. Причина возникновения этого шума в квантовании пикселей изображения на несколько дискретных уровней. Обычно он

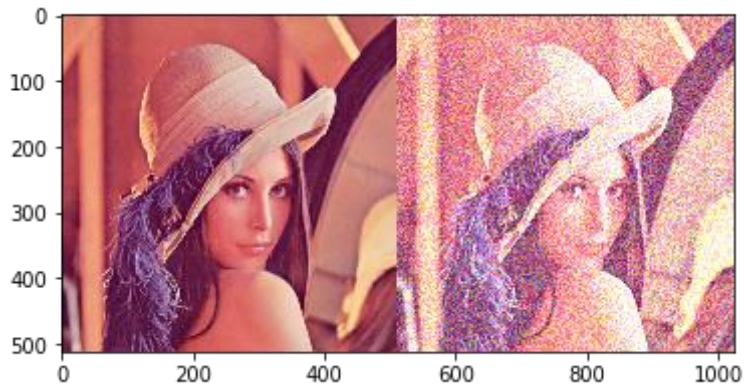
возникает при преобразовании аналоговых данных в цифровую форму и не часто встречается в реальности.

Для симуляции равномерного шума достаточно сгенерировать изображение, заполненное равномерно распределенными пикселями, и добавить его к исходному изображению.

Код:

```
# Генерация равномерного шума. [0, 128] - диапазон значений пикселей
шума
def uniform_noise(image, min_noise_value=0, max_noise_value=128):
    uniform_noise_img = np.random.uniform(min_noise_value,
max_noise_value, image.shape)
    uniform_noise_img = uniform_noise_img.astype(np.uint8)
    noised_image = cv2.add(image,uniform_noise_img)
    return noised_image


image = cv2.imread("lenna.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
uniform_noised_image = uniform_noise(image)

plt.imshow(np.hstack((image, uniform_noised_image)))
```

Результат:



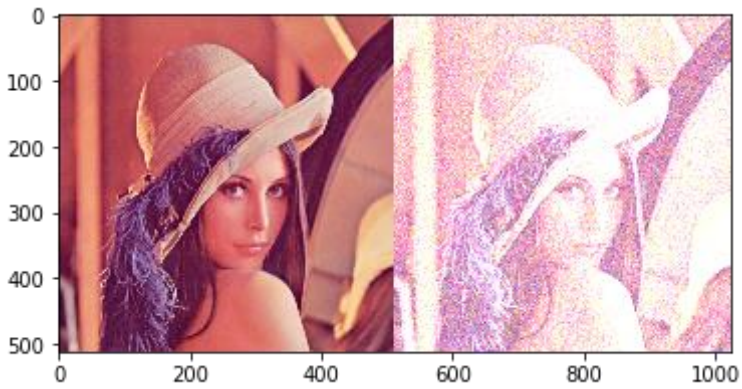
Гауссовский шум – это статистический шум, описываемый нормальным распределением. Гауссовский шум возникает в электронных компонентах (усилители и светоуловители) и обусловлен дискретным характером излучения теплых объектов.

Для симуляции гауссова шума, требуется также создать изображение шума и добавить его к целевому изображению.

Код:

```
# Генерация гауссова шума.
# 128 - математическое ожидание нормального распределения, 30 -
стандартное отклонение нормального распределения
def gaussian_noise(image, mean=128, std=30):
    gaussian_noise_img = np.random.normal(mean, std, image.shape)
    gaussian_noise_img = gaussian_noise_img.astype(np.uint8)
    noised_image = cv2.add(image, gaussian_noise_img)
    return noised_image

image = cv2.imread("lenna.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gaussian_noised_image = gaussian_noise(image)

plt.imshow(np.hstack((image, gaussian_noised_image)))
```

Результат:



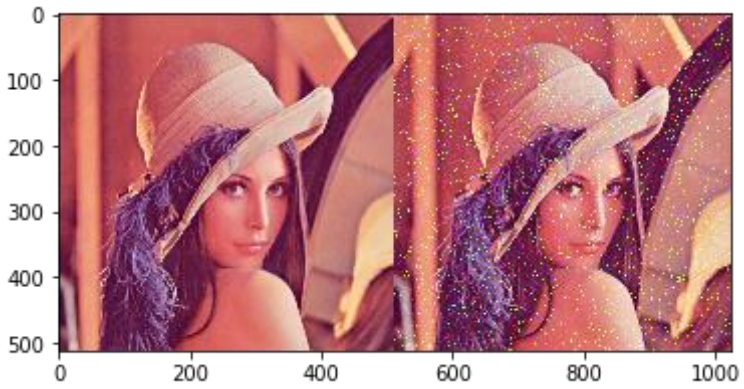
Импульсный шум или шум «соль и перец» (*salt-and-pepper*) – это разряженное появление максимального (255) и минимального (0) значений пикселей в изображении. Такой шум отличается наличием черных пикселей в ярких областях и белых пикселей в темных областях. Этот тип шума возникает из-за резких и внезапных помех в сигнале изображения из-за ошибок аналого-цифрового преобразования или передачи битов.

Чтобы эмулировать импульсный шум, можно создать равномерно зашумленный сигнал и применить бинаризацию по порогу, чтобы создать сетку из черных и белых пикселей. Интенсивность шума можно легко изменить, изменив пороговое значение.

Код:

```
# Генерация импульсного шума.
# threshold - значение порога бинаризации
def imp_noise(image, threshold=128):
    uniform_noise_img = np.random.uniform(0, 255, image.shape)
    uniform_noise_img = uniform_noise_img.astype(np.uint8)
    _, imp_noise_img = cv2.threshold(uniform_noise_img,245, maxval=255,
type=cv2.THRESH_BINARY)
    noised_image = cv2.add(image, imp_noise_img)
    return noised_image


image = cv2.imread("lenna.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
imp_noised_image = imp_noise(image)

plt.imshow(np.hstack((image, imp_noised_image)))
```

Результат:



Теперь рассмотрим функции сглаживания из OpenCV, а именно **cv2.blur**, **cv2.GaussianBlur** и **cv2.medianBlur**.

Функция **cv2.blur** принимает на вход изображение и размер окна. Принцип работы фильтра заключается в вычислении среднего арифметического и аналогичен тому, что рассматривалось в первом пункте работы.

Функция **cv2.GaussianBlur** работает аналогично **cv2.blur**, однако, ядро свертки суммирует пиксели с весами, распределенными по Гауссу. Дополнительно к размеру окна фильтра у **cv2.GaussianBlur** появляется

параметр *sigma*, который задает стандартное отклонение внутри окна фильтра.

Функция **cv2.medianBlur** реализует медианный фильтр. Результатом работы фильтра на конкретном пикселе является пиксель, имеющий медианное значение в окне.

В примере ниже реализована функция *test_smoothing_filter*. Она принимает на вход изображение, зашумляет его равномерным, Гауссовым и импульсным шумом, затем сглаживает фильтром, переданным в аргументе *mode* и визуализирует зашумленное и сглаженное изображение.

Код:

```
def smooth(image, mode):
    if mode == "blur":
        return cv2.blur(image, (5,5))
    elif mode == "GaussianBlur":
        return cv2.GaussianBlur(image, (5,5), 0)
    elif mode == "medianBlur":
        return cv2.medianBlur(imp_noised_image, 5)
    raise Exception(f'{mode} not implemented!')

def test_smoothing_filter(image, mode):
    f, axarr = plt.subplots(1,3, figsize=(10,10))

    uniform_noised_image = uniform_noise(image)
    blurred_image = smooth(uniform_noised_image, mode)
    axarr[0].set_title("Original")
    axarr[0].imshow(np.hstack((uniform_noised_image, blurred_image)))

    gaussian_noised_image = gaussian_noise(image)
    blurred_image = smooth(uniform_noised_image, mode)
    axarr[1].set_title("Gaussian noise")
    axarr[1].imshow(np.hstack((gaussian_noised_image, blurred_image)))

    imp_noised_image = imp_noise(image)
    blurred_image = smooth(uniform_noised_image, mode)
    axarr[2].set_title("Impulse noise")
    axarr[2].imshow(np.hstack((imp_noised_image, blurred_image)))
    plt.show()
```

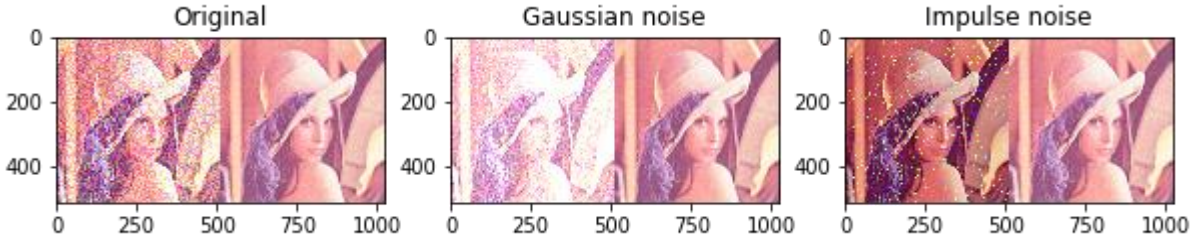
Ниже представлен пример вызова функции *test_smoothing_filter* для теста функций **cv2.blur**, **cv2.GaussianBlur** и **cv2.medianBlur**.

Код:

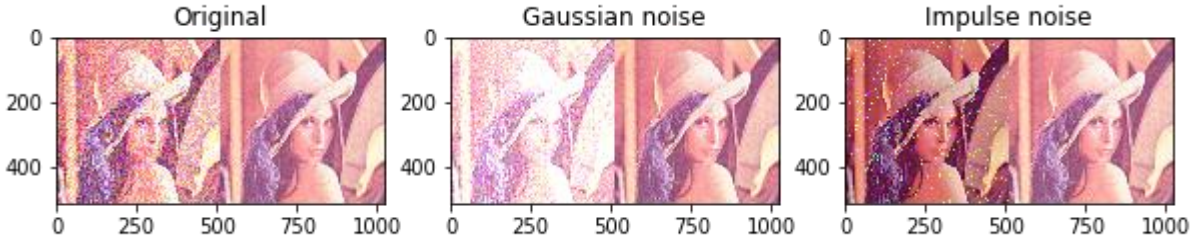
```
image = cv2.imread("lenna.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

print("cv2.blur:")
test_smoothing_filter(image, "blur")
print("==================================")
print("cv2.GaussianBlur:")
test_smoothing_filter(image, "GaussianBlur")
print("==================================")
print("cv2.medianBlur:")
test_smoothing_filter(image, "medianBlur")
```

Результат:

```
cv2.blur:
```




```
==========================================================================

cv2.GaussianBlur:
```



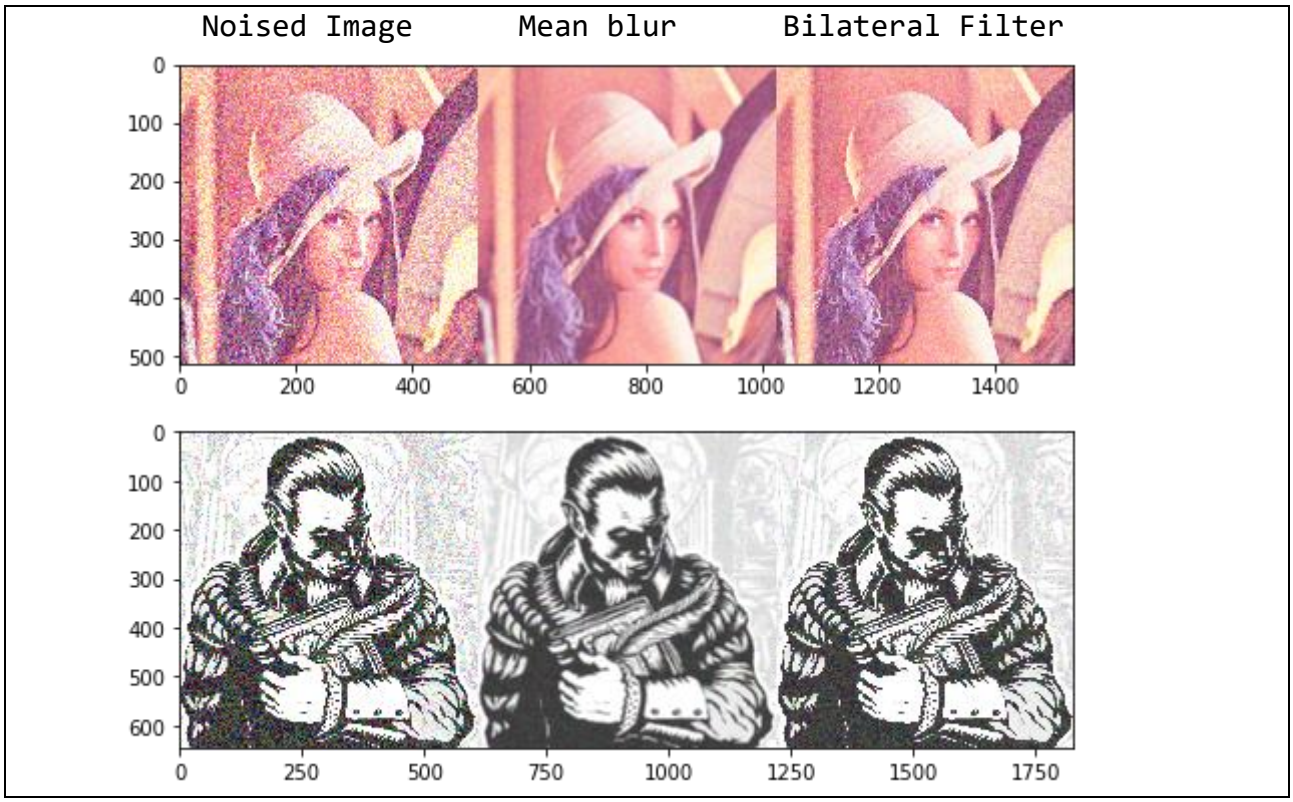

```
==========================================================================

cv2.medianBlur:
```

Помимо сглаживающих фильтров, рассмотренных выше, в OpenCV реализован также билатеральный фильтр (**cv2.bilateralFilter**), который работает значительно медленнее, однако, позволяет сохранить границы на изображении. В примере ниже, целевые изображения зашумляются. К зашумленному изображению применяются билатеральный фильтр и обычный блюр, визуализируются результаты. Как видно из визуализации, обычный блюр сохраняет больше деталей, но размывает границы, а билатеральный фильтр лучше сохраняет границы, но сглаживает детали на однородных областях.

Код:

```
print("\t Noised Image \t Mean blur \t Bilateral Filter")
for file_path in ["lenna.png", "s1r.jpg"]:
    image = cv2.imread(file_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    noised_image = uniform_noise(image, 0, 128)
    bilateral_blurred_image = cv2.bilateralFilter(noised_image.copy(),
9, 75, 75)
    mean_blurred_image = cv2.blur(noised_image.copy(), (7,7))
    plt.figure(figsize=(8,5))
    plt.imshow(np.hstack((noised_image,
mean_blurred_image,bilateral_blurred_image)))
    plt.show()
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Проварьируйте параметры зашумляющих функции (*uniform_noise*, *gassian_noise*, *imp_noise*). При каких значениях параметров удается получить наиболее зашумленное изображение?
2. Какой из фильтров лучше всего справляется с импульсным шумом?
3. Прочитайте справку по **cv2.bilateralFilter**. В чем смысл аргументов *d*, *sigmaColor* и *sigmaSpace*?
4. Модернизируйте функцию *imp_noise* таким образом, чтобы появился *pepper* (выдавались нулевые пиксели). Изменилось ли качество работы **cv2.medianBlur** на зашумленных изображениях, порожденных новой версией *imp_noise*?

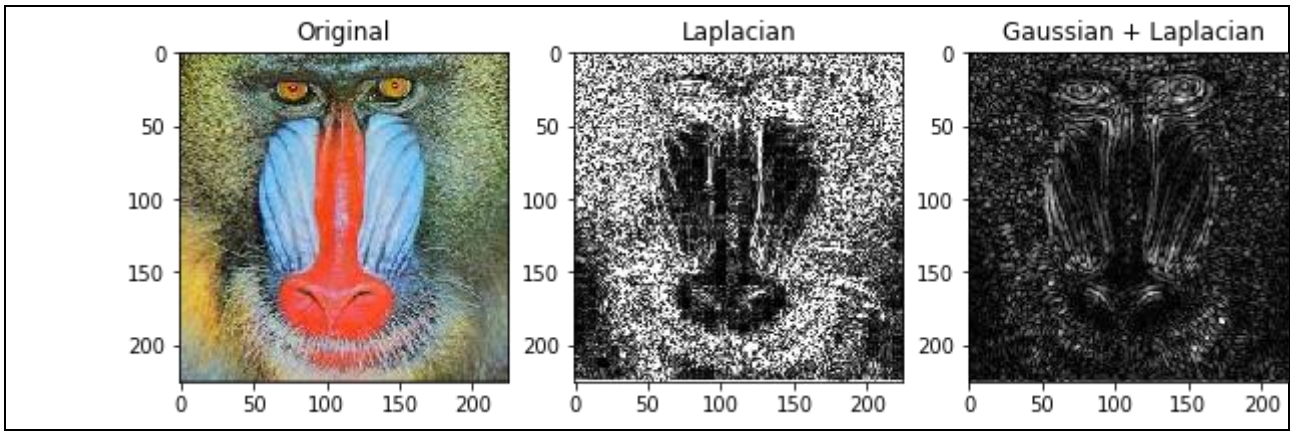
Оператор Лапласа и фильтр Гаусса

Рассмотрим пример применения оператора Лапласа для выделения границ на изображении. Функция **cv2.Laplacian** принимает те же аргументы, что и рассмотренная в лекции функция **cv2.Sobel**. Для улучшения результата перед применением оператора Лапласа можно сгладить изображение.

Код:

```
def laplacian_filter(image, ddepth=cv2.CV_16S, kernel_size=3):
    image_laplacian = cv2.Laplacian(image, ddepth, ksize=kernel_size)
    image_laplacian_u8 = cv2.convertScaleAbs(image_laplacian)
    return image_laplacian_u8

image = cv2.imread("baboon.jpg")
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_blurred = cv2.GaussianBlur(image, (7, 7), 0)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
image_blurred_gray = cv2.cvtColor(image_blurred, cv2.COLOR_BGR2GRAY)

laplacian_map = laplacian_filter(image_gray)
laplacian_map_blurred = laplacian_filter(image_blurred_gray)

f, axarr = plt.subplots(1,3, figsize=(10,10))
axarr[0].set_title("Original")
axarr[0].imshow(image_rgb)
axarr[1].set_title("Laplacian")
axarr[1].imshow(laplacian_map, cmap='gray')
axarr[2].set_title("Gaussian + Laplacian")
axarr[2].imshow(laplacian_map_blurred, cmap='gray')
plt.show()
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Разработчики библиотеки OpenCV утверждают, что функция **cv2.Laplacian** реализована с помощью оператора Собеля, попробуйте убедиться в этом. Используя **cv2.Sobel** найдите частные производные второго порядка для изображения "baboon.jpg" и суммируйте их. Совпадает ли результат с результатом функции *laplacian_filter*?

Настройка параметров детектора Canny при помощи интерактивного виджета

Детектор Canny реализован функцией **cv2.Canny** и принимает на вход изображение в оттенках серого, а также минимальный и максимальный пороги фильтрации. При этом пороги фильтрации рекомендуется выставлять в соотношении от 2:1 до 3:1.

Наличие двух порогов позволяет достаточно качественно подстроить алгоритм к конкретному изображению. Однако зачастую, чтобы добиться хорошего результата, нужно перебрать несколько вариантов. Поэтому, удобнее перебирать гиперпараметры алгоритма в интерактивном режиме. Ниже приведен пример использования библиотеки *ipywidgets* для создания интерактивного виджета, позволяющего динамически без перезапуска ячейки, изменять величину минимального порога, соотношение порогов, а также наличие предварительного сглаживания изображения и параметров фильтра сглаживания.

Для создания интерактивного виджета нужно всего лишь импортировать библиотеку *ipywidgets* и обернуть функцию *canny_setup*, которая принимает на вход гиперпараметры детектора и обрабатывает входное изображение в декоратор *@widgets.interact*. Рассмотрим параметры функции *canny_setup*:

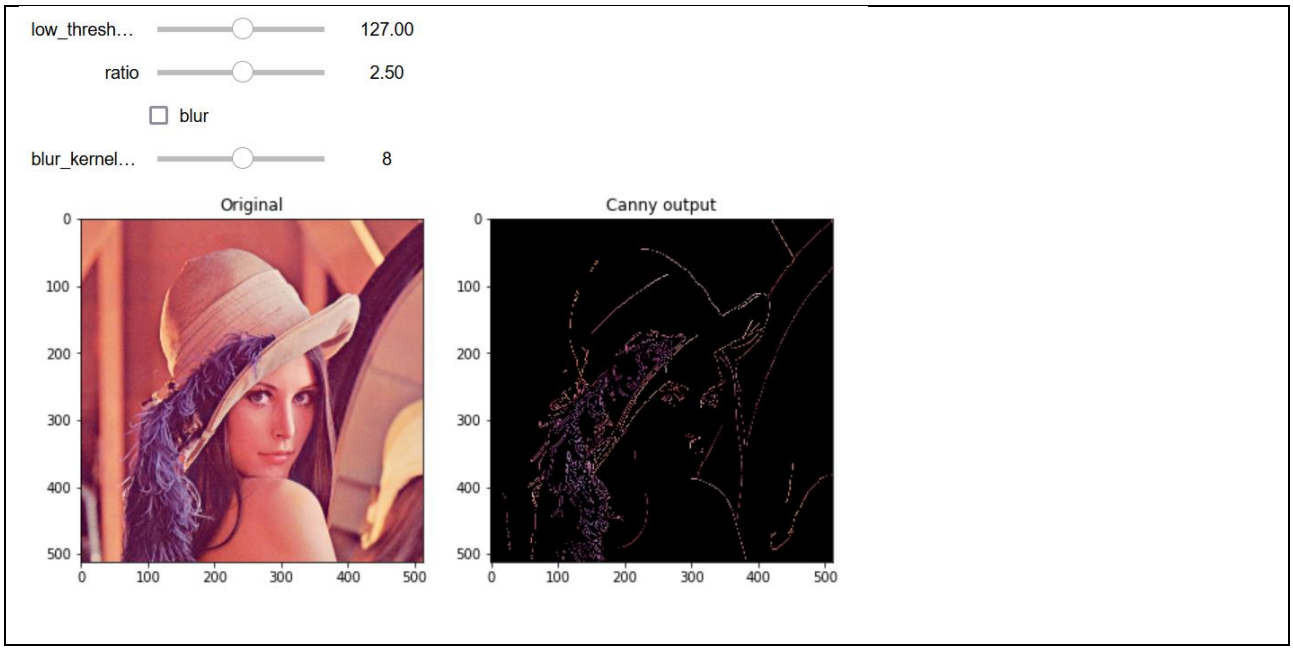
- *low_threshold* = (0.0, 255.0, 1.0) – величина нижнего порога фильтрации, может варьироваться о 0 до 255, изменяется с шагом 1. На отрисованном виджете. Те аргументы, которые определены подобным трехэлементным кортежем будут отображаться ползунком.
- *ratio* = (2.0, 3.0, 0.1) - множитель для верхнего порога фильтрации, может варьироваться о 2.0 до 3.0, изменяется с шагом 0.1
- *blur* = *False* – параметр, регламентирующий применения фильтра сглаживания перед детектором Canny. Параметры, заданные булевым значением в виджете, будут отображаться чекбоксом.
- *blur_ks* = (1, 15) - размер ядра свертки фильтра сглаживания, варьируется от 1 до 15, также будет отображаться ползунком

Для наглядности, найденные границы превращаются в маску, которая накладывается на исходное изображение таким образом, что ненулевое значение имеют лишь пиксели, соответствующие найденным границам.

Код:

```
import ipywidgets as widgets

image = cv2.imread("lenna.png")
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
image_vis = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

@widgets.interact()
def canny_setup(low_threshold=(0.0, 255.0, 1.0), ratio=(2.0, 3.0, 0.1),
blur=False, blur_ks=(1, 15)):
    local_image_gray = image_gray
    if blur:
        local_image_gray = cv2.blur(local_image_gray,
(blur_ks,blur_ks))
    detected_edges = cv2.Canny(local_image_gray, low_threshold,
low_threshold*ratio)
    mask = detected_edges != 0
    dst = image_vis * (mask[:,:,None].astype(image_vis.dtype))
    f, axarr = plt.subplots(1,2, figsize=(10,10))
    axarr[0].set_title("Original")
    axarr[0].imshow(image_vis)
    axarr[1].set_title("Canny output")
    axarr[1].imshow(dst)
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Поэкспериментируйте с виджетом и с графических элементов проварьируйте значения *low_threshold*, *ratio* и параметры блюра, чтобы выделить по возможности только самые главные контуры изображения.
2. Прочитайте в документации OpenCV про аргументы *apertureSize* и *L2gradient* функции **cv2.Canny**. Добавьте их в интерактивный виджет. Насколько сильно эти аргументы влияют на работу детектора?
3. По аналогии с *canny_setup* реализуйте аналогичный виджет, в котором изображения сглаживается фильтром, после чего к нему применяется **cv2.Laplacian** и производится подавление немаксимумов (NMS). Сильно ли отличаются результаты такого детектора границ от результатов Canny? Чем это обусловлено?

cv2.findContours и другие функции для работы с контурами

После нахождения границ на изображении, например, с помощью **cv2.Canny**, может возникнуть необходимость каким-то образом

обработать найденные границы. В этом случае могут помочь контуры. В терминах OpenCV контур – это кривая, соединяющая точки вдоль границы. Ниже представлен простой пример, в рамках которого на изображении выделяются границы с помощью Canny, затем к карте границ применяется функция **cv2.findContours**, которая выделяет отдельные контуры. Найденные контуры визуализируются с помощью функции **cv2.drawContours**.

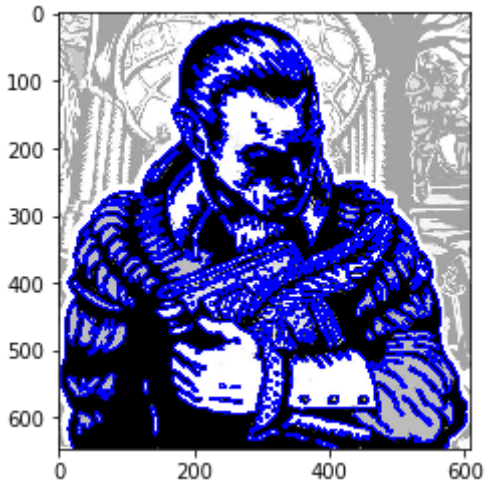
Рассмотрим подробнее функцию **cv2.findContours**:

- входное изображение, передаваемое первым аргументом, желательно должно быть бинарным. В данном случае, используется карта границ, имеющая значения 0 и 255;
- второй аргумент функции – *mode*, описывает режим восстановления контуров. Для контуров в OpenCV определено понятие иерархии (*hierarchy*), которая описывает как контуры вложены друг в друга. В данном случае, используется режим cv2.RETR_EXTERNAL, в котором выделяются лишь внешние контуры;
- третий аргумент функции – *method*, описывает режим аппроксимации контуров, т.е. как определяются точки, описывающие контур. В данном случае используется режим аппроксимации cv2.CHAIN_APPROX_NONE, при котором сохраняются все точки контура;
- функция возвращает список контуров (одномерных numpy-массивов, содержащих точки контуров) и numpy-массив *hierarchy*, описывающий вложенность контуров. В данном примере не используется.

Код:

```
image = cv2.imread("s1r.jpg")
image_vis = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

image_gray = cv2.blur(image_gray, (5,5))
detected_edges = cv2.Canny(image_gray, 130, 260)
contours, hierarchy = cv2.findContours(detected_edges,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
cv2.drawContours(image_vis, contours, -1, (0,0,255), thickness = 2)

plt.imshow(image_vis)
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Прочитайте в документации OpenCV про аргументы значения аргументов *mode* и *method*. Как разные значения этих аргументов влияют на результат в рамках рассмотренного примера?

Детектор Ши-Томаси. Субпиксельная точность.

Рассмотрим пример использования детектора особых точек Ши-Томаси, реализованного функцией *cv2.goodFeaturesToTrack*. Рассмотрим эту функцию подробнее.

- Первым аргументом на вход подается изображение в оттенках серого.
- Аргумент *maxCorners* - задает максимальное количество углов, которые вернет функция. Если углов больше, чем указано в *maxCorners*, то будут возвращены лишь наиболее ярко выраженные.
- Аргумент *qualityLevel* - задает своеобразный порог фильтрации углов по мере отклика *R*. Для этого максимальное найденное на данном сэмпле значение R умножается на *qualityLevel*. Полученное значение и является порогом.
- Аргумент *minDistance* - задает минимально допустимое Евклидово расстояние между углами.
- Функция возвращает numpy-массив формы (N, 1, 2), где N - количество найденных углов.

В приведенном ниже примере для двух изображений находятся и визуализируются углы.

Код:

```
def find_n_draw_corners(grayscale, original):
    corners = cv2.goodFeaturesToTrack(grayscale, maxCorners=100,
qualityLevel=0.01, minDistance=40)
    corners = np.int0(corners)
    for i in corners:
        x,y = i.ravel()
        cv2.circle(original, (x,y), 5, 255, 2)
    return image

f, axarr = plt.subplots(1,2, figsize=(10, 10))
for i, img_path in enumerate(["chess.jpg", "mine_zombie.jpg"]):
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    find_n_draw_corners(gray, img)
    axarr[i].imshow(img)
    axarr[i].axis('off')
plt.show()
```

Результат:

Задания и вопросы для самостоятельной работы:

1. Варьируя параметры функции **cv2.goodFeaturesToTrack**, попытайтесь найти как можно больше углов на изображении mine_zombie.jpg.
2. Прочитайте в документации OpenCV про значения аргументов *mask* и *blockSize* функции **cv2.goodFeaturesToTrack**. Как разные значения этих аргументов влияют на результат в рамках рассмотренного примера?
3. Иногда требуется получить более точную оценку позиции углов на изображении. Такая точность называется субпиксельной и чтобы ее добиться можно воспользоваться функцией **cv2.cornerSubPix**. С помощью официальной документации OpenCV, разберитесь как она работает и добавьте ее в пример выше. Насколько уточненные координаты пикселя в среднем отличаются от найденных **cv2.goodFeaturesToTrack**? Можно ли визуализировать на исходном изображении уточненные координаты?

Дескрипторы. Матчинг дескрипторов.

Рассмотрим пример построения дескрипторов особых точек, а также матчинга по дескрипторам. В терминах задачи поиска по базе изображений, рассмотренной в начале лекции, будем работать с базовым изображением (naturmort_1.jpg) и набором запросных изображений (naturmort_1.jpg, ..., naturmort_5.jpg). На каждом запросном изображений есть предмет, который также имеется на базовом изображении.

Для нахождения особых точек и дескрипторов будем использовать алгоритм **SIFT**. Для работы требуется с помощью функции **cv2.SIFT_create** создать объект, реализующий алгоритм **SIFT**. Затем с помощью метода *detectAndCompute*, вычисляются точки и дескрипторы. Метод *detectAndCompute* принимает на вход изображение в оттенках серого и маску, ограничивающую целевую область изображения (в данном случае маска не определена). Найденные особые точки возвращаются в виде списка объектов **cv2.KeyPoint** (непосредственно координаты хранятся в

поле **pt**). Найденные дескрипторы возвращаются в виде numpy-массива формы (N, 128), где N - число найденных точек (и, соответственно дескрипторов).

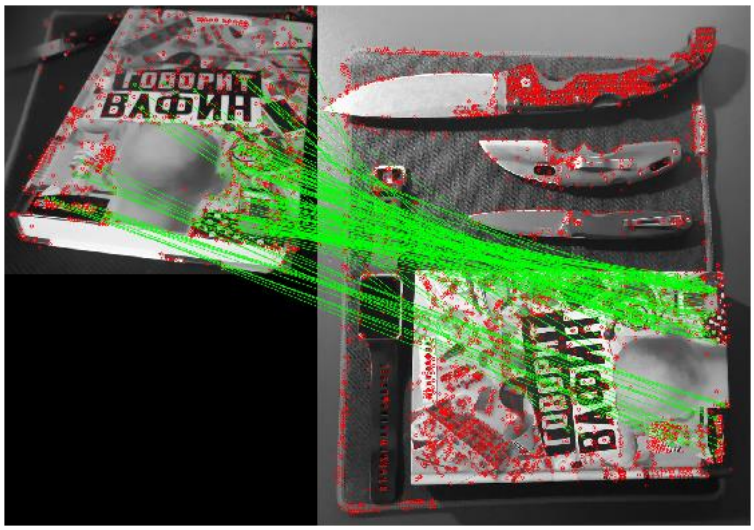
Далее с помощью объекта **cv2.FlannBasedMatcher**, требуется построить соответствие (матчинг) между найденными наборами дескрипторов. Аббревиатура FLANN расшифровывается как *Fast Library for Approximate Nearest Neighbors* – быстрая библиотека для аппроксимации ближайших соседей. **FLANN** содержит набор алгоритмов, оптимизированных для быстрого поиска ближайших соседей. Сам алгоритм k ближайших соседей (*k-Nearest Neighbours*, kNN) будет более подробно рассмотрен в дальнейших занятиях курса. Для создания объекта **cv2.FlannBasedMatcher** требуется определить два словаря. В словаре index_params задается используемый алгоритм матчинга (FLANN_INDEX_KDTREE) и его параметры (*trees* = 5). В словаре *search_params* задается количество проверок, проводимых при построении матчинга (в данном случае, 200). Чем выше это число, тем надежнее результат, однако и тем больше времени требуется на построение матчинга. После создания объекта **cv2.FlannBasedMatcher**, вызывается метод *knnMatch*, принимающий на вход два набор дескрипторов (*queryDescriptors*, *trainDescriptors*) и параметр k, задающий число рассматриваемых соседей для запросного дескриптора. В качестве ответа, метод возвращает список длины, соответствующей параметру *queryDescriptors*, в котором лежат списки, в которых лежит по два объекта класса **cv2.DMatch**. То есть для каждого дескриптора из набора, переданного в метод первым, возвращается информация по лучшему совпадению из второго набора.

После этого построенный матчинг фильтруется, результаты записываются в список matchesMask и используются в итоговой визуализации, получаемой с помощью функции **cv2.drawMatchesKnn**. Рассматриваемый пример построит 5 визуализации (для каждого из запросных изображений), на которых будут показаны ключевые точки запросного и базового изображения, а также матчинг между особыми точками.

Код:

```
for k in range(1, 6):
    img1 = cv2.imread(f'naturmort_2_{k}.jpg', cv2.IMREAD_GRAYSCALE)  #
queryImage
    img2 = cv2.imread('naturmort_1.jpg', cv2.IMREAD_GRAYSCALE)  #
trainImage

    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=200)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)

    matchesMask = [[0, 0] for i in range(len(matches))]
    # ratio test as per Lowe's paper
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.7 * n.distance:
            matchesMask[i] = [1, 0]
    draw_params = dict(matchColor=(0, 255, 0),
                       singlePointColor=(255, 0, 0),
                       matchesMask=matchesMask,
                       flags=cv2.DrawMatchesFlags_DEFAULT)
    img3 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, matches, None,
**draw_params)
    plt.figure(figsize=(10, 10))
    plt.axis('off')
    plt.imshow(img3, ), plt.show()
```

Результат:



Задания и вопросы для самостоятельной работы:

1. Как можно видеть на визуализации, текущий пример отрабатывает хорошо не на всех запросных изображениях (в частности на naturmort_2_3.jpg и naturmort_2_4.jpg). Чем это можно объяснить?
2. Попробуйте поварьировать параметры *checks* и величину порога при фильтрации матчинга (исходно 0.7). Помогло ли это улучшить матчинг на проблемных запросных изображениях?
3. Попробуйте, руководствуясь документацией OpenCV, заменить **SIFT** на другие алгоритмы (**SURF**, **BRIEF**, **ORB**). Есть ли принципиальные отличия в результате?