DIGITAL IMAGE PROCESSING COURSE - 505060
PRACTICE LABS

LAB 05. IMAGE FILTERING AND SMOOTHING

Requirements

(1) Follow the instructions with the help from your instructor.
(2) Finish all the exercises in class and do the homework at home. You can update your solutions after class and re-submit all your work together with the homework.
(3) Grading
Total score = 50% * Attendance + 50% * Exercises
Rules:
- If the number of finished exercises is less than **80% total number of excercises**, you will get **zero** for the lab.
- Name a source file as "**src_XX.py**" where XX is the exercise number, for ex., "src_03.py" is the source code for the Exercise 3.
- Add the text of your Student ID to each of the output image.
- Name an output image as "**image_XX_YY.png**" where XX is the exercise number and YY is the order of output images in the exercise, for ex., "image_03_02.png" is the second output image in the Exercise 3.
- Submit the source code and output image files directly to Google classroom assignment, donot compress the files.

If you submit the exercises with wrong rules, you will get **zero** for the lab or the corresponding exercises.

(4) Plagiarism check
If any 2 of the students have the same output images, then all will get zero for the corresponding exercises.

INTRODUCTION

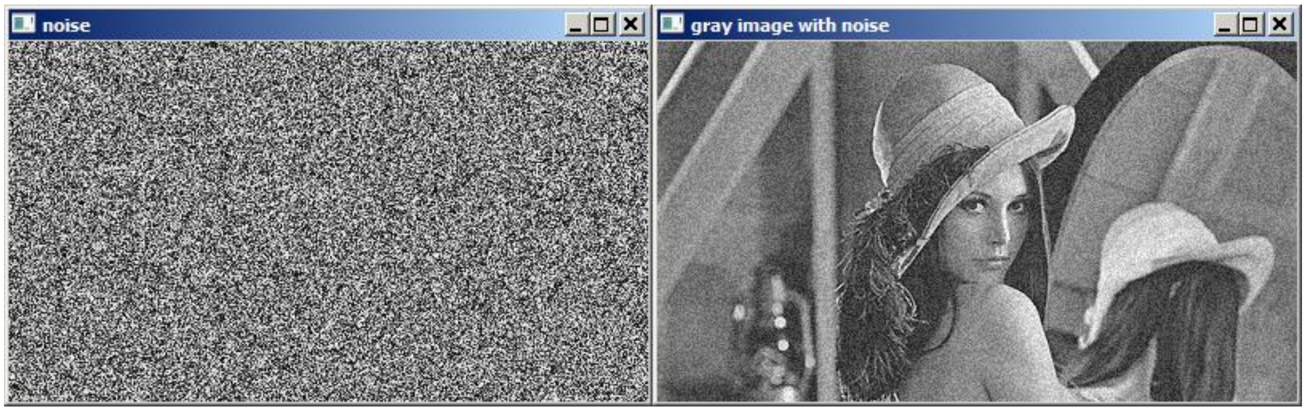
In this Lab, you will learn how to
- Add noise to an image
- Image filtering applied in image smoothing/sharpening
- Histogram equalization image contrast enhancement
- Motion estimation in videos

INSTRUCTIONS

1. Image noise

Here's an example with considerable salt and pepper noise. This occurs when there is a disturbance in the quality of the signal that's used to generate the image.

```python
import cv2
import numpy as np

gray_img = cv2.imread(r"C:\Users\THC\Desktop\XLA\Src\lena.jpg", 0)

# initialize noise image with zeros
noise_img = np.zeros(gray_img.shape[:2])
# fill the image with random numbers in given range
cv2.randu(noise_img, 0, 256)
noise_img = np.array(noise_img, dtype=np.uint8)

# add noise to existing image
noisy_gray = cv2.add(gray_img, np.array(0.2*noise_img, dtype=np.uint8) )

cv2.imshow("noise", noise_img)
cv2.imshow("gray image with noise", noisy_gray)
```



For color images, refer: https://www.programcreek.com/python/example/89357/cv2.randn

2. Image Filtering Using Convolution in OpenCV

```
filter2D(src, ddepth, kernel)
```

The filter2D() function requires three input arguments:

- The first argument is the source image
- The second argument is ***ddepth***, which indicates the depth of the resulting image. A value of -1 indicates that the final image will also have the same depth as the source image
- The final input argument is the ***kernel***, which we apply to the source image

```python
image = cv2.imread('test.jpg')
"""
Apply identity kernel
"""
kernel1 = np.array([[0, 0, 0],
                    [0, 1, 0],
                    [0, 0, 0]])
# filter2D() function can be used to apply kernel to an image.
# Where ddepth is the desired depth of final image. ddepth is -1 if...
# ... depth is same as original or source image.
identity = cv2.filter2D(src=image, ddepth=-1, kernel=kernel1)

# We should get the same image
cv2.imshow('Original', image)
cv2.imshow('Identity', identity)

cv2.waitKey()
cv2.imwrite('identity.jpg', identity)
cv2.destroyAllWindows()
```

Link: https://learnopencv.com/image-filtering-using-convolution-in-opencv/#apply-id-kernel

Blurring an Image using a Custom 2D-Convolution Kernel

Next, we will demonstrate how to blur an image. Here too, we will define a custom kernel, and use the filter2D() function in OpenCV to apply the filtering operation on the source image.

```python
"""
Apply blurring kernel
"""
kernel2 = np.ones((5, 5), np.float32) / 25
img = cv2.filter2D(src=image, ddepth=-1, kernel=kernel2)

cv2.imshow('Original', image)
cv2.imshow('Kernel Blur', img)

cv2.waitKey()
cv2.imwrite('blur_kernel.jpg', img)
cv2.destroyAllWindows()
```

Creating our own filter

As we have already seen OpenCV provides a function `cv2.filter2D()` to process our image with some arbitrary filter. So, let's create our custom filter. We just want to emphasize here that this filter will be a matrix usually of a size 3×3 or 5×5

By setting coefficient filter values we can perform different kind of filtering. For instance, we have added a following numbers and we will process our image with a random filter.

```python
filter_2 = np.array([[3, -2, -3], [-4, 8, -6], [5, -1, -0]])
```

```python
# Creating our arbitrary filter
filter = np.array([[3, -2, -3], [-4, 8, -6], [5, -1, -0]])
```

```
# Applying cv2.filter2D on our image
custom_img_1=cv2.filter2D(img1,-1,filter)
cv2_imshow(custom_img_1)
```

3. Image Blurring (Image Smoothing)

a. Averaging image using built-in functions in OpenCV

This is done by convolving image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replace the central element. This is done by the function cv.blur() or cv.boxFilter().

```
cv.blur (src, dst, ksize, anchor = new cv.Point(-1, -1), borderType =
cv.BORDER_DEFAULT)

cv.boxFilter (src, dst, ddepth, ksize, anchor = new cv.Point(-1, -1),
normalize = true, borderType = cv.BORDER_DEFAULT)
```

```
"""
Apply blur using `blur()` function
"""
img_blur = cv2.blur(src=image, ksize=(5,5))
image where ksize is the kernel size

# Display using cv2.imshow()
cv2.imshow('Original', image)
cv2.imshow('Blurred', img_blur)

cv2.waitKey()
cv2.imwrite('blur.jpg', img_blur)
cv2.destroyAllWindows()
```

b. Gaussian Blurring

```
GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]])
```

The GaussianBlur() function requires four input arguments:

- The first argument, src, specifies the source image that you want to filter.
- The second argument is ksize, which defines the size of the Gaussian kernel. Here, we are using a 5×5 kernel.
- The final two arguments are sigmaX and sigmaY, which are both set to 0. These are the Gaussian kernel standard deviations, in the X (horizontal) and Y (vertical) direction. The default setting of sigmaY is zero. If you simply set sigmaX to zero, then the standard deviations are computed from the kernel size (width and height respectively). You can also explicitly set the size of each argument to positive values greater than zero.

```python
"""
Apply Gaussian blur
"""
# sigmaX is Gaussian Kernel standard deviation
# ksize is kernel size
gaussian_blur = cv2.GaussianBlur(src=image, ksize=(5,5), \\
sigmaX=0, sigmaY=0)

cv2.imshow('Original', image)
cv2.imshow('Gaussian Blurred', gaussian_blur)

cv2.waitKey()
cv2.imwrite('gaussian_blur.jpg', gaussian_blur)
cv2.destroyAllWindows()
```

c. Median Blurring

We can also apply median blurring, using the medianBlur() function in OpenCV. In median blurring, each pixel in the source image is replaced by the median value of the image pixels in the kernel area.

```
medianBlur(src, ksize)
```

This function has just two required arguments:

- The first is the source image.
- The second is the kernel size, which must be an odd, positive integer.

```python
"""
Apply Median blur
"""
# medianBlur() is used to apply Median blur to image
# ksize is the kernel size
median = cv2.medianBlur(src=image, ksize=5)

cv2.imshow('Original', image)
cv2.imshow('Median Blurred', median)

cv2.waitKey()
cv2.imwrite('median_blur.jpg', median)
cv2.destroyAllWindows()
```

d. Bilateral Filtering

Bilateral filtering essentially applies a 2D Gaussian (weighted) blur to the image, while also considering the variation in intensities of neighboring pixels to minimize the blurring near edges (which we wish to preserve). What this means is that the shape of the kernel actually depends on the local image content, at every pixel location.

OpenCV provides the bilateralFilter() function to filter images.

```
bilateralFilter(src, d, sigmaColor, sigmaSpace)
```

This function has four required arguments:

- The first argument of the function is the source image.
- The next argument d, defines the diameter of the pixel neighborhood used for filtering.
- The next two arguments, sigmaColor and sigmaSpace define the standard deviation of the (1D) color-intensity distribution and (2D) spatial distribution respectively.
    - The sigmaSpace parameter defines the spatial extent of the kernel, in both the x and y directions (just like the Gaussian blur filter previously described).
    - The sigmaColor parameter defines the one-dimensional Gaussian distribution, which specifies the degree to which differences in pixel intensity can be tolerated.

The final (weighted) value for a pixel in the filtered image is a product of its spatial and intensity weight. Thus,

- pixels that are similar and near the filtered pixel will have influence
- pixels that are far away from the filtered pixel will have little influence (due to the spatial Gaussian)
- pixels that have dissimilar intensities will have little influence (due to the color-intensity Gaussian), even if they are close to the center of the kernel.

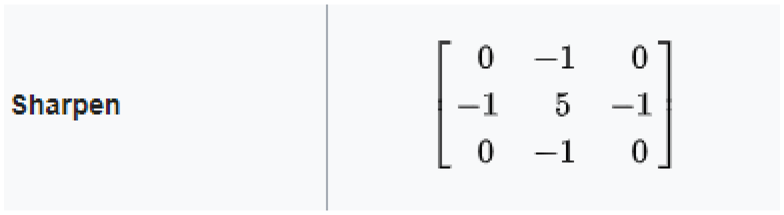
```
"""
Apply Bilateral Filtering
"""
# Using the function bilateralFilter() where d is diameter of each...
# ...pixel neighborhood that is used during filtering.
# sigmaColor is used to filter sigma in the color space.
# sigmaSpace is used to filter sigma in the coordinate space.
bilateral_filter = cv2.bilateralFilter(src=image, d=9, sigmaColor=75,
sigmaSpace=75)

cv2.imshow('Original', image)
cv2.imshow('Bilateral Filtering', bilateral_filter)

cv2.waitKey(0)
cv2.imwrite('bilateral_filtering.jpg', bilateral_filter)
cv2.destroyAllWindows()
```

4. Image Sharpening

One simple approach is to perform what is known as unsharp masking, where an unsharp, or smoothed, version of an image is subtracted from the original image. In the following example, a Gaussian smoothing filter has been applied first and the resulting image is subtracted from the original image:

```
smoothed = cv2.GaussianBlur(img, (9, 9), 10)
unsharped = cv2.addWeighted(img, 1.5, smoothed, -0.5, 0)
```

Another option is to use a specific kernel for sharpening edges and then apply the cv2.***filter2D***() function.



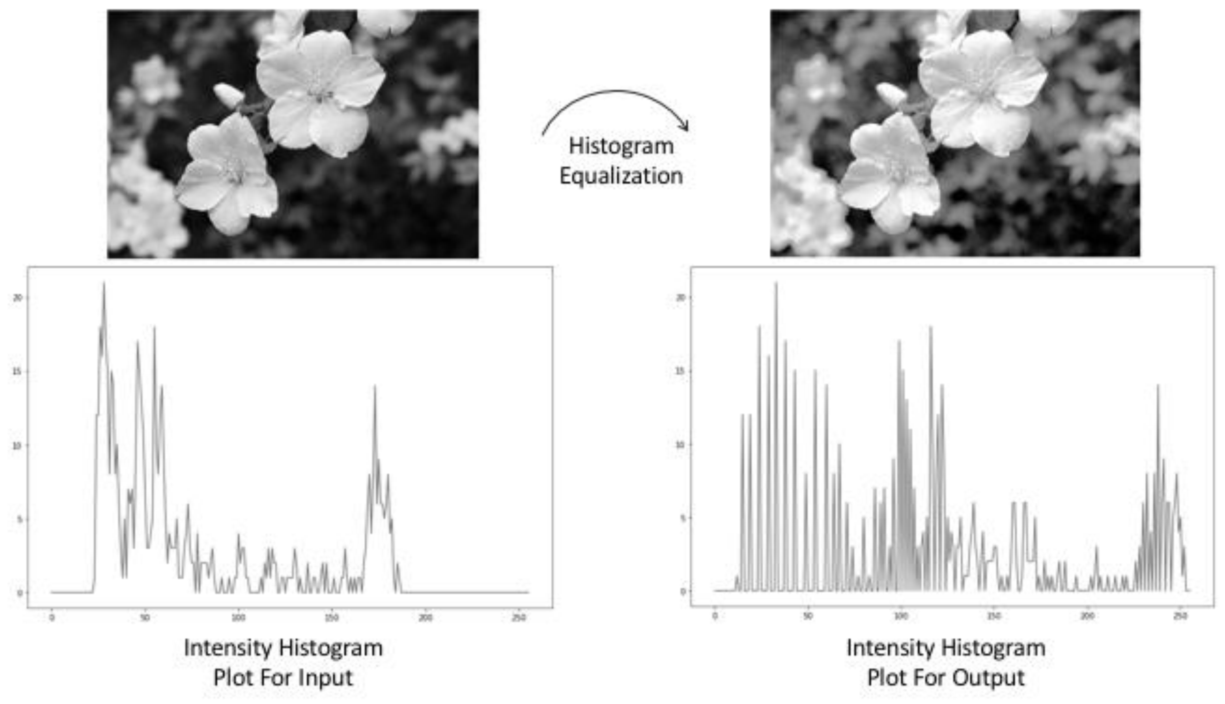
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Source: Wikipedia.

```
# load the required packages

import cv2
import numpy as np

# load the image into system memory

image = cv2.imread('images.jpg', flags=cv2.IMREAD_COLOR)

# display the image to the screen

cv2.imshow('AV CV- Winter Wonder', image)

kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

image_sharp = cv2.filter2D(src=image, ddepth=-1, kernel=kernel)

cv2.imshow('AV CV- Winter Wonder Sharpened', image_sharp)
cv2.waitKey()
cv2.destroyAllWindows()
```

5. The histogram equalization technique

The basic point operations, to change the brightness and contrast, help in improving photo quality but require manual tuning. Using histogram equalization technique, these can be found algorithmically and create a better-looking photo. Intuitively, this method tries to set the brightest pixels to white and the darker pixels to black. The remaining pixel values are similarly rescaled. This rescaling is performed by transforming original intensity distribution to capture all intensity distribution. An example of this equalization is as following:



```
# 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
```

EXERCISE

The image used for the exercise may be downloaded from this link or saved from below.

Ex5.1. Image histogram equalization

1. Show image histogram (gray)
2. Enhance image contrast by histogram equalization

   Hint: Image histogram link.

Ex5.2. Image blurring

3. Add some noise to the input image
4. Blur the image by using at least 05 different techniques (five outputs)

Ex5.3. Image sharpening

Sharpen the input image by using at least 02 different techniques.
Hints: using *addWeighted* or filter2D function

Download: https://learnopencv.com/wp-content/uploads/2021/06/input_image.jpg

Ex5.4. Simple Motion Estimation in Videos using OpenCV

Extract motion from a video.
https://learnopencv.com/simple-background-estimation-in-videos-using-opencv-c-python/

Hint: (Motion = current frame (image) - background (median frame))

1. Calculate a median frame (background estimation)
2. Convert the median frame to grayscale.
3. Loop over all frames in the video.
   Extract the current frame and convert it to grayscale.
4. Calculate the absolute difference between the current frame and the median frame. (Frame differencing)
5. Threshold the above image to remove noise and binarize the output.

HOMEWORK

Background estimation using background subtraction techniques.
https://www.geeksforgeeks.org/background-subtraction-opencv/