

Федеральное государственное автономное образовательное учреждение высшего образования  
«Национальный исследовательский университет ИТМО»

Отчет по лабораторной работе №3  
«Фильтрация и выделение контуров»  
по дисциплине «Техническое зрение»

Выполнил: студент гр. Р3338,  
Кирбаба Д.Д.  
Преподаватель: Шаветов С.В.,  
канд. техн. наук, доцент ФСУ и Р

Санкт-Петербург, 2022

## Цель работы

Освоение основных способов фильтрации изображений от шумов и выделения контуров.

## Теоретическое обоснование применяемых методов

Для успешного анализа изображения часто бывает необходимо удалить некоторые искажения. Например, в прошлой работе была проведена коррекция дисторсии, а сейчас будем исследовать следующий тип искажения – шум. Шумы подразделяются на следующие типы: импульсные, аддитивные, мультипликативные, шумы Гаусса, шумы квантования.

Импульсный шум характеризуется тем, что каждый пиксель зашумленного изображения с некоторой вероятностью будет принимать значения из множества {0, 255}.

Аддитивный и мультипликативный шумы имеют схожее определение, они дополняют исходное изображение матрицей значений, имеющей какое-либо распределение:

$$I_{new}(x, y) = I(x, y) + \eta(x, y) - \text{аддитивный шум},$$

$$I_{new}(x, y) = I(x, y) \cdot \eta(x, y) - \text{мультипликативный шум}.$$

Гауссов (нормальный) шум искажает изображение, добавляя к нему интенсивности, распределенные по нормальному закону:

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

Шум квантования имеет следующие черты: в результате наложения появляются артефакты, его почти невозможно устраниТЬ, а также его приближенно можно описать с помощью распределения Пуассона.

Основным методом фильтрации изображения является локальная фильтрация в скользящем окне – когда для вычисления новой интенсивности пикселя берутся во внимание соседние пиксели, с различными весами. В общем виде описывается так:

$$I_{new}(x, y) = \sum_m \sum_n \omega(m, n) \cdot I(x + m, y + n), \quad \omega - \text{маска фильтра}.$$

Фильтрация подразделяется на 2 больших раздела: низкочастотная и высокочастотная.

Низкочастотная фильтрация удаляет мелкие высокочастотные объекты с изображения, не влияет на монотонные области, тем самым сглаживая изображение. Ядро фильтра низкочастотной фильтрации обычно имеет следующие свойства:

- все коэффициенты ядра неотрицательны;
- в сумме коэффициенты дают единицу.

Приведем математическое описание низкочастотных фильтров:

$$I_{new}(x, y) = \frac{1}{m \cdot n} \sum_{i=0}^m \sum_{j=0}^n I(i, j) - \text{арифметический усредняющий},$$

$$I_{new}(x, y) = \left[ \prod_{i=0}^m \prod_{j=0}^n I(i, j) \right]^{\frac{1}{m \cdot n}} - \text{геометрический усредняющий},$$

$$I_{new}(x, y) = \frac{m \cdot n}{\sum_{i=0}^m \sum_{j=0}^n \frac{1}{I(i, j)}} - \text{гармонический усредняющий},$$

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

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} - \text{фильтр Гаусса.}$$

Перечисленные выше фильтры – линейны и хороши для фильтрации помех, распределенных нормально по изображению.

Однако для фильтрации импульсных шумов широко используются нелинейные, например медианные фильтры.

Классический медианный фильтр работает по следующему принципу: из окна произвольного размера берутся пиксели, сортируются и итоговым значением пикселя будет интенсивность пикселя из отсортированного массива с индексом  $i = \frac{N+2}{2}$ .

Опишем некоторые модификации медианного фильтра:

- взвешенный медианный фильтр – в маске используются натуральные коэффициенты, отражающие насколько сильно пиксель будет воздействовать на фильтрацию, при формировании массива пикселей, пиксель с коэффициентом  $n$  будет добавлен в массив  $n$  раз;
- адаптивный медианный фильтр – в процессе вычисления новых интенсивностей пикселей, скользящее окно будет изменять размер в зависимости от результата фильтрации. Таким образом, пока не будет найден пиксель, интенсивность которого, отличается от  $I_{min}$  и  $I_{max}$  будет происходить увеличение окна. Данный фильтр является лучшим вариантом для удаления импульсных шумов, однако очень ресурсоемкий.

Вообще говоря, можно обобщить медианный фильтр и выбирать в качестве результирующего пиксель с любым заданным индексом, такой подход называется ранговой фильтрацией.

Также полезным бывает фильтрация Винера, которая основывается на статистических характеристиках локальной окрестности пикселя:

$$\mu = \frac{1}{n \cdot m} \cdot \sum_{i=0}^n \sum_{j=0}^m I(i, j),$$

$$\sigma^2 = \frac{1}{n \cdot m} \cdot \sum_{i=0}^n \sum_{j=0}^m I^2(i, j) - \mu^2,$$

$$I_{new}(x, y) = \mu + \frac{\sigma^2 - \nu^2}{\sigma^2} (I(x, y) - \mu),$$

$\mu$  – среднее в окрестности,  $\sigma^2$  – дисперсия,  $\nu^2$  – дисперсия шума.

Если низкочастотные фильтры усиливают монотонные области, сглаживая изображение, то высокочастотные фильтры наоборот – повышают резкость изображения, выделяя области с высоким изменением интенсивностей пикселей. Использование высокочастотных фильтров позволяет выделять контуры изображений.

Ядро высокочастотной фильтрации имеет следующие свойства:

- существуют коэффициенты с отрицательными значениями;
- сумма всех коэффициентов равна нулю.

В основе определения перепадов интенсивностей лежит вычисление производной по направлению.

Рассмотрим следующие виды высокочастотных фильтров:

- фильтр Робертса – использует минимальные маски для вычисления производных:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix};$$

- фильтр Превитта – использует две маски размера  $3 \times 3$  для более точного вычисления производной:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix};$$

- фильтр Собела – схож с фильтром Превитта, однако благодаря использованию других коэффициентов имеет более точный результат работы:

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

- фильтр Лапласа – использует аппроксимацию вторых производных:

$$L(I(x, y)) = \frac{\delta^2 I}{\delta x^2} + \frac{\delta^2 I}{\delta y^2} \rightarrow \omega = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

На основе всех теорий для задачи получения контура единичной длины используется алгоритм Кэнни:

1. Сглаживание изображения фильтром Гаусса.
2. Вычисление градиентов пикселей фильтром Собела (направление градиентов округляется с шагом 45 градусов).
3. Подавление немаксимумов градиента (пиксель является краем, если его градиент больше градиента соседних пикселей, иначе является немаксимумом).

4. Выполнение двойной пороговой фильтрации ( $I > T_2$  – пиксель краевой,  $I < T_1$  – пиксель не краевой,  $T_1 < I < T_2$  – пиксель неоднозначен).
5. Уточнение края трассировкой области неоднозначности (если пиксель из области неоднозначности связан по восьмисвязности с краем, то пиксель краевой, иначе пиксель не является краевым).

## Ход выполнения работы

### 1. Типы шумов



*Figure 1: исходное изображение.*

#### 1.1. Импульсный шум

Наложим на изображение импульсный шум с вероятностью появления шума  $p = 0.01$  в каждом пикселе и отношением шума типа «соль» к типу «перец» равным 0.7.

*Listing 1. Наложение импульсного шума на Python.*

```
# Noise parameter
p = 0.01
# Salt vs pepper ratio
s_p_ratio = 0.7
# Generate random numbers
```

```

rng = np.random.default_rng()
vals = rng.random(img.shape)

img_new = np.copy(img)
img_new[vals < p * s_p_ratio] = 255 # Add salt
img_new[np.logical_and(vals >= p * s_p_ratio, vals < p)] = 0 # Add pepper

```



*Figure 2: изображение с импульсным шумом.*

Как видно, шумов типа «соль» в разы больше, чем шумов типа «перец», что и было предписано в программе.

## 1.2. Аддитивный шум

Добавим к изображению аддитивный шум, который будет происходить из экспоненциального распределения с коэффициентом  $\lambda = 5$ .

*Listing 2. Наложение аддитивного шума на Python.*

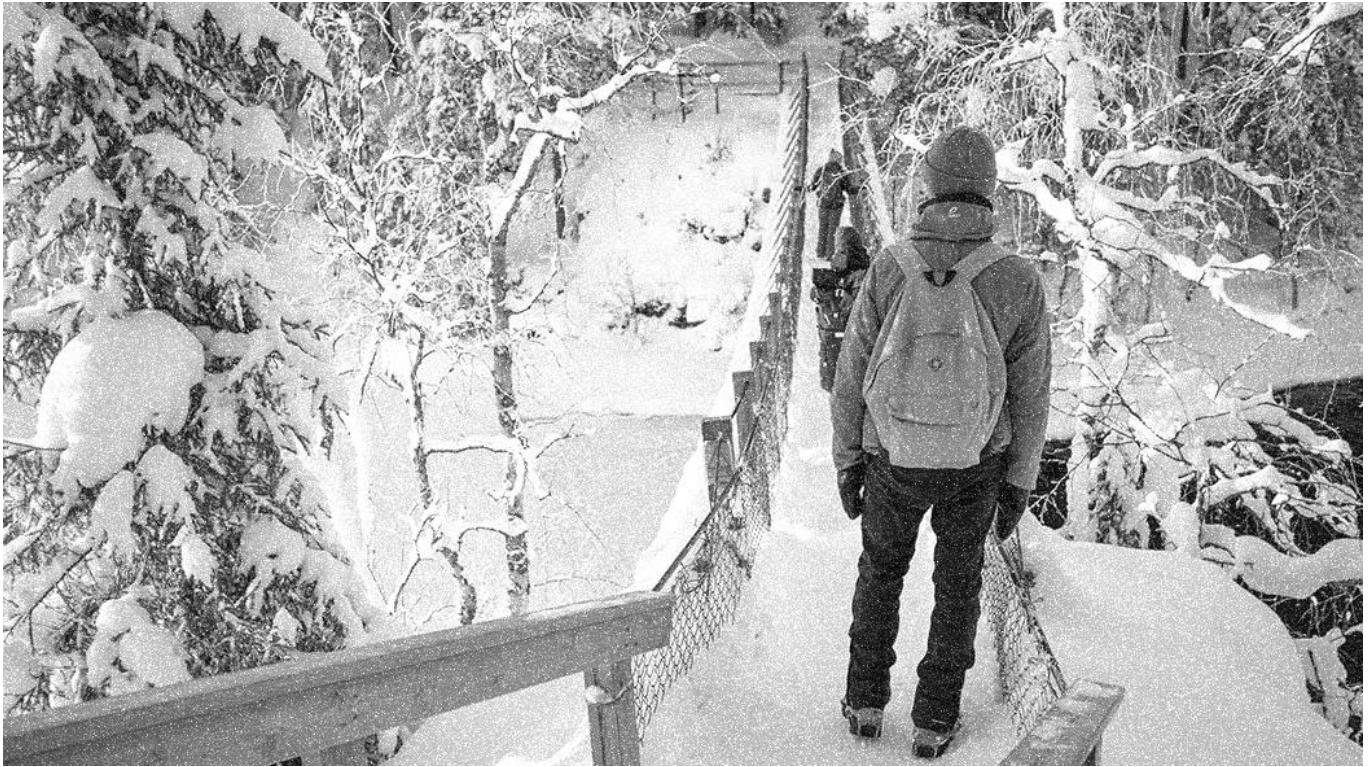
```

# Convert to float and normalize
img_float = img.astype(np.float32) / 255

# Generate random numbers using exp distribution
rng = np.random.default_rng()
exp = rng.exponential(5, img.shape)
exp = exp / np.max(exp) # Normalize

```

```
# Adding noise and convert to uint8
img_new = np.clip((img_float + exp) * 255, 0, 255).astype(np.uint8)
```



*Figure 3: изображение с аддитивным шумом.*

Так как функция плотности вероятности экспоненциального распределения имеет вид  $f(x) = 5e^{-5x}$ , то большое число значений шумов будет близко к нулю, а следовательно изображение будет не сильно зашумлено, что мы и наблюдаем.

### 1.3. Мультипликативный шум

Наложим на изображение мультипликативный шум, который будет происходить из равномерного распределения.

*Listing 3. Наложение мультипликативного шума на изображение на Python.*

```
# Generate random numbers
rng = np.random.default_rng()
uniform = rng.uniform(low=0, high=1, size=img.shape)
# Adding noise
img_new = np.clip(img * uniform, 0, 255).astype(np.uint8)
```



*Figure 4: изображение с мультипликативным шумом.*

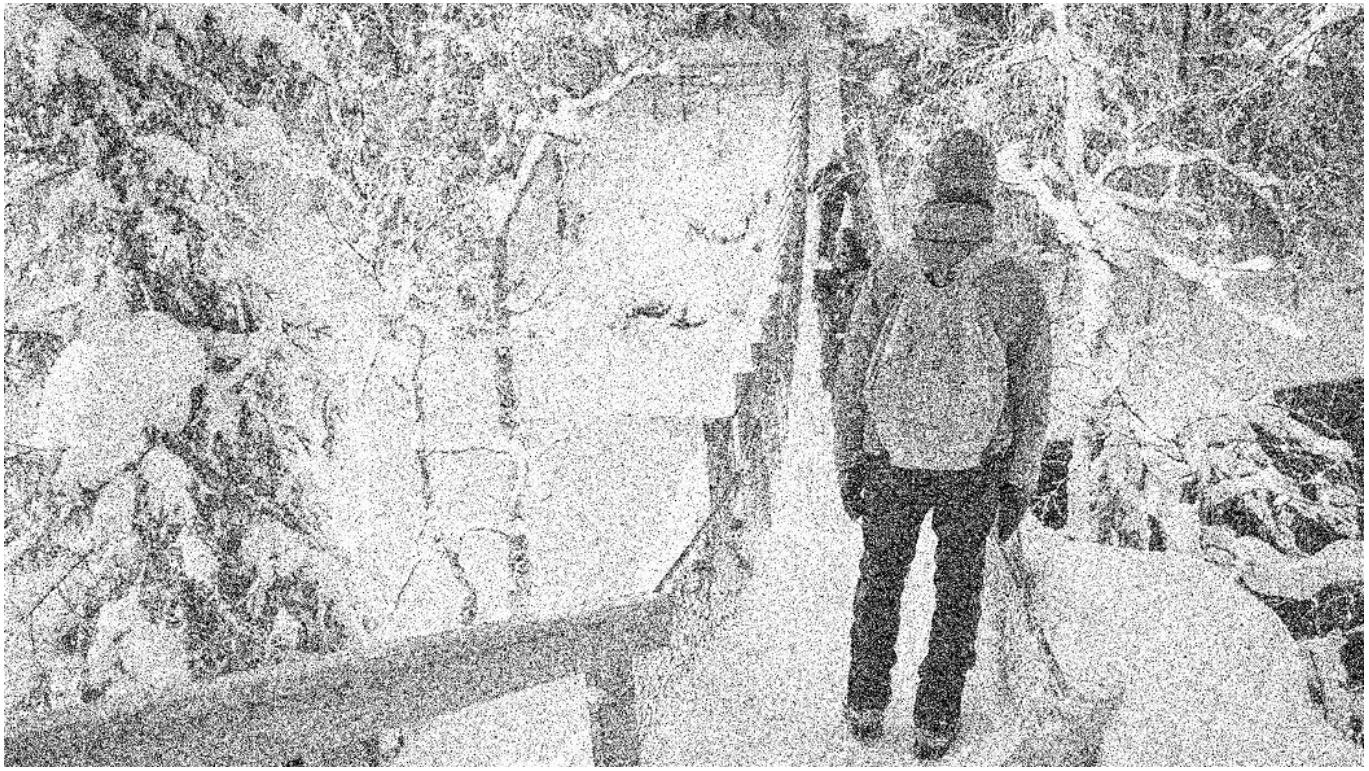
В отличии от аддитивного шума, тут мы наблюдаем более зашумленное изображение. Естественно, что данное изображение сложнее отфильтровать от шумов.

#### 1.4. Гауссов (нормальный) шум

Наложим Гауссов шум с параметрами  $\mu = 0.2$ ,  $\sigma^2 = 0.09$ . Это будет означать, что 67% значений случайной величины будет находиться в диапазоне  $[-0.1, 0.5]$ . При таких параметрах искаженное изображение будет более светлым.

*Listing 4. Наложение нормального шума на изображение на Python.*

```
# Set parameters
var = 0.09
mean = 0.2
# Generate random numbers
rng = np.random.default_rng()
gauss = rng.normal(mean, var ** 0.5, img.shape)
# Adding noise
img_new = np.clip(img + gauss * 255, 0, 255).astype(np.uint8)
```



*Figure 5: изображение с Гауссовым шумом.*

Как мы видим, действительно яркость изображения увеличилась при наложении шума с такими параметрами.

## 1.5. Шум квантования

Добавим к исходному изображению шум квантования, который аппроксимируем распределением Пуассона.

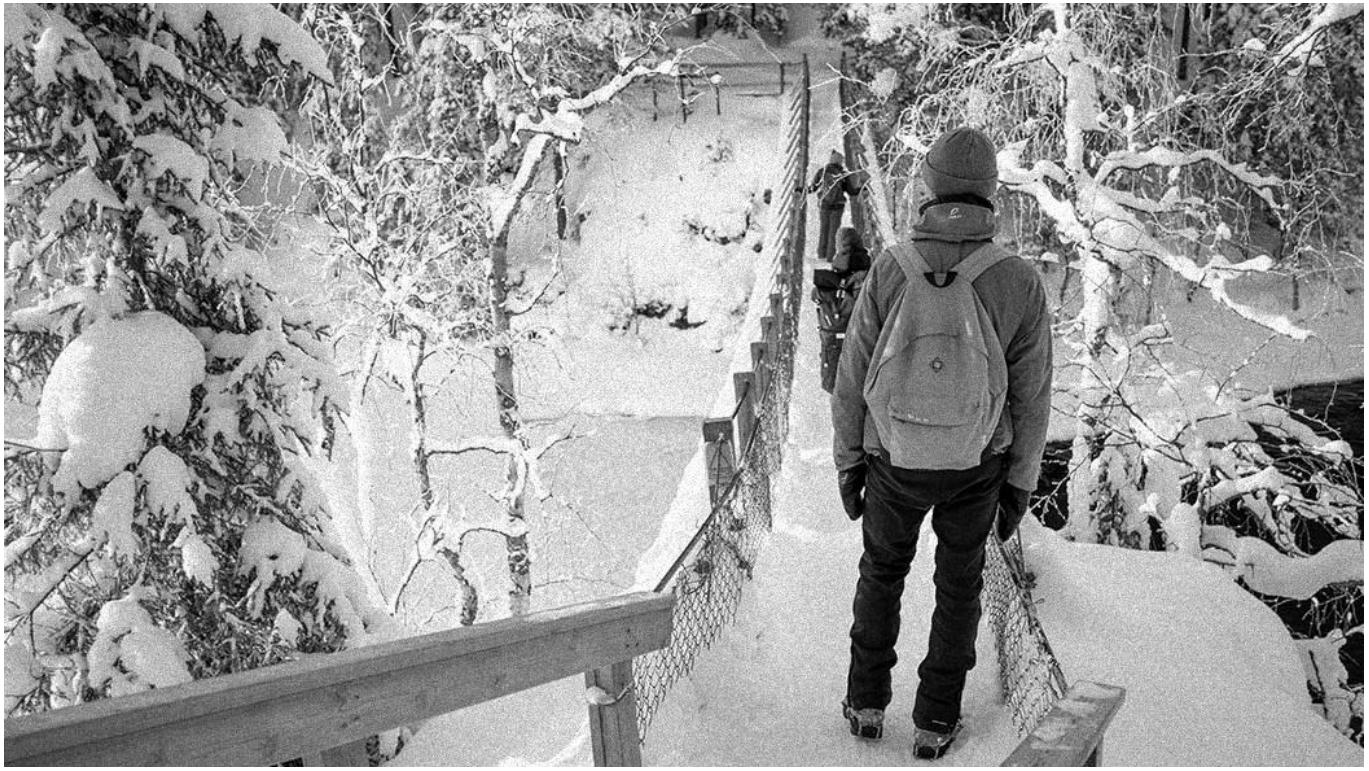
*Listing 5. Наложение шума квантования на изображение.*

```
# Convert to float and normalize
img_float = img.astype(np.float32) / 255

# Generate random numbers
rng = np.random.default_rng()

# Find distribution parameter lambda
labmda = len(np.unique(img_float))
labmda = 2 ** np.ceil(np.log2(labmda))

# Add noise
img_new = (255 * ((rng.poisson(img_float * labmda) /
float(labmda)).clip(0, 1))).astype(np.uint8)
```



*Figure 6: шум квантования.*

Данный шум является не устранимым.

## 2. Низкочастотная фильтрация

Попробуем устранить шумы, используя низкочастотные фильтры.

### 2.1. Фильтр Гаусса

Данный фильтр уменьшает высокочастотные компоненты изображения, сглаживая его.

Будем использовать маску фильтра размеров  $3 \times 3$  с одинаковой дисперсией по осям, равной  $\sigma = 5$ .

*Listing 6. Фильтр Гаусса на Python.*

```
# Set parameters
kernel_shape = np.array([3, 3])
sigmaX = 5

# Apply filtering
img_new = cv.GaussianBlur(img, kernel_shape, sigmaX,
cv.BORDER_REFLECT)
```

В начале применим фильтр к исходному, неискаженному изображению.



*Figure 7: неискаженное изображение после фильтра Гаусса.*

Как и ожидалось, изображение сгладилось, высокочастотные компоненты уменьшились.

Теперь применим фильтр к изображению с импульсным шумом.



*Figure 8: изображение с импульсным шумом после фильтра Гаусса.*

Импульсные шумы стали менее интенсивны, однако сколько-нибудь устранить их не удалось.

Применим Гауссовскую фильтрацию к изображению с аддитивным шумом.



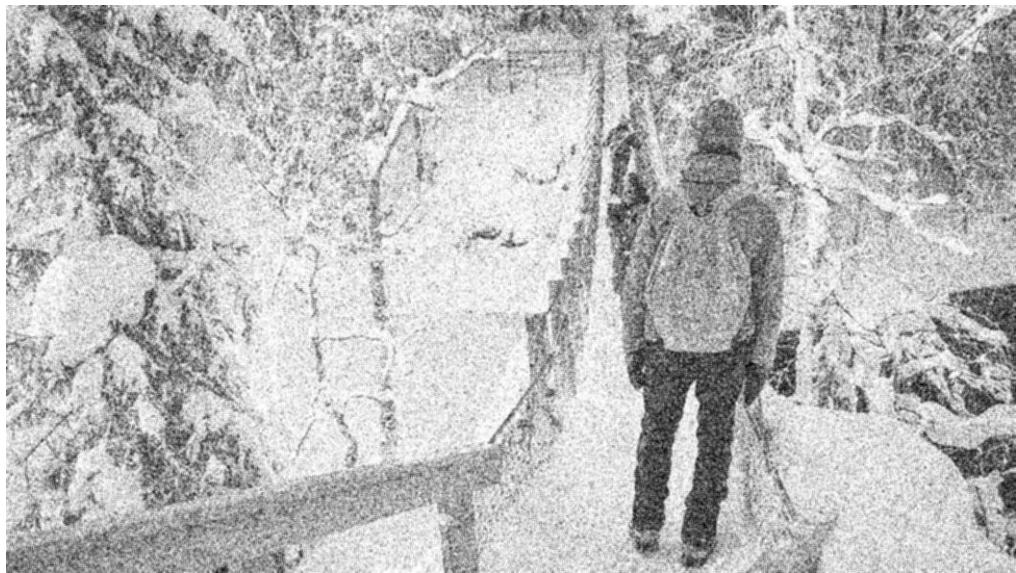
*Figure 9: изображение с аддитивным шумом после фильтра Гаусса.*

Применим Гауссовскую фильтрацию к изображению с мультипликативным шумом.



*Figure 10: изображение с мультипликативным шумом после фильтра Гаусса.*

Отфильтруем изображение с шумом Гаусса.



*Figure 11: изображение с шумом Гаусса после фильтра Гаусса.*

Таким же образом обработает изображение с шумом квантования.



*Figure 12: изображение с шумом квантования после фильтра Гаусса.*

Итого, фильтр Гаусса может успешно использоваться при обработке изображений с целью снижения уровня шума, однако он дает довольно сильное размытие изображения.

## 2.2. Контргармонический усредняющий фильтр

Проведем обработку искаженных в 1 пункте изображений контргармоническими фильтрами с коэффициентами  $Q = \{0.5, -0.5\}$ .

*Listing 7. Контргармонический фильтр на Python.*

```
# Set parameters
Q = -0.5
kernel_shape = np.array([3, 3])
kernel = np.ones(kernel_shape, dtype=np.float32)
div_state = np.seterr(divide='ignore')

# Convert to float and make image with border
img_copy = img.astype(np.float32) / 255
img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2),
int(kernel_shape[1] / 2), cv.BORDER_REFLECT)

# Create four auxiliary arrays
buf1 = np.power(img_copy, Q + 1)
buf2 = np.power(img_copy, Q)
numerator = np.zeros_like(img)
denominator = np.zeros_like(img)

img_new = np.zeros_like(img)
# Filtering
for i in range(kernel_shape[0]):
    for j in range(kernel_shape[1]):
        numerator = numerator + buf1[i:i + n_rows, j:j + n_cols]
        denominator = denominator + buf2[i:i + n_rows, j:j + n_cols]
img_new = np.divide(numerator, denominator)

# Convert back
img_new = np.clip(img_new * 255, 0, 255).astype(np.uint8)
```

Применим фильтр к исходному изображению без шумов.



Figure 13: изображение без шумов после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Применим фильтр к изображению с импульсным шумом.



Figure 14: изображение с импульсным шумом после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Обрабатаем изображение с аддитивным шумом.



Figure 15: изображение с аддитивным шумом после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Применим фильтр к изображению с мультипликативным шумом.



Figure 16: изображение с мультипликативным шумом после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Отфильтруем изображение с Гауссовским шумом.

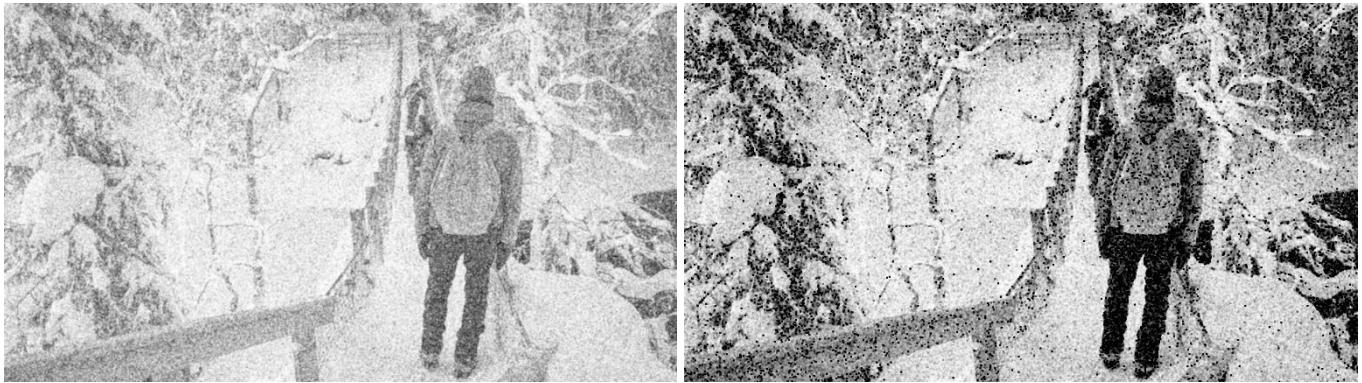


Figure 17: изображение с шумом Гаусса после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Применим контргармонические фильтры к изображению с шумом квантования.



Figure 18: изображение с шумом квантования после контргармонического фильтра  $Q = 0.5$  (слева) и  $Q = -0.5$  (справа).

Итого, контргармонический фильтр – усредняющий фильтр, позволяющий сменой параметра  $Q$  контролировать то, шумы какого типа необходимо устраниить ( $Q > 0$  – «перец»,  $Q < 0$  – «соль»).

Если на изображении имеются шумы одного типа, то использование такого фильтра может дать положительные результаты, однако в иных случаях лучше использовать фильтр Гаусса для снижения уровня любых шумов и уменьшения высокочастотных компонент.

### 3. Нелинейная фильтрация

#### 3.1. Медианная фильтрация

Применим к искаженным изображениям нелинейный медианный фильтр.

*Listing 8. Медианная фильтрация на Python.*

```
# Set parameters
kernel_shape = np.array([3, 3])

# Convert to float and make image with border
img_copy = img.astype(np.float32) / 255
img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2), int(kernel_shape[1] / 2), cv.BORDER_REPLICATE)

# Fill arrays
img_with_areas = np.zeros(img.shape + (np.prod(kernel_shape),),
dtype=np.float32)
for i in range(kernel_shape[0]):
    for j in range(kernel_shape[1]):
        img_with_areas[:, :, i * kernel_shape[1] + j] = img_copy[i:i + n_rows, j:j + n_cols]

# Sort arrays
img_with_areas.sort()

# Choose median pixel
img_new = img_with_areas[:, :, np.prod(kernel_shape) // 2]

# Convert back
img_new = np.clip(255 * img_new, 0, 255).astype(np.uint8)
```

Будем применять два вида медианной фильтрации: с маской  $3 \times 3$  и с маской  $5 \times 5$ .

Применим фильтр к изображению с импульсным шумом.



*Figure 19: изображение с импульсным шумом после медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).*

Видно, что медианный фильтр смог полностью устраниć импульсные шумы, также отметим, что при увеличении размера маски фильтра, изображение становится более размытым.

Применим данный фильтр к изображениям с аддитивным шумом.



Figure 20: изображение с аддитивным шумом после медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Данный фильтр смог довольно неплохо исправить аддитивный шум, особенно при использовании маски  $5 \times 5$ . Однако изображение стало размытым.

Обработаем изображение с мультипликативным шумом.



Figure 21: изображение с мультипликативным шумом после медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Фильтр немного уменьшил шумы.

Применим фильтр для изображения с шумом Гаусса.



Figure 22: изображение с гауссовским шумом после медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Применим фильтр для изображения с шумом квантования.



Figure 23: изображение с шумом квантования после медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Итого, медианный фильтр довольно успешно устраняет импульсные шумы, а также сглаживает изображение. Результат работы схож с усредняющим фильтром, однако для фильтрации импульсных шумов алгоритм медианного фильтра предпочтительнее.

Также плюсом данного фильтра является то, что он не вносит пикселей с новыми интенсивностями в изображение.

### 3.2. Взвешенная медианная фильтрация

Суть данного метода в том, что при формировании массива пикселей, в него добавляется количество пикселей, кратное значению в маске фильтра. Благодаря этому появляется

возможность устанавливать приоритеты влияния пикселей в зависимости от их расположения на результат.

*Listing 9. Взвешенная медианная фильтрация на Python.*

```
# Set parameters
kernel = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
kernel_shape = kernel.shape

# Convert to float and make image with border
img_copy = img.astype(np.float32) / 255
img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2), int(kernel_shape[1] / 2), cv.BORDER_REPLICATE)

# Fill arrays for each kernel item
img_with_areas = np.zeros(img.shape + (np.sum(kernel),), dtype=np.float32)
cur_inx = 0
for i in range(kernel_shape[0]):
    for j in range(kernel_shape[1]):
        # form array with same pixels
        expanded_arr = np.expand_dims(img_copy[i:i + n_rows, j:j + n_cols],
axis=2)
        res = expanded_arr
        for k in range(kernel[i, j] - 1):
            res = np.concatenate((res, expanded_arr), axis=2)
        # filling
        img_with_areas[:, :, cur_inx:(cur_inx + kernel[i, j])] = res
        cur_inx += kernel[i, j]

# Sort arrays
img_with_areas.sort()

# Choose layer with concrete rank
img_new = img_with_areas[:, :, np.prod(kernel_shape) // 2]

# Convert back
img_new = np.clip(255 * img_new, 0, 255).astype(np.uint8)
```

Будем рассматривать два вида фильтров:

$$1. \text{ Ядро } m = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix};$$

$$2. \text{ Ядро } m = \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 5 & 4 & 2 \\ 3 & 5 & 7 & 5 & 3 \\ 2 & 4 & 5 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}.$$

Применим фильтр к изображению с импульсным шумом.



Figure 24: изображение с импульсным шумом после взвешенного медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Так же, как и медианный фильтр выше, взвешенный медианный фильтр устранил импульсные шумы. Однако уровень размытия изображения в разы ниже за счет использования приоритетов пикселей в маске.

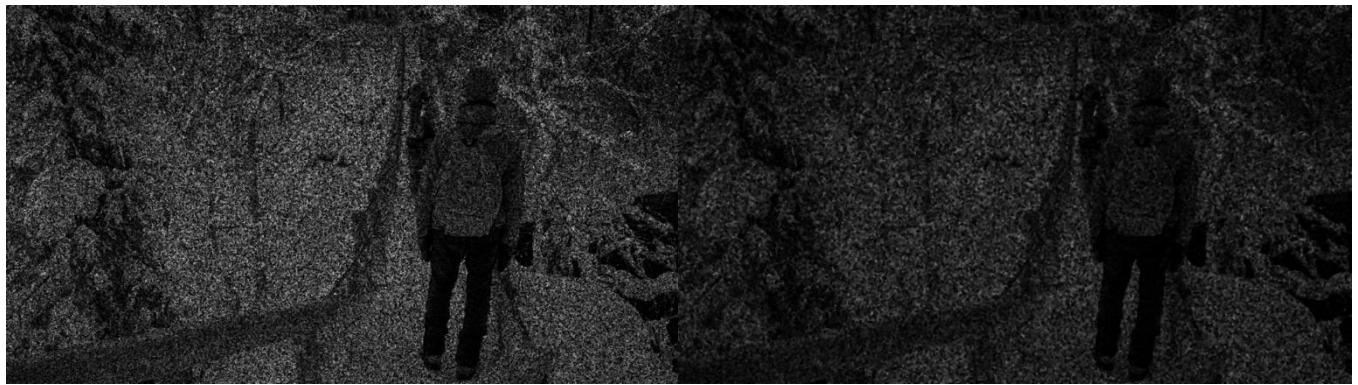
Применим фильтр к изображению с аддитивным шумом.



Figure 25: изображение с аддитивным шумом после взвешенного медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Фильтр с маской  $5 \times 5$  показал хороший результат, почти весь шум устранился.

Обрабатаем изображение с мультипликативным шумом.



*Figure 26: изображение с мультипликативным шумом после взвешенного медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).*

Результат неудовлетворительный, так как данный тип шума трудно устраниТЬ.

Применим фильтр к изображению с шумом Гаусса.



*Figure 27: изображение с мультипликативным шумом после взвешенного медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).*

Обрабатаем изображение с шумом квантования.



*Figure 28: изображение с мультиплекативным шумом после взвешенного медианного фильтра  $3 \times 3$  (слева) и  $5 \times 5$  (справа).*

Итого, данный фильтр имеет результаты схожие с простым медианным, однако за счет различных коэффициентах в ядре фильтра появляется возможность настройки фильтрации, а также при выбранной мной конфигурации размытие изображения намного меньше, чем при фильтрах того же размера в простом медианном фильтре.

### 3.3. Взвешенная ранговая фильтрация

Данный тип фильтрации ещё одна спецификация медианного фильтра, позволяющая более широко применять фильтры. Теперь в качестве результирующего пикселя будет браться не средний пиксель с индексом  $\frac{N+1}{2}$ , а с ранее заданным индексом  $r$ . Часто для инициализации фильтра использует не конкретное значение индекса, а запись через проценты, будем использовать этот вид записи.

Таким образом, заранее зная интенсивности преобладающих шумов в изображении можно выбрать соответствующий параметр  $r$  для лучшей фильтрации.

Так же, как и в пункте выше, рассмотрим два типа фильтров с различными ядрами и параметрами  $r$ :

$$1. \text{ Ядро } m = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}, \quad r = 70\%;$$

$$2. \text{ Ядро } m = \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 5 & 4 & 2 \\ 3 & 5 & 7 & 5 & 3 \\ 2 & 4 & 5 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}, \quad r = 30\%.$$

*Listing 10. Взвешенная ранговая фильтрация на Python.*

```
# Set parameters
kernel = np.array([[1, 2, 3, 2, 1], [2, 4, 5, 4, 2], [3, 5, 7, 5, 3], [2, 4,
5, 4, 2], [1, 2, 3, 2, 1]])
kernel_shape = kernel.shape
r = 0.3

# Convert to float and make image with border
img_copy = img.astype(np.float32) / 255
img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2), int(kernel_shape[1] /
2), cv.BORDER_REPLICATE)

# Fill arrays for each kernel item
img_with_areas = np.zeros(img.shape + (np.sum(kernel),), dtype=np.float32)
cur_inx = 0
for i in range(kernel_shape[0]):
    for j in range(kernel_shape[1]):
        # form array with same pixels
        expanded_arr = np.expand_dims(img_copy[i:i + n_rows, j:j + n_cols],
axis=2)
        res = expanded_arr
        for k in range(kernel[i, j] - 1):
            res = np.concatenate((res, expanded_arr), axis=2)
        # filling
        img_with_areas[:, :, cur_inx:(cur_inx + kernel[i, j])] = res
        cur_inx += kernel[i, j]

# Sort arrays
img_with_areas.sort()

# Choose layer with concrete rank
img_new = img_with_areas[:, :, int(np.sum(kernel) * r)]

# Convert back
img_new = np.clip(255 * img_new, 0, 255).astype(np.uint8)
```

Применим фильтр к изображению с импульсным шумом.



Figure 29: изображение с импульсным шумом после взвешенного рангового фильтра  $3 \times 3$  ( $r = 70$ ) (слева) и  $5 \times 5$  ( $r = 30$ ) (справа).

Как видно, шумы типа «соль» в фильтре с  $r = 70\%$  не полностью ушли, так как мы сделали приоритет на пиксели с высокой интенсивностью. Да и в принципе на данной картинке преобладают высокие интенсивности, поэтому логичнее использовать  $r < 50\%$ , что мы и сделали во втором типе фильтра и результат получился хороший.

Применим фильтр к изображению с аддитивным шумом.



Figure 30: изображение с аддитивным шумом после взвешенного рангового фильтра  $3 \times 3$  ( $r = 70$ ) (слева) и  $5 \times 5$  ( $r = 30$ ) (справа).

Результат данного преобразования описывается выводом выше.

Стоит сказать, что при использовании фильтра  $5 \times 5$  результат намного лучше, чем при использования медианного фильтра  $5 \times 5$  в прошлом пункте, это случилось благодаря правильному подбору параметра  $r = 30\%$ .

Применим взвешенный ранговый фильтр к изображению с мультипликативным шумом.

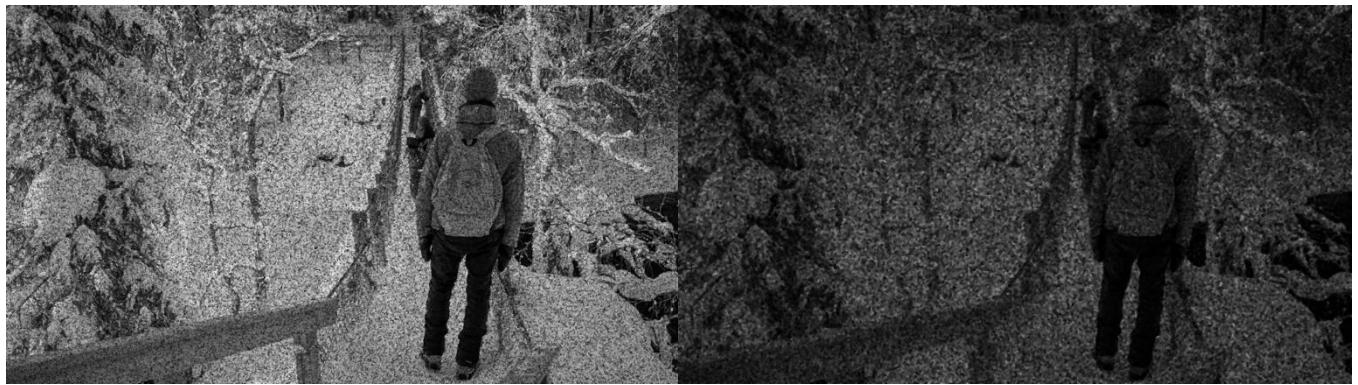


Figure 31: изображение с мультипликативным шумом после взвешенного рангового фильтра  $3 \times 3$  ( $r = 70$ ) (слева) и  $5 \times 5$  ( $r = 30$ ) (справа).

В данном случае лучше работают фильтры с параметром  $r < 50$ , так как много шумов типа «перец».

Применим фильтр к изображению с гауссовским шумом.



Figure 32: изображение с гауссовским шумом после взвешенного рангового фильтра  $3 \times 3$  ( $r = 70$ ) (слева) и  $5 \times 5$  ( $r = 30$ ) (справа).

Обрабатаем изображение с шумом квантования.



Figure 33: изображение с шумом квантования после взвешенного рангового фильтра  $3 \times 3$  ( $r = 70$ ) (слева) и  $5 \times 5$  ( $r = 30$ ) (справа).

Итого, данный тип фильтра является более гибким по сравнению с вышерассмотренными медианными фильтрами. При правильном подборе параметра  $r$ , результаты фильтрации могут давать очень хорошие результаты.

### 3.4. Фильтрация Винера.

Будем использовать два типа фильтрации:

$$1. \ kernel = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$
$$2. \ kernel = \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 5 & 4 & 2 \\ 3 & 5 & 7 & 5 & 3 \\ 2 & 4 & 5 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

Listing 11. Фильтрация Винера на Python.

```
# Set parameters
kernel = np.array([[1, 2, 3, 2, 1], [2, 4, 5, 4, 2], [3, 5, 7, 5, 3], [2, 4, 5, 4, 2], [1, 2, 3, 2, 1]], dtype=np.float32)
kernel_shape = kernel.shape

# Convert to float and make image with border
img_copy = img.astype(np.float32) / 255
img_copy = cv.copyMakeBorder(img_copy, int((kernel_shape[0] - 1) / 2),
int(kernel_shape[0] / 2), int((kernel_shape[1] - 1) / 2), int(kernel_shape[1] / 2), cv.BORDER_REPLICATE)

k_squared = np.power(kernel, 2)
```

```

# Calculate temporary matrices for img ** 2
img_squared = np.power(img_copy, 2)
m = np.zeros(img.shape[:2], np.float32)
q = np.zeros(img.shape[:2], np.float32)
# Calculate variance values
for i in range(kernel_shape[0]):
    for j in range(kernel_shape[1]):
        m = m + kernel[i, j] * img_copy[i:i + n_rows, j:j + n_cols]
        q = q + k_squared[i, j] * img_squared[i:i + n_rows, j:j + n_cols]
m = m / np.sum(kernel)
q = q / np.sum(kernel)
q = q - m * m

# Calculate noise as an average variance
v = np.sum(q) / img.size

# Do filtering
img_new = img_copy[(kernel_shape[0] - 1) // 2:(kernel_shape[0] - 1) // 2 +
n_rows, (kernel_shape[1] - 1) // 2:(kernel_shape[1] - 1) // 2 + n_cols]
img_new = np.where(q < v, m, (img_new - m) * (1 - v / q) + m)

# Convert back
img_new = np.clip(255 * img_new, 0, 255).astype(np.uint8)

```

Применим фильтр Винера к изображению с импульсным шумом.



Figure 34: изображение с импульсным шумом после фильтра Винера 3x3 (слева) и 5x5 (справа).

Как видно, для удаления импульсного шума фильтр Винера не подходит.

Обрабатаем изображение с аддитивным шумом фильтром Винера.



Figure 35: изображение с аддитивным шумом после фильтра Винера  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Применим фильтр Винера к изображению с мультипликативным шумом.



Figure 36: изображение с мультипликативным шумом после фильтра Винера  $3 \times 3$  (слева) и  $5 \times 5$  (справа).

Применим фильтр к изображению с шумом Гаусса.



Figure 37: изображение с гауссовским шумом после фильтра Винера 3x3 (слева) и 5x5 (справа).

Применим фильтр Винера к изображению с шумом квантования.



Figure 38: изображение с шумом квантования после фильтра Винера 3x3 (слева) и 5x5 (справа).

Итого, фильтр Винера основывается на статистических данных из локальных окрестностей. Однако в сравнении с другими фильтрами, результаты его работы хуже остальных.

### 3.5. Адаптивная медианная фильтрация.

В данной модификации фильтра скользящее окно размера  $s \times s$  адаптивно увеличивается в зависимости от результата фильтрации.

Будем использовать два вида фильтров с различными параметрами:

1.  $s_{initial} = 3, s_{max} = 7$ ;
2.  $s_{initial} = 5, s_{max} = 15$ .

*Listing 12. Адаптивная медианная фильтрация на Python.*

```
def calculate_intensity(img_padded_float, i, j, s, s_max):
    # Extract window
    window = img_padded_float[i - (s // 2):i + (s // 2) + 1, j - (s // 2):j +
(s // 2) + 1]
    # Calculate necessary values
    z_min = np.min(window)
    z_max = np.max(window)
    z_med = np.sort(window.reshape(-1))[window.size // 2]

    # Check condition 1
    if z_min < z_med < z_max:
        h, w = window.shape
        z = window[h // 2, w // 2]
        # Check condition 2
        if z_min < z < z_max:
            return z
        else:
            return z_med
    else:
        # Increase size of the window
        s += 2
        if s <= s_max:
            return calculate_intensity(img_padded_float, i, j, s, s_max)
        else:
            return img_padded_float[i, j]

# Set parameters
s_max = 15
s_initial = 3

# Convert to float and make padding
img_padded_float = img.astype(np.float32) / 255
offset = s_max // 2
img_padded_float = cv.copyMakeBorder(img_padded_float, offset, offset,
offset, offset, cv.BORDER_REPLICATE)
filtered_img_padded = np.zeros_like(img_padded_float, dtype=np.float32)

# Go through all pixels
for i in range(offset, n_rows + offset + 1):
    for j in range(offset, n_cols + offset + 1):
        filtered_img_padded[i, j] = calculate_intensity(img_padded_float, i,
j, s_initial, s_max)

filtered_img = filtered_img_padded[offset:-offset, offset:-offset]

# Convert back
filtered_img = np.clip(filtered_img * 255, 0, 255).astype(np.uint8)
```

Применим фильтр к изображению с импульсным шумом.



Figure 39: изображение с импульсным шумом после аддитивного фильтра  $s_{initial} = 3, s_{max} = 7$  (слева) и  $s_{initial} = 5, s_{max} = 15$  (справа).

Как видно, данный фильтр достаточно хорошо смог убрать импульсный шум.

Применим фильтр к изображению с аддитивным шумом.



Figure 40: изображение с аддитивным шумом после аддитивного фильтра  $s_{initial} = 3, s_{max} = 7$  (слева) и  $s_{initial} = 5, s_{max} = 15$  (справа).

Применим фильтр к изображению с мультипликативным шумом.

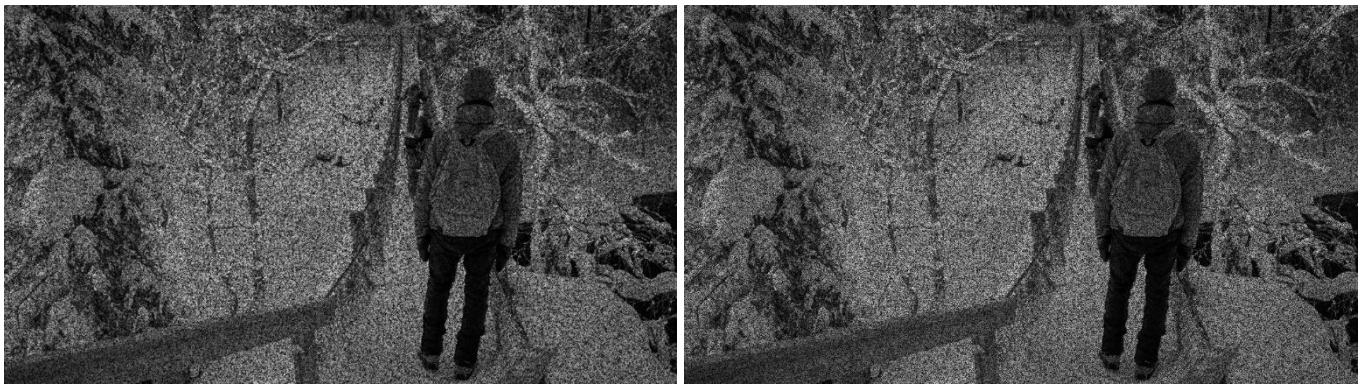


Figure 41: изображение с мультипликативным шумом после адаптивного фильтра  $s_{initial} = 3$ ,  $s_{max} = 7$  (слева) и  $s_{initial} = 5, s_{max} = 15$  (справа).

Применим фильтр к изображению с гауссовским шумом.



Figure 42: изображение с гауссовским шумом после адаптивного фильтра  $s_{initial} = 3, s_{max} = 7$  (слева) и  $s_{initial} = 5, s_{max} = 15$  (справа).

Применим фильтр к изображению с шумом квантования.



Figure 43: изображение с шумом квантования после аддативного фильтра  $s_{initial} = 3, s_{max} = 7$  (слева) и  $s_{initial} = 5, s_{max} = 15$  (справа).

Итого, данный фильтр является очень хорошим способом по уменьшению уровня шума в изображении. Однако данный метод является ресурс затратным и его область применения из-за этого значительно сократилась.

#### 4. Высокочастотная фильтрация

В данном пункте мы будем применять высокочастотные фильтры, позволяющие выделять края изображения на основе использования производной (перепадов интенсивностей).

Затем, использую алгоритм Кэнни, попробуем выделить контуры изображения.



Figure 44: исходное изображение.

## 4.1. Фильтр Робертса

Данный фильтр имеет минимально возможный размер маски  $2 \times 2$  для выделения границы изображения.

*Listing 13. Фильтр Робертса на Python.*

```
# Set parameters
g_x = np.array([[1, -1], [0, 0]])
g_y = np.array([[1, 0], [-1, 0]])

# Convert to float
img = img.astype(np.float32) / 255

# Apply filtering
img_x = cv.filter2D(img, -1, g_x, borderType=cv.BORDER_REFLECT)
img_y = cv.filter2D(img, -1, g_y, borderType=cv.BORDER_REFLECT)

# Calc magnitude
img_res = cv.magnitude(img_x, img_y)

# Invert colors to best readability
img_res = img_res * (-1) + 1

# Convert back
img_res = np.clip(img_res * 255, 0, 255).astype(np.uint8)
```



*Figure 45: границы изображения после фильтра Робертса.*

Данный фильтр быстро выделяет границы изображения, однако из-за одинаковых коэффициентов и маленького размера ядра, он слишком чувствителен к перепадам интенсивностей.

Наблюдая результирующую картинку, невозможно точно разобрать контуры объектов.

## 4.2. Фильтр Превитта

Данный фильтр более точно вычисляет производную, так как тут используется маска размером  $3 \times 3$ .

*Listing 14. Фильтр Превитта на Python.*

```
# Set parameters
g_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
g_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])

# Convert to float
img = img.astype(np.float32) / 255

# Apply filtering
img_x = cv.filter2D(img, -1, g_x, borderType=cv.BORDER_REFLECT)
img_y = cv.filter2D(img, -1, g_y, borderType=cv.BORDER_REFLECT)

# Calc magnitude
img_res = cv.magnitude(img_x, img_y)

# Invert colors to best readability
img_res = img_res * (-1) + 1

# Convert back
img_res = np.clip(img_res * 255, 0, 255).astype(np.uint8)
```

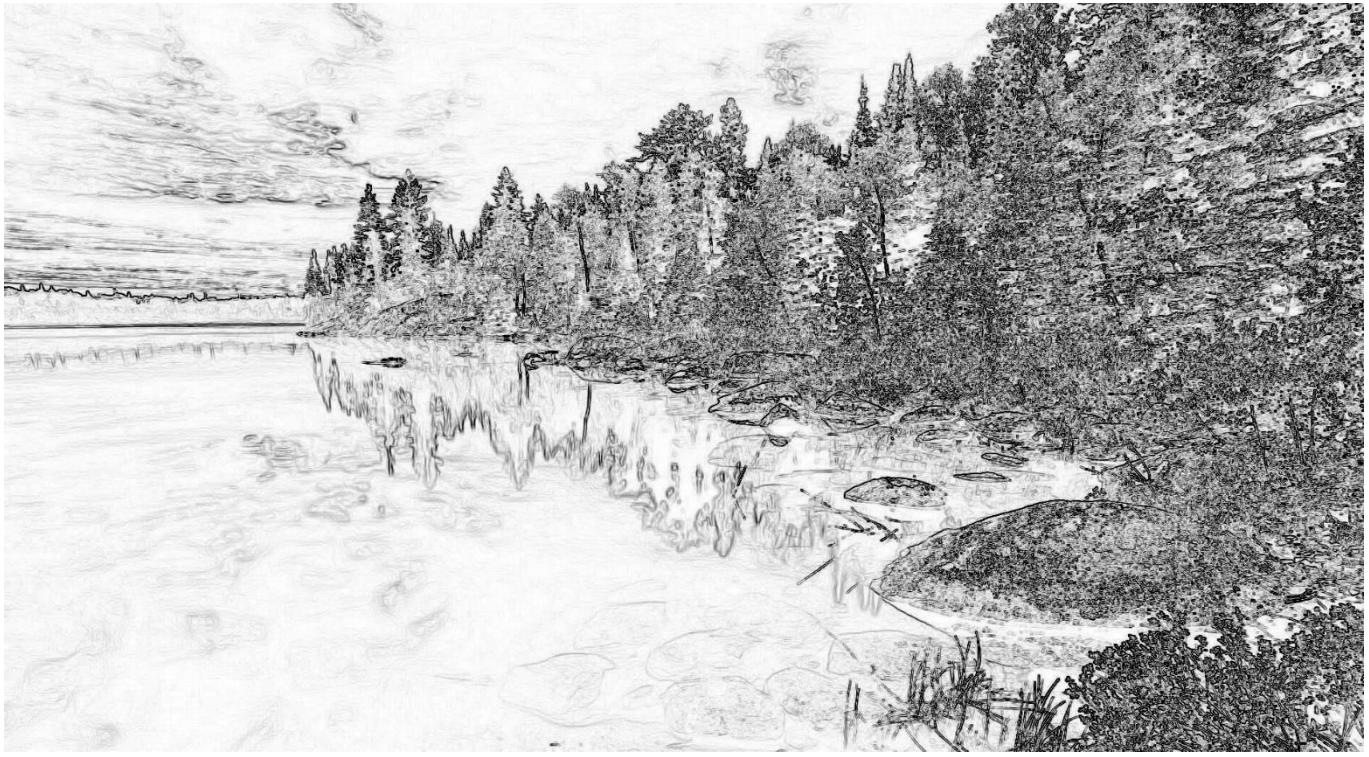


Figure 46: изображение после фильтра Превитта.

Здесь мы более отчетливо наблюдаем границы.

### 4.3. Фильтр Собела

Данный фильтр аналогичен фильтру Превитта, однако используются разные коэффициенты в маске фильтра, благодаря этого контуры выделяются ещё точнее, так как влияние ближайших пикселей больше, чем отдаленных.

Listing 15. Фильтр Собела на Python.

```
# Set parameters
g_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
g_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

# Convert to float
img = img.astype(np.float32) / 255

# Apply filtering
img_x = cv.filter2D(img, -1, g_x, borderType=cv.BORDER_REFLECT)
img_y = cv.filter2D(img, -1, g_y, borderType=cv.BORDER_REFLECT)

# Calc magnitude
img_res = cv.magnitude(img_x, img_y)
```

```

# Invert colors to best readability
img_res = img_res * (-1) + 1

# Convert back
img_res = np.clip(img_res * 255, 0, 255).astype(np.uint8)

```



*Figure 47: изображение после фильтра Собела.*

Границы стали ещё отчетливее.

#### 4.4. Фильтр Лапласа

Данный фильтр использует вторую производную для выделения перепадов интенсивностей, в отличие от фильтров рассмотренных выше.

*Listing 16. Фильтр Лапласа на Python.*

```

# Set parameters
w = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])

# Convert to float
img = img.astype(np.float32) / 255

# Apply filtering

```

```

img_res = cv.filter2D(img, -1, w, borderType=cv.BORDER_REFLECT)

# Invert colors to best readability
img_res = img_res * (-1) + 1

# Convert back
img_res = np.clip(img_res * 255, 0, 255).astype(np.uint8)

```



*Figure 48: изображение после фильтра Лапласа.*

#### 4.5. Алгоритм Кэнни

Данный алгоритм используется для выделения контуров изображения.

*Listing 17. Алгоритм Кэнни на Python.*

```

# Set parameters
t1 = 100
t2 = 200

# Apply Canny algorithm
img_res = cv.Canny(img, t1, t2)

# Invert colors for better readability
img_res = (img_res.astype(np.int16) * (-1) + 255).astype(np.uint8)

```



Figure 49: выделение контуров изображения алгоритмом Кэнни.

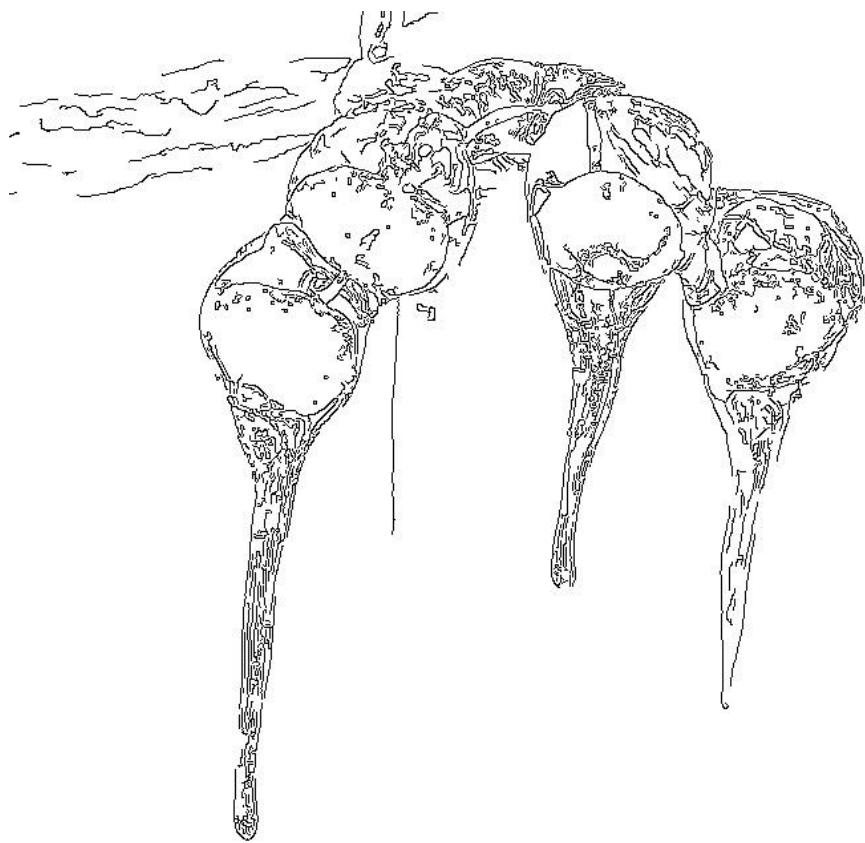
В принципе удалось выделить контуры объектов на изображении. Дело в том, что само изображение имеет сложно разделимые объекты в правой части, которые трудно выделить контуром.

Для примера, выделим контур на более простом изображении.



Figure 50: изображение для выделения контура алгоритмом Кэнни.

Применим алгоритм Кэнни.



*Figure 51: контур изображения, выделенный алгоритмом Кэнни.*

Наблюдаем, что контур объекта выделился хорошо.

Также, изменением параметров алгоритма, можно модифицировать результат (меняя параметры пороговых значений, размера ядра фильтра и т.д.).

## Выводы

В начале работы были исследованы на практике типы шумов на изображении. Шум может появляться в результате преобразования изображения, а также при кодировании-декодировании. Естественно, это безвозвратная потеря информации, однако существует возможность устраниить это искажение.

Далее были приведены практики по устранению шума с изображения, снижения высокочастотных компонент, а также выделения границ изображения. В основе фильтрации лежит операция свертки, также стоит отметить, что при реализации алгоритма необходимо пользоваться векторными операциями, а не посимвольными – так как это существенно увеличит скорость работы.

В заключении, было выполнено выделение контуров с помощью одного из наиболее распространенных и эффективных алгоритмов – Кэнни. Этот алгоритм опирается на изученную ранее теорию, поэтому для хорошей настройки параметров необходимо знать теорию.

В результате проделанной работы были поняты алгоритмы наложения шумов, фильтрации изображений, а также выделения контуров.