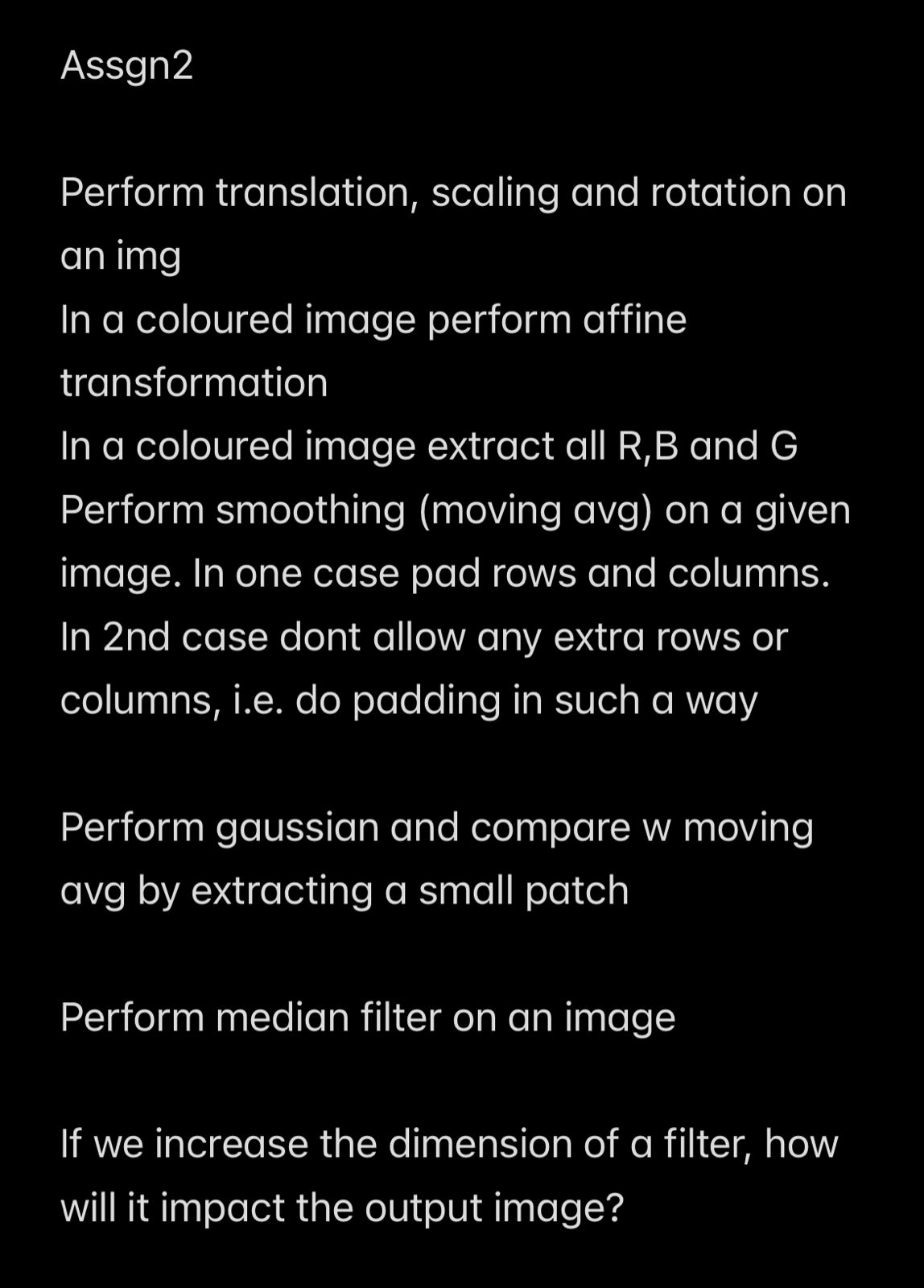


Assignment -2

Name – Akhil Dubey

Roll – 2020UCO1673

Branch – COE-3



**Original Image**



import cv2

import numpy as np

# Load the image

img = cv2.imread('input.jpg')

# Translation: shift the image 200 pixels to the right and 100 pixels up

M = np.float32([[1, 0, 200], [0, 1, -100]])

rows, cols = img.shape[:2]

translated\_img = cv2.warpAffine(img, M, (cols, rows))

# Rotation: rotate the image by 45 degrees counterclockwise

M = cv2.getRotationMatrix2D((cols/2, rows/2), -45, 1)

rotated\_img = cv2.warpAffine(img, M, (cols, rows))

# Scaling: scale the image by a factor of 2 along both axes

M = np.float32([[2, 0, 0], [0, 2, 0]])

scaled\_img = cv2.warpAffine(img, M, (2 \* cols, 2 \* rows))

# Display the original, translated, rotated, and scaled images

cv2.imshow('Original', img)

cv2.imshow('Translated', translated\_img)

cv2.imshow('Rotated', rotated\_img)

cv2.imshow('Scaled', scaled\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Translated Image**



**Rotated**

****

**Scaled**

****

#correct

# Define the affine transformation matrix

rows, cols = img.shape[:2]

affine\_matrix = np.float32([[1, 0.5, 0], [0.5, 1, 0]])

# Perform the affine transformation

result = cv2.warpAffine(img, affine\_matrix, (cols, rows))

# Display the original image and the transformed image

cv2.imshow('Original', img)

cv2.imshow('Transformed', result)

cv2.waitKey(0)

cv2.destroyAllWindows()



#correct

# Split the image into its R, G, and B channels

b, g, r = cv2.split(img)

# Display the R, G, and B channels of the image

cv2.imshow('Red', r)

cv2.imshow('Green', g)

cv2.imshow('Blue', b)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Red**







def moving\_average(image, ksize, padding=False):

    ksize = (ksize, ksize)

    if padding:

        image = cv2.copyMakeBorder(image, ksize[0] // 2, ksize[0] // 2, ksize[1] // 2, ksize[1] // 2, cv2.BORDER\_REPLICATE)

    return cv2.blur(image, ksize)

# Apply moving average with and without padding

result\_no\_padding = moving\_average(img, ksize=15, padding=False)

result\_with\_padding = moving\_average(img, ksize=15, padding=True)

# Display the results

cv2.imshow("Original", img)

cv2.imshow("Result (no padding)", result\_no\_padding)

cv2.imshow("Result (with padding)", result\_with\_padding)

cv2.waitKey(0)

***With Padding***



***Without Padding***



#correct

# Define the kernel size for the smoothing operations

kernel\_size = (5, 5)

# Pad the image with extra rows and columns

img\_padded = cv2.copyMakeBorder(img, kernel\_size[0]//2, kernel\_size[0]//2, kernel\_size[1]//2, kernel\_size[1]//2, cv2.BORDER\_REPLICATE)

# Extract a small patch from the padded image

patch = img\_padded[50:150, 50:150, :]

# Perform moving average smoothing with padding

smooth\_padded\_ma = cv2.blur(patch, kernel\_size)

# Perform Gaussian smoothing with padding

smooth\_padded\_gaussian = cv2.GaussianBlur(patch, kernel\_size, 0)

# Display the original patch, the smoothed patch with moving average, and the smoothed patch with Gaussian smoothing

cv2.imshow('Original', patch)

cv2.imshow('Smoothed with Moving Average', smooth\_padded\_ma)

cv2.imshow('Smoothed with Gaussian', smooth\_padded\_gaussian)

cv2.waitKey(0)

cv2.destroyAllWindows()





# Define the kernel size for the median filtering operation

kernel\_size = 3

# Perform median filtering

smooth\_median = cv2.medianBlur(img, kernel\_size)

# Display the original image and the smoothed image with median filtering

cv2.imshow('Original', img)

cv2.imshow('Smoothed with Median Filter', smooth\_median)

cv2.waitKey(0)

cv2.destroyAllWindows()



# Apply median filter

result = cv2.medianBlur(img, ksize=15)

# Display the results

cv2.imshow("Original", img)

cv2.imshow("Result", result)

cv2.waitKey(0)

cv2.destroyAllWindows()



Increasing the dimension of a filter has the following impact on the output image:

1. Smoothing effect: Larger filters tend to produce a smoother output image compared to smaller filters, as they average the intensity values of more pixels in the surrounding neighborhood. This can help to reduce noise and other small-scale details in the image.
2. Computational cost: Larger filters require more computation and processing time compared to smaller filters. This can slow down the overall processing time of an image and may not be feasible for real-time applications.
3. Blurring effect: As the filter size increases, the output image will become more blurred, as more pixels are averaged and their intensity values are spread out over a larger area. This can lead to the loss of important fine details in the image, such as edges and textures.
4. Border effect: Larger filters may result in border effects, such as edge ringing and border smoothing. This is because the filter will average the intensity values of pixels from both inside and outside the image, which can cause smoothing and other artifacts along the borders.

In general, it is important to carefully consider the trade-off between the smoothing effect and the blurring effect when choosing the size of a filter. In some cases, using multiple filters of different sizes and combining their outputs may be a more effective approach to achieving the desired smoothing effect while preserving important details in the image.