Computer Vision Assignment-2 Report

By - Devansh Gupta (20171100)

Image Mosaicing

Image stitching is the process of combining multiple photographic images with overlapping fields of view to produce a segmented panorama or high-resolution image. The image stitching process can be divided into three main components: image registration, calibration, and blending.

PROCEDURE:-

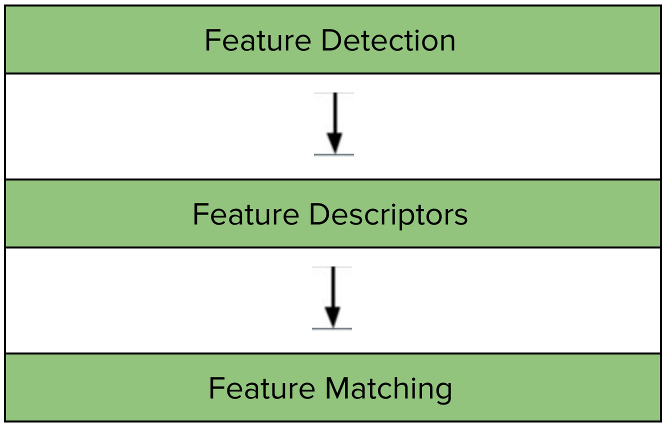
1. *Use any feature detector and descriptor (e.g. SIFT) to find matches between two partially overlapping images.*

We use feature detector to find the salient, stable feature points in an image. To ensure our featuring matching is robust we try to select features which are not affected by:

- Object position/pose
- Scale
- Illumination
- Minor image artifacts/noise/blur

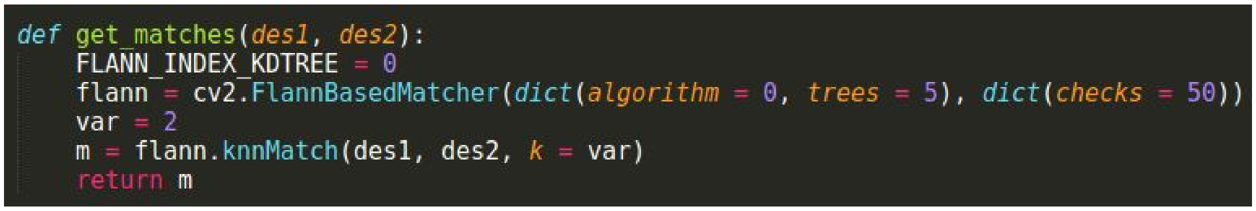
After Identifying the features we use descriptors to describe the properties of these robust features. Then we do feature matching between images with partial overlap, using these descriptors. We are using SURF detector to find the features.

We use cv2.FlannBasedMatcher(), cv2.drawMatchesKnn and cv2.xfeatures2d.SURF_create(), for feature matching:

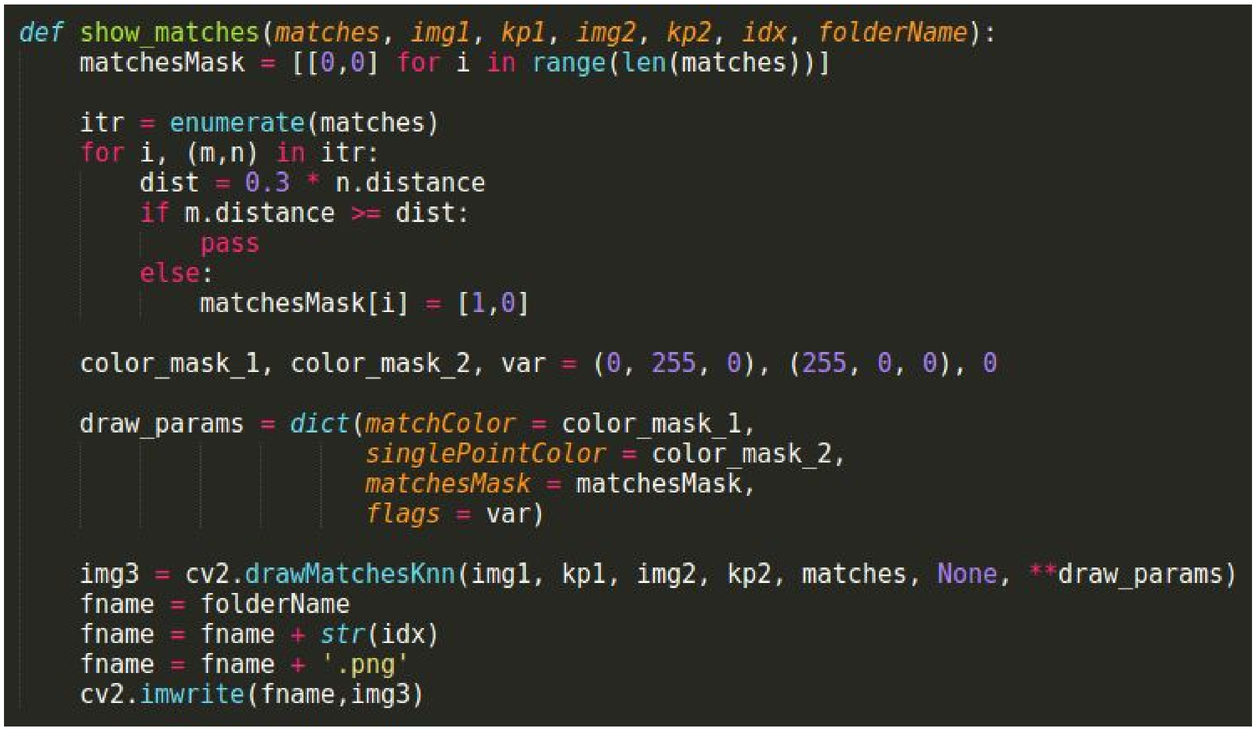


Functions:

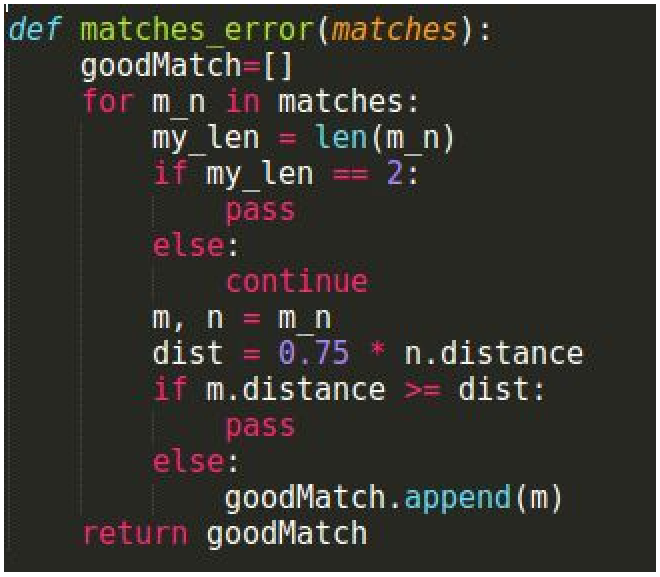
- cv2.xfeatures2d.SURF_create()
- cv2.FlannBasedMatcher()
- cv2.drawMatchesKnn



```
def get_matches(des1, des2):
    FLANN_INDEX_KDTREE = 0
    flann = cv2.FlannBasedMatcher(dict(algorithm = 0, trees = 5), dict(checks = 50))
    var = 2
    m = flann.knnMatch(des1, des2, k = var)
    return m
```

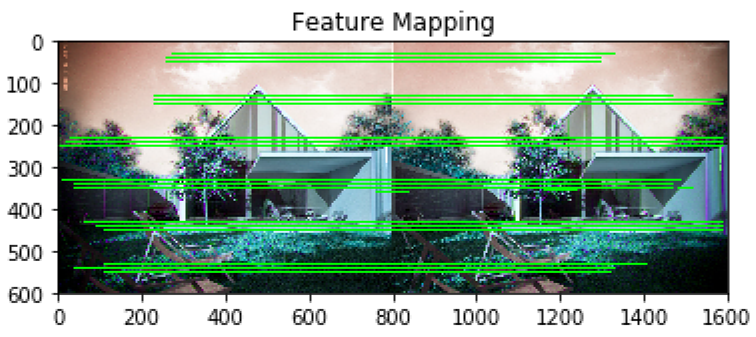


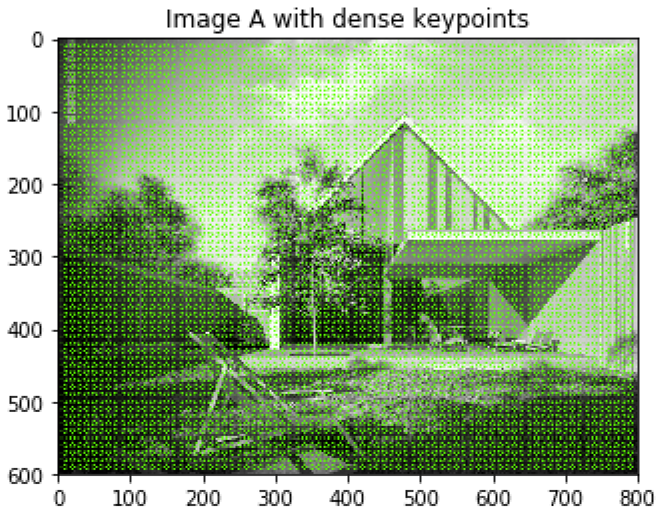
```
def show_matches(matches, img1, kp1, img2, kp2, idx, folderName):
    matchesMask = [[0,0] for i in range(len(matches))]

    itr = enumerate(matches)
    for i, (m,n) in itr:
        dist = 0.3 * n.distance
        if m.distance >= dist:
            pass
        else:
            matchesMask[i] = [1,0]

    color_mask_1, color_mask_2, var = (0, 255, 0), (255, 0, 0), 0

    draw_params = dict(matchColor = color_mask_1,
                       singlePointColor = color_mask_2,
                       matchesMask = matchesMask,
                       flags = var)

    img3 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, matches, None, **draw_params)
    fname = folderName
    fname = fname + str(idx)
    fname = fname + '.png'
    cv2.imwrite(fname,img3)
```



```
def matches_error(matches):
    goodMatch=[]
    for m_n in matches:
        my_len = len(m_n)
        if my_len == 2:
            pass
        else:
            continue
        m, n = m_n
        dist = 0.75 * n.distance
        if m.distance >= dist:
            pass
        else:
            goodMatch.append(m)
    return goodMatch
```

Results for feature mapping:





Estimating homography matrix

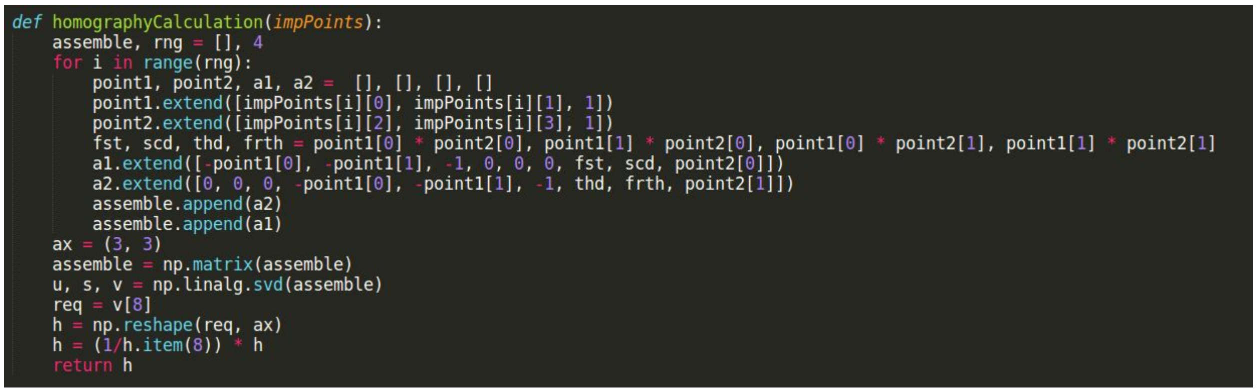
We calculate the homography matrix between the two images by using the homographyCalculation() and using them in ransac. We perform RANSAC to get better results of the homography matrix.

Then by using the final homography matrix we transform the features of one image into the other images reference frame.

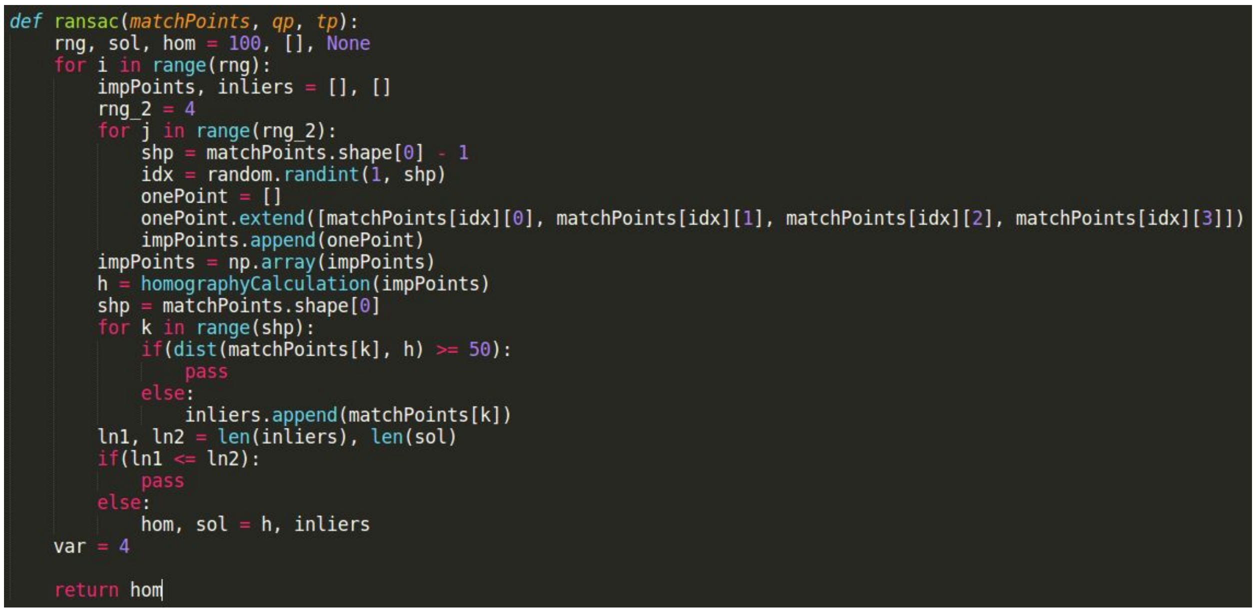
Functions:

- ransac()
- dist()
- homographyCalculation()

Code:

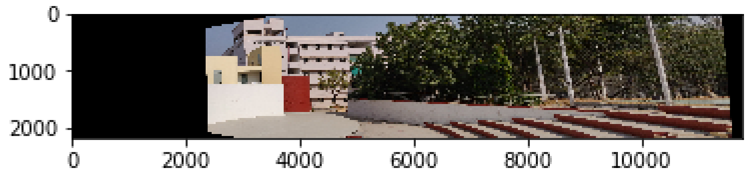


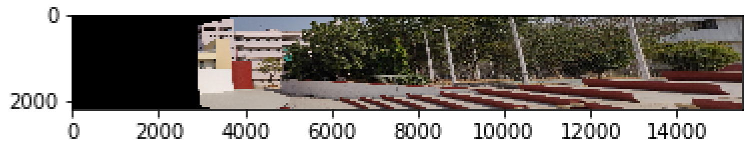
```python
def homographyCalculation(impPoints):
    assemble, rng = [], 4
    for i in range(rng):
        point1, point2, a1, a2 =  [], [], [], []
        point1.extend([impPoints[i][0], impPoints[i][1], 1])
        point2.extend([impPoints[i][2], impPoints[i][3], 1])
        fst, scd, thd, frth = point1[0] * point2[0], point1[1] * point2[0], point1[0] * point2[1], point1[1] * point2[1]
        a1.extend([-point1[0], -point1[1], -1, 0, 0, 0, fst, scd, point2[0]])
        a2.extend([0, 0, 0, -point1[0], -point1[1], -1, thd, frth, point2[1]])
        assemble.append(a2)
        assemble.append(a1)
    ax = (3, 3)
    assemble = np.matrix(assemble)
    u, s, v = np.linalg.svd(assemble)
    req = v[8]
    h = np.reshape(req, ax)
    h = (1/h.item(8)) * h
    return h
```

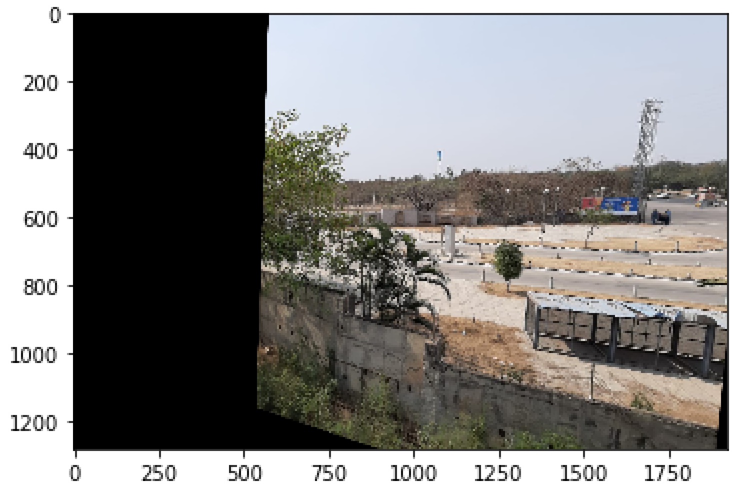


```python
def ransac(matchPoints, qp, tp):
    rng, sol, hom = 100, [], None
    for i in range(rng):
        impPoints, inliers = [], []
        rng_2 = 4
        for j in range(rng_2):
            shp = matchPoints.shape[0] - 1
            idx = random.randint(1, shp)
            onePoint = []
            onePoint.extend([matchPoints[idx][0], matchPoints[idx][1], matchPoints[idx][2], matchPoints[idx][3]])
            impPoints.append(onePoint)
        impPoints = np.array(impPoints)
        h = homographyCalculation(impPoints)
        shp = matchPoints.shape[0]
        for k in range(shp):
            if(dist(matchPoints[k], h) >= 50):
                pass
            else:
                inliers.append(matchPoints[k])
        ln1, ln2 = len(inliers), len(sol)
        if(ln1 <= ln2):
            pass
        else:
            hom, sol = h, inliers
    var = 4

    return hom
```

Results:







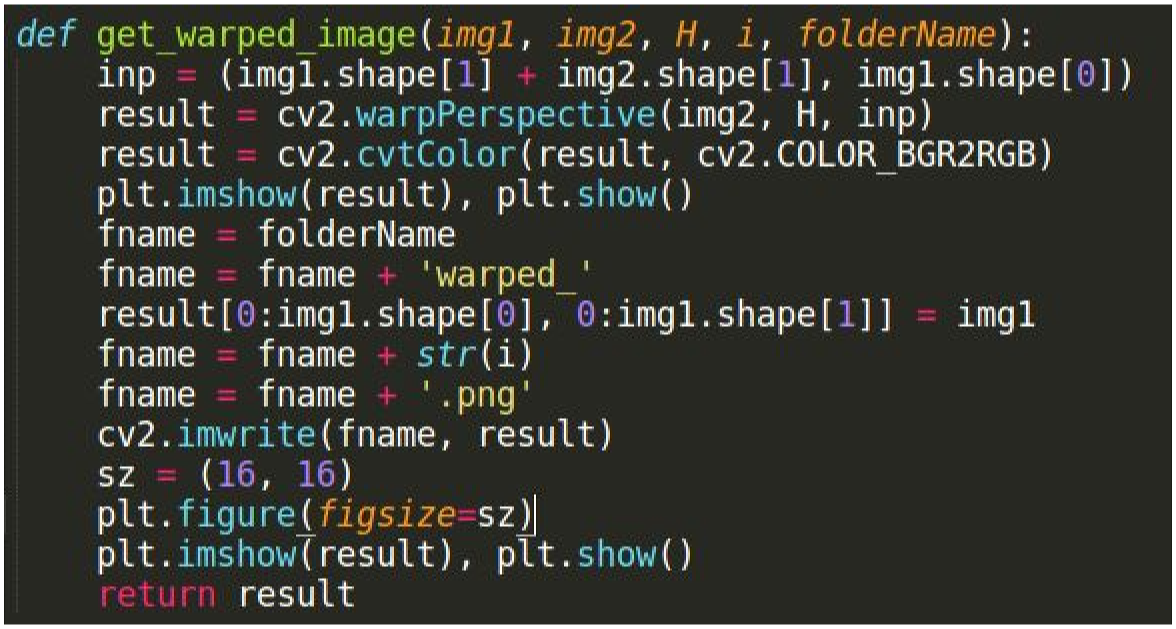
Transforming images to one reference frame and stitching them

After the Homography Matrix and transformation, we have all the features with respect to the coordinate frame of the first image. We then use the cv2.warpPerspective() function to do image stitching and get the final warped image.

Functions:

- get_warped_image()
- cv2.warpPerspective()

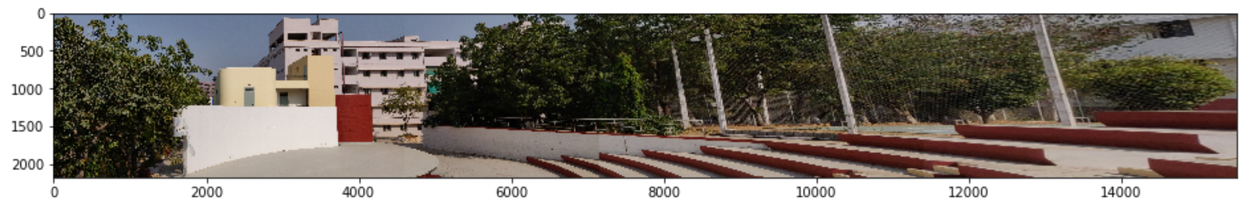
Code:

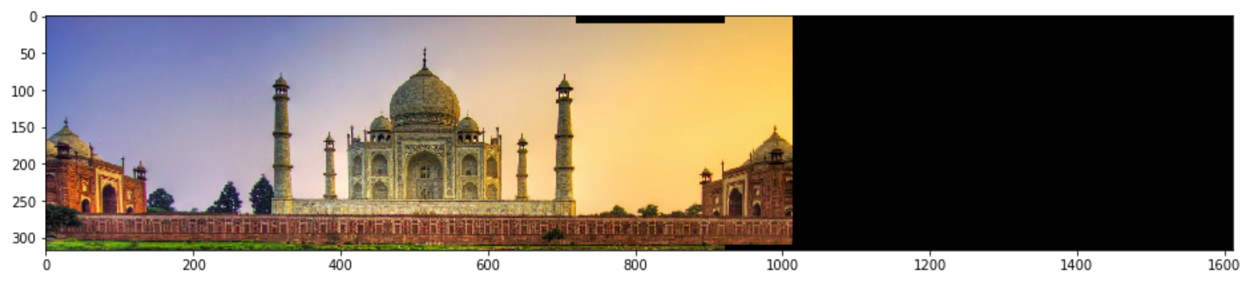


```python
def get_warped_image(img1, img2, H, i, folderName):
    inp = (img1.shape[1] + img2.shape[1], img1.shape[0])
    result = cv2.warpPerspective(img2, H, inp)
    result = cv2.cvtColor(result, cv2.COLOR_BGR2RGB)
    plt.imshow(result), plt.show()
    fname = folderName
    fname = fname + 'warped_'
    result[0:img1.shape[0], 0:img1.shape[1]] = img1
    fname = fname + str(i)
    fname = fname + '.png'
    cv2.imwrite(fname, result)
    sz = (16, 16)
    plt.figure(figsize=sz)
    plt.imshow(result), plt.show()
    return result
```

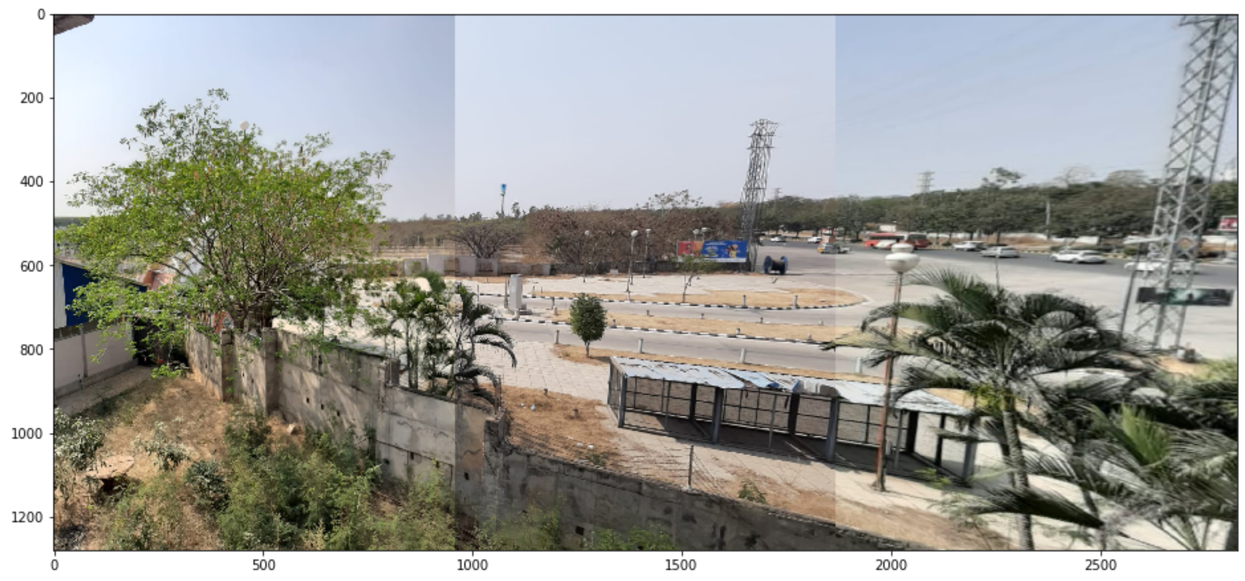
Results:

for provided dataset:





for the images taken by my own camera:



Bonus question

1) Yes, we can think of an algorithm to handle jumbled order. basic idea is to compute pairwise correlation of the end of one image with the beginning of all images and store the best result of this for every single image. Do the same for the beginning of one image with end of all images. now, the image with lowest best-correlation with its end and the beginning of any other image will be the end of our panaroma. also the image with lowest best-correlation of its beginning with the end of any other image will be start of our panaroma. after this we can construct a chain using the prestored information of successor of each image. This will give us the perfect order. now all we need to do is perform individual image stitching on all the consecutive pairs in the orders. this should give us the panaroma.

2) Yes, i think it is possible to stitch images of different scenes which are noisy without human intervention. but the noise should be below a threshold.
We can first cluster images of same type using a clustering algorithm like K-means. now, we can perform the above algorithm of part 1 in these individual clusters. using thresholding we can handle the problems arising due to the noisy nature of the images.

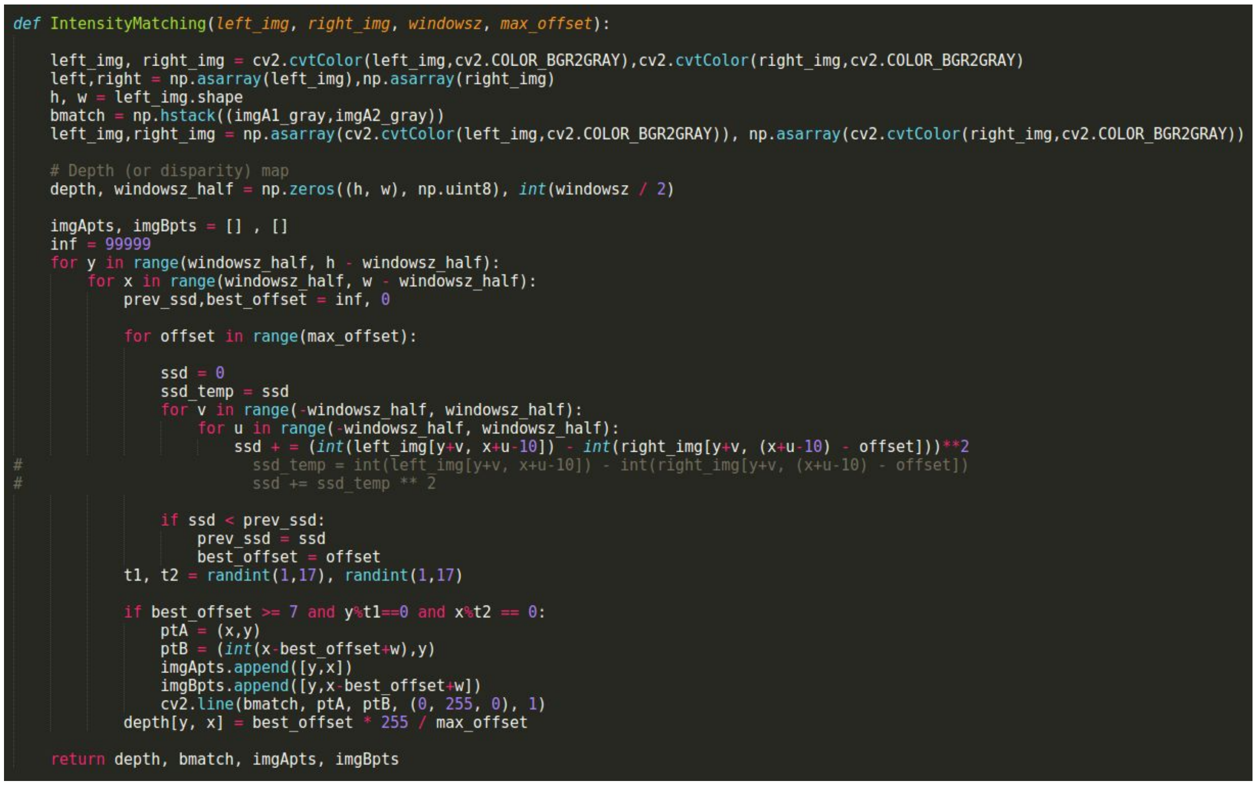
3) This being said, if the images are very noisy, it will be not be possible without human intervention as the ordering part might be hampered.

Stereo Correspondence

Intensity based correlation

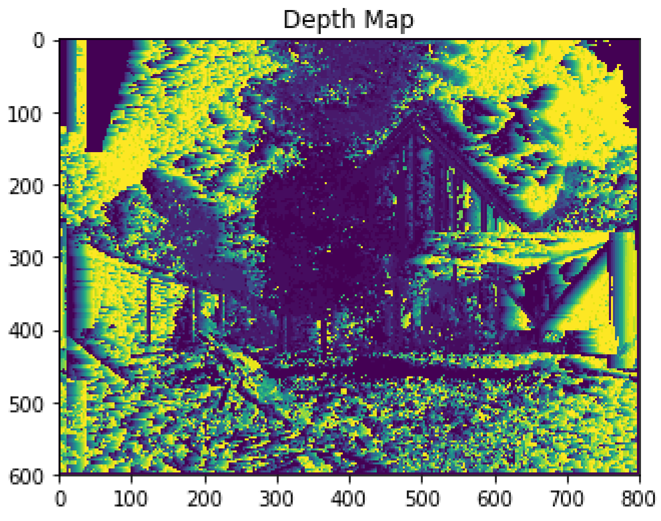
We select a window size. We move the window across the image and perform local window search in which we iteratively find the sum of squared differences (**SSD**) value for the block. We calculate this value and check if it smaller than the previous ssd at this block and update the value of the SSD of the block appropriately. We then get the optimal offset value and match the corresponding pixels.

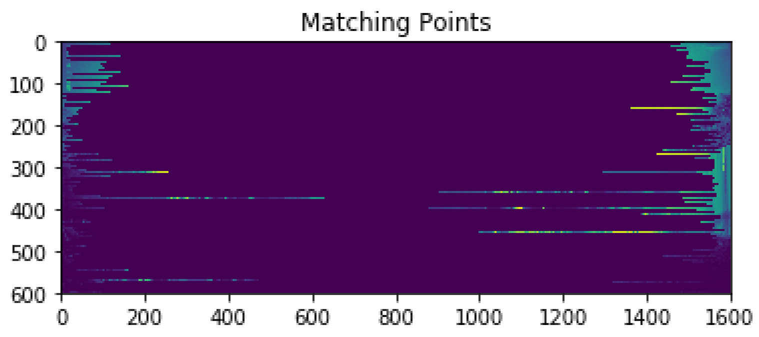
Code :-



```python
def IntensityMatching(left_img, right_img, windowsz, max_offset):

    left_img, right_img = cv2.cvtColor(left_img,cv2.COLOR_BGR2GRAY),cv2.cvtColor(right_img,cv2.COLOR_BGR2GRAY)
    left,right = np.asarray(left_img),np.asarray(right_img)
    h, w = left_img.shape
    bmatch = np.hstack((imgA1_gray,imgA2_gray))
    left_img,right_img = np.asarray(cv2.cvtColor(left_img,cv2.COLOR_BGR2GRAY)), np.asarray(cv2.cvtColor(right_img,cv2.COLOR_BGR2GRAY))

    # Depth (or disparity) map
    depth, windowsz_half = np.zeros((h, w), np.uint8), int(windowsz / 2)

    imgApts, imgBpts = [] , []
    inf = 99999
    for y in range(windowsz_half, h - windowsz_half):
        for x in range(windowsz_half, w - windowsz_half):
            prev_ssd,best_offset = inf, 0

            for offset in range(max_offset):

                ssd = 0
                ssd_temp = ssd
                for v in range(-windowsz_half, windowsz_half):
                    for u in range(-windowsz_half, windowsz_half):
                        ssd + = (int(left_img[y+v, x+u-10]) - int(right_img[y+v, (x+u-10) - offset]))**2
#                         ssd_temp = int(left_img[y+v, x+u-10]) - int(right_img[y+v, (x+u-10) - offset])
#                         ssd += ssd_temp ** 2

                if ssd < prev_ssd:
                    prev_ssd = ssd
                    best_offset = offset
            t1, t2 = randint(1,17), randint(1,17)

            if best_offset >= 7 and y%t1==0 and x%t2 == 0:
                ptA = (x,y)
                ptB = (int(x-best_offset+w),y)
                imgApts.append([y,x])
                imgBpts.append([y,x-best_offset+w])
                cv2.line(bmatch, ptA, ptB, (0, 255, 0), 1)
            depth[y, x] = best_offset * 255 / max_offset

    return depth, bmatch, imgApts, imgBpts
```

Results:



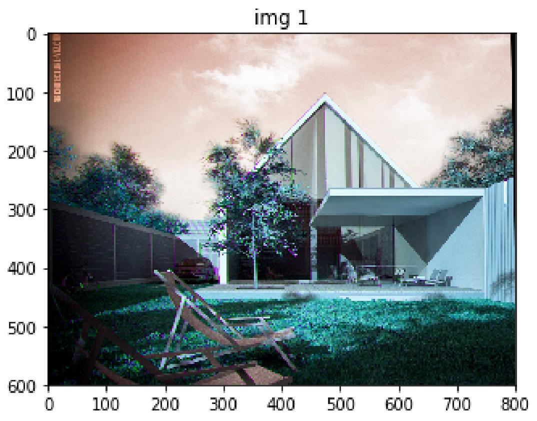


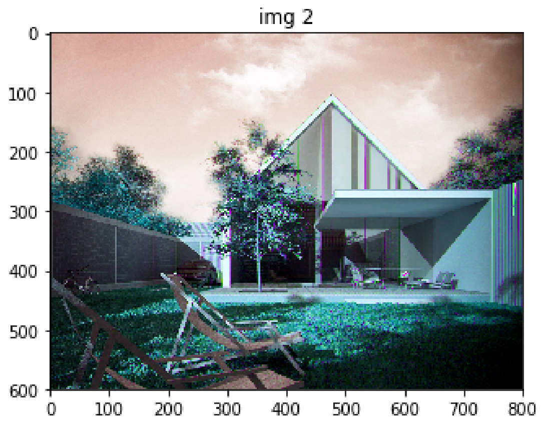
Rectifying images

```
def img_rectify(img1, img2):
    sift = cv2.xfeatures2d.SIFT_create()

    kp1, desc1 = sift.detectAndCompute(img1,None)
    kp2, desc2 = sift.detectAndCompute(img2,None)

    FLANN_INDEX_KDTREE = 1
    index_params = {'algorithm': FLANN_INDEX_KDTREE, 'trees': 10}
    search_params = {'checks': 50}
    flann = cv2.FlannBasedMatcher(index_params,search_params)
    matches = flann.knnMatch(desc1, desc2, k=2)

    pts1 = []
    pts2 = []
    for i, (m,n) in enumerate(matches):
        if m.distance < 0.65 * n.distance:
            pts1.append(kp1[m.queryIdx].pt)
            pts2.append(kp2[m.trainIdx].pt)
    pts1 = np.int32(pts1)
    pts2 = np.int32(pts2)
    F, _ = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)

    _, H1, H2 = cv2.stereoRectifyUncalibrated(pts1.flatten(),
pts2.flatten(), F, (img1.shape[0], img1.shape[1]), threshold=3)

    img1_rectified = cv2.warpPerspective(img1, H1, (img1.shape[1],
img1.shape[0]))
    img2_rectified = cv2.warpPerspective(img2, H2, (img2.shape[1],
img2.shape[0]))

    return img1_rectified, img2_rectified
```

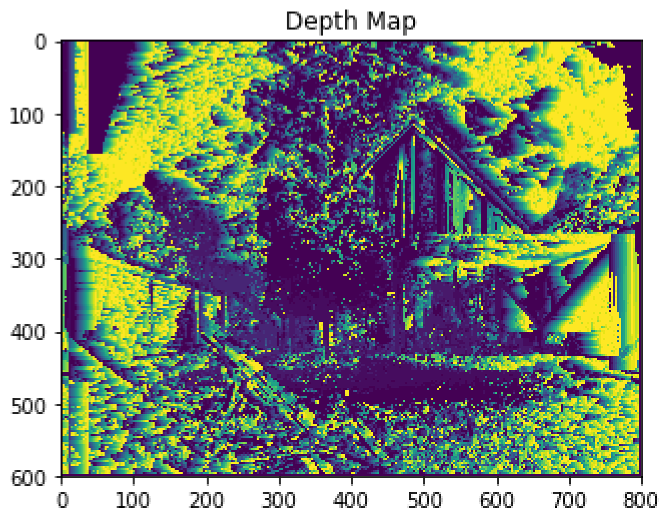
Results:

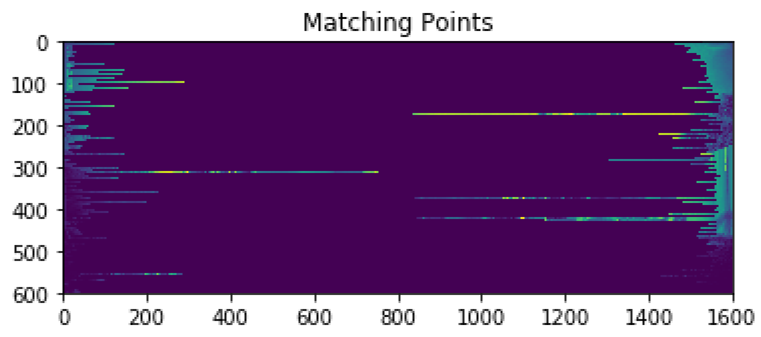
Rectified Images:





Results of intensity based correlation performed on rectified images:





Dynamic programming Solution

The above methods use Greedy Matching to generate the depth map. Dynamic time warping (DTW) is a well-known technique to find an optimal alignment of given images and compute depth. We use it to compute inter word dissimilarity. We use Dynamic programming to find the depth between the given images.

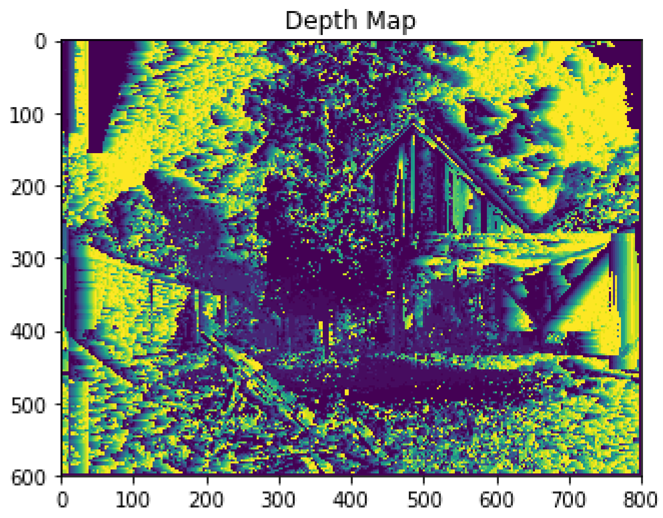
Code :-

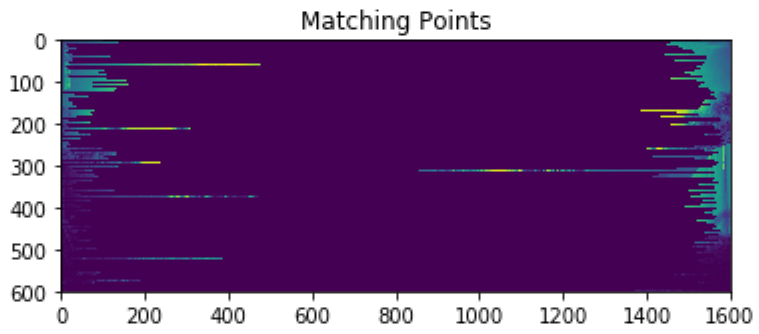
```
def DP_matching(leftimg, rightimg,wins,occlusionConstant = 0.1):
      h, w = leftimg.shape
      depth,dsi =np.zeros((h-wins, w-wins+exc)), np.zeros((w-wins, w-wins+ exc))
      for y in range(1,h-wins):
      C = np.zeros((w-wins, w-wins))
      Pointers = np.zeros((w-wins, w-wins))

      for leftX in range(1,w-wins+ exc):
            leftPatch = leftimg[y:y +wins + exc - 1,leftX : leftX + wins + exc - 1]

      for rightX in range(1,w-wins):
            rightPatch = rightimg[y : y + wins + exc - 1, rightX : rightX + wins + exc - 1]
            diffSq = (leftPatch - rightPatch)**2
            SSE = np.sum(np.sum(diffSq.T));
            dsi[rightX][leftX] = SSE;
```

```python
dsi = dsi/np.max(np.max(dsi));

for i in range(2,w-wins+exc):
      C[i][1] = C[i-1][1] + occlusionConstant
      C[1][i] = C[1][i-1] + occlusionConstant
      Pointers[i][1] = 2
      Pointers[1][i] = 3

for i in range(2,w-wins+exc):
      for j in range(2,w-wins+exc):
            c1 = C[i-1][j-1] + dsi[i][j]
            c2 = C[i-1][j] + occlusionConstant
            c3 = C[i][j-1] + occlusionConstant
            if(C[i][j] > c1):
            C[i][j] = c1
            Pointers[i][j] = 1
            if(C[i][j] > c2):
            C[i][j] = c2
            Pointers[i][j] = 2
            if(C[i][j] > c3):
            C[i][j] = c3
            Pointers[i][j] = 3

pathy,pathx = [],[]
for i in range(w-wins+exc-1,exc,-1):
      for j in range(w-wins+exc-1,exc,-1):
            if(Pointers[i][j]==2):
            i=i-1
            elif(Pointers[i][j]==3):
            j=j-1
            else:
            i = i-1
            j = j-1
            pathx.append(i)
            pathy.append(j)

newScanline = np.zeros((w-wins,1))

for zX in range(1, len(pathx)-1):
      xLoc = pathx[zX]
      yLoc = pathy[zX]
```

```
            if(zX < len(pathx)):
                if( xLoc == pathx[zX+1] + 1):
                if(yLoc == pathy[zX+1] + 1):
                    newScanline[xLoc] = math.sqrt((xLoc - yLoc)** 2)
            elif(~(zX < len(pathx) and xLoc == pathx[zX+1] + 1 and yLoc == pathy[zX+1] + 1)):
                newScanline[xLoc] = 0
        newScanline = newScanline.flatten()
        depth[y, :] = newScanline;
        depth = depth / np.max(np.max(depth))

        return depth
```

Results:

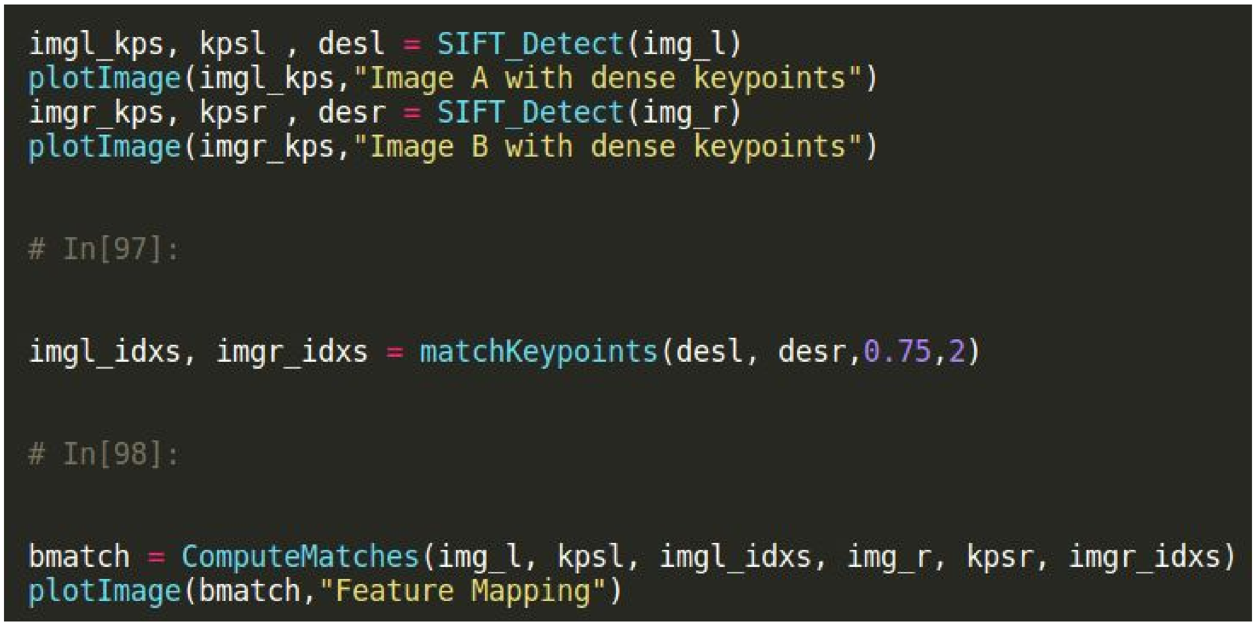




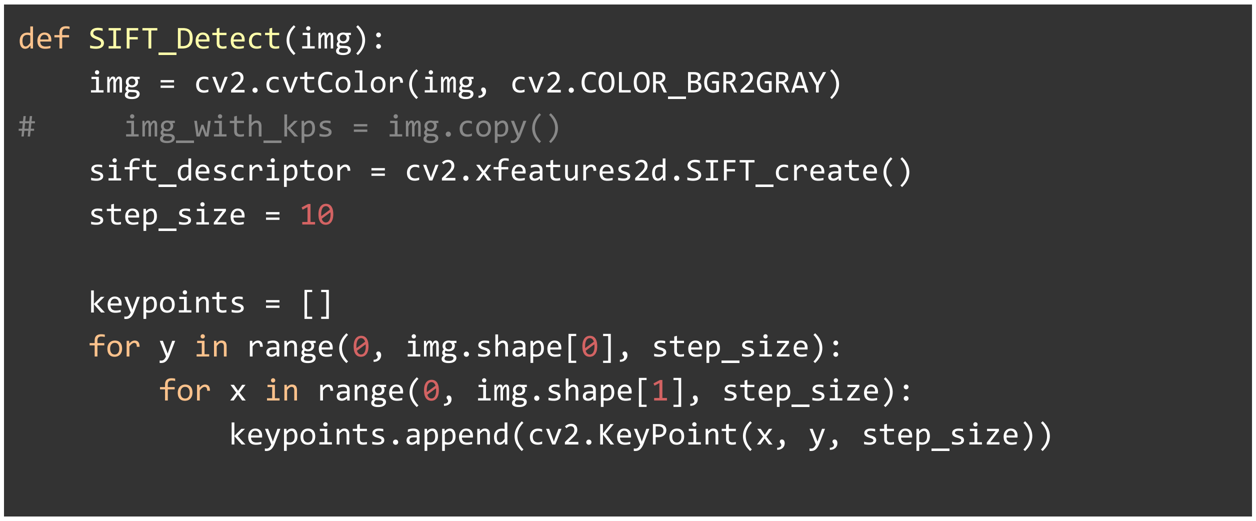
Question 2 bonus (SIFT based matching and comparison)

When we use normal SIFT descriptor we pass only specified key points to the function. The difference is that with dense SIFT you get a SIFT descriptor at every location, while with normal sift you get a SIFT descriptions at the locations determined by Lowe's algorithm.

Code :-

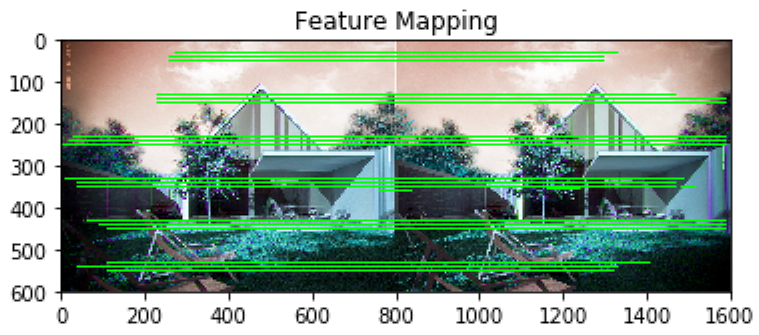


```
imgl_kps, kpsl , desl = SIFT_Detect(img_l)
plotImage(imgl_kps,"Image A with dense keypoints")
imgr_kps, kpsr , desr = SIFT_Detect(img_r)
plotImage(imgr_kps,"Image B with dense keypoints")


# In[97]:


imgl_idxs, imgr_idxs = matchKeypoints(desl, desr,0.75,2)


# In[98]:


bmatch = ComputeMatches(img_l, kpsl, imgl_idxs, img_r, kpsr, imgr_idxs)
plotImage(bmatch,"Feature Mapping")
```



```
def SIFT_Detect(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#     img_with_kps = img.copy()
    sift_descriptor = cv2.xfeatures2d.SIFT_create()
    step_size = 10

    keypoints = []
    for y in range(0, img.shape[0], step_size):
        for x in range(0, img.shape[1], step_size):
            keypoints.append(cv2.KeyPoint(x, y, step_size))
    keypoints, description = sift_descriptor.compute(img, keypoints)
    kps = np.float32([kp.pt for kp in keypoints])
    img_with_kps = cv2.drawKeypoints(img,keypoints,img, color=(93,
255, 0),flags=0)
    return img_with_kps, kps, description;
```

Results:



Approaches to the correspondence problem can be broadly classified into two categories: the intensity-based matching and the feature-based matching techniques.

intensity-based matching :-

In this method, the matching process is applied directly to the intensity profiles of the two images.

Feature-based matching :-

while in the second, features are first extracted from the images and the matching process is applied to the features.