Name: Vanessa Giovani

NIM: 2301887566

Course: Computer Vision

Class Code: LA02

I. Study Case

1) Image Filtering

```
figure_size = 9

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
grayFilter = cv2.medianBlur(gray, figure_size)
plt.figure(figsize=(11,6))

plt.subplot(121), plt.imshow(gray, cmap='gray'),plt.title('Before Median Filter')
plt.xticks([]), plt.yticks([])

plt.subplot(122), plt.imshow(grayFilter, cmap='gray'),plt.title('After Median Filter')
plt.xticks([]), plt.yticks([])
plt.show()
```

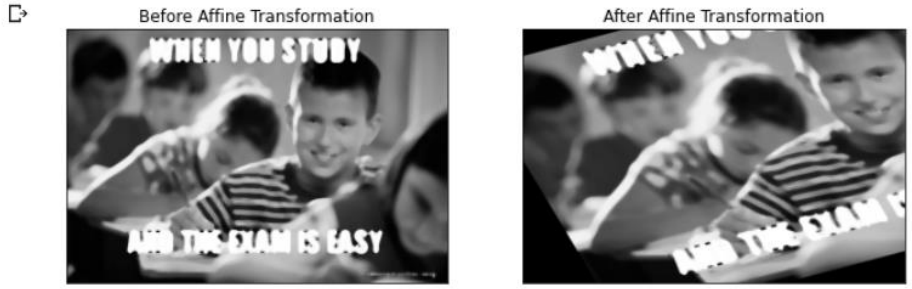

To begin with, I use median filter with the application of turning the sample picture into grayscale first to improve its result after filtered. The median filter calculates the pixel intensities' median who gathers in the center pixel in a nxn kernel. Why median filter? Not only it performs better than Gaussian filter because of its better salt-and-pepper noise elimination and not producing artifacts on a color image. Using medianBlur() as the median filter function with 9x9 median filter on the sample picture, its results proves that some of the salt-and-pepper noise are removed and the edge side still retains.

The process of median filter starts with looking at the nearby neighbours from each pixel in the image to determine the representative of the using value. Later on, the pixel value who becomes the chosen one is replaced with median of the neighbourhood values. The median value of the pixel is calculated from sorting of the pixel values (from surrounding neighbourhood into numerical order) and replacing the pixel value with the midel pixel value.

2) Affine Transformation

I apply two different picture sample: with median filter and without median filter

```
# affine transformation using grayscale image with filter
rows, cols = grayFilter.shape

one = numpy.float32([[50,50], [200,50], [50,200]])
two = numpy.float32([[10,100], [200,50], [100,250]])

# translation
Matrix = cv2.getAffineTransform(one, two)

# rotation
rotate = cv2.warpAffine(grayFilter, Matrix, (cols, rows))

plt.figure(figsize=(11,6))
plt.subplot(121), plt.imshow(grayFilter, cmap='gray'),plt.title('Before Affine Transformation')
plt.xticks([]), plt.yticks([])

plt.subplot(122), plt.imshow(rotate, cmap='gray'),plt.title('After Affine Transformation')
plt.xticks([]), plt.yticks([])
plt.show()
```




```
[50] # affine transformation using grayscale image without filter
rows, cols = gray.shape

one = numpy.float32([[50,50], [200,50], [50,200]])
two = numpy.float32([[10,100], [200,50], [100,250]])

# translation
Matrix = cv2.getAffineTransform(one, two)

# rotation
rotate = cv2.warpAffine(gray, Matrix, (cols, rows))

plt.figure(figsize=(11,6))
plt.subplot(121), plt.imshow(gray, cmap='gray'),plt.title('Before Affine Transformation')
plt.xticks([]), plt.yticks([])

plt.subplot(122), plt.imshow(rotate, cmap='gray'),plt.title('After Affine Transformation')
plt.xticks([]), plt.yticks([])
plt.show()
```

In this case, I apply translations and one of the linear transformations which is rotation. First step, we first shape the sample image to obtain its width and height. Next thing is we create two transformation matrixes with its x and y axis inside and consist of the information needed to image shifting. We use warpAffine() function where its arguments contain the gray image, Matrix, and the output image size to rotate the image in the desired angle (center point).

The matrix form that was used in the program:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Affine transformations represent 2-vector of (x, y) as a 3-vector adding 1 like (x, y, 1). Translation is a way to translate the vector by tx along the x axis and ty along the y axis to get a new vector while rotation works by doing a rotation matrix with the process of the vector extending from the starting point. To illustrate, a circle with the radius of 1 has a triangle inside where it shows a component of cosΘ and sinΘ. We can calculate its components with the matrix of figure (1) and later on can be simplified into figure (2).

$$\mathrm{v} = \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \end{bmatrix} = \begin{bmatrix} \mathrm{r}\cos(\theta) \\ \mathrm{r}\sin(\theta) \end{bmatrix} \qquad \mathrm{v} = \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \end{bmatrix} = \mathrm{x}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mathrm{y}\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

(1) (2)

Through the basis of the triangle, we define a vector of $\mathrm{b_x} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$

If we do rotation, then the new basis would be $\mathrm{b_y} = \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$

Therefore, we combine it together and formed

$$\mathrm{x}\begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + \mathrm{y}\begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \end{bmatrix}$$

Because this is 2x2 matrix, in the affine transformations, thus we add the matrix into 3x3 and conclude the final form as:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ 1 \end{bmatrix}$$

3) Image Matching

I apply ORB detector algorithm and Brute-Force matcher to complete the program result. Why ORB? ORB has fast and accurate orientation than FAST and has variance and correlation of BRIEF. Not only that, ORB's performance is as better as SIFT and SURF. The implementation has several steps which are take the query image and convert it into grayscale, initialize the ORB detector and detect the keypoints, compute the descriptors belonging to both the images, match keypoints using Brute Force Matcher, and last is show the image.

```
[52] # add image path and name
     name = 'sample2.jpg'
     url = "https://p14cdn4static.sharpschool.com/UserFiles/Servers/Server_9177

     # retrieve image
     urllib.request.urlretrieve(url, name)
     image2 = mpimg.imread(name)

     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
     image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
```

To start with, I will explain the code implementation by starting from loading the images needed for the matching. Next step, I initiate the ORB feature descriptor -> ORB_create()

```
[67] # use feature descriptor
     orb = cv2.ORB_create()

     # detect and compute keypoints
     keypoint_image, descriptor_image = orb.detectAndCompute(image, None)
     keypoint_scene, descriptor_scene = orb.detectAndCompute(image2, None)
```

Then we detect and compute the keypoints for the image and image2 and there we got keypoint_image, keypoint_scene, descriptor_image, and descriptor_scene.

```
[85] # initialize matcher
     matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
     matches = matcher.match(descriptor_image, descriptor_scene)

     matches = sorted(matches, key = lambda x:x.distance)
```

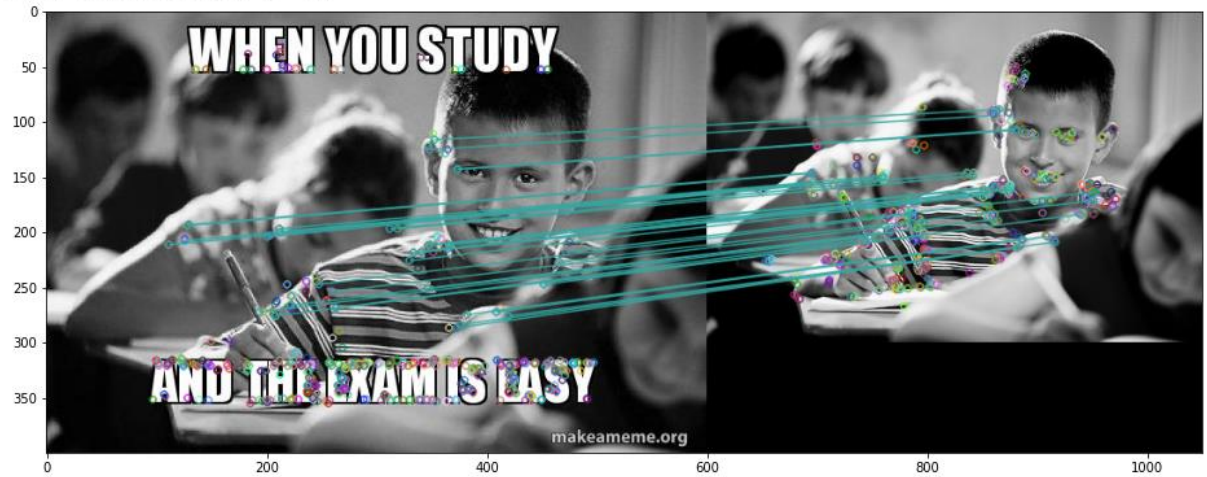
I use norm hamming to have a better computation result by measuring using Hamming distance, and the second parameter next to the NORM_HAMMING is using crossCheck, it is a Boolean variable where I apply True which only those matches with the true value can be returned. The matches then will be sorted in the order of their distance using key = lambda x:x distance.

```
#draw matches
#param 1: img gray
#param 2: img keypoint
#param 3: scene gray
#param 4: scene keypoint
#param 5: matches
#param 6: none (belom dibuat)
#param 7: match color
#param 8: matches mask (none)

result = cv2.drawMatches(
    image,
    keypoint_image,
    image2,
    keypoint_image2,
    matches[:35],
    None,
    matchColor = [50, 168, 164],
)
```

Later on, we determine the result by using drawMatches() where it can stack two images horizontally and draw lines between the images to display the matches. Param 5 is the number of matches we draw in the first 35 matches.

```
print("Number of detected points >> ", len(keypoint_image))

plt.rcParams['figure.figsize'] = [16,9]
plt.imshow(result)
plt.show()
```

```
Number of detected points >>  500
```




In conclusion, the result above displays that there are 500 detected points on the image and the lines between the image and the scene are both matched.