

ASSIGNMENT 1 – IMAGE MATCHING

IMAGE MATCHING

In image matching we want to find the similarity of image by extracting the features of two images and more. Using features from each image we then identify and correspond the same or similar structure / content from those images.

One of the methods to do Image Matching is SIFT, in this assignment we will use SIFT as our feature extractor so we could get what features are similar between two images. The Scale Invariant Feature Transform (SIFT) will extract a set of descriptors from an image. The extracted descriptors are related to image translation, rotation, and scaling. This SIFT descriptor also has advantages to a wide family of image transformations, such as slight changes of viewpoint, noise, blur, contrast changes, and scene deformations.

How It Works Generally

SIFT detects series of key points using multiscale image representation (known as octaves). This multiscale image representation then consists of a group of images that consist of increasingly blurred images. Key points are the same thing as interest points. They are spatial locations, or points in the image that define what is interesting or what stand out in the image.

After we got multiscale images representation with different image blur, we then subtract all the images and get Different of Gaussian (DoG). After get the DoG, then we try to find the local extrema so we could know what are key points of the image. From this DoG on every octave, we also could generate the descriptor. This descriptor is really important to match one image with another. If the descriptor has almost the similar characteristics, then we could decide that the key point of one image is also available in another image.

In this report, I will explain step by step along with the explanation of my code. This is the summary of how SIFT work in general:

Stage	Description
1	Compute- the Image Pyramid (Gaussian Scale-Space) Input: Image Output: Multi-Scale and Multi-Octaves Image
2	Compute-the Difference of Gaussians (DoG) Input: Multi-Scale and Multi-Octaves Image Output: DoG Image result in Every Octaves
3	Find Candidate key points (3D Discrete Extrema of DoG) Input: DoG Image result in Every Octaves Output: (Xd, Yd, Scale) list of candidate key points (position and scale)
4	Refine Candidate Key Points Location with Sub-Pixel Precision Input: DoG Image and (Xd, Yd, Scale) Output: (X, Y, Scale, Orientation in Local Space) list of interpolated extrema

5	Filter unstable Key Points from noise in images Input: DoG Image and (X, Y, Scale, Orientation in Local Space) Output: (X, Y, Scale, Orientation in Local Space) list of filtered key points
6	Filter unstable Key Points in the edges Input: DoG Image and (X, Y, Scale) Output: (X, Y, Scale, Orientation in Local Space) list of filtered key points
7	Convert keypoint and orientation in the input image size Input: (dm,dn) Scale-space gradient and (X, Y, Scale) list of key points Output: (X, Y, Scale, Orientation) list of key points with scale and orientation
8	Build the key points descriptors Input: (dm,dn) Scale-space gradient and (X, Y, Scale, Orientation) Output: {(x, y, orientation, scale,f)} list of described keypoints

All of the steps are summarized in the SIFT function below:

```
# ===== #
#      SIFT      #
# ===== #

def SIFT(image,
        sigma=1.6,
        num_octaves=4,
        num_scale=5,
        assumed_blur=0.5,
        image_border_width=5):
    # ===== #
    # TODO: implement your SIFT function here
    # ===== #

    image_name = os.path.basename(image)
    image = cv2.imread(image, 0)
    image = image.astype('float32')

    firstImage = generateFirstImage(image, sigma, assumed_blur)
    gaussianKernel = generateGaussianSigma(sigma, num_scale)

    gaussian_images = make_pyramid(firstImage, image_name,
    gaussianKernel, num_octaves, num_scale)

    dog_images = make_DOG(image_name, gaussian_images, num_octaves)

    num_intervals = num_scale - 3
    keypoints = findScaleSpaceExtrema(gaussian_images, dog_images,
    num_intervals, sigma, image_border_width)

    keypoints = removeDuplicateKeypoints(keypoints)
    keypoints = convertKeypointsToInputImageSize(keypoints)
    descriptors = generateDescriptors(keypoints, gaussian_images)

    return keypoints, descriptors
```

STAGE 1 – COMPUTE THE IMAGE PYRAMID

Input: Image

Output: Multi-Scale and Multi-Octaves Image

The purpose of computing the image pyramid is to get the scale-space of the image. In our world, objects are meaningful only at a certain scale. Our eyes could see a cat perfectly at any certain scale although the cat is big or small because that cat is a point of interest or key points. In this stage, we also aim to create a scale-space of an image to help our system get key points in the next step.

Here is the code to receive image as an input:

```
image_name = os.path.basename(image)
image = cv2.imread(image, 0)
image = image.astype('float32')

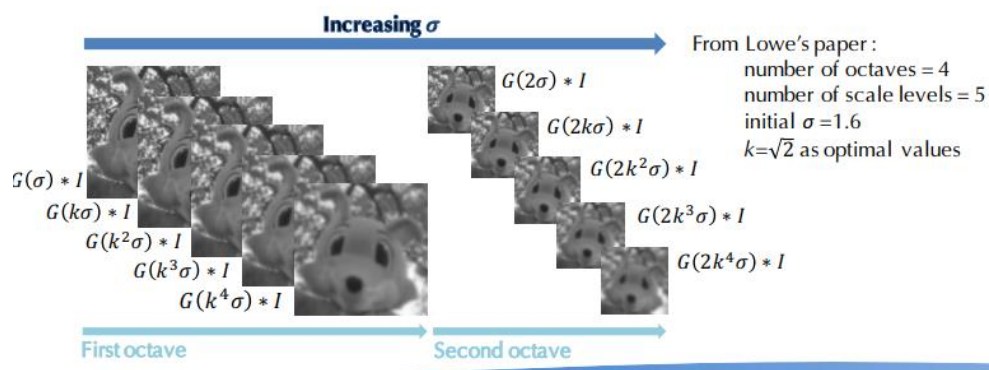
firstImage = generateFirstImage(image, sigma, assumed_blur)
gaussianKernel = generateGaussianSigma(sigma, num_scale)
```

After loading the image with OpenCV, we then make the image array as float32 to make the computation faster a little bit. Then we generate the first image in the first octaves. We scale the original images 2 times bigger than its size so when we create the image pyramid later, the pyramid still gets the key points although we already reduce the size many times in the octaves. In this code, we also generate the Gaussian kernel first, the gaussian kernel is sigma for blurring the image. This sigma is generating according to the number of scales, for example, if our num of scale is 5, it will generate 5 sigmas. The base sigma is 1.6 according to Lowe's Paper. Here is our default value for this code:

```
def SIFT(image,
        sigma=1.6,
        num_octaves=4,
        num_scale=5,
        assumed_blur=0.5,
        image_border_width=5):
```

All of the default values are the recommendation of Lowe's Paper (The State-of-the-Art paper that introduces SIFT methods for matching the images).

From the code, we only generate 5 sigmas because the num_scales are 5. Why we do not generate 20 different sigmas (num_scale * num_octaves [5*4]) according to the figure below?



The answer is we do not need to generate all 20 sigmas because later we will reduce the size of the images in every octave, so because the images will be getting smaller and smaller in every octave, indirectly it will affect the gaussian blur of the images and the sigma effect of the gaussian filter will be multiplied by itself. For example, if we reduce the image in half, the gaussian effect of the sigma will be twice, so we do not need to create all the sigmas for 20 pictures.

After everything getting ready, we then make the Gaussian Pyramid like the code below:

```
gaussian_images = make_pyramid(firstImage, image_name,  
gaussianKernel, num_octaves, num_scale)
```

The most important code of this make_pyramid function is below:

```
image = blur_gaussian(image, sigma[b])
```

First, we will apply gaussian filter to the image using gaussian_kernel that we already generate before.

```
downsampling_scale = 1 / 2  
image = downsample(gaussian_images_in_octave[0], downsampling_scale)
```

Second, we will down be sampling every image by half in each octave, so the gaussian blur will be twice big from previous octaves because of the size of images already reduced half.

These two steps could not be reversed, we should apply the gaussian blur first before down sampling the images to the next octaves. If we do it reversely, the feature of the images will be disappearing and make the images broken in the last octaves.

This is the input (Figure 1) and the output (Table 1) of this step:

Input:



Figure 1 - Input of the Image Pyramid Step

In the Table 1, we could see that the image size will be reduced half in every octave while also the blur will be increased within scale and octaves. This blurred image then will be really useful for the next step to create a Difference of Gaussian.

Output:













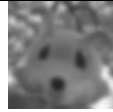
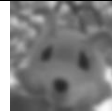






	Scale 1	Scale 2	Scale 3	Scale 4	Scale 5
Octave 1					
Octave 2					
Octave 3					
Octave 4					

Table 1 - Table of Multi-Scale and Multi-Octaves Image

STAGE 2 – COMPUTE THE DIFFERENCE OF GAUSSIAN (DOG)

Input: Multi-Scale and Multi-Octaves Image (Table 1)

Output: DoG Image result in Every Octave

After finding the Multi-Scale and Multi-Octaves Image, we could use that as input to generate a Difference of Gaussian. The Difference of Gaussian is simply the subtraction of every Gaussian Images in every octave. This DoG Images are great for finding out interesting points of the image. The clear figures below will be a clear explanation of what is DoG:

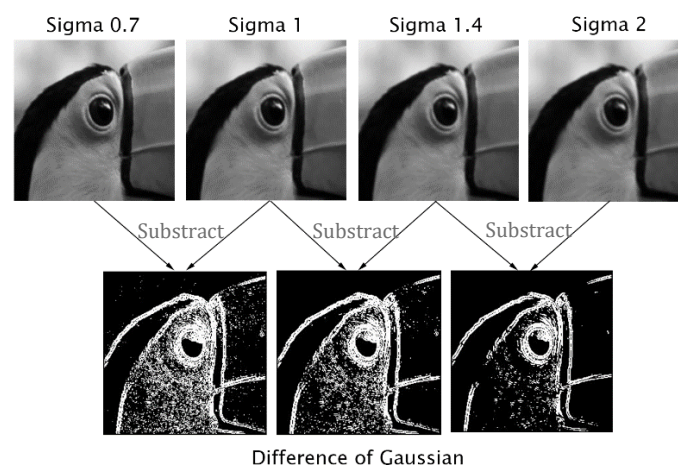


Figure 2 - What Difference of Gaussian Do

According to the figure above, in our case if we have 5 Multi-Scale images in every octave, we will get 4 of Difference of Gaussian in each octave. So, the total DoG Images we got will be 16 DoG Images.

This is the code implementation:

```
dog_images = make_DOG(image_name, gaussian_images, num_octaves)
```

Here is the snippet of the code how we subtract the image in different scale of image to get Difference of Gaussian Images:

```
im1 = gaussian_image[num_octaves][a]
im1_ = cv2.imread(list_of_image[a], cv2.IMREAD_GRAYSCALE)
im2 = gaussian_image[num_octaves][a + 1]
im2_ = cv2.imread(list_of_image[a + 1], cv2.IMREAD_GRAYSCALE)

DoGim = subtract(im1, im2)
DoGim_ = im2_ - im1_
```

This is the input and the output of this step:

Input: Table 1

Output:










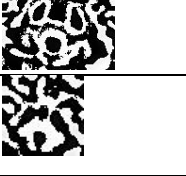

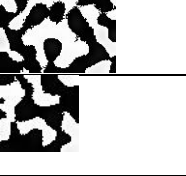
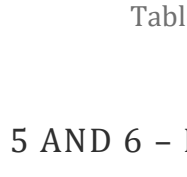
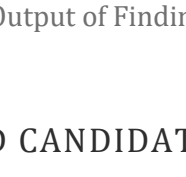
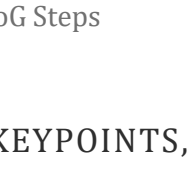

	Scale 1	Scale 2	Scale 3	Scale 4
Octave 1				
Octave 2				
Octave 3				
Octave 4				

Table 2 - Output of Finding DoG Steps

STAGE 3 ,4, 5 AND 6 – FIND CANDIDATE KEYPOINTS, REFINE CANDIDATE KEYPOINTS, FILTER UNSTABLE KEY POINTS FROM NOISE IN IMAGES, AND FILTER UNSTABLE KEY POINTS IN THE EDGES

Input: DoG Images (Table 2)

Output: (X, Y, Scale, Orientation) list of key points with scale and orientation

After find the DoG Image, we tried to find the key points of the image. We need to compare each pixel with its 8 Neighbors as well as 9 pixels in the next scale and 9 pixels in previous scale. This way we will compare each pixel with total of 26 other pixels and see if that pixel in that coordinate is maximum or either minimum. If the pixel is the maximum or

the minimum between all 26 neighbors, we will take a note at the coordinates of pixel and consider it as key points candidate. The figure below will give us nice representation about how we compare each pixel with its 26 neighbors:

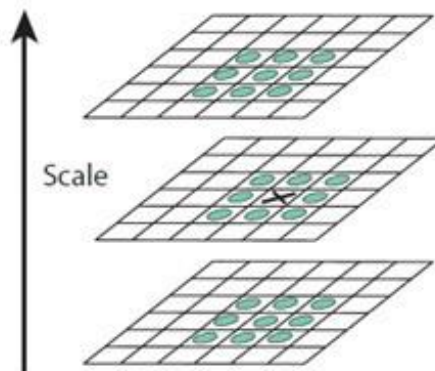


Figure 3 - Compare Pixels with its 26 neighbors

We will just compare DoG images in scales 2 and scales 3 only, since image in scale 1 does not have previous comparison and image in the last scale also does not have next comparison. In term of pixels, we also not comparing the border of image because it does not have enough 8 neighbors surrounding on it.

This is the snippet code how we do the comparison:

```
keypoints = findScaleSpaceExtrema(gaussian_images, dog_images,
num_intervals, sigma, image_border_width)
```

We then decide if pixel is extrema (minimum or maximum) with *isPixelAnExtremum* function and then if that pixel is the pixel extremum, we then refine candidate key points with *localizeExtremumViaQuadraticFit*. What *localizeExtremumViaQuadraticFit* is checking the whether the pixels have enough contrast by checking their intensities, if the pixel is not meet the contrast threshold. They used Taylor series expansion of scale space to get a more accurate location of extrema, and if the intensity at this extremum is less than a threshold value (we use 0.04), it is rejected. The code is shown below:

```
if isPixelAnExtremum(first_image[i - 1:i + 2, j - 1:j + 2], second_image[i - 1:i + 2, j - 1:j + 2],
third_image[i - 1:i + 2, j - 1:j + 2], threshold):
    localization_result = localizeExtremumViaQuadraticFit(i, j, image_index + 1, octave_index,
num_intervals, dog_images_in_octave,
sigma, contrast_threshold,
image_border_width)
```

Key points from previous step still produce a lot of key points. DoG has a higher response for edges, so key points that lie along an edge need to be removed. Here *localizeExtremumViaQuadraticFit* also use 2x2 Hessian Matrix (H) to compute whether key points lie along an edge or not. The code snippet is shown below:

```
hessian = computeHessianAtCenterPixel(pixel_cube)
```

After we already decide that key points through Quadratic Fit, we will compute the orientation in the local space that really useful later if we would to create the descriptor like shown below:

```
if localization_result is not None:
    keypoint, localized_image_index = localization_result
    keypoints_with_orientations = computeKeypointsWithOrientations(keypoint, octave_index,
                                                                    gaussian_images[
                                                                    octave_index][
                                                                    localized_image_index])
    for keypoint_with_orientation in keypoints_with_orientations:
        keypoints.append(keypoint_with_orientation)
```

For assigning the orientation, we know that we try finding key points in every different octave, so we have scale invariance between the key points from different octaves. We need to assign an orientation to each key point to make it rotation invariance.

We will take a neighborhood around the key point location depending on the scale and the gradient magnitude with direction calculated in that region. We need to take an orientation histogram with 36 bins that can cover 360 degrees. The highest peak in the histogram will be taken with any peak above 80% also taken to calculate the orientation. Finally, it will create key points with same location and scale, but different direction.

STAGE 7 AND STAGE 8 - CONVERT KEYPOINT AND ORIENTATION IN THE INPUT IMAGE SIZE AND BUILD THE KEYPOINT DESCRIPTOR

Input: (X, Y, Scale, Orientation) list of key points with scale and orientation

Output: {(x, y, orientation, scale,f)} list of described key points

After we got all stable key points, we need to convert the key points back to its original size. The snippet code shown like this:

```
keypoints = convertKeypointsToInputImageSize(keypoints)
```

Here we need the octave information so we could rescale our key points precisely.

```
def convertKeypointsToInputImageSize(keypoints):
    """Convert keypoint point, size, and octave to input image size
    """
    converted_keypoints = []
    for keypoint in keypoints:
        keypoint.pt = tuple(0.5 * array(keypoint.pt))
        keypoint.size *= 0.5
        keypoint.octave = (keypoint.octave & ~255) |
        ((keypoint.octave - 1) & 255)
        converted_keypoints.append(keypoint)
    return converted_keypoints
```

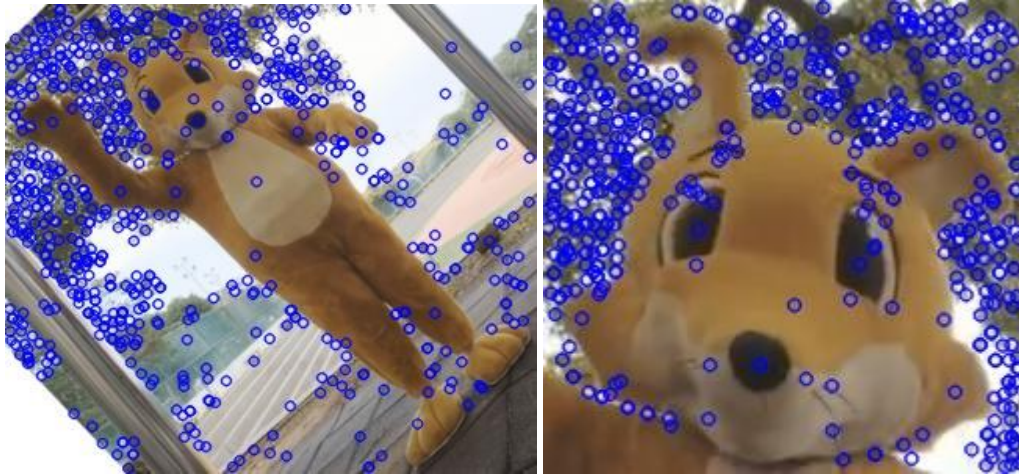
After that we need to create 128 vectors of descriptor so we could do image matching later. The snippet code shown like below:

```
descriptors = generateDescriptors(keypoints, gaussian_images)
```

At this point, each key point has coordinate, scale, and orientation. Next is to compute a descriptor for the local image region about each key point that is highly distinctive and

invariant as possible to variations such as changes in viewpoint and illumination. To do this, a 16x16 window around the key point is taken. It is divided into 16 sub-blocks of 4x4 size. So, 4 X 4 descriptors over 16 X 16 sample array were used in practice. 4 X 4 X 8 directions give 128 bin values. These all 128 values will represent that image for doing image matching.

Here is the result of key point in bamboo_set folder:



FINAL STAGE – IMAGE MATCHING

For matching the image, we use FLANN method like shown in the snippet_code below:

```
img1 = cv2.imread(img1, 0)
img2 = cv2.imread(img2, 0)
# Initialize and use FLANN
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1, des2, k=2)
```

FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than BFMatcher for large datasets.

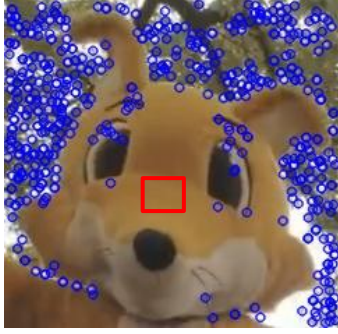
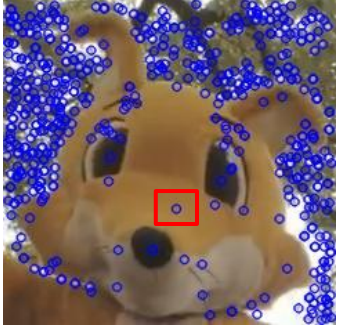
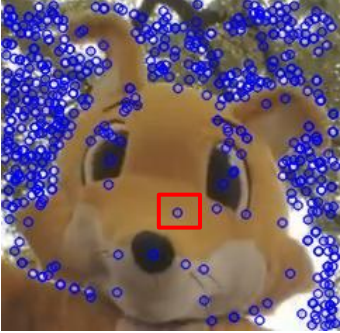
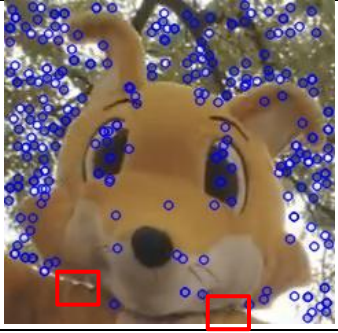
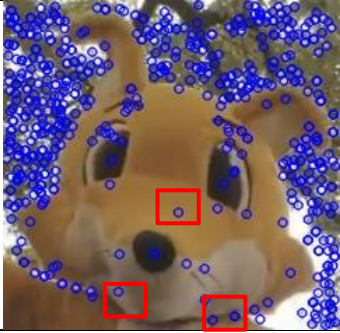
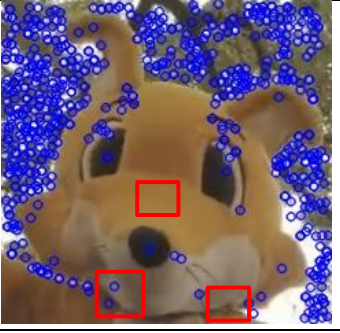
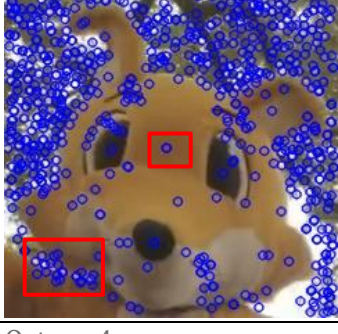
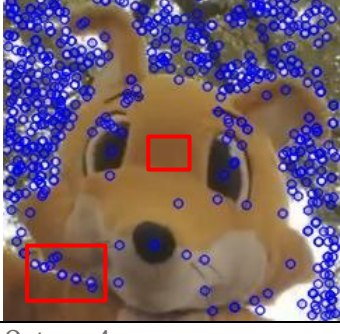
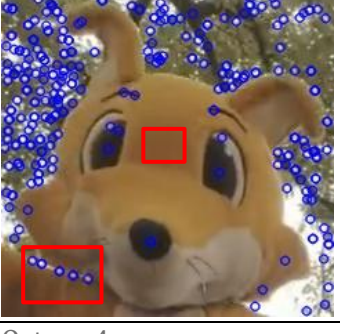
Here is the result from image matching between image in bamboo_set folder:



Figure 4 - Example of the Image Matching result for bamboo_set and own_image

PARAMETER CHANGE

Here I mark the text in blue color if I change some parameter

		
Octave 2 Scale 5 Contrast 0.04	Octave 4 Scale 5 Contrast 0.04	Octave 8 Scale 5 Contrast 0.04
		
Octave 4 Scale 4 Contrast 0.04	Octave 4 Scale 5 Contrast 0.04	Octave 4 Scale 8 Contrast 0.04
		
Octave 4 Scale 5 Contrast 0.02	Octave 4 Scale 5 Contrast 0.04	Octave 4 Scale 5 Contrast 0.08

HOW TO USE THE PROGRAM

From command prompt:

```
python 0860812.py --input bamboo_fox --octave 4 --num_scale 5
```

The default value of octave will be 4 and num_scale will be 5, change bamboo_fox with another image set in the /img folder.

The result could be seen at /result folder