Feature Detection

In [143…
```
import warnings
warnings.filterwarnings("ignore")

import cv2
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

In [144…
```
img_1 = cv2.imread('D:/AI/Computer Vision/image pairs/image pairs_01_01.jpg')
img_2= cv2.imread('D:/AI/Computer Vision/image pairs/image pairs_01_02.jpg')
```

In [145…
```
# Convert to grayscale
img_1_gray = cv2.cvtColor(img_1, cv2.COLOR_BGR2GRAY)
img_2_gray = cv2.cvtColor(img_2, cv2.COLOR_BGR2GRAY)
```

In [146…
```
# Display traning image and testing image
fx, plots = plt.subplots(1, 2, figsize=(20,10))

plots[0].imshow(img_1_gray, cmap='gray')
plots[1].imshow(img_2_gray, cmap='gray')

plt.show()
```

Scale Space Extrema Detection

Generate Image Pyramid

In [147…
```
img = img_1_gray
def quick_resize(img, scale_percent):

    # scale_percent - percent of the original image's scale

    width = int(img.shape[1] * scale_percent / 100)
    height = int(img.shape[0] * scale_percent / 100)
    dim = (width, height)

    # resize image
    resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
```

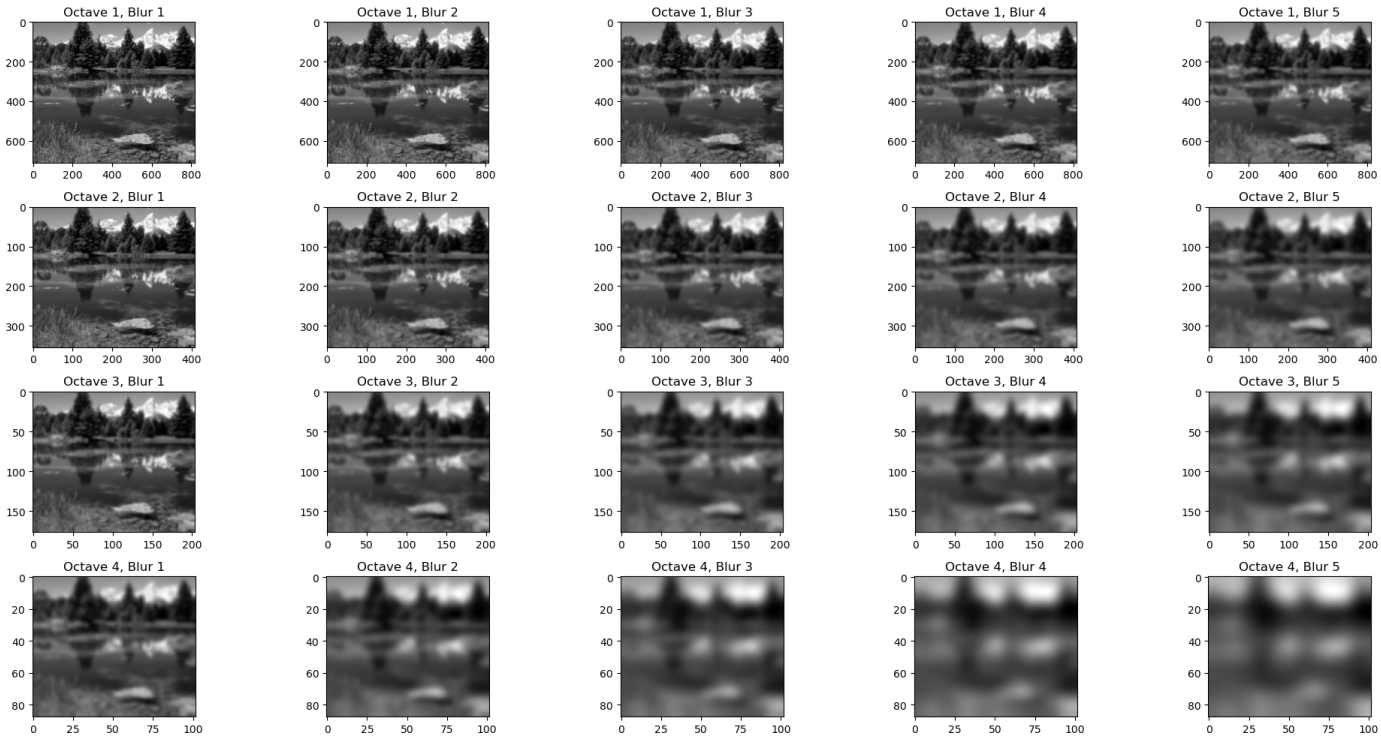
```
    return resized
```

In [148…

```
def generate_image_pyramid(img, num_octaves = 4, num_intervals = 5, step_size = 1):

    """
    Generate image pyramid for SIFT.

    args:
    img - image object
    num_octaves = number of octave (scales)
    num_intervals - number of gaussian operator intervals
    step_size - size of each interval for gaussian operator

    out:
    img_pyramid - image pyramid list. each element corresponds to an octave (list), and
    """

    img_pyramid = []
    imgs_scaled = img

    for i in range(num_octaves):

        # Gaussian Operator parameter
        sigmas = np.arange(1, step_size * (num_intervals + 1), step_size)

        # Create a list of gaussian blurs using different values of sigma parameter
        gaussian_images = []
        for sigma in sigmas:
            gaussian_images.append(cv2.GaussianBlur(imgs_scaled, (0,0), sigma, cv2.BORDE

        # Append each list of gaussian images to the main img_pyramid list
        img_pyramid.append(gaussian_images)

        # Scale down image by half the original size
        imgs_scaled = quick_resize(imgs_scaled, 50)

    return img_pyramid
```

In [149…

```
img_pyramid = generate_image_pyramid(img)
num_blurs = len(img_pyramid[0])
num_octaves = len(img_pyramid)

fig, ax = plt.subplots(num_octaves, num_blurs, figsize=(4 * num_blurs, 2.5 * num_octaves

# Plot image pyramid (rows - octaves, columns - blurs)
for i in range(num_octaves):
    for j in range(num_blurs):
        ax[i,j].imshow(img_pyramid[i][j], cmap='gray')
        ax[i,j].set_title(f'Octave {i+1}, Blur {j+1}')

plt.tight_layout()
```



In [150…

```python
def calculate_dog(img_pyramid, num_octaves = 4):

    DoG = []
    num_intervals = len(img_pyramid[0])
    num_octaves = len(img_pyramid)

    for i in range(num_octaves):

        DoG_row = []
        for j in range(num_intervals - 1):

            # Subtract images of adjacent intervals within the same octave
            DoG_row.append(img_pyramid[i][j+1] - img_pyramid[i][j])

        DoG.append(DoG_row)

    return DoG
```

In [151…

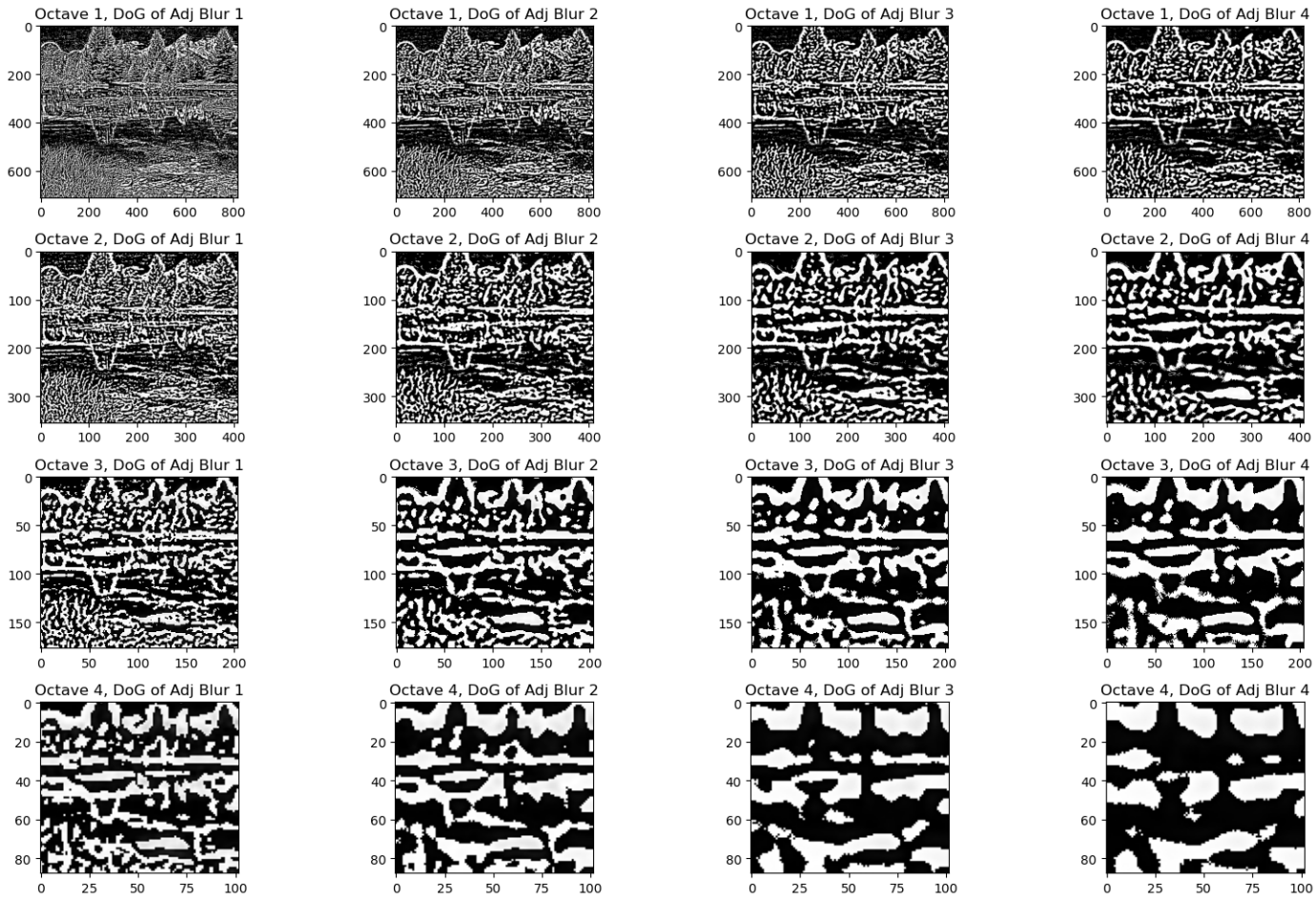
```python
DoG = calculate_dog(img_pyramid)
np.array(DoG[0]).shape
```

Out[151]:

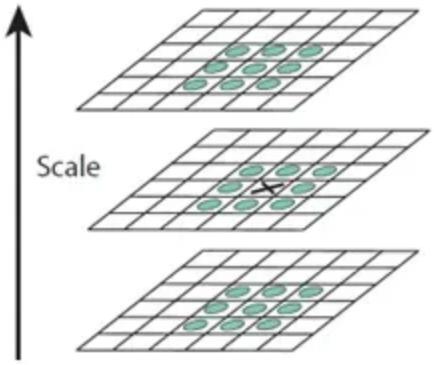
```
(4, 710, 818)
```

In [152…

```python
num_cols = len(DoG[0])
num_rows = len(DoG)

fig, ax = plt.subplots(num_rows, num_cols, figsize=(4 * num_cols, 2.5 * num_rows))

# Plot generated DoG images
for i in range(num_rows):
    for j in range(num_cols):
        ax[i,j].imshow(DoG[i][j], cmap='gray')
        ax[i,j].set_title(f'Octave {i+1}, DoG of Adj Blur {j+1}')

plt.tight_layout()
```



To perform Keypoint Localization, need to define our region of interest. The region of interest, based on the figure below, for any given point are the 9 points above and below, and 8 points surrounding our point of



interest, if any.

In [166…

```
def get_region_of_interest(octave, DoG_image_index, pixel_coordinates, PRINT_INFO = Fals

    """
    Gets the region if interest surrounding a point in a 3x3x3 cube, if any.

    Args:
        octave - list of DoG images. Each element from this list must be a 2D image of t
        DoG_image_index - index to identify current octave
        pixel_coordinate - coordinate of the current pixel being considered

    Out:
        prev_octave - region from the prev octave (top of the stack)
        curr_octave - region of the current octave
        next_octave - region of the next octave (bottom of the stack)
    """

    pixel_row_index, pixel_col_index = pixel_coordinates
```

```
    # Calculate upper and lower bounds (for cases where the point of interest is at the
    row_lower_bound = max(0,pixel_row_index - 1)
    row_upper_bound = pixel_row_index + 2
    col_lower_bound = max(0,pixel_col_index - 1)
    col_upper_bound = pixel_col_index + 2

    # If current octave is the top-most of the octave stack
    if DoG_image_index == 0:
        prev_octave = np.array([])
    else:
        prev_octave = octave[DoG_image_index - 1][row_lower_bound : row_upper_bound, col

    # If the current octave is at bottom-most of the octave stack
    if DoG_image_index == len(octave) - 1:
        next_octave = np.array([])
    else:
        next_octave = octave[DoG_image_index + 1][row_lower_bound : row_upper_bound, col

    curr_octave = octave[DoG_image_index][row_lower_bound : row_upper_bound, col_lower_b

    if PRINT_INFO:
        print(f'Previous Octave:\n {prev_octave}\n')
        print(f'Current Octave:\n {curr_octave}\n')
        print(f'Next Octave:\n {next_octave}')

    return prev_octave.tolist(), curr_octave.tolist(), next_octave.tolist()
```

In [167…

```
def is_center_pixel_extremum(pixel_value, prev_octave, curr_octave, next_octave):

    """
    Checks whether the point of interest / pixel is the extremum (local max or local min

    args:
    pixel_value - value of the pixel in the point of interest
    prev_octave, curr_octave, next_octave - region of interest (list)

    """

    # Concatenate individual regions
    region_of_interest = prev_octave + curr_octave + next_octave

    # Flatten region of interest
    region_of_interest_flatten = [item for sublist in region_of_interest for item in sub

    # Check for extremum case
    is_extremum = any(pixel_value == [max(region_of_interest_flatten), min(region_of_int

    return is_extremum
```

In [168…

```
# FOR TESTING PURPOSES
# Sample DoG list of images for testing purposes
sample_octave = np.array([[[7, 4, 2, 3],
                           [3, 11, 0, 2],
                           [2, 8, 5, 5]],

                          [[8, 2, 5, 5],
                           [3, 0, 9, 7],
                           [4, 3, 3, 8]],

                          [[2, 7, 1, 9],
                           [5, 4, 8, 9],
                           [6, 3, 5, 2]]])
```

```
# Test get_region_of_interest()
sample_DoG_image_index = 1
pixel_coordinates = (1,1)
pixel_value = sample_octave[sample_DoG_image_index][pixel_coordinates]

print(f'Pixel Value: {pixel_value}')

prev_octave_sample, curr_octave_sample, next_octave_sample = get_region_of_interest(samp

print(f'Prev Layer: {prev_octave_sample}')
print(f'Curr Layer: {curr_octave_sample}')
print(f'Next Layer: {next_octave_sample}')
```

```
Pixel Value: 0
Prev Layer: [[7, 4, 2], [3, 11, 0], [2, 8, 5]]
Curr Layer: [[8, 2, 5], [3, 0, 9], [4, 3, 3]]
Next Layer: [[2, 7, 1], [5, 4, 8], [6, 3, 5]]
```

In [169…

```
# Test is_center_pixel_extremum()
is_center_pixel_extremum(pixel_value, prev_octave_sample, curr_octave_sample, next_octav
```

Out[169]:

```
False
```

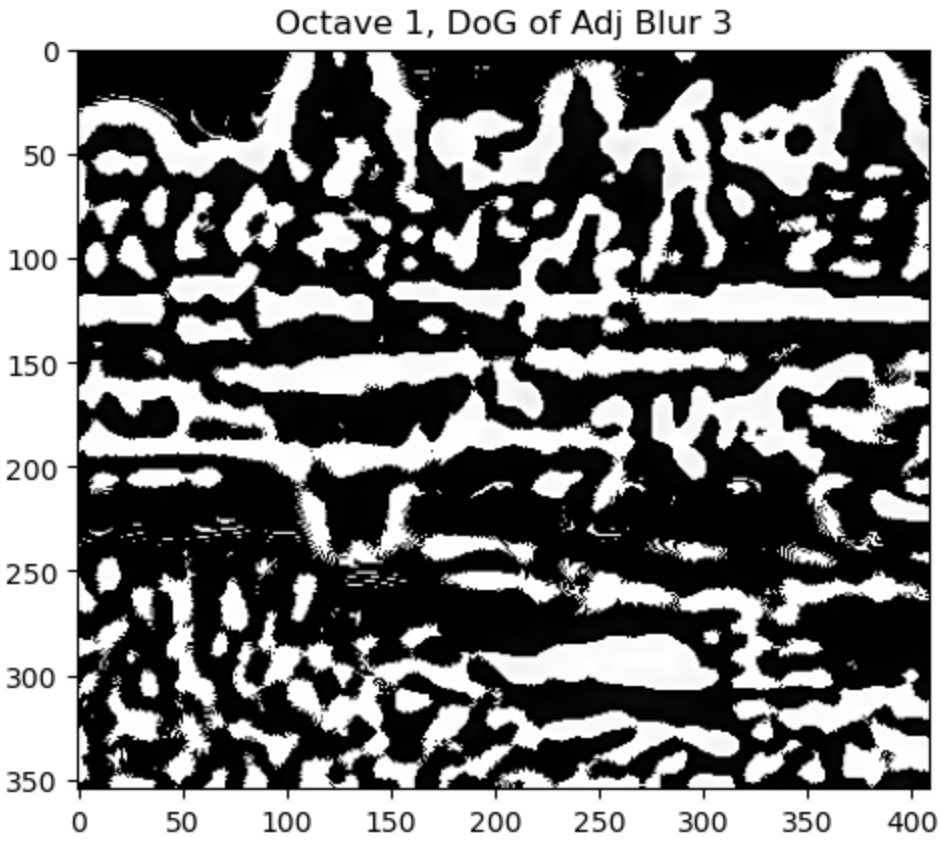
Keypoint Localization for ONE DoG layer

In [170…

```
# Manually define DoG image via octave_index and DoG_image_index
# Generate keypoint for a specific DoG Image
octave_index = 1
DoG_image_index = 3
octave = DoG[octave_index]
DoG_image = octave[DoG_image_index]
plt.imshow(DoG_image,cmap='gray')
plt.title(f'Octave {octave_index}, DoG of Adj Blur {DoG_image_index}')
```

Out[170]:

```
Text(0.5, 1.0, 'Octave 1, DoG of Adj Blur 3')
```



In [171…

```
keypoints = []
```

```
# iterate across DoG images (2d array) in an octave (list)
for pixel_row_index, image_row in enumerate(DoG_image):

    # iterate across image rows (1d array) in a DoG image (2d array)
    for pixel_col_index, center_pixel_value in enumerate(image_row):

        # get coordinate of point of interest
        center_pixel_coordinate = (pixel_row_index, pixel_col_index)

        # get region of interest
        prev_octave, curr_octave, next_octave = get_region_of_interest(octave, DoG_image

        # check if point of interest is the extremum of the region of interest
        if is_center_pixel_extremum(center_pixel_value, prev_octave, curr_octave, next_o

            # append to keypoint output list
            keypoints.append([octave_index, DoG_image_index, (pixel_row_index, pixel_col

print(len(keypoints))
```

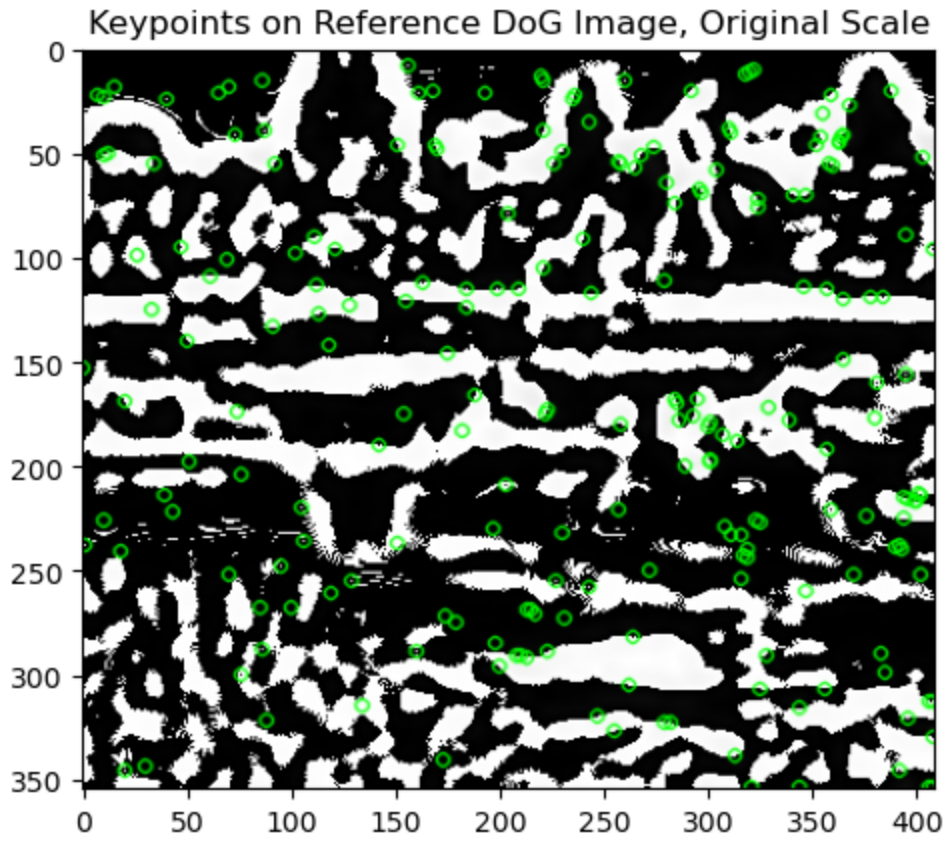
```
213
```

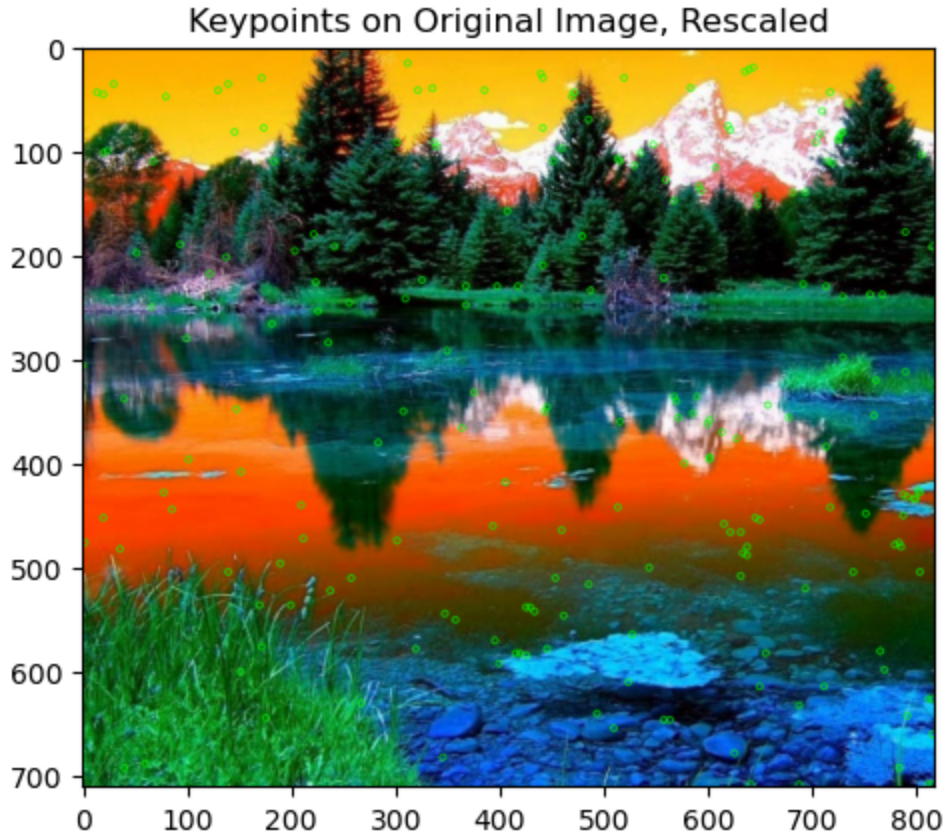
In [172…

```
# plot generated keypoits on the DoG image
plt.figure()
raw_keypoint_objects = [cv2.KeyPoint(kp_y, kp_x, 1) for _, _, (kp_x, kp_y) in keypoints]
plt.imshow(cv2.drawKeypoints(DoG_image, raw_keypoint_objects,  None, (0, 255, 0)))
plt.title('Keypoints on Reference DoG Image, Original Scale')

# plot generated keypoints on the original image (rescaled)
plt.figure()
keypoint_objects = [cv2.KeyPoint(kp_y * 2**octave_index, kp_x * 2**octave_index, 1) for
plt.imshow(cv2.drawKeypoints(img_1, keypoint_objects,  None, (0, 255, 0)))
plt.title('Keypoints on Original Image, Rescaled')
```

Out[172]:

```
Text(0.5, 1.0, 'Keypoints on Original Image, Rescaled')
```





Filter Keypoints on ONE DoG Image

Keypoints on FLAT regions have a corresponding low magnitude harris corner response |R|

Keypoints on EDGE regions have a corresponding negative magnitude harris corner response R

To reject these keypoints, we define a positive threshold value, and only accept keypoints that corresponds to values higher than the threshold value.

(ref: https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html)

In [173…

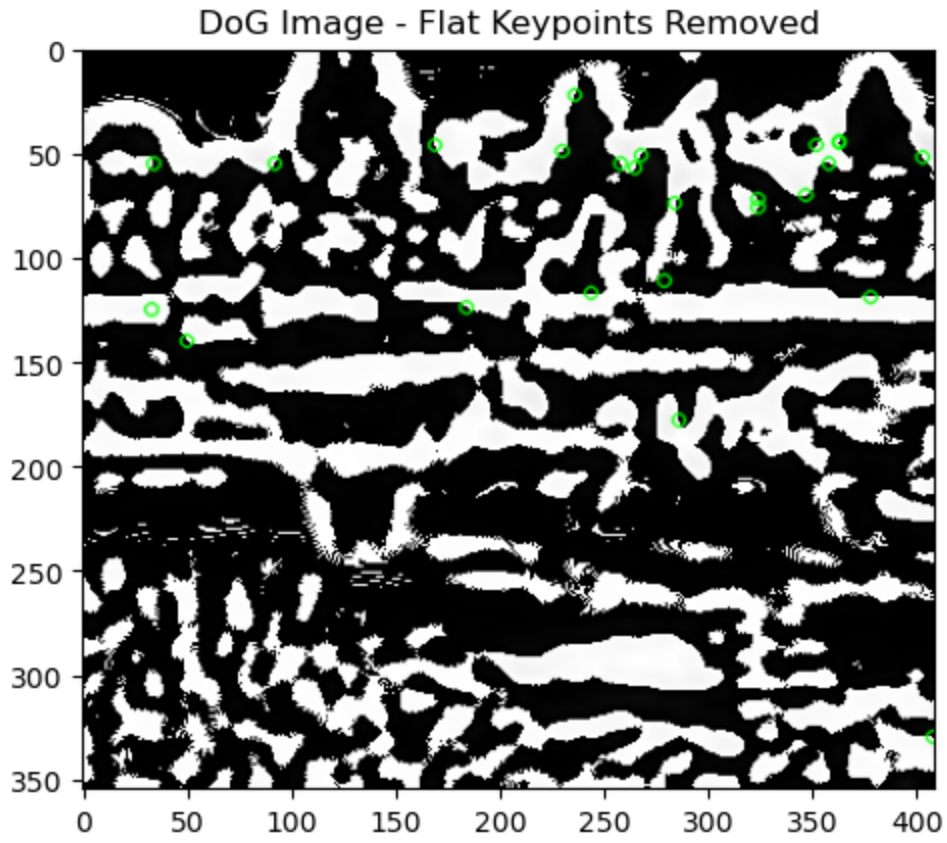
```
keypoints_filtered = []
# Set Harris Corner Detection parameters
block_size = 2  # Neighborhood size for corner detection
ksize = 3  # Aperture parameter for Sobel operator
k = 0.03  # Harris corner detection free parameter

# Apply Harris Corner Detection
corners = cv2.cornerHarris(img_1_gray, block_size, ksize, k)

# Threshold for rejecting keypoints on edges and flats
threshold = 0.001 * corners.max()

for i, (_, _, kp) in enumerate(keypoints):

    # extract individual keypoint coordinates
    kp_x, kp_y = kp

    # Reject Flats and Edges via thresholding the Harris Corner response value
    if corners[kp_x*2**octave_index, kp_y*2**octave_index] > threshold:
        keypoints_filtered.append(keypoints[i])
```

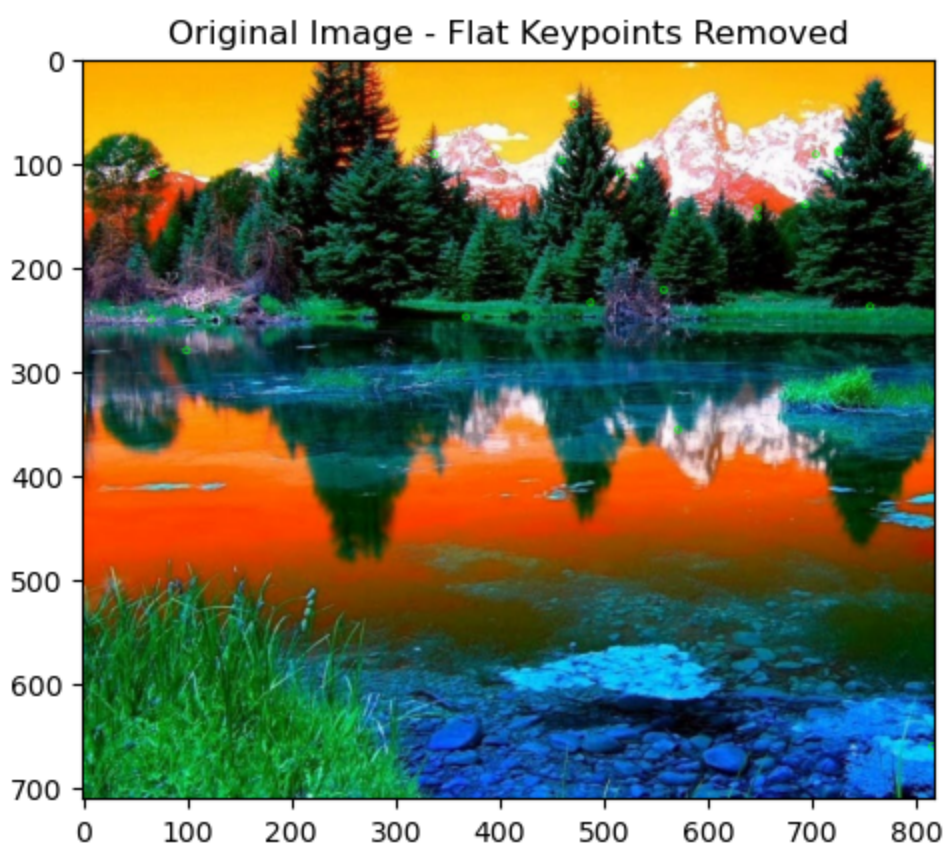
In [174…

```
# plot generated keypoits on the DoG image
plt.figure()
raw_keypoint_objects = [cv2.KeyPoint(kp_y, kp_x, 1) for _, _, (kp_x, kp_y) in keypoints_
```

```
plt.imshow(cv2.drawKeypoints(DoG_image, raw_keypoint_objects,  None, (0, 255, 0)))
plt.title('DoG Image - Flat Keypoints Removed')

# plot generated keypoints on the original image (rescaled)
plt.figure()
rescale_factor = 2**octave_index
keypoint_objects = [cv2.KeyPoint(kp_y * rescale_factor, kp_x * rescale_factor, 1) for _,
plt.imshow(cv2.drawKeypoints(img_1, keypoint_objects,  None, (0, 255, 0)))
plt.title('Original Image - Flat Keypoints Removed')
```

Out[174]:
```
Text(0.5, 1.0, 'Original Image - Flat Keypoints Removed')
```





Keypoint Localization (Across All DoG Layers + Filtering)

In [175…

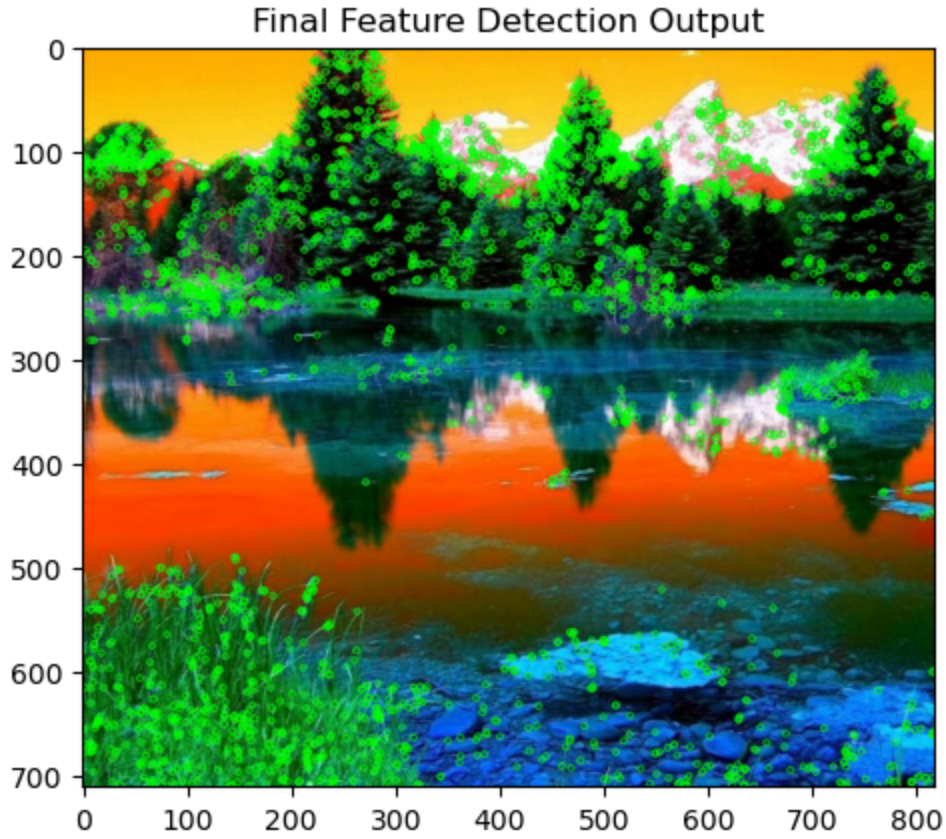
```
def SIFT_feature_detection(DoG, ref_image, block_size = 2, ksize = 3, k = 0.03, threshol

    """

    args:
    block_size - Neighborhood size for corner detection
    ksize - Aperture parameter for Sobel operator
    k - Harris corner detection free parameter

    # Threshold for classifying keypoints as edges or flats

    out:
    keypoints - list of keypoints generated via the SIFT algorithm
              - format: [octave_index, DoG_image_index, (pixel_row_index, pixel_col_inde

    """

    keypoints = []

    # Calculate corner coefficient to filter out flats and edges
    # NOTE: not sure whether or not to manually calculate the hessian matrix or to use t
    corners = cv2.cornerHarris(ref_image, block_size, ksize, k)

    # Define threshold
    threshold = threshold_parameter * corners.max()

    # iterate across octaves in our DoG list
    for octave_index, octave in enumerate(DoG):

        # iterate across DoG images (2d array) in an octave (list)
        for DoG_image_index, DoG_image in enumerate(octave):

            # iterate across image rows (1d array) in a DoG image (2d array)
            for pixel_row_index, image_row in enumerate(DoG_image):
```

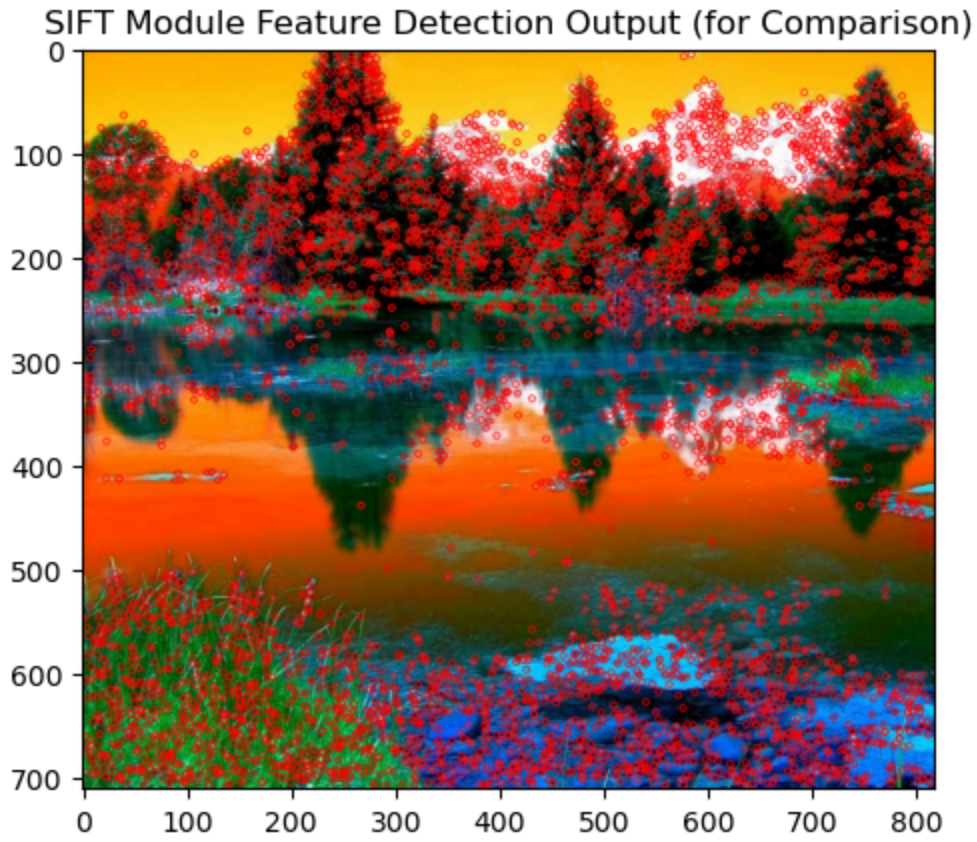
```
                # iterate across columns in an image rows
                for pixel_col_index, center_pixel_value in enumerate(image_row):

                    # get coordinate of point of interest
                    center_pixel_coordinate = (pixel_row_index, pixel_col_index)

                    # get region of interest
                    prev_octave, curr_octave, next_octave = get_region_of_interest(octav

                    # check if point of interest is the extremum of the region of intere
                    if is_center_pixel_extremum(center_pixel_value, prev_octave, curr_oc

                        # scale pixel coordinates back to the size of the original image
                        scaled_pixel_row_index = pixel_row_index*(2**octave_index)
                        scaled_pixel_col_index = pixel_col_index*(2**octave_index)

                        # Accept only if extremum detected is NOT an flat or edge
                        if corners[scaled_pixel_row_index, scaled_pixel_col_index] > thr

                            # append to keypoint output list
                            keypoints.append([octave_index, DoG_image_index, (pixel_row_

    return keypoints
```

In [176…
```
keypoints = SIFT_feature_detection(DoG, img, block_size = 2, ksize = 3, k = 0.03, thresh
```

In [177…
```
plt.figure()
keypoint_objects = [cv2.KeyPoint(kp_y * 2**octave_index, kp_x * 2**octave_index, 1) for
plt.imshow(cv2.drawKeypoints(img_1, keypoint_objects,  None, (0, 255, 0)))
plt.title('Final Feature Detection Output')

plt.figure()
sift = cv2.SIFT_create()
kp_ = sift.detect(img_1_gray,None)
plt.imshow(cv2.drawKeypoints(img_1, kp_,  None, (255, 0, 0)))
plt.title('SIFT Module Feature Detection Output (for Comparison)')
```

Out[177]:
```
Text(0.5, 1.0, 'SIFT Module Feature Detection Output (for Comparison)')
```





Feature Matching

We are reading the input images again in the following piece of code

In [187…

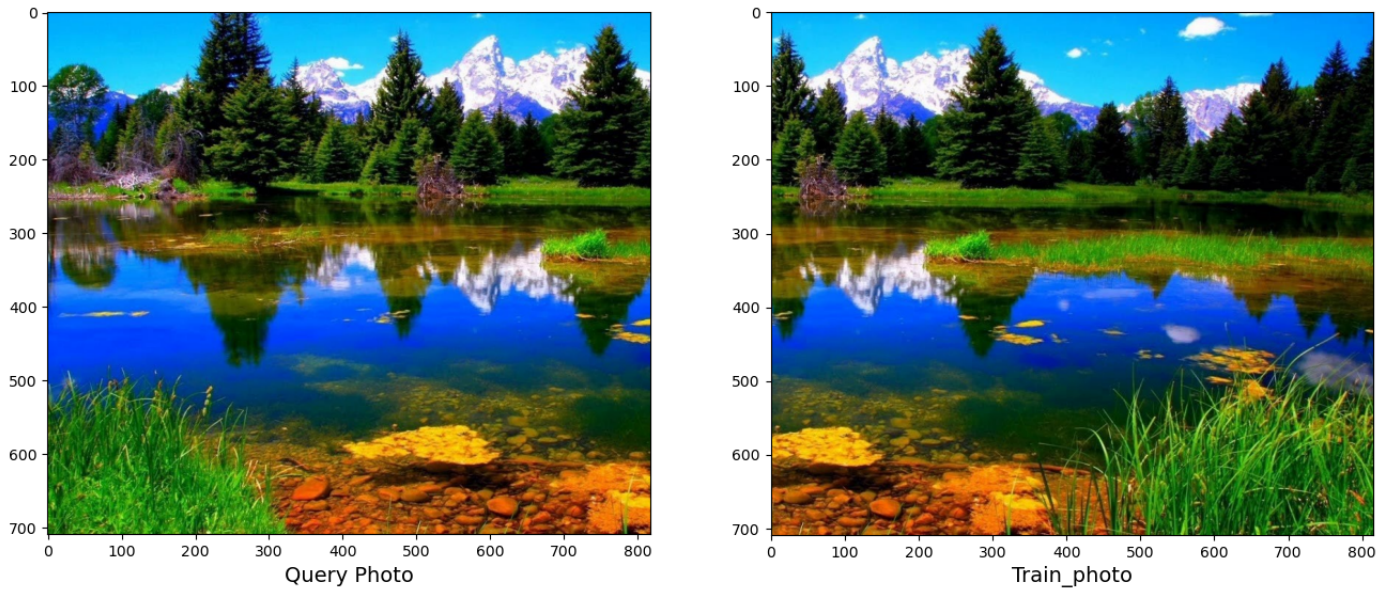
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
import imageio
cv2.ocl.setUseOpenCL(False)
import warnings
warnings.filterwarnings('ignore')
```

In [188…
```
feature_extraction_algo = 'sift'
feature_to_match = 'bf'
```

In [198…
```
train_photo = cv2.imread("D:/AI/Computer Vision/image pairs/image pairs_01_02.jpg")
# Covert BGR to RBG opencv2 takes images in RGB format
train_photo = cv2.cvtColor(train_photo, cv2.COLOR_BGR2RGB)
train_photo_gray = cv2.cvtColor(train_photo, cv2.COLOR_RGB2GRAY) #SIFT computes only on
query_photo = cv2.imread("D:/AI/Computer Vision/image pairs/image pairs_01_01.jpg")
# Covert BGR to RBG opencv2 takes images in RGB format
query_photo = cv2.cvtColor(query_photo, cv2.COLOR_BGR2RGB)
query_photo_gray = cv2.cvtColor(query_photo, cv2.COLOR_RGB2GRAY) #SIFT computes only on

#To view the images
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, constrained_layout = False, figsize=(16
ax1.imshow(query_photo, cmap='gray')
ax1.set_xlabel('Query Photo', fontsize=14)
ax2.imshow(train_photo, cmap='gray')
ax2.set_xlabel('Train_photo', fontsize=14)
```

Out[198]:
```
Text(0.5, 0, 'Train_photo')
```



In [199…
```
def select_descriptor_method(image, method=None):
    assert method is not None, "Please define a descriptor method. Accepted values are:
    if method == 'sift':
        descriptor = cv2.SIFT_create()
#     if method == 'surf':
#         descriptor = cv2.SURF_create()
    if method == 'brisk':
        descriptor =cv2.BRISK_create()
    if method == 'orb':
        descriptor = cv2.ORB_create()
    (keypoints, features) = descriptor.detectAndCompute(image, None)

    return (keypoints, features)
# keypoints will be a list of keypoints and descriptors is (Number of keypoints) * 128
# We need both keypoints and descriptors. Keypoints only contain info aout position. Doe
# SIFT, SURF, BRISK, BRIEF (Different image descriptors)
```

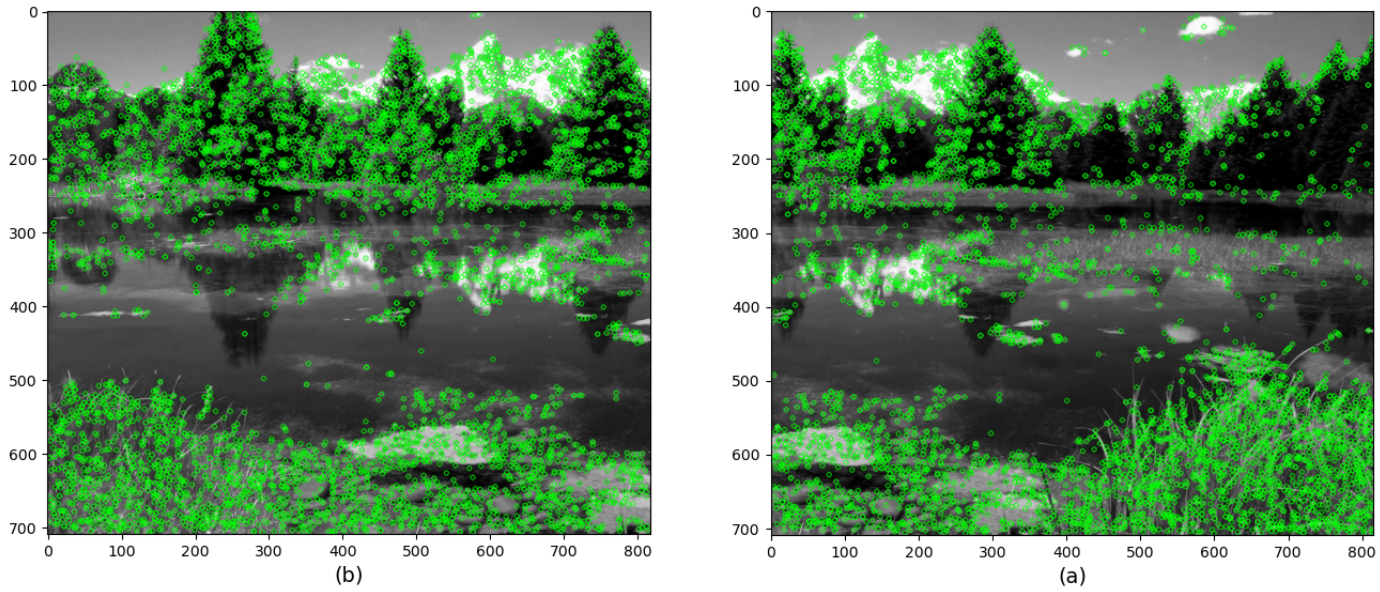
In [200…
```
keypoints_train_img, feature_train_img = select_descriptor_method(train_photo_gray, meth
keypoints_query_img, feature_query_img = select_descriptor_method(query_photo_gray, meth
```

```
# print('keypoints_query_img', key_points_query_img)
# print('feature/descriptor_quer_img', feature_query_img)
for keypoint in keypoints_query_img:
    x, y=keypoint.pt
    size = keypoint.size
    orientation = keypoint.angle
    response = keypoint.response
    octave = keypoint.octave
    class_id = keypoint.class_id
print("x,y",x,y)
print("size:", size)
print("orientation:", orientation)
print("response:", response)
print("class_id:", class_id)
# Feature descriptor is a vector of size len(keypoints)*128
print(feature_query_img.shape)
```

```
x,y 814.8831787109375 124.9290771484375
size: 1.9654314517974854
orientation: 238.29833984375
response: 0.022851983085274696
class_id: -1
(5911, 128)
```

In [201…

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(16,9), constrained_layout = Fa
ax1.imshow(cv2.drawKeypoints(query_photo_gray, keypoints_query_img, None, color=(0,255,0
ax1.set_xlabel('(b)', fontsize=14)
ax2.imshow(cv2.drawKeypoints(train_photo_gray, keypoints_train_img, None, color=(0,255,0
ax2.set_xlabel('(a)', fontsize=14)
# plt.savefig('feature_extraction_sample_img_2.jpg', bbox_inches='tight', dpi=300, forma
plt.show()
```



In [202…

```
#Brute Force Matcher
#Brute force matcher - Given 2 sets of feature, each feature from set 1 is compared agai
def create_matching_object(method, crossCheck):
    if method == 'sift' or method == 'surf':
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=crossCheck) #For l2 Distance/eucledia
    elif method == 'orb' or method == 'brisk':
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=crossCheck) #For Hamming distanc
    return bf
```

In [203…

```
def keypoints_matching(feature_train_img, feature_query_img, method):
    bf =create_matching_object(method, crossCheck=True)
    best_matches = bf.match(feature_train_img,feature_query_img)
    raw_matches = sorted(best_matches, key = lambda x: x.distance)
```

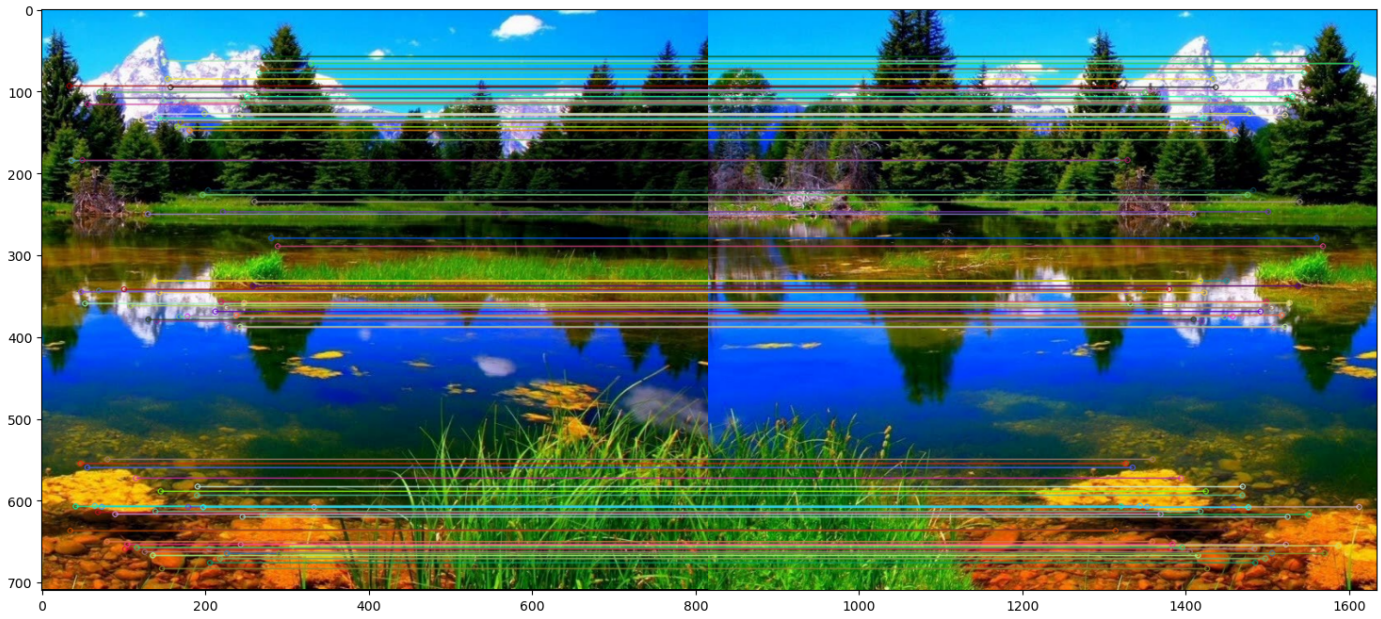
```
    print('raw matches with brute force', len(raw_matches))
    return raw_matches
```

In [204…
```
def keypoints_matching_knn(feature_train_img, feature_query_img, ratio, method):
    bf =create_matching_object(method, crossCheck=False)
    raw_matches = bf.knnMatch(feature_train_img, feature_query_img, k=2)
    print('raw matches with brute force', len(raw_matches))
    knn_matches = []
    for m,n in raw_matches:
        if m.distance < n.distance * ratio:
            knn_matches.append(m)
    return knn_matches
```

In [205…
```
print('Drawing matched features for', feature_to_match)
fig = plt.figure(figsize=(20,8))
if feature_to_match == 'bf':
    matches = keypoints_matching(feature_train_img, feature_query_img, method=feature_ex
    mapped_feature_image = cv2.drawMatches(train_photo, keypoints_train_img, query_photo
elif feature_to_match == 'knn':
    matches = keypoints_matching_knn(feature_train_img, feature_query_img, method=featur
    mapped_feature_image_knn = cv2.drawMatches(train_photo, keypoints_train_img, query_p
plt.imshow(mapped_feature_image)
```

```
Drawing matched features for bf
raw matches with brute force 2798
<matplotlib.image.AxesImage at 0x18fc9121e10>
```

Out[205]:



In [ ]:
```
feature_to_match = 'bf'
print('Drawing matched features for', feature_to_match)
fig = plt.figure(figsize=(20,8))
if feature_to_match == 'bf':
    matches = keypoints_matching(feature_train_img, feature_query_img, method=feature_ex
    mapped_feature_image = cv2.drawMatches(train_photo, keypoints_train_img, query_photo
elif feature_to_match == 'knn':
    matches = keypoints_matching_knn(feature_train_img, feature_query_img, ratio=0.70, m
    mapped_feature_image_knn = cv2.drawMatches(train_photo, keypoints_train_img, query_p
plt.imshow(mapped_feature_image_knn)
```

Image Stitching - Homography Matrix

In [ ]:
```
def homography_stitching(keypoints_train_img, keypoints_query_img, matches, reprojThresh
    #Convert to numpy array
```

```
    keypoints_train_img = np.float32([keypoint.pt for keypoint in keypoints_train_img])
    keypoints_query_img = np.float32([keypoint.pt for keypoint in keypoints_query_img])

    if len(matches) > 4:
        # construct the two sets of points
        points_train = np.float32([keypoints_train_img[m.queryIdx] for m in matches]).re
        points_query = np.float32([keypoints_query_img[m.trainIdx] for m in matches]).re

        # Calculate the homography between the sets of points
        (H, status) = cv2.findHomography(points_train, points_query, cv2.RANSAC, reprojT

        return (matches, H, status)
    else:
        return None
```

In [207…
```
M = homography_stitching(keypoints_train_img, keypoints_query_img, matches, reprojThresh
if M is None:
    print('Error')

(matches, Homography_Matrix, status) = M
print(Homography_Matrix)
```

```
[[ 9.99998478e-01 -1.12814114e-05  4.63000485e+02]
 [ 1.31611435e-05  1.00000623e+00  6.59070462e-04]
 [-8.49152219e-08  3.77886730e-08  1.00000000e+00]]
```

In [208…
```
width = query_photo.shape[1] + train_photo.shape[1]
print(width)
height = max(query_photo.shape[0], train_photo.shape[0])
print(height)
```

```
1634
710
```

In [209…
```
result=cv2.warpPerspective(train_photo, Homography_Matrix, (width, height))
# print(result)

result[0:query_photo.shape[0], 0:query_photo.shape[1]] = query_photo

plt.figure(figsize=(16,10))
plt.axis('off')
plt.imshow(result)
```

Out[209]:
```
<matplotlib.image.AxesImage at 0x18fbda1c290>
```