Лабораторна робота № 1.

Попередня обробка зображень

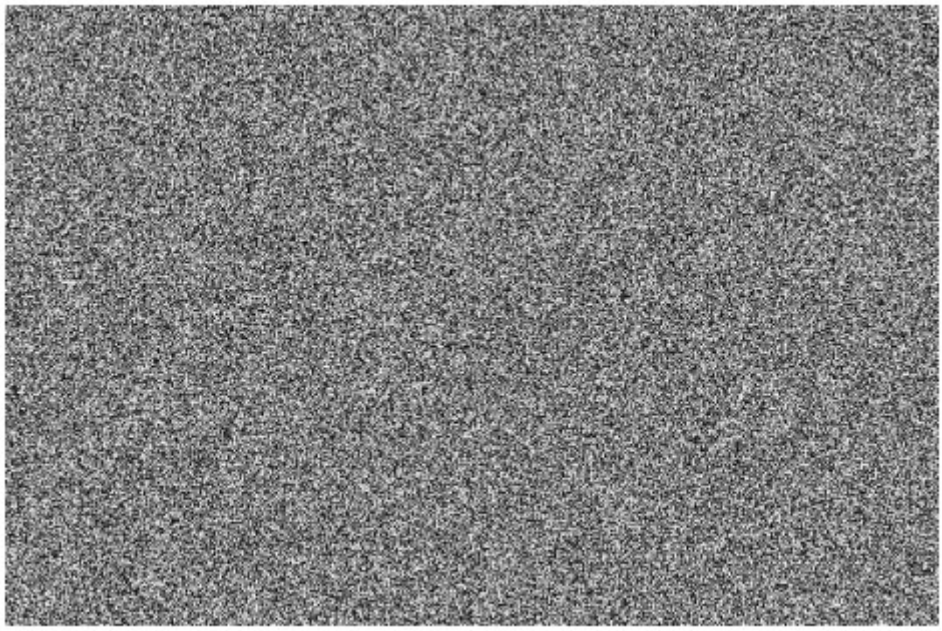
Мета - вивчити просторову фільтрацію зображень, методи мінімізації шуму, морфології, виділення країв і границь та елементи біблотеки OpenCV для розвязання цих завдань

У світі комп'ютерного зору фільтрація зображень використовується для модифікації зображень на етапі поперднього опрацювання. Ці зміни, по суті, дозволяють прояснити зображення, щоб отримати потрібну інформацію. Фільтрація може включати в себе все, що завгодно - видобуток країв з зображення, його розмиття, видалення небажаних об'єктів тощо.

Існує багато причин для використання фільтрації зображень. Наприклад, зйомка при сонячному світлі або в темряві вплине на чіткість зображення, тому можливо необхідновикористовувати фільтри зображень, щоб змінити зображення згідно власних потреб. Аналогічно, зображення може бути розмитим або зашумленим, яке потребувати уточнення і фокусування.

Навдемо приклад здійснення фільтрацію зображень засобами OpenCV.

На зображенні наведено шум типу “сіль і перець”. Такий шум виникає коли відбувається погіршення якості сигналу, що використовується для створення зображення.



Таке зображення вище може бути легко згенеровано за допомогою OpenCV наступним способом:

```
# initialize noise image with zeros
noise = np.zeros((400, 600))

# fill the image with random numbers in given range
cv2.randu(noise, 0, 256)
```

Додамо зважений шум до сірого зображення (зліва), щоб вхідне зображення виглядало так як наведено справа:

Код для цих дій виглядає наступним чином:

```
# add noise to existing image
noisy_gray = gray + np.array(0.2*noise, dtype=np.int)
```

Тут значення 0.2 використовується в якості параметра, що збільшує або зменшує значення для створення шуму різної інтенсивності.

У деяких додатках шум грає важливу роль в розширенні можливостей системи. Це особливо важливо при використанні моделей глибокого навчання. Шум стає способом перевірки точності моделі глибокого навчання і вбудовування його в алгоритм комп'ютерного зору.

Лінійна фільтрація зображення

1D лінійна фільтрація зображення

Найпростіший фільтр - це точковий оператор. Значення інтенсивності кожного пікселя множиться на скалярне значення. Цю операцію можна записати в такий спосіб:

$$g(i,j) = K \times f(i,j)$$

Тут:

- Вхідне зображення *F* та значення інтесивності пікселя в координатах (*i, j*) позначається як *f*(*i ,j*).
- Вихідне зображення є *G* та значення інтесивності пікселя в координатах (*i, j*) позначається як *g*(*i, j*).
- *K* – скалярна константа.

Цей тип роботи із зображенням називається лінійним фільтром. Крім множення на скалярний значення значення інтесивності кожного пікселя може бути збільшеним або зменшеним на постійну величину. Таким чином, можна записати загальну точкову операцію:

$$g(i,j) = K \times f(i,j) + L$$

Ця операція може бути застосована як до зображень в градаціях сірого, так і до зображень у форматі RGB. Для RGB-зображень кожен канал буде модифікований за допомогою цієї операції окремо.

Нижче наведені результати впливу різних значень *K* і *L*. Вхідне зображення наведено зліва. У другому зображенні: *K* = 0,5 і *L* = 0, а в третьому - *K* = 1 і *L* = 10. Для останнього зображення (праворуч) - *K* = 0.7 і *L* = 25.

Як видно з порівняння рисунків, зміна *K* змінює яскравість зображення, а зміна - контрастність зображення:

Ці зображення можуть бути згенерованими таким кодом:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
def point_operation(img, K, L):
"""
Applies point operation to given grayscale image
"""
img = np.asarray(img, dtype=np.float)
img = img*K + L
# clip pixel values
img[img > 255] = 255
img[img < 0] = 0
return np.asarray(img, dtype = np.int)
def main():
# read an image
img = cv2.imread('../figures/flower.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# k = 0.5, l = 0
out1 = point_operation(gray, 0.5, 0)
# k = 1., l = 10
out2 = point_operation(gray, 1., 10)
# k = 0.8, l = 15
out3 = point_operation(gray, 0.7, 25)
res = np.hstack([gray,out1, out2, out3])
plt.imshow(res, cmap='gray')
plt.axis('off')
plt.show()
if __name__ == '__main__':
main()
```

2D лінійна фільтрація зображення

Наведений фільтр “1D лінійної фільтрації зображення” є точковим. З іншого боку на реальному зображенні пікселі зображення мають інформацію навколо даного пікселя. І це дуже важливо. Так, наприклад, якщо на попередньому вхідному зображенні вибрати піксель пелюстки (значення функції інтесивності в ньому) і рухатися навколо нього, то отримувані значення будуть досить близькі. Це дає додаткову інформацію про зображення. Для видобутку цієї інформації при фільтрації існує кілька фільтрів-сусідів (“2D лінійна фільтрації зображення”).

У фільтрах-сусідах є матриця ядра, яка захоплює інформацію про локальну області навколо пікселя. Щоб пояснити ці фільтри, почнемо з вхідного зображення, як показано нижче:

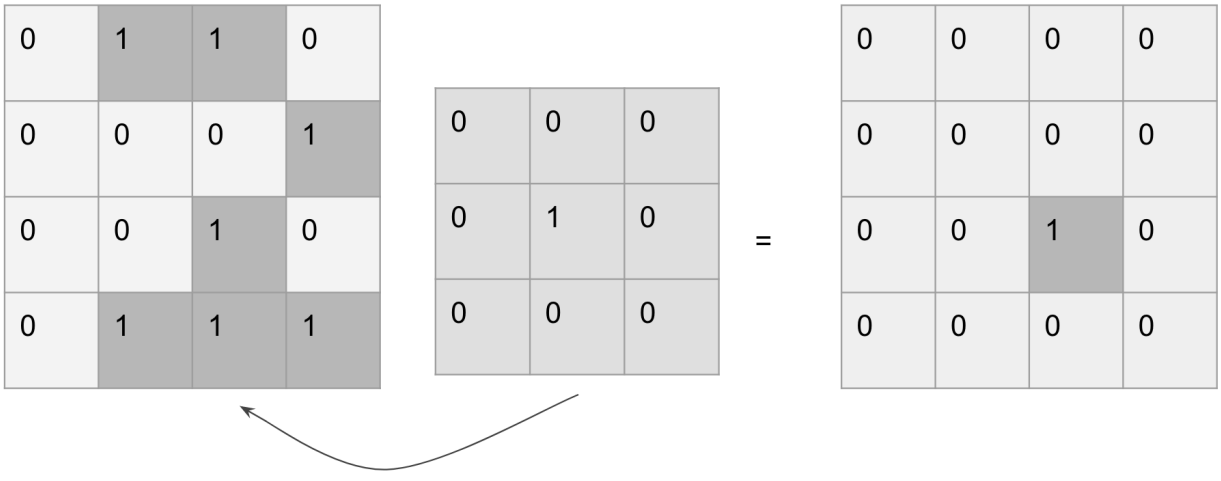
0	1	1	0
0	0	0	1
0	0	1	0
0	1	1	1

Це просте бінарне зображення числа 2.

Для отримання певної інформації з цього зображення можна безпосередньо використовувати всі значення пікселів. Але замість цього, для спрощення, можна застосувати до нього локальні фільтри. Для цього визначаємо матрицю менше заданого зображення, яка працює в околі цільового пікселя. Ця матриця називається ядром, приклад наведено нижче:

0	0	0
0	1	0
0	0	0

Операція визначається напочатку накладенням матриці ядра на вихідне зображення, потім взяттям добутку відповідних пікселів і поверненням підсумовування всіх добутків. На наступному малюнку нижня границя 3 x 3 в оригінальному документі накладається на задану матрицю ядра і множаться відповідні значення пікселів з ядра і зображення. Результуюче зображення показано праворуч і є сумою всіх попередніх добутків пікселів:



Ця операція повторюється ковзанням ядра по рядках зображення, а потім по стовпцях зображення. Це може бути реалізовано наступним чином.

```
# design a kernel matrix, here is uniform 5×5
kernel = np.ones((5,5),np.float32)/25

# apply on the input image, here grayscale input
dst = cv2.filter2D(gray,-1,kernel)
```

Однак, як як можна побачити, кутовий піксель буде мати сильний вплив і призведе до меншого розміру зображення, тому що ядро, при накладенні, буде перебувати поза межами зображення. Це призводить до появи чорної області або дірок разом з границею зображення. Для виправлення цього використовуються деякі відомі техніки:

- Padding кутів з постійними значеннями може бути 0 або 255, за замовчуванням OpenCV буде використовувати цю техніку.
- Дзеркальне відображення пікселя по краю в зовнішню область.
- Створення шаблону пікселів навколо зображення

Вибір їх буде залежати від поставленого завдання. У звичайних випадках Padding може дати задовільні результати.

Ефект від роботи ядра найбільш важливий, тому що зміна цих значень істотно змінює вихідні дані.

Розглянемо напочатку прості фільтри на основі ядра, а також визначимо їх вплив при зміні розміру.

Box фільтрація

Матриця box-фільтрації визначається так:

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

Цей фільтр усереднює значення пікселя. Його застосування призводить до розмивання зображення. Результати представлені в такий спосіб:

При частотному аналізі зображення цей фільтр є низькочастотним фільтром.

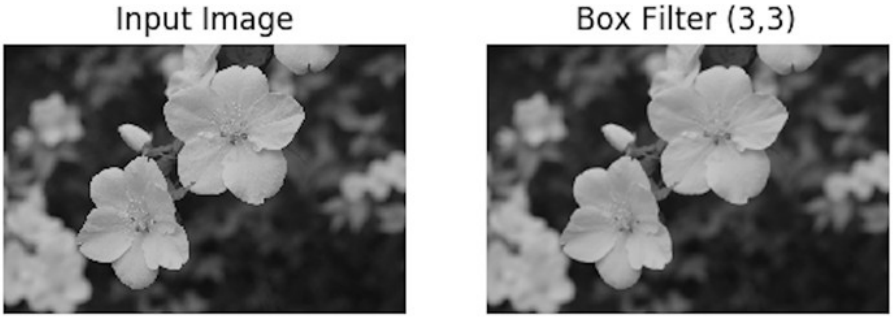
Аналіз частотної області виконується з використанням перетворення Фур'є зображення, що виходить за рамки даної лабораторної роботи.

Можна помітити, що при зміні розміру ядра, зображення стає все більш розмитим:

У міру збільшення розмір ядра результуюче зображення стає більш розмитим. Це пов'язано з усередненням пікових значень в невеликій області, де застосовується ядро.

Результат застосування ядра розміром 20 × 20 можна побачити на

Однак, якщо використовувати фільтр дуже малого розміру (3,3), то ефект на виході буде також дуже малий, так як розмір ядра досить малий у порівнянні з розміром зображення. У більшості додатків розмір ядра вибирається евристично, приймаючи до уваги тільки розмір зображення:



Повний код для генерації зображень, відфільтрованих за допомогою боксів, виглядає так

```
def plot_cv_img(input_image, output_image):
"""
Converts an image from BGR to RGB and plots
"""
fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].imshow(cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB))
ax[0].set_title('Input Image')
ax[0].axis('off')
ax[1].imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
ax[1].set_title('Box Filter (5,5)')
ax[1].axis('off')
plt.show()
def main():
# read an image
img = cv2.imread('../figures/flower.png')
# To try different kernel, change size here.
kernel_size = (5,5)
# opencv has implementation for kernel based box blurring
blur = cv2.blur(img,kernel_size)
# Do plot
plot_cv_img(img, blur)
if __name__ == '__main__':
main()
```

Властивості лінійних фільтрів

В багатьох випадках на в задачах комп'ютерного зору на етпаі попереднього опрацювання зображень потрібно декілька покрокових перетворень вхідних зображень. Це буває достатньо легко зробити завдяки кільком властивостями, пов'язаними з лінійними фільтрами:

- Лінійні фільтри є комутативними, так що можна виконувати операції множення над фільтрами в будь-якому порядку, а результат залишається тим самим:

  a * b = b * a

- Вони асоціативні за своєю природою, тобто порядок застосування фільтра не впливає на результат:

  (a * b) * c = a * (b * c)

- Навіть в разі підсумовування двох фільтрів, можна виконати перше підсумовування, а потім застосувати фільтр, або ж можна застосувати фільтр окремо, а потім підсумувати результати. Загальний результат залишається тим самим.

- Застосування маштабувального коефіцієнта до одного фільтру і множення на інший еквівалентно такій послідовності операцій: множення обох фільтрів, а потім застосування маштабувального коефіцієнта.

Ці властивості відіграють значну роль також в інших завданнях комп'ютерного зору, таких як виявлення (ідентифікація) об'єктів і сегментація. Іноді відповідна комбінація цих фільтрів підвищує якість видобутої інформації і, як результат,

підвищує точність.

Нелінійна фільтрація зображень

Хоча в багатьох випадках лінійні фільтри є достатніми для отримання необхідних результатів, в деяких інших випадках його продуктивність може бути значно збільшена за рахунок використання нелінійної фільтрації зображень.

Нелінійна фільтрація зображень є складніша в практичній реалізації ніж лінійна фільтрація. Однак ця складність може дати кращі результати в завданнях комп'ютерного зору.

Розглянемо, як працює нелінійна фільтрація зображень при застосуванні до різних зображень.

Фільтр Гауса

Застосування box-фільтра з Padding кутів не приводить до плавного розмиття вхідної фотографії. Щоб поліпшити це, фільтр можна зробити більш гладким по краях. Одним з популярних таких фільтрів є фільтр Гаусса. Це нелінійний фільтр, який підсилює ефект центрального пікселя і поступово зменшує його в міру віддалення пікселя від центру. Математично функція Гаусса задається як:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\exp(\frac{(x-\mu)^2}{2\sigma^2})$$

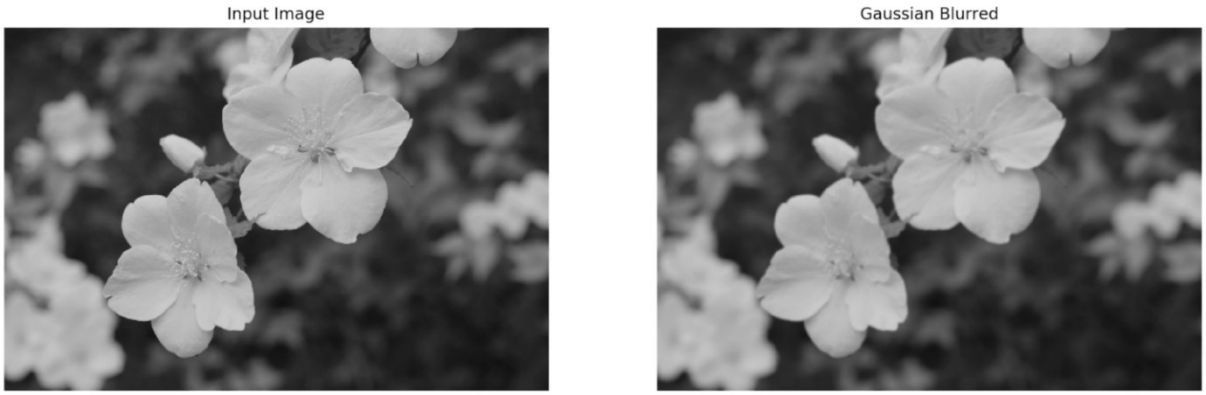
де μ – середнє та σ - дисперсія.

Приклад матриці ядра для такого фільтра в 2D дискретній області наведено нижче:

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Цей 2D масив використовується в нормалізованій формі. Зазначимо, що ефект цього фільтра також залежить від його ширини, так як зміна ширини ядра має різні ефекти на виході.

Застосування гаусового ядра в якості фільтра видаляє високочастотні компоненти, що призводить до видалення сильних країв і, отже, до розмиття фотографії:



Незважаючи на те, що цей фільтр краще розмиває, ніж box-фільтр. Його реалізація в OpenCV є також досить проста:

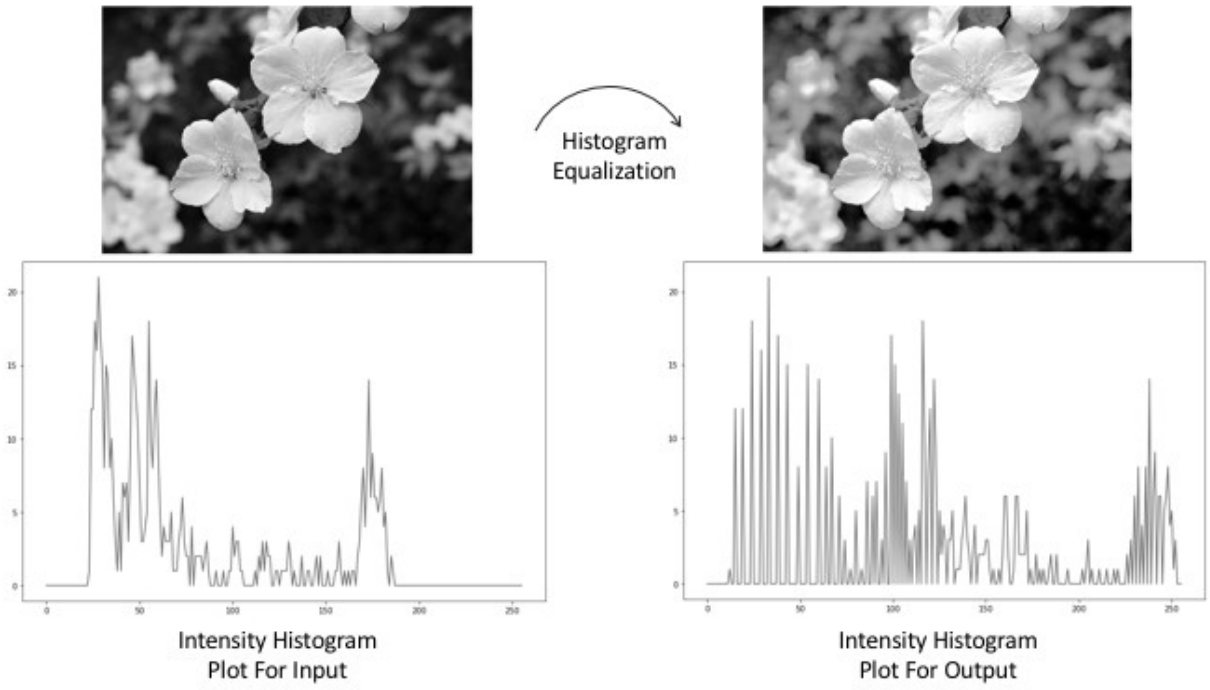
```
def plot_cv_img(input_image, output_image):
"""
Converts an image from BGR to RGB and plots
"""
fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].imshow(cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB))
ax[0].set_title('Input Image')
ax[0].axis('off')
ax[1].imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
ax[1].set_title('Gaussian Blurred')
ax[1].axis('off')
plt.show()
def main():
# read an image
img = cv2.imread('../figures/flower.png')
# apply gaussian blur,
# kernel of size 5x5,
# change here for other sizes
kernel_size = (5,5)
# sigma values are same in both direction
blur = cv2.GaussianBlur(img,(5,5),0)
plot_cv_img(img, blur)
if __name__ == '__main__':
main()
```

Метод вирівнювання гістограм

Основні точкові операції, для зміни яскравості і контрастності, допомагають поліпшити якість зображень, але вимагають ручного налаштування. Використовуючи техніку вирівнювання гістограм, це можна зробити алгоритмічно, тобто автоматизувати процес і створити більш красиве зображення.

Інтуїтивно цей метод намагається встановити для найяскравіших пікселів білий колір, а для більш темних - чорний. Решта значення пікселів аналогічним чином “ремаштабується”. Це маштабування виконується шляхом перетворення вихідного розподілу інтенсивності, щоб відобразити всі розподіли інтенсивності.

Приклад такого вирівнювання є такми:



Зображення зліва є вхідним і є прикладом для вирівнювання гістограм. Справа розташовано оброблене зображення. На ньому контрастність значно збільшується.

Вхідна гістограма показана на нижніх рисунках. Гістограма вхідного зображення (зліва) показує, що на зображенні видно не всі кольори. Після застосування вирівнювання отримує гістограму, графік якої наведено на правому нижньому рисунку.

Для візуалізації результатів вирівнювання на зображенні, вхідні дані і результати підсумовуються на наступному зображенні.

Нижче наведено код для отримання наведених зображень:

```
def plot_gray(input_image, output_image):
"""
Converts an image from BGR to RGB and plots
```

```
"""
# change color channels order for matplotlib
fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].imshow(input_image, cmap='gray')
ax[0].set_title('Input Image')
ax[0].axis('off')
ax[1].imshow(output_image, cmap='gray')
ax[1].set_title('Histogram Equalized ')
ax[1].axis('off')
plt.savefig('../figures/03_histogram_equalized.png')
plt.show()
def main():
# read an image
img = cv2.imread('../figures/flower.png')
# grayscale image is used for equalization
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# following function performs equalization on input image
equ = cv2.equalizeHist(gray)
# for visualizing input and output side by side
plot_gray(gray, equ)
if __name__ == '__main__':
main()
```

Медіанна фільтрація зображень

Медіанна фільтрація зображень аналогічна фільтрації по сусідству. Ключовою технікою тут, звичайно ж, є використання медіанного значення. Таким чином, фільтр є нелінійним. Він досить корисний для видалення флуктуаційних шумів, таких наприклад як сіль і перець.

Замість того, щоб використовувати добуток або суму значень функцій інтесивності пікселів околу, цей фільтр обчислює медіанне значення області. Це призводить до видалення випадкових пікових значень в області, які можуть бути викликані такими шумами, як сіль і перець. Далі це буде показано на прикладі з використанням різних розмірів ядра.

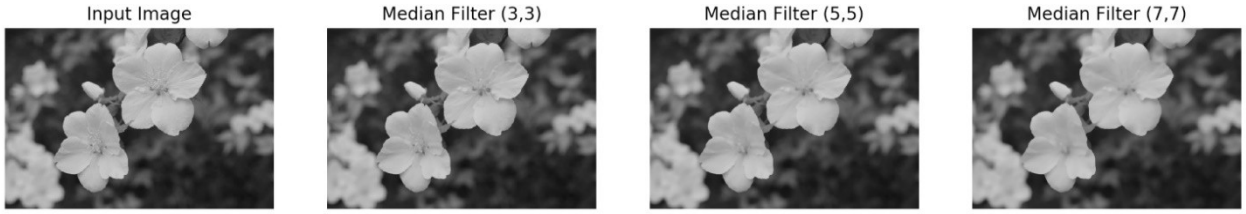
Нехай зображенні на перший вхід додається з канальним випадковим шумом:

```
# read the image
flower = cv2.imread('../figures/flower.png')

# initialize noise image with zeros
noise = np.zeros(flower.shape[:2])

# fill the image with random numbers in given range
cv2.randu(noise, 0, 256)

# add noise to existing image, apply channel wise
noise_factor = 0.1
noisy_flower = np.zeros(flower.shape)

for i in range(flower.shape[2]):

      noisy_flower[:,:,i] = flower[:,:,i] + np.array(noise_factor*noise,
      dtype=np.int)

# convert data type for use
noisy_flower = np.asarray(noisy_flower, dtype=np.uint8)
```

Створене зашумлене зображення використовується для медіанної фільтрації зображень так:

```
# apply median filter of kernel size 5
kernel_5 = 5
median_5 = cv2.medianBlur(noisy_flower,kernel_5)

# apply median filter of kernel size 3
kernel_3 = 3
median_3 = cv2.medianBlur(noisy_flower,kernel_3)
```

На наступному зображенні можна побачити результуюче зображення після зміни розміру ядра (вказано в дужках). Сама права фотографія - найбільш згладжена з усіх.



Найбільш поширеним на сьогодні додатком з медіанним розмиттям є додаток для смартфона, який фільтрує вхідне зображення і додає додаткові артефакти для додавання художніх ефектів.

Код для генерації попередніх зображень є таки:

```
def plot_cv_img(input_image, output_image1, output_image2, output_image3):
"""
Converts an image from BGR to RGB and plots
"""
fig, ax = plt.subplots(nrows=1, ncols=4)
ax[0].imshow(cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB))
ax[0].set_title('Input Image')
ax[0].axis('off')
ax[1].imshow(cv2.cvtColor(output_image1, cv2.COLOR_BGR2RGB))
ax[1].set_title('Median Filter (3,3)')
ax[1].axis('off')
ax[2].imshow(cv2.cvtColor(output_image2, cv2.COLOR_BGR2RGB))
ax[2].set_title('Median Filter (5,5)')
ax[2].axis('off')
ax[3].imshow(cv2.cvtColor(output_image3, cv2.COLOR_BGR2RGB))
ax[3].set_title('Median Filter (7,7)')
ax[3].axis('off')
plt.show()
def main():
# read an image
img = cv2.imread('../figures/flower.png')
# compute median filtered image varying kernel size
median1 = cv2.medianBlur(img,3)
median2 = cv2.medianBlur(img,5)
median3 = cv2.medianBlur(img,7)
# Do plot
```

```
plot_cv_img(img, median1, median2, median3)
if __name__ == '__main__':
main()
```

Детектори границь

Розглянемо найбільш відомі методи виділення границь, а саме фільтри Roberts, Sobel, Prewitt, Kirsch, Canny, Laplacian та способи їх імплементації в cv2.

Градієнти зображення широко використовуються в задачах детектування об'єктів і сегментації. Саме на них будується детектування границь з використанням вказаних фільтрів. У цьому розділі розглянемо, як обчислювати градієнти зображення.

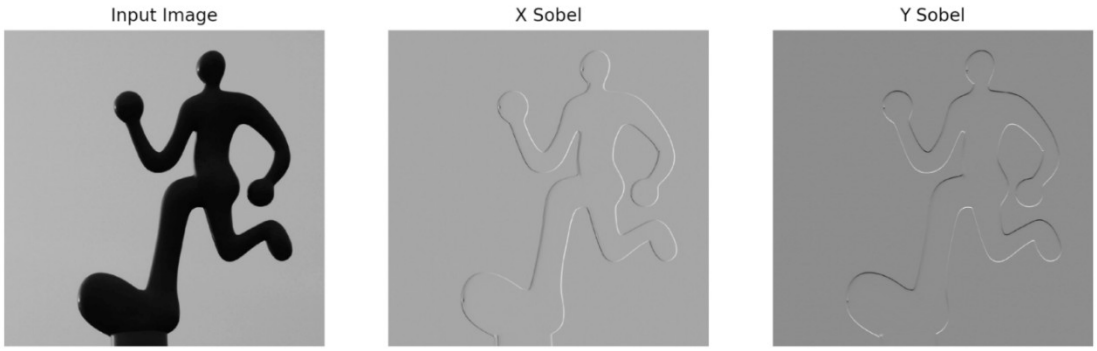
Перша похідна зображення - це застосування матриці ядра, яка обчислює зміну напрямку. дним з таких фільтрів є фільтр Собела, а ядро в x напрямку є таким:

$$\frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

А в y напрямку таким:

$$\frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

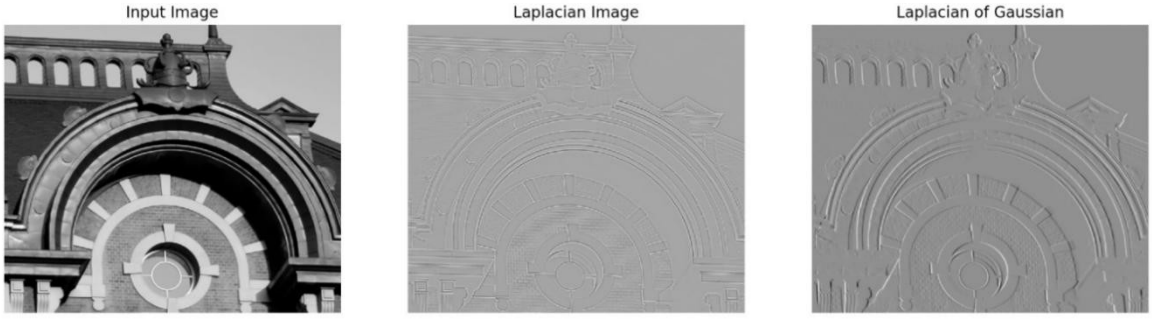
Ці фільтри застосовується аналогічно лінійному box-фільтру шляхом обчислення значень на накладеному ядрі з зображенням. Потім фільтр зміщується вздовж зображення для обчислення всіх значень. Нижче наведено приклад результату, де X і Y позначають напрямок ядра Собела:



Наведені зображення-результати ще також називають похідною зображення по відношенню до заданого напрямку (X або Y). Світліші результуючі фотографії (середня і права) є додатними градієнтами, темніші області визначаються від’ємними, а сірі - нульовими.

У той час як фільтри Собела відповідають похідним першого порядку,

Лапласовий фільтр (лапласіан) дає похідну другого порядку зображення. Лапласовий фільтр застосовується подібно до фільтра Собеля:



Код для отримання наведених фотографій з використанням фільтрів Собела і Лапласа виглядає так:

```
# sobel
x_sobel = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
y_sobel = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

# laplacian
lapl = cv2.Laplacian(img,cv2.CV_64F, ksize=5)

# gaussian blur
blur = cv2.GaussianBlur(img,(5,5),0)

# laplacian of gaussian
log = cv2.Laplacian(blur,cv2.CV_64F, ksize=5)
```

Наведемо використання інших фільтрів для детекції границь:

```
#Custom convolution filtering
import cv2
import numpy as np

image = cv2.imread('./13.jpg',0)
image = cv2.resize(image,(800,800))

#Custom convolution kernel
# Robertsedge operator
kernel_Roberts_x = np.array([
    [1, 0],
    [0, -1]
    ])

kernel_Roberts_y = np.array([
    [0, -1],
    [1, 0]
    ])

# Sobel edge operator
kernel_Sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]])
kernel_Sobel_y = np.array([
    [1, 2, 1],
    [0, 0, 0],
```

```
    [-1, -2, -1]])

# Prewitt edge operator
kernel_Prewitt_x = np.array([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]])
kernel_Prewitt_y = np.array([
    [1, 1, 1],
    [0, 0, 0],
    [-1, -1, -1]])

# Kirsch edge detection operator
def kirsch(image):
    m,n = image.shape
    list=[]
    kirsch = np.zeros((m,n))
    for i in range(2,m-1):
        for j in range(2,n-1):
            d1 = np.square(5 * image[i-1, j-1]+5 * image[i-1, j]+5 * image[i-1, j+1] -
                  3 * image[i, j-1]-3 * image[i, j+1]-3 * image[i+1, j-1] -
                  3 * image[i+1, j]-3 * image[i+1, j+1])
            d2 = np.square((-3) * image[i-1, j-1]+5 * image[i-1, j]+5 * image[i-1, j+1] -
                  3 * image[i, j-1]+5 * image[i, j+1]-3 * image[i+1, j-1] -
                  3 * image[i+1, j]-3 * image[i+1, j+1])
            d3 = np.square((-3) * image[i-1, j-1]-3 * image[i-1, j]+5 * image[i-1, j+1] -
                  3 * image[i, j-1]+5 * image[i, j+1]-3 * image[i+1, j-1] -
                  3 * image[i+1, j]+5 * image[i+1, j+1])
            d4 = np.square((-3) * image[i-1, j-1]-3 * image[i-1, j]-3 * image[i-1, j+1] -
                  3 * image[i, j-1]+5 * image[i, j+1]-3 * image[i+1, j-1] +
                  5 * image[i+1, j]+5 * image[i+1, j+1])
            d5 = np.square((-3) * image[i-1, j-1]-3 * image[i-1, j]-3 * image[i-1, j+1]-3
                  * image[i, j-1]-3 * image[i, j+1]+5 * image[i+1, j-1] +
                  5 * image[i+1, j]+5 * image[i+1, j+1])
            d6 = np.square((-3) * image[i-1, j-1]-3 * image[i-1, j]-3 * image[i-1, j+1] +
                  5 * image[i, j-1]-3 * image[i, j+1]+5 * image[i+1, j-1] +
                  5 * image[i+1, j]-3 * image[i+1, j+1])
            d7 = np.square(5 * image[i-1, j-1]-3 * image[i-1, j]-3 * image[i-1, j+1] +
                  5 * image[i, j-1]-3 * image[i, j+1]+5 * image[i+1, j-1] -
                  3 * image[i+1, j]-3 * image[i+1, j+1])
            d8 = np.square(5 * image[i-1, j-1]+5 * image[i-1, j]-3 * image[i-1, j+1] +
                  5 * image[i, j-1]-3 * image[i, j+1]-3 * image[i+1, j-1] -
                  3 * image[i+1, j]-3 * image[i+1, j+1])

#Take the maximum value in each direction, the effect is not good, use another method
            list=[d1, d2, d3, d4, d5, d6, d7, d8]
            kirsch[i,j]= int(np.sqrt(max(list)))
                        # : Rounding the die length in all directions
            #kirsch[i, j] =int(np.sqrt(d1+d2+d3+d4+d5+d6+d7+d8))
    for i in range(m):
        for j in range(n):
            if kirsch[i,j]>127:
                kirsch[i,j]=255
            else:
                kirsch[i,j]=0
    return kirsch

# CannyEdge Detection k is the Gaussian kernel size, t1, t2 is the threshold size
def Canny(image,k,t1,t2):
    img = cv2.GaussianBlur(image, (k, k), 0)
    canny = cv2.Canny(img, t1, t2)
    return canny

kernel_Laplacian_1 = np.array([
    [0, 1, 0],
    [1, -4, 1],
    [0, 1, 0]])
kernel_Laplacian_2 = np.array([
    [1, 1, 1],
    [1, -8, 1],
    [1, 1, 1]])
```

```
# Two convolution kernels do not have rotation invariance
kernel_Laplacian_3 = np.array([
    [2, -1, 2],
    [-1, -4, -1],
    [2, 1, 2]])
kernel_Laplacian_4 = np.array([
    [-1, 2, -1],
    [2, -4, 2],
    [-1, 2, -1]])

# 5*5 LoG Convolution Template
kernel_LoG = np.array([
    [0, 0, -1, 0, 0],
    [0, -1, -2, -1, 0],
    [-1, -2, 16, -2, -1],
    [0, -1, -2, -1, 0],
    [0, 0, -1, 0, 0]])

# convolution
output_1 = cv2.filter2D(image, -1, kernel_Prewitt_x)
output_2 = cv2.filter2D(image, -1, kernel_Sobel_x)
output_3 = cv2.filter2D(image, -1, kernel_Prewitt_x)
output_4 = cv2.filter2D(image, -1, kernel_Laplacian_1)
output_5 = Canny(image,3,50,150)
output_6 = kirsch(image)

# Show sharpening effect
image = cv2.resize(image, (800, 600))
output_1 = cv2.resize(output_1, (800, 600))
output_2 = cv2.resize(output_2, (800, 600))
output_3 = cv2.resize(output_3, (800, 600))
output_4 = cv2.resize(output_4, (800, 600))
output_5 = cv2.resize(output_5, (800, 600))
output_6 = cv2.resize(output_6, (800, 600))
cv2.imshow('Original Image', image)
cv2.imshow('sharpen_1 Image', output_1)
cv2.imshow('sharpen_2 Image', output_2)
cv2.imshow('sharpen_3 Image', output_3)
cv2.imshow('sharpen_4 Image', output_4)
cv2.imshow('sharpen_5 Image', output_5)
cv2.imshow('sharpen_6 Image', output_6)

if cv2.waitKey(0) & 0xFF == 27:
    cv2.destroyAllWindows()
```

Результати виконнання виділення границі наведено нижче:

original image

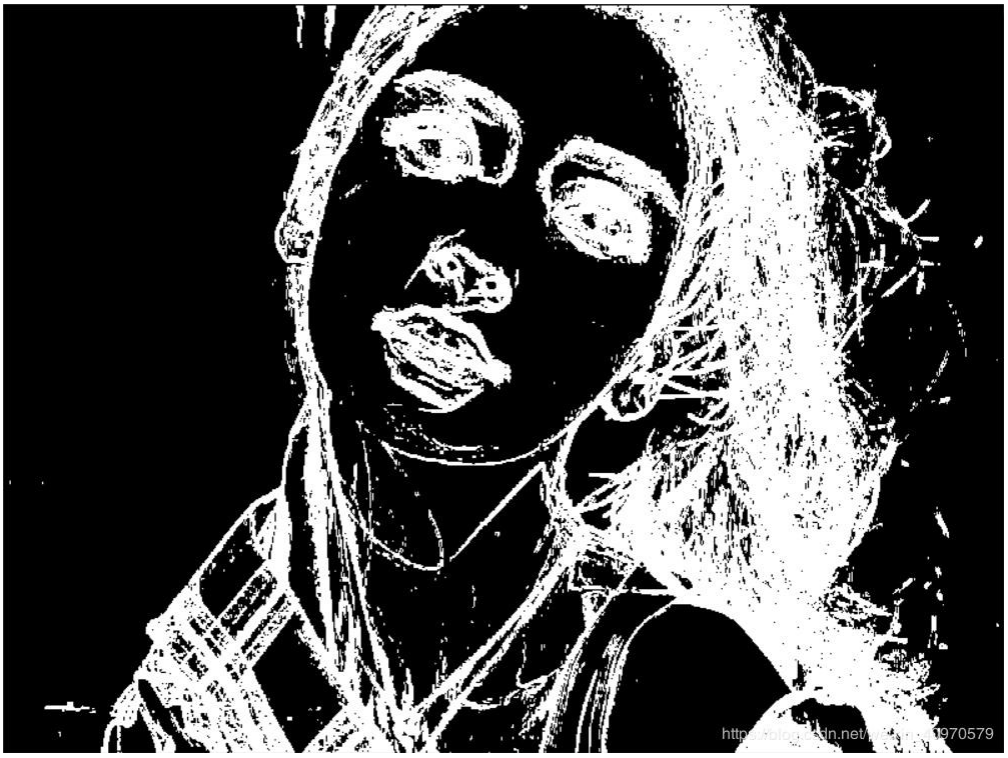
Roberts operator in the x-axis direction

Sobel operator in the x-axis direction

Prewitt operator in the x-axis direction

Laplacian operator in the x-axis direction

Canny edge detection


Kirsch edge detection operator

Завдання:

Вибрати з інтернету два зображення з різною деталізацією об’єктів та два зображення з різним контрастом. Без використання жодних бібліотек для обробки зображень (наприклад Open CV), виконати відповідне завдання (номер завдання вказано у рейтинговій таблиці)

1. Виконати 1D лінійну фільтрацію зображення з різними значеннми K і L. Провести порівняльний аналіз
2. Виконати 2D лінійну фільтрацію зображення з різними значеннями ядра. Провести порівняльний аналіз
3. Виконати box-фільтрацію зображення з різними значеннями ядра. Провести порівняльний аналіз
4. Виконати медіанну фільтрацію зображення з різними значеннями ядра. Провести порівняльний аналіз
5. Виконати фільтрацію Гауса з різними значеннями параметрами ядра розмиття. Провести порівняльний аналіз
6. Виконати гістограмний зсув ліворуч(див. лекція №1). Провести порівняльний аналіз
7. Виконати гістограмний зсув праворуч (див. лекція №1). Провести порівняльний аналіз
8. Виконати гістограмне збільшення контрастності (див. лекція №1). Провести порівняльний аналіз
9. Виконати гістограмне зменшення контрастності (див. лекція №1). Провести порівняльний аналіз
10. Виконати гістограмне збільшення гамми (див. лекція №1). Провести порівняльний аналіз
11. Виконати гістограмне зменшення гамми (див. лекція №1). Провести порівняльний аналіз
12. Виконати детекцію границь на зображеннях за допомогою операторів Roberts, Sobel. Провести порівняльний аналіз
13. Виконати детекцію границь на зображеннях за допомогою операторів Sobel, Prewitt. Провести порівняльний аналіз.
14. Виконати детекцію границь на зображеннях за допомогою операторів Canny. Провести порівняльний аналіз.
15. Виконати детекцію границь на зображеннях за допомогою операторів Kirsch, Laplacian. Провести порівняльний аналіз.

Сторонні бібліотеки для обробки зображень можна використовувати лише для виводу зображень на екран.
Допускається використання таких мов: Python, C++, Java, C#, Matlab.

Додаткове завдання (оцінюється у додаткові бали) - вибрати один із методів морфолологічної обробки (див. Леція №2) і написати приклад його застосування без використання сторонніх біблотек.

Вартість додаткових балів - 2 бали, які зараховуються безпосередньо до сумарної оцінки (іспит + практика), за умови що сумарна оцінка не перевищує 100 балів