Лабораторная работа №4
Фильтрация и выделение контуров

Цель работы

Освоение основных способов фильтрации изображений от шумов и выделения контуров.

Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB или библиотеки OpenCV для работы фильтрацией изображений и методами низкочастной и высокочастотной фильтрации. Лабораторная работа рассчитана на 4 часа.

Теоретические сведения

Типы шумов

Цифровые изображения, полученные различными оптико-электронными приборами, могут содержать в себе разнообразные искажения, обусловленные разного рода помехами, которые принято называть *шумом*. Шум на изображении затрудняет его обработку автоматическими средствами, и, поскольку шум может иметь различную природу, для его успешного подавления необходимо определить адекватную математическую модель. Рассмотрим наиболее распространенные модели шумов. В среде MATLAB шум может быть наложен на изображение с помощью функции `imnoise()`. К сожалению, в библиотеке OpenCV отсутствуют функции для наложения шумов на изображения, но они могут быть реализованы используя возможности данной библиотеки. С другой стороны, в библиотеке `SciPy` для языка порграммирования Python есть функция `skimage.util.random_noise()`, предназначенная для наложения шумов на изображение и аналогичная функции `imnoise()` из среды MATLAB.

Импульсный шум

При импульсном шуме сигнал искажается выбросами с очень большими отрицательными или положительными значениями ма-

Рис. 4.1 — Исходное полутоновое изображение.

лой длительностью и может возникать, например, из-за ошибок декодирования. Такой шум приводит к появлению на изображении белых («соль») или черных («перец») точек, поэтому зачастую называется *точечным* шумом. Для его описания следует принять во внимание тот факт, что появление шумового выброса в каждом пикселе $I(x,y)$ не зависит ни от качества исходного изображения, ни от наличия шума в других точках и имеет вероятность появления p, причем значение интенсивности пикселя $I(x,y)$ будет изменено на значение $d \in [0,255]$:

$$I(x,y) = \begin{cases} d, \text{ с вероятностью } p, \\ s_{x,y}, \text{ с вероятностью } (1-p), \end{cases} \tag{4.1}$$

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

В среде MATLAB импульсный шум задается параметром `'salt & pepper'` функции `imnoise()`: `imnoise(I,'salt & pepper')`. При использовании языка программирования Python наложение данного шума выполняется функцией `skimage.util.random_noise(I, 's&p')` библиотеки обработки изображений `SciPy` . Необязательный параметр `amount` используется для изменения вероятности добавления шума к пикселю. Следующий необязательный параметр `salt_vs_pepper` используется для задания соотношения вероятностей шумов типа «соль» и «перец». Функция `skimage.util.random_noise(I, 'salt')` накладывает только шум типа «соль», а функция `skimage.util.random` — только шум ти-

па «перец». В библиотеке OpenCV отсутствует функция для наложения импульсного шума, однако она может быть реализована на основе матричных операций над изображениями (используя класс `Mat` в C++ и массивы NumPy в Python). В Приложении 4.1 представлен пример программной реализации наложения импульсного шума с использованием библиотеки OpenCV.

Аддитивный шум

Аддитивный шум описывается следующим выражением:

$$I_{new}(x,y) = I(x,y) + \eta(x,y), \tag{4.2}$$

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

Мультипликативный шум

Мультипликативный шум описывается следующим выражением:

$$I_{new}(x,y) = I(x,y) \cdot \eta(x,y), \tag{4.3}$$

где I_{new} — зашумленное изображение, I — исходное изображение, η — не зависящий от сигнала мультипликативный шум, умножающий зарегистрированный сигнал. В качестве примера можно привести зернистость фотопленки, ультразвуковые изображения и т.д. Частным случаем мультипликативного шума является *спекл*-шум, который появляется на изображениях, полученных устройствами с когерентным формированием изображений, например, медицинскими сканерами или радарами. На таких изображениях можно отчетливо наблюдать светлые пятна, крапинки (спеклы), которые разделены темными участками изображения.

В среде MATLAB спекл-шум накладывается на изображение `I` функцией `imnoise(I,'speckle')`. В языке программирования Python спекл-шум можно наложить на изображение функцией `skimage.util.random_noise(I, 'speckle')` библиотеки `SciPy`. Необязательные параметры `mean` и `var` используются для задания параметров нормального распределения, используемого при наложении шума. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 4.1

представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Гауссов (нормальный) шум

Гауссов шум на изображении может возникать в следствие недостатка освещенности сцены, высокой температуры и т.д. Модель шума широко распространена в задачах низкочастотной фильтрации изображений. Функция распределения плотности вероятности $p(z)$ случайной величины z описывается следующим выражением:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-\mu)^2}{2\sigma^2}}, \tag{4.4}$$

где z — интенсивность изображения (например, для полутонового изображения $z \in [0{,}255]$), μ — среднее (математическое ожидание) случайной величины z, σ — среднеквадратичное отклонение, дисперсия σ^2 определяет мощность вносимого шума. Примерно 67% значений случайной величины z сосредоточено в диапазоне $[(\mu - \sigma),(\mu + \sigma)]$ и около 96% в диапазоне $[(\mu - 2\sigma),(\mu + 2\sigma)]$.

В среде MATLAB шум может быть задан с помощью функции `imnoise(I,'gaussian')` или `imnoise(I,'localvar')` в случае нулевого математического ожидания. В языке программирования Python шум Гаусса можно наложить на изображение функцией `skimage.util.random_noise(I, 'gaussian')` библиотеки `SciPy`. Необязательные параметры `mean` и `var` используются для задания параметров нормального распределения, используемого при наложении шума. Функция `skimage.util.random_noise(I, 'localvar')` позволяет задать локальную дисперсию шума для каждой точки изображения дополнительным параметром `local_vars`. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 4.1 представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Шум квантования

Зависит от выбранного шага квантования и самого сигнала. Шум квантования может приводить, например, к появлению лож-

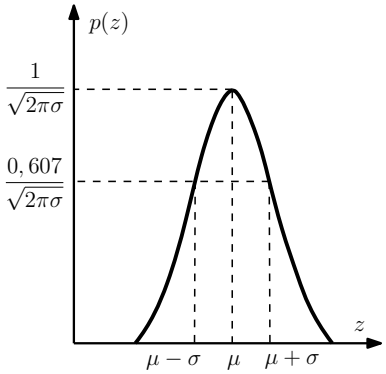


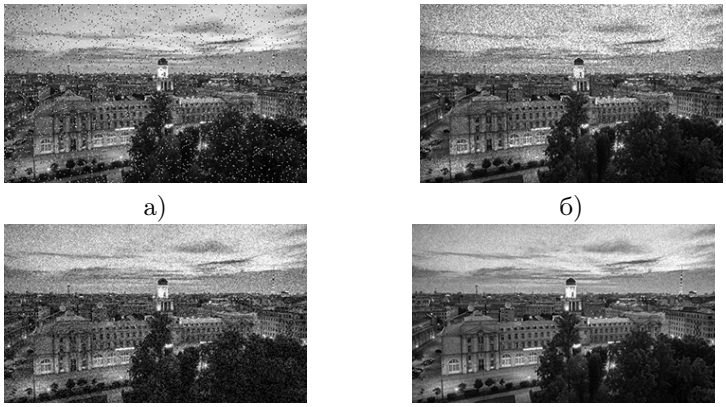
Рис. 4.2 — Функция распределения плотности вероятности $p(z)$.

ных контуров вокруг объектов или убирать слабо контрастные детали на изображении. Такой шум не устраняется.

Приближенно шум квантования можно описать распределением Пуассона и задать функцией `imnoise(I,'poisson')` в среде MATLAB. В языке программирования Python шум квантования можно наложить, используя функцию `skimage.util.random_noise(I, 'poisson')` библиотеки `SciPy`. В языке программирования C++ необходимо реализовывать данный тип шума самостоятельно. В приложении 4.1 представлен пример программной реализации наложения шумов с использованием библиотеки OpenCV и языков программирования C++ и Python.

Фильтрация изображений

Рассмотрим основные методы фильтрации изображений. Если для вычисления значения интенсивности каждого пикселя учитываются значения соседних пикселей в некоторой окрестности, то такое преобразование называется *локальным*, а сама окрестность — *окном*, представляющим собой некоторую матрицу, называемую *маской*, *фильтром*, *ядром фильтра*, а сами значения элементов матрицы называются *коэффициентами*. Центр маски совмещается с анализируемым пикселем, а коэффициенты маски умножаются на значения интенсивностей пикселей, накрытых маской. Как



а) б) в) г)

Рис. 4.3 — Результат наложения: а) шума типа «соль» и «перец», б) спекл-шума, в) нормального шума, г) шума Пуассона.

правило, маска имеет квадратную форму размера 3×3, 5×5 и т.п. Фильтрация изображения I, имеющего размеры $M \times N$, с помощью маски размера $m \times n$ описывается формулой:

$$I_{new}(x,y) = \sum_s \sum_t w(s,t) I(x+s, y+t), \tag{4.5}$$

где s и t — координаты элементов маски относительно ее центра (в центре $s = t = 0$). Такого рода преобразования называются *линейными*. После вычисления нового значения интенсивности пикселя $I_{new}(x,y)$ окно w, в котором описана маска фильтра, сдвигается и вычисляется интенсивность следующего пикселя, поэтому подобное преобразование называется *фильтрацией в скользящем окне*.

В среде MATLAB фильтрация изображения может быть осуществлена при помощи функции `filter2(mask,I)`, где матрица `mask` задает маску фильтра. Маска может задаваться либо вручную, либо с помощью функции `fspecial()`. В библиотеке OpenCV фильтрация изображений выполняется функцией `cv::filter2D(src, dst, ddepth, kernel)` в C++ и `dst = cv.filter2D(src, ddest, kernel)` в Python. Параметр `src` — входное изображение, `dst` — выходное изображение, `kernel` — мас-

ка фильтра, а параметр `ddepth` задает глубину цвета выходного изображения. Значение глубины цвета -1 сохраняет глубину цвета исходного изображения неизменной. Маску фильтра можно задать вручную, создав новый объект типа `Mat` в C++:

```
mask = (Mat_<double>(3, 3) << 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

Или NumPy массив в Python:

```
mask = np.float64([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
```

Низкочастотная фильтрация

Низкочастотные пространственные фильтры ослабляют высокочастотные компоненты (области с сильным изменением интенсивностей) и оставляют низкочастотные компоненты изображения без изменений. Используются для снижения уровня шума и удаления высокочастотных компонент, что позволяет повысить точность исследования содержания низкочастотных компонент. В результате применения низкочастотных фильтров получим сглаженное или размытое изображение. Главными отличительными особенностями ядра низкочастотного фильтра являются:

1. неотрицательные коэффициенты маски;
2. сумма всех коэффициентов равна единице.

Примеры ядер низкочастотных фильтров:

$$w = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}, w = \frac{1}{10}\begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 1\\ 1 & 1 & 1\end{bmatrix}. \tag{4.6}$$

Рассмотрим основные виды низкочастнотных сглаживающих фильтров.

Арифметический усредняющий фильтр

Данный фильтр усредняет значение интенсивности пикселя по окрестности с использованием маски с одинаковыми коэффициентами, например, для маски размером 3×3 коэффициенты равны

1/9, при 5×5 — 1/25. Благодаря такому нормированию значение результата фильтрации будет приведено к диапазону интенсивностей исходного изображения. Графически двумерная функция, описывающая маску фильтра, похожа на параллелепипед, поэтому в англоязычной литературе используется название *box*-фильтр. Арифметическое усреднение достигается при использовании следующей формулы:

$$I_{new}(x,y) = \frac{1}{m \cdot n} \sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j), \tag{4.7}$$

где $I_{new}(x,y)$ — значение интенсивности пикселя отфильтрованного изображения, $I(i,j)$ — значение интенсивностей пикселей исходного изображения в маске, m и n — ширина и высота маски фильтра соответственно. Данный алгоритм эффективен для слабо зашумленных изображений.

В среде MATLAB фильтрация изображения `I` с маской размера 3×3 может быть осуществлена при помощи функции `filter2(fspecial('average',3),I)`. При использовании библиотеки OpenCV аналогичная фильтрация может быть осуществлена при помощи функции `cv::blur(I, I_out, Size(3, 3))` в C++ или `cv2.blur(I, (3, 3))` в Python.

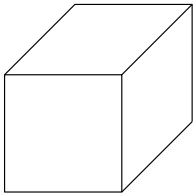


Рис. 4.4 — Графическое представление *box*-фильтра.

Геометрический усредняющий фильтр

Геометрическое усреднение рассчитывается при помощи формулы:

$$I_{new}(x,y) = \left[\prod_{i=0}^{m} \prod_{j=0}^{n} I(i,j) \right]^{\frac{1}{m \cdot n}}. \tag{4.8}$$

Эффект от применения данного фильтра аналогичен предыдущему методу, однако отдельные объекты исходного изображения искажаются меньше. Фильтр может использоваться для подавления высокочастотного аддитивного шума с лучшими статистическими характеристиками по сравнению с арифметическим усредняющим фильтром.

Геометрический усредняющий фильтр может быть задан через арифметический усредняющий фильтр и функции логарифма и экспоненты, что делает упрощает реализацию в матричном виде:

$$I_{new}(x,y) = e^{\frac{1}{m \cdot n} \sum_{i=0}^{m} \sum_{j=0}^{n} \ln(I(i,j))}. \tag{4.9}$$

Гармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x,y) = \frac{m \cdot n}{\sum_{i=0}^{m} \sum_{j=0}^{n} \frac{1}{I(i,j)}}. \tag{4.10}$$

Фильтр хорошо подавляет шумы типа «соль» и не работает с шумами типа «перец».

Контргармонический усредняющий фильтр

Фильтр базируется на выражении:

$$I_{new}(x,y) = \frac{\sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j)^{Q+1}}{\sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j)^{Q}}, \tag{4.11}$$

где Q — порядок фильтра. Контргармонический фильтр является обобщением усредняющих фильтров и при $Q > 0$ подавляет шумы типа «перец», а при $Q < 0$ — шумы типа «соль», однако одновременное удаление белых и черных точек невозможно. При $Q = 0$ фильтр превращается в арифметический, а при $Q = -1$ — в гармонический.

Фильтр Гаусса

Пиксели в скользящем окне, расположенные ближе к анализируемому пикселю, должны оказывать большее влияние на результат фильтрации, чем крайние. Поэтому коэффициенты весов маски можно описать колоколообразной функцией Гаусса (4.4). При фильтрации изображений используется двумерный фильтр Гаусса:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-y^2}{2\sigma^2}}. \tag{4.12}$$

Чем больше параметр σ, тем сильнее размывается изображение. Как правило, радиус фильтра $r = 3\sigma$. В таком случае размер маски $2r + 1 \times 2r + 1$ и размер матрицы $6\sigma + 1 \times 6\sigma + 1$. За пределами данной окрестности значения функции Гаусса будут пренебрежимо малы. В среде MATLAB фильтрация изображения фильтром Гаусса может быть осуществлена при помощи функции `imgaussfilt(I)`. При использовании OpenCV фильтрация изображения фильтром Гаусса может быть осуществлена при помощи функции `cv::GaussianBlur(I, I_out)` в C++ и функции `cv2.cv::GaussianBlur(I)` в Python.

Нелинейная фильтрация

Низкочастотные фильтры линейны и оптимальны в случае, когда имеет место нормальное распределение помех на цифровом изображении. Линейные фильтры локально усредняют импульсные помехи, сглаживая изображения. Для устранения импульсных помех лучше использовать нелинейные, например, *медианные* фильтры.

Медианная фильтрация

В классическом медианном фильтре используется маска с единичными коэффициентами. Произвольная форма окна может задаваться при помощи нулевых коэффициентов. Значения интенсивностей пикселей в окне представляются в виде вектора-столбца и сортируются по возрастанию. Отфильтрованному пикселю присвается медианное (среднее) в ряду значение интенсивности. Номер медианного элемента после сортировки может быть вычислен по формуле $n = \frac{N+1}{2}$, где N — число пикселей, участвующих в

сортировке. В среде MATLAB медианный фильтр может быть реализован с использованием функции `medfilt2(I)`. При использовании OpenCV, медианная фильтрация изображения может быть выполнена при помощи функции `cv::medianBlur(I, I_out, ksize)` в C++ и `cv2.medianBlur(I, ksize)` в Python.

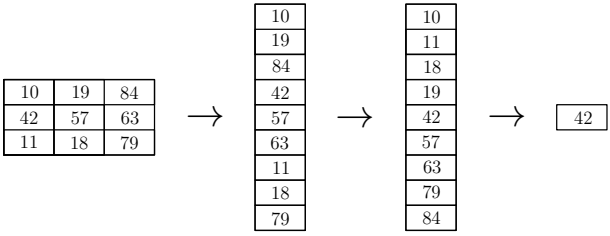


Рис. 4.5 — Иллюстрация работы медианной фильтрации в окне 3×3.

Взвешенная медианная фильтрация

В данной модификации медианной фильтрации в маске используются весовые коэффициенты (числа $2, 3$ и т.д.), чтобы отразить бóльшее влияние на результат фильтрации пикселей, расположенных ближе к фильтруемому элементу. Медианная фильтрация качественно удаляет имульсные шумы, а также на полутоновых изображениях не вносит новых значений интенсивностей. При увеличении размера окна увеличивается шумоподавляющая способность фильтра, но начинают искажаться очертания объектов. В MATLAB может быть реализован с использованием функции `medfilt2(I, [m n])`, где второй аргумент `[m n]` функции `medfilt2()` задает маску фильтра размера $m \times n$. В библиотеке OpenCV отсутствует функция для взвешенной медианной фильтрации, но она может быть реализована с использованием возможностей библиотеки. В Приложении 4.2 представлен пример программной реализации взвешенного медианного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Адаптивная медианная фильтрация

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

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 $s \leqslant s_{max}$, 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 $z_{i,j}$.

Ранговая фильтрация

Обобщением медианной фильтрации является *ранговый фильтр* порядка r, выбирающий из полученного вектора-столбца элементов маски пиксель с номером $r \in [1,N]$, который и будет являться результатом фильтрации.

1. Если число пикселей в окне N нечетное и $r = \frac{N+1}{2}$, тогда ранговый фильтр является *медианным*. В случае окна 3×3 в MATLAB можно воспользоваться функцией `ordfilt2(I,5,ones(3,3))`.

2. Если $r = 1$, фильтр выбирает наименьшее значение интенсивности и называется *min-фильтром*. В MATLAB задается функцией `ordfilt2(I,1,ones(3,3))`.

3. Если $r = N$, фильтр выбирает максимальное значение интенсивности и называется *max-фильтром*. В MATLAB задается функцией `ordfilt2(I,9,ones(3,3))`.

Ранг иногда записывается в процентах, например для *min-фильтра* ранг равен 0%, медианного — 50%, *max-фильтра* — 100%.

В библиотеке OpenCV отсутствует функция для ранговой фильтрации изображения, но она может быть реализована с использованием возможностей библиотеки. В Приложении 4.2 представлен пример программной реализации данного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Винеровская фильтрация

Использует пиксельно-адаптивный метод Винера, основанный на статистических данных, оцененных из локальной окрестности каждого пикселя.

$$\mu = \frac{1}{n \cdot m} = \sum_{i=0}^{m} \sum_{j=0}^{n} I(i,j), \tag{4.13}$$

$$\sigma^2 = \frac{1}{n \cdot m} = \sum_{i=0}^{m} \sum_{j=0}^{n} I^2(i,j) - \mu^2, \tag{4.14}$$

$$I_{new}(x,y) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} \left(I(x,y) - \mu \right), \tag{4.15}$$

где μ — среднее в окрестности, σ^2 — дисперсия, v^2 — дисперсия шума. В MATLAB реализуется при помощи функции `wiener2(I,[m n])`.

В библиотеке OpenCV отсутствует функция для фильтрации изображения методом Вейнера, но она может быть реализована с использованием возможностей библиотеки. В Приложении 4.3 представлен пример программной реализации данного фильтра с использованием библиотеки OpenCV и языков программирования C++ и Python.

Высокочастотная фильтрация

Высокочастотные пространственные фильтры усиливают высокочастотные компоненты (области с сильным изменением интенсивностей) и ослабляют низкочастотные составляющие изображения. Используются для выделения перепадов интенсивностей и определения границ (контуров) на изображениях. Для этого в MATLAB может быть использована функция `edge()`. В результате применения высокочастотных фильтров повышается резкость изображения.

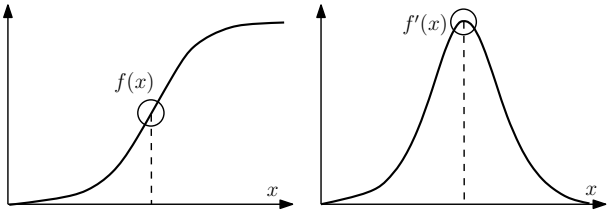


Рис. 4.6 — Функция интенсивности и ее первая производная, максимум производной соответствует краю.

Высокочастотные фильтры аппроксимируют вычисление производных по направлению, при этом приращение аргумента Δx принимается равным 1 или 2. Главными отличительными особенностями являются:

1. коэффициенты маски фильтра могут принимать отрицательные значения;

2. сумма всех коэффициентов равна нулю.

Фильтр Робертса

Фильтр Робертса работает с минимально допустимой для вычисления производной маской размерности 2×2, поэтому является быстрым и довольно эффективным. Возможные варианты масок для нахождения градиента по осям Ox и Oy:

$$G_x = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}, \tag{4.16}$$

либо

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}. \tag{4.17}$$

В результате применения дифференциального оператора Робертса получим оценку градиента по направлениям G_x и G_y. Модуль градиента всех краевых детекторов может быть вычислен по формуле $G = \sqrt{G_x^2 + G_y^2} = |G_x| + |G_y|$, а направление градиента — по формуле $\arctan\left(\frac{G_y}{G_x}\right)$.

В MATLAB с использованием дифференциального оператора Робертса границы можно выделить при помощи функции `edge(I, 'Roberts')`. При использовании библиотеки OpenCV он реализуется двумя вызовами функции `cv::filter2D` с масками G_x and G_y и вычислением среднеквадратичного значения полученных изображений с помощью функции `cv::magnitude`. На языке программирования C++ программная реализация примет вид:

```
Mat G_x = (Mat_<double>(2, 2) << -1, 1, 0, 0);
Mat G_y = (Mat_<double>(2, 2) << 1, 0, -1, 0);
Mat I_x, I_y, I_out;
cv::filter2D(I, I_x, -1, G_x);
cv::filter2D(I, I_y, -1, G_y);
cv::magnitude(I_x, I_y, I_out);
```

На языке программирования Python программная реализация будет аналогичной:

```
G_x = np.array([[1, -1], [0, 0]])
G_y = np.array([[1, 0], [-1, 0]])
I_x = cv.filter2D(I, -1, G_x)
I_y = cv.filter2D(I, -1, G_y)
I_out = cv.magnitude(I_x, I_y)
```

Фильтр Превитта

В данном подходе используются две ортогональные маски размером 3×3, позволяющие более точно вычислить производные по осям Ox и Oy:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}. \quad (4.18)$$

В MATLAB границы фильтром Превитта можно выделить при помощи функции `edge(I, 'Prewitt')`. При использовании библиотеки OpenCV фильтр Превитта реализуется аналогично с фильтром Робертса, как было описано выше.

Фильтр Собела

Данный подход аналогичен фильтру Робертса, однако используются разные веса в масках. Типичный пример фильтра Собела:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \quad (4.19)$$

В MATLAB границы фильтром Собела можно выделить при помощи функции `edge(I, 'Sobel')`. При использовании библиотеки OpenCV он может быть реализован аналогично с фильтром Робертса, как было описано выше, или же при помощи функции `cv::Sobel(I, I_out, ddepth, dx, dy)` в C++ или `cv2.Sobel(I, ddepth, dx, dy)` в Python.

Фильтр Лапласа

Фильтр Лапласа использует аппроксимацию вторых производных по осям Ox и Oy, в отличие от предыдущих подходов, использующих первую производную:

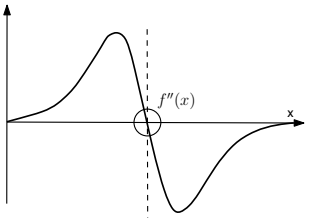


Рис. 4.7 — Вторая производная функции яркости меняет знак (проходит через ноль в точке, соответствующей краю).

$$L(I(x,y)) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}. \tag{4.20}$$

Формула 4.20 может быть аппроксимирована следующей маской:

$$w = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{4.21}$$

Алгоритм Кэнни

Одним из самых распространенных и эффективных алгоритмов выделения контуров на изображении является *алгоритм Кэнни*. Данный алгоритм позволяет не только определять краевые пиксели, но и связные граничные линии. Алгоритм состоит из следующих шагов:

1. Устранение мелких деталей путем сглаживания исходного изображения с помощью фильтра Гаусса.

2. Использование дифференциального оператора Собела для определения значений модуля градиента всех пикселей изображения, причем результат вычисления округляется с шагом $45°$.

3. Анализ значений модулей градиента пикселей, расположенных ортогонально исследуемому. Если значение модуля градиента исследуемого пикселя больше, чем у ортогональных соседних пикселей, то он является *краевым*, в противном случае — *немаксимумом*.

4. Выполнение двойной пороговой фильтрации краевых пикселей, отобранных на предыдущем шаге:

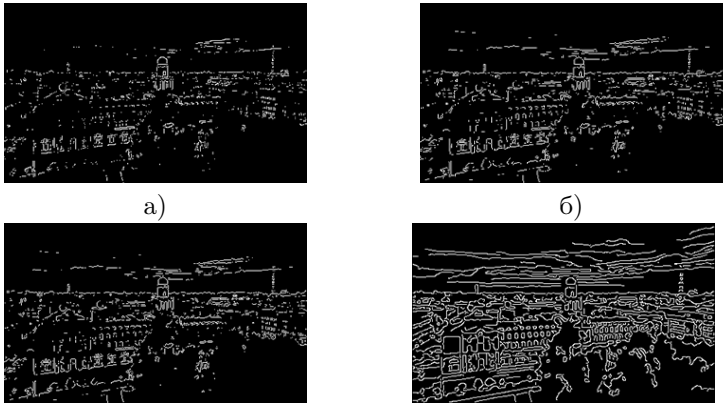
   (a) Если значение модуля градиента выше порога t_2, то наличие края в пикселе является достоверным.

   (b) Если значение модуля градиента ниже порога t_1, то пиксель однозначно не является краевым.

   (c) Если значение модуля градиента лежит в интервале $[t_1, t_2]$, то такой пиксель считается *неоднозначным*.

5. Подавление всех неоднозначных пикселей, не связанных с достоверными пикселями по 8-связности.

В MATLAB алгоритмом Кэнни можно выделить границы при помощи функции `edge(I, 'Canny')`. В библиотеке OpenCV алгоритм Canny реализуется функцией `cv::Canny(I, I_out, t1, t2)` в C++ и `cv2.Canny(I, t1, t2)` в Python.



а) б) в) г)

Рис. 4.8 — Результат выделения границ: а) фильтром Робертса, б) фильтром Превитта, в) фильтром Собела, г) алгоритмом Кэнни.

Порядок выполнения работы

1. *Типы шумов.* Выбрать произвольное изображение. Получить искаженные различными шумами изображения с помощью функции `imnoise()` с отличными от значений по умолчанию параметрами.

2. *Низкочастотная фильтрация.* Обработать полученные в предыдущем пункте искаженные изображения фильтром Гаусса и контргармоническим усредняющим фильтром с различными значениями параметра Q.

3. *Нелинейная фильтрация.* Обработать полученные в первом пункте искаженные изображения медианной, взвешенной медианной, ранговой и винеровской фильтрациями при различных размерах маски и ее коэффициентов. Реализовать адаптивную медианную фильтрацию.

4. *Высокочастотная фильтрация.* Выбрать исходное изображение. Выделить границы фильтрами Робертса, Превитта, Собела, Лапласа, алгоритмом Кэнни.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Теоретическое обоснование применяемых методов и функций фильтрации изображений.
4. Ход выполнения работы:
   (a) Исходные изображения;
   (b) Листинги программных реализаций;
   (c) Комментарии;
   (d) Результирующие изображения.
5. Выводы о проделанной работе.

Вопросы к защите лабораторной работы

1. В чем заключаются основные недостатки адаптивных методов фильтрации изображений?
2. При каких значениях параметра Q контргармонический фильтр будет работать как арифметический, а при каких — как гармонический?
3. Какими операторами можно выделить границы на изображении?
4. Для чего на первом шаге выделения контуров, как правило, выполняется низкочастотная фильтрация?

Приложение 4.1. Реализация наложения шумов на изображение с использованием библиотеки OpenCV

Наиболее эффективным способом наложения шумов при использовании языка программирования C++ будет попиксельная обработка изображения и вычисление шума методами, описанными в параграфе «Теоретические сведения». С другой стороны, при наличии дополнительных ресурсов ОЗУ можно реализовать наложение шумов с помощью матричных операций OpenCV.

Листинг 4.1. Наложение импульсного шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Noise parameter
2   double d = 0.05;
3   // Salat vs pepper distribution
4   double s_vs_p = 0.5;
5   // Split an image into layers
6   vector<Mat> I_out_bgr;
7   split(I, I_out_bgr);
8   // Process layers
9   for (int i = 0; i < I_out_bgr.size(); i++)
10  {
11    Mat vals(I_out_bgr[i].size(), CV_32F);
12    randu(vals, Scalar(0), Scalar(1));
13    if (I_out_bgr[i].depth() == CV_8U)
14      I_out_bgr[i].setTo(Scalar(255),
15        vals < d * s_vs_p);
16    else
17      I_out_bgr[i].setTo(Scalar(1),
18        vals < d * s_vs_p);
19    I_out_bgr[i].setTo(Scalar(0),
20      (vals >= d * s_vs_p) & (vals < d));
21  }
22  // Merge layers to create an output image
23  merge(I_out_bgr, I_out);
```

При наложении мультипликативного шума использование целочисленного представления цвета может привести к потере точности, поэтому входное изображение необходимо сконвертировать в представление с использованием вещественных чисел.

Листинг 4.2. Наложение мультипликативного шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Variance parameter
2   double var = 0.05;
3   // Split an image into layers
4   vector<Mat> I_out_bgr;
5   split(I, I_out_bgr);
6   // Process layers
7   for (int i = 0; i < I_out_bgr.size(); i++)
8   {
9     Mat gauss(I_out_bgr[i].size(), CV_32F);
10    randn(gauss, Scalar(0),
11          Scalar(sqrt(var)));
12    if (I_out_bgr[i].depth() == CV_8U)
13    {
14      Mat I_out_bgr_f;
15      I_out_bgr[i].convertTo(I_out_bgr_f,
16        CV_32F);
17      I_out_bgr_f += out_bgr_f.mul(gauss);
18      I_out_bgr_f.convertTo(I_out_bgr[i],
19        I_out_bgr[i].type());
20    }
21    else
22      I_out_bgr[i] += I_out_bgr[i].mul(gauss);
23  }
24  // Merge layers to create an output image
25  merge(I_out_bgr, I_out);
```

Листинг 4.3. Наложение гауссова шума на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Mean parameter
2   double mean = 0;
```

```
3   // Variance parameter
4   double var = 0.05;
5   // Split an image into layers
6   vector<Mat> I_out_bgr;
7   split(I, I_out_bgr);
8   // Process layers
9   for (int i = 0; i < I_out_bgr.size(); i++)
10  {
11    Mat gauss(I_out_bgr[i].size(), CV_32F);
12    randn(gauss, Scalar(mean),
13          Scalar(sqrt(var)));
14    if (I_out_bgr[i].depth() == CV_8U)
15    {
16      Mat I_out_bgr_f;
17      I_out_bgr[i].convertTo(I_out_bgr_f,
18        CV_32F);
19      I_out_bgr_f += gauss * 255;
20      I_out_bgr_f.convertTo(I_out_bgr[i],
21        I_out_bgr[i].type());
22    }
23    else
24      I_out_bgr[i] += gauss;
25  }
26  // Merge layers to create an output image
27  merge(I_out_bgr, I_out);
```

Поскольку в библиотеке OpenCV отсутствуют функции для генерации матрицы случайных чисел Пуассона, то для генерации шума квантования изображение должно быть обработано попиксельно.

Листинг 4.4. Наложение шума квантования на изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Convert image to floats
2   if (I.depth() == CV_8U)
3     I.convertTo(out, CV_32F, 1.0 / 255);
4   else
5     out = img.clone();
```

```
6   // Calculate quantization parameter
7   size_t vals = unique(out).size();
8   vals = (size_t)pow(2, ceil(log2(vals)));
9   int rows = out.rows;
10  int cols = out.cols * out.channels();
11  // If the image is continuous then
12  // we can process it as a single row
13  if (out.isContinuous())
14  {
15    cols *= rows;
16    rows = 1;
17  }
18  // Create poisson generator
19  using param_t =
20    std::poisson_distribution<int>::
21    param_type;
22  std::default_random_engine engine;
23  std::poisson_distribution<> poisson;
24  // Process image pixel-by-pixel
25  for (int i = 0; i < rows; i++)
26  {
27    float *ptr = out.ptr<float>(i);
28    for (int j = 0; j < cols; j++)
29      ptr[j] = float(poisson(engine,
30        param_t({ ptr[j] * vals }))) / vals;
31  }
32  // Convert back to uchar if needed
33  if (img.depth() == CV_8U)
34    out.convertTo(out, CV_8U, 255);
```

В представленной реализации функции наложения шума квантования используется функция `unique`, которая формирует массив из всех уникальных цветов пикселей изображения. Эта функция отсутствует в библиотеке OpenCV, но она может быть легко реализована.

Листинг 4.5. Поиск уникальных значений элементов матрицы с использованием библиотеки OpenCV и языка программирования C++.

```
vector<float> unique(const cv::Mat& I,
  bool sort = false)
{
  if (I.depth() != CV_32F)
  {
  std::cerr <<
    "unique() Only works with CV_32F Mat" <<
    std::endl;
  return std::vector<float>();
  }
std::vector<float> out;
// Acquire matrix size
int rows = img.rows;
int cols = img.cols * img.channels();
if (img.isContinuous())
{
  cols *= rows;
  rows = 1;
}
// Process each pixel
for (int y = 0; y < rows; y++)
{
  const float* row_ptr = img.ptr<float>(y);
  for (int x = 0; x < cols; x++)
  {
    float value = row_ptr[x];
    if (std::find(out.begin(), out.end(),
        value) == out.end())
      out.push_back(value);
  }
}
// Sort if needed
if (sort)
  std::sort(out.begin(), out.end());
return out;
}
```

Реализация функций наложения шума на языке Python заметно проще, так как все необходимые дополнительные функции уже присутствуют в библиотеке NumPy.

Листинг 4.6. Наложение импульсного шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Noise parameter
d = 0.05
# Salat vs pepper distribution
s_vs_p = 0.5
# Generate radnom numbers
rng = np.random.default_rng()
vals = rng.random(I.shape)
# Salt
I_out = np.copy(I)
if out.dtype == np.uint8:
  I_out[vals < d * s_vs_p] = 255
else:
  I_out[vals < d * s_vs_p] = 1.0
# Pepper
I_out[np.logical_and(vals >= d * s_vs_p,
      vals < d)] = 0
```

Листинг 4.7. Наложение мультипликативного шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Variance parameter
var = 0.05
# Generate radnom numbers
rng = np.random.default_rng()
gauss = rng.normal(0, var ** 0.5, I.shape)
# Process uchar and float images separately
if I.dtype == np.uint8:
  I_f = I.astype(np.float32)
  I_out = (I_f + I_f * gauss). \
        clip(0, 255).astype(np.uint8)
else:
  I_out = I + I * gauss
```

Листинг 4.8. Наложение гауссова шума на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
# Mean parameter
mean = 0
# Variance parameter
var = 0.01
# Generate radnom numbers
rng = np.random.default_rng()
gauss = \
  rng.normal(mean, var ** 0.5, I.shape)
gauss = gauss.reshape(I.shape)
# Process uchar and float images separately
if I.dtype == np.uint8:
  I_out = (I.astype(np.float32) +
    gauss * 255).clip(0, 255). \
    astype(np.uint8)
else:
  I_out = (I + gauss).astype(np.float32)
```

Листинг 4.9. Наложение шума квантования на изображение с использованием библиотеки OpenCV и языка программирования Python.

```
rng = np.random.default_rng()
if I.dtype == np.uint8:
  I_f = I.astype(np.float32) / 255
  vals = len(np.unique(I_f))
  vals = 2 ** np.ceil(np.log2(vals))
  I_out = (255 * \
    (rng.poisson(I_f * vals) / \
    float(vals)).clip(0, 1)). \
    astype(np.uint8)
else:
  vals = len(np.unique(I))
  vals = 2 ** np.ceil(np.log2(vals))
  I_out = \
    rng.poisson(I * vals) / float(vals)
```

Приложение 4.2. Реализация взвешенной ранговой фильтрации изображений с использованием библиотеки OpenCV

Реализации взвешенного медианного и рангового фильтров отличаются только одной строкой, то есть выбором элемента в отфильтрованном массиве на последнем шаге фильтрации. По этой причине их реализация может быть объединена в одну функцию следующим образом:

Листинг 4.10. Взвешенная ранговая фильтрация изображения с использованием библиотеки OpenCV и языка программирования C++. .

```
1   // Filter parameters
2   int k_size[] = { 3, 3 };
3   Mat kernel =
4     Mat::ones(k_size[0], k_size[1], CV_64F);
5   int rank = 4;
6   // Convert to float
7   // and make image with border
8   Mat I_copy;
9   if (I.depth() == CV_8U)
10    I.convertTo(I_copy, CV_32F, 1.0 / 255);
11  else
12    I_copy = I;
13  cv::copyMakeBorder(I_copy, I_copy,
14    int((k_size[0] - 1) / 2),
15    int(k_size[0] / 2),
16    int((k_size[1] - 1) / 2),
17    int(k_size[1] / 2), cv::BORDER_REPLICATE);
18  // Split into layers
19  vector<Mat> bgr_planes;
20  cv::split(I_copy, bgr_planes);
21  // Process all layers
22  for (int k = 0; k < bgr_planes.size(); k++)
23  {
24    Mat I_tmp = Mat::zeros(I.size(),
25      bgr_planes[k].type());
26    // Allocate memory for arrays
```

```
27      vector<double> c;
28      c.reserve(k_size[0] * k_size[1]);
29      // For each image pixel
30      for (int i = 0; i < I.rows; i++)
31        for (int j = 0; j < I.cols; j++)
32        {
33          // Empty array
34          c.clear();
35          // Fill array
36          for (int a = 0; a < k_size[0]; a++)
37            for (int b = 0; b < k_size[1]; b++)
38              c.push_back(bgr_planes[k].
39                at<float>(i + a, j + b) *
40                kernel.at<double>(a, b));
41          // Sort array
42          std::sort(c.begin(), c.end());
43          // Select id with given rank
44          I_tmp.at<float>(i, j) =
45            float(c[rank]);
46        }
47      bgr_planes[k] = I_tmp;
48    }
49    // Merge back
50    cv::merge(bgr_planes, I_out);
51    // Convert back to uint if needed
52    if (I.depth() == CV_8U)
53      I_out.convertTo(I_out, CV_8U, 255);
```

При реализации взвешенного рангового фильтра на языке программирования Python можно значительно ускорить выполнение кода за счет использования операций с массивами NumPy, однако для этого решения потребуется хранить несколько экземпляров исходного изображения, по одному для каждого элемента окна фильтра.

Листинг 4.11. Взвешенная ранговая фильтрация изображения с использованием библиотеки OpenCV и языка программирования Pyhton.

```
1   # Filter parameters
```

```
  k_size = (3, 3)
  rank = 4
  kernel = np.ones(k_size, dtype = np.float32)
  rows, cols = I.shape[0:2]
  # Convert to float
  # and make image with border
  if I.dtype == np.uint8:
    I_copy = I.astype(np.float32) / 255
  else:
    I_copy = I
  I_copy = cv.copyMakeBorder(I_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)
  # Fill arrays for each kernel item
  I_layers = np.zeros(I.shape +
    (k_size[0] * k_size[1], ),
    dtype = np.float32)
  if I.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] * \
          I_copy[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] * \
          I_copy[i:i + rows, j:j + cols, :]
  # Sort arrays
  I_layers.sort()
  # Choose layer with rank
  if I.ndim == 2:
    I_out = I_layers[:, :, rank]
  else:
    I_out = I_layers[:, :, :, rank]
```

```
40  # Convert back to uint if needed
41  if (I.dtype == np.uint8):
42    I_out = (255 * I_out).clip(0, 255). \
43      astype(np.uint8)
```

Приложение 4.3. Реализация фильтра Винера с использованием библиотеки OpenCV

Реализаций фильтра Винера требует двухпроходной обработки входного изображения: первый проход для оценки значения ошибки изображения, а второй проход непосредственно для фильтрации изображения.

Листинг 4.12. Винеровская фильтрация изображения с использованием библиотеки OpenCV и языка программирования C++. .

```
1   // Define parameters
2   int k_size[] = { 5, 5 };
3   Mat kernel = Mat::ones(k_size[0],
4     k_size[1], CV_64F);
5   double k_sum = cv::sum(kernel)[0];
6   // Convert to float
7   // and make image with border
8   Mat I_copy;
9   if (I.depth() == CV_8U)
10    I.convertTo(I_copy, CV_32F, 1.0 / 255);
11  else
12    I_copy = I;
13  cv::copyMakeBorder(I_copy, I_copy,
14    int((k_size[0] - 1) / 2),
15    int(k_size[0] / 2),
16    int((k_size[1] - 1) / 2),
17    int(k_size[1] / 2),
18    cv::BORDER_REPLICATE);
19  // Split into layers
20  vector<Mat> bgr_planes;
21  cv::split(I_copy, bgr_planes);
22  // Process all layers
```

```
23  for (int k = 0; k < bgr_planes.size(); k++)
24  {
25    Mat I_tmp = Mat::zeros(I.size(),
26      bgr_planes[k].type());
27    double v(0);
28    // Calculate the average of all
29    // the local estimated variances
30    for (int i = 0; i < I.rows; i++)
31      for (int j = 0; j < I.cols; j++)
32      {
33        // Calculate variance
34        double m(0), q(0);
35        for (int a = 0; a < k_size[0]; a++)
36          for (int b = 0; b < k_size[1]; b++)
37          {
38            double t = bgr_planes[k].
39              at<float>(i + a, j + b) *
40              kernel.at<double>(a, b);
41            m += t;
42            q += t * t;
43          }
44        m /= k_sum;
45        q /= k_sum;
46        q -= m * m;
47        v += q;
48      }
49    v /= I.cols * I.rows;
50    // For each image pixel
51    for (int i = 0; i < I.rows; i++)
52      for (int j = 0; j < I.cols; j++)
53      {
54        // Calculate variance
55        double m(0), q(0);
56        for (int a = 0; a < k_size[0]; a++)
57          for (int b = 0; b < k_size[1]; b++)
58          {
59            double t = bgr_planes[k].
60              at<float>(i + a, j + b) *
```

```
61                kernel.at<double>(a, b);
62              m += t;
63              q += t * t;
64            }
65          m /= k_sum;
66          q /= k_sum;
67          q -= m * m;
68          // Calculate pixel value
69          double im = bgr_planes[k].
70            at<float>(i + (k_size[0] - 1) / 2,
71            j + (k_size[1] - 1) / 2);
72          if (q < v)
73            I_tmp.at<float>(i, j) = float(m);
74          else
75            I_tmp.at<float>(i, j) =
76              float((im - m) * (1 - v / q) + m);
77      }
78      bgr_planes[k] = I_tmp;
79    }
80    // Merge back
81    cv::merge(bgr_planes, I_out);
82    // Convert back to uint if needed
83    if (I.depth() == CV_8U)
84      I_out.convertTo(I_out, CV_8U, 255);
```

Листинг 4.13. Винеровская фильтрация изображения с использованием библиотеки OpenCV и языка программирования Python.

```
1   # Define parameters
2   k_size = (7, 7)
3   kernel = np.ones((k_size[0], k_size[1]))
4   # Convert to float
5   # and make image with border
6   if I.dtype == np.uint8:
7     img_copy = I.astype(np.float32) / 255
8   else:
9     img_copy_nb = I
10  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 all layers
  for plane in bgr_planes:
    # Calculate temporary matrices for I ** 2
    plane_power = np.power(plane, 2)
    m = np.zeros(I.shape[0:2], np.float32)
    q = np.zeros(I.shape[0:2], np.float32)
    # Calculate variance values
    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 / np.sum(kernel)
    q = q - m * m
    # Calculate noise as an average variance
    v = np.sum(q) / I.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
  I_out = cv.merge(bgr_planes_2)
  # Convert back to uint if needed
```

```
49   if (I.dtype == np.uint8):
50     I_out = (255 * I_out).clip(0, 255). \
51       astype(np.uint8)
```