Computer Vision
CS 6476 , Spring 2018

PS 4

Professor :
Cedric Pradalier

Author :
Melisande Zonta

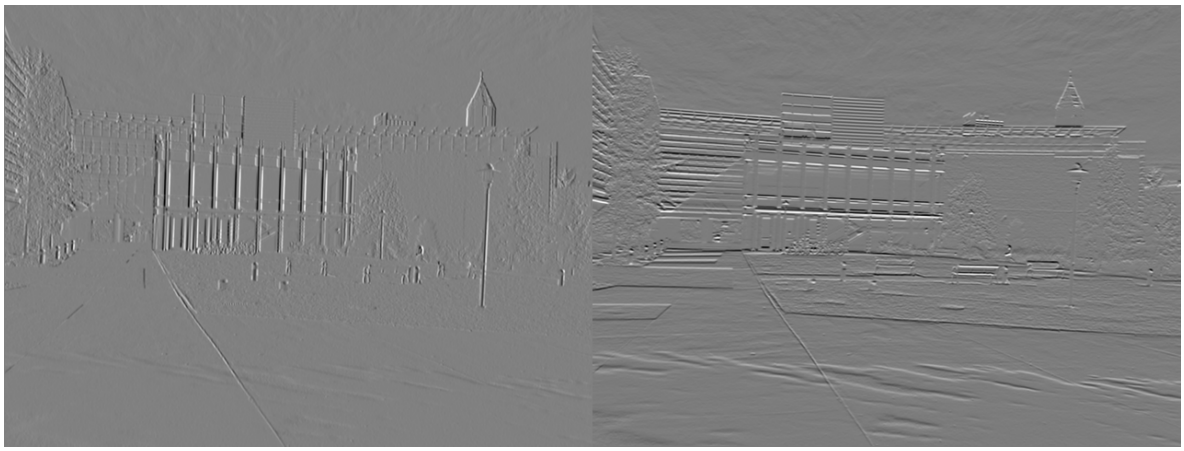
March 23, 2018

Contents

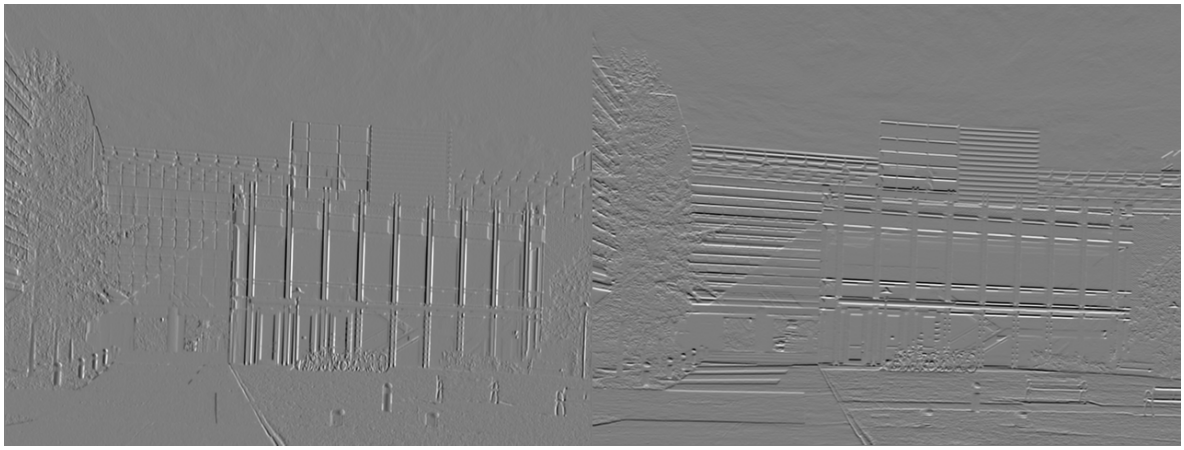
1 Harris Corners

1.1 Intensity Gradients Computation

The X/Y gradients for the two pairs of images (simA/simB and transA/transB) are respectively presented in Figure 1a and Figure 1b.



a



b

Figure 1: *a.* ps4-1-1-simA. *b.* ps4-1-1-transA.

```
1  def compute_gradients(image,aperture_size):
2
3      # Gaussian derivative kernels
4      g = cv2.getGaussianKernel(aperture_size, -1) * cv2.getGaussianKernel(
       aperture_size, -1).T
5      gy, gx = np.gradient(g)
6
7      # Filter with the kernel computed below and apply to the image and retrieve
        gradients
8      Ix = cv2.filter2D(image, cv2.CV_32F, gx)
9      Iy = cv2.filter2D(image, cv2.CV_32F, gy)
10
```

```
11      return Ix, Iy
```

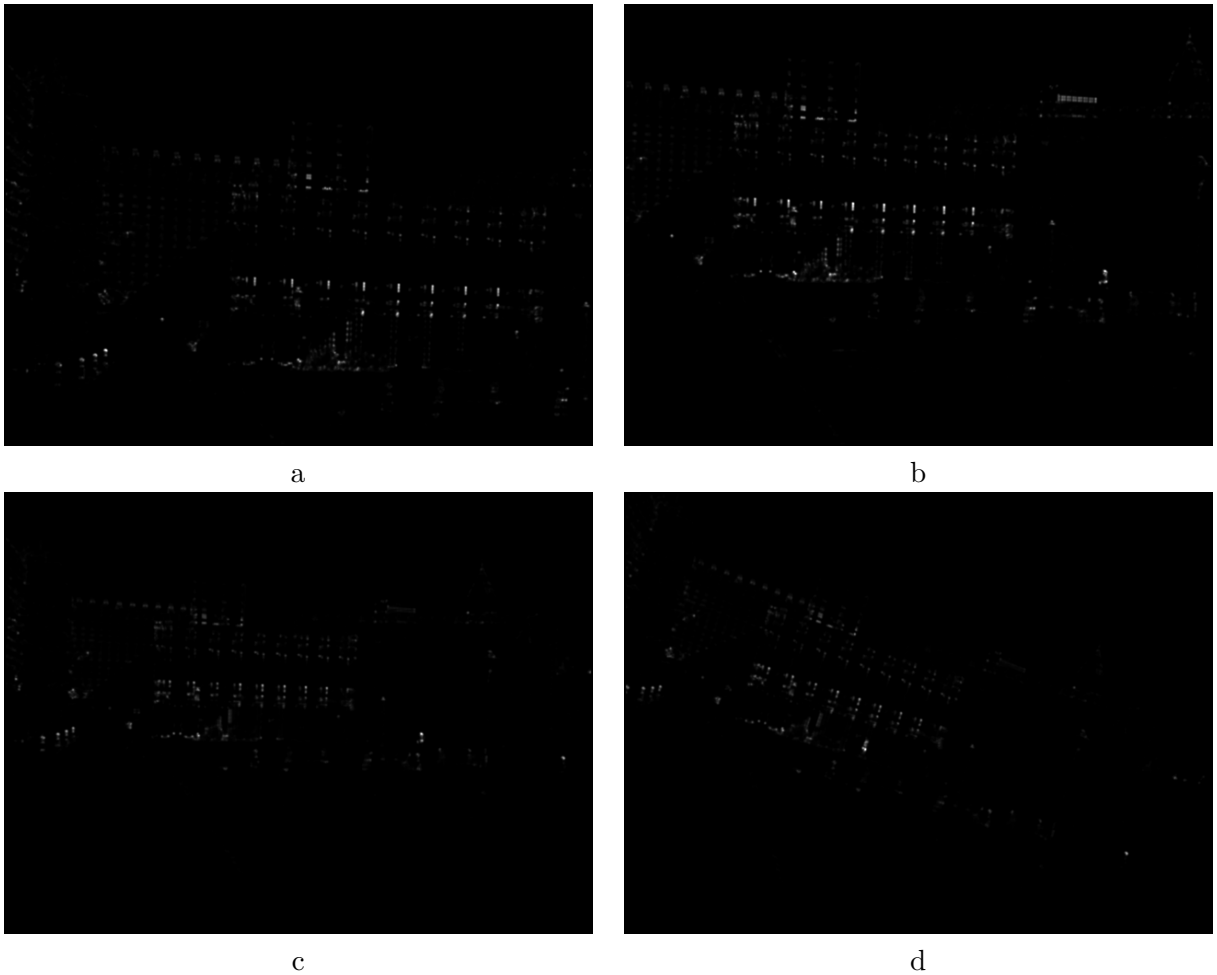
1.2 Harris Value Computation

The Figures 2.a,b,c,d present respectively the Harris value for simA, simB, transA and transB. The weights used for the computation correspond to a Gaussian kernel of size 5 (thus accounting for the 24 surrounding pixels in the sum). The Harris scoring is done using ? = 0.04 which is the default value used in the OpenCV implementation (cornerHarris).

The formula we applied in the following code is :

$$H = \sum_p w_p \nabla I_p (\nabla I_p)^T = \sum_p w_p \begin{pmatrix} I_{x_p}^2 & I_{x_p} I_{y_p} \\ I_{x_p} I_{y_p} & I_{y_p}^2 \end{pmatrix} =$$

$$= \sum_p \begin{pmatrix} w_p I_{x_p}^2 & w_p I_{x_p} I_{y_p} \\ w_p I_{x_p} I_{y_p} & w_p I_{y_p}^2 \end{pmatrix} = \begin{pmatrix} \sum_p w_p I_{x_p}^2 & \sum_p w_p I_{x_p} I_{y_p} \\ \sum_p w_p I_{x_p} I_{y_p} & \sum_p w_p I_{y_p}^2 \end{pmatrix}$$



a b c d

Figure 2: *a.* ps4-1-2-transA. *b.* ps4-1-2-transB. *c.* ps4-1-2-simA. *d.* ps4-1-2-simB.

```
def Harris_corner_response(Ix,Iy,alpha,window):

    Ix2 = Ix**2
    Iy2 = Iy**2
    Ixy = np.multiply(Ix,Iy)

    rows,cols = Ix.shape
    window_size = window.shape[0]

    R = np.zeros([rows,cols])

    for r in range(int(floor(window_size / 2.0)), int(rows - floor(window_size / 2.0))):
        min_r = int(r-floor(window_size/2.0))
        max_r = int(r+ceil(window_size/2.0))
        for c in range(int(floor(window_size / 2.0)), int(cols - floor(window_size / 2.0))):

            # Initializing the matrix M

            M = np.zeros([2,2])

            # Compute the dimension of the window in function of the pixel location

            min_c = int(c-floor(window_size/2.0))
            max_c = int(c+ceil(window_size/2.0))

            # Determining the matrix M values

            M[0, 0] = np.sum(np.multiply(window, Ix2[min_r:max_r, min_c:max_c]))
            M[1, 0] = np.sum(np.multiply(window, Ixy[min_r:max_r, min_c:max_c]))
            M[0, 1] = np.sum(np.multiply(window, Ixy[min_r:max_r, min_c:max_c]))
            M[1, 1] = np.sum(np.multiply(window, Iy2[min_r:max_r, min_c:max_c]))


            # Compute the value of R for the matrix M at pixel location [r,c]

            R[r,c] = np.linalg.det(M) - alpha*np.trace(M)

    return R
```

1.3 Harris Value Filtering

The filtering of the Harris value is achieved using a combination of thresholding and non-maximum suppression.
To begin with, every point of the matrix below the threshold is put to zero. Then, the current maximum is found and recorded as a detected corner while its value in the matrix is set to 0, along with all the points lying in the window around (non-maximum suppression). The resulting list of coordinates is used to draw circles around each detected corner on the original image, as shown in Figures 3.a, b, c and d.

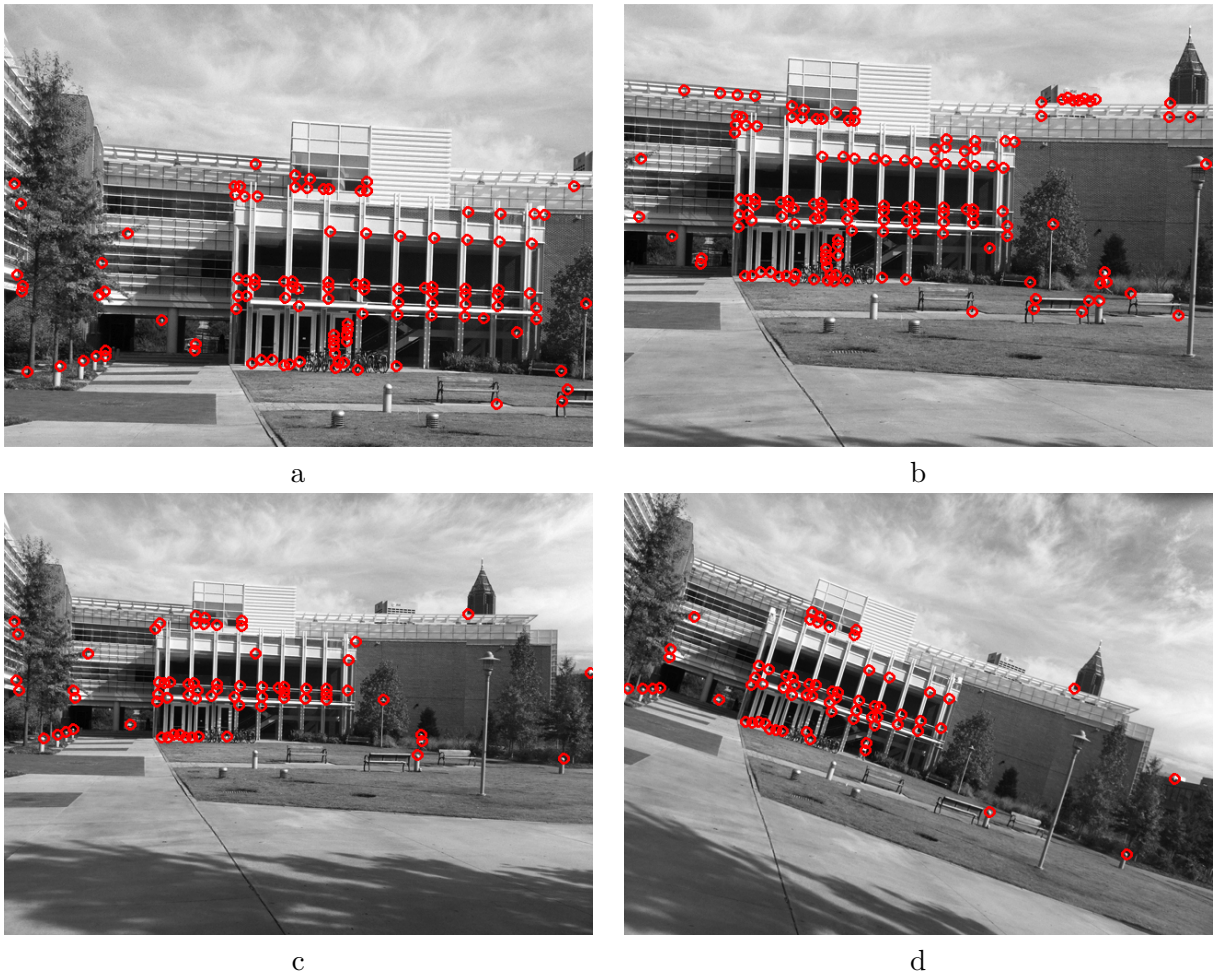


Figure 3: *a.* ps4-1-3-transA. *b.* ps4-1-3-transB. *c.* ps4-1-3-simA. *d.* ps4-1-3-simB.

We notice that some pixels belonging to the same physical object do not appear exactly the same on the two images of a pair. This is particularly true for the translational case, where the number of detected corners (using the same Harris and filtering parameters) differs quite a lot. This difference cannot be simply explained by the extra corners detected on the right image that were invisible on the left one. We see for example many new matches on the building's facade, where some of the corners at the pylons get detected twice on the right, while new corners appear at the top and among the bicycles.
This could be explained by a slight change in lighting and shadowing which has quite a large effect on the detection which appears to be very sensitive to those.
Dual detection of corners does not seem worrying since we skipped corners too close to each other using the non-maximum suppression window. This will be accounted for in the RANSAC, supposing we use a threshold during this step that is equal or smaller to the non-maximum suppression window during the consensus voting.
The detection of corners for the similarity case leads to approximately the same number

of corners detected. We can see some differences though, with a few new detection on the building?s facade and in the grass. One noticeable one is the background street lamp that is no longer detected on the second image. The intensity of the top part is noticably lower, probably due to a lighting variation.

```
def Harris_thresholding(R, threshold):
    R[R < threshold] = 0
    return R

def non_maxima_suppression(Rt,w):
    corners = []
    while np.argwhere(Rt > 0).shape[0]:
        i = np.unravel_index(np.argmax(Rt), Rt.shape) # Find current maximum

        #Prepare row slicing
        if w > i[0] or w > (Rt.shape[0] - i[0]):
            wr = min(i[0], Rt.shape[0] - i[0]) * 2
            if wr == 2*i[0]: #Close to upper boundary
                rslice = slice(int(i[0]-floor(wr/2.0)), int(i[0]+ceil(w/2.0)))
            else: # Close to lower boundary
                rslice = slice(int(i[0]-floor(w/2.0)), int(i[0]+ceil(wr/2.0)))
        else:
            rslice = slice(int(i[0]-floor(w/2.0)), int(i[0]+ceil(w/2.0)))

        #Prepare column slicing
        if w > i[1] or w > (Rt.shape[1] - i[1]):
            wc = min(i[1], Rt.shape[1] - i[1]) * 2
            if wc == 2*i[1]: #Close to left boundary
                cslice = slice(int(i[1]-floor(wc/2.0)), int(i[1]+ceil(w/2.0)))
            else: #Close to right boundary
                cslice = slice(int(i[1]-floor(w/2.0)), int(i[1]+ceil(wc/2.0)))
        else:
            cslice = slice(int(i[1]-floor(w/2.0)), int(i[1]+ceil(w/2.0)))

        Rt[rslice, cslice] = 0 #Zero out all values inside window
        corners.append(tuple(i))

    return corners
```

2 SIFT features

2.1 Gradient Orientation Computation

The direction of the gradients is simply computed by taking the two gradient intensity matrices computed at the previous step. The gradient orientation is then represented on the image using simple arrows. Figure 4a, b show the result for the similarity pair while Figure 4c, d are for the translation pair.

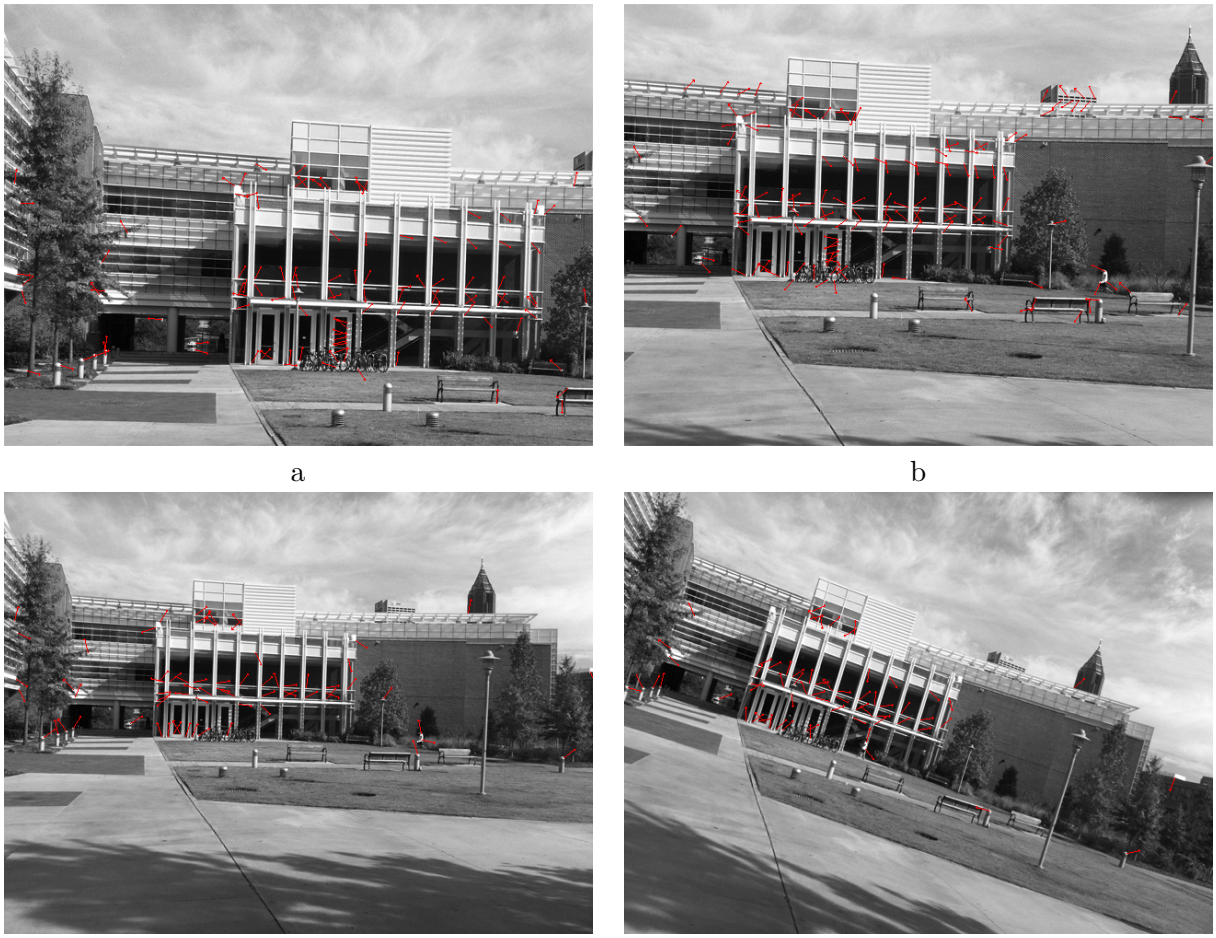


Figure 4: *a.* ps4-2-1-transA. *b.* ps4-2-1-transB. *c.* ps4-2-1-simA. *d.* ps4-2-1-simB.

```
def compute_angle(Ix,Iy): return np.arctan2(Iy, Ix)


def directions(image, corners,angle,l):
    image_col = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
    for c in corners:
        [y_a, x_a] = c
        x_b = int(floor(x_a + l * cos(angle[c])))
        y_b = int(floor(y_a + l * sin(angle[c])))
        cv2.arrowedLine(image_col, (x_a, y_a), (x_b, y_b), (0,0,255), 1)
    return image_col
```

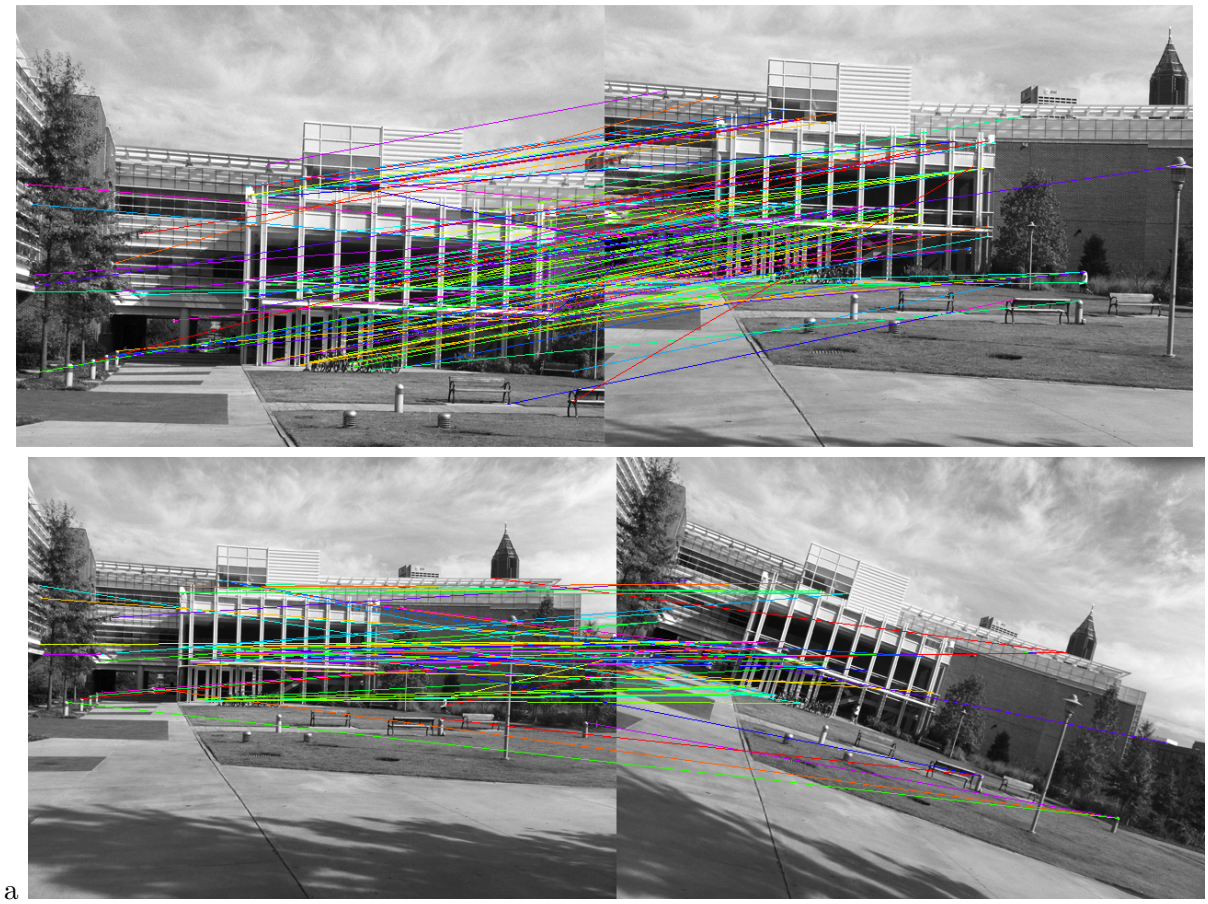
2.2 SIFT

The matching process using SIFT is done following several steps:

1. Keypoints objects are created based on the corners data (positions and orientations) ;
2. Descriptors are extracted using the SIFT compute method available in OpenCV ;

3. The descriptors are matched using the brute-force matcher available in OpenCV ;
4. A putative-pair image is generated, drawing lines in different colors (generated from the HSV space) to link matched descriptors in the two images.

The resulting putative-pair image for the similarity case is visible in Figure 5.a while Figure 5.b corresponds to the translational case.



a b

Figure 5: *a.* ps4-2-2-transA-transB. *b.* ps4-2-2-simA-simB.

```
def descriptors_description(image, corners, directions):

    #Keypoints description
    keypoints = []
    for c in corners:
        keypoints.append(cv2.KeyPoint(c[1], c[0], 1, directions[c]))
    #Computation of the descriptors
    sift = cv2.xfeatures2d.SIFT_create()
    _ , descriptors = sift.compute(image, keypoints)

    return keypoints, descriptors

```

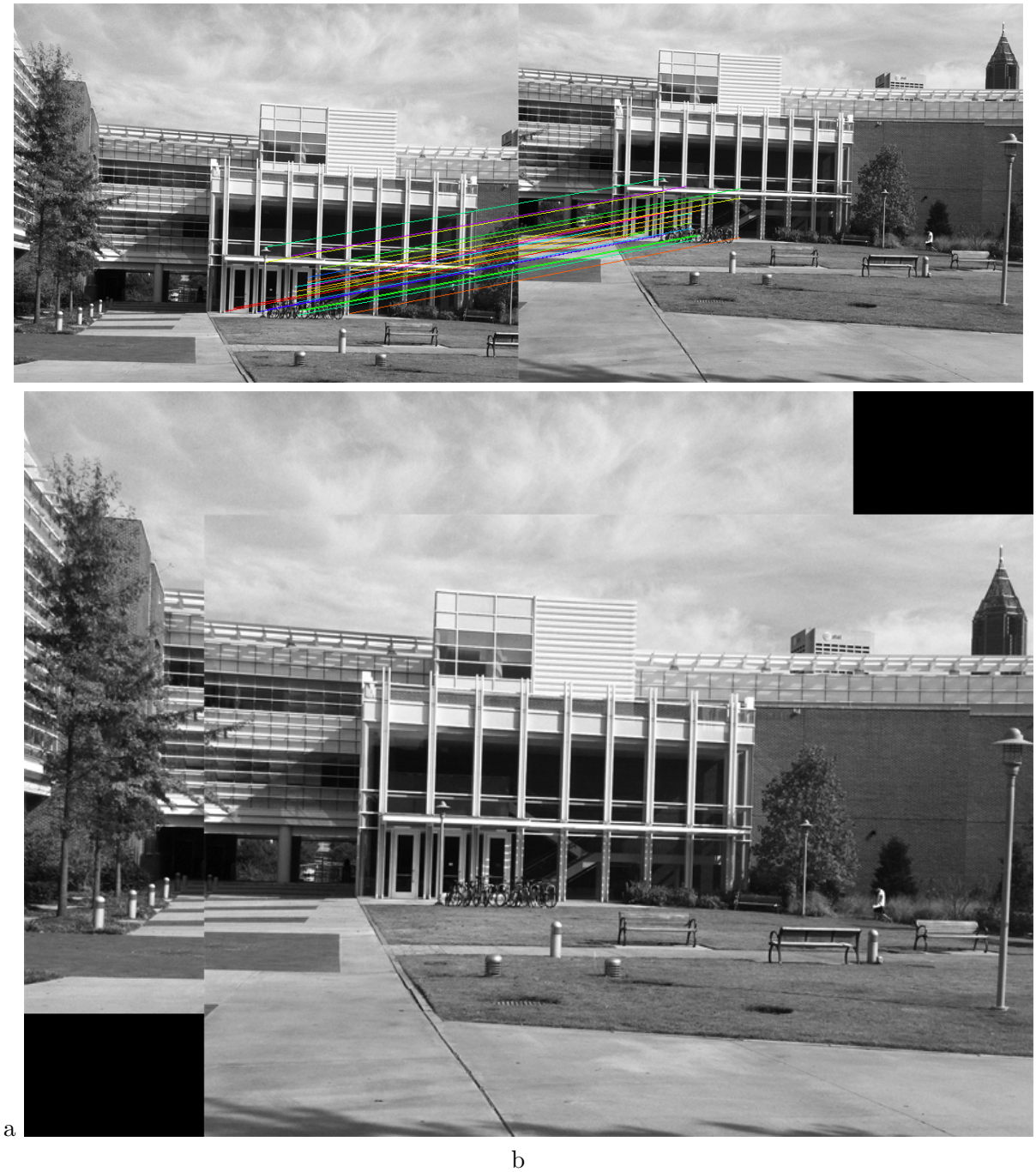
```
13 def descriptors_matching(desc1, desc2):
14
15     bfmatcher = cv2.BFMatcher()
16     matching_points = bfmatcher.match(desc1, desc2)
17     return matching_points
18
19
20 def generate_colors(N, sat, val):
21     #Used in order to identify the matches with different colors
22     colors = []
23     step = int(floor(179/N))
24     hsv = np.zeros([1, 1, 3], dtype="uint8")
25     hsv[0,0,1] = sat
26     hsv[0,0,2] = val
27     for n in range(N):
28         bgr = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
29         colors.append(tuple([int(bgr[0,0,0]), int(bgr[0,0,1]), int(bgr[0,0,2])
       ]))
30         hsv[0,0,0] += step
31     return colors
32
33 def draw_matches(image1, keypoints1, image2, keypoints2, matches):
34     image1_color = cv2.cvtColor(image1, cv2.COLOR_GRAY2BGR)
35     image2_color = cv2.cvtColor(image2, cv2.COLOR_GRAY2BGR)
36     result = np.hstack((image1_color, image2_color))
37     cols = image1.shape[1]
38
39     matches = sorted(matches, key = lambda x:x.distance)
40     colors = generate_colors(15, 255, 255)
41
42     for i,m in enumerate(matches):
43         kp1 = keypoints1[m.queryIdx]
44         kp2 = keypoints2[m.trainIdx]
45         x1 = int(kp1.pt[0])
46         y1 = int(kp1.pt[1])
47         x2 = int(kp2.pt[0]) + cols
48         y2 = int(kp2.pt[1])
49         cv2.line(result, (x1, y1), (x2, y2), colors[i%len(colors)], 1)
50     return result
```

3 RANSAC

3.1 RANSAC for Translation

The translational case for RANSAC consists in randomly taking a pair of matching keypoints, computing the (dx, dy) translation parameters and checking how many other pairs match this translation parameters (for a given tolerance value).
Using a tolerance of 10 pixels in the consensus computation we get the following result (represented in Figure 6.a): The best translation is $dx = 139$ and $dy = 95$ with 23 matches (21.1%) In order to get a better idea of how well the transformation applies, we use a custom function to draw the images on top of each other, which results in Figure 6.b. We notice that although the junction looks pretty good on the ground portion at the left, other parts mismatch completely, as top left part of the building. A variation in the mismatch seems strange considering the fact that the transformation between the left and right images is

purely translational. One possible explanation could be some kind of distorsion in the image, or a non-purely translational transformation.



a

b

Figure 6: *a.* ps4-3-1-transA-transB-lines. *b.* ps4-3-1-transA-transB-result.

```
1 def translational_ransac(matches, keypoints1, keypoints2, tolerance):
2
3     pool = list(matches)
```

```
    length = len(pool)
    consensus = {}


    while length:
        #Extract random element from the pool and remove it
        index = random.randrange(length)
        match = pool[index]
        pool[index] = pool[length-1]
        length -= 1

        #Compute translations
        dx = int(keypoints1[match.queryIdx].pt[0]) - int(keypoints2[match.
    trainIdx].pt[0])
        dy = int(keypoints1[match.queryIdx].pt[1]) - int(keypoints2[match.
    trainIdx].pt[1])

        #Compute consensus set
        if (dx, dy) not in consensus:
            set = []
            for m in matches:
                #Extract coordinates
                x1, y1 = [int(c) for c in keypoints1[m.queryIdx].pt]
                x2, y2 = [int(c) for c in keypoints2[m.trainIdx].pt]
                #Tests
                Xp = (x1 - x2) < dx + tolerance
                Xn = (x1 - x2) > dx - tolerance
                Yp = (y1 - y2) < dy + tolerance
                Yn = (y1 - y2) > dy - tolerance
                if (Xp and Xn) and (Yp and Yn):
                    set.append(m)
            consensus[(dx, dy)] = set

    #Get biggest consensus set
    translation = max(consensus, key=lambda k:len(consensus[k]))
    return translation, consensus[translation]


def best_match_after_translation(image1, image2, translation):
    rows, cols = image1.shape
    dx, dy = translation
    result = np.zeros([rows + dy, cols + dx, 3], dtype="uint8")
    result[:rows, :cols, :] = cv2.cvtColor(image1, cv2.COLOR_GRAY2BGR)
    result[dy:(rows+dy), dx:(cols+dx), :] = cv2.cvtColor(image2, cv2.
    COLOR_GRAY2BGR)
    return result
```

3.2 RANSAC for Similarity

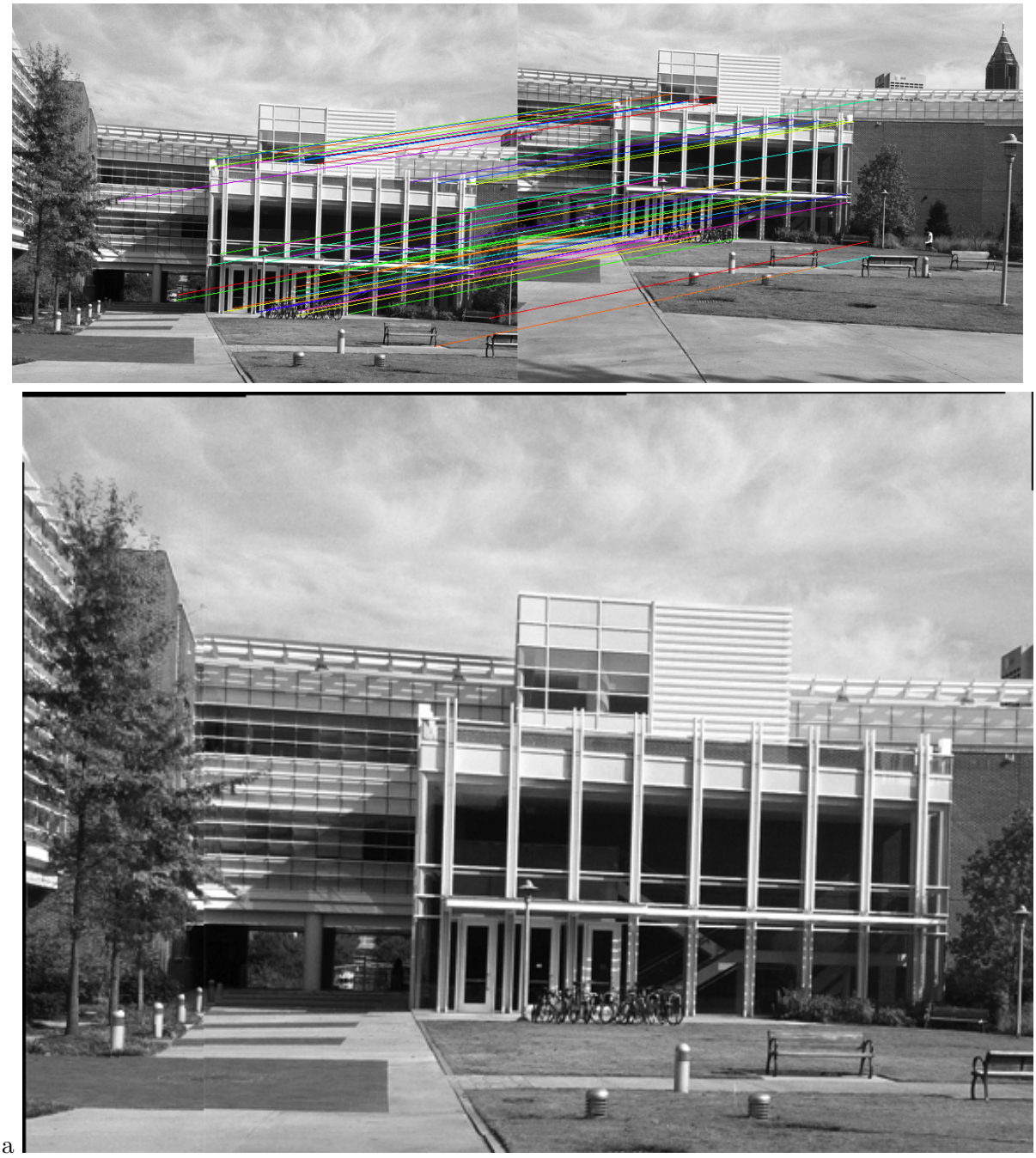
The similarity transformation case for RANSAC uses the same general method as the translational case. However, since computing the transformation now requires a set of two pairs of matching keypoints, we take a random sample from the list of all the combinations of matched pairs.

We then compute the 4 parameters (a,b,c,d) of the similarity transform using a least-square method, which gives us the similarity matrix. We then compute the consensus set (for

a given tolerance value) and return the matrix that corresponds to the largest set. Using a tolerance of 3 pixels in the consensus computation we get the following result (represented in Figure 7.a):

```
The best similarity with 50 matches (45.9%) is:
 [[ 8.99280576e-01 -5.58977279e-03 -1.01750945e+02]
 [ 5.58977279e-03  8.99280576e-01 -5.87517209e+01]]
```

In order to get a better idea of how well the transformation applies, we use a custom function to draw the images on top of each other, which results in Figure 7.b. The interpolation of pixels was not taken into account in this function since the idea was simply to get a rough idea of the matching. We notice that the images here match pretty well on the left side (grass and building junctions) but give worse result on the bottom part (tree shadows and carved line in the ground). The junction in the clouds is more difficult to appreciate, because of the low contrast and the many missing pixels. This again could be explained by a distorsion in the images, probably due to the lens of the camera.



a

b

Figure 7: *a.* ps4-3-2-transA-transB-lines. *b.* ps4-3-2-transA-transB-result.

```
def similarity_ransac(matches, keypoints1, keypoints2, tolerance):

    pool = list(itertools.combinations(list(range(len(matches))), 2))
    consensus = {}
    length = len(pool)
```

```
    while length:
        # Extract two random elements from the pool and remove it
        index = random.randrange(length)
        match1 = matches[pool[index][0]]
        match2 = matches[pool[index][1]]
        pool[index] = pool[length-1]
        length -= 1

        #Extract left point
        xl1, yl1 = [int(c) for c in keypoints1[match1.queryIdx].pt]
        xl2, yl2 = [int(c) for c in keypoints1[match2.queryIdx].pt]

        #Extract right point
        xr1, yr1 = [int(c) for c in keypoints2[match1.trainIdx].pt]
        xr2, yr2 = [int(c) for c in keypoints2[match2.trainIdx].pt]

        #Compute similarity matrix
        L = np.array([[xl1, -yl1, 1, 0],
                      [yl1, xl1, 0, 1],
                      [xl2, -yl2, 1, 0],
                      [yl2, xl1, 0, 1]])
        R = np.vstack([xr1, yr1, xr2, yr2])
        X,_,_,_ = np.linalg.lstsq(L,R)
        a = X[0,0]; b = X[1,0]; c = X[2,0]; d = X[3,0];
        S = np.array([[a, -b, c],
                      [b, a, d]])

        #Compute consensus set
        if (a, b, c, d) not in consensus:
            set = []
            for m in matches:
                #Extract coordinates
                xl, yl = [int(c) for c in keypoints1[m.queryIdx].pt]
                xr, yr = [int(c) for c in keypoints2[m.trainIdx].pt]
                #Compute resulting point
                P = S.dot(np.vstack([xl, yl, 1]))
                #Tests
                Xerr = abs(xr - P[0,0]) < tolerance
                Yerr = abs(yr - P[1,0]) < tolerance
                if Xerr and Yerr:
                    set.append(m)
            consensus[(a, b, c, d)] = set

    #Get biggest consensus set
    params = max(consensus, key=lambda k:len(consensus[k]))
    similarity = np.array([[params[0], -params[1], params[2]],
                           [params[1], params[0], params[3]]])
    return similarity, consensus[params]


def best_match_after_similirarity(image1, image2, similarity):

    rows, cols = image1.shape
    dx = int(floor(cols/2))
    dy = int(floor(rows/2))

    image1_bgr = cv2.cvtColor(image1, cv2.COLOR_GRAY2BGR)
```

```
63      image2_bgr = cv2.cvtColor(image2, cv2.COLOR_GRAY2BGR)
64
65      result = np.zeros([2*rows, 2*cols, 3], dtype="uint8")
66
67      min_r = result.shape[0]
68      max_r = 0
69      min_c = result.shape[1]
70      max_c = 0
71      for r in range(rows):
72          for c in range(cols):
73              P = similarity.dot(np.vstack([c, r, 1]))
74              y = int(P[1,0]) + dy
75              x = int(P[0,0]) + dx
76              result[y, x, :] = image1_bgr[r,c,:]
77              min_r = y if y < min_r else min_r
78              max_r = y if y > max_r else max_r
79              min_c = x if x < min_c else min_c
80              max_c = x if x > max_c else max_c
81
82      result[dy:(rows+dy), dx:(cols+dx), :] = image2_bgr
83
84      return result[min_r:max_r, min_c:max_c, :]
```