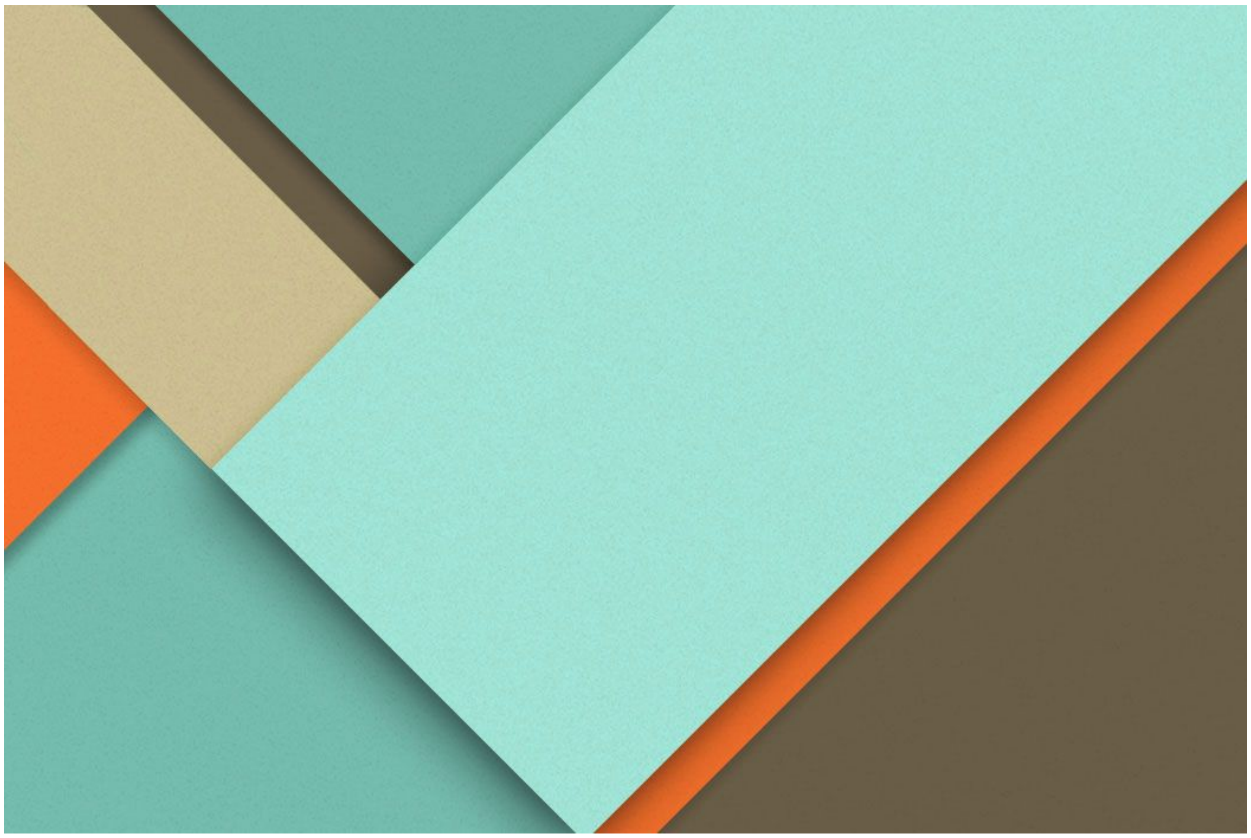


Image Mosaics

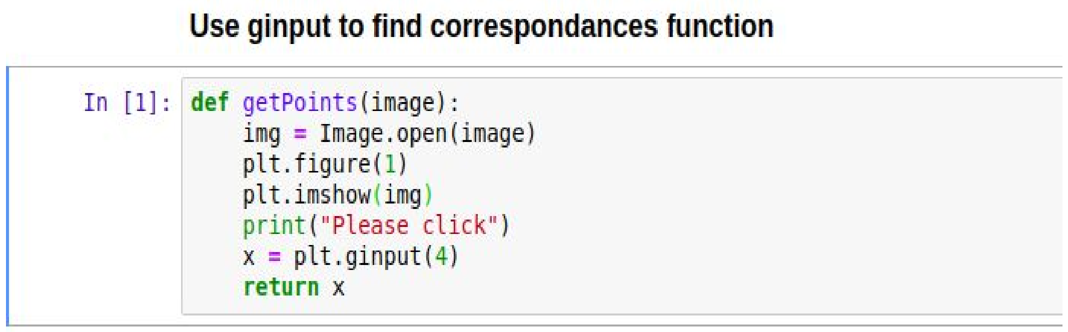
Overview

We will implement an image stitcher that uses image warping and homographies to automatically create an image mosaic.

Implementation details:

1) **Manually using *ginput* to find correspondences:**

For calculating the homography, we require four points from each image, we implement getPoints() function that return the coordinates of the chosen points that will later be used as matches in the homography function.



Use ginput to find correspondances function

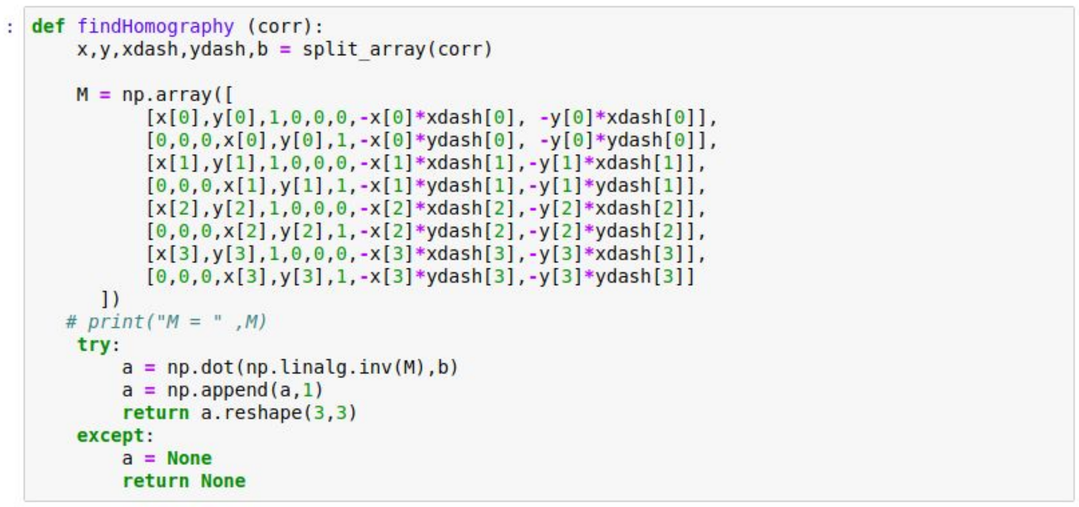
```
In [1]: def getPoints(image):
            img = Image.open(image)
            plt.figure(1)
            plt.imshow(img)
            print("Please click")
            x = plt.ginput(4)
            return x
```

2) Calculate the Homography:

With four matching points (xi,yi), we are able to solve the system of linear equations in figure(1) and implement findHomography() function that takes the correspondences as input, splits them into 4 vectors x,y,x',y' and returns the calculated homography.

$$\begin{pmatrix} x'_0 \\ y'_0 \\ x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0x'_0 & -y_0x'_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -x_0y'_0 & -y_0y'_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \end{pmatrix} . \begin{pmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \\ a_{20} \\ a_{21} \end{pmatrix}$$

figure(1)



```
: def findHomography (corr):
      x,y,xdash,ydash,b = split_array(corr)

      M = np.array([
            [x[0],y[0],1,0,0,0,-x[0]*xdash[0], -y[0]*xdash[0]],
            [0,0,0,x[0],y[0],1,-x[0]*ydash[0], -y[0]*ydash[0]],
            [x[1],y[1],1,0,0,0,-x[1]*xdash[1],-y[1]*xdash[1]],
            [0,0,0,x[1],y[1],1,-x[1]*ydash[1],-y[1]*ydash[1]],
            [x[2],y[2],1,0,0,0,-x[2]*xdash[2],-y[2]*xdash[2]],
            [0,0,0,x[2],y[2],1,-x[2]*ydash[2],-y[2]*ydash[2]],
            [x[3],y[3],1,0,0,0,-x[3]*xdash[3],-y[3]*xdash[3]],
            [0,0,0,x[3],y[3],1,-x[3]*ydash[3],-y[3]*ydash[3]]
        ])
     # print("M = " ,M)
      try:
          a = np.dot(np.linalg.inv(M),b)
          a = np.append(a,1)
          return a.reshape(3,3)
      except:
          a = None
          return None
```

3) Warping:

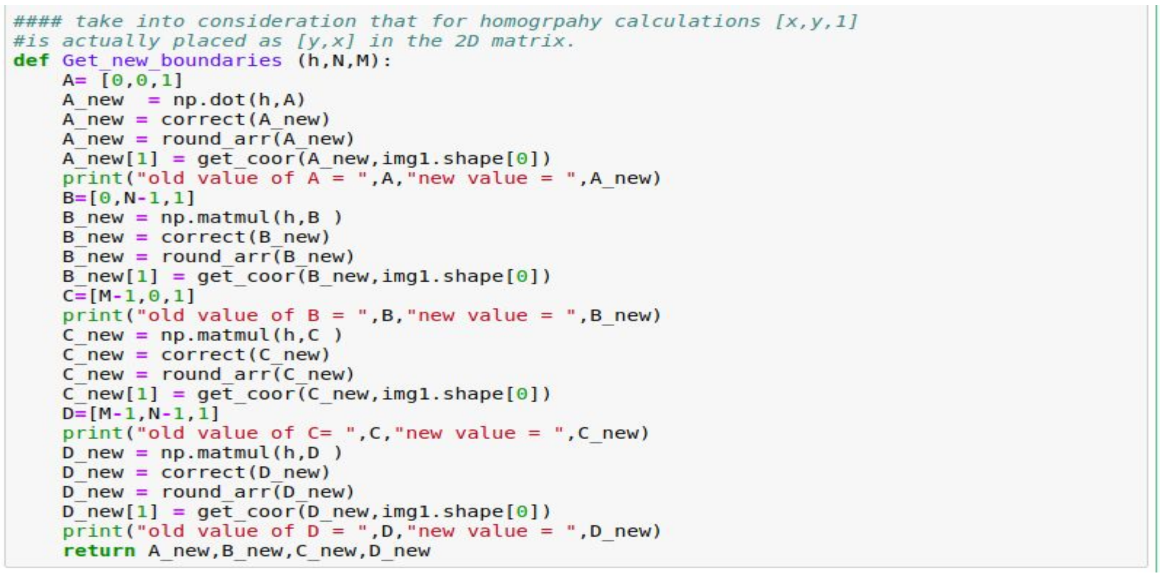
After obtaining the homography (h) we start getting into the warping process.

In our case, we consider the left most picture as the destination image and warp the rightmost image to that of the left, this can be repeated if multiple images are present. In this case we focus on just two images however.

<u>So the warping goes as follows:</u>

1) Determine where each corner of the src image will land after warping it into the dst image (4 corners in our case).

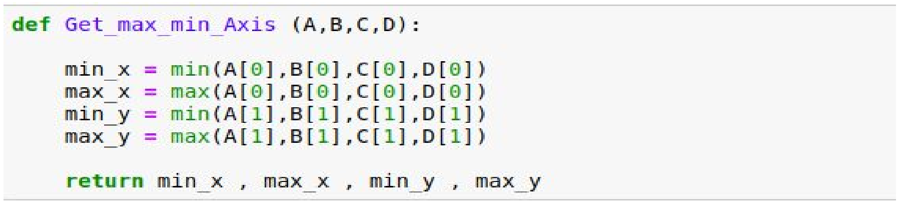
   This is known as **forward warping.**



```
#### take into consideration that for homogrpahy calculations [x,y,1]
#is actually placed as [y,x] in the 2D matrix.
def Get_new_boundaries (h,N,M):
    A= [0,0,1]
    A_new  = np.dot(h,A)
    A_new = correct(A_new)
    A_new = round_arr(A_new)
    A_new[1] = get_coor(A_new,img1.shape[0])
    print("old value of A = ",A,"new value = ",A_new)
    B=[0,N-1,1]
    B_new = np.matmul(h,B )
    B_new = correct(B_new)
    B_new = round_arr(B_new)
    B_new[1] = get_coor(B_new,img1.shape[0])
    C=[M-1,0,1]
    print("old value of B = ",B,"new value = ",B_new)
    C_new = np.matmul(h,C )
    C_new = correct(C_new)
    C_new = round_arr(C_new)
    C_new[1] = get_coor(C_new,img1.shape[0])
    D=[M-1,N-1,1]
    print("old value of C= ",C,"new value = ",C_new)
    D_new = np.matmul(h,D )
    D_new = correct(D_new)
    D_new = round_arr(D_new)
    D_new[1] = get_coor(D_new,img1.shape[0])
    print("old value of D = ",D,"new value = ",D_new)
    return A_new,B_new,C_new,D_new
```

2) Determine the size of the warped image, since the warped image won't be of a rectangular shape we convert it into one to be able to continue our calculations.

We achieve so by getting max_x,max_y,min_x,min_y and creating out new grid.*(where A,B,C,D are the coordinates of the corners of the warped image)*



```
def Get_max_min_Axis (A,B,C,D):

    min_x = min(A[0],B[0],C[0],D[0])
    max_x = max(A[0],B[0],C[0],D[0])
    min_y = min(A[1],B[1],C[1],D[1])
    max_y = max(A[1],B[1],C[1],D[1])

    return min_x , max_x , min_y , max_y
```

3) To avoid holes, we use inverse warping. Each pixel in the obtained bounding box will undergo a multiplication with the inverse of the homography. We then determine its actual position and acquire this intensity to be set as its intensity in the new grid.

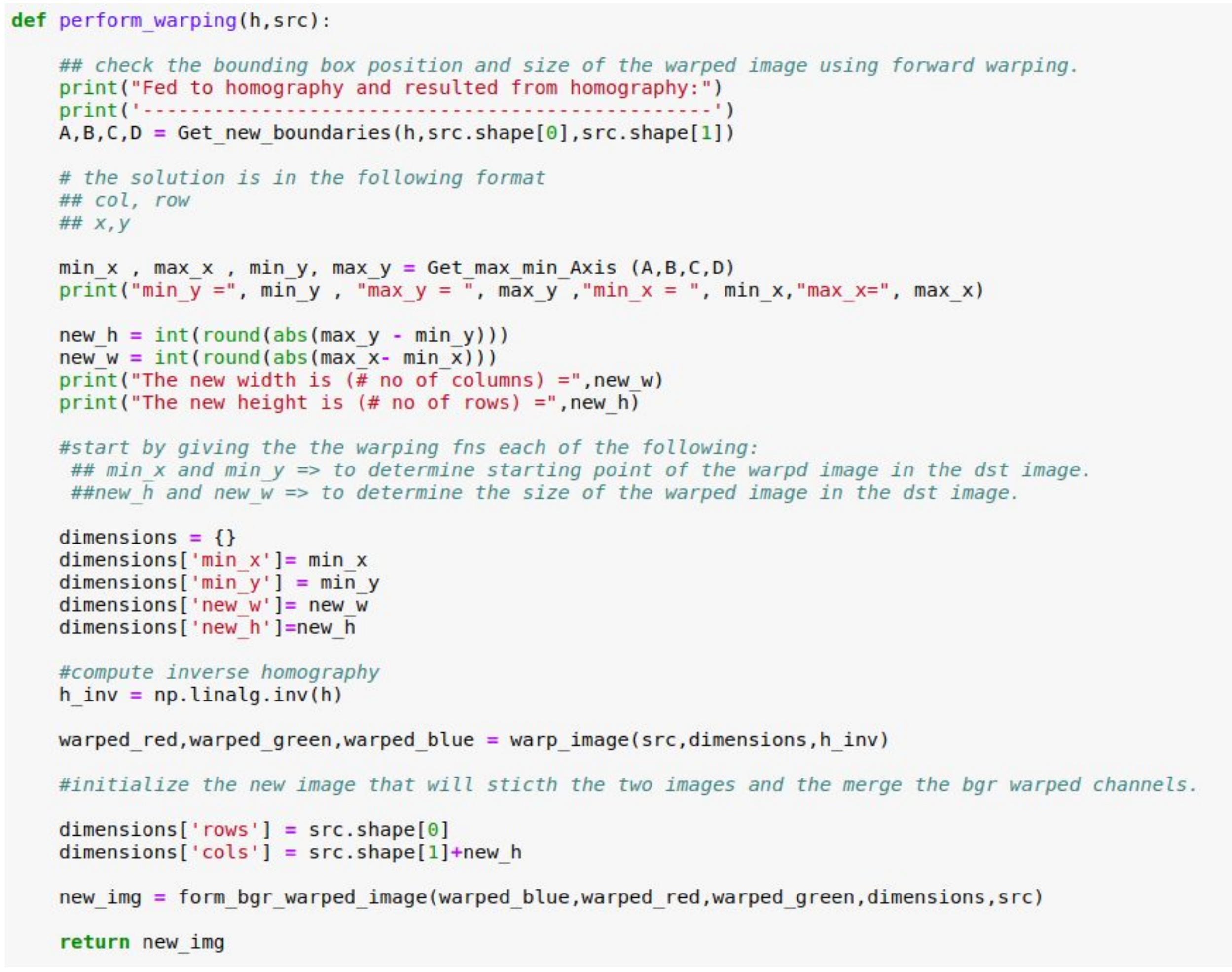
In case we landed between two pixels in the actual image, we use linear interpolation to determine the intensity.

Take into consideration that if the output from h_inverse * [x',y',1] resulted in negative coordinates, we set this pixel intensity to zero.

We repeat this process from each channel in case of having a 3 channel image, which is the case in all of our test images.

The explained procedure above is implemented in the perfrom_warping function, that takes the src image and the calculated homography and returns the new warped image in the destination image.

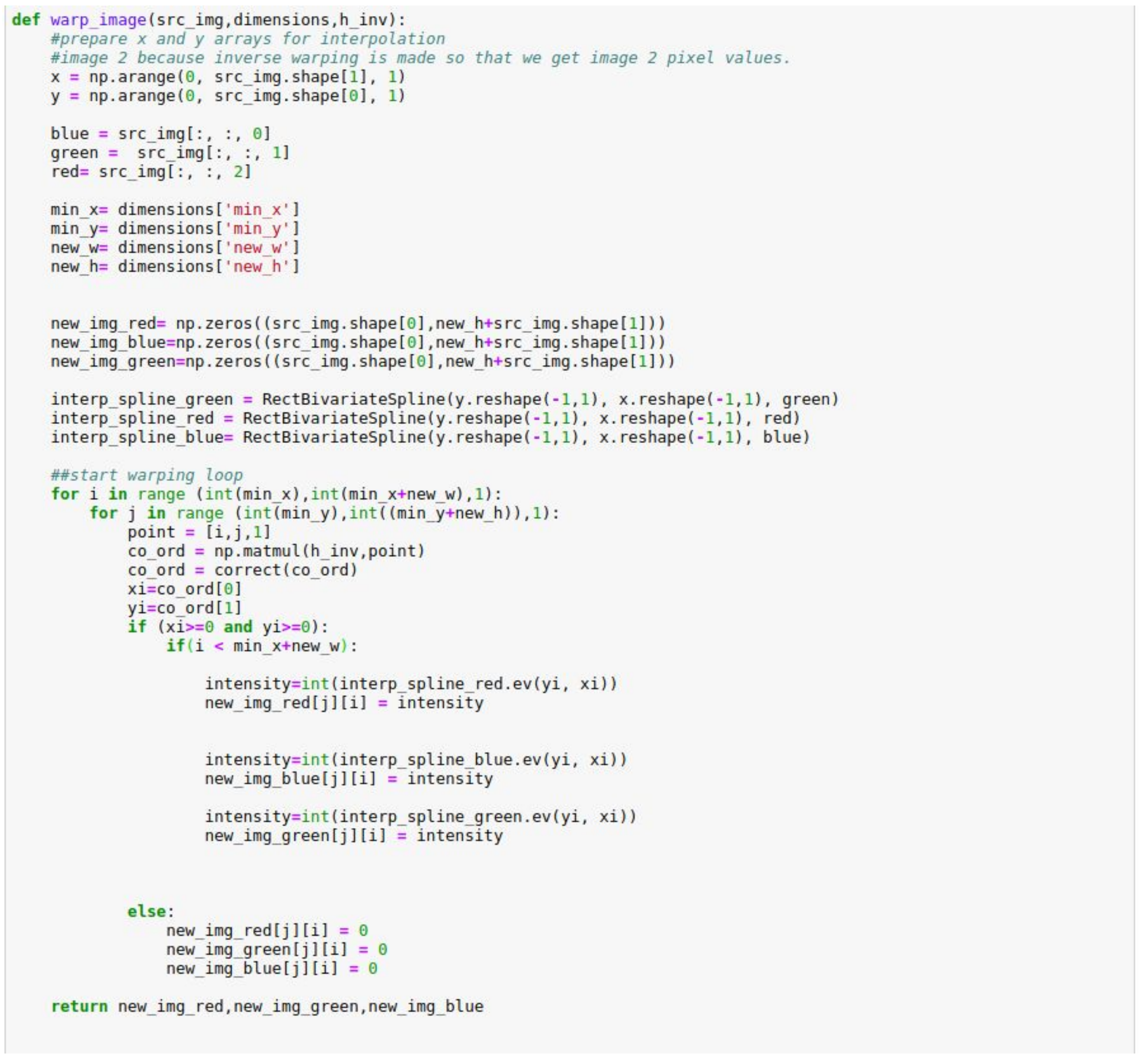
The code is shown in this figure.



```python
def perform_warping(h,src):

    ## check the bounding box position and size of the warped image using forward warping.
    print("Fed to homography and resulted from homography:")
    print('------------------------------------------------')
    A,B,C,D = Get_new_boundaries(h,src.shape[0],src.shape[1])

    # the solution is in the following format
    ## col, row
    ## x,y

    min_x , max_x , min_y, max_y = Get_max_min_Axis (A,B,C,D)
    print("min_y =", min_y , "max_y = ", max_y ,"min_x = ", min_x,"max_x=", max_x)

    new_h = int(round(abs(max_y - min_y)))
    new_w = int(round(abs(max_x- min_x)))
    print("The new width is (# no of columns) =",new_w)
    print("The new height is (# no of rows) =",new_h)

    #start by giving the the warping fns each of the following:
     ## min_x and min_y => to determine starting point of the warpd image in the dst image.
     ##new_h and new_w => to determine the size of the warped image in the dst image.

    dimensions = {}
    dimensions['min_x']= min_x
    dimensions['min_y'] = min_y
    dimensions['new_w']= new_w
    dimensions['new_h']=new_h

    #compute inverse homography
    h_inv = np.linalg.inv(h)

    warped_red,warped_green,warped_blue = warp_image(src,dimensions,h_inv)

    #initialize the new image that will sticth the two images and the merge the bgr warped channels.

    dimensions['rows'] = src.shape[0]
    dimensions['cols'] = src.shape[1]+new_h

    new_img = form_bgr_warped_image(warped_blue,warped_red,warped_green,dimensions,src)

    return new_img
```

The actual implementation of the warping however is in a separate function called *warp_image()* where interpolation and inverse warping take place.

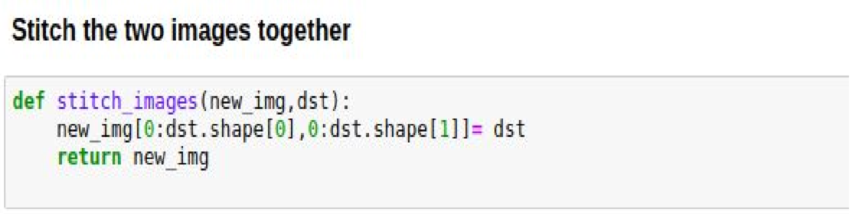
The code is shown in this figure.



```
def warp_image(src_img,dimensions,h_inv):
    #prepare x and y arrays for interpolation
    #image 2 because inverse warping is made so that we get image 2 pixel values.
    x = np.arange(0, src_img.shape[1], 1)
    y = np.arange(0, src_img.shape[0], 1)

    blue = src_img[:, :, 0]
    green =  src_img[:, :, 1]
    red= src_img[:, :, 2]

    min_x= dimensions['min_x']
    min_y= dimensions['min_y']
    new_w= dimensions['new_w']
    new_h= dimensions['new_h']


    new_img_red= np.zeros((src_img.shape[0],new_h+src_img.shape[1]))
    new_img_blue=np.zeros((src_img.shape[0],new_h+src_img.shape[1]))
    new_img_green=np.zeros((src_img.shape[0],new_h+src_img.shape[1]))

    interp_spline_green = RectBivariateSpline(y.reshape(-1,1), x.reshape(-1,1), green)
    interp_spline_red = RectBivariateSpline(y.reshape(-1,1), x.reshape(-1,1), red)
    interp_spline_blue= RectBivariateSpline(y.reshape(-1,1), x.reshape(-1,1), blue)

    ##start warping loop
    for i in range (int(min_x),int(min_x+new_w),1):
        for j in range (int(min_y),int((min_y+new_h)),1):
            point = [i,j,1]
            co_ord = np.matmul(h_inv,point)
            co_ord = correct(co_ord)
            xi=co_ord[0]
            yi=co_ord[1]
            if (xi>=0 and yi>=0):
                if(i < min_x+new_w):

                    intensity=int(interp_spline_red.ev(yi, xi))
                    new_img_red[j][i] = intensity


                    intensity=int(interp_spline_blue.ev(yi, xi))
                    new_img_blue[j][i] = intensity

                    intensity=int(interp_spline_green.ev(yi, xi))
                    new_img_green[j][i] = intensity



            else:
                new_img_red[j][i] = 0
                new_img_green[j][i] = 0
                new_img_blue[j][i] = 0

    return new_img_red,new_img_green,new_img_blue
```

4) Stitching:

We finally stitch the leftmost image with the warped image and return the resulting mosaic.

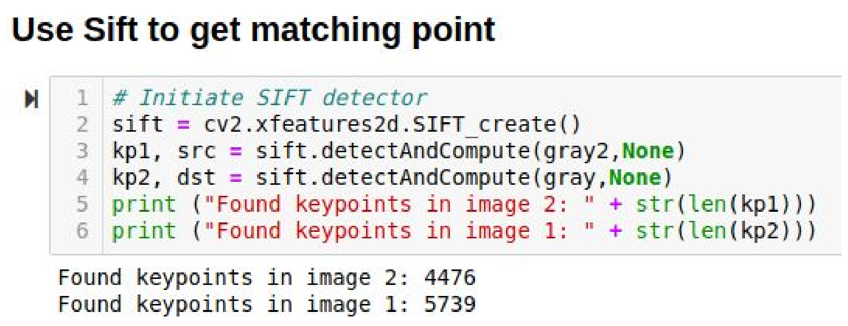


Stitch the two images together

```
def stitch_images(new_img,dst):
    new_img[0:dst.shape[0],0:dst.shape[1]]= dst
    return new_img
```

Repeat the warping using Sift instead of ginput:

For calculating the homography we initiate a sift detector and get the key points and descriptors in the source and destination images



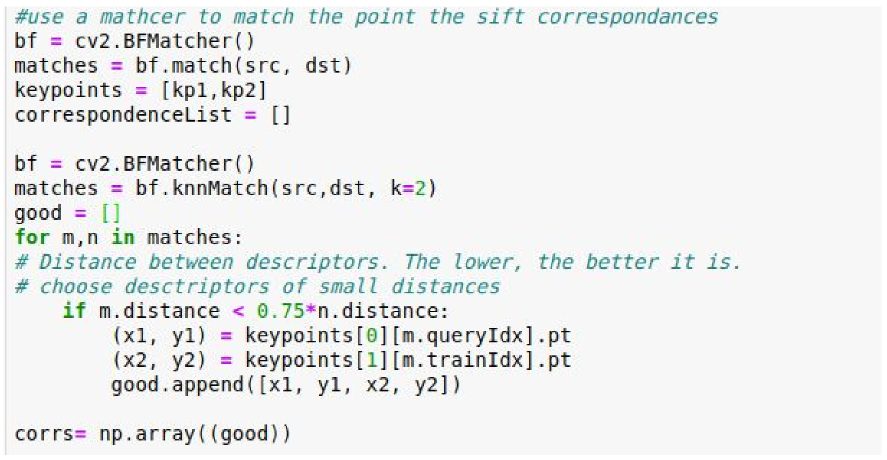
Use Sift to get matching point

```
# Initiate SIFT detector
sift = cv2.xfeatures2d.SIFT_create()
kp1, src = sift.detectAndCompute(gray2,None)
kp2, dst = sift.detectAndCompute(gray,None)
print ("Found keypoints in image 2: " + str(len(kp1)))
print ("Found keypoints in image 1: " + str(len(kp2)))
```

```
Found keypoints in image 2: 4476
Found keypoints in image 1: 5739
```

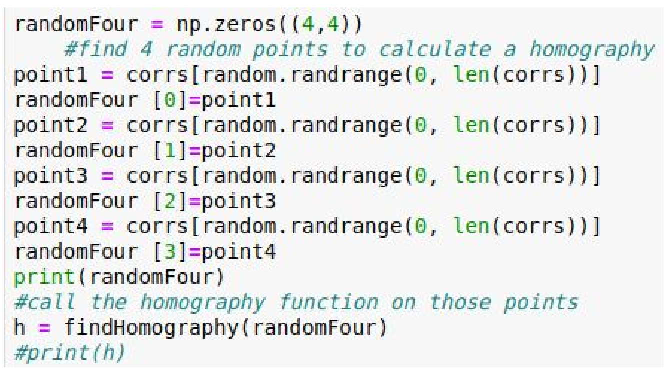
Then we match the the keypoints and descriptors of the source and destination points using Brute-Force Matching to the nearest 2 neighbours (k=2)

Then compute the good matches by applying the ratio test



```
#use a mathcer to match the point the sift correspondances
bf = cv2.BFMatcher()
matches = bf.match(src, dst)
keypoints = [kp1,kp2]
correspondenceList = []

bf = cv2.BFMatcher()
matches = bf.knnMatch(src,dst, k=2)
good = []
for m,n in matches:
# Distance between descriptors. The lower, the better it is.
# choose desctriptors of small distances
    if m.distance < 0.75*n.distance:
        (x1, y1) = keypoints[0][m.queryIdx].pt
        (x2, y2) = keypoints[1][m.trainIdx].pt
        good.append([x1, y1, x2, y2])

corrs= np.array((good))
```

Choose 4 random points from the good matches list and get the homography from them



```
randomFour = np.zeros((4,4))
    #find 4 random points to calculate a homography
point1 = corrs[random.randrange(0, len(corrs))]
randomFour [0]=point1
point2 = corrs[random.randrange(0, len(corrs))]
randomFour [1]=point2
point3 = corrs[random.randrange(0, len(corrs))]
randomFour [2]=point3
point4 = corrs[random.randrange(0, len(corrs))]
randomFour [3]=point4
print(randomFour)
#call the homography function on those points
h = findHomography(randomFour)
#print(h)
```

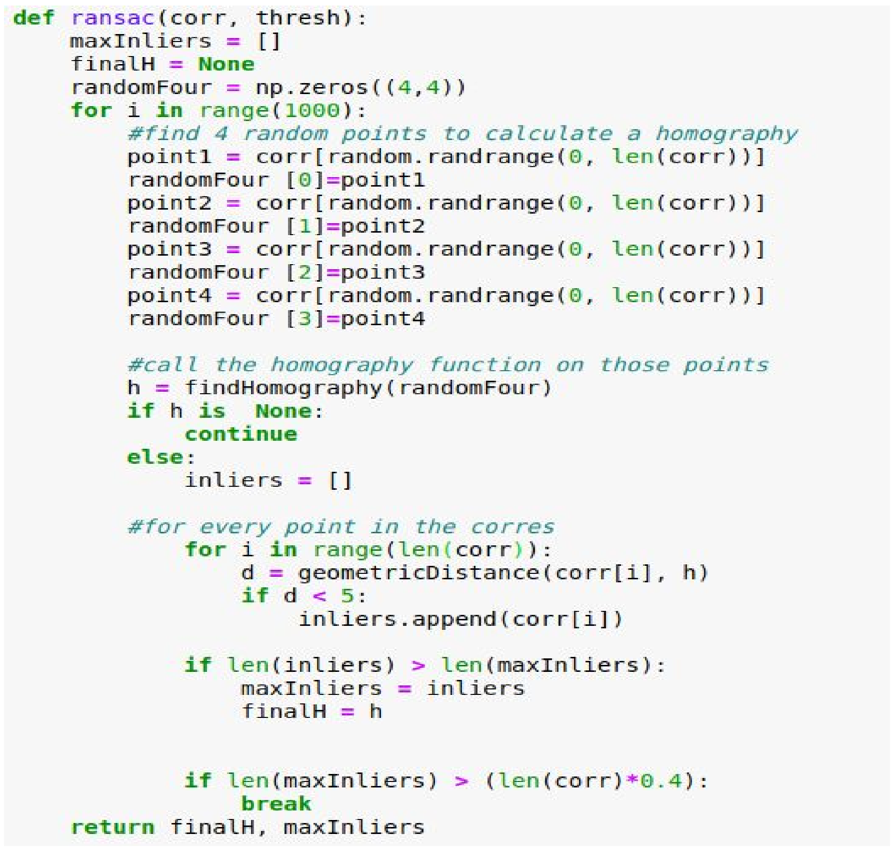
Repeat the warping using sift and ransac to improve the final outcome

Like in the previous step we use the sift detector to get the key points and descriptors then we use the brute force matcher to find the best match for every descriptor.

Then we apply the ransac algorithm:

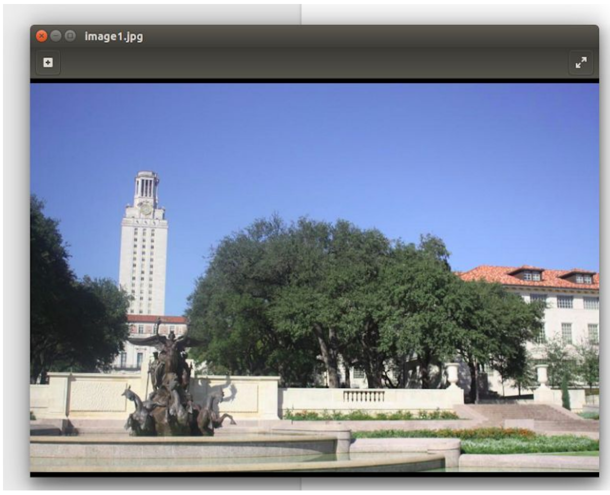
1) Choose 4 random points from the matches and get the homography according to them
2) According to the homography calculated we loop over every point in the matches and make forward warping to the source match to get its estimated location
3) If the distance between the estimated point and the real point is small we consider this match as an inlier
4) If the no of inliers > threshold then this homography is acceptable otherwise we got to point 1 and choose another 4 random points from the list of matches

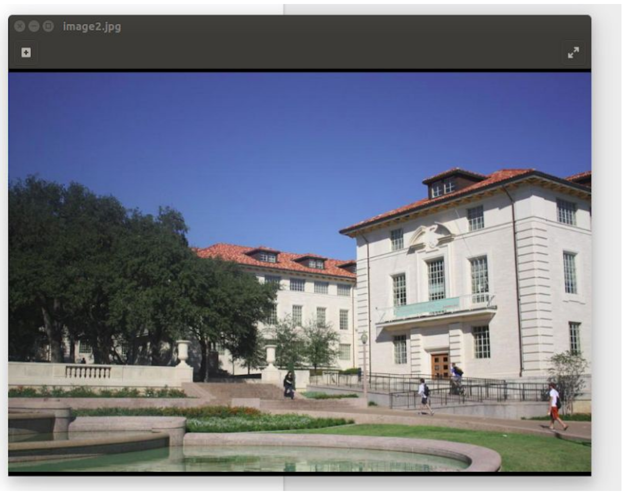
The code is shown in the following figure



```
def ransac(corr, thresh):
    maxInliers = []
    finalH = None
    randomFour = np.zeros((4,4))
    for i in range(1000):
        #find 4 random points to calculate a homography
        point1 = corr[random.randrange(0, len(corr))]
        randomFour [0]=point1
        point2 = corr[random.randrange(0, len(corr))]
        randomFour [1]=point2
        point3 = corr[random.randrange(0, len(corr))]
        randomFour [2]=point3
        point4 = corr[random.randrange(0, len(corr))]
        randomFour [3]=point4

        #call the homography function on those points
        h = findHomography(randomFour)
        if h is  None:
            continue
        else:
            inliers = []

        #for every point in the corres
            for i in range(len(corr)):
                d = geometricDistance(corr[i], h)
                if d < 5:
                    inliers.append(corr[i])

            if len(inliers) > len(maxInliers):
                maxInliers = inliers
                finalH = h


            if len(maxInliers) > (len(corr)*0.4):
                break
    return finalH, maxInliers
```

Sample tests

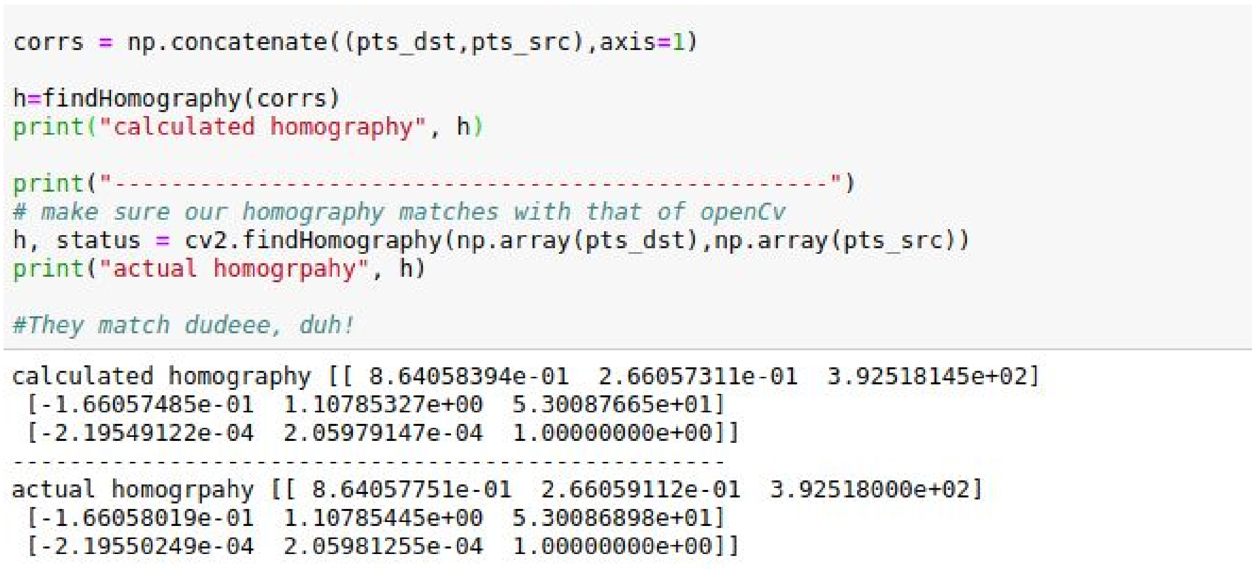




These two images will be used for testing

I. Testing homography function

We test our homography against OpenCv.findHomography function to make sure their results match.



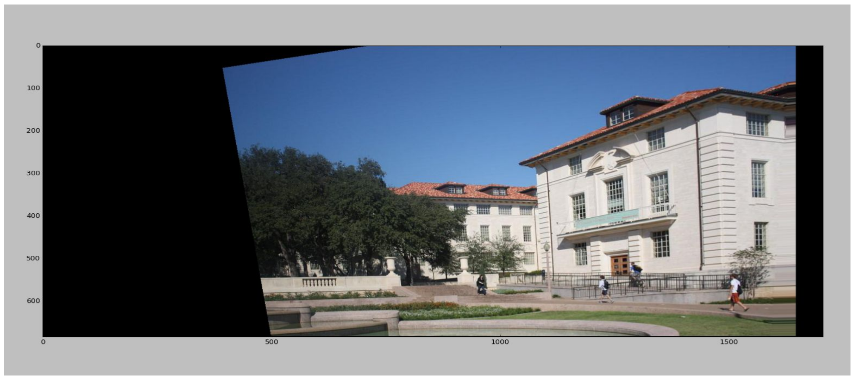
```
corrs = np.concatenate((pts_dst,pts_src),axis=1)

h=findHomography(corrs)
print("calculated homography", h)

print("--------------------------------------------------")
# make sure our homography matches with that of openCv
h, status = cv2.findHomography(np.array(pts_dst),np.array(pts_src))
print("actual homogrpahy", h)

#They match dudeee, duh!
```

```
calculated homography [[ 8.64058394e-01  2.66057311e-01  3.92518145e+02]
 [-1.66057485e-01  1.10785327e+00  5.30087665e+01]
 [-2.19549122e-04  2.05979147e-04  1.00000000e+00]]
--------------------------------------------------
actual homogrpahy [[ 8.64057751e-01  2.66059112e-01  3.92518000e+02]
 [-1.66058019e-01  1.10785445e+00  5.30086898e+01]
 [-2.19550249e-04  2.05981255e-04  1.00000000e+00]]
```

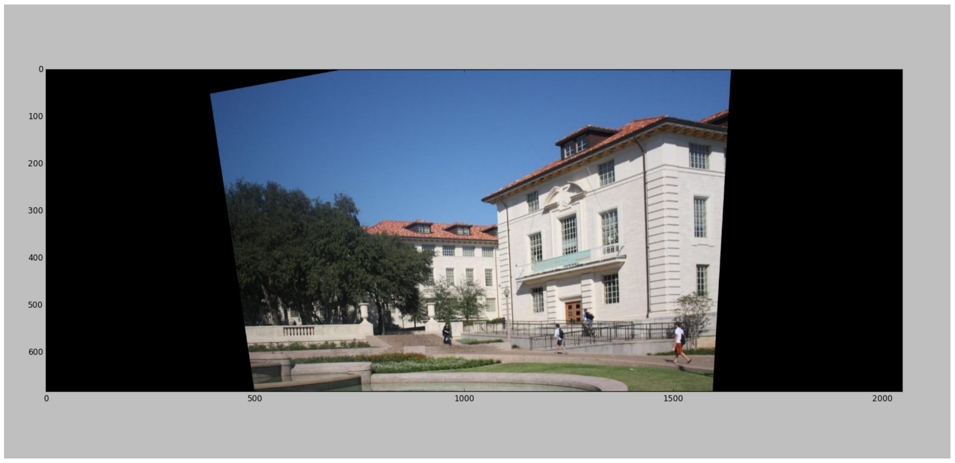
II. Warping

1) With points obtained from ginput()

We plot the resulting image from our warping function and compare it with openCv cv2.warpPerspective.

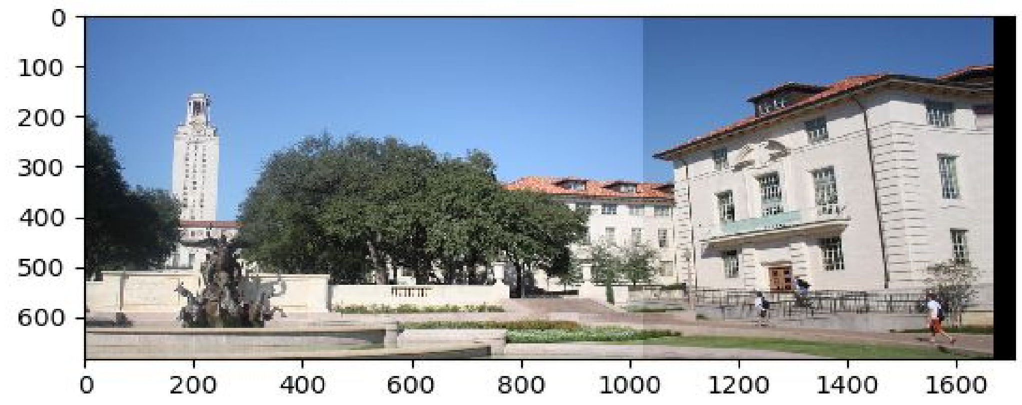


Implemented warped image

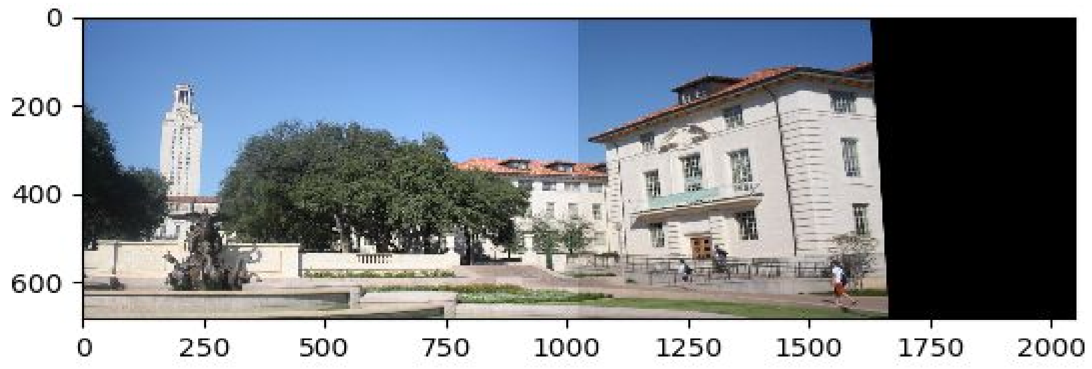


openCv warped image.

III. Stitching



Implemented stitching image

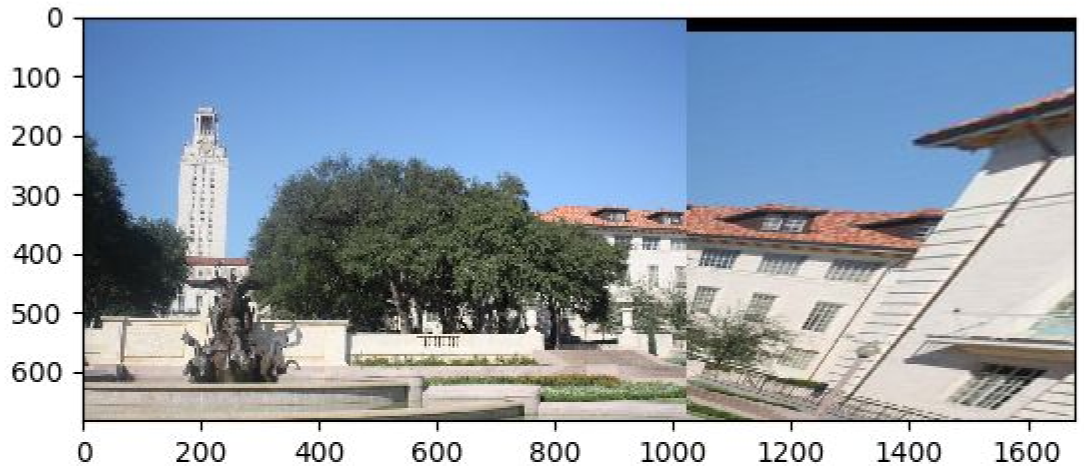


OpenCv stitched image

2) With points obtained from SIFT()

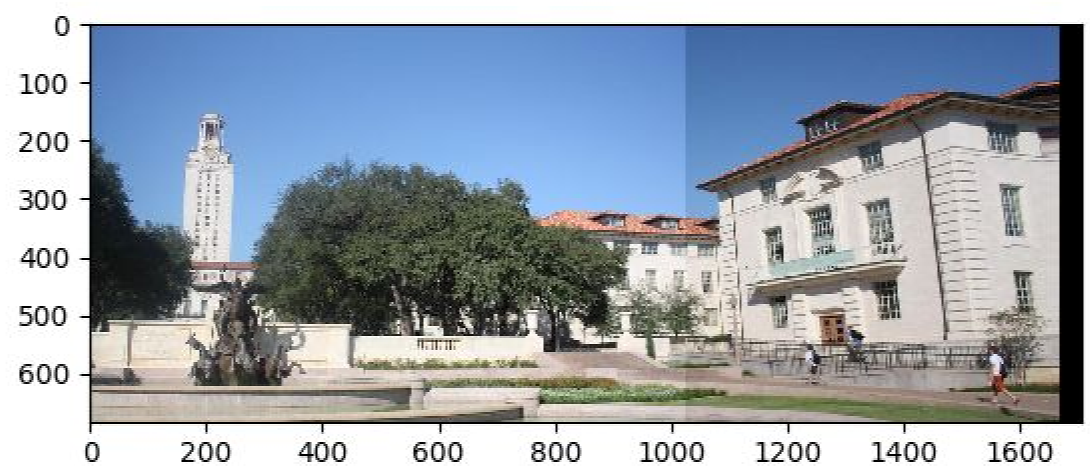
- Comparison between Sift before and after ransac

a) <u>Before ransac:</u>

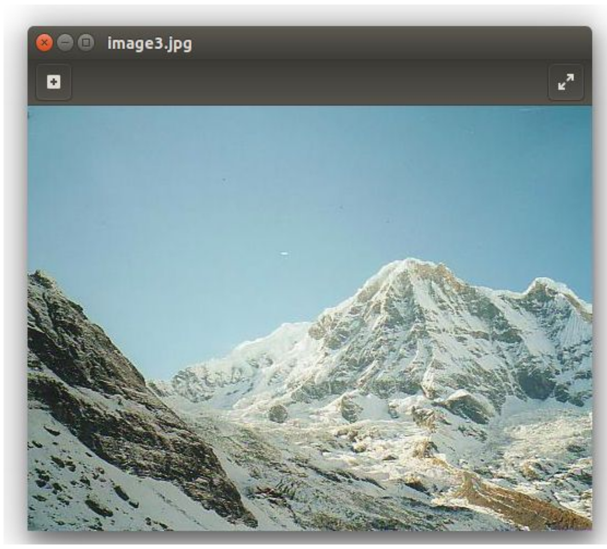


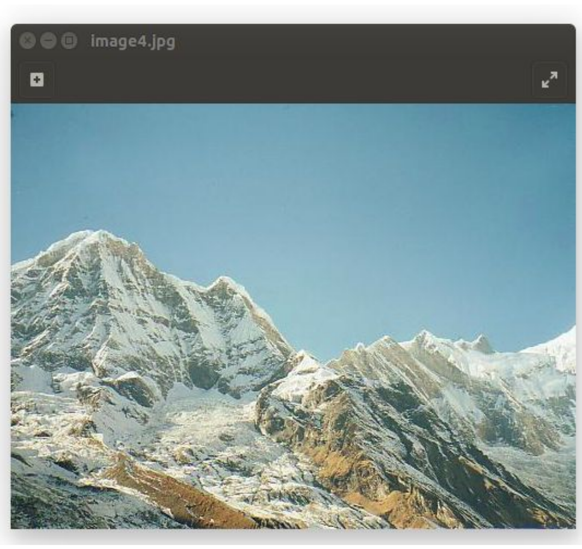
Test sample 2:

b) <u>After ransac:</u>

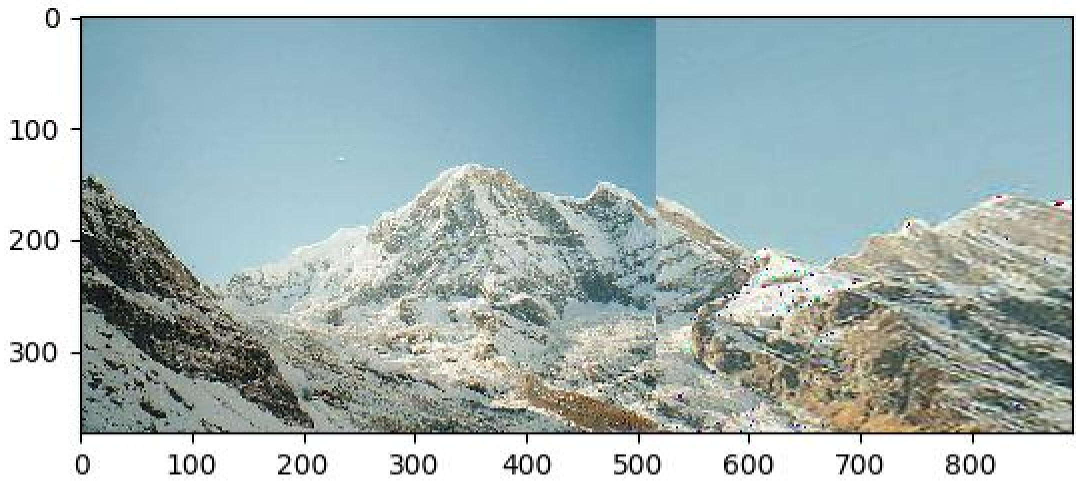


Test sample two:





Before Ransac:



After ransac: