



НИУ ИТМО

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

“Фильтрация изображений”

Выполнили:

Александр Иванов, Ф ТЕХ.ЗРЕНИЕ 1.1

Ани Аракелян, ТЕХ.ЗРЕНИЕ 1.1

Никита Братушка, ТЕХ.ЗРЕНИЕ 1.3

Преподаватель:

Шаветов С. В.

Санкт-Петербург, 2024

Содержание

1. Типы шумов	4
1.1. Импульсный шум	4
1.2. Аддитивный шум	5
1.3. Мультипликативный шум	6
1.4. Гауссов (нормальный) шум	7
1.5. Шум квантования	8
2. Фильтрация изображений	9
2.1. Низкочастотная фильтрация	10
2.1.1. Контргармонический усредняющий фильтр	10
2.1.2. Фильтр Гаусса	17
2.2. Нелинейная фильтрация	21
2.2.1. Медианная фильтрация	21
2.2.2. Взвешенная медианная фильтрация	25
2.2.3. Адаптивная медианная фильтрация	28
2.2.4. Ранговая фильтрация	33
2.2.5. Винеровская фильтрация	39
2.3. Высокочастотная фильтрация	42
2.3.1. Фильтр Робертса	42
2.3.2. Фильтр Превитта	42
2.3.3. Фильтр Собела	43
2.3.4. Фильтр Лапласа	44
2.3.5. Алгоритм Кэнни	45

3. Выводы	46
4. Ответы на вопросы	47
A Исходный код	48

1. Типы шумов

Шум – разнообразные искажения на цифровых изображениях, обусловленные разного рода помехами.

В данной лабораторной работе мы рассмотрим наиболее распространенные модели шумов на примере воздействия их на изображения 1.



Рис. 1: Исходное изображение

1.1. Импульсный шум

Зашумленное изображение I описывается следующей системой, причем значение интенсивности пикселя $I(x, y)$ будет изменено на значение $d \in [0, 255]$:

$$\begin{cases} d, & \text{с вероятностью } p, \\ s_{x,y}, & \text{с вероятностью } (1 - p), \end{cases} \quad (1)$$

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



Рис. 2: Результат воздействия импульсного шума на изображение 1

Итак, на изображении 2 отчетливо видны появившиеся белые («соль») и черные («перец») точки, что является характерным для импульсного шума, именно из-за этого он часто называется точечным шумом

1.2. Аддитивный шум

Аддитивный шум описывается следующим выражением:

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

где I_{new} — зашумленное изображение, I — исходное изображение, η — не зависящий от сигнала аддитивный шум с гауссовым или любым другим распределением функции плотности вероятности.



Рис. 3: Результат воздействия аддитивного шума на изображение 1

Наложив аддитивный шум на исходное изображение, мы получили как будто выцветшую картинку, на которой также появилась небольшая зернистость

1.3. Мультиплексивный шум

Мультиплексивный шум описывается следующим выражением:

$$I_{new}(x, y) = I(x, y) \cdot \eta(x, y), \quad (3)$$

Частным случаем мультиплексивного шума является спекл-шум, который мы и рассмотрим



Рис. 4: Результат воздействия спекл-шума на изображение 1

Мы получили изображение на котором можно отчетливо наблюдать светлые пятна, крапинки (спеклы), которые разделены темными участками изображения, что соответственно характерно при наложении спекл-шума

1.4. Гауссов (нормальный) шум

Функция распределения плотности вероятности $p(z)$ случайной величины z описывается следующим выражением:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-\mu)^2}{2\sigma^2}}, \quad (4)$$

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



Рис. 5: Результат воздействия Гауссова шума на изображение 1

При применении Гауссового (нормального) шума к изображению, оно становится более размытым или зернистым. Он добавляет случайные значения пикселям изображения, что приводит к потере деталей и четкости.

1.5. Шум квантования

Приближенно шум квантования можно описать распределением Пуассона. Такой шум не устраняется.



Рис. 6: Результат воздействия шума квантования на изображение 1

Шум на первый взгляд может показаться незаметным, но он есть, при сильном растяжении изображения это видно. На картинке появилась очень мелкая зернистость

2. Фильтрация изображений

Локальным преобразованием называется такое преобразование, при котором для вычисления значения интенсивности каждого пикселя учитываются значения соседних пикселей в некоторой окрестности, называемой *окном*, представляющей собой некоторую матрицу, которую также называют *маской*, *фильтром*, *ядром фильтра*, а сами значения элементов матрицы соответственно *коэффициентами*. Как правило, маска имеет квадратную форму.

Фильтрация изображения I , имеющего размеры $M \times N$, с помощью маски размера $m \times n$

описывается формулой:

$$I_{new}(x, y) = \sum_s \sum_t w(s, t) I(x + s, y + t), \quad (5)$$

где s и t — координаты элементов маски относительно ее центра (в центре $s = t = 0$). Такого рода преобразования называются *линейными*.

Фильтрация в скользящем окне — преобразование, при котором после вычисления нового значения интенсивности пикселя $I_{new}(x, y)$ окно w , в котором описана маска фильтра, сдвигается и вычисляется интенсивность следующего пикселя.

2.1. Низкочастотная фильтрация

Низкочастотные пространственные фильтры ослабляют высокочастотные компоненты (области с сильным изменением интенсивностей) и оставляют низкочастотные компоненты изображения без изменений. Отличительными особенностями ядра низкочастотного фильтра являются: неотрицательные коэффициенты маски и то, что сумма всех коэффициентов равна единице.

2.1.1. Контргармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x, y) = \frac{\sum_{i=0}^m \sum_{j=0}^n I(i, j)^{Q+1}}{\sum_{i=0}^m \sum_{j=0}^n I(i, j)^Q}, \text{ где } Q \text{ — порядок фильтра.} \quad (6)$$

Рассмотрим применение фильтра при $Q > 0$ и $Q < 0$ к различным типам шумов.



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 7: Результат применения контргармонического усредняющего фильтра при значении $Q = -1.85$ к различным типам шумов



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 8: Результат применения контргармонического усредняющего фильтра при значении $Q = -0.85$ к различным типам шумов



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 9: Результат применения контргармонического усредняющего фильтра при значении $Q = -0.5$ к различным типам шумов



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 10: Результат применения контргармонического усредняющего фильтра при значении $Q = 0.5$ к различным типам шумов



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 11: Результат применения контргармонического усредняющего фильтра при значении $Q = 0.85$ к различным типам шумов



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 12: Результат применения контргармонического усредняющего фильтра при значении $Q = 1.85$ к различным типам шумов

При контргармоническом фильтре одновременное удаление белых и черных точек невозможно, такой фильтр является обобщением усредняющих фильтров. Результаты обработки доказывают, что при $Q < 0$ подавляются шумы типа «соль», и заметно, что изображения становятся темнее, а при $Q > 0$ подавляются шумы типа «перец», и картинки будто обесцвечиваются, причем чем больше значения мы берем, тем больше виден эффект.

2.1.2. Фильтр Гаусса

При фильтрации изображений будем использовать двумерный фильтр Гаусса:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-y^2}{2\sigma^2}} \quad (7)$$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 13: Результат применения фильтра Гаусса к различным типам шумов при $n = 3$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 14: Результат применения фильтра Гаусса к различным типам шумов при $n = 7$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 15: Результат применения фильтра Гаусса к различным типам шумов при $n = 11$

Рассмотрев применение фильтра Гаусса при различных n , можно сделать вывод: чем большее значение n мы берём больше получаем эффект размытия на изображениях.

2.2. Нелинейная фильтрация

2.2.1. Медианная фильтрация



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 16: Результат применения медианной фильтрации к различным типам шумов при $k = 3$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 17: Результат применения медианной фильтрации к различным типам шумов при $k = 5$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 18: Результат применения медианной фильтрации к различным типам шумов при $k = 7$

2.2.2. Взвешенная медианная фильтрация



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 19: Результат применения взвешенной медианной фильтрации к различным типам шумов при $k = 3$



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 20: Результат применения взвешенной медианной фильтрации к различным типам шумов при $k = 5$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 21: Результат применения взвешенной медианной фильтрации к различным типам шумов при $k = 7$

2.2.3. Адаптивная медианная фильтрация

Обозначим через z_{min} , z_{max} , z_{med} минимальное, максимальное и медианное значения интенсивностей в окне, $z_{i,j}$ — значение интенсивности пикселя с координатами (i, j) , s_{max} — максимально допустимый размер окна. Алгоритм адаптивной медианной фильтрации состоит из следующих шагов:

1. Вычисление значений $z_{min}, z_{max}, z_{med}, A_1 = z_{med} - z_{min}, A_2 = z_{med} - z_{max}$ пикселя (i, j) в заданном окне.
 - (a) Если $A_1 > 0$ и $A_2 < 0$, перейти на шаг 2. В противном случае увеличить размер окна.
 - (b) Если текущий размер окна $s \leq s_{max}$, повторить шаг 1. В противном случае результат фильтрации равен $z_{i,j}$.
2. Вычисление значений $B_1 = z_{i,j} - z_{min}, B_2 = z_{i,j} - z_{max}$.
 - (a) Если $B_1 > 0$ и $B_2 < 0$, результат фильтрации равен $z_{i,j}$. В противном случае результат фильтрации равен z_{med} .
3. Изменение координат (i, j) .
 - (a) Если не достигнут предел изображения, перейти на шаг 1. В противном случае фильтрация окончена.

Основной идеей является увеличение размера окна до тех пор, пока алгоритм не найдет медианное значение, не являющееся импульсным шумом, или пока не достигнет максимального размера окна. В последнем случае алгоритм вернет величину $z_{i,j}$.



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 22: Результат применения адаптивной медианной фильтрации к различным типам шумов при $k = 3$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 23: Результат применения адаптивной медианной фильтрации к различным типам шумов при $k = 5$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 24: Результат применения адаптивной медианной фильтрации к различным типам шумов при $k = 3$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 25: Результат применения адаптивной медианной фильтрации к различным типам шумов при $k = 9$

2.2.4. Ранговая фильтрация



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 26: Результат применения ранговой фильтрации к различным типам шумов при $k = 3, r = 1$



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 27: Результат применения ранговой фильтрации к различным типам шумов при $k = 3, r = 5$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 28: Результат применения ранговой фильтрации к различным типам шумов при $k = 3, r = 9$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 29: Результат применения ранговой фильтрации к различным типам шумов при $k = 5, r = 1$



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 30: Результат применения ранговой фильтрации к различным типам шумов при $k = 5, r = 14$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультиплексионный шум

Рис. 31: Результат применения ранговой фильтрации к различным типам шумов при $k = 5, r = 25$

2.2.5. Винеровская фильтрация



(а) Исходное изображение



(б) Импульсный шум



(с) Аддитивный шум



(д) Гауссов шум



(е) Шум квантования



(ф) Мультипликативный шум

Рис. 32: Результат применения Винеровской фильтрации к различным типам шумов при $k = 3$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 33: Результат применения Винеровской фильтрации к различным типам шумов при $k = 5$



(a) Исходное изображение



(b) Импульсный шум



(c) Аддитивный шум



(d) Гауссов шум



(e) Шум квантования



(f) Мультипликативный шум

Рис. 34: Результат применения Винеровской фильтрации к различным типам шумов при $k = 7$

2.3. Высокочастотная фильтрация

2.3.1. Фильтр Робертса



Рис. 35: Фильтр Робертса

Рис. 36: Результат применения фильтра Робертса

Наблюдаем, что изображение стало практически черным, но, при этом, границы объектов на изображении различимы.

2.3.2. Фильтр Превитта



Рис. 37: Фильтр Превитта

Рис. 38: Результат применения фильтра Превитта

В отличие от прошлого фильтра, границы объектов заметны отчетливо, но, в то же время, присутствуют артефакты – белые пиксели там, где нет явных границ.

2.3.3. Фильтр Собела



Рис. 39: Фильтр Собела

Рис. 40: Результат применения фильтра Собела

Границы стали еще более выраженным, количество артефактов увеличилось.

2.3.4. Фильтр Лапласа



Рис. 41: Фильтр Лапласа

Рис. 42: Результат применения фильтра Лапласа

Границы выделились наиболее четко, количество артефактов уменьшилось.

2.3.5. Алгоритм Кэнни



Рис. 43: Фильтр Лапласа

Рис. 44: Результат применения алгоритма Кэнни

Наблюдаем наиболее яркие границы объектов.

3. Выводы

В ходе выполнения лабораторной работы были изучены основные способы фильтрации изображений от шумов и выделения контуров. Были применены различные методы фильтрации, которые мы анализировали в процессе работы.

Таким образом, фильтрация изображений и выделение контуров играют важную роль в обработке изображений, позволяя улучшить их качество и провести анализ содержимого более эффективно.

4. Ответы на вопросы

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

A1. Адаптивные фильтры обычно более сложны, чем неадаптивные. Они требуют больше вычислительных ресурсов и времени для корректировки своих параметров в процессе фильтрации.

Несмотря на то, что адаптивные фильтры хорошо уменьшают шум, они иногда могут размывать границы на изображении.

Эффективность адаптивных фильтров во многом зависит от выбора параметров. Выбор подходящих параметров для данного изображения или набора изображений может быть сложной задачей.

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

A2. Q – это порядок фильтра. Контргармонический фильтр является обобщением усредняющих фильтров. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ – в гармонический.

Q3. Какими операторами можно выделить границы на изображении?

A3. Можно выделить границы с использованием дифференциального оператора Робертса.

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

A4. Для того, чтобы избавиться от возможных артефактов при выделении контуров. Низкочастотная фильтрация позволяет убрать мелкие детали и шумы на изображении, тем самым улучшить результат выделения контуров.

A Исходный код

```
def additive_noise(image, mean=4, sigma=0.1):
    """
    Additive noise function.

    Adds log-normal distribution noise to image

    Defined with the following expression:

    NOISY_IMAGE(x,y) = SOURCE_IMAGE(x,y) + NOISE(x,y)
    """

    rng = np.random.default_rng()
    lognormal = rng.lognormal(mean, sigma**0.5, image.shape)
    image_f = image.astype(np.float32)
    image_out = (image_f + lognormal).clip(0, 255).astype(np.uint8)
    return image_out
```

```
def noise_creation(option: int, image, mode: str = 's&p'):
    """Adds noise to image"""
    match option:
        case 1:
            # Impulse noise
            im = ski.util.random_noise(image, mode, amount=0.15)
            image_display(im, 'Impulse noise', 'Noisy_images')
        case 2:
            # Additive noise
            ad = additive_noise(image)
            image_display(ad, 'Additive noise', 'Noisy_images')
        case 3:
            # Gaussian noise
            gs = ski.util.random_noise(image, mode='gaussian', mean=0.01,
var=0.1)
            image_display(gs, 'Gaussian noise', 'Noisy_images')
        case 4:
            # Speckle noise
            sp = ski.util.random_noise(image, mode='speckle', mean=0.01,
var=0.1)
            image_display(sp, 'Speckle noise', 'Noisy_images')
        case 5:
            # Poisson noise
            po = ski.util.random_noise(image, mode='poisson')
            image_display(po, 'Poisson noise', 'Noisy_images')
        case _:
            # displaying source image
            image_display(image, 'Source image', 'Noisy_images')

    return 0
```

```
def contraharmonic(image, m: int, n: int, q):
    """
```

```
Contraharmonic mean filter.
```

```
Filter based on contraharmonic mean:
```

```
FILTERED_IMAGE(x,y)=(SRC_IMAGE(0,0))^(q+1)+...+(SRC_IMAGE(m,n))^(q+1) /  
    (SRC_IMAGE(0,0))^q+...+(SRC_IMAGE(m,n))^q  
"""  
kernel = np.ones((m, n), dtype=np.float32)  
num = np.power(image, q+1) # numerator  
den = np.power(image, q) # denominator  
filtered_image = cv.filter2D(src=num, ddepth=-1, kernel=kernel) /  
    cv.filter2D(src=den, ddepth=-1, kernel=kernel)  
return filtered_image.clip(0, 255).astype(np.uint8)
```

```
def rang_filter(src_image, k, rank):  
    """Rang filtering"""  
    # Filter parameters  
    k_size = (k, k)  
    kernel = np.ones(k_size, dtype=np.float32)  
    rows, cols = src_image.shape[0:2]  
    # Convert to float  
    # and make image with border  
    if src_image.dtype == np.uint8:  
        copied_image = src_image.astype(np.float32) / 255  
    else:  
        copied_image = src_image  
    copied_image = cv.copyMakeBorder(copied_image, int((k_size[0] - 1) / 2),  
        int(k_size[0] / 2),  
        int((k_size[1] - 1) / 2), int(k_size[1]  
        / 2), cv.BORDER_REPLICATE)  
    # Fill arrays for each kernel item  
    I_layers = np.zeros(src_image.shape + (k_size[0] * k_size[1],),  
        dtype=np.float32)  
    if src_image.ndim == 2:  
        for i in range(k_size[0]):  
            for j in range(k_size[1]):  
                I_layers[:, :, i * k_size[1] + j] = kernel[i, j] *  
                    copied_image[i:i + rows, j:j + cols]  
    else:  
        for i in range(k_size[0]):  
            for j in range(k_size[1]):  
                I_layers[:, :, :, i * k_size[1] + j] = kernel[i, j] *  
                    copied_image[i:i + rows, j:j + cols, :]  
    # Sort arrays  
    I_layers.sort()  
    # Choose layer with rank  
    if src_image.ndim == 2:  
        filtered_image = I_layers[:, :, rank]  
    else:  
        filtered_image = I_layers[:, :, :, rank]  
  
    return filtered_image
```

```

def wiener(src, k):
    """Wiener filter"""
    rows, cols = src.shape[0:2]
    # Define parameters
    k_size = (k, k)
    kernel = np.ones((k_size[0], k_size[1]))
    # Convert to float
    # and make image with border
    if src.dtype == np.uint8:
        img_copy = src.astype(np.float32) / 255
    else:
        img_copy_nb = src
    img_copy = cv.copyMakeBorder(img_copy, int((k_size[0] - 1) / 2),
                                int(k_size[0] / 2), int((k_size[1] - 1) / 2),
                                int(k_size[1] / 2), cv.BORDER_REPLICATE)
    # Split into layers
    bgr_planes = cv.split(img_copy)
    bgr_planes_2 = []
    k_power = np.power(kernel, 2)
    for plane in bgr_planes:
        plane_power = np.power(plane, 2)
        m = np.zeros(src.shape[0:2], np.float32)
        q = np.zeros(src.shape[0:2], np.float32)
        for i in range(k_size[0]):
            for j in range(k_size[1]):
                m = m + kernel[i, j] * plane[i:i + rows, j:j + cols]
                q = q + k_power[i, j] * plane_power[i:i + rows, j:j + cols]
        m = m / np.sum(kernel)
        q = q - m * m
        v = np.sum(q) / src.size
        # Do filter
        plane_2 = plane[(k_size[0] - 1) // 2:
                          (k_size[0] - 1) // 2 + rows, (k_size[1] - 1) // 2:
                          (k_size[1] - 1) // 2 + cols]
        plane_2 = np.where(q < v, m, (plane_2 - m) * (1 - v / q) + m)
        bgr_planes_2.append(plane_2)
    # Merge image back
    filtered_image = cv.merge(bgr_planes_2)
    if src.dtype == np.uint8:
        filtered_image = (255 * filtered_image).clip(0, 255).astype(np.uint8)

    return filtered_image

```

```

def ad_median_filter(src, k, s=3):
    """Adaptive Median Filtering"""
    rows, cols = src.shape[0:2]
    img_out = np.zeros((rows, cols))
    # filling arrays for each kernel
    for r in range(rows):
        for c in range(cols):
            while s <= k:

```

```

        window = src[max(0, r - s // 2):min(rows, r + s // 2 + 1),
max(0, c - s // 2):min(cols, c + s // 2 + 1)]
        mn = np.min(window)
        mx = np.max(window)
        md = np.median(window)
        if mn < md < mx:
            if mn < src[r, c] < mx:
                img_out[r, c] = src[r, c]
            else:
                img_out[r, c] = md
            break
        else:
            s += 2
    if s > k:
        img_out[r, c] = src[r, c]

return img_out.astype(np.uint8)

```

```

def image_filtering(option: int, noisy_image, name: str, kx: int = 3, ky: int = 3, m=3, n=3, q=0.0, k=0, r=0):
    """Image Filtering"""
    match option:
        case 1:
            # Gaussian Blur
            gb = cv.GaussianBlur(noisy_image, (kx, ky), 0)
            image_display(gb, f'Gaussian Blur_{name}_({kx},{ky})',
'Gaussian Blur')
        case 2:
            # Contraharmonic mean filter
            ch = contraharmonic(noisy_image, m, n, q)
            image_display(ch, f'Contraharmonic_{name}_({m},{n}={m,n}),q={q})',
'Contraharmonic Filter')
        case 3:
            # Median Filter
            md = cv.medianBlur(noisy_image, k)
            image_display(md, f'Median_{name}_({k={k}})', 'Median Filter')
        case 4:
            # 2D-Median Filter
            mdf = scipy.signal.medfilt2d(noisy_image, k)
            image_display(mdf, f'Median_2D{name}_({k={k}})', 'Median_2D_Filter')
        case 5:
            # Rang Filter
            rn = rang_filter(noisy_image, k, r)*255
            image_display(rn, f'Rang_{name}_({k={k}},r={r+1})', 'Rang Filter')
        case 6:
            # Wiener Filter
            wn = wiener(noisy_image, k)
            image_display(wn, f'Wiener_{name}_({k={k}})', 'Wiener Filter')
        case 7:
            # Adaptive Median Filter
            am = ad_median_filter(noisy_image, k)
            image_display(am, f'Adaptive_Median_{name}_k={k}', 'Adaptive_Median_Filter')

```

```

    case _:
        # displaying noisy image
        image_display(noisy_image, 'Noisy_image', 'Noisy_images')
return 0

```

```

def robertson_detection(src_image):
    g_x = np.array([[1, -1], [0, 0]])
    g_y = np.array([[1, 0], [-1, 0]])
    i_x = cv.filter2D(src_image, -1, g_x)
    i_y = cv.filter2D(src_image, -1, g_y)

    abs_grad_x = cv.convertScaleAbs(i_x)
    abs_grad_y = cv.convertScaleAbs(i_y)

    return cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)

```

```

def prewitt_detection(src_image):
    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]])
    i_x = cv.filter2D(src_image, -1, g_x)
    i_y = cv.filter2D(src_image, -1, g_y)

    abs_grad_x = cv.convertScaleAbs(i_x)
    abs_grad_y = cv.convertScaleAbs(i_y)

    return cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)

```

```

def sobel_detection(src_image):
    i_x = cv.Sobel(src_image, cv.CV_16S, 1, 0)
    i_y = cv.Sobel(src_image, cv.CV_16S, 0, 1)

    abs_grad_x = cv.convertScaleAbs(i_x)
    abs_grad_y = cv.convertScaleAbs(i_y)

    return cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)

```

```

def laplassian_detection(src_image):
    g_d = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
    return cv.filter2D(src_image, -1, g_d)

```

```

def edge_detection(option: int, noisy_image, title):
    match option:
        case 1:
            rb = robertson_detection(noisy_image)
            image_display(rb, f'Robertson_{title}', 'Edge_Detection')
        case 2:
            pr = prewitt_detection(noisy_image)
            image_display(pr, f'Prewitt_{title}', 'Edge_Detection')

```

```
case 3:  
    sb = sobel_detection(noisy_image)  
    image_display(sb, f'Sobel_{title}', 'Edge_Detection')  
case 4:  
    lp = laplassian_detection(noisy_image)  
    image_display(lp, f'Laplassian_{title}', 'Edge_Detection')  
case 5:  
    cn = cv.Canny(noisy_image, 100, 200)  
    image_display(cn, f'Canny_{title}', 'Edge_Detection')  
case _:  
    pass
```