ECE661: Homework 4

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

September 26, 2018

1 Logic and Methodology

For this Homework we are given two images of a single scene taken from two different viewpoints and we are required to extract **interest points** from the two image and then establish correspondences between interest points in the two images.

For detection of interest points we will be using the following methods:

a) Harris Corner Detector

b) SIFT/SURF points

For establishing correspondences we will be using the following methods:

a) Normalized Cross Correlation

b) Sum of Squared Differences

c) Euclidean Distance for SIFT/SURF

After completion of the above two tasks we need to plot the two images in single image and draw lines connecting matching points.

1.1 Corner Detection

1.1.1 Harris Corner Detector

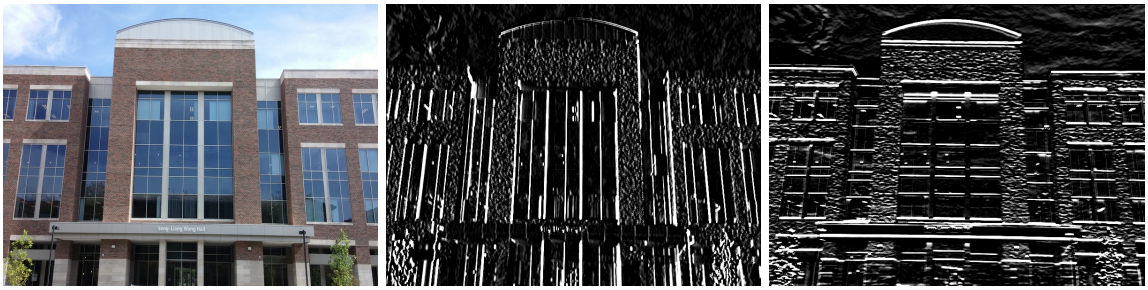
Harris Corner detector works on the principle of identifying the points whose vicinity have a significant gray level **variations** in two different directions. In order to evaluate the **variations** ,(d_x,d_y) in two directions we convolute given image with a differential operator such as **Haar** filter.

Haar Filter The Haar filter is a MxM operator, where M is the smallest even integer greater than 4σ, σ is the scale value. So the operators (d_x,d_y) are the x and y derivative estimates at a pixel scale of σ.

The Haar filters look like this: for $\sigma = 1.2 \Rightarrow M = 6$

$$d_x = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix} \quad d_y = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

when operated upon an image we can see the x & y derivatives of the image as shown in the following figure:



(a) original image (b) dx image (c) dy image

Figure 1: Effect of Haar filters with $\sigma = 1.2$

Note: The Haar filter is an even sized operator and will not have a definite center compared to odd sized operators like the LOG operator. So, when convoluting the image with the Haar filter the convention is to pick $(\frac{M}{2}+1)$th column and row as the center location. For instance, for a 4x4 Haar filter the columns and rows will be indexed as $[-2,-1,0,1]$ and the $(0,0)$ element will be taken as the center for convolution.Also, the image can be padded using a pad thickness of $M/2$.

Now that we have found out (d_x,d_y), we need construct a 5σx5σ window at every pixel of the given image and construct a matrix C given by:

$$C = \begin{bmatrix} (\sum d_x)^2 & \sum d_x d_y \\ \sum d_x d_y & (\sum d_y)^2 \end{bmatrix}$$

where the summations are over the $5\sigma x5\sigma$ window. The Matrix C has a **full rank**(ie Det$(C)\neq 0$) at a genuine corner. Also as stated earlier, a corner is a point where we have significant variation in two directions, this leads us to investigate in to the eigen vector directions and their corresponding weights (ie the eigen values). Assuming that the eigen values are such that $\lambda_1 \geq \lambda_2$, then for a point to be a harris corner we set a threshold on the ratio $\frac{\lambda_2}{\lambda_1}$.

Also, computationally finding the eigen values at each and every pixel can be quite costly. So, we can estimate the ratio using the following relation: $r = \frac{\lambda_2}{\lambda_1}$

$$tr(C) = \lambda_2 + \lambda_1 = (\sum d_x)^2 + (\sum d_y)^2$$

$$det(C) = \lambda_2 * \lambda_1 = ((\sum d_x)^2 * (\sum d_y)^2) - (\sum d_x d_y)^2$$

$$\frac{det(C)}{tr(C)^2} = \frac{\lambda_2 * \lambda_1}{(\lambda_2 + \lambda_1)^2} = \frac{r}{(1+r)^2}$$

which means we can set a threshold on the value of $\frac{det(C)}{tr(C)^2}$ and avoid solving the eigen value problem.

Discussion on filtering As we will be iterating over all pixel coordinates, the interest points detected by harris corners at any scale σ will be present in clusters. Therefore, we need to filter out the most suitable point from the clusters as our canditate for the key point. For, this we will simply iterate over all raw points and check if the point has the highest value in its neighbourhood and also if it passes a global threshold value. The points that satisy these criteria will qualify as our output.

The function written out in Python for detection of Harris corners and for generating Haar filters is:

```
#harris corner detection
def Harris_Corners(img,scale):
    """
    Input: img= image object for which corners are to be detected
        scale= smoothing scale for image
    Output: pts=[[x,y],[],[]..] List of corner coordinate
        Ix: xderivative Image
        Iy: y derivative image
    """
    #find derivatives of image dx and dy
    dx,dy=Haar_filter(scale)
    #convolve dx dy to get Ix Iy
    Ix=cv2.filter2D(img,-1,dx) #by default pixelExtrapolation at borders
    Iy=cv2.filter2D(img,-1,dy)
    #pad Ix and Iy for next step
    #find pad
    window_size=int(np.ceil(5*scale))
    if window_size%2==0: window_size+=1
    pad=int((window_size-1)/2.0)
    Ix_pad=cv2.copyMakeBorder(Ix,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    Iy_pad=cv2.copyMakeBorder(Iy,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    R=np.zeros(np.shape(img))#stores info of eigenvalue ratios
    #loop over pixels and update R if a corner
    for i in range(np.shape(img)[0]):
        for j in range(np.shape(img)[1]):
            #take slice of Ix Iy with window of 2*pad+1 size
            Ix_window=Ix_pad[(i+pad)-pad:(i+pad)+pad+1,(j+pad)-pad:(j+pad)+pad+1]
            Iy_window=Iy_pad[(i+pad)-pad:(i+pad)+pad+1,(j+pad)-pad:(j+pad)+pad+1]
            #find Ix**2 Iy**2 IxIy
            Ix2=np.sum(np.multiply(Ix_window,Ix_window))
            Iy2=np.sum(np.multiply(Iy_window,Iy_window))
            Ixy=np.sum(np.multiply(Ix_window,Iy_window))
            #now C=[[Ix2,Ixy],[Ixy,Iy2]] check if a corner
            #using determinant of C as check for rank2
            detC=Ix2*Iy2-(Ixy**2)
            if detC>0:
                R[i,j]=detC/((Ix2+Iy2)**2)
    #remove points on the edges of image upto a pixel wdith of p pixels
    p=10
    R[:p,:]=0*R[:p,:]#top border
    R[-p:,:]=0*R[-p:,:]#bottom border
    R[:,:p]=0*R[:,:p]#left border
    R[:,-p:]=0*R[:,-p:]#right border
    #Suppression of non-maxium points in vicinity of a maxima
    #so that we dont get a cluster of points at a single location
    non_max_window=31 #pixels, should be odd
    threshold=np.mean(R)
    pad = int((non_max_window-1)/2)
    R_pad=cv2.copyMakeBorder(R,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    pts=[]
    for i in range(np.shape(img)[0]):
        for j in range(np.shape(img)[1]):
            if R[i,j]>0:
                R_window=R_pad[(i+pad)-pad:(i+pad)+pad+1,(j+pad)-pad:(j+pad)+pad+1]
                if R[i,j]==np.max(R_window) and R[i,j]>threshold:
                    pts.append((j,i))# x is j col y is i row
    return(pts,Ix,Iy)

#function for haar filter
def Haar_filter(scale):
    """
    Input: scale= defines the sigma for the filter and also the size of filter
    Output: dx, dy haar filter kernels
    """
    N=int(np.ceil(4*scale)) # size of kernel is smallest even integer greater than 4*sigma
```

```
if N%2!=0:
    N=N+1 #if N was odd: add 1 to it
dx=np.ones((N,N))
dy=np.ones((N,N))
dx[:,:int(N/2)]=-1*dx[:,:int(N/2)]
dy[:int(N/2),:]=-1*dy[:int(N/2),:]
return(dx,dy)
```

1.1.2 SIFT Corner Detection

SIFT stands for Scale-Invariant Feature Transform and is a robust method for finding key points which are invariant to scale, orientation, and illumination. The method was developed by **David G. Lowe** and a detailed explanation of the method can be found in his paper at `https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf`.

SIFT involves the following steps:

Step-1: Extrema detection from DOG pyramid After constructing a DOG pyramid $D(x, y, \sigma)$ by taking differences of σ smoothed images at different octave ans sub-octave levels, we can then take the DOG as an approximation to LOG and estimate wether any point (x, y, σ) is a local extrema or not within its 3x3x3 volumetric neighbourhood.
We do this step so as to get atleast two different values of σ within each octave.

Step-2: KeyPoint Localization After identifying the extrema points from step-1, we imporve our solution by estimating the true extrema point using Taylor series in the vicinity of the extrema point.

$$D(\vec{X}) = D(\vec{X_0}) + J^T(\vec{X_0})\vec{x} + \frac{1}{2}\vec{X^T}H(\vec{X_0})\vec{X^T}$$

As at the true extrema point the derivative of $\frac{\partial D(\vec{X})}{\partial \vec{X}} = 0$.
and thus $H(\vec{X_0})\vec{X} = -J(\vec{X_0})$ is used to solve for $\vec{X}$ which is the true extremum.

Step-3: Removal of weak extrema After identification of $\vec{X} = (x, y, \sigma)$ from step-2, we remove the extrema points with $|D(\vec{X})| \leq 0.03$

Step-4: Identify Dominant Local Orientation In order to make the descriptors invariant to in-plane roatation, we identify the dominant local orientation associated to each extrema point. This is done by constructing a KxK window in xy plane around an extremum and then evaluate the magintude and orientation of the σ smoothed image ($ff(x, y, \sigma)$) using the formulas:

$$m(x, y) = \sqrt{|ff(x+1, y, \sigma) - ff(x, y, \sigma)|^2 + |ff(x, y+1, \sigma) - ff(x, y, \sigma)|^2}$$

$$\theta(x, y) = tan^{-1}(\frac{ff(x, y+1, \sigma) - ff(x, y, \sigma)}{ff(x+1, y, \sigma) - ff(x, y, \sigma)})$$

After identifying the magnitudes and orientation at all the pixels in the KxK neighbourhood we construct a histogram with 36 bins for the full range of 360°. So we will have 36 columns/bars in the histogram and the heights of the bars is the sum of all the magnitudes whose θ falls in that region. The Peak of the parabola gives us the dominant orientation.

Step-5: Construction of descriptor We now construct a 128-bit descriptor. In this step, we construct a 16x16 neighbour of points accross the extremum and divide the region into 4x4 cells. The magnitudes of gradient are weighted by a guaasian whose σ is half the weidth of neighbourhood. Then for each cell an 8 bin orientation histogram is calculated, with the new orientation wrt the dominant local orientation. We then concatenate the 8 bin histograms for 16 cells to obraint a $16 * 8 = 128$ bit descriptor. We then normalize this decriptor which makes it inavriant to illumination.

Python function for SIFT The documentation for the python function for the SIFT can be found at `https://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html`.
When we use the detectAndCompute function in OpenCV it gives us a KeyPoint structure and descriptor. It is important to know how the information is organized in the KeyPoint structure which can be found at `https://docs.opencv.org/3.4/d2/d29/classcv_1_1KeyPoint.html`

1.2 Feature Correspondences

After detection of corner points using the methods stated in section 1.1 , we now have to correlate similar points in the two images. We will discuss three methods for this task. However, the last method discussed in this section is only suitable for cases when we have a descriptor associated with the location of an interest point (ie SIFT or SURF).

1.2.1 Normalized Cross Correlation(NCC)

In this method, we evaluate correlation between any and every pair of interest points. For this, we construct a $(M+1)*(M+1)$ window at the two pixel locations in the two images and evaluate the NCC metric using:

$$NCC = \frac{\sum_i \sum_j (f_1(i,j) - m_1)(f_2(i,j) - m_2)}{\sqrt{\sum_i \sum_j (f_1(i,j) - m_1)^2 \sum_i \sum_j (f_2(i,j) - m_2)^2}}$$

where $f_1(i,j)$ and $f_2(i,j)$ are the pixel values at (i,j) neighbour for image 1 and 2 respectively and m_1 and m_2 are the means of the pixel values over the complete window in image 1 and 2 respectively.

We then set a fixed threshold on the value of NCC to get a matching point pair. For my code, I have chosen a fixed value of **0.95** as the threshold for NCC.

The python code for evaluating NCC and giving point pairs is as follows:

```
#function for Normalized Cross Correlation for point correspondences
def NCC(img1,pts1,img2,pts2):
    """
    Input: img1, img2 = grayscale images of a scene from different viewpoints, sizes are
        identical
            pts1,pt2 =[(x,y),...] list of tuples interest points detected for img1 and img2
                respectively
    Output: pt_pairs= [[(x1,y1),(x2,y2)],....] list of list of point pair tuples which pass
        a certain
            confidence threshold
    """
    M=31 # window is of size M
    pad = int((M-1)/2)
    #pad img1 and img2
    img1_pad=cv2.copyMakeBorder(img1,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    img2_pad=cv2.copyMakeBorder(img2,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    # loop over length of pts1 and pts2: exhaustive search
    threshold=0.95
    pt_pairs=[]

    for ipts1 in pts1:
        NCC=np.zeros((len(pts2),1))
```

```
            k=0
            for ipts2 in pts2:
                #make window of size MxM at both the point locations
                win1=img1_pad[ipts1[1]+pad-pad:ipts1[1]+pad+pad+1,ipts1[0]+pad-pad:ipts1[0]+pad+
                    pad+1]
                win2=img2_pad[ipts2[1]+pad-pad:ipts2[1]+pad+pad+1,ipts2[0]+pad-pad:ipts2[0]+pad+
                    pad+1]
                #find means for both the window
                mean1=np.mean(win1)
                mean2=np.mean(win2)
                #find NCC
                num=np.sum(np.multiply( (win1-mean1*np.ones((M,M))),(win2-mean2*np.ones((M,M))))
                    )
                deno=np.sum(np.power((win1-mean1*np.ones((M,M))),2))
                deno=deno*(np.sum(np.power((win2-mean2*np.ones((M,M))),2)))
                deno=np.sqrt(deno)
                NCC[k]=num/deno
                k+=1
            #check if NCC_max is greater than threshold
            NCC_max=np.max(NCC)
            if NCC_max>threshold:
                i_max=np.argmax(NCC)
                pt_pairs.append([ipts1,pts2[i_max]])
        return(pt_pairs)
```

1.2.2 Sum of Squared Differences (SSD)

AS the name suggests, in this method we calculate the sum of squared differences of pixel values in a $(M+1)*(M+1)$ window for each and every pair of interest points. The metric for SSD is given as follow:

$$SSD = \sum_i \sum_j |f_1(i,j) - f_2(i,j)|^2$$

where $f_1(i,j)$ and $f_2(i,j)$ are the pixel values at (i,j) neighbour for image 1 and 2 respectively. It can be seen clearly that for SSD we will need to have a lower threshold as this metric is defining the "difference" in the two points.

For my code, I have normalized the list of SSD values using the minimum and maximyum values to bring it to the range of $0-1$ and then use a threshold value of 0.25.

The python code for evaluating SSD and giving point pairs is as follows:

```
#function for Squared Sum Differences for Point correspondences
def SSD(img1,pts1,img2,pts2):
    """
    Input: img1, img2 = grayscale images of a scene from different viewpoints, sizes are
        identical
            pts1,pt2 =[(x,y),...] list of tuples interest points detected for img1 and img2
                respectively
    Output: pt_pairs= [[(x1,y1),(x2,y2)],....] list of list of point pair tuples which pass
        a certain
            confidence threshold
    """
    M=51 # window is of size M
    pad = int((M-1)/2)
    #pad img1 and img2
    img1_pad=cv2.copyMakeBorder(img1,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    img2_pad=cv2.copyMakeBorder(img2,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
    # loop over length of pts1 and pts2: exhaustive search
    pt_pairs=[]
    SSD=np.zeros((len(pts1),len(pts2)))
```

```
    i=0
    for ipts1 in pts1:
        j=0
        for ipts2 in pts2:
            #make window of size MxM at both the point locations
            win1=img1_pad[ipts1[1]+pad-pad:ipts1[1]+pad+pad+1,ipts1[0]+pad-pad:ipts1[0]+pad+
                pad+1]
            win2=img2_pad[ipts2[1]+pad-pad:ipts2[1]+pad+pad+1,ipts2[0]+pad-pad:ipts2[0]+pad+
                pad+1]
            #find SSD
            SSD[i,j]=np.sum(np.power((win1-win2),2))
            j+=1
        i+=1
    #using a dynamic threshold as criteria for matching points
    #converting SSD to 0-1 range and then using a dynamic threshold
    SSD=(SSD-np.min(SSD)*np.ones(np.shape(SSD)))/(np.max(SSD)-np.min(SSD))
    dy_threshold=0.25
    i=0
    for ipts1 in pts1:
        #check if SSD_max is less than dy_threshold
        SSD_min=np.min(SSD[i,:])
        if SSD_min<dy_threshold:
            j_min=np.argmin(SSD[i,:])
            pt_pairs.append([ipts1,pts2[j_min]])
        i+=1
    return(pt_pairs)
```

1.2.3 Euclidean Distance for SIFT/SURF

This method is suitable only when we have a descriptor at any interest point. Therefore, a suitable choice for SIFT and SURF. In this method, we evaluate the euclidean distance between two points using the descriptor vector and then set a threshold to identify valid pairs. The metric is given as follow:

$$distance = ||d_1 - d_2||$$

where the norm is an L2 Norm and d_1 & d_2 are the descriptors at the points in image 1 and image 2 respectively.
To find the threshold for the euclidean distance we take a similar approach to that was taken for SSD. We normalize the list of the Euclidean distance and then set the threshold as **2** to find a valid point pair.

Note: In order to use my own function for Euclidean matching, I had to convert the output given by SIFT function to my format.

The python code for evaluating the Euclidean distance and give out point pairs is as follows:

```
#function for Euclidean match for SIFT points
def Euclidean_sift_match(pts1,des1,pts2,des2):
    """
    Input: pts1,pts2= [(x,y),()...] list of tuples of interest points
           des1,des2 = vector of 128 size, sift descriptor
    Output: pt_pairs= [[(x1,y1),(x2,y2)],....] list of list of point pair tuples such that
            Euclidean distance is less than a dynamic threshold
    """
    pt_pairs=[]
    sq_eu=np.zeros((len(pts1),len(pts2)))
    for i in range(len(pts1)):
        for j in range(len(pts2)):
            sq_eu[i,j]=np.linalg.norm(des1[i,:]-des2[j,:])

    sq_eu=sq_eu/np.min(sq_eu)
    dy_threshold=2
    print(dy_threshold)
    for i in range(len(pts1)):
        sq_eu_min=np.min(sq_eu[i,:])
```

```
            if sq_eu_min<dy_threshold:
                j_min=np.argmin(sq_eu[i,:])
                pt_pairs.append([pts1[i],pts2[j_min]])
    return(pt_pairs)

#function for converting sift key points into my format
def convert_SIFT_myFormat(kp):
    """
    Input: kp: keypoint structure given by SIFT, point coordinates in kp.pt
    Output: [(x,y),...] : list of tuples of coorinates
    """
    mykp=[]
    for ikp in kp:
        #convert points to floored integer
        mykp.append((int(ikp.pt[0]),int(ikp.pt[1])))
    return(mykp)
```

1.3 Plotting

Now that we have the information of interest points and matching pairs from the above two tasks we need to plot the two images in single canvas and draw lines connecting matching points. This is a trivial task and will need no explanation.
The Python code for plotting is:

```
#function for plotting image with interest points of two scenes and lines joining matching
    points
def plot_matching_points(img1,pts1,img2,pts2,pt_pairs):
    """
    Input: img1,img2: colored image matrices of same row size of the same scene from diff
        viewpoints
            pts1,pts2: [(x,y),..] list of tuples of intresting points in img1 and img2
                respectively
            pt_paris:[ [(x1,y1), (x2,y2)],... ] list of list of tuples of point pairs with
                matching metric
    OutPut: img3: image matrix with both scenes placed side by side horizontally,
            with marked points of interests in both the images and lines joining elements of
                 pt_pairs
    """
    m=np.shape(img1)[0];n1=np.shape(img1)[1];n2=np.shape(img2)[1]
    img3=np.zeros((m,n1+n2,3))
    #assign img1 and img2 pixels to img3
    img3[:,:n1,:]=img1
    img3[:,n1:,:]=img2
    #draw interest points on both images
    for ipts1 in pts1: cv2.circle(img3,ipts1,2,(0,0,255),-1)
    for ipts2 in pts2:
        x=n1+ipts2[0]
        cv2.circle(img3,(x,ipts2[1]),2,(0,0,255),-1)
    #draw line joining matching points
    for i in range(len(pt_pairs)):
        cv2.line(img3,pt_pairs[i][0],(pt_pairs[i][1][0]+n1,pt_pairs[i][1][1]),(0,255,0),1)
    return(img3)
```

2 Results

In the following images the Interest points detected are highlighted in red dots. Also, matching point pairs are joined by a green line.

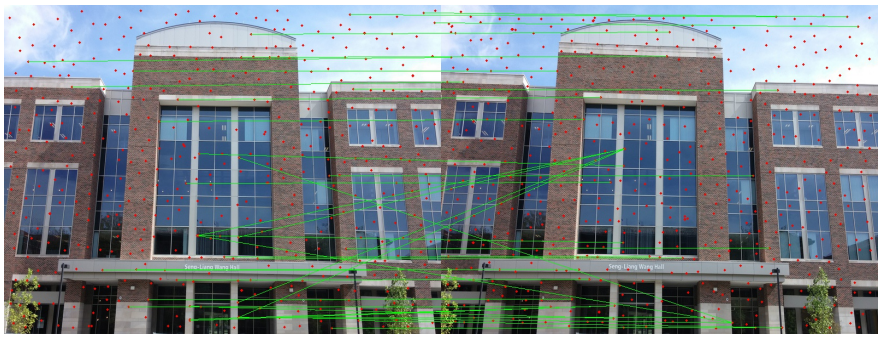


Figure 2: results of image 1 with scale=1 using NCC

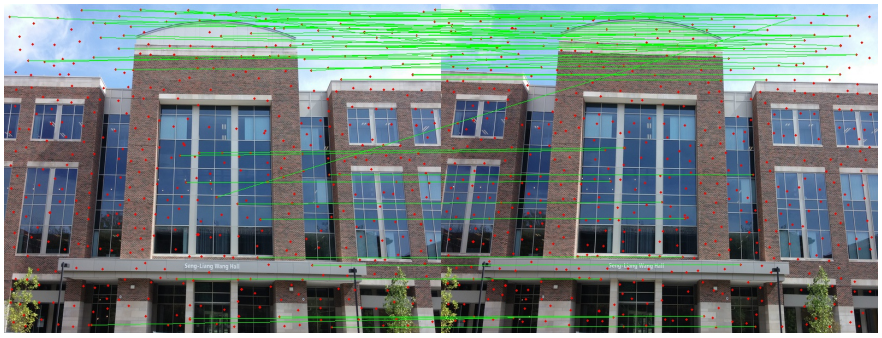


Figure 3: results of image 1 with scale=1 using SSD

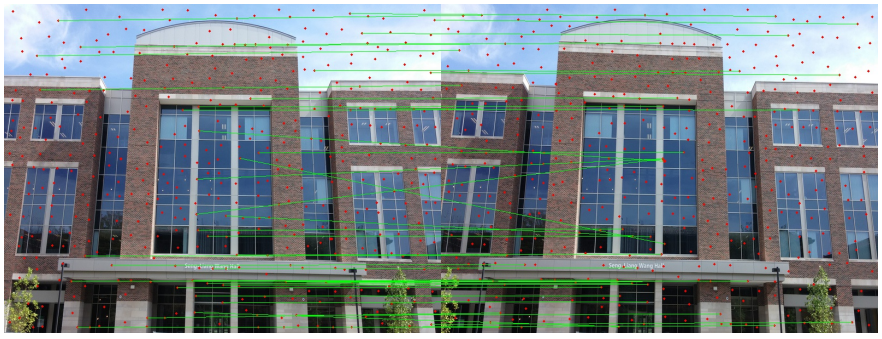


Figure 4: results of image 1 with scale=2 using NCC

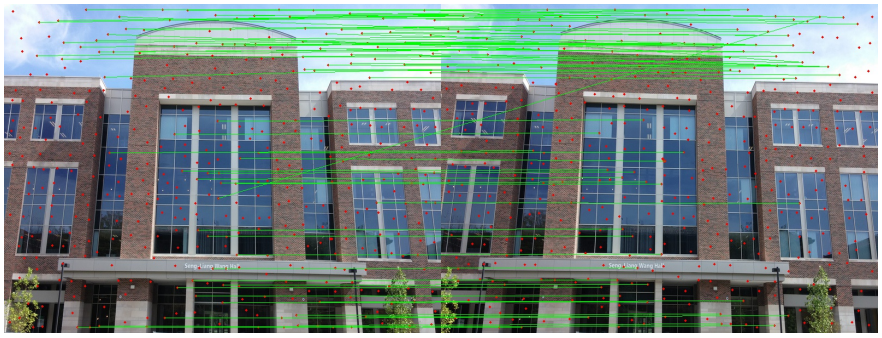


Figure 5: results of image 1 with scale=2 using SSD

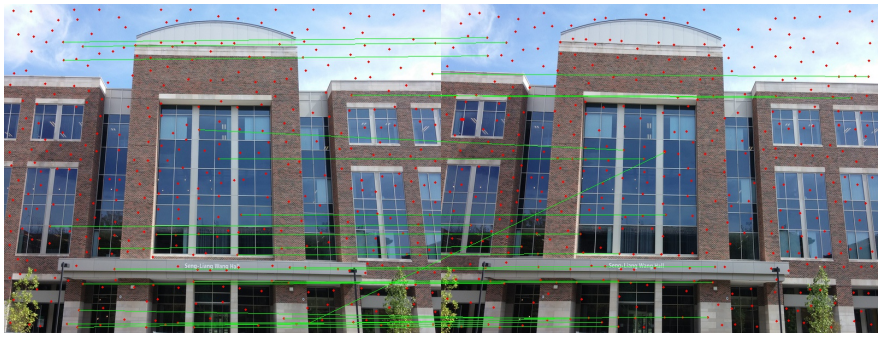


Figure 6: results of image 1 with scale=3 using NCC

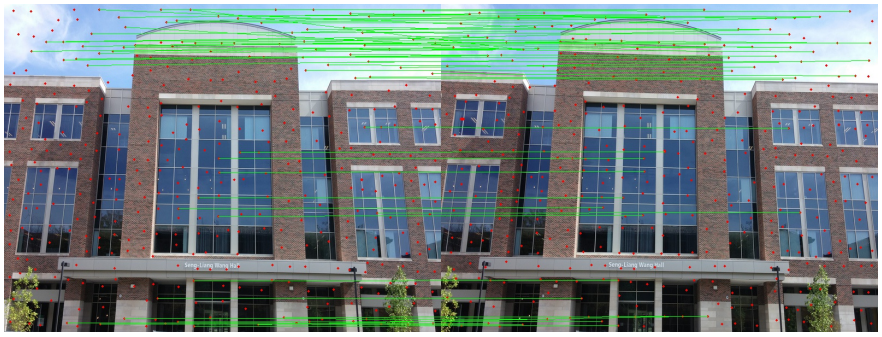


Figure 7: results of image 1 with scale=3 using SSD

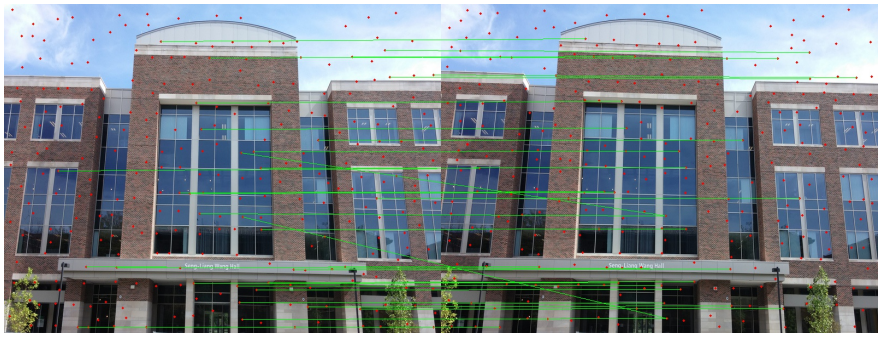


Figure 8: results of image 1 with scale=4 using NCC

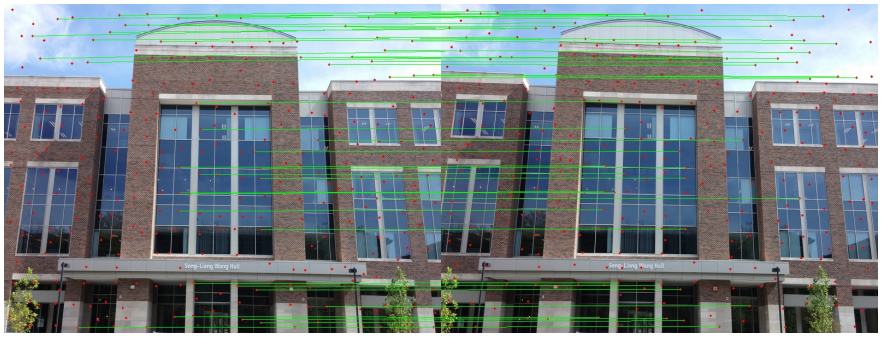


Figure 9: results of image 1 with scale=4 using SSD

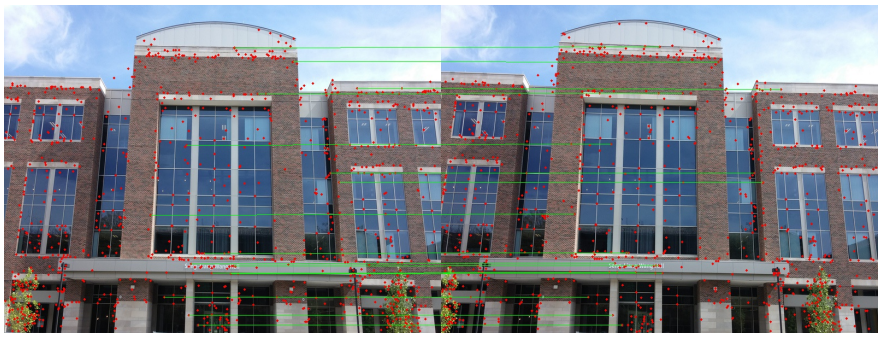


Figure 10: results of image 1 using SIFT and Euclidean distance

Figure 11: results of image 2 with scale=1 using NCC

Figure 12: results of image 2 with scale=1 using SSD

Figure 13: results of image 2 with scale=2 using NCC

Figure 14: results of image 2 with scale=2 using SSD

Figure 15: results of image 2 with scale=3 using NCC

Figure 16: results of image 2 with scale=3 using SSD

Figure 17: results of image 2 with scale=4 using NCC

Figure 18: results of image 2 with scale=4 using SSD

Figure 19: results of image 2 using SIFT and Euclidean distance

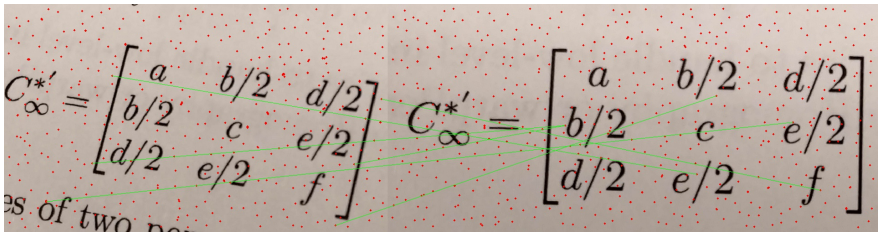


Figure 20: results of image 3 with scale=1 using NCC

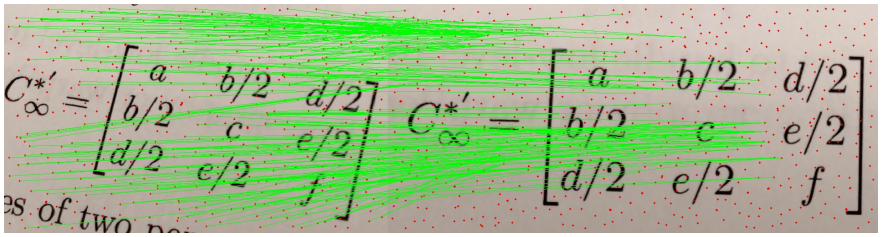


Figure 21: results of image 3 with scale=1 using SSD

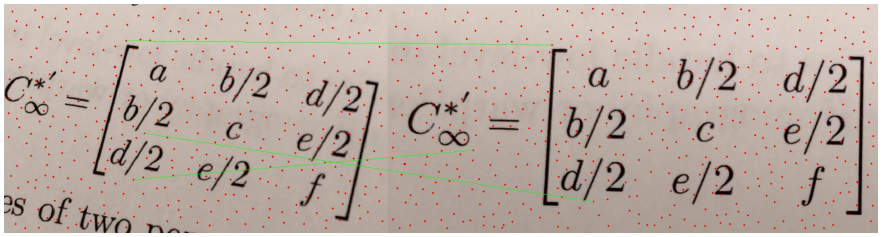


Figure 22: results of image 3 with scale=2 using NCC

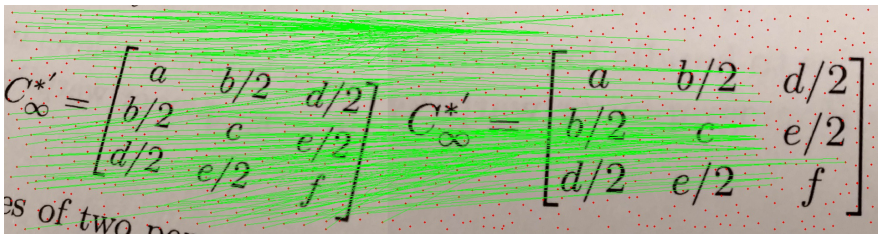


Figure 23: results of image 3 with scale=2 using SSD

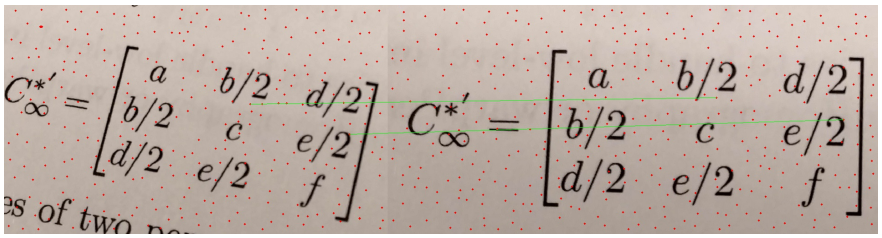


Figure 24: results of image 3 with scale=3 using NCC

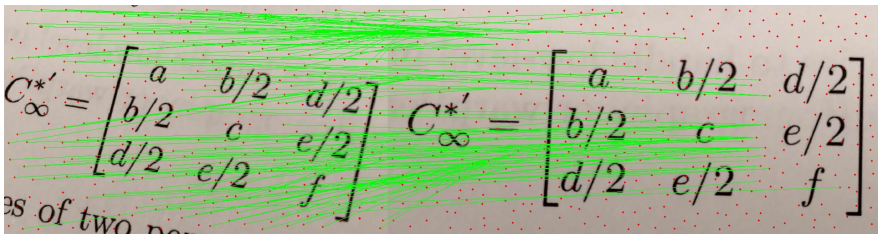


Figure 25: results of image 3 with scale=3 using SSD

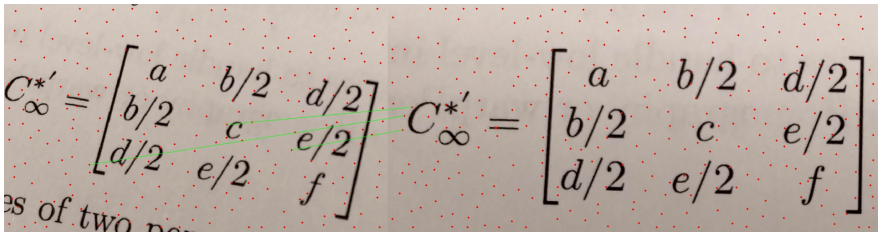


Figure 26: results of image 3 with scale=4 using NCC

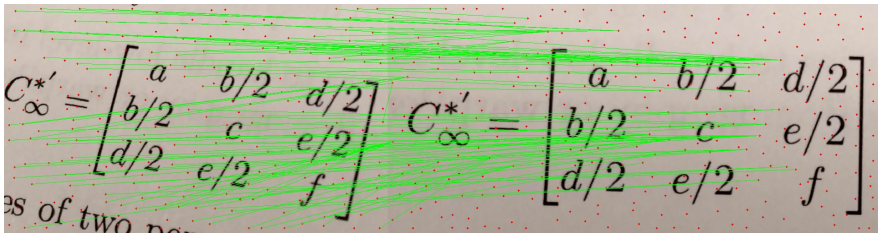


Figure 27: results of image 3 with scale=4 using SSD

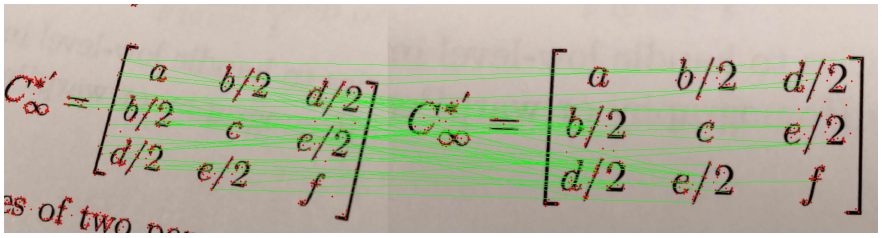


Figure 28: results of image 3 using SIFT and Euclidean distance

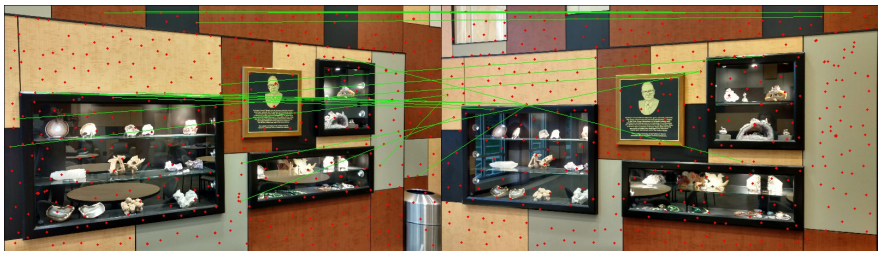


Figure 29: results of image 4 with scale=1 using NCC

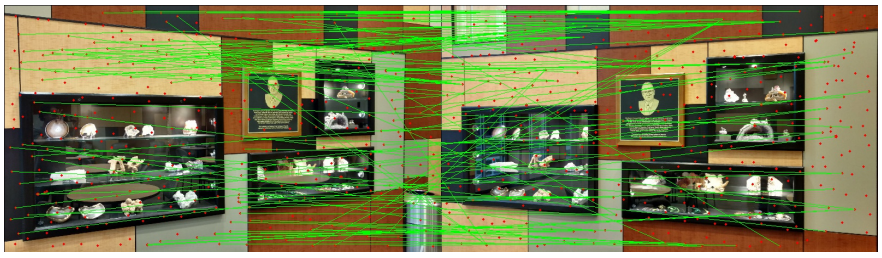


Figure 30: results of image 4 with scale=1 using SSD

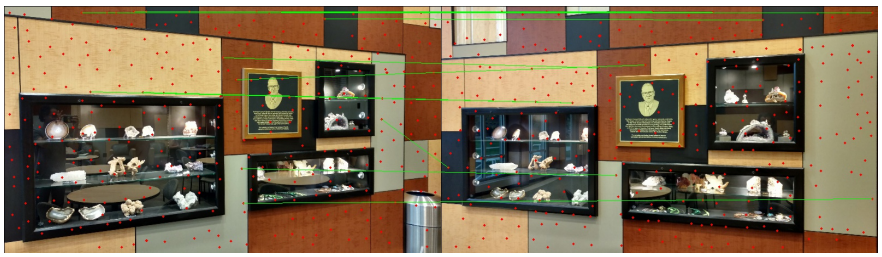


Figure 31: results of image 4 with scale=2 using NCC

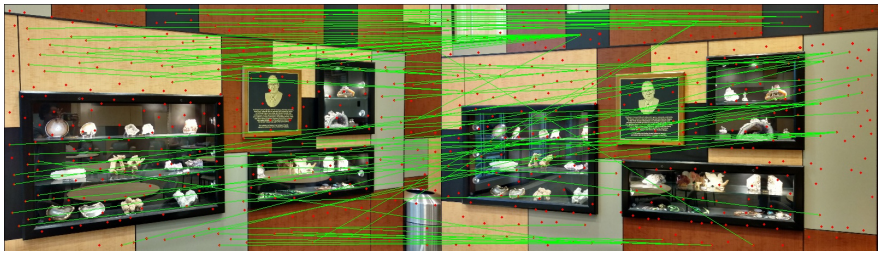


Figure 32: results of image 4 with scale=2 using SSD

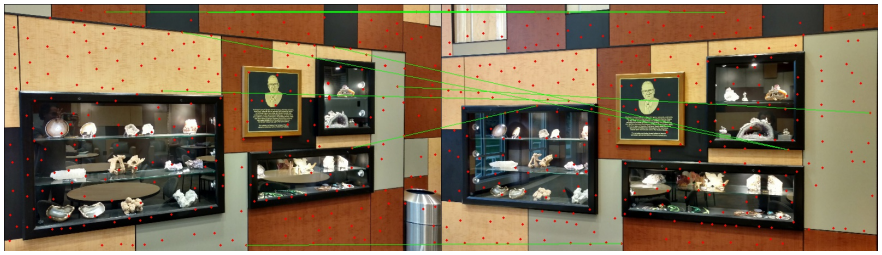


Figure 33: results of image 4 with scale=3 using NCC

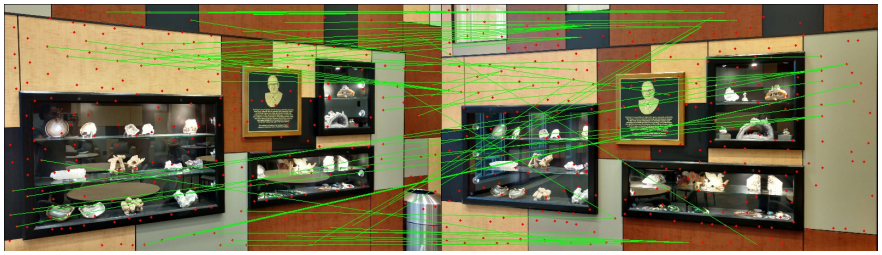


Figure 34: results of image 4 with scale=3 using SSD

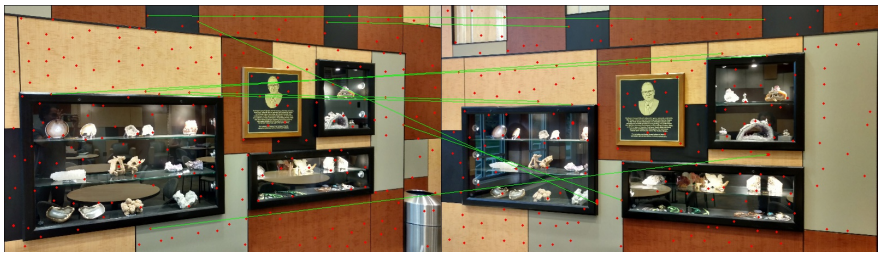


Figure 35: results of image 4 with scale=3 using NCC

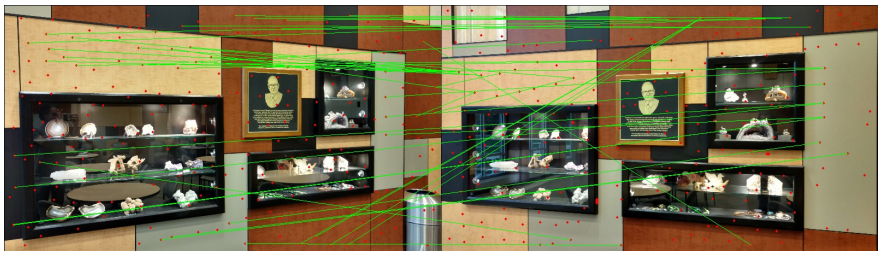


Figure 36: results of image 4 with scale=3 using SSD

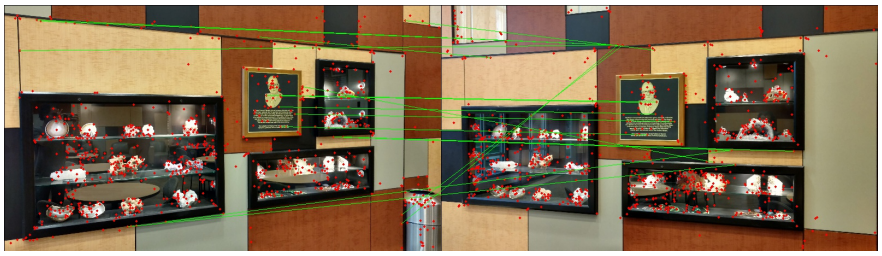


Figure 37: results of image 4 using SIFT and Euclidean distance

3 Observations:

Observation on Interest Points extraction:

Harris Corner detector The harris corner method gave points pretty much all over the image. And was found to be affected by illumination which means that even for a plain background with slight changes in illumination it will give us false interest points. Also, When using Harris Corner detectot with different scale values, the number of corner points did reduce higher values of σ but the points detected at higher scale were still all over the place.

SIFT This method gave better interest points which were related to actual points in the image. Also, as SIFT gives a descriptor for every interest point this makes it more useful when we carry out feature matching.

Observations on Point Correspondences Methods:

Normalized Cross Correlation This method was used for points detected using Harris Corner detector. From observation the results of the matching points given by this method were more accurate than Sum of Squared differences method. Another advantage of this method was that we did not need to implement a dynamic threshold for this method. However, as we are calculating the metric at each pixel therefore this method is relatively doing computationally costly than Sum of Squared Differences.

Sum of Squared Differences This method was also used for points detected using Harris Corner detector. From observation the results of matches were not as good as NCC but, as we are doing less computation at each pixel level therefore this method is relatively computationally cheaper. Also, in this method coming up with a criteria for dynamic threshold is not clear as the metric values represent a random data, thus making it difficult to find a cutoff value.

Euclidean Distance This method is only applicable when we have a descriptor associated with the key point. This mehtod is the most easiest to implement and was found to give the best correlation results.

4 Source Code

The Complete code for carrying out the whole task is:

```
"""
ECE661: hw4
@author: rahul deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""

#import libraries
import numpy as np
import cv2
import sys
sys.path.append('../../')
import MyCVModule as MyCV
#define path
readpath='../HW4Pics/' # path of images to be read
savepath='../results/' #path for saving results of images
imgname='1'
#read image in grayscale
img1=cv2.imread(readpath+imgname+'scene1.jpg',0)
img2=cv2.imread(readpath+imgname+'scene2.jpg',0)
img1color=cv2.imread(readpath+imgname+'scene1.jpg',-1)
img2color=cv2.imread(readpath+imgname+'scene2.jpg',-1)
#----------Harris corner detection----------##
scale_list=[1,2,3,4]
for iscale in range(len(scale_list)):
    print(scale_list[iscale])
    scale=scale_list[iscale]
    print('NCC')
    pts1,Ix1_g,Iy1_g=MyCV.Harris_Corners(img1,scale); print('pts1 done');print(len(pts1))
    pts2,Ix2_g,Iy2_g=MyCV.Harris_Corners(img2,scale); print('pts2 done');print(len(pts2))

    pt_pairs_NCC=MyCV.NCC(img1,pts1,img2,pts2); print('pt_pairs_NCC done');print(len(
        pt_pairs_NCC))
    NCC_img=MyCV.plot_matching_points(img1color,pts1,img2color,pts2,pt_pairs_NCC); print('
        combined images')
    cv2.imwrite(savepath+imgname+'_'+str(iscale)+'_NCC.jpg',NCC_img); print('image saved')

    print('SSD')
    pt_pairs_SSD=MyCV.SSD(img1,pts1,img2,pts2);print(len(pt_pairs_SSD))
    SSD_img=MyCV.plot_matching_points(img1color,pts1,img2color,pts2,pt_pairs_SSD)
    cv2.imwrite(savepath+imgname+'_'+str(iscale)+'_SSD.jpg',SSD_img)

print('SIFT')
##sift key points
sift=cv2.xfeatures2d.SIFT_create()
sift_pts1,sift_des1=sift.detectAndCompute(img1,None);print('Sift pts1 found')
sift_pts2,sift_des2=sift.detectAndCompute(img2,None);print('Sift pts2 found')
##convert to my format for using NCC and SSD
sift_mypts1=MyCV.convert_SIFT_myFormat(sift_pts1)
sift_mypts2=MyCV.convert_SIFT_myFormat(sift_pts2)

print('SIFT Euclidean')
pt_pairs_sift_eu=MyCV.Euclidean_sift_match(sift_mypts1,sift_des1,sift_mypts2,sift_des2)
new_img=MyCV.plot_matching_points(img1color,sift_mypts1,img2color,sift_mypts2,
    pt_pairs_sift_eu)
cv2.imwrite(savepath+imgname+'_SIFT_EU.jpg',new_img)
```