# Computer Vision Assignment (2)

**Team members**

| | |
|---|---|
| **Nouran Hisham** | **6532** |
| **Nourhan Waleed** | **6609** |
| **Kareem Sabra** | **6594** |

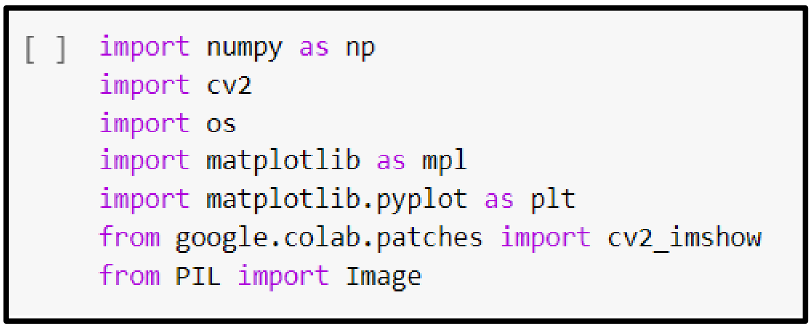
# Part 1

# Augmented Reality with Planar Homographies

## Problem statement:

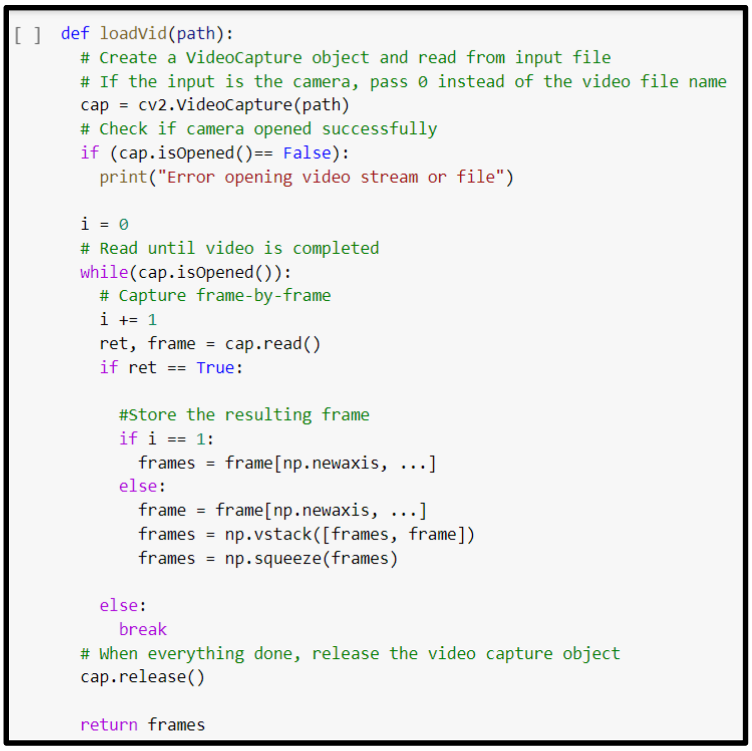
In this part of the assignment, we will be implementing an AR application step by step using planar homographies. We will first learn to find point correspondences between two images and use these to estimate the homography between them. Using this homography, we will then overlay images and finally implement our own AR application.

## Imports:

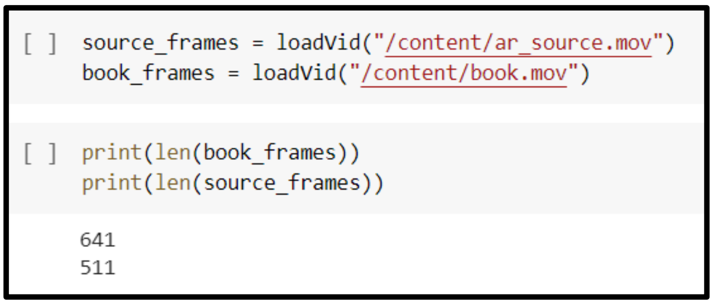


```
[ ]  import numpy as np
     import cv2
     import os
     import matplotlib as mpl
     import matplotlib.pyplot as plt
     from google.colab.patches import cv2_imshow
     from PIL import Image
```

## Loading videos and converting them into frames using the given .py file:



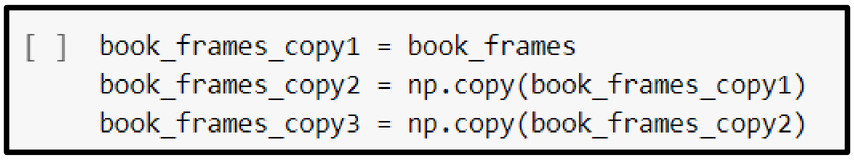
```
[ ]  def loadVid(path):
       # Create a VideoCapture object and read from input file
       # If the input is the camera, pass 0 instead of the video file name
       cap = cv2.VideoCapture(path)
       # Check if camera opened successfully
       if (cap.isOpened()== False):
         print("Error opening video stream or file")

       i = 0
       # Read until video is completed
       while(cap.isOpened()):
         # Capture frame-by-frame
         i += 1
         ret, frame = cap.read()
         if ret == True:

           #Store the resulting frame
           if i == 1:
             frames = frame[np.newaxis, ...]
           else:
             frame = frame[np.newaxis, ...]
             frames = np.vstack([frames, frame])
             frames = np.squeeze(frames)

         else:
           break
       # When everything done, release the video capture object
       cap.release()

       return frames
```



```
[ ] source_frames = loadVid("/content/ar_source.mov")
    book_frames = loadVid("/content/book.mov")

[ ] print(len(book_frames))
    print(len(source_frames))

    641
    511
```

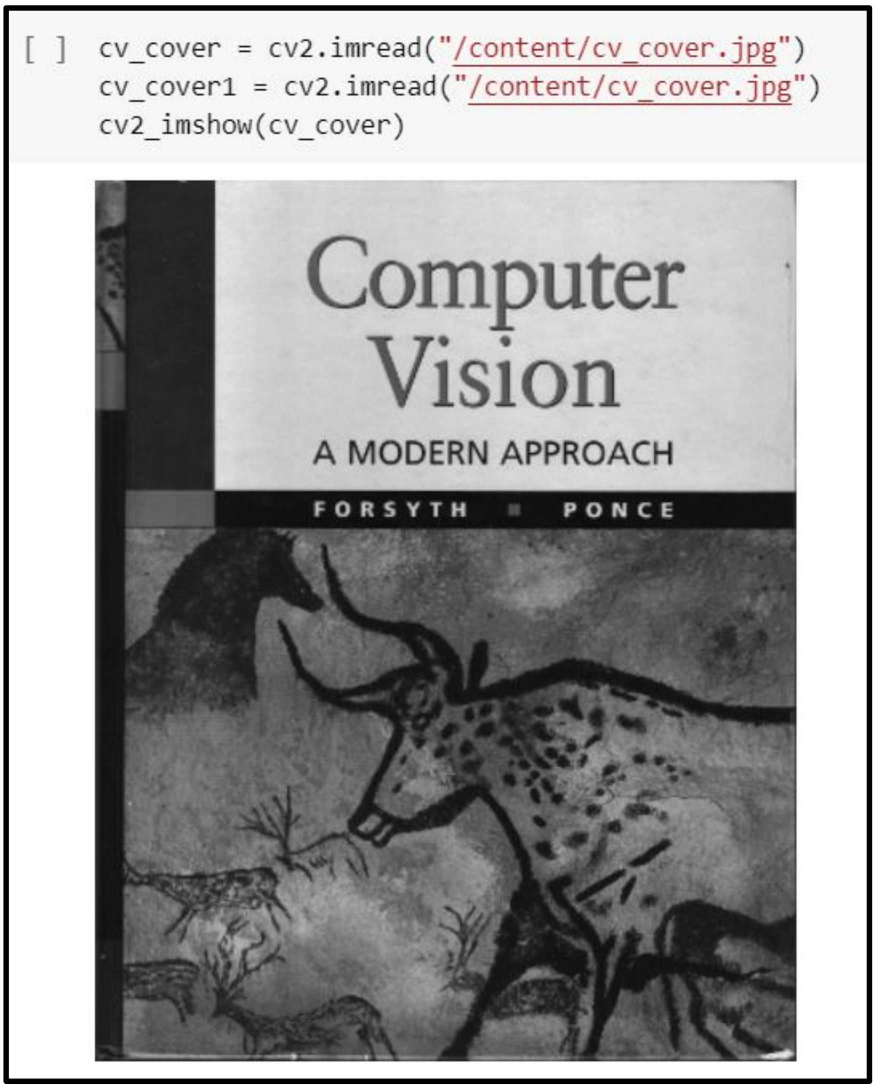
## Making copies of the book frames:

This is to always use the original as there are multiple processes done on them



```
[ ] book_frames_copy1 = book_frames
    book_frames_copy2 = np.copy(book_frames_copy1)
    book_frames_copy3 = np.copy(book_frames_copy2)
```

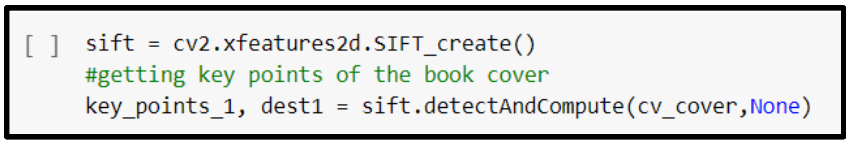
## Loading the book cover image:



```
[ ] cv_cover = cv2.imread("/content/cv_cover.jpg")
    cv_cover1 = cv2.imread("/content/cv_cover.jpg")
    cv2_imshow(cv_cover)
```

## 1.1 Getting Correspondences:

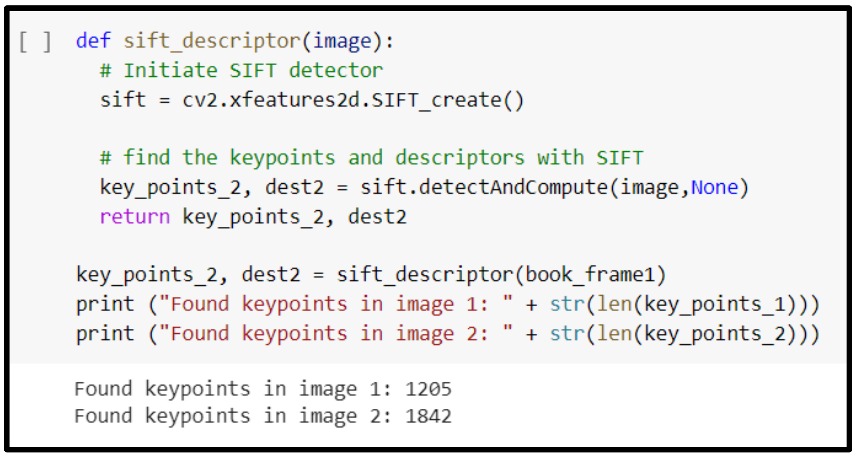
Initiating SIFT descriptor and getting key points of the book cover as they're going to be used with each book video frame



```
[ ]  sift = cv2.xfeatures2d.SIFT_create()
     #getting key points of the book cover
     key_points_1, dest1 = sift.detectAndCompute(cv_cover,None)
```

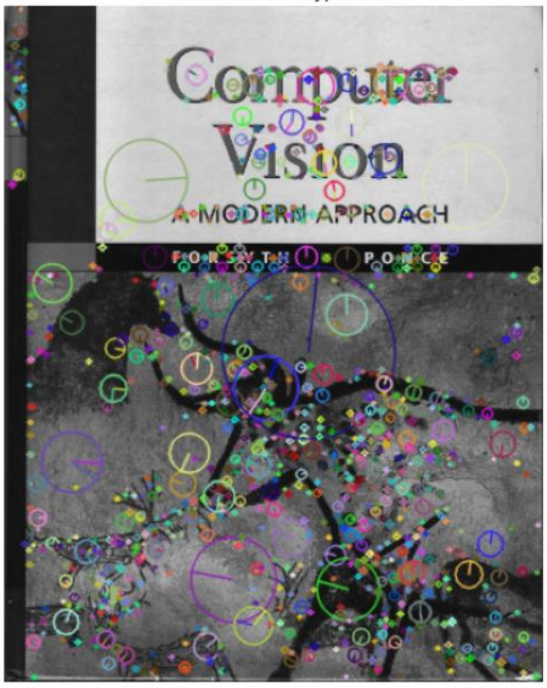
The first step is to find the correspondences between the book cover image and the first frame of the book video. We used SIFT descriptor from OpenCV library to find key points in each image.

A SIFT descriptor of a local region (key point) is a 3-D spatial histogram of the image gradients. The gradient at each pixel is regarded as a sample of a three-dimensional elementary feature vector, formed by the pixel location and the gradient orientation.

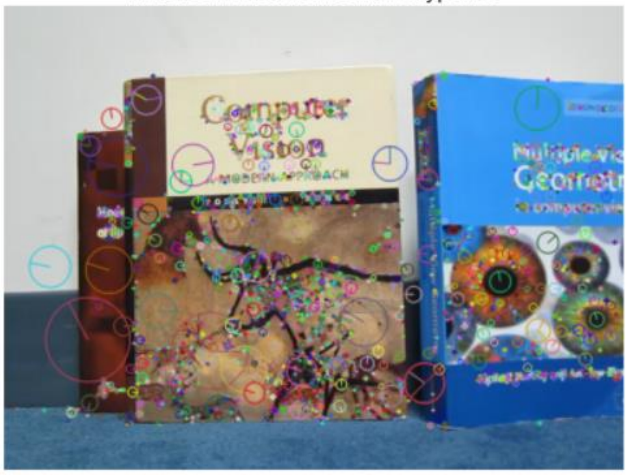


```
[ ]  def sift_descriptor(image):
       # Initiate SIFT detector
       sift = cv2.xfeatures2d.SIFT_create()

       # find the keypoints and descriptors with SIFT
       key_points_2, dest2 = sift.detectAndCompute(image,None)
       return key_points_2, dest2

     key_points_2, dest2 = sift_descriptor(book_frame1)
     print ("Found keypoints in image 1: " + str(len(key_points_1)))
     print ("Found keypoints in image 2: " + str(len(key_points_2)))

     Found keypoints in image 1: 1205
     Found keypoints in image 2: 1842
```

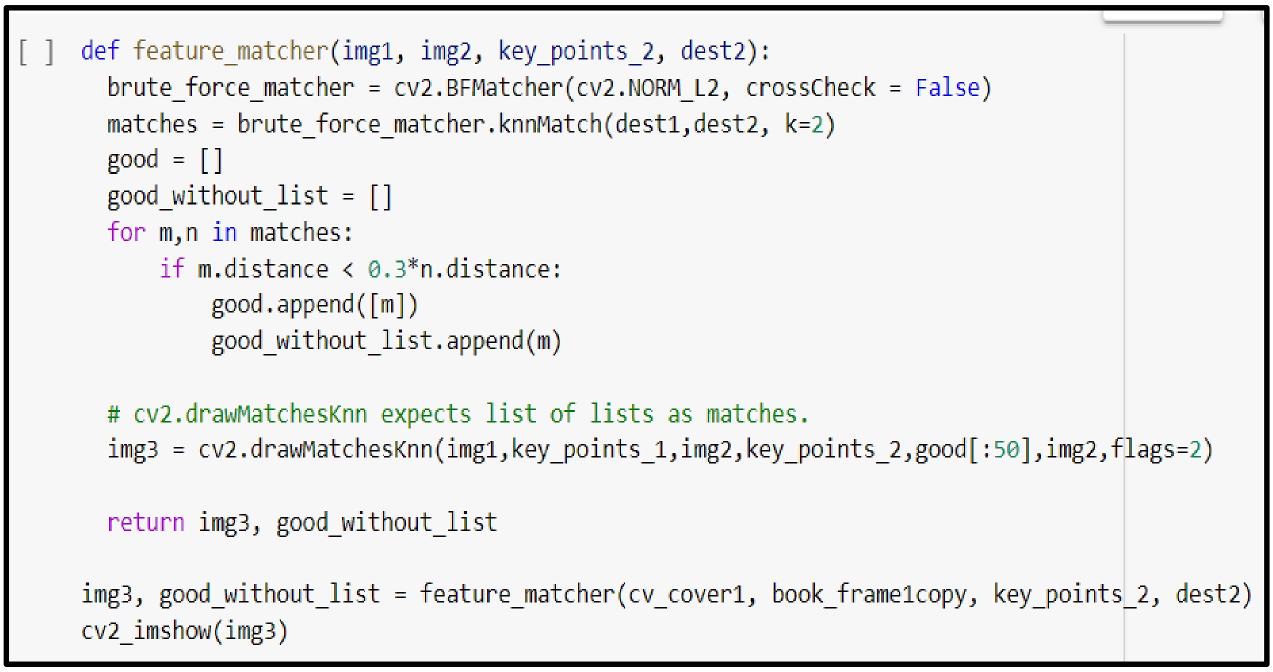
CV cover with keypoints



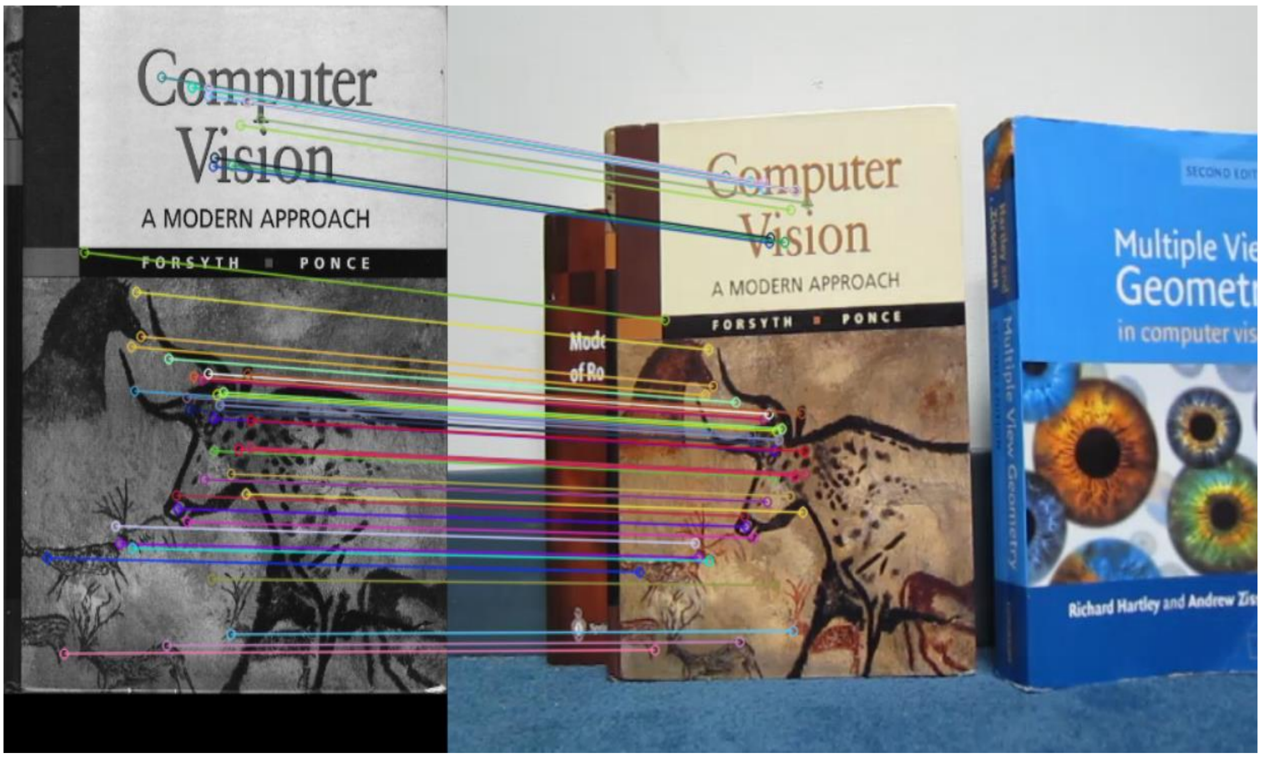
First book video frame with keypoints



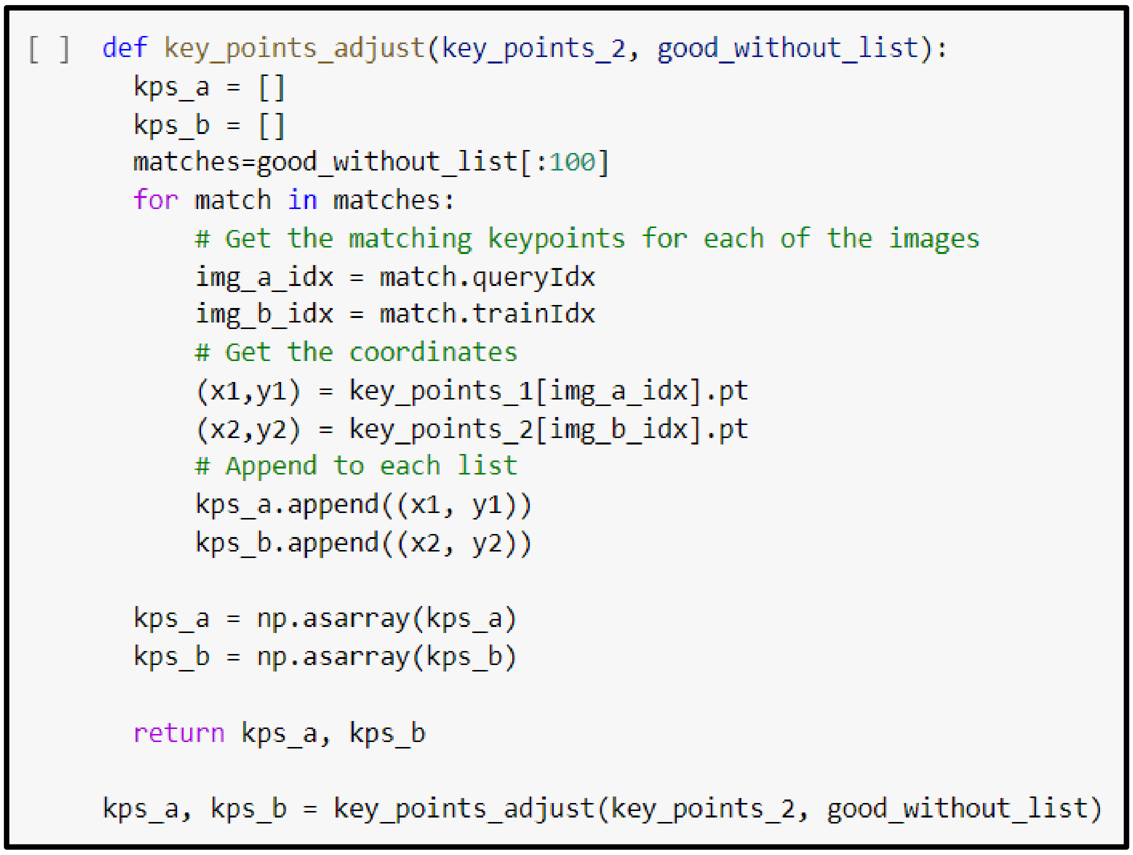
Then, we used the brute force matcher from OpenCV to get the correspondences. We used the matching way as KNN with size 2 and applied ratio checking of 0.3 between the best 2 matches to filter the good correspondences.



```
[ ]  def feature_matcher(img1, img2, key_points_2, dest2):
       brute_force_matcher = cv2.BFMatcher(cv2.NORM_L2, crossCheck = False)
       matches = brute_force_matcher.knnMatch(dest1,dest2, k=2)
       good = []
       good_without_list = []
       for m,n in matches:
           if m.distance < 0.3*n.distance:
               good.append([m])
               good_without_list.append(m)

       # cv2.drawMatchesKnn expects list of lists as matches.
       img3 = cv2.drawMatchesKnn(img1,key_points_1,img2,key_points_2,good[:50],img2,flags=2)

       return img3, good_without_list

     img3, good_without_list = feature_matcher(cv_cover1, book_frame1copy, key_points_2, dest2)
     cv2_imshow(img3)
```



## Adjusting the key points to calculate the homography matrix:



```
[ ]  def key_points_adjust(key_points_2, good_without_list):
       kps_a = []
       kps_b = []
       matches=good_without_list[:100]
       for match in matches:
           # Get the matching keypoints for each of the images
           img_a_idx = match.queryIdx
           img_b_idx = match.trainIdx
           # Get the coordinates
           (x1,y1) = key_points_1[img_a_idx].pt
           (x2,y2) = key_points_2[img_b_idx].pt
           # Append to each list
           kps_a.append((x1, y1))
           kps_b.append((x2, y2))

       kps_a = np.asarray(kps_a)
       kps_b = np.asarray(kps_b)

       return kps_a, kps_b

     kps_a, kps_b = key_points_adjust(key_points_2, good_without_list)
```

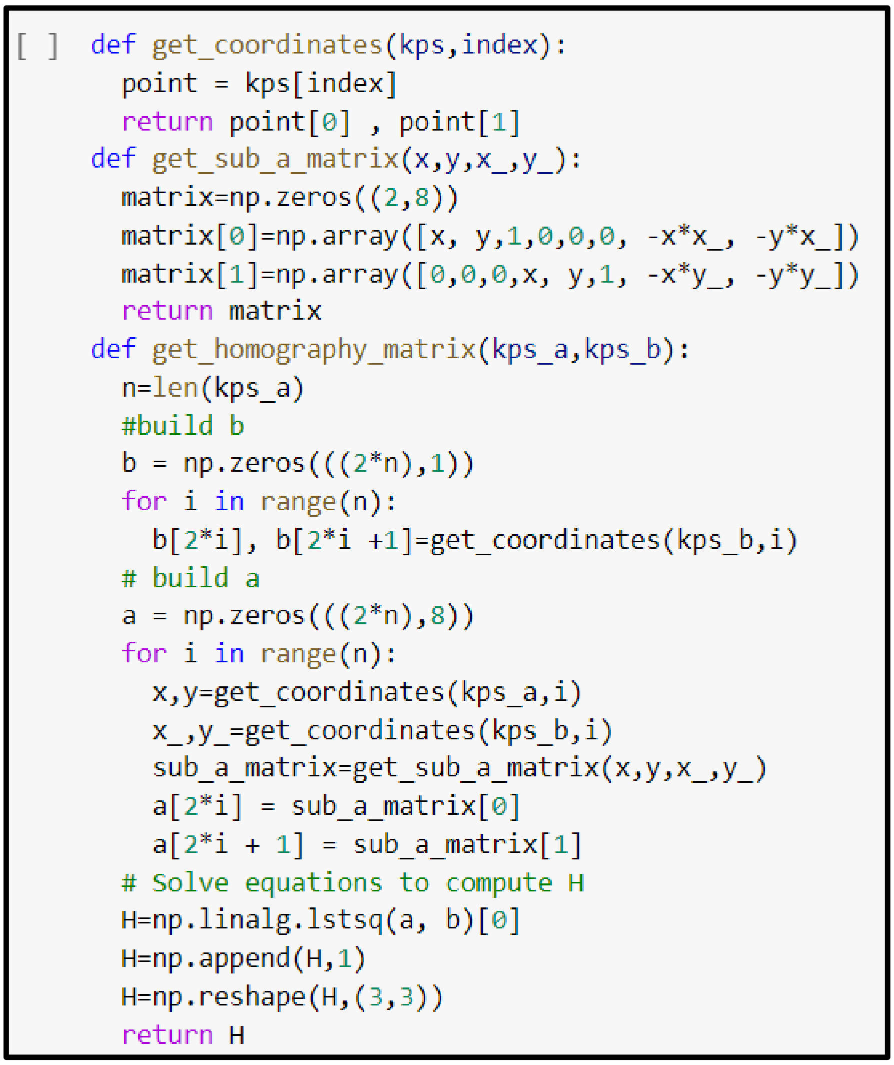
## 1.2 Compute the Homography Parameters

This function that takes a set of corresponding image points and computes the associated 3×3 homography matrix H. The homography matrix transforms any point p in one view to its corresponding homogeneous coordinates in the second view, p’, such that p’ = Hp. The function takes a list of n >= 4 pairs of corresponding points from the two views, where each point is specified with its 2D image coordinates.

$$
\begin{array}{r}
\\
\text{Point 1}\\
\\
\text{Point 2}\\
\\
\text{Point 3}\\
\\
\text{Point 4}\\
\\
\text{additional points}
\end{array}
\begin{array}{c}
\text{2N x 8}\\
\left[\begin{array}{cccccccc}
x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1\\
0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1\\
x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2\\
0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2\\
x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3\\
0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3\\
x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4\\
0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4
\end{array}\right]\\
\cdots
\end{array}
\begin{array}{c}
\text{8 x 1}\\
\left[\begin{array}{c}
h_{11}\\ h_{12}\\ h_{13}\\ h_{21}\\ h_{22}\\ h_{23}\\ h_{31}\\ h_{32}
\end{array}\right]\\
\\
\end{array}
=
\begin{array}{c}
\text{2N x 1}\\
\left[\begin{array}{c}
x'_1\\ y'_1\\ x'_2\\ y'_2\\ x'_3\\ y'_3\\ x'_4\\ y'_4
\end{array}\right]\\
\cdots
\end{array}
$$

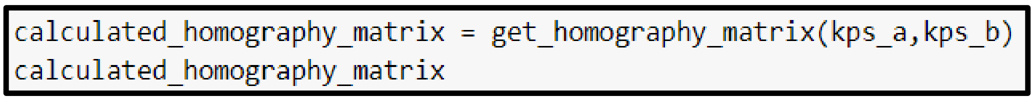
(source: Robert Collins, CSE486, Penn State)

We set up a solution using a system of linear equations $Ax = b$, where the 8 unknowns of H are stacked into an 8-vector x, the 2n-vector b contains image points from one view, and the $2n \times 8$ matrix A is filled appropriately so that the full system gives us $\lambda p = Hp$. There are only 8 unknowns in H because we set H3,3 = 1.

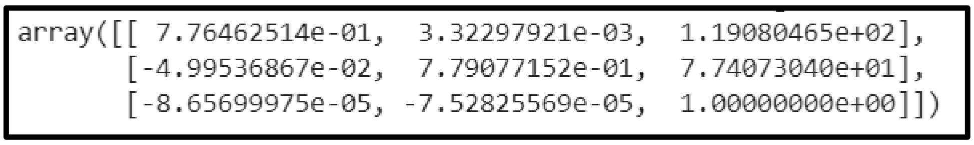


```
[ ]  def get_coordinates(kps,index):
       point = kps[index]
       return point[0] , point[1]
     def get_sub_a_matrix(x,y,x_,y_):
       matrix=np.zeros((2,8))
       matrix[0]=np.array([x, y,1,0,0,0, -x*x_, -y*x_])
       matrix[1]=np.array([0,0,0,x, y,1, -x*y_, -y*y_])
       return matrix
     def get_homography_matrix(kps_a,kps_b):
       n=len(kps_a)
       #build b
       b = np.zeros(((2*n),1))
       for i in range(n):
         b[2*i], b[2*i +1]=get_coordinates(kps_b,i)
       # build a
       a = np.zeros(((2*n),8))
       for i in range(n):
         x,y=get_coordinates(kps_a,i)
         x_,y_=get_coordinates(kps_b,i)
         sub_a_matrix=get_sub_a_matrix(x,y,x_,y_)
         a[2*i] = sub_a_matrix[0]
         a[2*i + 1] = sub_a_matrix[1]
       # Solve equations to compute H
       H=np.linalg.lstsq(a, b)[0]
       H=np.append(H,1)
       H=np.reshape(H,(3,3))
       return H
```

## <u>The homography matrix using the function we wrote:</u>

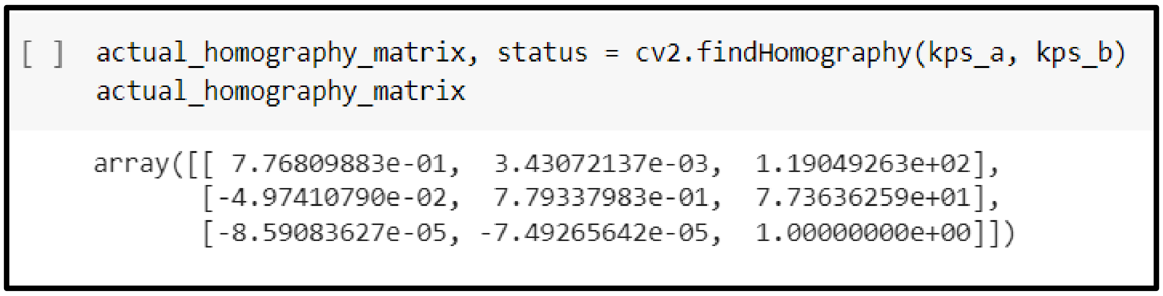


```
calculated_homography_matrix = get_homography_matrix(kps_a,kps_b)
calculated_homography_matrix
```



```
array([[ 7.76462514e-01,  3.32297921e-03,  1.19080465e+02],
       [-4.99536867e-02,  7.79077152e-01,  7.74073040e+01],
       [-8.65699975e-05, -7.52825569e-05,  1.00000000e+00]])
```

To verify, we checked the homography matrix using OpenCV built-in function and it was approximately equal to ours

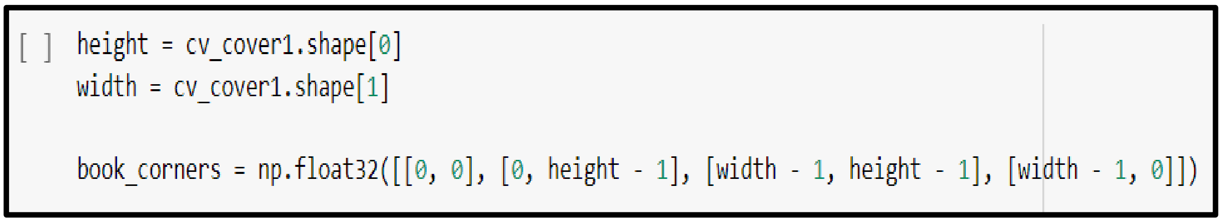


```
[ ]  actual_homography_matrix, status = cv2.findHomography(kps_a, kps_b)
     actual_homography_matrix
```

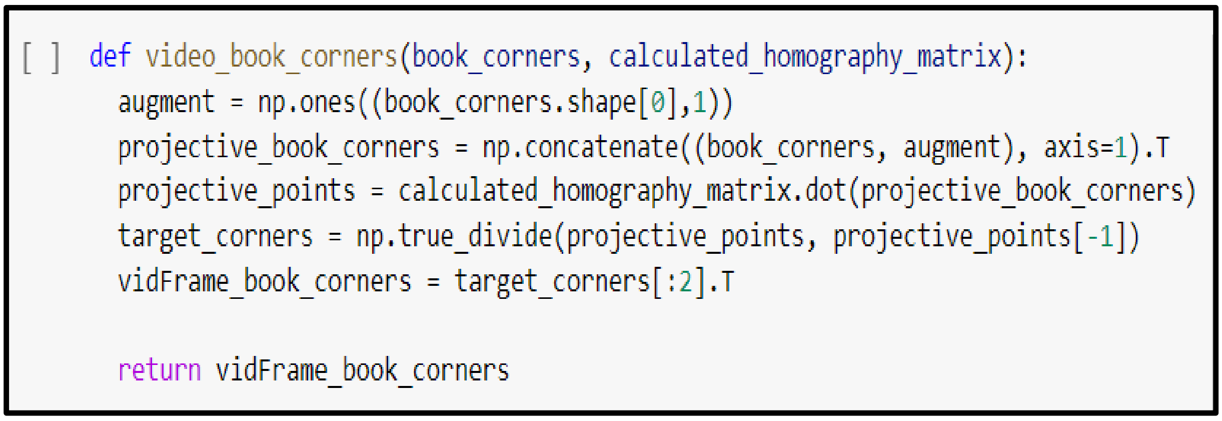
```
array([[ 7.76809883e-01,  3.43072137e-03,  1.19049263e+02],
       [-4.97410790e-02,  7.79337983e-01,  7.73636259e+01],
       [-8.59083627e-05, -7.49265642e-05,  1.00000000e+00]])
```

## 1.3 Calculate Book Coordinates:

First, we need to get the coordinates of the book cover corners to be able to map them to the book video frames using the calculated homography matrix

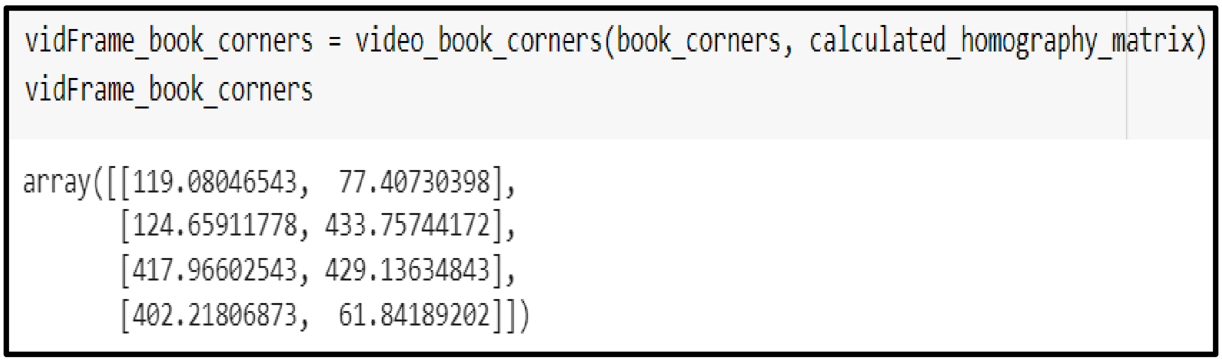


```
[ ]  height = cv_cover1.shape[0]
     width = cv_cover1.shape[1]

     book_corners = np.float32([[0, 0], [0, height - 1], [width - 1, height - 1], [width - 1, 0]])
```



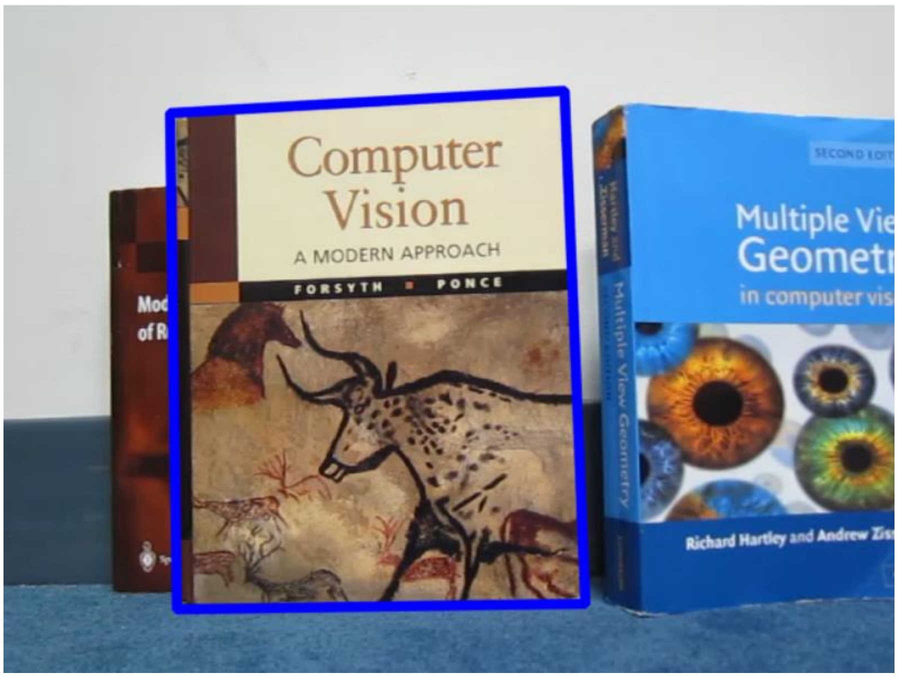
```
[ ]  def video_book_corners(book_corners, calculated_homography_matrix):
       augment = np.ones((book_corners.shape[0],1))
       projective_book_corners = np.concatenate((book_corners, augment), axis=1).T
       projective_points = calculated_homography_matrix.dot(projective_book_corners)
       target_corners = np.true_divide(projective_points, projective_points[-1])
       vidFrame_book_corners = target_corners[:2].T

       return vidFrame_book_corners
```

## The mapped coordinates of the book corners in the first book video frame:



```
vidFrame_book_corners = video_book_corners(book_corners, calculated_homography_matrix)
vidFrame_book_corners
```

```
array([[119.08046543,  77.40730398],
       [124.65911778, 433.75744172],
       [417.96602543, 429.13634843],
       [402.21806873,  61.84189202]])
```

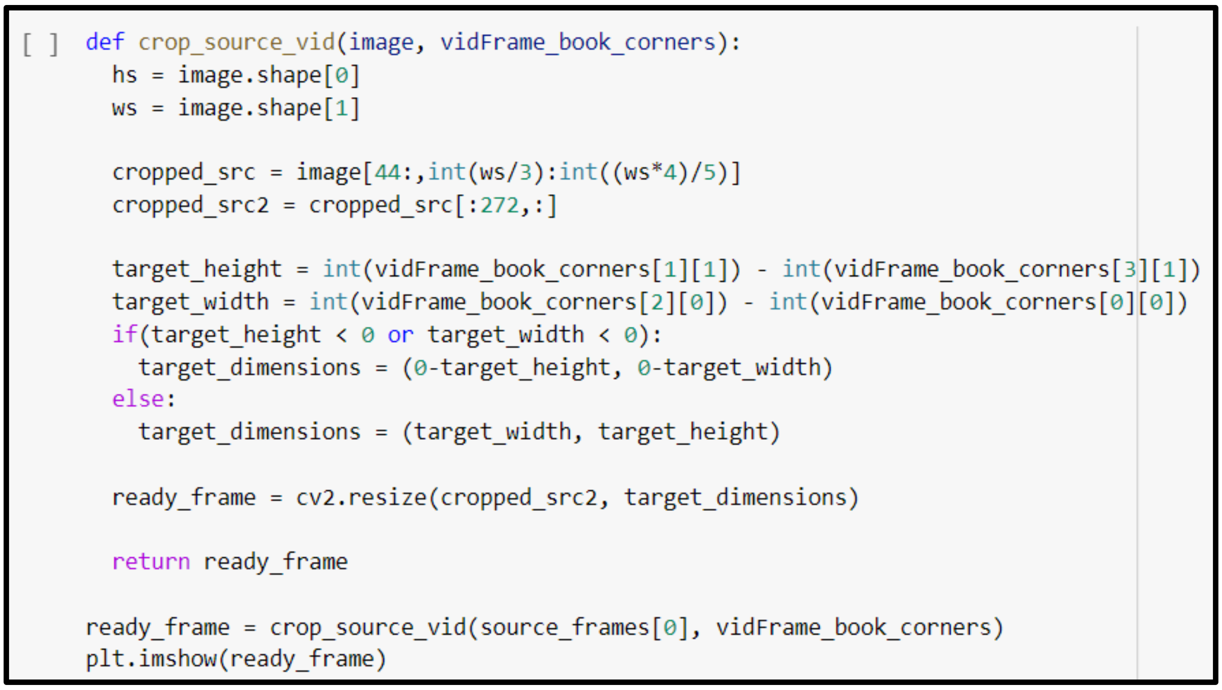


## 1.4 <u>Crop AR Video Frames:</u>

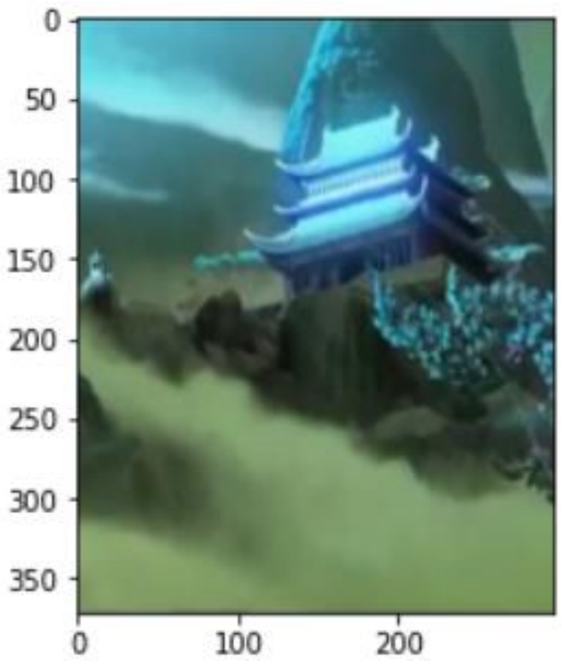
First, we need to crop only the middle part of the source video to be able to overlay it on our book cover, this was achieved by cropping out the first 1/3 width and the last 1/5 width of the video. Additionally, we cropped out the top and bottom 44 black pixels.

Then, we calculated the needed dimensions of the book in the other video using its mapped corners coordinates to be able to resize the source video frame as close to the book video frame as possible.

At some frames, the dimensions yield negative results, so we subtract them from 0 to get the positive value.



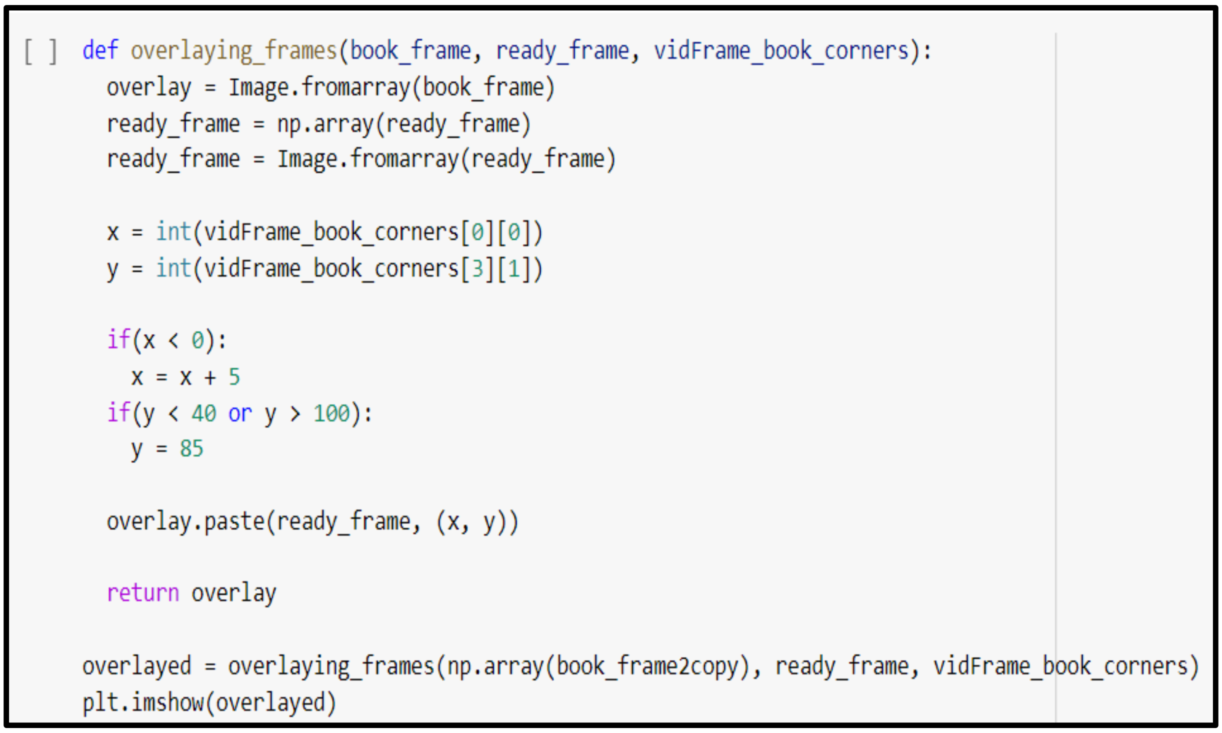
```
[ ] def crop_source_vid(image, vidFrame_book_corners):
      hs = image.shape[0]
      ws = image.shape[1]

      cropped_src = image[44:,int(ws/3):int((ws*4)/5)]
      cropped_src2 = cropped_src[:272,:]

      target_height = int(vidFrame_book_corners[1][1]) - int(vidFrame_book_corners[3][1])
      target_width = int(vidFrame_book_corners[2][0]) - int(vidFrame_book_corners[0][0])
      if(target_height < 0 or target_width < 0):
        target_dimensions = (0-target_height, 0-target_width)
      else:
        target_dimensions = (target_width, target_height)

      ready_frame = cv2.resize(cropped_src2, target_dimensions)

      return ready_frame

    ready_frame = crop_source_vid(source_frames[0], vidFrame_book_corners)
    plt.imshow(ready_frame)
```



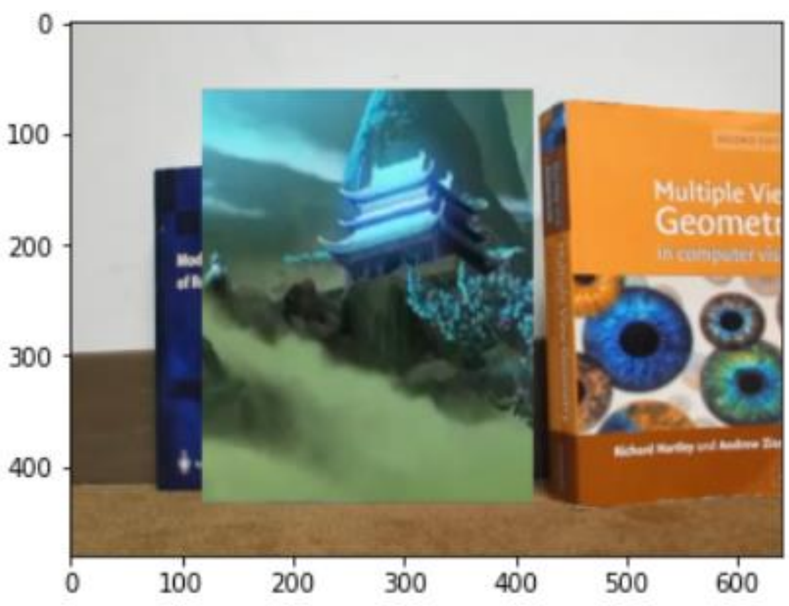
## 1.5 Overlay the First Frame of the Two Videos:

Now, we need to overlay the cropped source video frame onto the cv book in the book video frame which was achieved by placing the cropped source video frame at the top left corner and top right corner of the book in the book video frame.

We noticed an abnormal behavior at the last video frames, so we handled them by adding a certain default value to the x-coordinate and equating the y-coordinate to another default value that is mostly repeated which was our way to improve the output video as much as possible.

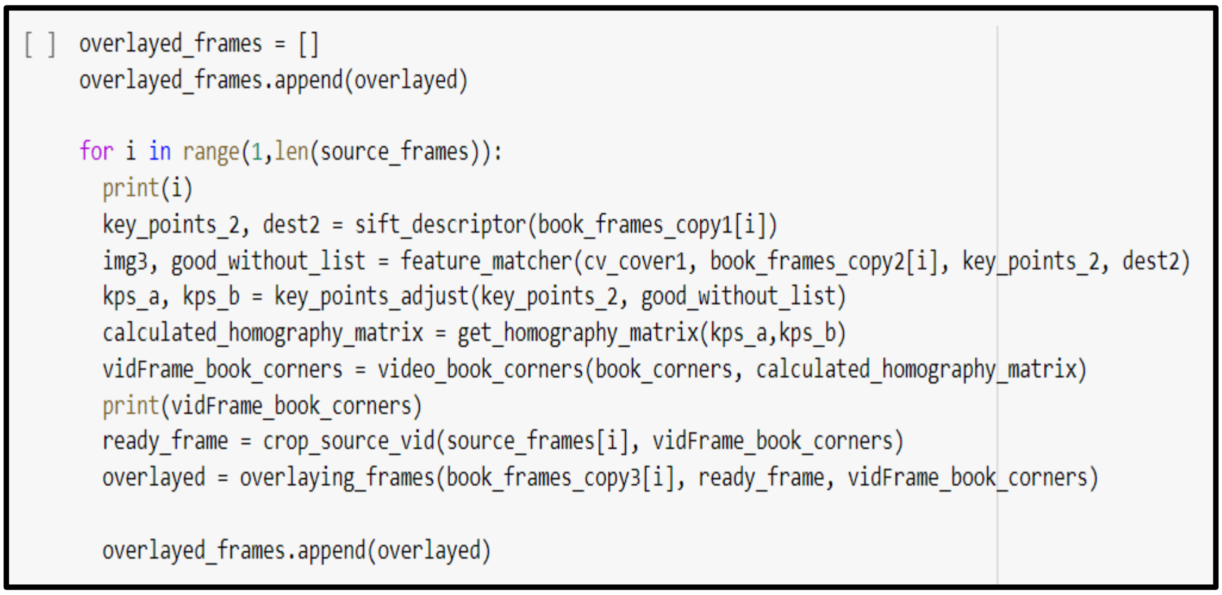


```
[ ] def overlaying_frames(book_frame, ready_frame, vidFrame_book_corners):
      overlay = Image.fromarray(book_frame)
      ready_frame = np.array(ready_frame)
      ready_frame = Image.fromarray(ready_frame)

      x = int(vidFrame_book_corners[0][0])
      y = int(vidFrame_book_corners[3][1])

      if(x < 0):
        x = x + 5
      if(y < 40 or y > 100):
        y = 85

      overlay.paste(ready_frame, (x, y))

      return overlay

    overlayed = overlaying_frames(np.array(book_frame2copy), ready_frame, vidFrame_book_corners)
    plt.imshow(overlayed)
```



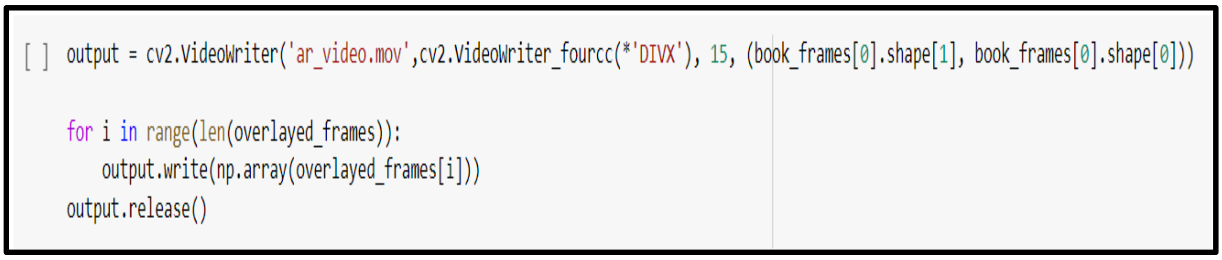
## 1.6 Creating AR Application:

Now, all we need to do is repeat all those previous steps on all the remaining video frames by looping around them and calling the previous functions in the exact same order. We calculated the new homography matrix between each frame and the book cover.



```
[ ] overlayed_frames = []
    overlayed_frames.append(overlayed)

    for i in range(1,len(source_frames)):
      print(i)
      key_points_2, dest2 = sift_descriptor(book_frames_copy1[i])
      img3, good_without_list = feature_matcher(cv_cover1, book_frames_copy2[i], key_points_2, dest2)
      kps_a, kps_b = key_points_adjust(key_points_2, good_without_list)
      calculated_homography_matrix = get_homography_matrix(kps_a,kps_b)
      vidFrame_book_corners = video_book_corners(book_corners, calculated_homography_matrix)
      print(vidFrame_book_corners)
      ready_frame = crop_source_vid(source_frames[i], vidFrame_book_corners)
      overlayed = overlaying_frames(book_frames_copy3[i], ready_frame, vidFrame_book_corners)

      overlayed_frames.append(overlayed)
```

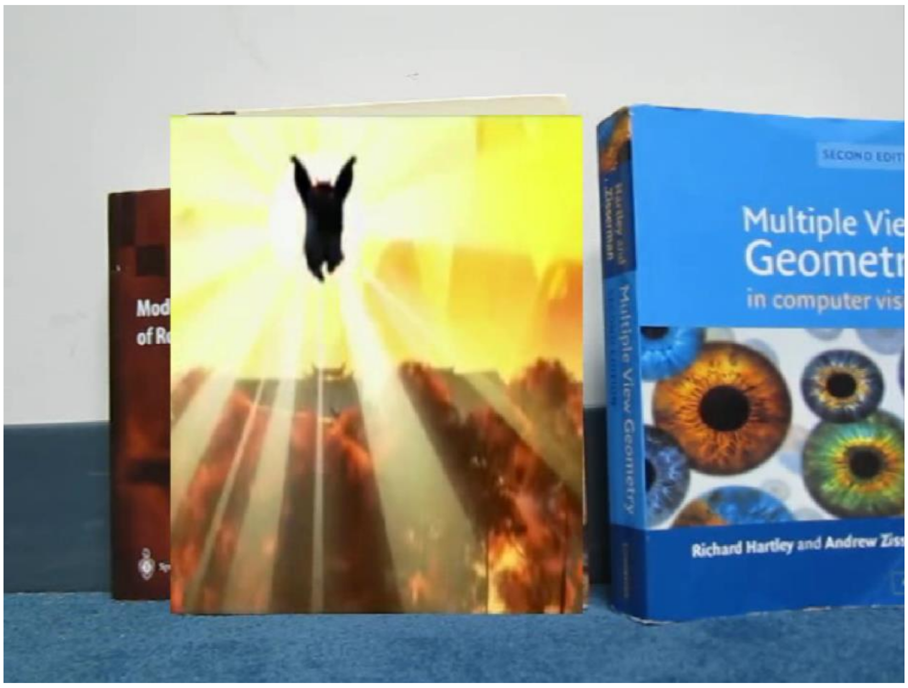
## Converting all the overlayed frames into a video:



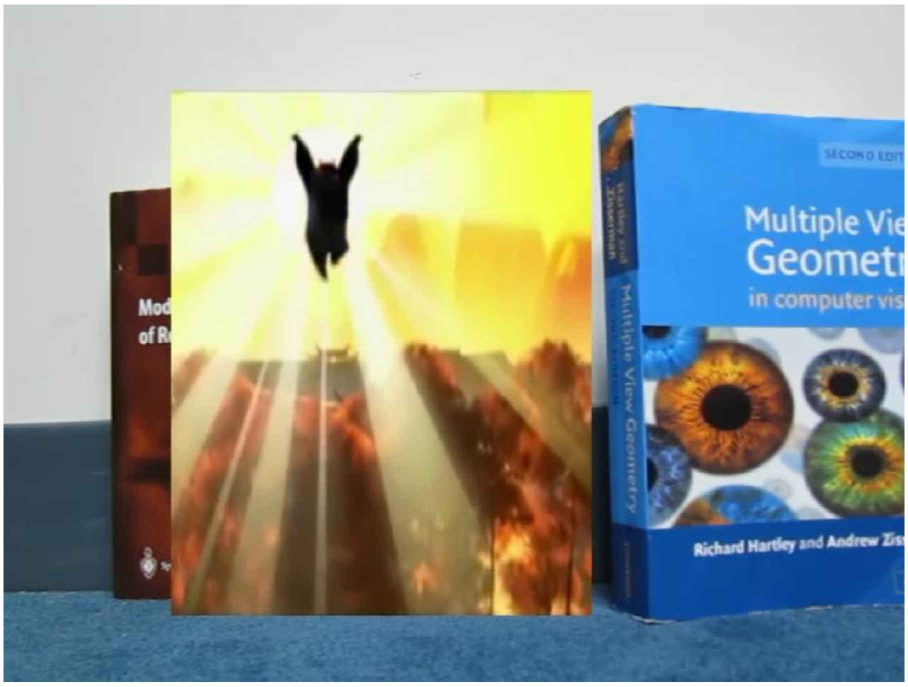
```
[ ] output = cv2.VideoWriter('ar_video.mov',cv2.VideoWriter_fourcc(*'DIVX'), 15, (book_frames[0].shape[1], book_frames[0].shape[0]))

    for i in range(len(overlayed_frames)):
        output.write(np.array(overlayed_frames[i]))
    output.release()
```

## Screenshots of some video frames:

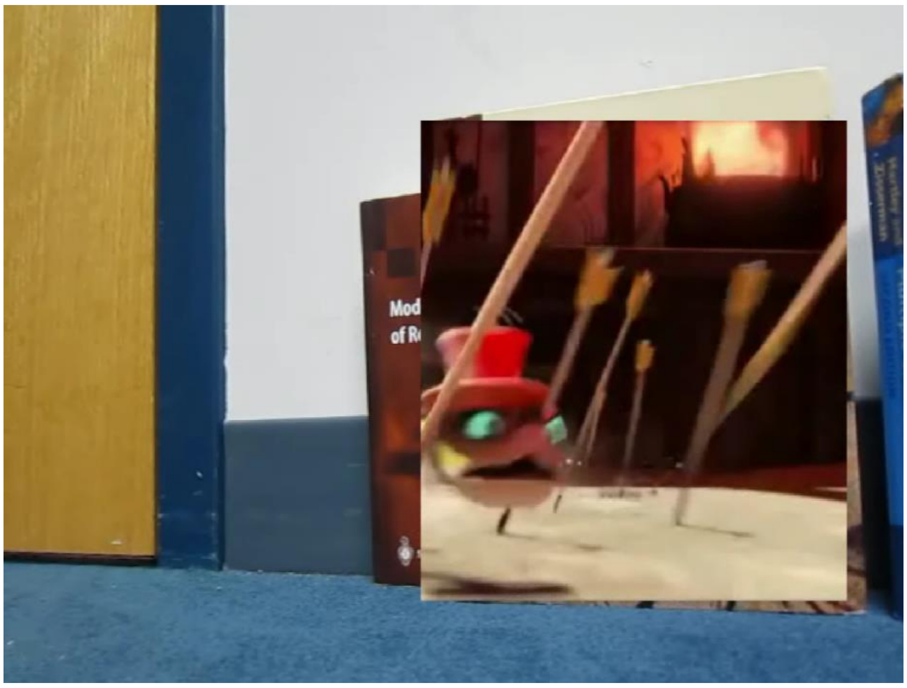
## Before improvement:



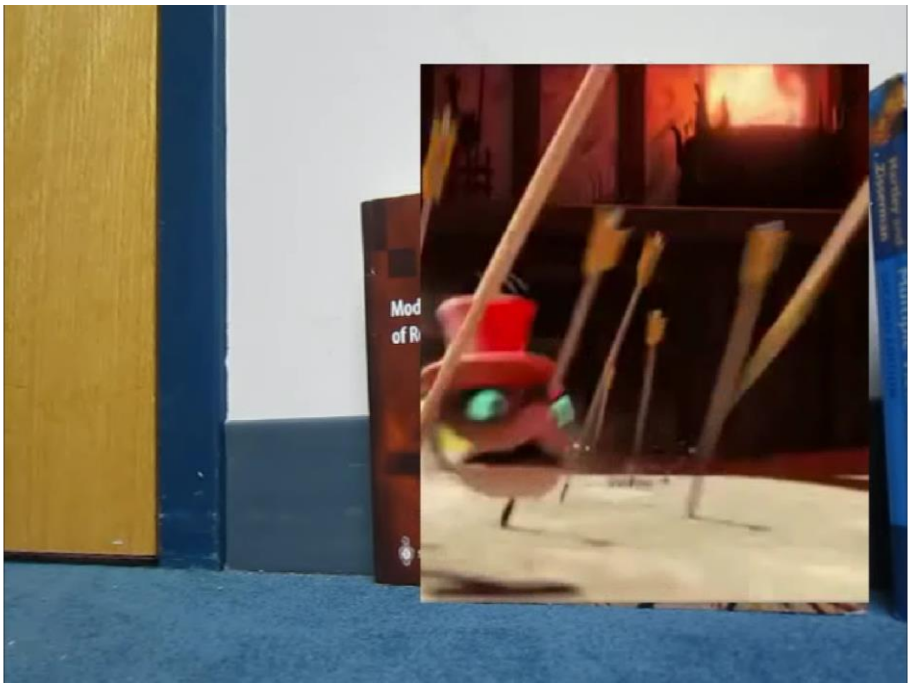
## After improvement:



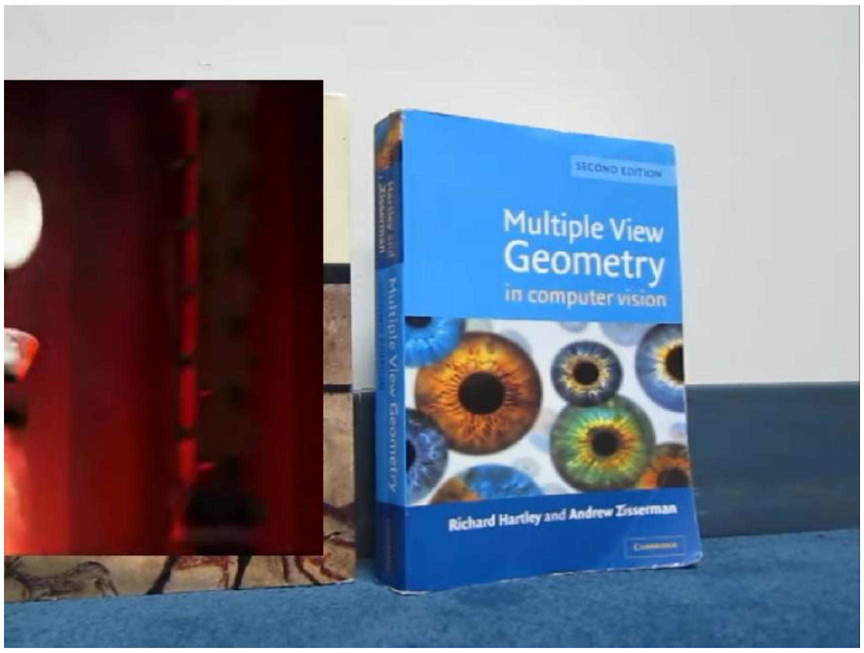
## <u>**Before improvement:**</u>



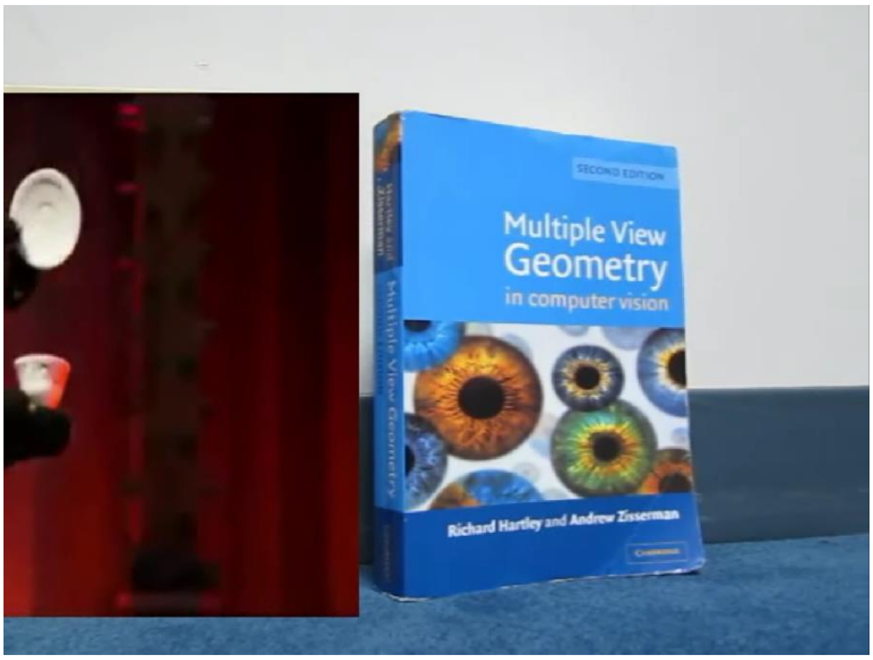
## <u>**After improvement:**</u>



## Before improvement:


SECOND EDITION
Multiple View Geometry in computer vision
Richard Hartley and Andrew Zisserman

## After improvement:


SECOND EDITION
Multiple View Geometry in computer vision
Richard Hartley and Andrew Zisserman

## Links:

**Google colab notebook:** https://colab.research.google.com/drive/1rVmO-8JQQ861LLob9pLLtQ4LsonT0828?usp=sharing

**Output video before improvement:** https://drive.google.com/file/d/1k9VC8-mqBHrR9NffmKHsKGi-E4L_EW5F/view?usp=sharing

**Output video after improvement:** https://drive.google.com/file/d/1S8AWHc3eYGC4Yll6Fr0vHdwyVo-WsB_T/view?usp=sharing

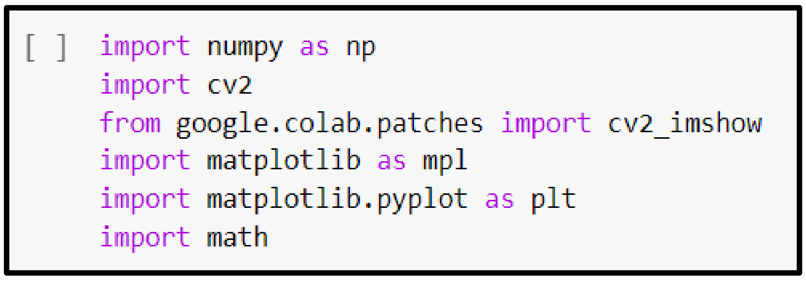
# Part 2

# Image Mosaics

## Problem statement:

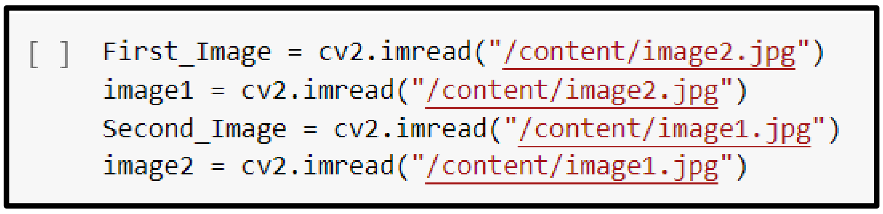
In this part of the assignment, we will implement an image stitcher that uses image warping and homographies to automatically create an image mosaic. We will focus on the case where we have two input images that should form the mosaic, where we warp one image into the plane of the second image and display the combined views.

## Imports:

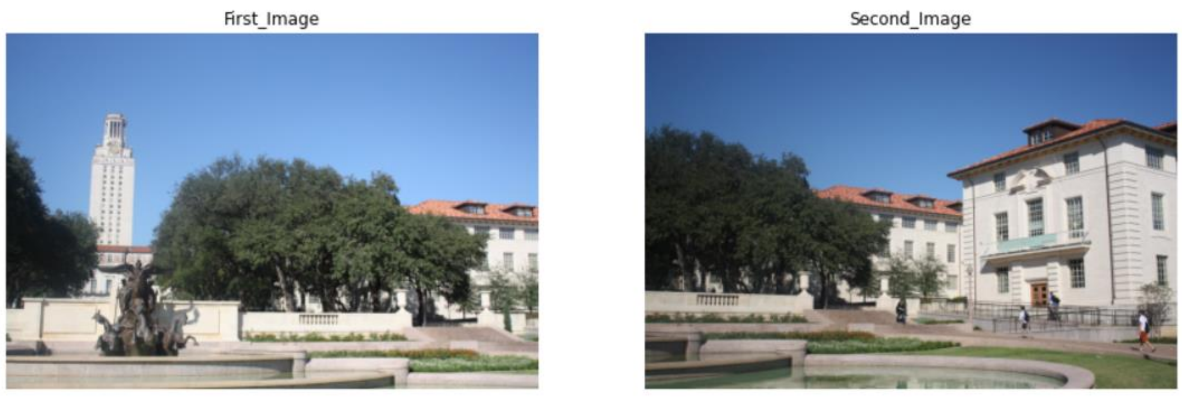


```
[ ]  import numpy as np
     import cv2
     from google.colab.patches import cv2_imshow
     import matplotlib as mpl
     import matplotlib.pyplot as plt
     import math
```

## Loading images:

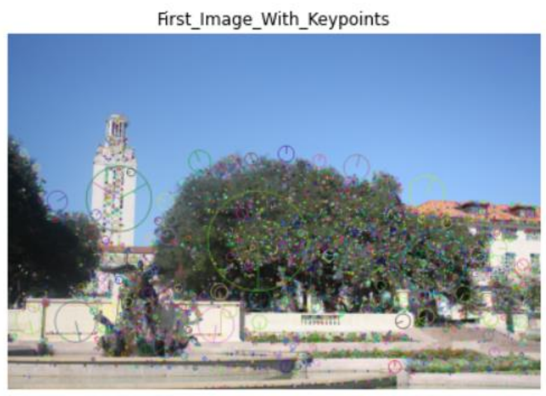


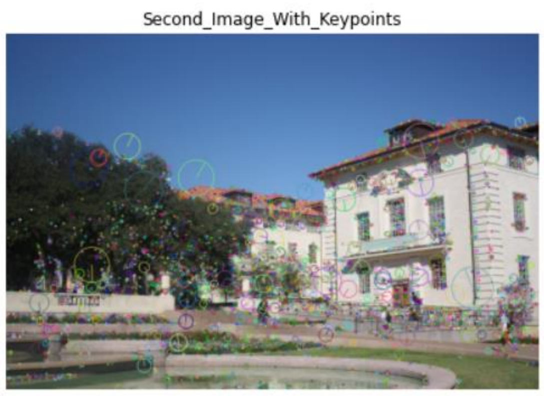
```
[ ]  First_Image = cv2.imread("/content/image2.jpg")
     image1 = cv2.imread("/content/image2.jpg")
     Second_Image = cv2.imread("/content/image1.jpg")
     image2 = cv2.imread("/content/image1.jpg")
```

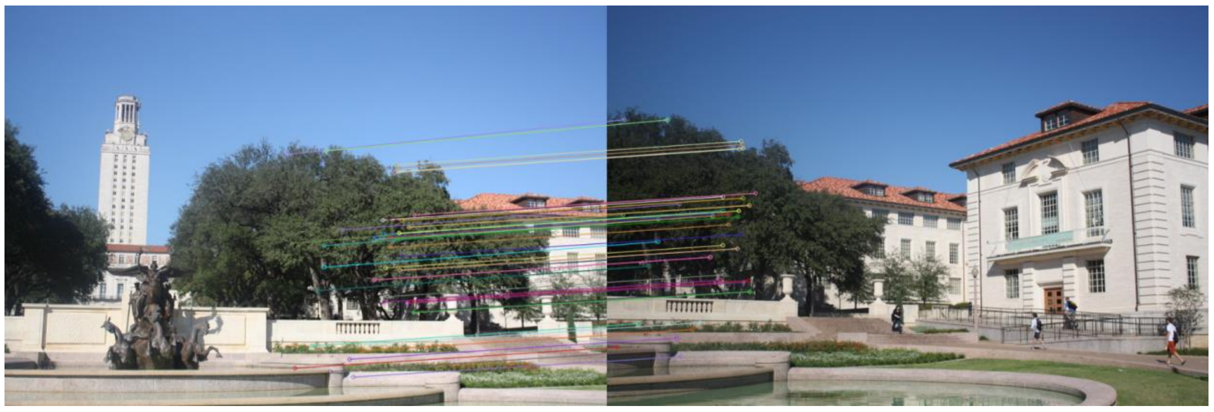


## 2.1 Getting Correspondences and Compute the Homography Parameters:

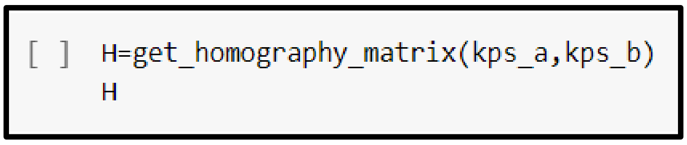
In this point, we used the same functions we used in part (1) of the assignment so we will just show the output.



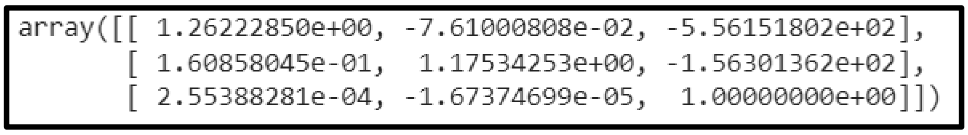




## Calculated homography matrix:

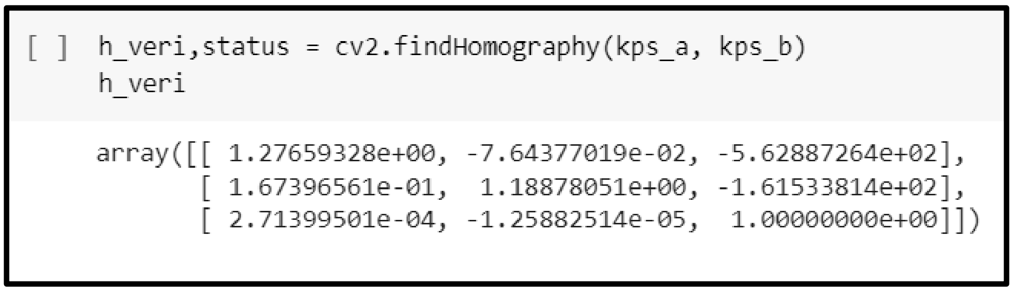


```
[ ]  H=get_homography_matrix(kps_a,kps_b)
     H
```



```
array([[ 1.26222850e+00, -7.61000808e-02, -5.56151802e+02],
       [ 1.60858045e-01,  1.17534253e+00, -1.56301362e+02],
       [ 2.55388281e-04, -1.67374699e-05,  1.00000000e+00]])
```

## OpenCV homography matrix:

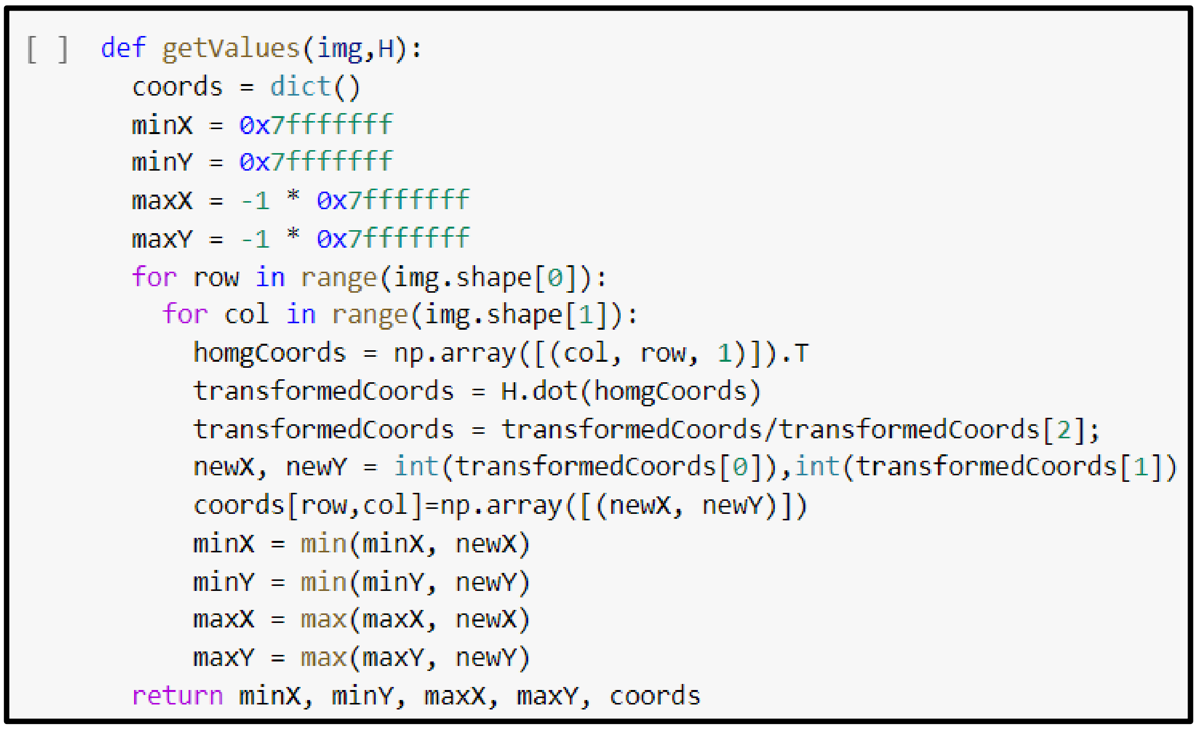


```
[ ]  h_veri,status = cv2.findHomography(kps_a, kps_b)
     h_veri

     array([[ 1.27659328e+00, -7.64377019e-02, -5.62887264e+02],
            [ 1.67396561e-01,  1.18878051e+00, -1.61533814e+02],
            [ 2.71399501e-04, -1.25882514e-05,  1.00000000e+00]])
```

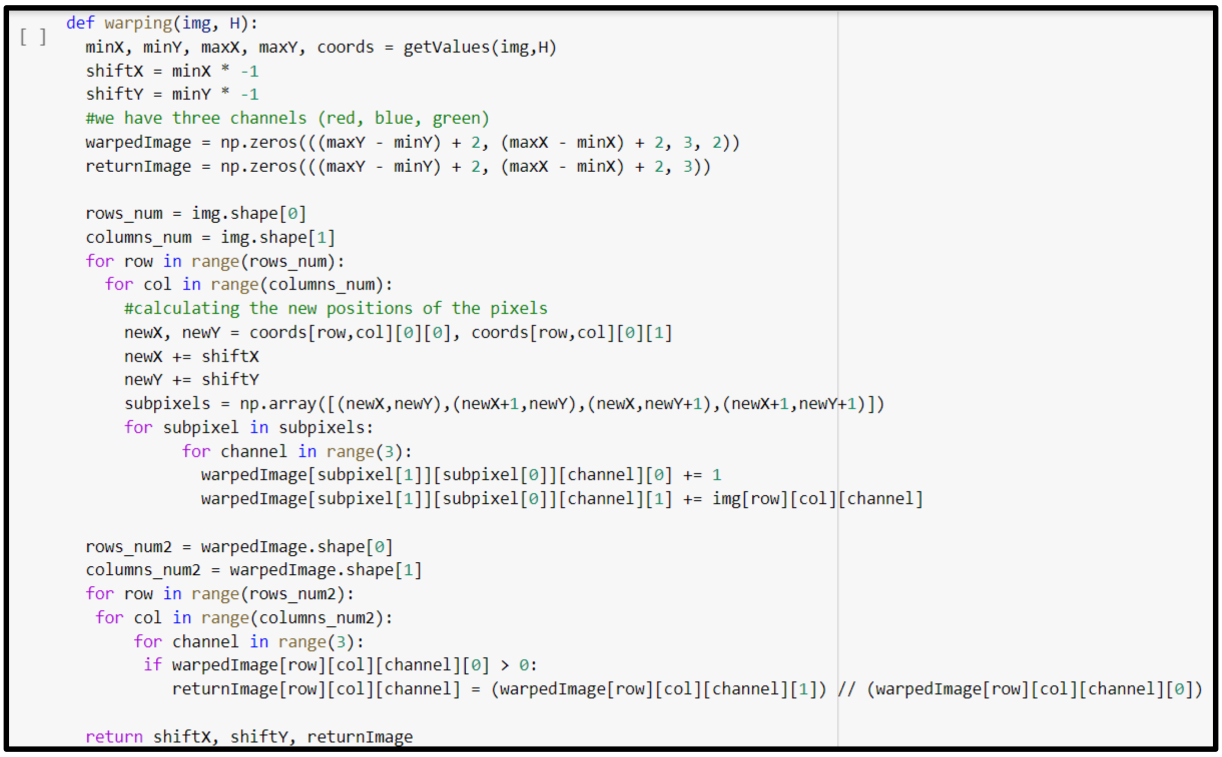
## 2.2 Warping Between Image Planes:

This function takes the recovered homography matrix and an image, and return a new image that is the warp of the input image using H. For coloured images like in our case, we warped each RGB channel separately and then stacked together to form the output.

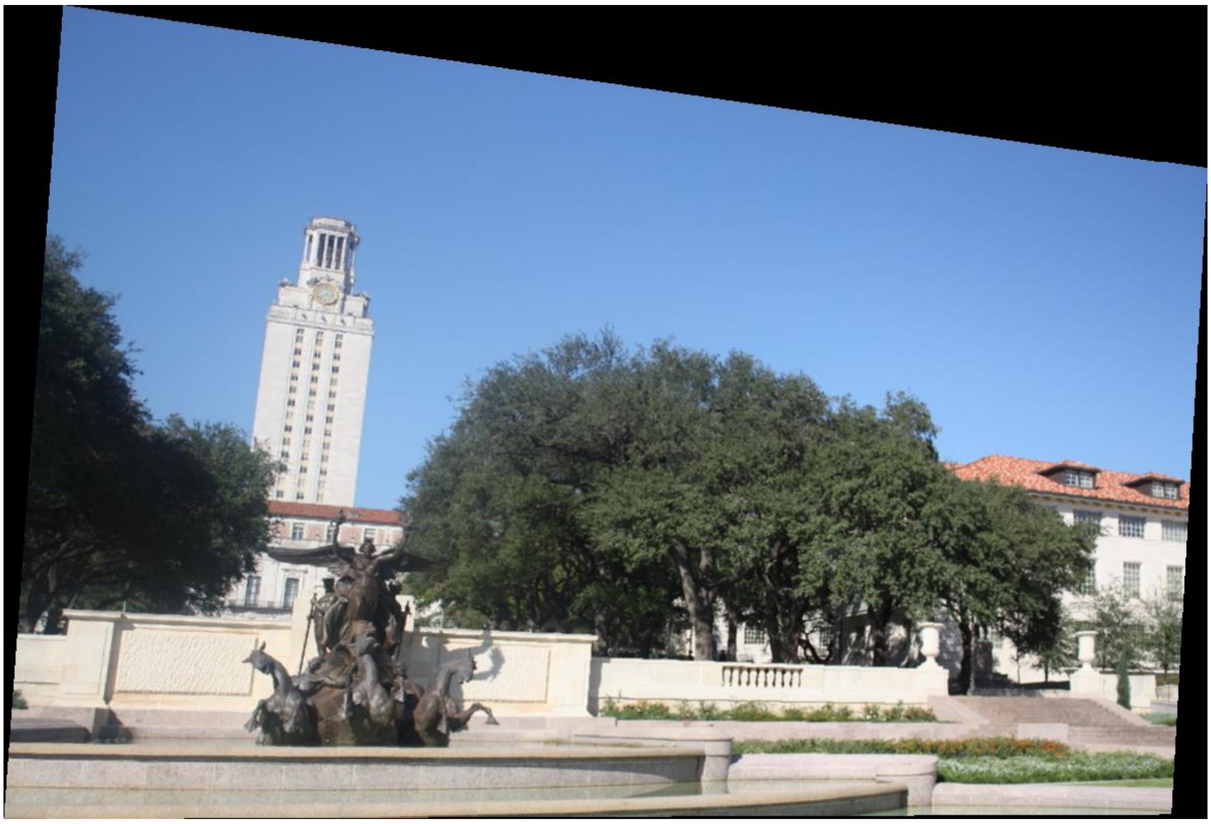
We wrote a function that returns the corners of the image and the new coordinates after mapping them by the homography matrix.



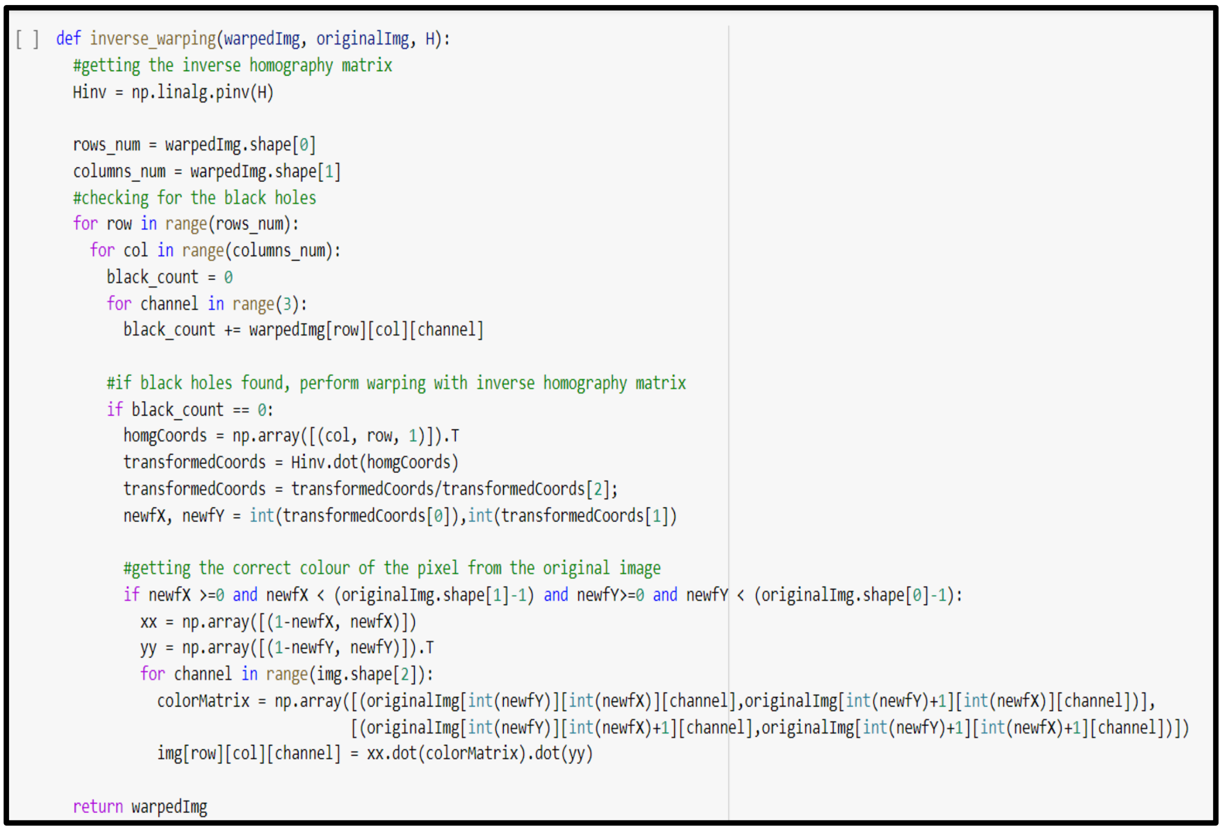
```
[ ]  def getValues(img,H):
       coords = dict()
       minX = 0x7fffffff
       minY = 0x7fffffff
       maxX = -1 * 0x7fffffff
       maxY = -1 * 0x7fffffff
       for row in range(img.shape[0]):
         for col in range(img.shape[1]):
           homgCoords = np.array([(col, row, 1)]).T
           transformedCoords = H.dot(homgCoords)
           transformedCoords = transformedCoords/transformedCoords[2];
           newX, newY = int(transformedCoords[0]),int(transformedCoords[1])
           coords[row,col]=np.array([(newX, newY)])
           minX = min(minX, newX)
           minY = min(minY, newY)
           maxX = max(maxX, newX)
           maxY = max(maxY, newY)
       return minX, minY, maxX, maxY, coords
```



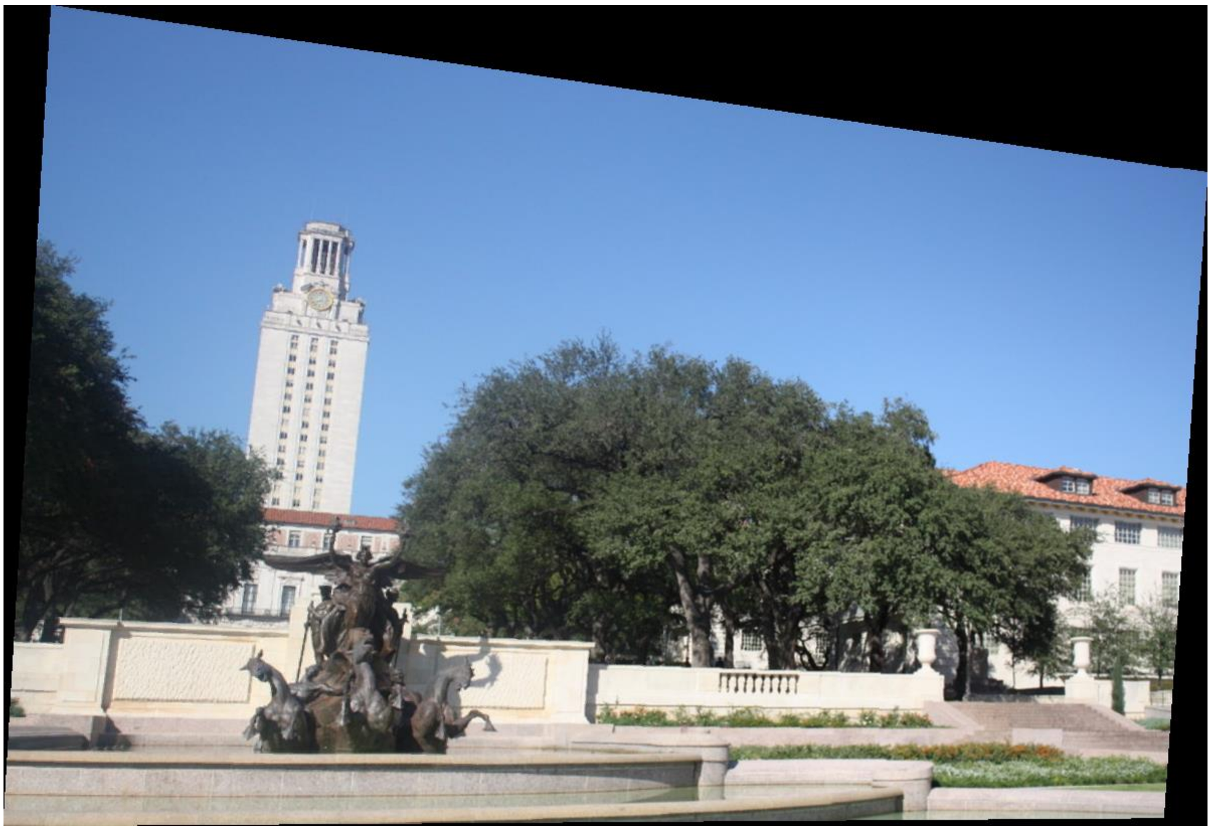
```
[ ]  def warping(img, H):
       minX, minY, maxX, maxY, coords = getValues(img,H)
       shiftX = minX * -1
       shiftY = minY * -1
       #we have three channels (red, blue, green)
       warpedImage = np.zeros(((maxY - minY) + 2, (maxX - minX) + 2, 3, 2))
       returnImage = np.zeros(((maxY - minY) + 2, (maxX - minX) + 2, 3))

       rows_num = img.shape[0]
       columns_num = img.shape[1]
       for row in range(rows_num):
         for col in range(columns_num):
           #calculating the new positions of the pixels
           newX, newY = coords[row,col][0][0], coords[row,col][0][1]
           newX += shiftX
           newY += shiftY
           subpixels = np.array([(newX,newY),(newX+1,newY),(newX,newY+1),(newX+1,newY+1)])
           for subpixel in subpixels:
                 for channel in range(3):
                   warpedImage[subpixel[1]][subpixel[0]][channel][0] += 1
                   warpedImage[subpixel[1]][subpixel[0]][channel][1] += img[row][col][channel]

       rows_num2 = warpedImage.shape[0]
       columns_num2 = warpedImage.shape[1]
       for row in range(rows_num2):
        for col in range(columns_num2):
            for channel in range(3):
             if warpedImage[row][col][channel][0] > 0:
                returnImage[row][col][channel] = (warpedImage[row][col][channel][1]) // (warpedImage[row][col][channel][0])

       return shiftX, shiftY, returnImage
```



To avoid holes in the output, we used an inverse warp. We warped the points from the source image into the reference frame of the destination and computed the bounding box in that new reference frame. Then, we sampled all points in that destination bounding box from the proper coordinates in the source image.

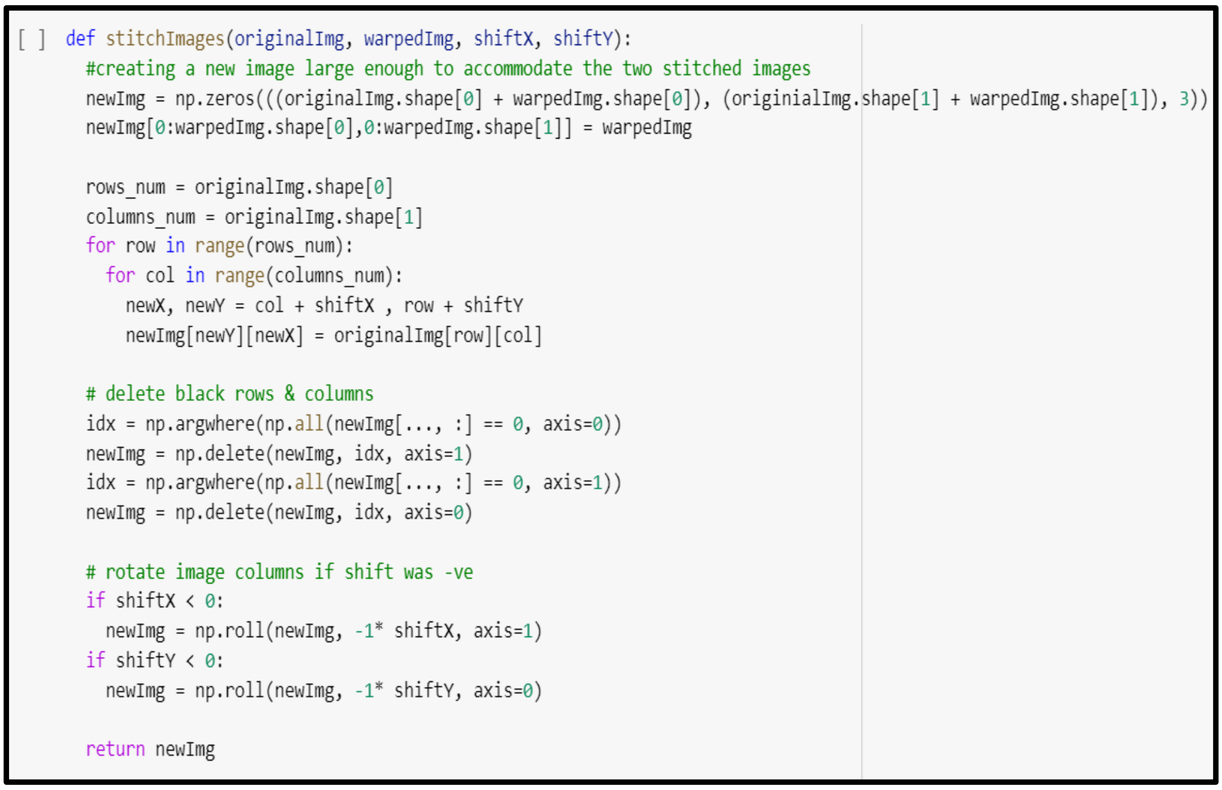


```
[ ] def inverse_warping(warpedImg, originalImg, H):
      #getting the inverse homography matrix
      Hinv = np.linalg.pinv(H)

      rows_num = warpedImg.shape[0]
      columns_num = warpedImg.shape[1]
      #checking for the black holes
      for row in range(rows_num):
        for col in range(columns_num):
          black_count = 0
          for channel in range(3):
            black_count += warpedImg[row][col][channel]

          #if black holes found, perform warping with inverse homography matrix
          if black_count == 0:
            homgCoords = np.array([(col, row, 1)]).T
            transformedCoords = Hinv.dot(homgCoords)
            transformedCoords = transformedCoords/transformedCoords[2];
            newfX, newfY = int(transformedCoords[0]),int(transformedCoords[1])

            #getting the correct colour of the pixel from the original image
            if newfX >=0 and newfX < (originalImg.shape[1]-1) and newfY>=0 and newfY < (originalImg.shape[0]-1):
              xx = np.array([(1-newfX, newfX)])
              yy = np.array([(1-newfY, newfY)]).T
              for channel in range(img.shape[2]):
                colorMatrix = np.array([(originalImg[int(newfY)][int(newfX)][channel],originalImg[int(newfY)+1][int(newfX)][channel])],
                                       [(originalImg[int(newfY)][int(newfX)+1][channel],originalImg[int(newfY)+1][int(newfX)+1][channel])])
                img[row][col][channel] = xx.dot(colorMatrix).dot(yy)

      return warpedImg
```



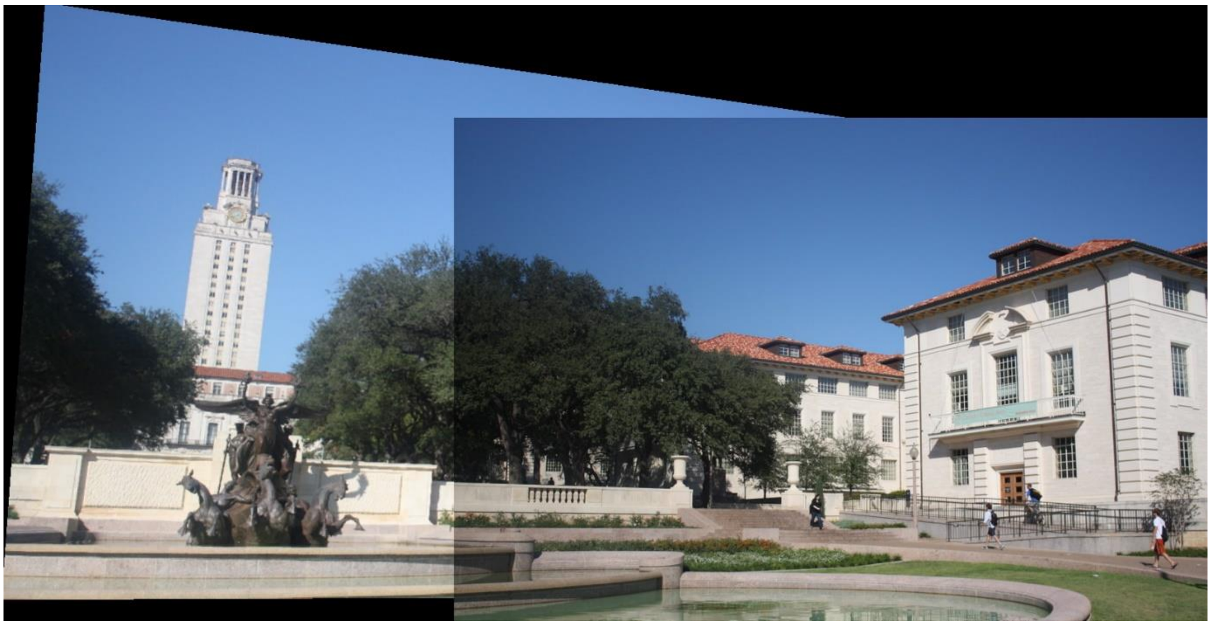
## 2.3 Create the output mosaic:

Once we have the source image warped into the destination images frame of reference, we created a merged image showing the mosaic. We made a new image large enough to hold both views and overlayed one view onto the other.

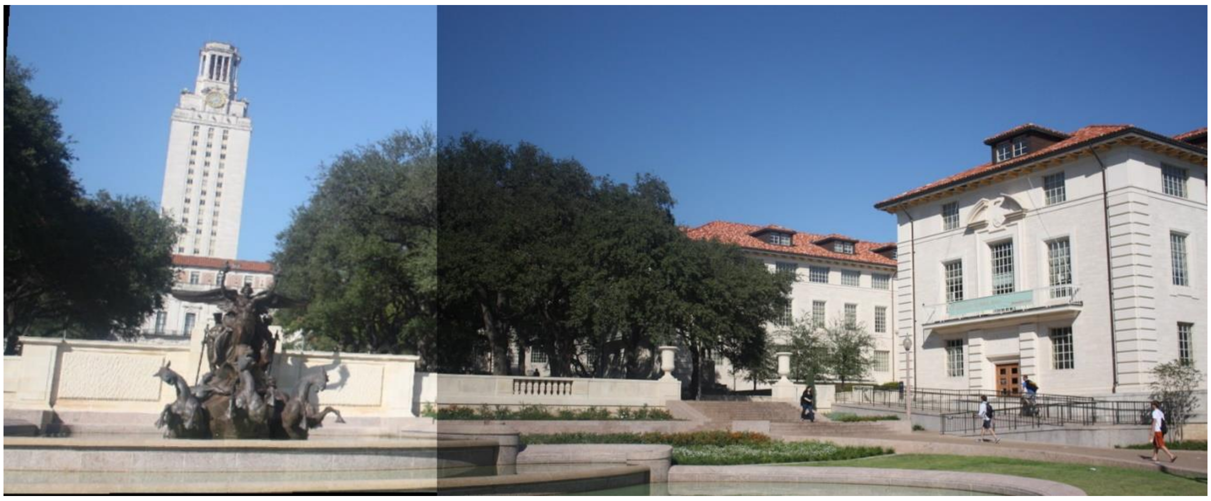


```
[ ]  def stitchImages(originalImg, warpedImg, shiftX, shiftY):
       #creating a new image large enough to accommodate the two stitched images
       newImg = np.zeros(((originalImg.shape[0] + warpedImg.shape[0]), (originialImg.shape[1] + warpedImg.shape[1]), 3))
       newImg[0:warpedImg.shape[0],0:warpedImg.shape[1]] = warpedImg

       rows_num = originalImg.shape[0]
       columns_num = originalImg.shape[1]
       for row in range(rows_num):
         for col in range(columns_num):
           newX, newY = col + shiftX , row + shiftY
           newImg[newY][newX] = originalImg[row][col]

       # delete black rows & columns
       idx = np.argwhere(np.all(newImg[..., :] == 0, axis=0))
       newImg = np.delete(newImg, idx, axis=1)
       idx = np.argwhere(np.all(newImg[..., :] == 0, axis=1))
       newImg = np.delete(newImg, idx, axis=0)

       # rotate image columns if shift was -ve
       if shiftX < 0:
         newImg = np.roll(newImg, -1* shiftX, axis=1)
       if shiftY < 0:
         newImg = np.roll(newImg, -1* shiftY, axis=0)

       return newImg
```

## Images after stitching:

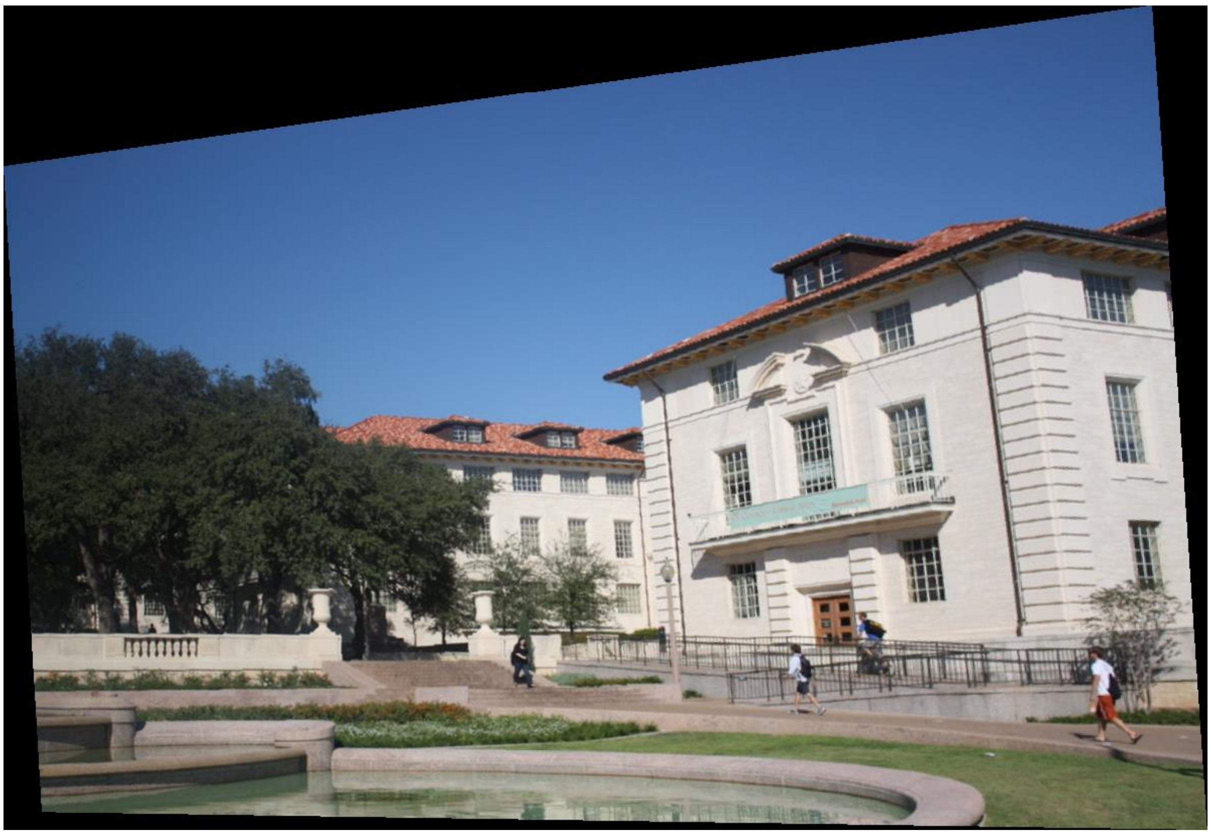


## After cropping out the black regions:

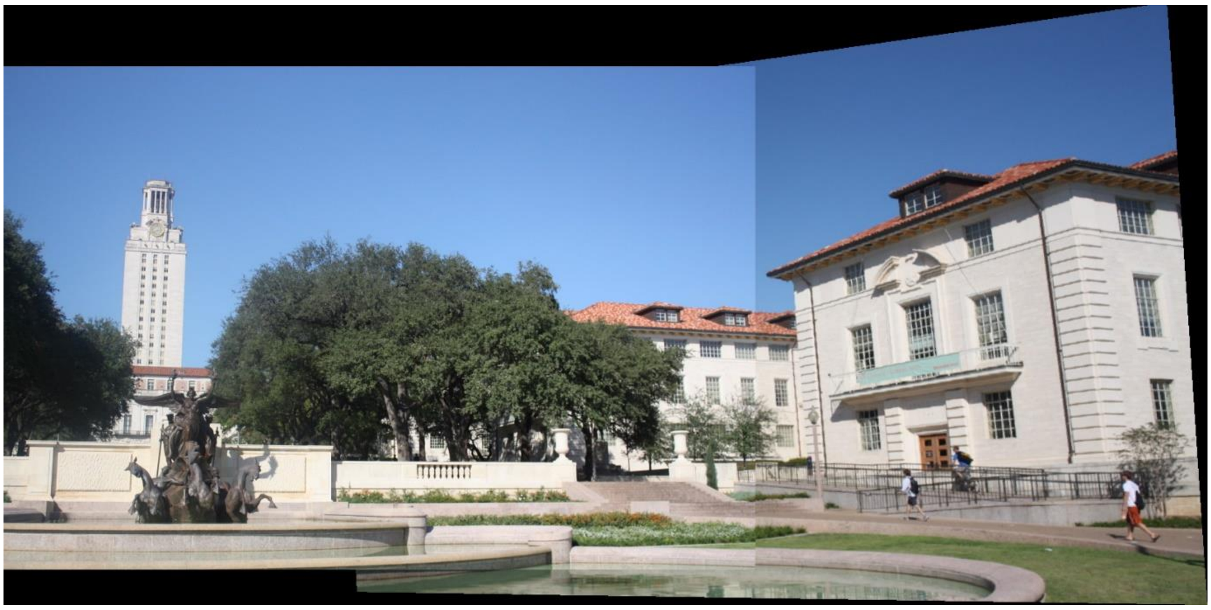


There was a note at the end of the pdf that the order in which we warp one image into the other matters in the results, so we tried warping the second image first and stitching the other one to it

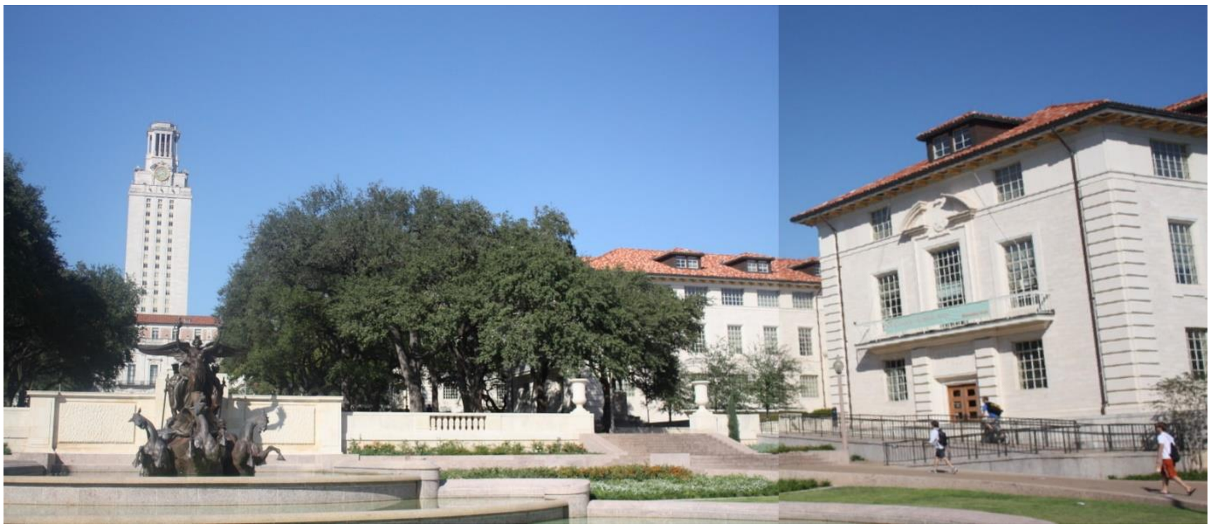
## **Warped image:**



## **Stitched images:**



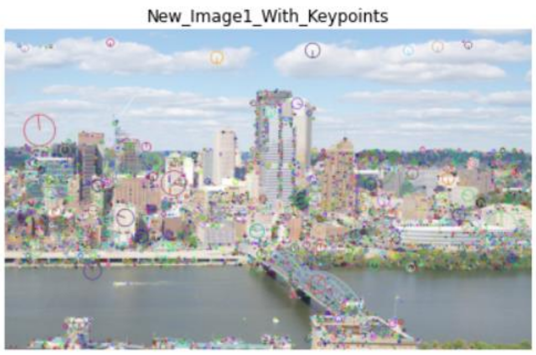
## Stitched images after cropping:

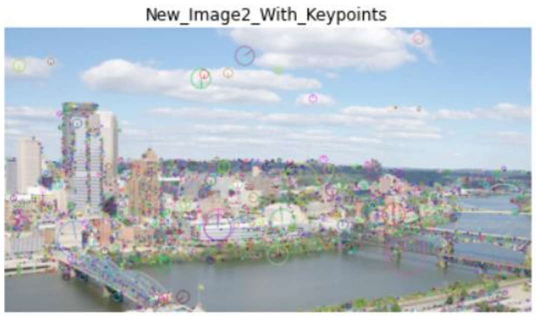


## Trying out two images of our own

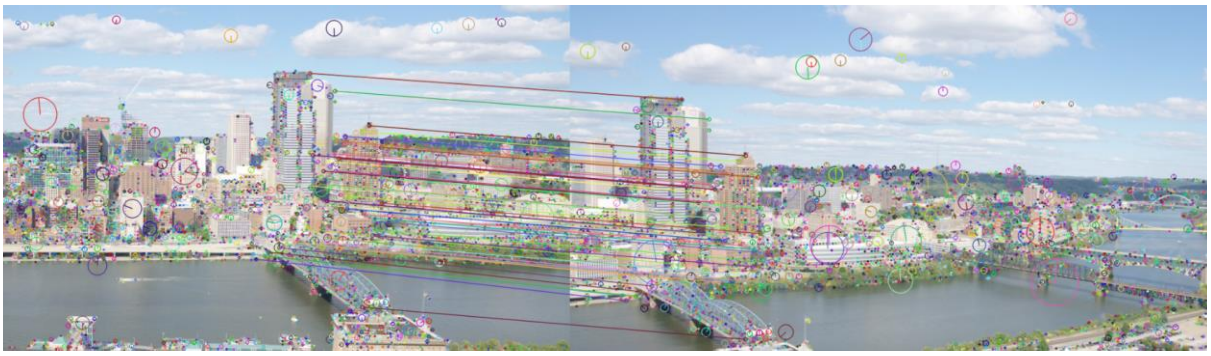
### The images:

### Images with key points:

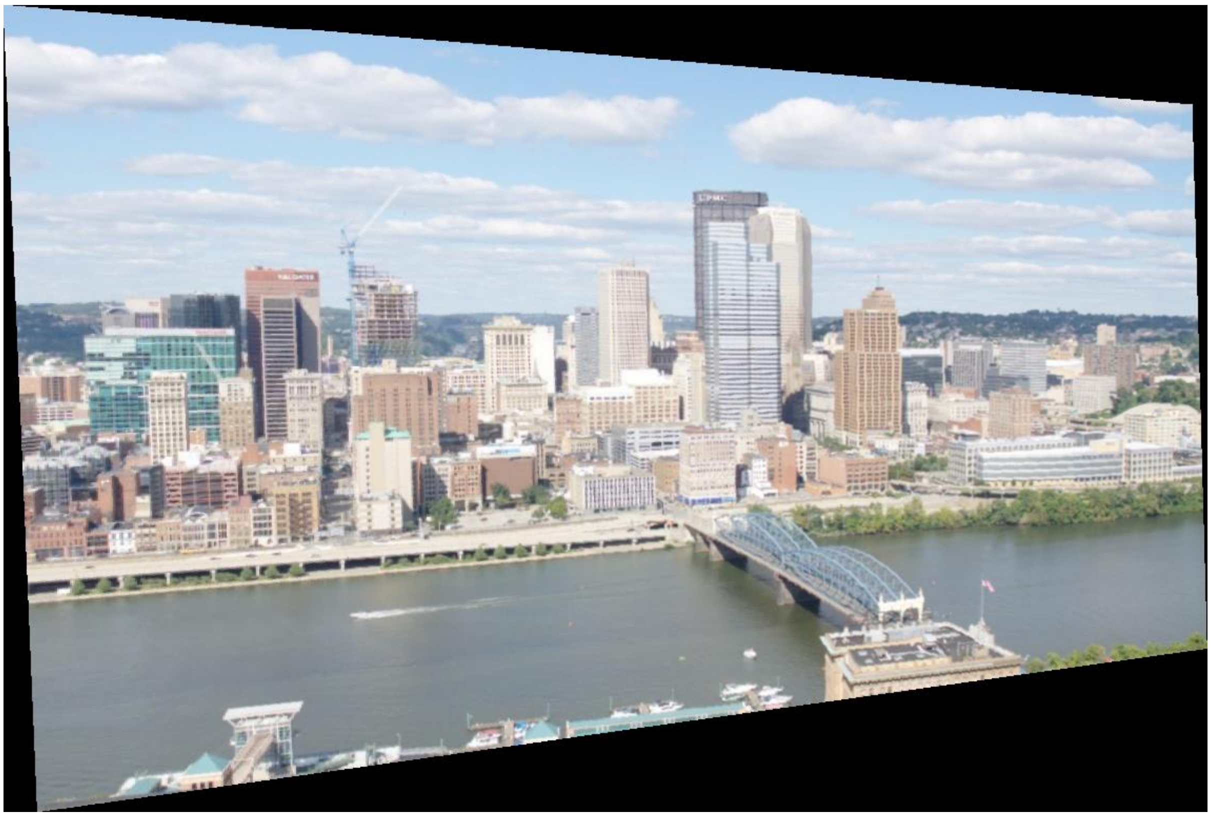




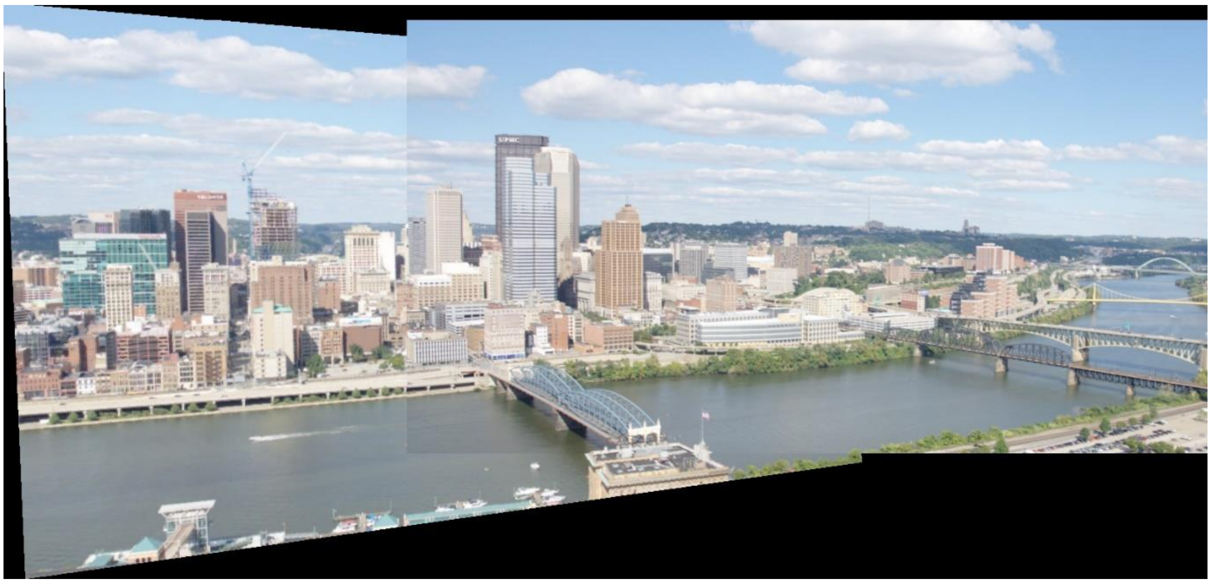
## Images with features matched:



## Warped image:



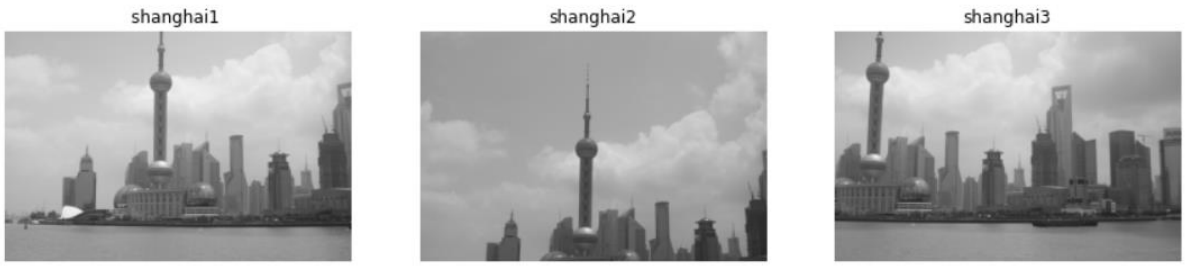
## Stitched images:



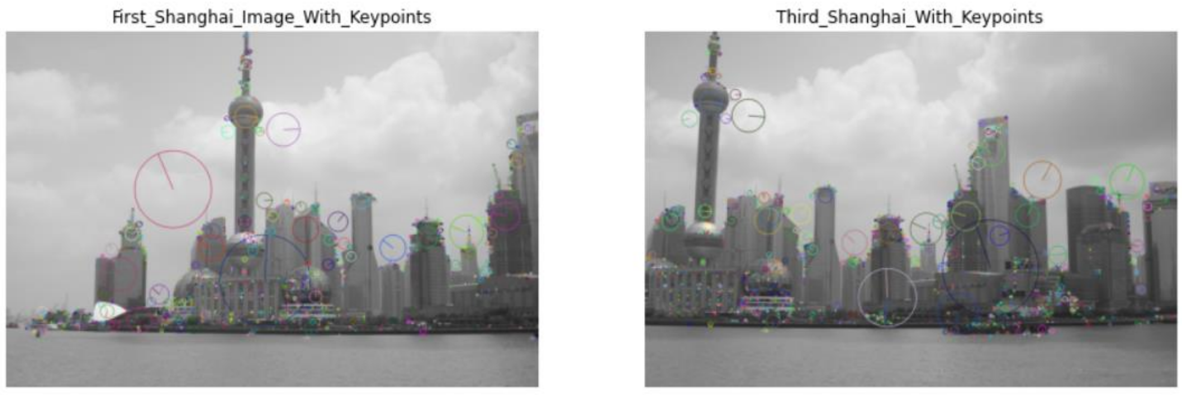
# Bonus

In the bonus part of this assignment, we tried out stitching 3 images not just two.

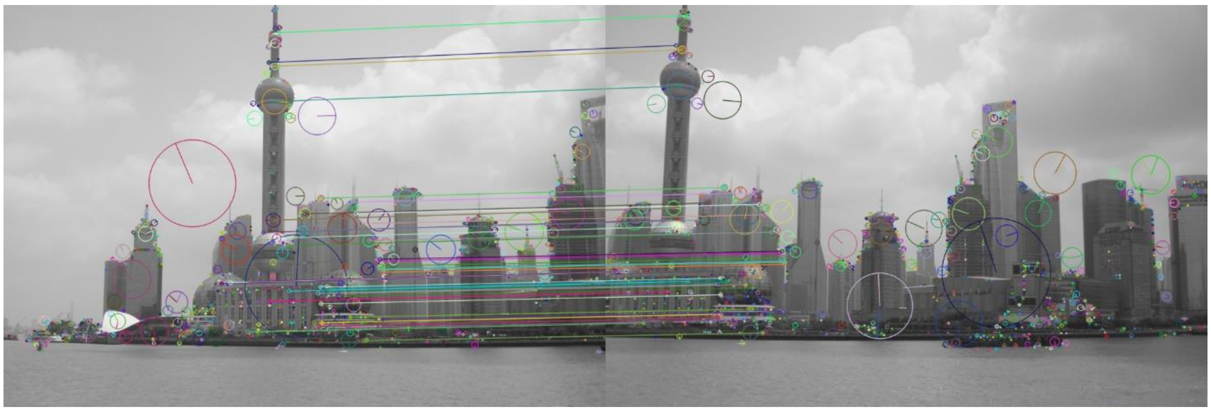
## The images:



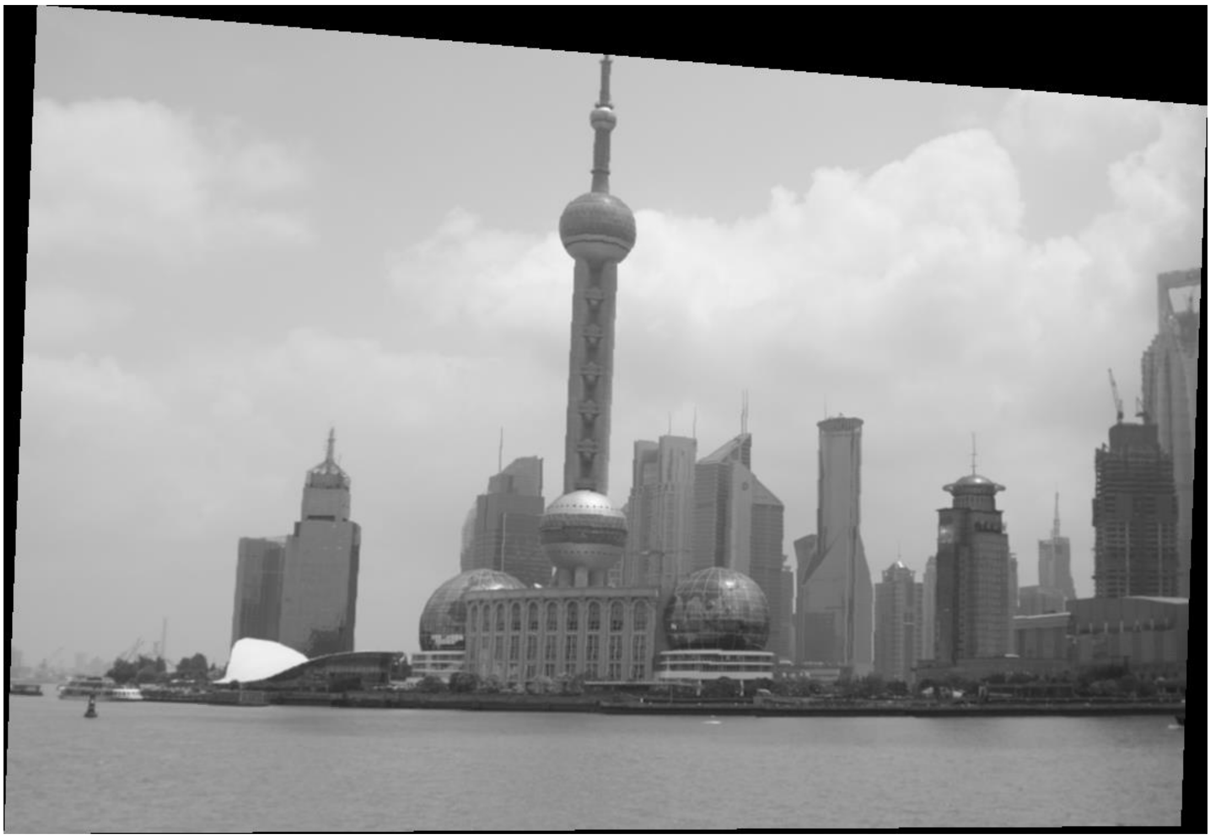
## First and third images with key points:


First_Shanghai_Image_With_Keypoints
Third_Shanghai_With_Keypoints

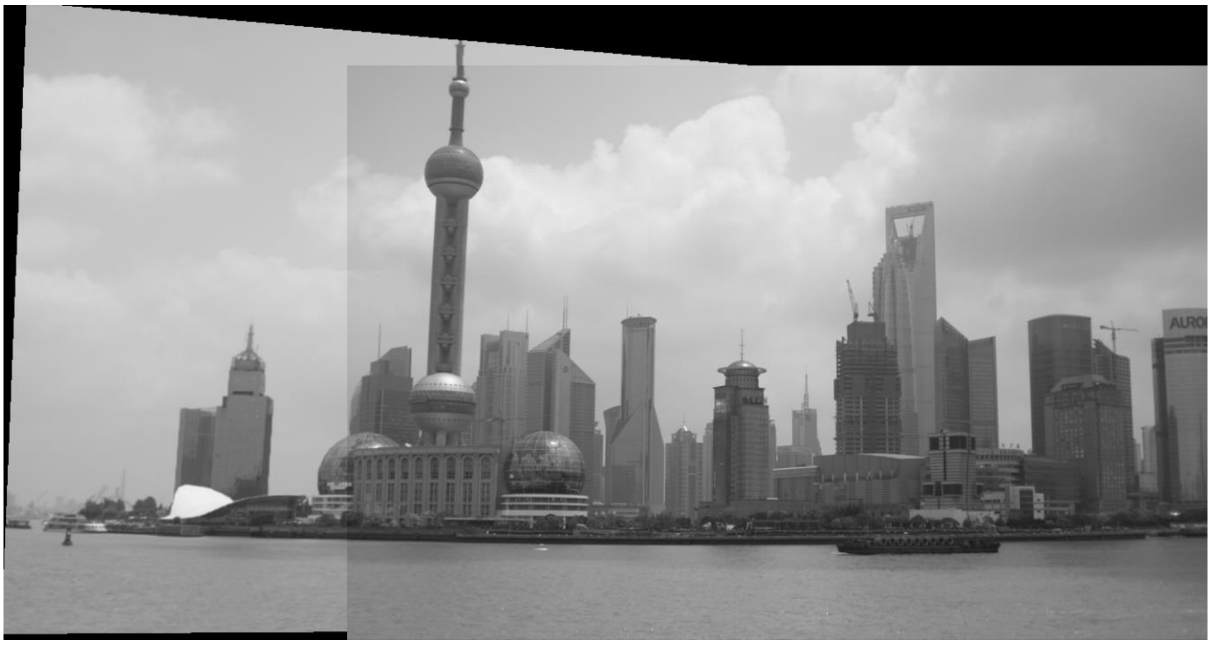
## First and third images features matched:



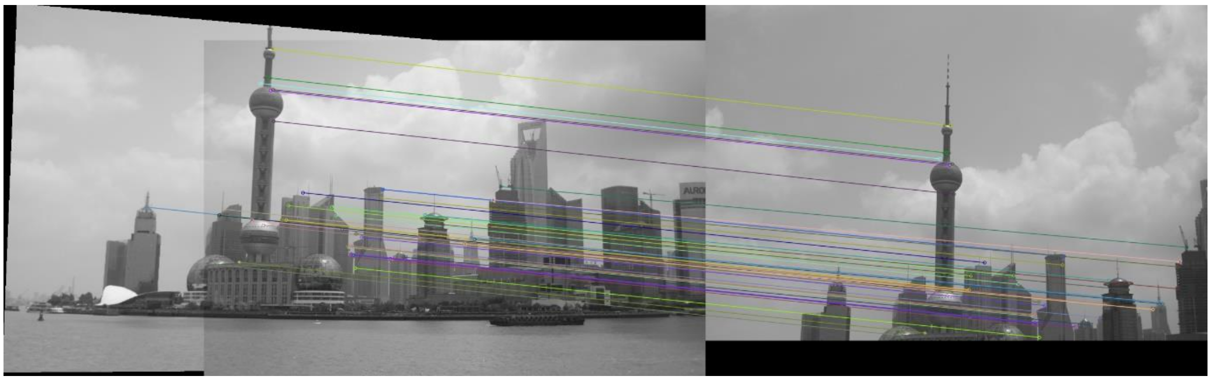
## First image warped:



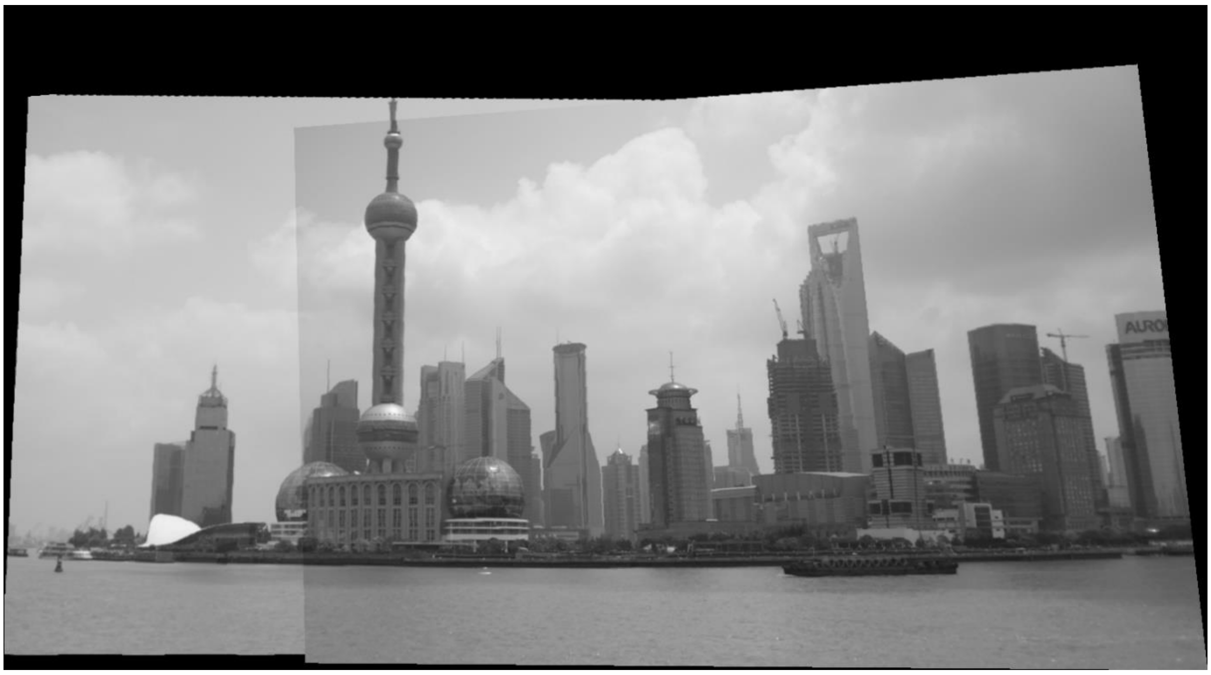
## First and third image stitched:



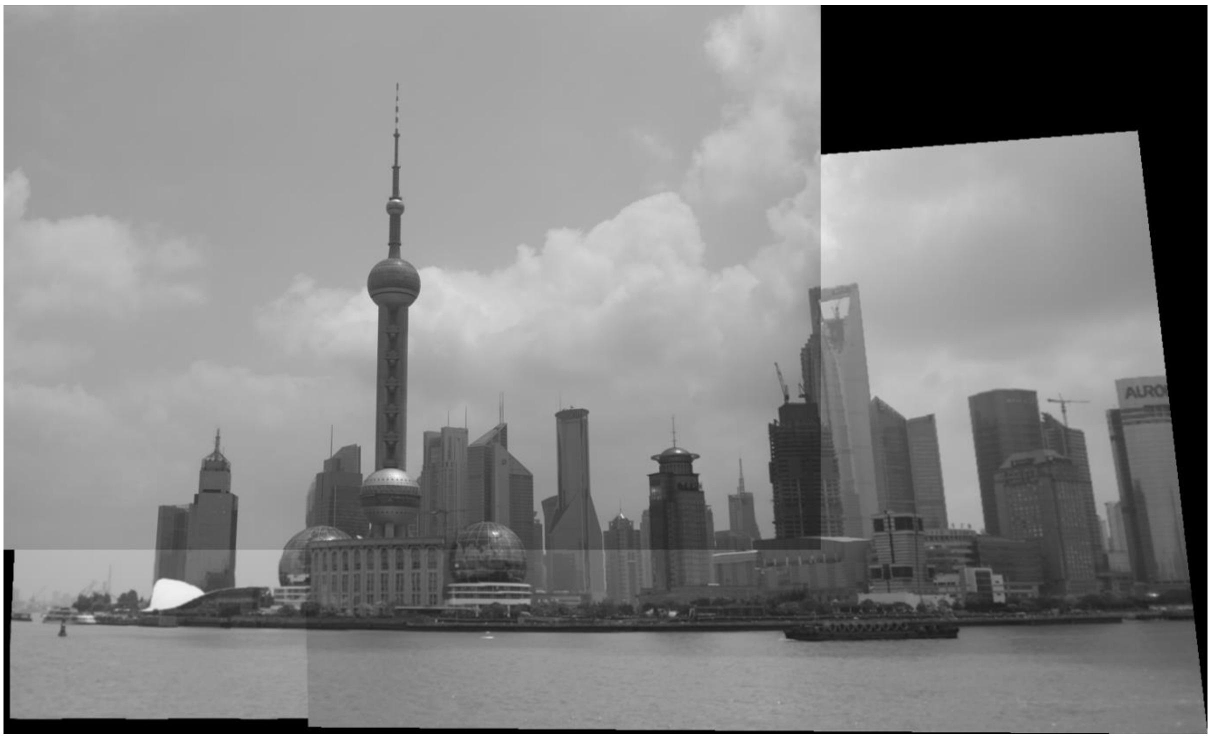
## Features matched between the stitched image and the second one:



## The stitched image warped:



## The 3 images stitched:



## Links:

**Google colab notebook:**
[https://colab.research.google.com/drive/1G_ikbmfbqg_ujfpNCJnUaoOqX28Wi3Hx?usp=sharing](https://colab.research.google.com/drive/1G_ikbmfbqg_ujfpNCJnUaoOqX28Wi3Hx?usp=sharing)