HW1

January 18, 2020

1 CSE 152 Intro to Computer Vision Spring 2020 - Assignment 1

1.0.1 Instructor: David Kriegman

1.0.2 Assignment Published On: Friday, January 17, 2020

1.0.3 Due On: Thursday, January 30th, 2020 11:59 pm

1.1 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- All solutions must be written in this notebook
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), but you may not use packages that directly solve the problem unless explicitly stated.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- It is highly recommended that you begin working on this assignment early.
-

1.2 Late policy - 10% per day late penalty after due date.

Welcome to CSE152 Intro to Computer Vision! This course gives you a comprehensive introduction to computer vison providing broad coverage including low level vision, inferring 3D properties from images, and object recognition. We will be using a variety of tools in this class that will require some initial configuration. To ensure smooth progress, we will setup the majority of the

tools to be used in this course in this assignment. You will also practice some basic image manipulation techniques. Finally, you will need to export this Ipython notebook as pdf and submit it to Gradescope along with .ipynb file before the due date.

1.2.1 Piazza, Gradescope and Python

Piazza

Go to Piazza and sign up for the class using your ucsd.edu email account. You'll be able to ask the professor, the TAs and your classmates questions on Piazza. Class announcements will be made using Piazza, so make sure you check your email or Piazza frequently.

Gradescope

Every student will get an email regarding gradescope signup once enrolled in this class. All the assignments are required to be submitted to gradescope for grading. Make sure that you mark each page for different problems.

Python

We will use the Python programming language for all assignments in this course, with a few popular libraries (numpy, matplotlib). Assignments will be given in the format of browser-based Jupyter/Ipython notebook that you are currently viewing. We expect that many of you have some experience with Python and Numpy. And if you have previous knowledge in Matlab, check out the numpy for Matlab users page. The section below will serve as a quick introduction to Numpy and some other libraries.

2 Homework 1

In this homework, we will go through what we learned during the first three weeks, including basic linear algebra, least squares method, feature descriptor and matching, and bag of visual words.

You should finish this homework in this `hw1.ipynb` file using our provided templates. You can add other functions to solve the problems if necessary, but please only add them in this file.

The due date for this homework is 11:59PM, Jan 30th, Thursday. Please submit this `hw1.ipynb` to Gradescope.

[ ]:

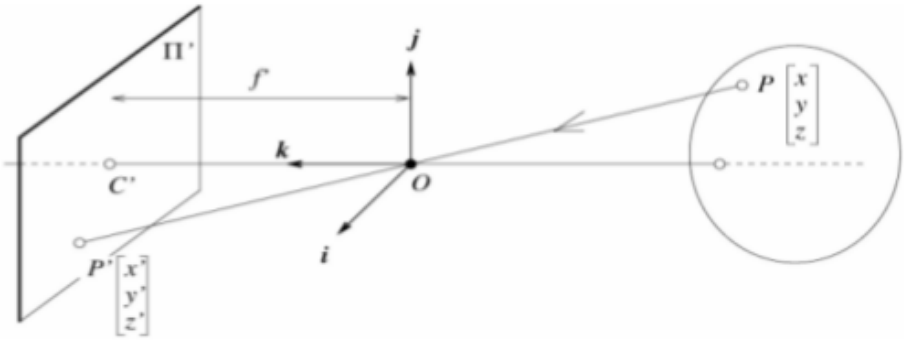
3 Question 1: Projection and Homogenous Coordinates (10 points)

3.0.1 1.1: Planar Projection [7 pts]

Consider a perspective projection where a point

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

is projected onto an image plane Π' represented by $k = f' > 0$ as shown in the following figure.



The first second and third coordinate axes are denoted by i, j, k respectively.

Consider the location of points Q1 and Q2 in the world coordinate system. The locations are parameterized by the equations below:

$$Q1 = [\text{-2, 3,-5}] + t[3\ 6\ 10]$$
$$Q2 = [\ 3,\text{-4,-2}] + t[3\ 6\ 10]$$

where $ t ≤ -1.

Calculate the projection of points Q1 and Q2 onto the image plane when $ t = -1$, and the limit as t approaches $-∞$.$Identifythevanishingpointsof Q1 and Q2$.

3.0.2 1.2: Vanishing Point [3 pts]

Explain why two lines that are parallel have the same vanishing point:

[ ]:

4 Question 2: Image Projection and Rigid Transformations [20 points]

In this problem we will practice rigid body transformations and image formations through the pinhole perspective camera model. The goal will be to photograph the following four points

$$^{A}P_1 = [\text{-1 -0.5 2}]^T$$

,

$$^{A}P_2 = [1\ \text{-0.5 2}]^T$$

,

$$^{A}P_3 = [1\ 0.5\ 2]^T$$

,

$$^{A}P_4 = [\text{-1 0.5 2}]^T$$

To do this we will need two matrices. Recall, first, the following formula for rigid body transformation

$$^{B}P = {}^{B}_{A}R\,{}^{A}P + {}^{B}O_{A}$$

Where ^{B}P is the point coordinate in the target (B) coordinate system. ^{A}P is the point coordinate in the source (A) coordinate system. $^{B}_{A}R$ is the rotation matrix from A to B, and $^{B}O_{A}$ is the origin of the coordinate system A expressed in B coordinates.

The rotation and translation can be combined into a single 4×4 extrinsic parameter matrix, P_e, so that $^{B}P = P_e\,{}^{A}P$ where ^{A}P and ^{B}P are in homogeneous coordinates.

Once transformed, the points can be photographed using the intrinsic camera matrix, P_i which is a 3×4 matrix.

Once these are found, the image of a point, ^{B}P, can be calculated as $P_i P_e\,{}^{A}P$.

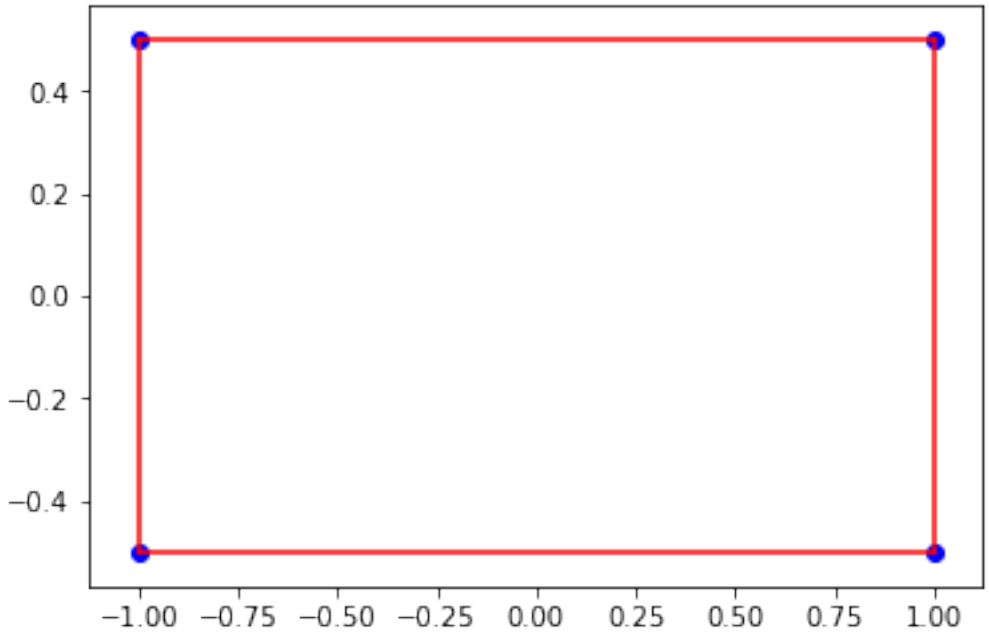
We will consider four different settings of focal length, viewing angles and camera positions below. For each of these calculate:

`a) Extrinsic transformation matrix`

`b) Intrinsic camera matrix under the perspective camera assumption`

`c) Calculate the image of the four vertices and plot using the supplied functions`

`Your output should look something like the following image (Your output values might not match,`



We will not use a full intrinsic camera matrix (e.g. that maps centimeters to pixels, and defines the coordinates of the center of the image), but only parameterize this with f, the focal length. In

other words: the only parameter in the intrinsic camera matrix under the perspective assumption is f.

1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
2. [Translation]. ${}^{B}O_{A} = [0\ 0\ 1]^{T}$. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. ${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 1]^{T}$.
4. [Translation and Rotation, long distance]. Focal length = 5. ${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 13]^{T}$.

   You can refer the Richard Szeliski starting page 36 for image formation and the extrinsic matrix.

   Intrinsic matrix calculation for perspective camera models was covered in class and can be referred in slide 2
   https://cseweb.ucsd.edu/classes/wi20/cse152-a/lec2.pdf

For your answers, you will write the extrinsic and intrinsic matrices below. After this section below we provide starter code for you to complete part c)

4.0.1 2.1: Extrinsic and Intrinsic Matrices [5 points]

Write the extrinsic and intrinsic matrices for each camera

Camera 1:

- P_e:
- P_i:

Camera 2:

- P_e:
- P_i:

Camera 3:

- P_e:
- P_i:

Camera 4:

- P_e:
- P_i:

4.0.2 2.2: Image Calculation [12 points]

You are provided the following starter code as a guide to structure your code and plot the projected points. You are free to modify or use different functions for calculating the rigid transformations and the image projection but please use the plot_points() and main() functions provided.

Here is a list of the provided functions:

- to_homog(): converts points from Euclidean to homogenous coordinates
- from_homog(): converts points from homogenous coordinates back to Euclidean coordinates
- project_points(): the function that takes P_i, P_e, and the point coordinates and calculates the projection onto the camera
- camera_1(): function where you define you define your intrinsic and extrinsic parameters for each camera - there are also functions for camera 2,3,4
- plot_points(): plots the projected camera points
- main(): takes the camera coordinates

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

# convert points from Euclidian to homogeneous
def to_homog(points):

    ## Your Code Here ##


    return points_homog


# convert points from homogeneous to Euclidian
def from_homog(points_homog):

    ## Your Code Here ##

    return points_eud



# project 3D euclidian points to 2D Euclidian
def project_points(P_int, P_ext, pts):

    ## Your Code Here ##

    return projected_points
```

[10]:
```
# Explicitly state the matrices for the four cases as described in the problem
# in the four camera functions given below. Make sure that we can see the formula
```

```
# (if one exists) being used to fill in the matrices. Feel free to document with
# comments any thing you feel the need to explain.

def camera1():

    ## Your Code Here ##

    return P_i, P_e

def camera2():

    ## Your Code Here ##

    return P_i, P_e

def camera3():

    ## Your Code Here ##

    return P_i, P_e

def camera4():

    ## Your Code Here ##

    return P_i, P_e



# Use the following code to display your outputs
# You are free to change the axis parameters to better
# display your quadrilateral but do not remove any annotations

def plot_points(points, title='', style='.-r', axis=[]):

    inds = list(range(points.shape[1]))+[0]
    plt.plot(points[0,inds], points[1,inds],style)

    for i in range(len(points[0,inds])):
        plt.annotate(str("{0:.3f}".format(points[0,inds][i]))+","+str("{0:.3f}".
 ↪format(points[1,inds][i])),(points[0,inds][i], points[1,inds][i]))

    if title:
        plt.title(title)
    if axis:
        plt.axis(axis)
```

```
    plt.tight_layout()

def main():

    point1 = np.array([[-1,-.5,2]]).T
    point2 = np.array([[1,-.5,2]]).T
    point3 = np.array([[1,.5,2]]).T
    point4 = np.array([[-1,.5,2]]).T
    points = np.hstack((point1,point2,point3,point4))

    for i, camera in enumerate([camera1,camera2,camera3,camera4]):

        P_int_proj, P_ext = camera()
        ax1 = plt.subplot(2, 2, i+1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camera %d␣
↪Projective'%(i+1))#, axis=[-0.6,2.5,-0.75,0.75])

        ax1.margins(0.5)
        plt.show()

main()
```

4.0.3 2.3: Reverse Transformation [3 points]

If the transformation from a B frame to an A frame is given by ${}^{A}P = {}^{A}_{B}R\,{}^{B}P + {}^{A}O_{B}$, what is the transformation to go from the A frame to the B frame?

Express your answer for homogeneous coordinates as a 4x4 matrix

Answer:

[ ]:

5 Question 3: Filtering and Template Matching [15 points]

In this part, you will be implementing a series of filters, convolving them with images and observing the output. Additionally, you will look at how to utilize filters for template matching. You may use the convolve function from scipy in order to perform the convolutions. You cannot use any library functions to create the filters

5.0.1 Prerequisite: Image Filtering with Box filter and a Gaussian Filter

Load the `chair.png` image and smooth it with a 5x5 box filter and a 7x7 gaussian filter. The gaussian filter will be provided for you and you will have to implement the box filter yourself.

[11]:
```
from scipy.signal import convolve
from skimage import filters
from imageio import imread

I = imread('dog.jpg', as_gray=True)
plt.imshow(I,cmap = 'gray')
plt.axis('off')
plt.title('original image')
plt.show()


# gaussian blurring is provided for you
def gaussian2d(filter_size=7, sig=1.0):
    """
    Creates 2D Gaussian kernel with side length `filter_size` and a sigma of␣
 ↪`sig`.
    Source: https://stackoverflow.com/a/43346070
    """
    ax = np.arange(-filter_size // 2 + 1., filter_size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sig))
    return kernel / np.sum(kernel)


### YOUR CODE HERE ###

box_filter=None # define a box filter
gauss_filter = gaussian2d()

### END YOUR CODE HERE ###


I_box = convolve(I, box_filter, mode='same')
I_gauss = convolve(I,gauss_filter,mode='same')

plt.imshow(I_gauss,cmap = 'gray')
plt.axis('off')
plt.title('guassian-smoothed image')
plt.show()

plt.imshow(I_box,cmap = 'gray')
plt.axis('off')
plt.title('box-smoothed image')
plt.show()


# you can play around with the filter size and
```

```
# the sigma on the gaussian filter but be sure to submit it
# with a window = 7 and sigma = 1
```

5.0.2 3.1: Vertical Edge filter [5 points]

Implement a vertical gradient filter to extract vertical edges from the `chair.jpg` image

[13]:
```
img = imread('chair.jpg', as_gray=True)


### YOUR CODE HERE ###

vert_filter = None
y_grad_img = None

### END YOUR CODE ###

plt.imshow(y_grad_image,cmap = 'gray')
plt.axis('off')
plt.title('vertical edge filter')
plt.show()
```

5.0.3 3.2: Linearly Separable Filters [5 points]

A 2D linearly-separable convolution filter can be split into 2 distinct filters and reduce run-time. The 2D Sobel Operator:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Can be split into 2 separate 1D kernels. You will have to determine what these 1D Kernels are. There are two functions below that you will use to convolve the sobel operator with the image. In the first function, use the 2D kernel listed above. In the second function, convolve the image with the first 1D kernel, and then convolve with the second 1D Kernel. Are the outputs the same? How was the run-time affected?

[7]:
```
import time
img = imread('geisel.jpeg', as_gray=True)

def kernel_2D(img):

    ### YOUR CODE HERE ###
    sobel_2D = None
    ### END YOUR CODE ###

    start_time = time.time()
    I_2D = convolve(img, Sobel_2D, mode='same')
    end_time = time.time()

    print(np.round(start_time-end_time,4))
    plt.imshow(I_2D,cmap = 'gray')
    plt.axis('off')
    plt.title('2D Kernel')
    plt.show()

def kernel_sep(img):

    ### YOUR CODE HERE ###

    sep_1 = None # define the filters
    sep_2 = None
    start_time = time.time()
    I_Separated = None # write your convolution operations here

    ### END YOUR CODE ###

    end_time = time.time()

    print(np.round(start_time-end_time,4))
```

```
    plt.imshow(I_Separated,cmap = 'gray')
    plt.axis('off')
    plt.title('Linearly Separated Kernels')
    plt.show()

kernel_2D(img)
kernel_sep(img)
```

5.0.4 3.3: Filters as Templates [5 pts]

Suppose that you are a clerk at a grocery store. One of your responsibilites is to check the shelves periodically and stock them up whenever there are sold-out items. You got tired of this laborious task and decided to build a computer vision system that keeps track of the items on the shelf.

Luckily, you have learned in CSE 152A (or are learning right now) that convolution can be used for template matching: a template g is multiplied with regions of a larger image f to measure how similar each region is to the template. Note that you will want to flip the filter before giving it to your convolution function, so that it is overall not flipped when making comparisons. You will also want to subtract off the mean value of the image or template (whichever you choose, subtract the same value from both the image and template) so that your solution is not biased toward higher-intensity (white) regions.

The template of a product (template.jpg) and the image of the shelf (shelf.jpg) is provided. We will use convolution to find the product in the shelf.

[14]:
```
import numpy as np
from imageio import imread
import matplotlib.pyplot as plt
from scipy.signal import convolve
%matplotlib inline

# Load template and image in grayscale
img = imread('shelf.jpg')
img_gray = imread('shelf.jpg', as_gray=True)
temp = imread('template.jpg')
temp_gray = imread('template.jpg', as_gray=True)

# Perform a convolution between the image and the template

### START YOUR CODE HERE ###

'''

    CODE GOES HERE


'''
```

```
### END YOUR CODE HERE ###


out = np.zeros_like(img_gray)

# Find the location with maximum similarity
y, x = (np.unravel_index(out.argmax(), out.shape))

# Display product template
plt.figure(figsize=(20,16))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display convolution output
plt.subplot(3, 1, 2)
plt.imshow(out)
plt.title('Convolution output (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

Template

Convolution output (white means more correlated)

Result (blue marker on the detected location)

[ ]:

6 Question 4: Corner Detection, Feature Descriptors and Feature Matching [20 points]

6.0.1 4.1: Corner Detection [10 points]

Corner Detection

Next, you will implement a corner detector to detect photo-identifiable features in the image.

This should be done according to http://cseweb.ucsd.edu/classes/wi20/cse152-a/lec4.pdf. You should fill in the function corner_detect with inputs image, nCorners, smoothSTD, windowSize, where smoothSTD is the standard deviation of the smoothing kernel and windowSize is the window size for Gaussian smoothing, corner detection, and non-maximum suppression. In the lecture the corner detector was implemented using a hard threshold. Do not do that; instead return the nCorners strongest corners after non-maximum suppression. You can use your previous convolution function alongside the previous Gaussian kernel function in order to smooth the image.

For each image, detect 100 corners with a Gaussian standard deviation of 2.0 and a window size of 11. For non-max suppression use a window size of 3. Display your outputs in a matplotlib figure.

[13]:
```
def rgb2gray(rgb):
    """ Convert rgb image to grayscale.
    """
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

def corner_detect(image, nCorners, smoothSTD, windowSize):

    """Detect corners on a given image.

    Args:
        image: Given a grayscale image on which to detect corners.
        nCorners: Total number of corners to be extracted.
        smoothSTD: Standard deviation of the Gaussian smoothing kernel.
        windowSize: Window size for Gaussian smoothing kernel, corner detector,␣
 ↪and non maximum suppression.

    Returns:
        Detected corners (in image coordinate) in a numpy array (n*2).

    """

    ### START CODE HERE###

    corners = np.zeros((nCorners, 2))

    ### END CODE HERE ###

    return corners
```

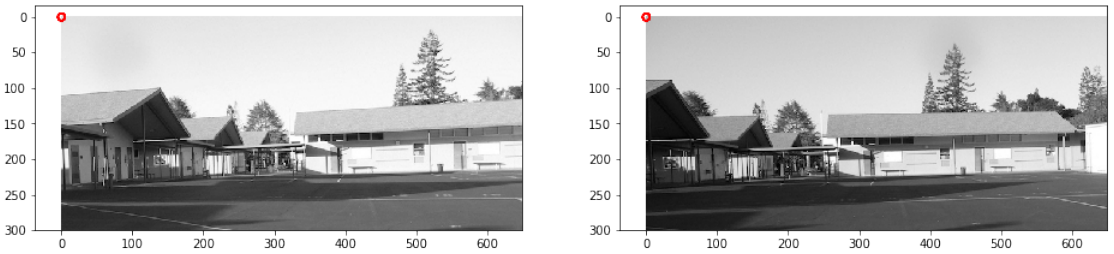
```
# detect corners on the two provided images
# adjust your corner detection parameters here
nCorners = 100
smoothSTD = 2
windowSize = 11

# read images and detect corners on images
imgs = []
corners = []
for i in range(2):
    img = imread('almond' + str(i) + '.jpg')
    imgs.append(rgb2gray(img))
    corners.append(corner_detect(imgs[-1], nCorners, smoothSTD, windowSize))

def show_corners_result(imgs, corners):
    fig = plt.figure(figsize=(16, 16))
    ax1 = fig.add_subplot(221)
    ax1.imshow(imgs[0], cmap='gray')
    ax1.scatter(corners[0][:, 0], corners[0][:, 1], s=35, edgecolors='r',␣
 ↪facecolors='none')

    ax2 = fig.add_subplot(222)
    ax2.imshow(imgs[1], cmap='gray')
    ax2.scatter(corners[1][:, 0], corners[1][:, 1], s=35, edgecolors='r',␣
 ↪facecolors='none')
    plt.show()

show_corners_result(imgs, corners)
```



6.0.2 4.2: Feature Matching using SIFT [10 points]

We have two overlapping images of a scene, and we would like to detect and match features (corners) across them. In order to do so, we will compute feature descriptors for the patches around the points and perform matching using a "sum of squared differences" metric.

First, you will want to implement a function which computes said metric. Given two vectors, `ssd` should return the sum of squared differences of their values:

$$ssd(v, w) = \sum_i (v_i - w_i)^2$$

[ ]:
```
def ssd(v1, v2):
    """Compute SSD (sum of squared differences) for two NumPy arrays.
    This function will come in handy for descriptor matching later.
    """

    ### START CODE HERE ###


    ### END CODE HERE ###
    return None
```

Now use your `ssd` function to perform feature matching. In the `SIFT_matching` function, you should extract SIFT descriptors for the patches around the corners you've detected, then give each potential pair of matching descriptors to the `ssd` function to compute the distance between them. Also make use of Lowe's nearest neighbor thresholding idea, which says that the similarity of the closest match should be much higher than the similarity of the next-closest match. You can check this using the ratio of the SSD distance with the closest match and the next-closest match. By Lowe's empirical findings, you should only accept the best match if the ratio between its distance and the second-best distance is ~0.75 or less. For better-looking results, please use a ratio of 0.3 for the purposes of this homework.

We will use OpenCV's implementation of SIFT. To install OpenCV with pip, you can run

```
pip install opencv-python==3.4.2.16
pip install opencv-contrib-python==3.4.2.16
```

Please use this version of OpenCV, as earlier or later versions may not have everything you need. After creating a SIFT object with `cv.xfeatures2d.SIFT_create`, you should use the `sift.compute` function, not `sift.detectAndCompute` function, since we still want to use the corners that we've detected.

To summarize: in this question, you should extract SIFT descriptors for each corner. Then you should match SIFT descriptors according to an SSD metric. Check the ratio of the best match's distance to the second best match's distance and only accept the best match if it is significantly better than the next best match (i.e. it involves a significantly lower distance).

If necessary, tune the parameters so that you end up with about 20 feature matches (starting with your 100 corners from the previous question).

[16]:
```
import cv2

def SIFT_matching(img1, img2, corners1, corners2, SSDth, nn_threshold):
    """Compute matchings between two windows based on SIFT descriptors.
```

```
    Args:
        img1: Image 1.
        img2: Image 2.
        corners1: Corners in image 1 (nx2)
        corners2: Corners in image 2 (nx2)
        SSDth: SSD distance threshold
        NNth: nearest neighbor threshold

    Returns:
        matching result - a list of (c1, c2) tuples, where
        c1 is the 1x2 corner location in image 1, and
        c2 is the 1x2 corner location in image 2.

    """



    ### YOUR CODE HERE ###
    """
    (feel free to change the below code, it is only provided for reference)
    (you do not need to print out keypoints or descriptors in this function)
    """

    ### END CODE HERE

    if img1.dtype == np.float64:
        img1 = (img1 * 255.0).astype(np.uint8)
    sift = cv2.xfeatures2d.SIFT_create()
    keypoints1 = [cv2.KeyPoint(c[0], c[1], 1) for c in corners1]
    keypoints1, descriptors1 = sift.compute(img1, keypoints1)
    corners1 = [(kp.pt[0], kp.pt[1]) for kp in keypoints1]
    print('%d keypoints' % len(keypoints1))
    print(corners1)
    print('%d descriptors' % len(descriptors1))
    print(descriptors1)

    matching = []
    return matching

# plot matching result
def show_matching_result(img1, img2, matching):
    fig = plt.figure(figsize=(16, 16))
    plt.imshow(np.hstack((img1, img2)), cmap='gray')
    for p1, p2 in matching:
        plt.scatter(p1[0], p1[1], s=35, edgecolors='r', facecolors='none')
        plt.scatter(p2[0] + img1.shape[1], p2[1], s=35, edgecolors='r',␣
 ↪facecolors='none')
```

```
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
    plt.show()

# match corners
SSDth = 50000
NNth = 0.3
matching = SIFT_matching(imgs[0]/255, imgs[1]/255, corners[0], corners[1],␣
 ↪SSDth, NNth)
show_matching_result(imgs[0], imgs[1], matching)
```

6.1 Submission Instructions

Remember to submit a PDF version of this notebook to Gradescope. Please make sure the contents in each cell are clearly shown in your final PDF file.

There are multiple options for converting the notebook to PDF: 1. You can find the export option at File → Download as → PDF via LaTeX 2. You can first export as HTML and then convert to PDF 3. Convert to a Latex document and use overleaf to convert to PDF (very useful if working on windows)

[ ]: