Computer Assignment 5

Feature Detection, Image Stitching

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy.ndimage import convolve as convolveim
import scipy.ndimage.filters as filters
import scipy.ndimage as ndimage
```

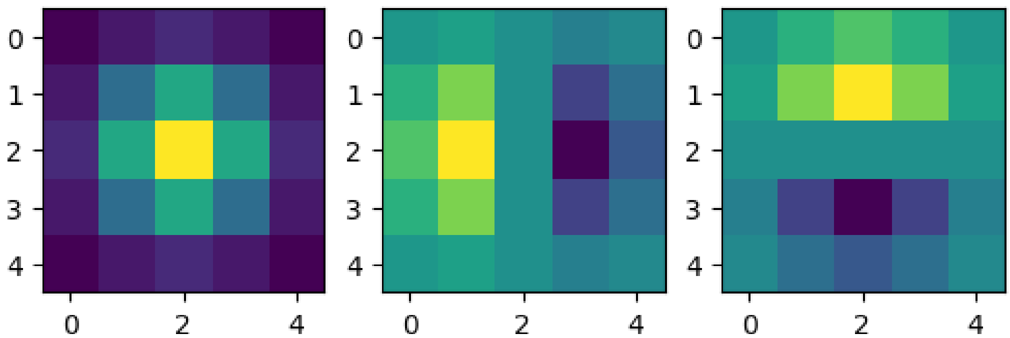
PART A - Harris detector

- Write your own program for Harris corner point detection at a fixed scale. Your program should contain the following steps:
- (a) Generate gradient images of Ix and Iy using filters corresponding to derivative of Gaussian functions of a chosen **scale σ and window size w** (let us use w=4σ+1). You can use the convolve function from scipy.
- (b) Compute three images **Ix^2 , Iy^2 , Ix* Iy**.
- (c) To determine the Harris value at each pixel, we should apply Gaussian weighting over a window size of WxW centered at this pixel to each of the image Ix^2, Iy^2, and Ix Iy, and then sum the weighted average. This is equivalent to convolve each of these images by a Gaussian filter with size WxW. Let us use a Gaussian filter with scale 2σ, and window size W=6σ+1.
- (d) Generate an image of Harris corner value based on the images from (c). See slide 18 and 19 from *feature detection* lecture notes.
- (e) Detect local maxima in the Harris value image (For each pixel, look at a 7x7 window centered at the pixel. Keep the pixel only if it is greater than all other pixels in this window). **Pick the first N feature points with largest Harris values.**
- (f) Mark each detected point using a small circle. Display the image with the detected points.
- You should write your own functions to generate Guassian and derivative of Gaussian filters from the analytical forms (This is similar to Part2 in CA02).
- Apply your Harris detector to a test image (you can just work on gray scale image). Using σ=1, N=100. Do the features detected make sense?

```
def gauss(size, sigma):
    """
    This function will generate 3 filters given the size of the filter
and sigma of Gaussian:
    1: gaussian filter;
    2: derivative of gaussian filters in x and y direction.
    """
    # define the x range
    # x_ax = np.arange(0,size) - size/2 + 0.5
    if np.mod(size,2)==0:
        x_ax = np.arange(size) - size/2 + 0.5 # for even size
```

```
    else:
        x_ax = np.arange(size) - size//2# for odd size

    ############################################## TODO
##############################################
    # make 1D gaussian filter
    gauss = (1/sigma)*np.exp(-0.5*(x_ax/sigma)**2)
    # Compose 2D Gaussian filter from 1D, using the separability
property of 2D Gaussian
    gauss2 = np.outer(gauss, gauss.T)
    # Normalize the filter so that all coefficients sum to 1
    gauss1 = gauss2/np.sum(gauss2)

    # Create derivatives of gaussian
    gauss1_dx = np.matrix(np.zeros((np.shape(gauss1)),
dtype="float32"))
    gauss1_dy = np.matrix(np.zeros((np.shape(gauss1)),
dtype="float32"))
    for j in range(0, len(x_ax)):
        # derivative filter in x
        gauss1_dx[:, j] = (gauss1[:, j] * (-
x_ax[j])/(sigma*sigma)).reshape(size,1)
        ############################################## TODO
##############################################
        # similarly define the difference in y
        gauss1_dy[j, :] = gauss1[j, :] * -x_ax[j]/(sigma**2)
    return gauss1, gauss1_dx, gauss1_dy

# Visualize the filters you created to make sure you are working with
the correct filters
gauss1, gauss1_dx, gauss1_dy = gauss(5,1)
plt.figure()
plt.subplot(1,3,1)
plt.imshow(gauss1)
plt.subplot(1,3,2)
plt.imshow(gauss1_dx)
plt.subplot(1,3,3)
plt.imshow(gauss1_dy)
```

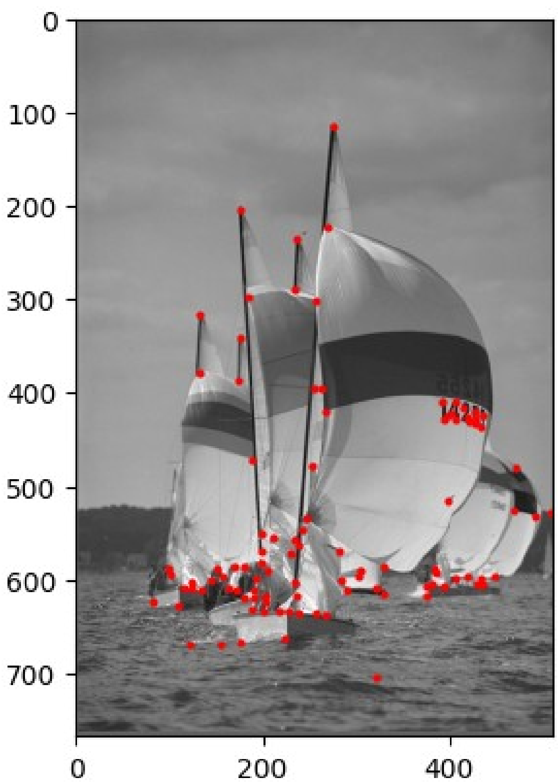
```
<matplotlib.image.AxesImage at 0x7f3bd4a376a0>
```

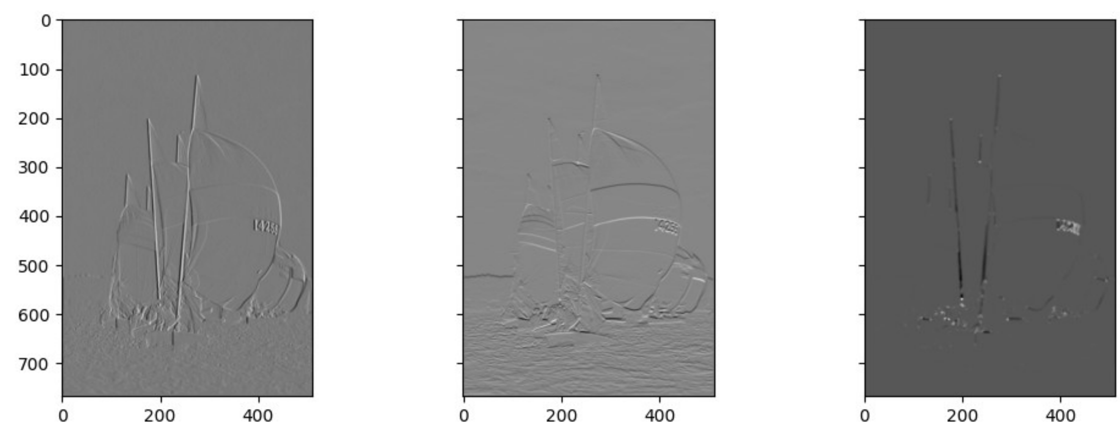


```
def harris(Ix, Iy , input_image,N):
    """
    The input to this function are the gradient images in x and y
directions, the original image and N
    The function will output two arrays/lists x and y which are the N
points with largest harris values,
    and an image of the harris values.
    """
    l, m = np.shape(input_image)
    ################################################ TODO
###############################################
    #Forming 3 images
    #Ix square
    Ix2 = Ix**2
    #Iy square
    Iy2 = Iy**2
    #Ix*Iy
    Ixy = Ix*Iy

    # Smooth image Ix2, Iy2, Ixy  with Gaussian filter with sigma=2,
size=7.
    # Get the gauss filter for smoothing (reuse what you have)
    gauss_smooth, _, _ = gauss(7,2)
    Ix2_smooth = convolveim(Ix2, gauss_smooth, mode='nearest')
    ################################################ TODO
###############################################
    # CONVOLVE as shown above
    Iy2_smooth = convolveim(Iy2, gauss_smooth, mode='nearest')
    Ixy_smooth = convolveim(Ixy, gauss_smooth, mode='nearest')
    # By doing this, Ix2_smooth, Iy2_smooth, Ixy_smooth are the three
values needed to calculate
    # the A matrix for each pixel.

    ################################################ TODO
###############################################
    # Write code segment to find N harris points in the image
    # Refer to the page 17 of slides on features for the equation
    det = Ix2_smooth*Iy2_smooth - Ixy_smooth**2
    trace = Ix2_smooth + Iy2_smooth
```

```
    H = det - 0.06 * (trace**2)

    ################################################## TODO
##################################################
    # Save a copy of the original harris values before detecting local
max
    H0 = H
    # Detect local maximum over 7x7 windows
    local_max_win = 7
    a = int(np.floor(local_max_win/2))
    H = np.pad(H,((a,a),(a,a)), 'constant')
    # Initialize a mask to be all ones. The mask corresponds to the
local maximum in H
    H_max = np.ones(H.shape)
    for i in range(a,l+a):
        for j in range(a,m+a):
            # Take a WxW patch centered at point (i,j), check if the
center point is larger than all other points
            # in this patch. If it is NOT local max, set H_max[i,j] =
0
            patch = H[i - 3: i + 4, j - 3: j + 4]
            # print(patch.shape)
            if np.max(patch) != patch[3,3]:
              H_max[i,j] = 0

    # Multiply the mask with H, points that are not local max will
become zero
    H = H_max*H
    H = H[a:-a,a:-a]

    # Find largest N points' coordinates
    # Hint: use np.argsort() and np.unravel_index() to sort H and get
the index in sorted order
    H = np.argsort(-H,axis=None)
    N_idx = H[0:N]
    x, y = np.unravel_index(N_idx, (l,m))

    # x,y should be arrays/lists of x and y coordinates of the harris
points.
    return x,y,H0

##### IMPORTANT: Convert your image to float once you load the image.
######
input_image = cv2.imread('9.png',0).astype('float')

img = cv2.normalize(input_image, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

################################################## TODO
##################################################
```

```
# Generating the gaussian filter
sigma = 1
size = int(4*sigma + 1)
# Function call to gauss
gauss_filt, gauss_filt_dx, gauss_filt_dy = gauss(size, sigma)


################################################ TODO
###############################################
# Convolving the filter with the image
# Convolve image with dx filter
Ix = convolveim(img,gauss_filt_dx,mode ='nearest')
# Convolve image with dy filter
Iy = convolveim(img, gauss_filt_dy,mode ='nearest')

x,y,H0 = harris(Ix, Iy ,img,100)
################################################ TODO
###############################################
# Plot: Ix, Iy, Original image with harris point labeled in red dots,
H0 harris value image
# Hint: you may use "plt.plot(y,x, 'ro')"   # Note: x is vertical and
y is horizontal in our above definition
                                            # But when plotting the
point, the definition is reversed

plt.plot(y,x, 'ro', markersize=2)
plt.imshow(img, cmap='gray')
plt.show()

fig, ax = plt.subplots(1, 3, figsize=(12, 4), sharey=True)
ax[0].imshow(Ix, cmap='gray')
ax[1].imshow(Iy, cmap='gray')
ax[2].imshow(H0.astype(np.float32), cmap='gray')

plt.show()
```





We can see that features marked with red points denotes major points like corner points of ship, boundary of letter written

PART B - SIFT descriptor

Write a program that can generate SIFT descriptor for each detected feature point using your program in Prob. 1. You may follow the following steps:

- Generate gradient images Ix and Iy as before. Furthermore, determine the gradient magnitude and orientation from Ix and Iy at every pixel.

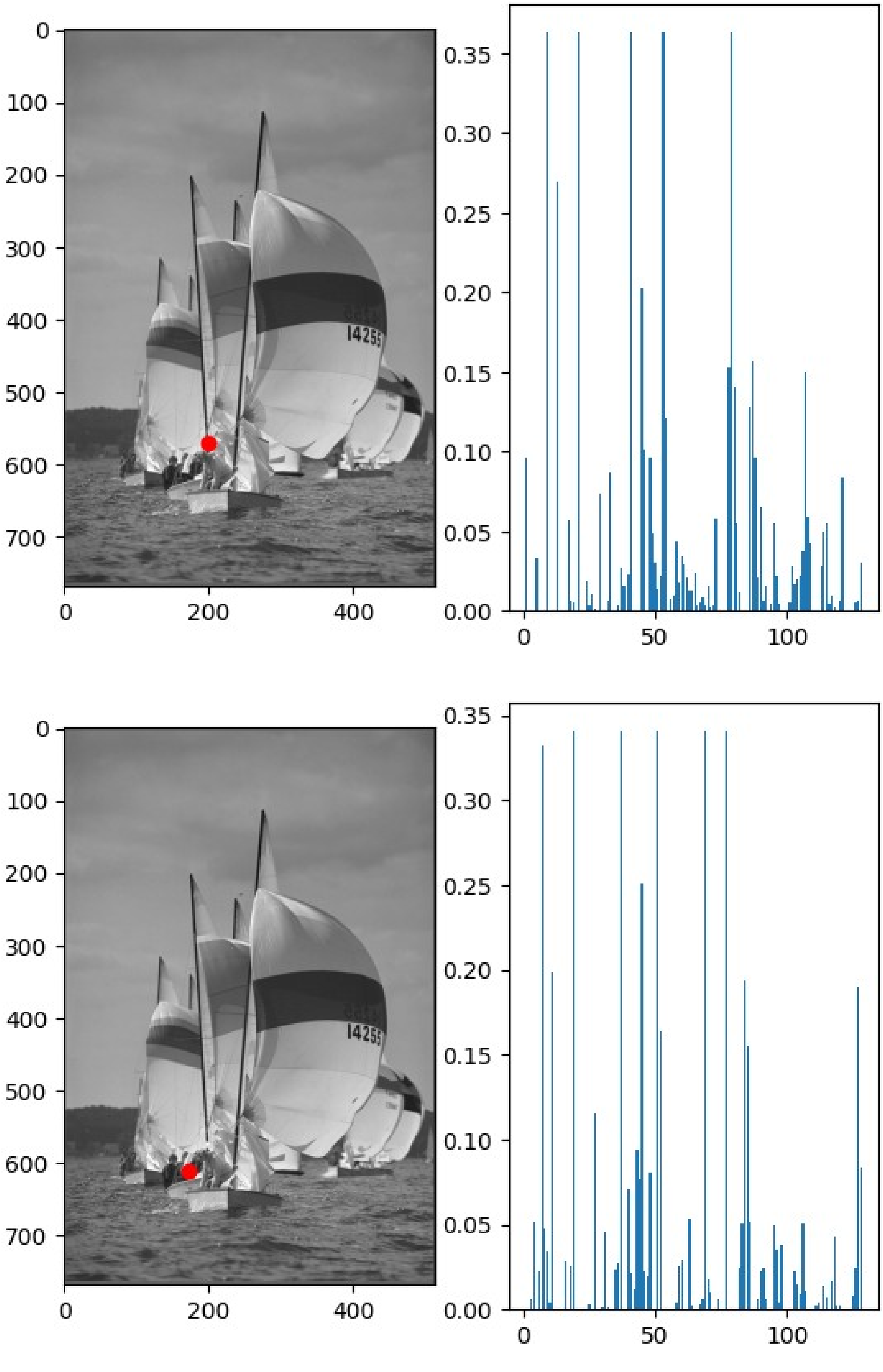
- Quantize the orientation of each pixel to one of the N=8 bins. Suppose your original orientation is x. To quantize the entire range of 360 degree to 8 bins, the bin size is q=360/N=45 degree. Assuming your orientation is determined with a range of [0,360]. x_q will range from 0 to 7, with 0 corresponding to degree (0, 22.5) and (360-22.5, 360). You can perform quantization using: x_q=floor((x+q/2)/q); but if x_q=N, change to -> x_q=0
- Then for each detected feature point, follow these steps to generate the SIFT descriptor:
  - i) Generate a patch of size 16x16 centered at the detected feature point;
  - ii) Multiply the gradient magnitude with a Guassian window with scale= patch width/2.
  - iii) Generate a HoG for the entire patch using the weighted gradient magnitude.
  - iv) Determine the dominant orientation of the patch by detecting the peak in the Hog determined in (iii).
  - v) Generate a HoG for each of the 4x4 cell in the 16x16 patch.
  - vi) Shift each HoG so that the dominant orientation becomes the first bin.
  - vii) Concatenate the HoG for all 16 cells into a single vector.
  - viii) Normalize the vector. That is, divide each entry by L2 norm of the vector.
  - ix) Clip the normalized vector so that entries >0.2 is set to 0.2.
  - x) Renormalize the vector resulting from (ix).
- **For this assignment, you are not asked to do multiscale processing. You only need to generate the SIFT descriptors for those feature points detected by the Harris detector at the original image scale (from Part A).**

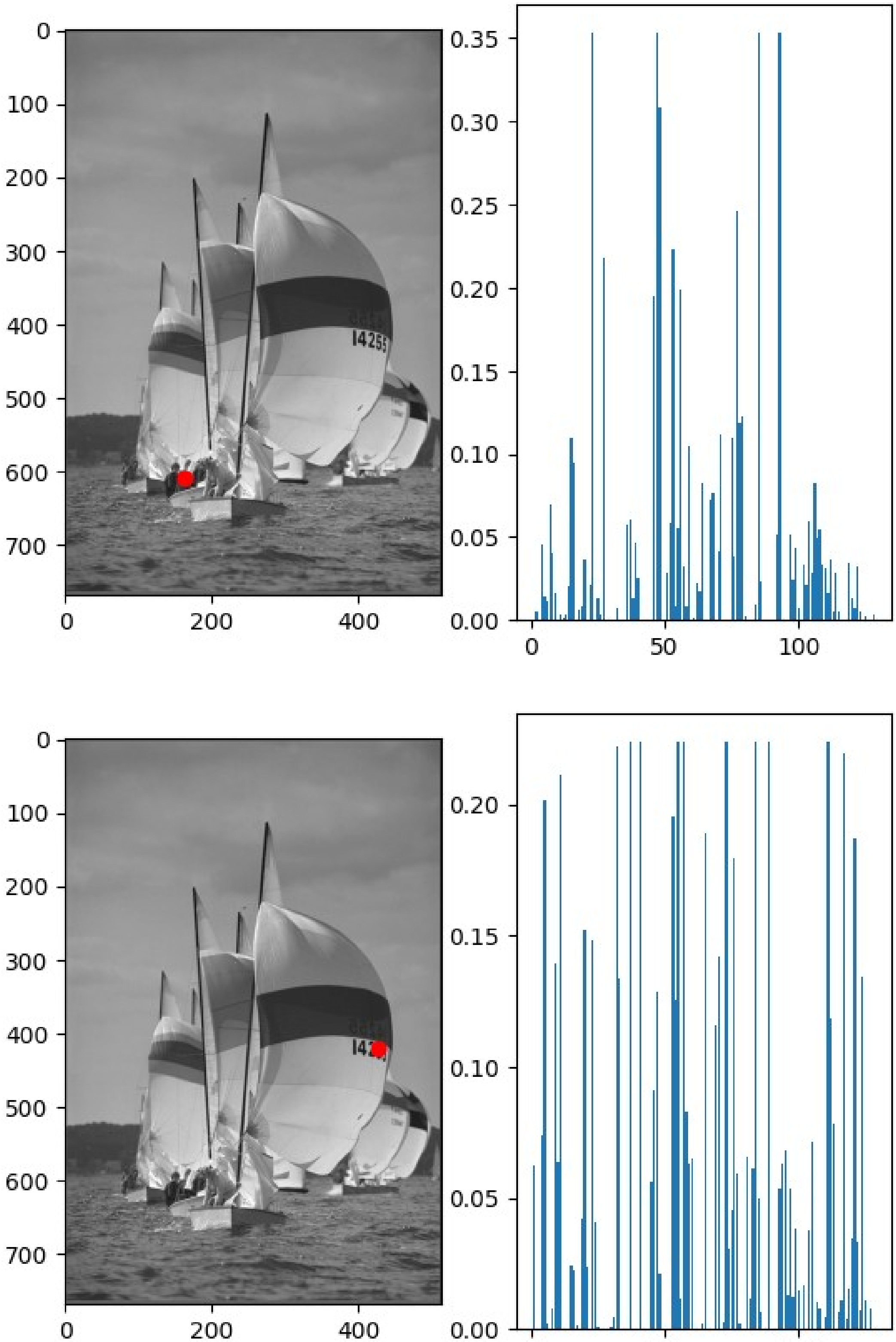
```
def histo(theta4,mag4):
    """
    theta4: an array of quantized orientations, with values 0,1,2...7.
    mag4: an array of the same size with magnitudes
    """
    temp = np.zeros((1,8),dtype='float32')
    ################################################ TODO
###############################################
    # write code segment to add the magnitudes of all vectors in same
orientations
    for i in range(8):
        temp[0][i] = np.sum(mag4[np.where(theta4==i)])

    # temp should be a 1x8 vector, where each value corresponds to an
orientation and
    # contains the sum of all gradient magnitude, oriented in that
orientation
    return temp

def descriptor(theta16,mag16):
    """
    Given a 16x16 patch of theta and magnitude, generate a (1x128)
```

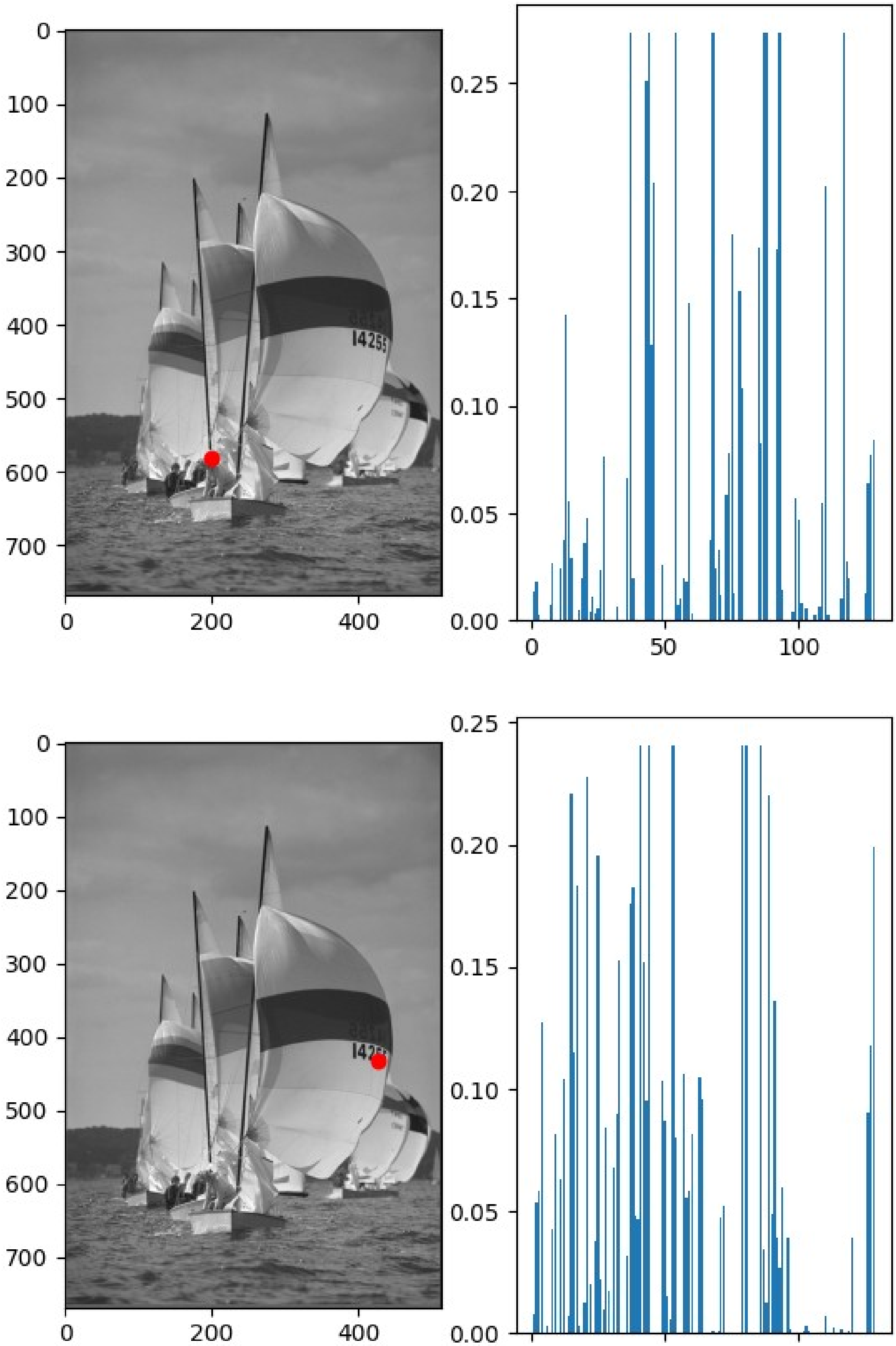
```
descriptor
    """
    filt,_,_ = gauss(16,8)
    mag16_filt = mag16*filt

    # array to store the descriptor. Note that in the end descriptor
should have size (1, 128)
    desp = np.array([])
    ################################################ TODO
###############################################
    # Make function call to histo, with arguments theta16 and
mag16_fil
    # This is used for find the location of maximum theta
    histo16 = histo(theta16, mag16_filt)
    maxloc_theta16 = np.argmax(histo16)

    for i in range(0,16,4):
        for j in range(0,16,4):
            ################################################ TODO
###############################################
            # Use histo function to create histogram of oriantations
on 4x4 pathces in the neighbourhood of the harris points
            # You should shift your histogram for each cell so that
the dominant orientation of the 16x16 patch becomes the first
quantized orientation
            # You should update the variable desp to store all the
orientation magnitude sum for each sub region of size 4x4
            theta4 = theta16[i:i+4,j:j+4]
            mag4 = mag16_filt[i:i+4,j:j+4]

            histo4 = histo(theta4, mag4)
            histo4 = np.roll(histo4, maxloc_theta16)
            desp = np.append(desp, histo4)

    ################################################ TODO
###############################################
    # normalize descriptor, clip descriptor, normalize descriptor
again
    desp = desp / np.linalg.norm(desp)
    desp = np.clip(desp, a_min=None, a_max=0.2)
    desp = desp / np.linalg.norm(desp)

    desp = np.matrix(desp)

    return desp

def part_B(input_image):

    # Normalize the image
    img = cv2.normalize(input_image, None, alpha=0, beta=255,
```

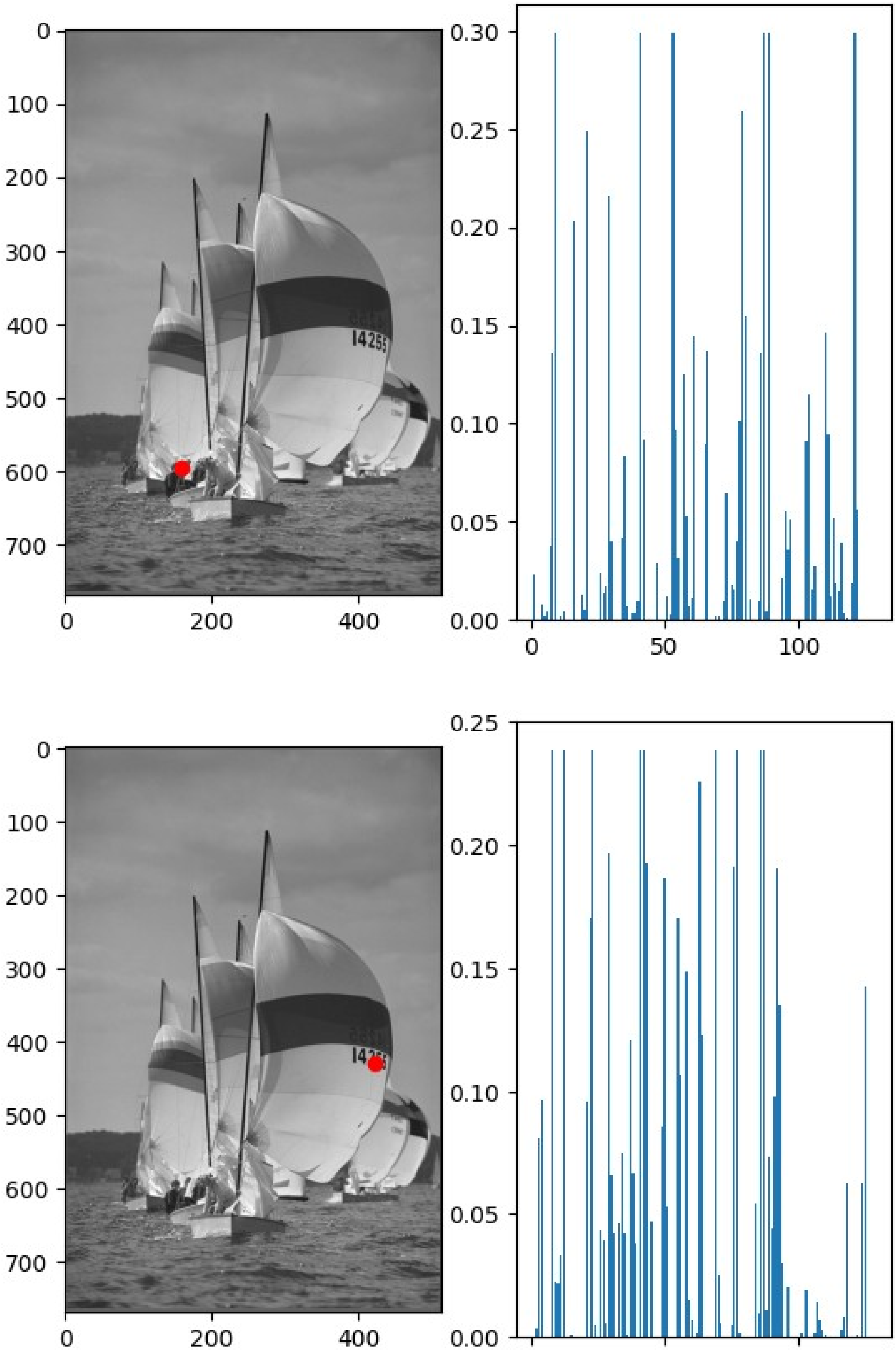
```
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

    # Generate derivative of Gaussian filters, using sigma=1, filter
window size=4*sigma+1
    sigma = 1
    _, filt_dx, filt_dy = gauss(5, 1)

    ############################################### TODO
###############################################
    # Image convolved with filt_dx and filt_dy
    img_x =  convolveim(img, filt_dx, mode='nearest')
    img_y =  convolveim(img, filt_dy, mode='nearest')

    # Calculate magnitude and theta, then quantize theta.
    mag = np.sqrt(img_x ** 2 + img_y ** 2)
    theta = np.arctan2(img_y, img_x)
    theta = (theta/(2*np.pi))*360
    theta = theta*(theta>=0) + (360+theta)*(theta < 0)

    ############################################### TODO
###############################################
    # Quantize theta to 0,1,2,... 7, see instructions above
    q = 45
    N = 8

    theta_q = np.floor((theta + q/2) / q)
    theta_q[theta_q == N] = 0

    ############################################### TODO
###############################################
    # Call harris function to find 100 feature points
    x,y,_ = harris(img_x, img_y , img,100)

    # Pad 15 rows and columns. You will need this extra border to get
a patch centered at the feature point
    #    when the feature points lie on the original border of the
image.
    theta_q = cv2.copyMakeBorder(theta_q.astype('uint8'), 7,8,7,8,
cv2.BORDER_REFLECT)
    ############################################### TODO
###############################################
    mag =  cv2.copyMakeBorder(mag.astype('uint8'), 7,8,7,8,
cv2.BORDER_REFLECT) # similarly add border to the magnitude image
    final_descriptor = np.zeros((1,128))
    final_points = np.vstack((x,y))

    for i in range(final_points.T.shape[0]):
        # Since you have already added 15 rows and columns, now the
new coordinates of the feature points are (x+8, y+8).
        # Then the patch should be [x[i]:x[i]+16,y[i]:y[i]16]
```

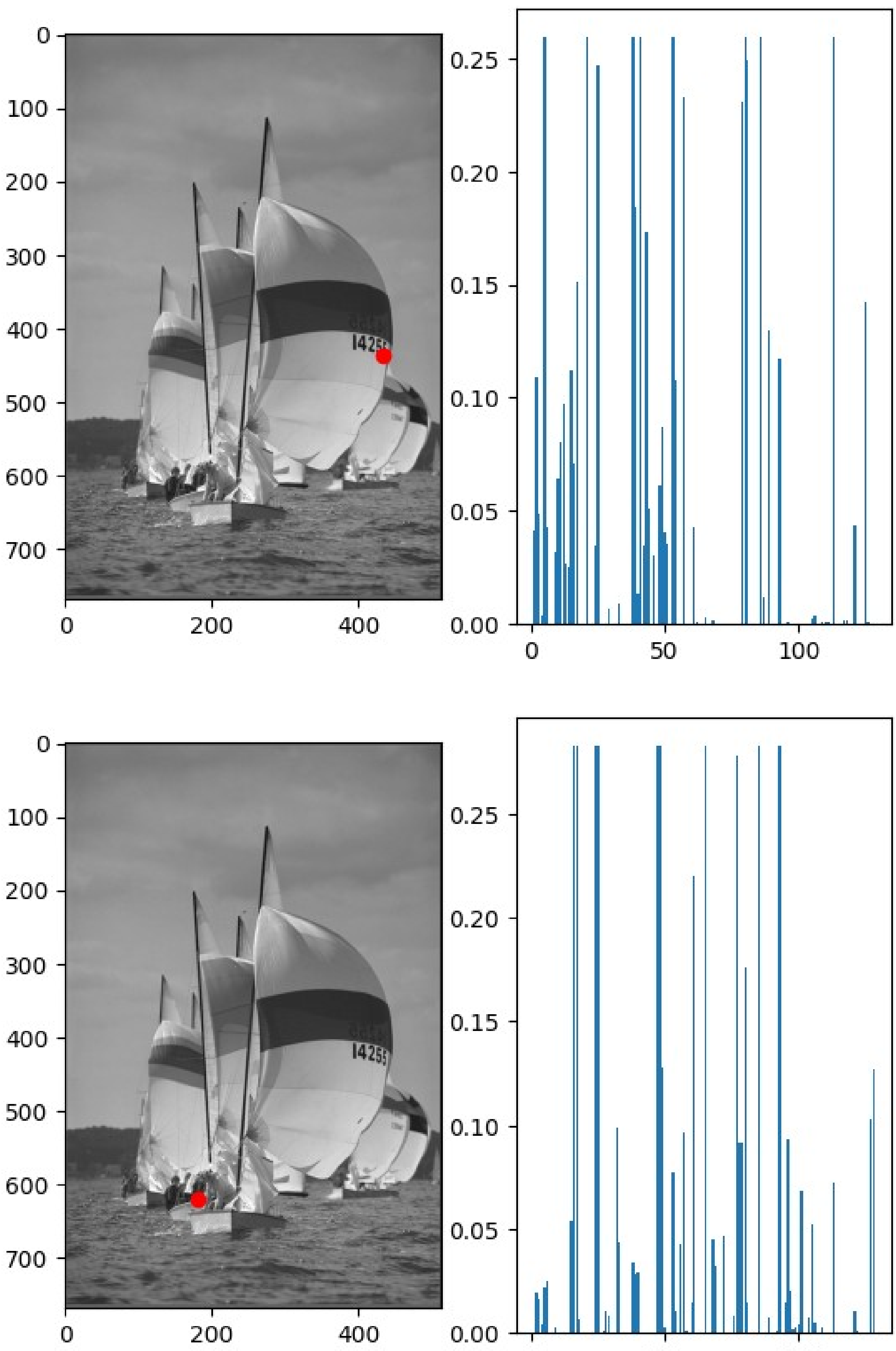
```
        # Your patch should be centered at the feature point.
        theta_temp = theta_q[x[i]:x[i]+16,y[i]:y[i]+16]
        # similarly, take a 16x16 patch of mag around the point
        mag_temp =  mag[x[i]:x[i]+16,y[i]:y[i]+16]
        # function call to descriptors
        temp2 =  descriptor(theta_temp, mag_temp)
        final_descriptor = np.vstack((final_descriptor,temp2))

    # Initially, final descriptor has a row of zeros. We are deleting
that extra row here.
    final_descriptor = np.delete(final_descriptor,0,0)
    final_descriptor = np.nan_to_num(final_descriptor)
    final_descriptor = np.array(final_descriptor)

    # Combine x,y to form an array of size (Npoints,2) each row
correspond to (x,y)
    # You could use np.hstack() or np.vstack()
    # final_points = np.vstack(x,y)

    return final_descriptor,final_points.T

input_image = cv2.imread('9.png',0).astype('float') # input image


# Visualization the results. Plot the feature point similiar to Part1
and plot SIFT features as bar
final_descriptor , final_points = part_B(input_image)

for i in range(0,20):
    f, (ax1, ax2) = plt.subplots(1, 2)
    ax1.imshow(input_image,cmap='gray')
    ax1.autoscale(False)
    ax1.plot(final_points[i][1],final_points[i][0], 'ro')
    ax2.bar(np.arange(1,129),final_descriptor[i,:])
    plt.show()
```

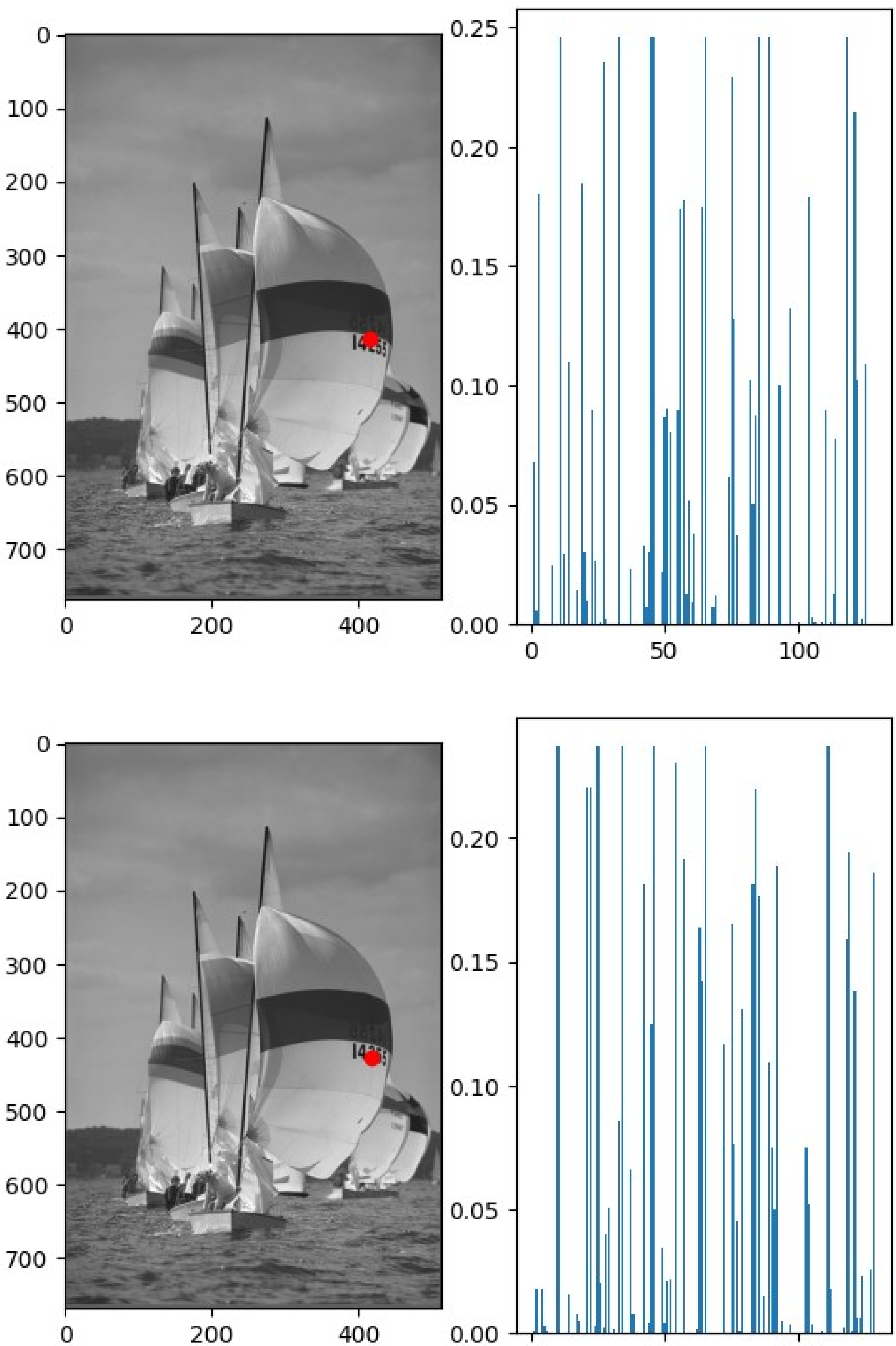

0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
50
100


0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
50
100
0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0
50
100

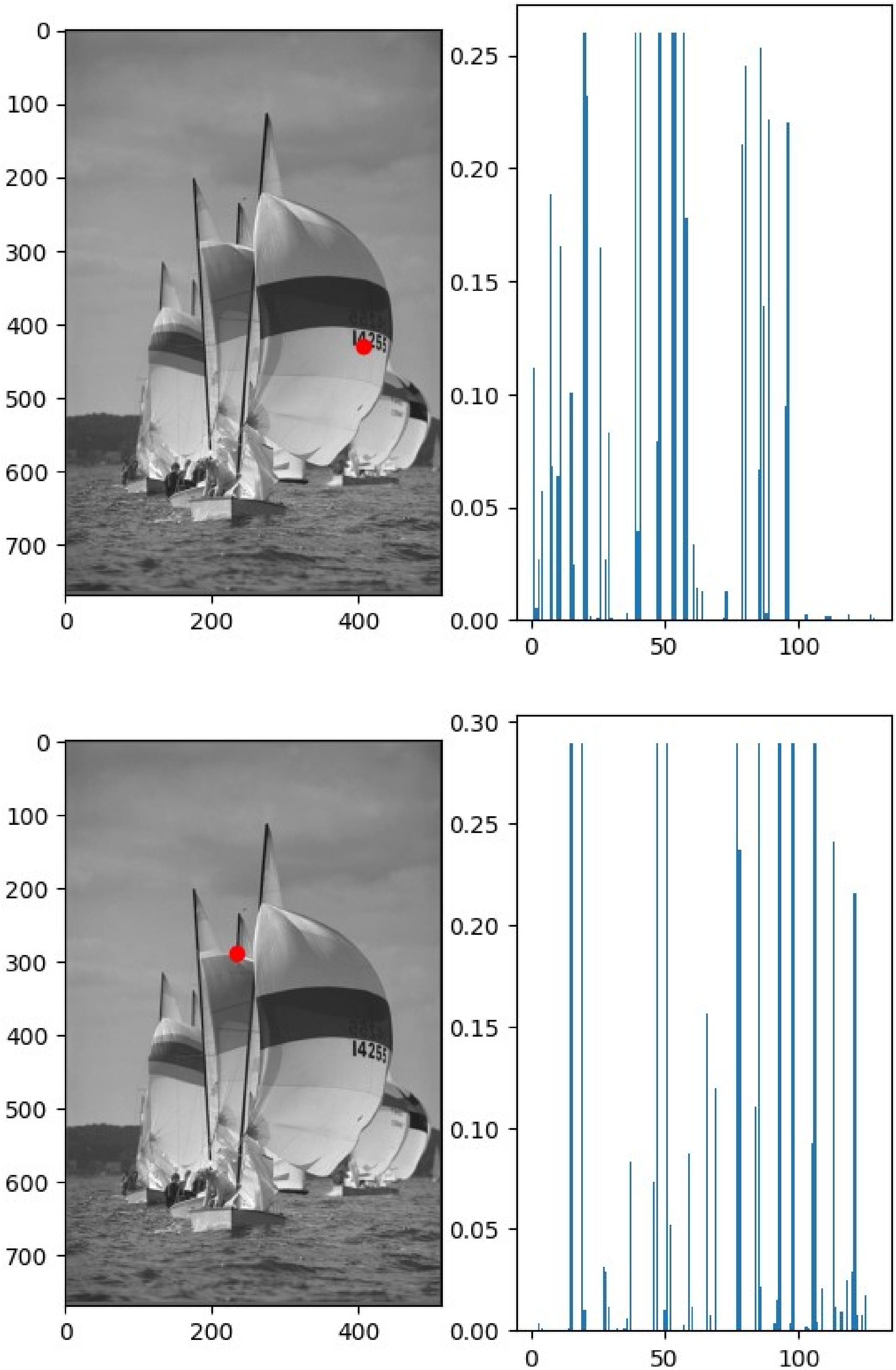

0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0
50
100
0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0
50
100


0
100
200
300
400
500
600
700
0
200
400
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
0
100
200
300
400
500
600
700
0
200
400
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100

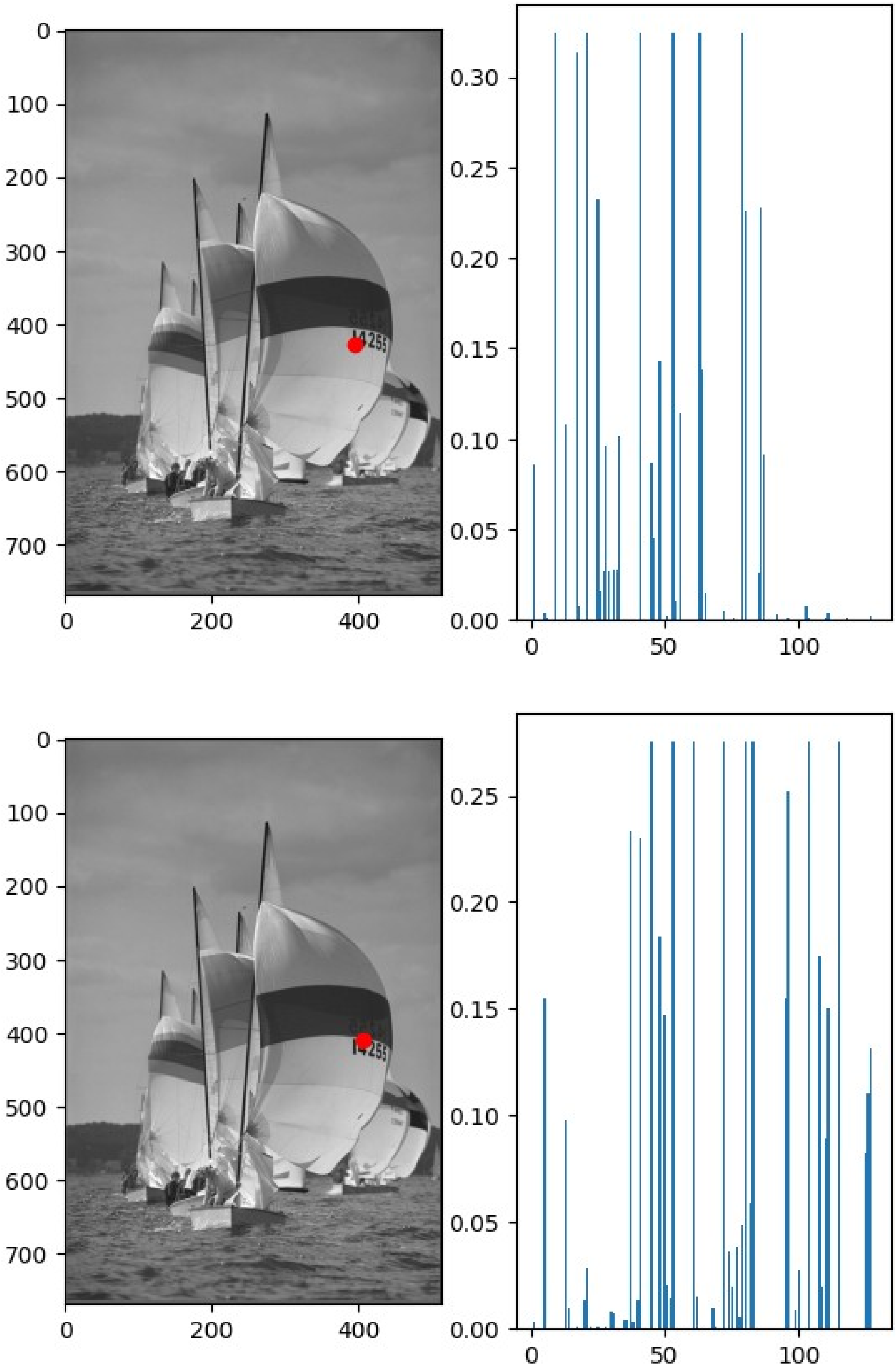

0
100
200
300
400
500
600
700
0
200
400
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
14255



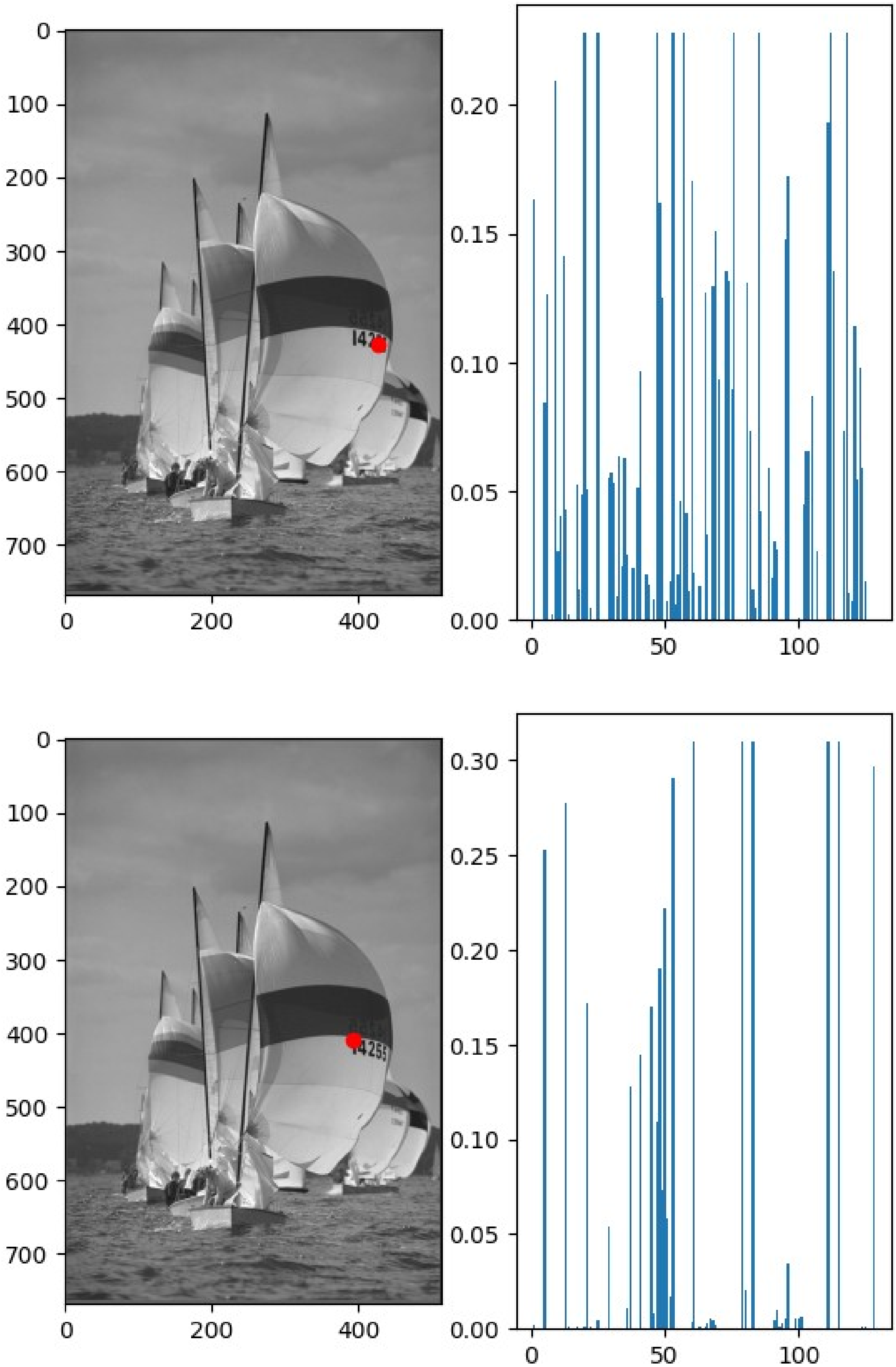
0
100
200
300
400
500
600
700
0 200 400
0.25
0.20
0.15
0.10
0.05
0.00
0 50 100
0
100
200
300
400
500
600
700
0 200 400
0.20
0.15
0.10
0.05
0.00
0 50 100

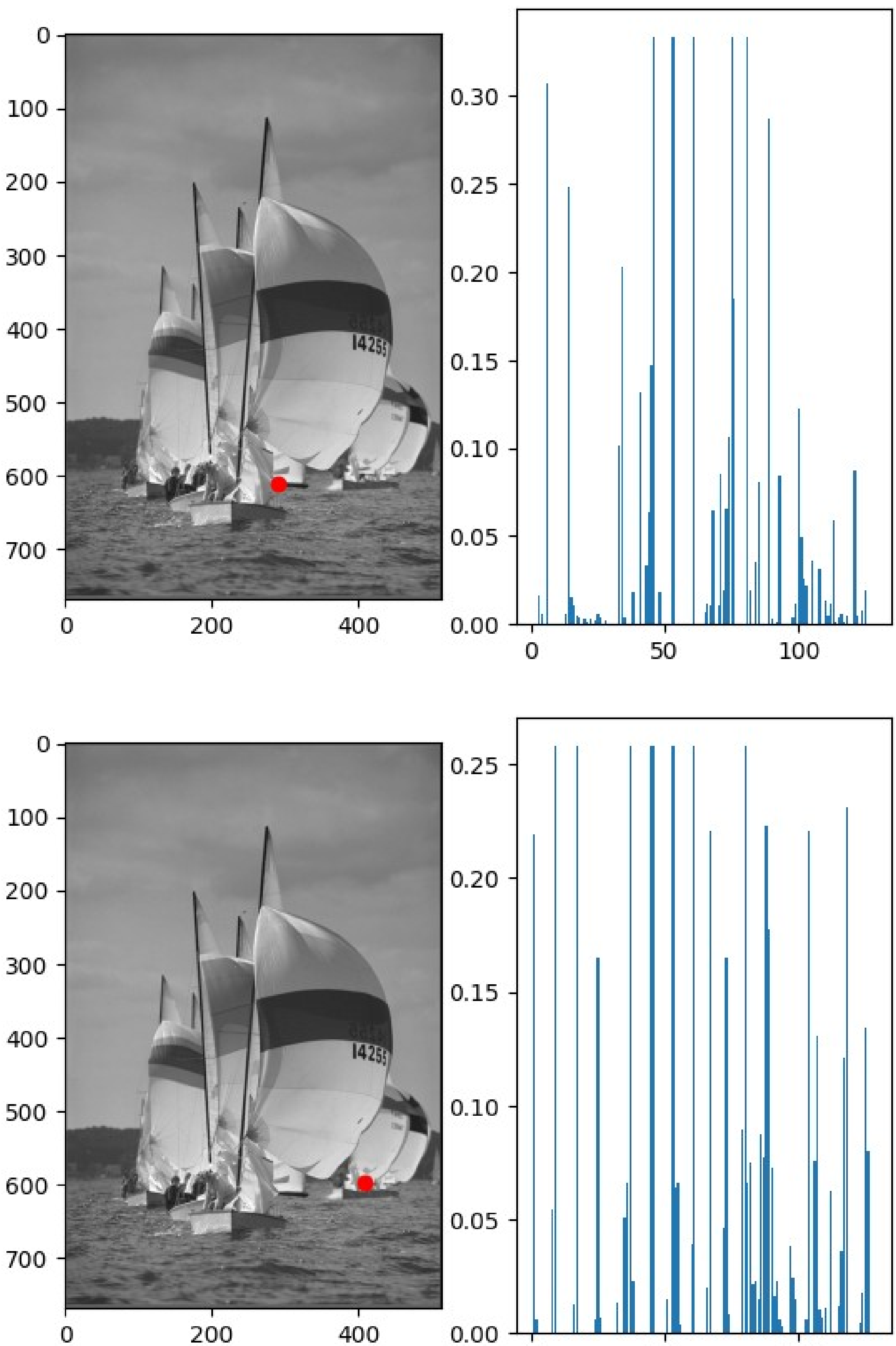


0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
50
100



0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
50
100
0
100
200
300
400
500
600
700
0
200
400
0.00
0.05
0.10
0.15
0.20
0.25
0
50
100


0
100
200
300
400
500
600
700
0
200
400
0.20
0.15
0.10
0.05
0.00
0
50
100
0
100
200
300
400
500
600
700
0
200
400
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100



0 100 200 300 400 500 600 700
0 200 400
0.00 0.05 0.10 0.15 0.20 0.25 0.30
0 50 100
0 100 200 300 400 500 600 700
0 200 400
0.00 0.05 0.10 0.15 0.20 0.25
0 50 100

PART C - correspondance in 2 images

Finding corresponding points in two images based on SIFT descriptors.

- Using your program in Part A and Part B to detect feature points and generate their descriptors for two images provided with this assignment.(image named **left** and **right**)
- Write a program that can find matching points between the two images. For each detected point p in the first image, compute its distance to each detected point in the second image (using Euclidean distance between two SIFT descriptors, not spatial distance) to find two closest points q1 and q2 in the second image. Let us call the distance of p to q1 and q2 by d1 and d2, you will take point q1 as the matching point for p if d1/d2 <r. Otherwise, you assume there is ambiguity between q1 and q2 and skip the feature point p in image 1. You can experiment with different threshold r (for example, r=[0.95, 0.8, 0.65, 0.5]). Obviously r should be <1. At the end of this process, you should have a set of matching pairs.
- Create an image that shows the matching results. For example you can create a large image that has the left and right images side by side, and draw lines between matching pairs in these two images. Do the matched points look reasonable? You can use **cv2.line()** to draw line between each matching pair. Display the image after you add lines into the image array using the cv2.line() function.

```
img1 = cv2.imread('left.jpg',0).astype('float') # read left image
img_1 = cv2.normalize(img1, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

descriptor_1, keypoints_1 = part_B(img_1)

img2 = cv2.imread('right.jpg',0).astype('float') # read right image
img_2 = cv2.normalize(img2, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

descriptor_2, keypoints_2 = part_B(img_2)


# Display detected points in the two images
plt.figure(figsize = (10,10))
plt.imshow(img_1,'gray')
plt.plot(keypoints_1[:,1],keypoints_1[:,0],'ro',ms=3)
plt.figure(figsize = (10,10))
plt.imshow(img_2,'gray')
plt.plot(keypoints_2[:,1],keypoints_2[:,0],'ro',ms=3)
```

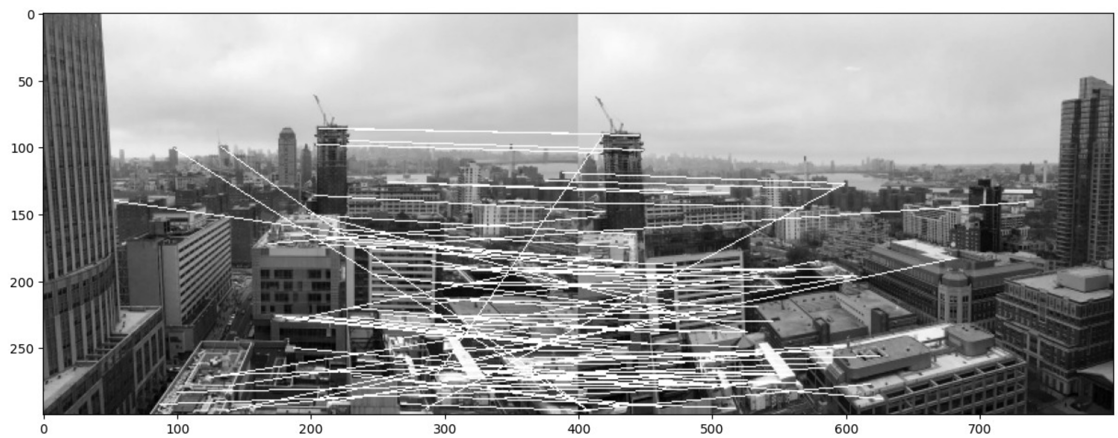
```
[<matplotlib.lines.Line2D at 0x7f3bd8f224d0>]
```


0
50
100
150
200
250
300
350

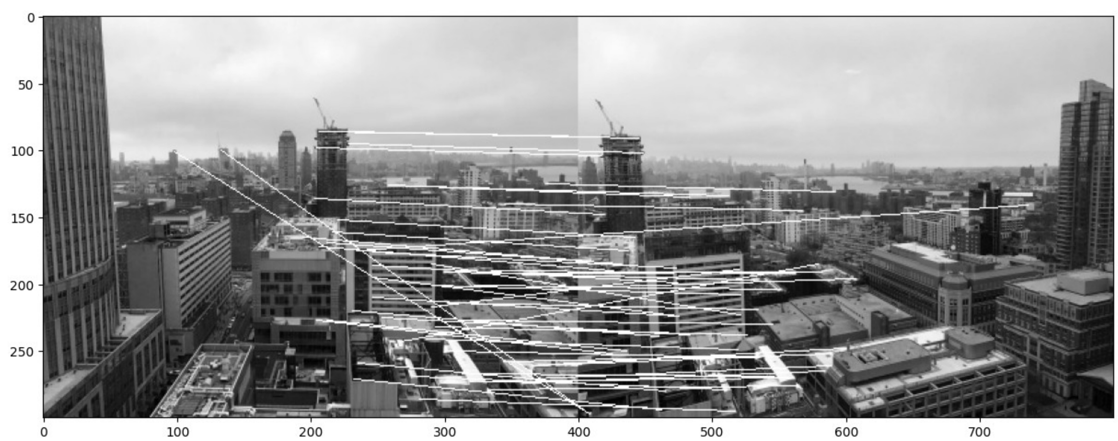

0
50
100
150
200
250
300
350

```
# write function to find corresponding points in image
def points_matching(kp1, descriptor1, kp2, descriptor2, threshold):
    matched_loca = list() # list of all corresponding points pairs.
Point pairs can be stored as tuples
    ############################################### TODO
###############################################
    # Find matching points between img1 and img2 using the algorithm
described in the above
    # For distance measuring, you may use np.linalg.norm()
    # You could implement it as nested loop for simplicity.

    for index, value in enumerate(kp1):
      dist = np.sum((descriptor1[index][None,:] - descriptor2)**2,
axis=1)
      dist_index = np.argsort(dist)
      dist1, dist2 = dist[dist_index[0:2]]
      if (dist1/dist2) < threshold:
          matched_loca.append((index, dist_index[0]))
    return matched_loca

# Test different thresholds for the matching
l,m = np.shape(img_1)

for r in [0.95, 0.8, 0.65, 0.5]:
    matched_loca = points_matching(keypoints_1, descriptor_1,
keypoints_2, descriptor_2, r)
    final_image = np.concatenate((img_1,img_2),axis=1)
    print('threshold: ', r)
    print('number of corresponding poitnts found:',len(matched_loca))
    ############################################### TODO
###############################################
    # Write code segment to draw lines joining corresponding points
    # Use cv2.line() to draw the line on final_image
    # Remember the x,y coordinate in numpy and OpenCV is opposite and
you need to add image width for pt2

    for pt in matched_loca:
      # print(keypoints_1[pt[0]][0],keypoints_1[pt[0]][1])
      final_image = cv2.line(final_image, (keypoints_1[pt[0]]
[1],keypoints_1[pt[0]][0]), (keypoints_2[pt[1]]
[1]+m,keypoints_2[pt[1]][0]), (255,255,255), 1)

    plt.figure(figsize=(15,15))
    plt.imshow(final_image, cmap='gray')
    plt.show()
```

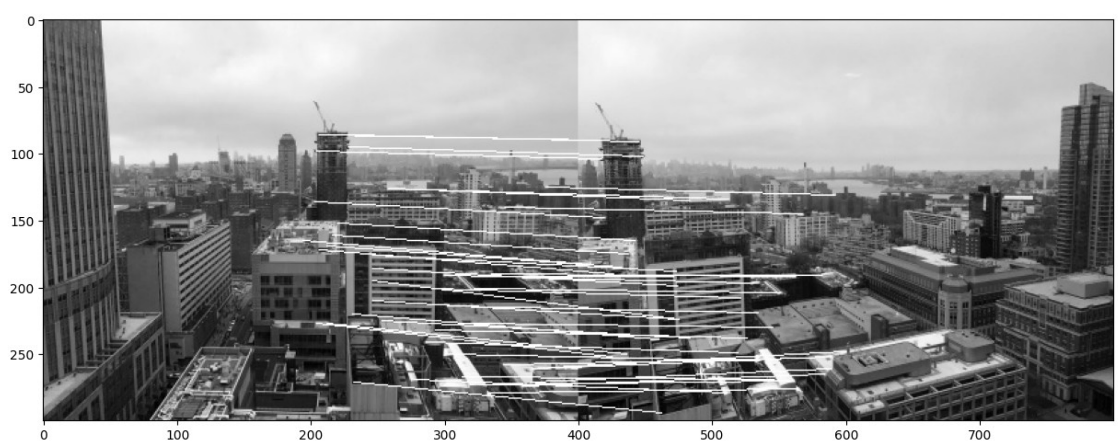
```
threshold:  0.95
number of corresponding poitnts found: 79
```



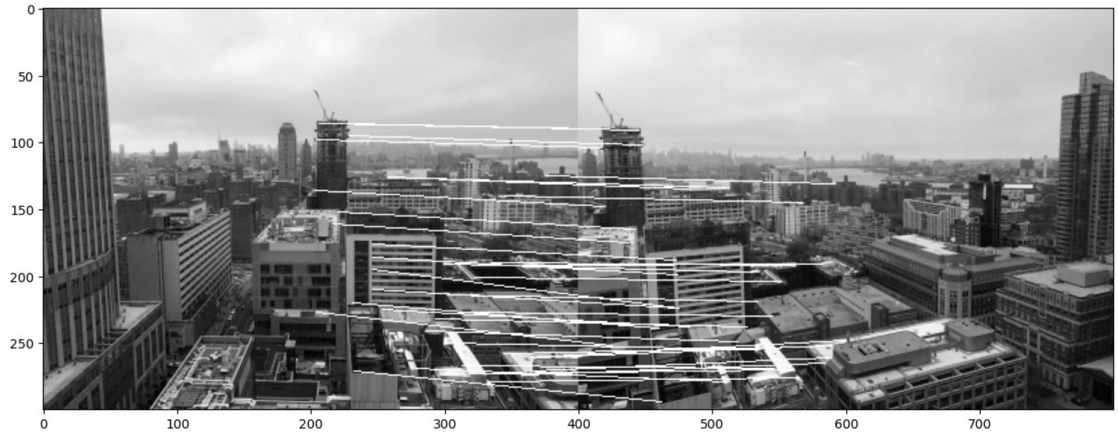
threshold: 0.8
number of corresponding poitnts found: 51



threshold: 0.65
number of corresponding poitnts found: 42



threshold: 0.5
number of corresponding poitnts found: 37



For threshold, 0.5, we can see that matched points are much reasonable.

PART D - panorama stiching

Stitch two images into a panorama using SIFT feature detector and descriptor. In this part, you may use functions from cv2.

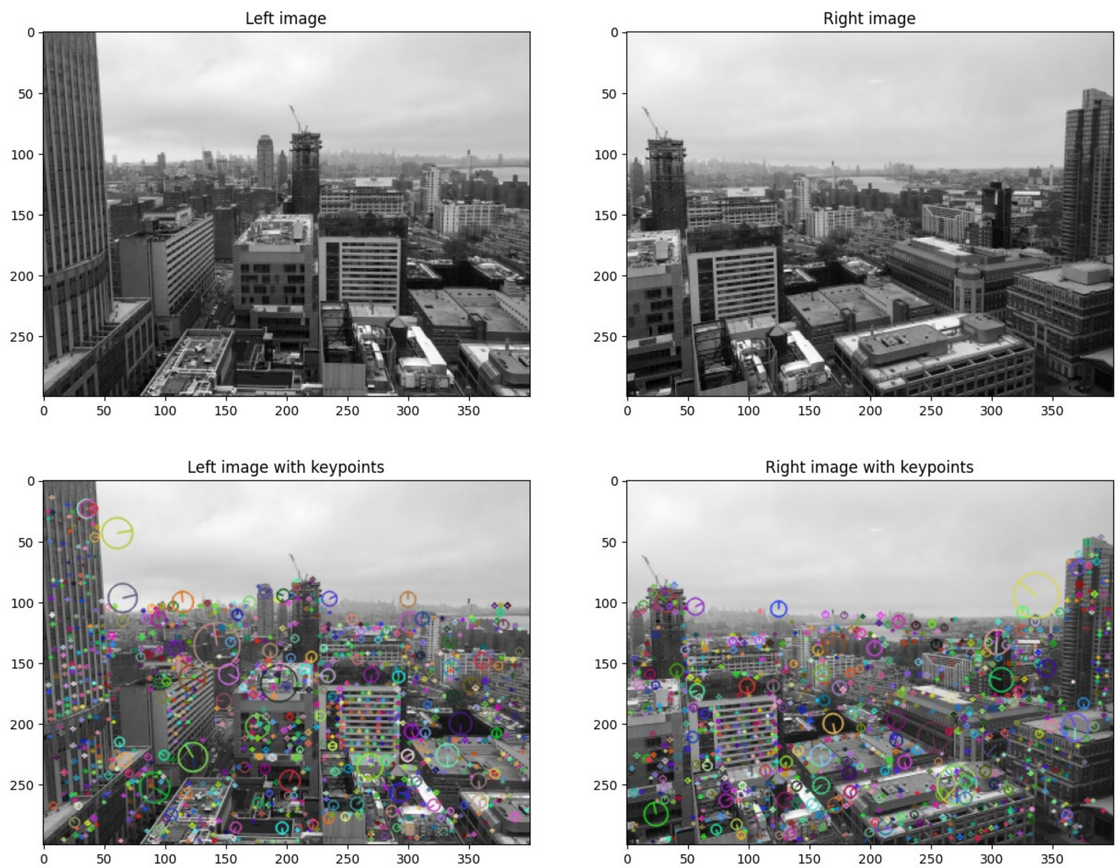
- Read in the following image pair (left and right)
- Detect SIFT points and extract SIFT features from each image by using the following OpenCV sample code.
  - sift = cv2.SIFT_create()
  - skp = sift.detect(img,None)
  - (skp, features) = sift.compute(img, skp)
- **Where skp is a list of all the key points found from img and features is the descriptor for the image. Each element in skp is an OpenCV 'key points class' object, and you can check the corresponding coordinate by skp[element_index].pt**
- Detect and mark feature points, calculate their descriptor using cv2 functions.
- Find the corresponding point pairs between left and right images based on their SIFT descriptors. You can reuse your program for Part-C.
- Apply RANSAC method to these matching pairs to find the largest subset of matching pairs that are related by the same homography. You can use the function cv2.findHomography(srcPoints, dstPoints, cv2.RANSAC)
- Create an image that shows the matching results by drawing lines between corresponding points. You can use the drawMatches function below
- Apply the homography to the right image. You can use cv2.warpPerspective() to apply the homography transformation to the image.

- Stitch the transformed right image and the left image together to generate the panorama.

In your report, show the left and right images, the left and right images with SIFT points indicated, the image that illustrates the matching line between corresponding points, the transformed left image, and finally the stitched image.

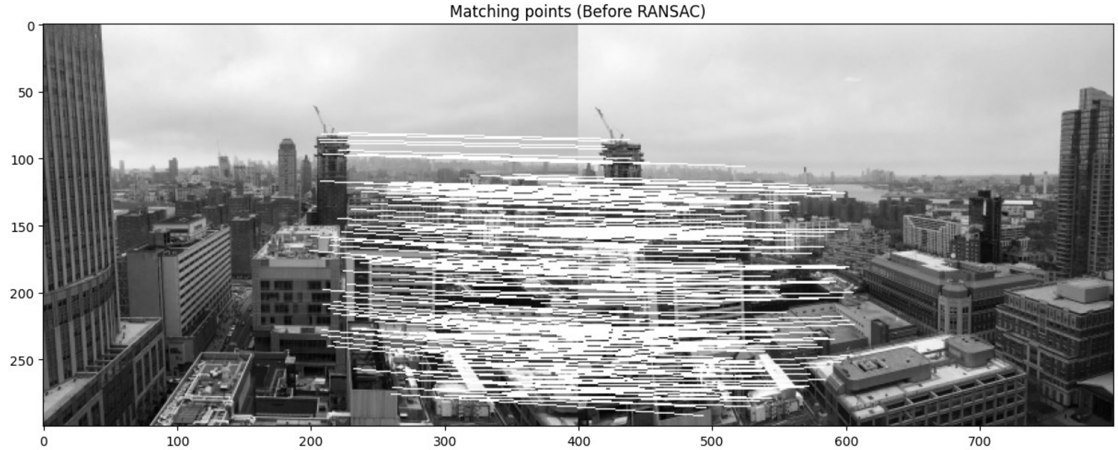
```
def drawMatches(imageA, imageB, kpsA, kpsB, matches, status, lw=1):
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
    vis = np.zeros((max(hA, hB), wA + wB), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for ((trainIdx, queryIdx), s) in zip(matches, status):
        # only process the match if the keypoint was successfully
        # matched
        if s == 1:
            # draw the match
            ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))
            ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx]
[1]))
            cv2.line(vis, ptA, ptB, (255, 255, 255), lw)
    # return the visualization
    return vis

img1 = cv2.imread('left.jpg',0) # read left image
img2 = cv2.imread('right.jpg',0) # read right iamge

# Depending on your OpenCV version, you could set up SIFT differently
# sift = cv2.SIFT_create()
sift = cv2.xfeatures2d.SIFT_create()

################################################ TODO
###############################################
# Use sift.detect to detect features in the images
kp1 = sift.detect(img1, None)
kp2 = sift.detect(img2, None)

# Visualize the keypoints
img1_kps = cv2.drawKeypoints(img1,kp1,None,flags =
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
img2_kps = cv2.drawKeypoints(img2,kp2,None,flags =
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

plt.figure(figsize=(15,15))
plt.subplot(121)
plt.imshow(img1, cmap='gray')
```

```python
plt.title('Left image')
plt.subplot(122)
plt.imshow(img2, cmap='gray')
plt.title('Right image')
plt.show()

plt.figure(figsize=(15,15))
plt.subplot(121)
plt.imshow(img1_kps)
plt.title('Left image with keypoints')
plt.subplot(122)
plt.imshow(img2_kps)
plt.title('Right image with keypoints')
plt.show()
```

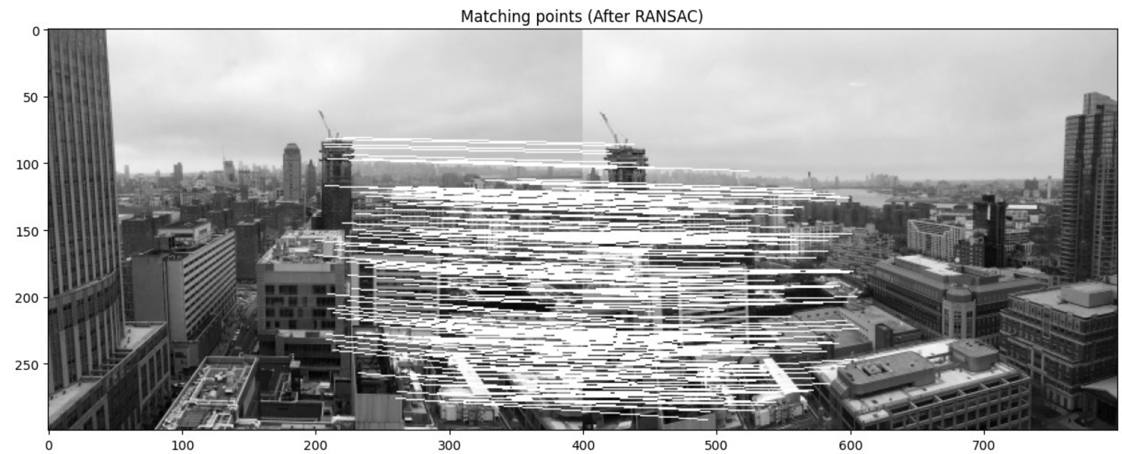



```python
################################################ TODO
###############################################
# Use sift.compute to generate sift descriptors/features
(kp1, features1) = sift.compute(img1,kp1)
(kp2, features2) = sift.compute(img2, kp2)


kp1 = np.float32([kp.pt for kp in kp1])
kp2 = np.float32([kp.pt for kp in kp2])
```

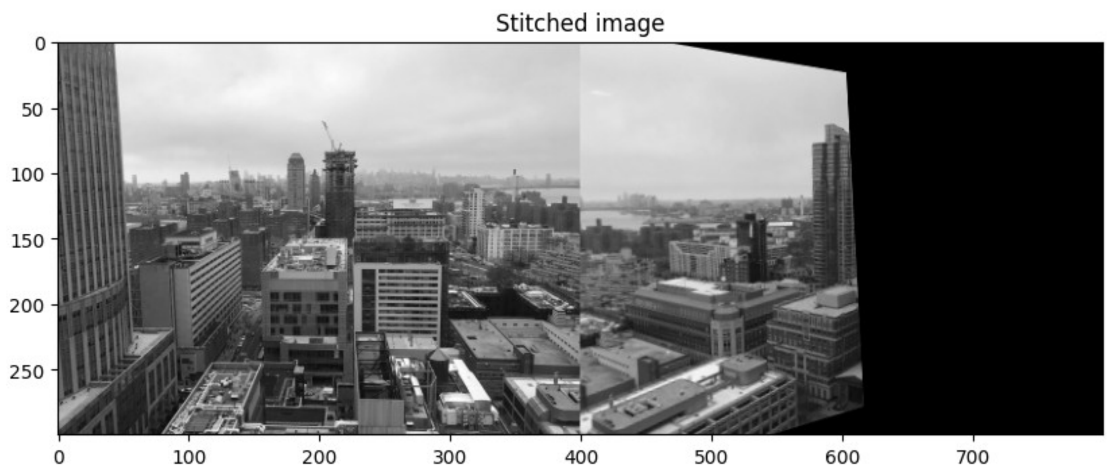
```
matcher = cv2.DescriptorMatcher_create("BruteForce")
################################################ TODO
###############################################
# Use knnMatch function in matcher to find corresonding features
# For robustness of the matching results, we'd like to find 2 best
matches (i.e. k=2 for knnMatch)
# and return their matching distances
rawMatches = matcher.knnMatch(features1,features2, k=2)
matches = []

# Now we validate if the matching is reliable by checking if the best
maching distance is less than
# the second matching by a threshold, for example, 20% of the 2nd best
maching distance
for m in rawMatches:
    ################################################ TODO
###############################################
    # Ensure the distance is within a certain ratio of each other
(i.e. Lowe's ratio test)
    # Test the distance between points. use m[0].distance and
m[1].distance
    if len(m) == 2 and m[0].distance/m[1].distance < 0.5:
        matches.append((m[0].trainIdx, m[0].queryIdx))

ptsA = np.float32([kp1[i] for (_,i) in matches])
ptsB = np.float32([kp2[i] for (i,_) in matches])

################################################ TODO
###############################################
### Similar to what we did in part C
### Create an image img_match that shows the matching results by
drawing lines between corresponding points.
img_match = np.concatenate((img1,img2),axis=1)
l,m = np.shape(img1)

for p1, p2 in zip(ptsA, ptsB):
  # print(p1[1], p2[1])
  cv2.line(img_match, (int(p1[0]), int(p1[1])), (int(p2[0]+m),
int(p2[1])), (255,255,255), 1)
plt.figure(figsize=(15,15))
plt.imshow(img_match, cmap='gray')
plt.title('Matching points (Before RANSAC)')
plt.show()
```




```
############################################### TODO
###############################################
# Find homography with RANSAC
(H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC)

img_ransac = drawMatches(img1,img2,kp1,kp2,matches,status)
plt.figure(figsize=(15,15))
plt.imshow(img_ransac,'gray')
plt.title('Matching points (After RANSAC)')
plt.show()

############################################### TODO
###############################################
# fill in the arguments to warp the second image to fit the first
image.
# For the size of the resulting image, you can set the height to be
the same as the original, width to be twice the original.
# First transform the right image, then fill in the left part with the
orignal left image
result = cv2.warpPerspective(img2, H, (img2.shape[1]*2,
img2.shape[0]))

translation_matrix = np.float32([ [1,0,img1.shape[0]+100], [0,1,0] ])
result = cv2.warpAffine(result, translation_matrix, (img2.shape[1]*2,
img2.shape[0]))
# For blending, you could just overlay img1 to the corresponding
positions on warped img2
result[0:img1.shape[0], 0:img1.shape[1]] = img1

plt.figure(figsize=(10,10))
plt.imshow(result,'gray')
plt.title('Stitched image')
```



Text(0.5, 1.0, 'Stitched image')



Please answer the following questions based on your observation:

- For these 2 images, the matched features points are not necessary from the same depth (and therefore not on the same plane), why we could still relate them by a homography?
- Why the right image looks a bit blurry?

Answer 1

As we can observe in the image that camera is very far away from the scene. Hence, all the features in image can be considered to be at same depth (depth variation is small within the scene). Hence, we were able to relate them.

Answer 2

Due to perspective transformation, it performs interpolation and hence it looks bit blurry.

Interactive Correspondence Visualization

- Just for interactive visualization, not in assignments

```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

matched_idx = status.nonzero()[0]

def visualize_match(x):
    """
    This function visualize the matches interactively
    You could change the visualization of the matching keypoints by
toggling a bar
    Need to have the matches and status ready
        matches: coarse matching results obtained from knnMatch
        status: the refined matching results provided by
cv2.findHomography,
                the positive match determined by RANSAC is marked with
1,
                while the negative match is marked with 0
    """
    idx = matched_idx[x]
    img_ransac =
drawMatches(img1,img2,kp1,kp2,matches[idx:idx+1],status[idx:idx+1],
lw=2)

    plt.figure(figsize=(25,25))
    plt.imshow(img_ransac,'gray')
    plt.title('Matching points (After RANSAC)')
    plt.show()

interact(visualize_match, x=widgets.IntSlider(min=0,
max=len(matched_idx)-1, step=1, value=100));
```

{"model_id":"01eb95cdedd645788a14d527eef72e32","version_major":2,"version_minor":0}