Computer Vision
Assignment 2

Students:

Mennatullah Ibrahim Mahmoud - 6221
Nour El-din Hazem Abdelsalam - 6261
Anas Ahmed Bekheit - 6659

Part 1:Augmented Reality with Planar Homographies

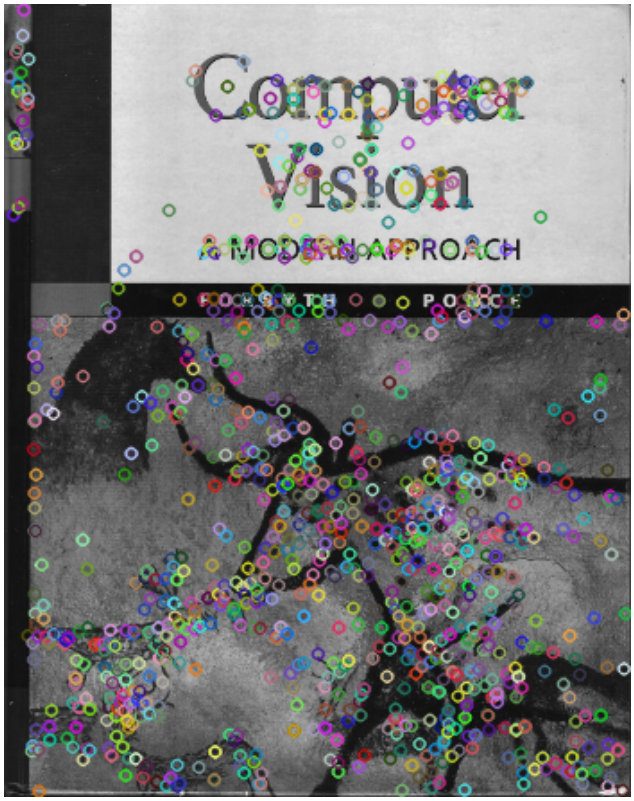
First step we calculate keypoints and descriptors using sift detector and descriptor then we draw the keypoints using cv2.drawKeypoints.
Keypoints (x,y) are local extrema in different scales of the image and they can be used to identify an image. Each kepoint has a descriptor vector that from its name is used to describe the features of this key point. The descriptor is 128 long and the information in it is the histogram of orientation of a four 16x16 patches around the keypoint. Each patch is represented by four histograms. Each histogram is represented by 8 bins. Total=4*4*8=128.

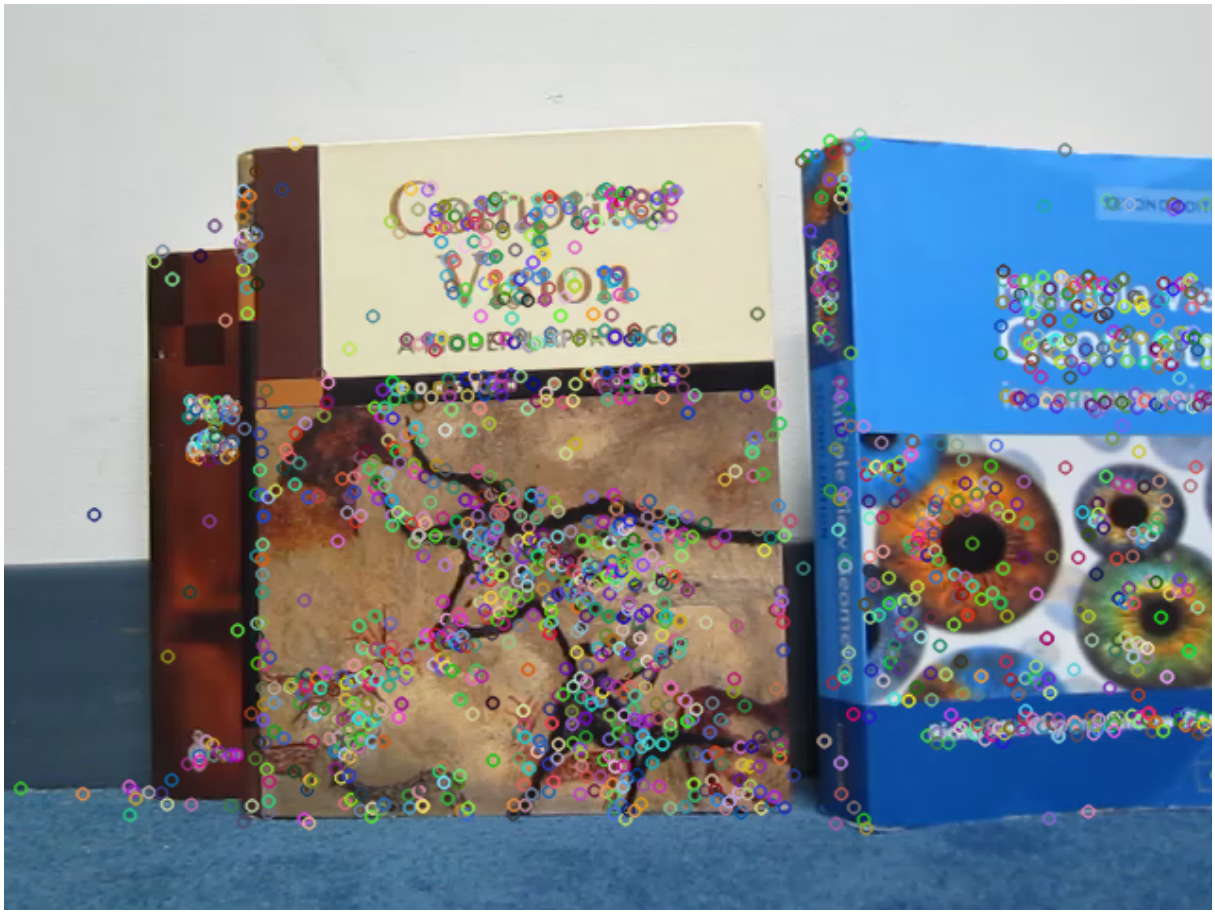
Code:

```
def sift(image):
  gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  sift = cv2.SIFT_create()
  keypoints, descriptors = sift.detectAndCompute(gray_image, None)
  sift_image = cv2.drawKeypoints(image, keypoints, 0)
  return sift_image, keypoints, descriptors
```

```
sift_image1, keypoints1, descriptors1 = sift(cover)
sift_image2, keypoints2, descriptors2 = sift(original_frames[0])
```

Results:





Next step we get the correspondences using brute force matcher from opencv and use the knn matcher to get the closest two descriptors to the descriptor of the key point. After this we calculate the ratio between the distance of the first match and the second match between the keypoint descriptor. If this ratio is a small number, it means that the first descriptor is way closer to the keypoint descriptor than the second one and it means we are more confident about considering this as a match. We sort the matches by this ratio and we choose the first 50 corresspondences. We return the matches sorted by the distance to the matched descriptors and draw 50 correspondences using drawMatches function from cv2.

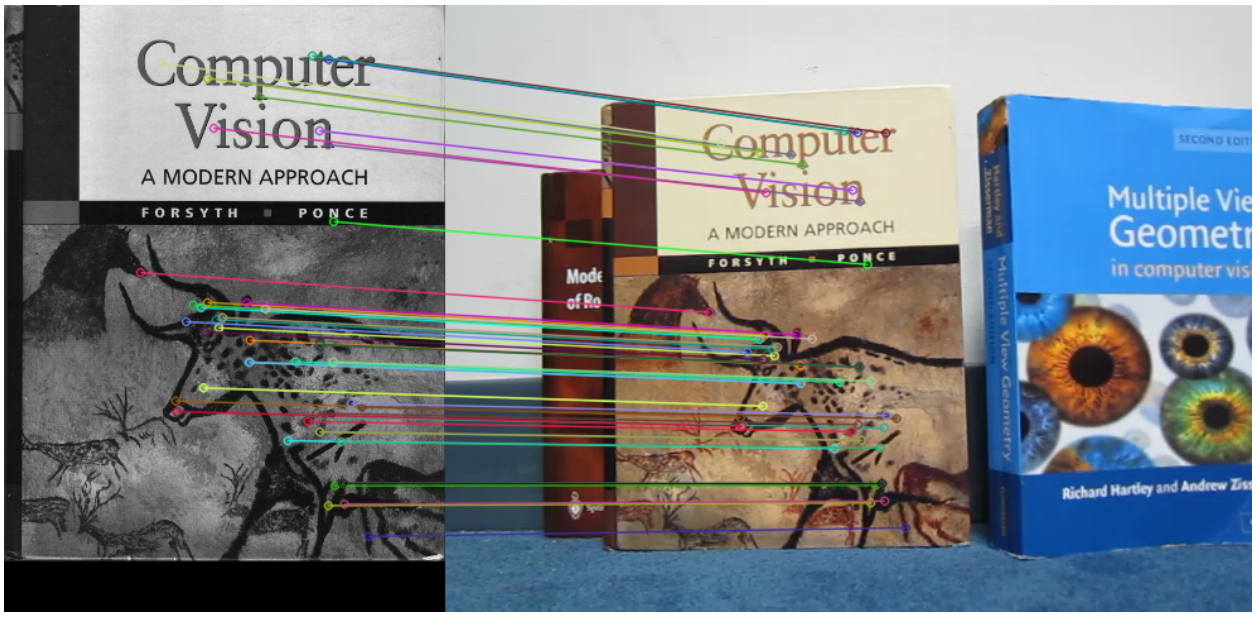
Code:

```
def key_matching(descriptors1, descriptors2, nkp=50):
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)
    #print(f"Number of matches before filtering = {len(matches)}")
    ratios = []
    for (i,j) in matches:
      ratios.append((i, i.distance/j.distance))
    ratios = sorted(ratios,key=lambda x:x[1])[0:nkp]
    final_matches = [x[0] for x in ratios]
    #print(f"Number of matches after filtering = {len(final_matches)}")
    final_matches = sorted(final_matches, key=lambda x:x.distance)
    return final_matches
```

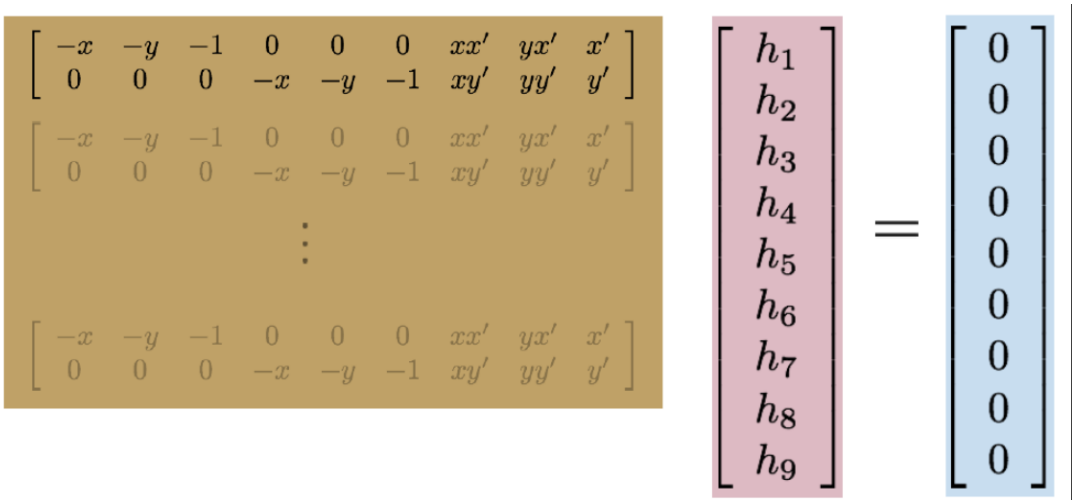
```
matches = key_matching(descriptors1, descriptors2)
matched_img = cv2.drawMatches(cover, keypoints1, original_frames[0], keypoints2, matches, 0, flags=2)
```

```
Number of matches before filtering = 1205
Number of matches after filtering = 50
```

Results:



Next step we repeat the last step but this time we will calculate the top 300 matches (according to ratio between 1st and 2nd descriptors distances) and get the 15 matches with the smalles distances.
We get the corresponding keypoints from each match and we use these correspondoncies to calculate the A matrix, then by applying SVD to the A matrix we get the eigen vector corresponding to the smallest eigen value then we reshape it to get the homography matrix.



Code:

```
def get_kp(matches):
  image1_points=[]
  image2_points=[]
  for m in matches:
    i = m.queryIdx
    j = m.trainIdx
    image1_points.append(keypoints1[i].pt)
    image2_points.append(keypoints2[j].pt)
  return image1_points, image2_points
```

```
def constructA(image1_points,image2_points,start=0,end=50):
  A = np.zeros(((end-start)*2,9))
  for i in range(0,(end-start)*2,2):
    (x1,y1) = image1_points[i//2]
    (x2,y2) = image2_points[i//2]
    A[i] = np.array([-1*x1,-1*y1,-1,0,0,0,x1*x2,y1*x2,x2])
    A[i+1] = np.array([0,0,0,-1*x1,-1*y1,-1,x1*y2,y1*y2,y2])
  return A
```

```
def get_homography(A):
  U, s, Vh = linalg.svd(A)
  H = Vh[-1,:] / Vh[-1,-1]
  H = H.reshape(3, 3)
  return H
```

Next step we checked our homography function with the built-in function. The results were almost identical.

Code:

```
def builtin_check(matches,H,point):
  image1_points2, image2_points2 = get_kp(matches)
  h, status = cv2.findHomography(np.array(image1_points2), np.array(image2_points2))
  transformedPoint1 = H @ np.array(point).transpose()
  transformedPoint1 /= transformedPoint1[-1]
  transformedPoint2 = h @ np.array(point).transpose()
  transformedPoint2 /= transformedPoint2[-1]
  print(f"Our function: {transformedPoint1}")
  print(f"Built-in function: {transformedPoint2}")
  if sum(abs(transformedPoint1-transformedPoint2))/2 <1:
    print("Acceptable Range")
  else:
    print("Too far off!")
  if len(h[abs(h - H) >20])==0:
    print("Almost no difference")
    return True
  return False
```

```
image1_points , image2_points = get_kp(matches)
A =constructA(image1_points,image2_points)
H = get_homography(A)
```

```
builtin_check(matches, H, [100,100,1])
```

```
Our function: [200.46777862 153.06750905   1.        ]
Built-in function: [200.43774218 153.02489693   1.        ]
Acceptable Range
Almost no difference
True
```

Next step we detect the 4 corners of the book in the video using the homography matrix we calculated. We use the extreme points in the book cover image and calculate the new corners by multplying these points with the homography matrix acquired in the previous step. We first turn the heterogeneous coordinates into homogenous one and after acquiring the new points we divide by the third coordinate so we can transform it back to heterogeneous coordinate. Then we draw a frame around the book.

Code:

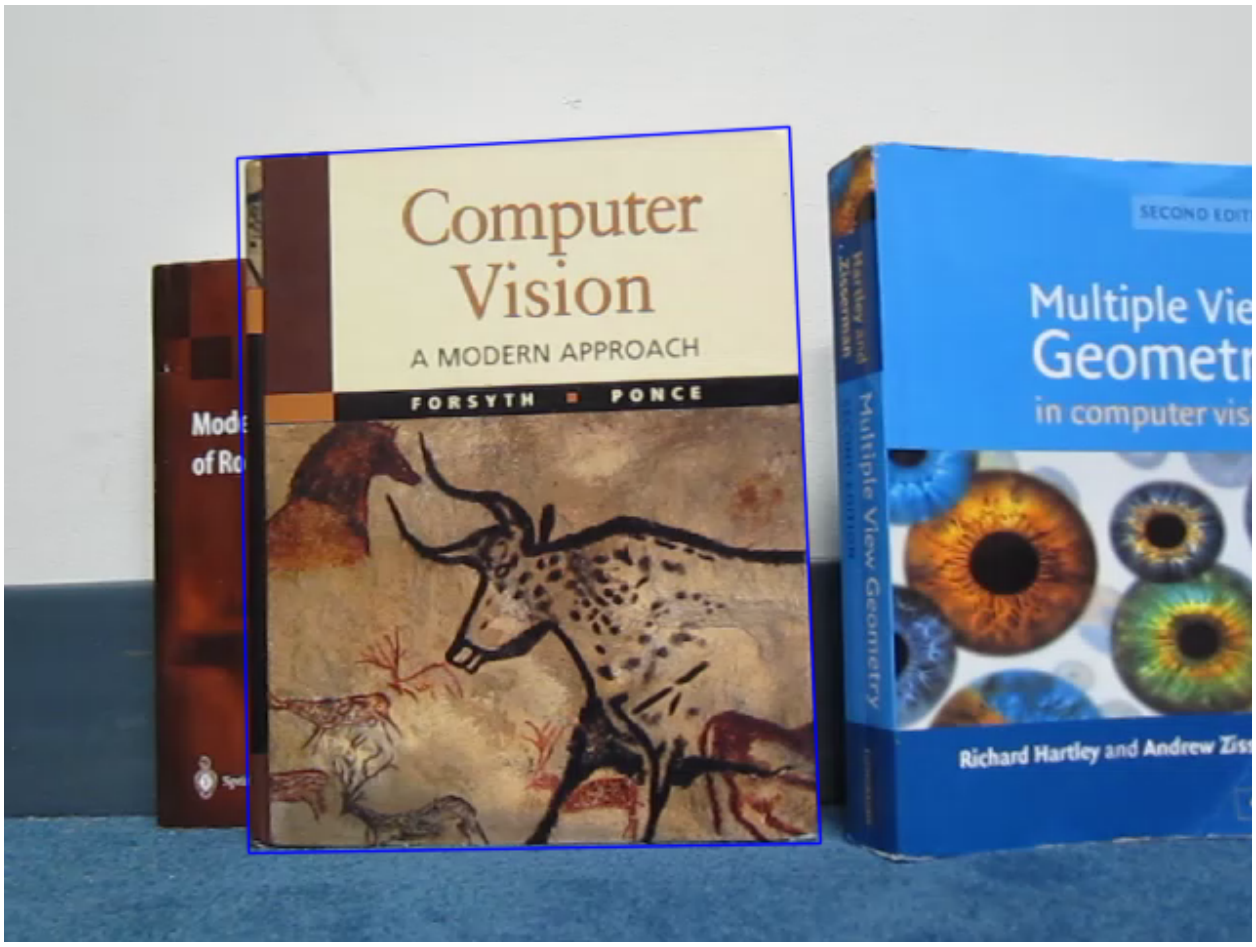
```
def get_new_points(points, H):
  new_points =[]
  for original_point in points:
    point = original_point.copy()
    point.append(1)
    point = np.array(point)
    point = point[...,np.newaxis]
    new_point = H @ point
    new_point = new_point/new_point[-1]
    new_points.append(new_point.T)
  new_points = np.array(new_points)[:,:,0:2]
  new_points = new_points.squeeze()
  return new_points
```

```
def draw_frame(img, corners):
  copy_image = img.copy()
  corner_points = [(i,j) for i,j in np.round(corners).astype('int64')]
  copy_image = cv2.line(copy_image, corner_points[0],corner_points[1], (255, 0, 0) , 1, cv2.LINE_AA)
  copy_image = cv2.line(copy_image, corner_points[0], corner_points[2], (255, 0, 0) , 1, cv2.LINE_AA)
  copy_image = cv2.line(copy_image, corner_points[2], corner_points[3], (255, 0, 0) , 1, cv2.LINE_AA)
  copy_image = cv2.line(copy_image, corner_points[1], corner_points[3], (255, 0, 0) , 1, cv2.LINE_AA)
  return copy_image
```

```
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
new_corners = get_new_points(corners, H)
```

```
copy_image = draw_frame(original_frames[0],new_corners)
cv2_imshow(copy_image)
```

Output:



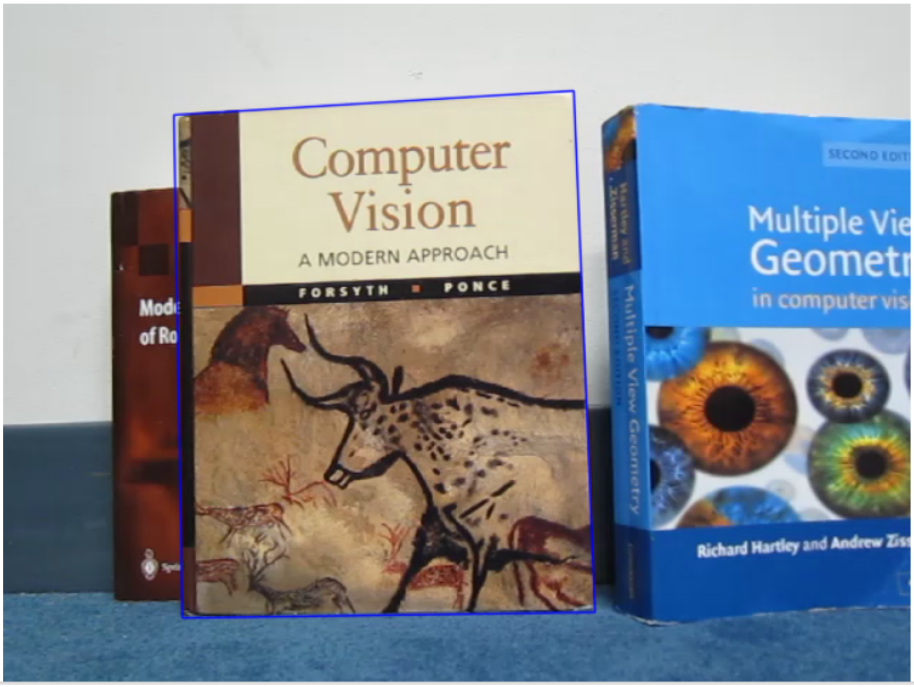
Next step we checked our output with the built-in function output

Code:

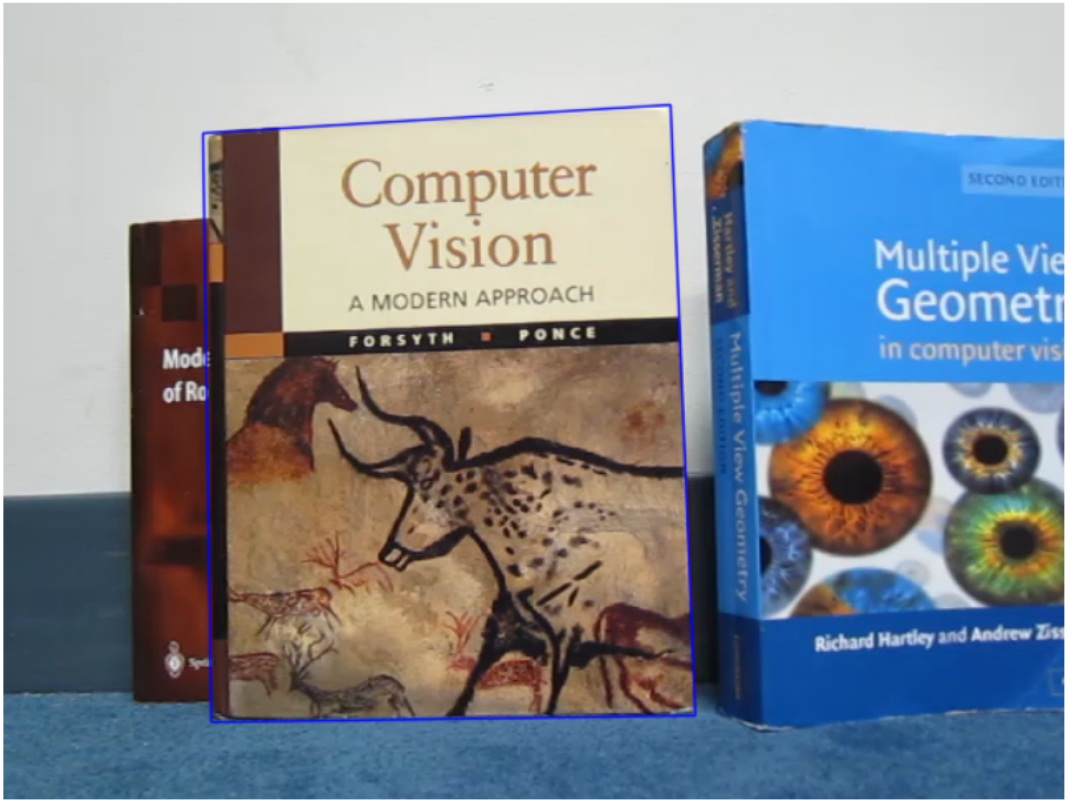
```
i = 0
sift_image1, keypoints1, descriptors1 = sift(cover)
sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
matches = key_matching(descriptors1, descriptors2,nkp=300)
image1_points , image2_points = get_kp(matches)
A =constructA(image1_points,image2_points,start=0,end=15)
H = get_homography(A)
h, status = cv2.findHomography(np.array(image1_points)[0:15], np.array(image2_points)[0:15])
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
new_corners = get_new_points(corners, H)
new_corners2 = get_new_points(corners, h)
copy_image = draw_frame(original_frames[i],new_corners)
print("Our Function")
cv2_imshow(copy_image)
copy_image = draw_frame(original_frames[i],new_corners2)
print("Built-in Function")
cv2_imshow(copy_image)
```

Output:

```
Number of matches before filtering = 1205
Number of matches after filtering = 300
Our Function
```




```
Built-in Function
```

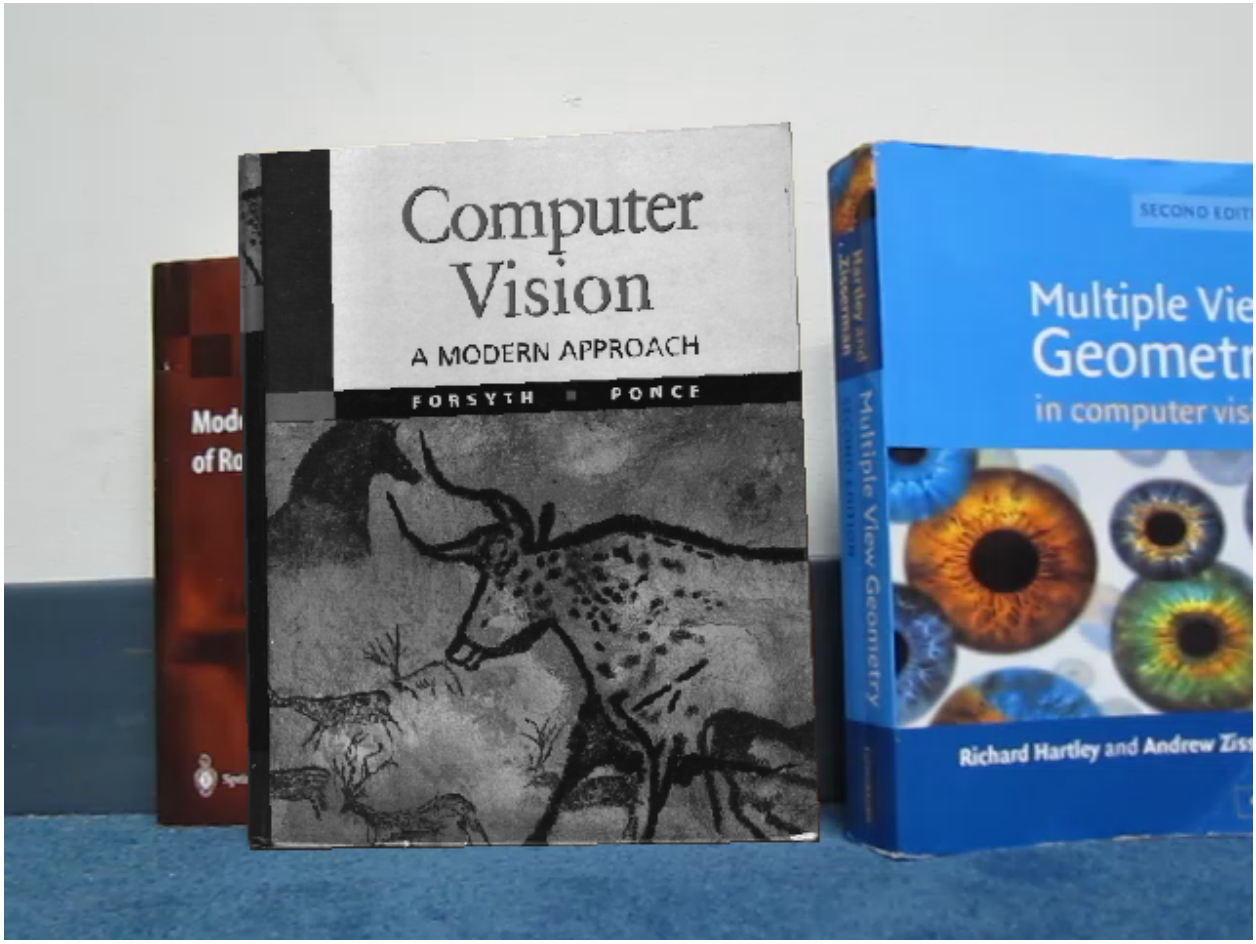


Next step we overlay the cover photo to the book photo in the video

Code:

```
def overlay(overlay_image, original_image, H):
  indices = [[i,j] for i in range(overlay_image.shape[1]) for j in range(overlay_image.shape[0])]
  new_points = get_new_points(indices,H)
  copy_image = original_image.copy()
  for i,(x,y) in enumerate(new_points):
    [x2,y2] = indices[i]
    if round(x) < original_image.shape[1] and round(y) < original_image.shape[0] and round(x) > 0 and round(y) > 0:
      copy_image[round(y),round(x)] = overlay_image[y2,x2]
  return copy_image
```

Output:



Next step we saved video of all frames with the book outlined to make sure that the four corners are correctly detected in every frame.

Code:

```
sift_image1, keypoints1, descriptors1 = sift(cover)
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
frames_stack = []
for i in tqdm(range(original_frames.shape[0])):
  sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
  matches = key_matching(descriptors1, descriptors2,nkp=300)
  image1_points , image2_points = get_kp(matches)
  A =constructA(image1_points,image2_points,start=0,end=15)
  H = get_homography(A)
  new_corners = get_new_points(corners, H)
  copy_image = draw_frame(original_frames[i],new_corners)
  frames_stack.append(copy_image)
```

```
100%|██████████| 641/641 [02:33<00:00,  4.18it/s]
```

```
write_video("/content/drive/MyDrive/asg2_cv/output_outline.mov",frames_stack,fps1)
```

Next step we center the movie frame on the book, by scaling and translating the movie frame center to the center of the book frame. We first scale the movie frame so the heights of both frame matches then we calculate the center of the book frame to calculate the amount of translation we need to align both centers. After this we built a function to check if the pixels are within the frame borders. We do this by substituting the x and y of the pixel in the four equations of the frame borders. If the pixel lies underneath the top line, to the right of the left line, to the left of the right line and above the bottom line, we save its index. After this, by simple slicing we equate the pixels in the original video frames lying at these indices by the value of their corresponding pixels in the movie frames. We apply these two functions and save the output.

Code:

```
def center_frames(movie_frame,frame_shape,corners):
  ratio = frame_shape[0]/movie_frame.shape[0]
  copy_frame = cv2.resize(movie_frame, (round(movie_frame.shape[1]*ratio),round(movie_frame.shape[0]*ratio)))
  midpoint1 = (corners[0] + corners[1])/2
  midpoint2 = (corners[2] + corners[3])/2
  midpoint = (midpoint1+midpoint2)//2
  center = [copy_frame.shape[1]//2,copy_frame.shape[0]//2]
  translation = -1*midpoint + center
  black = np.zeros(copy_frame.shape)
  for i in range(black.shape[0]):
    if i+translation[1] < copy_frame.shape[0]:
      black[i,:,:] = copy_frame[i+round(translation[1]),:,:]
  for j in range(black.shape[1]):
    if j+translation[0] < black.shape[1]:
      black[:,j,:] = black[:,j+round(translation[0]),:]
  output = black[:,0:frame_shape[1],:]
  return output
```

```
def check_if_inside(corners,point):
  p1,p2,p3,p4 = corners
  #down
  if (((p2[1]-p1[1])/(p2[0]-p1[0])) - ((p1[1]-point[1])/(p1[0]-point[0])) <=0 and point[0]>=p1[0]) or (((p2[1]-p1[1])/(p2[0]-p1[0])) - ((p1[1]-point[1])/(p1[0]-point[0]))
    #left
    if (round(p1[0]) == round(p3[0]) and point[0] >= round(p1[0])) or (((p3[1]-p1[1])/(p3[0]-p1[0])) - ((p1[1]-point[1])/(p1[0]-point[0])) >=0 and point[0]>=p1[0] and p3[0
      #up
      if ((p4[1]-p3[1])/(p4[0]-p3[0])) - ((p4[1]-point[1])/(p4[0]-point[0])) <=0 and point[0]<=p4[0]:
        #right
        if (round(p2[0]) == round(p4[0]) and point[0] <= round(p2[0])) or (((p4[1]-p2[1])/(p4[0]-p2[0])) - ((p4[1]-point[1])/(p4[0]-point[0])) >=0):
          return True

  return False
```

```
def overlay_without_homography(original_frame, movie_frame, corners):
  copy = original_frame.copy()
  indices = np.array([[j,k] for k in range(original_frame.shape[0]) for j in range(original_frame.shape[1])])
  boolean = [check_if_inside(corners,x) for x in indices]
  filtered_indices = np.flip(indices[boolean],axis=1)
  for index in filtered_indices:
    copy[index[0]][index[1]] = movie_frame[index[0]][index[1]]
  return copy
```

```
sift_image1, keypoints1, descriptors1 = sift(cover)
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
frames_stack = []
for i in tqdm(range(original_frames.shape[0])):
  if i < movie_frames.shape[0]:
    sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
    matches = key_matching(descriptors1, descriptors2,nkp=300)
    image1_points , image2_points = get_kp(matches)
    A =constructA(image1_points,image2_points,start=0,end=15)
    H = get_homography(A)
    new_corners = get_new_points(corners, H)
    centered_frame = center_frames(movie_frames[i],original_frames[i].shape,new_corners)
    final_frame = overlay_without_homography(original_frames[i],centered_frame,new_corners)
  else:
    final_frame = original_frames[i]
  frames_stack.append(final_frame)
```

```
100%|██████████| 641/641 [52:57<00:00,  4.96s/it]
```

```
write_video("/content/drive/MyDrive/asg2_cv/output_without_homography.mov",frames_stack,fps1)
```

We also tried another solution which is scaling the movie frame to make sure its height matches the book cover image and we copped the movie frame (from its center) by the width of the book cover image and then applied homography to each pixel and put its value in the new position in the original video frames then we saved the output video

Code:

```
def crop_center(image, reference):
  img_copy = image.copy()
  img_copy = cv2.resize(img_copy, (round(img_copy.shape[1]*reference.shape[0]/img_copy.shape[0]),reference.shape[0]))
  center = (img_copy.shape[0]//2,img_copy.shape[1]//2)
  img_copy  = img_copy[:,center[1]-reference.shape[1]//2:center[1]+reference.shape[1]//2]
  return img_copy
```

```
sift_image1, keypoints1, descriptors1 = sift(cover)
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
frames_stack = []
for i in tqdm(range(original_frames.shape[0])):
  if i < movie_frames.shape[0]:
    sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
    matches = key_matching(descriptors1, descriptors2,nkp=300)
    image1_points , image2_points = get_kp(matches)
    A =constructA(image1_points,image2_points,start=0,end=15)
    H = get_homography(A)
    final_frame = overlay(crop_center(movie_frames[i],cover),original_frames[i],H)
  else:
    final_frame = original_frames[i]
  frames_stack.append(final_frame)
```

```
100%|██████████| 641/641 [30:29<00:00,  2.85s/it]
```

```
write_video("/content/drive/MyDrive/asg2_cv/output_with_homography.mov",frames_stack,fps1)
```

We also tried the previous two methods but this time after cropping the black parts from the movie to obtain better finished output.

Code:

```
cv2_imshow(movie_frames[0])
```

```
cv2_imshow(movie_frames[0][44:317])
```

```
sift_image1, keypoints1, descriptors1 = sift(cover)
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
frames_stack = []
for i in tqdm(range(original_frames.shape[0])):
  if i < movie_frames.shape[0]:
    sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
    matches = key_matching(descriptors1, descriptors2,nkp=300)
    image1_points , image2_points = get_kp(matches)
    A =constructA(image1_points,image2_points,start=0,end=15)
    H = get_homography(A)
    new_corners = get_new_points(corners, H)
    centered_frame = center_frames(movie_frames[i][44:317],original_frames[i].shape,new_corners)
    final_frame = overlay_without_homography(original_frames[i],centered_frame,new_corners)
  else:
    final_frame = original_frames[i]
  frames_stack.append(final_frame)
```

```
100%|██████████| 641/641 [52:45<00:00,  4.94s/it]
```

```
write_video("/content/drive/MyDrive/asg2_cv/output_without_homography_cropped.mov",frames_stack,fps1)
```

Next step we apply cropping and homography to the whole video

Code:

```
sift_image1, keypoints1, descriptors1 = sift(cover)
corners = [[0,0],[cover.shape[1]-1,0],[0,cover.shape[0]-1],[cover.shape[1]-1,cover.shape[0]-1]]
frames_stack = []
for i in tqdm(range(original_frames.shape[0])):
  if i < movie_frames.shape[0]:
    sift_image2, keypoints2, descriptors2 = sift(original_frames[i])
    matches = key_matching(descriptors1, descriptors2,nkp=300)
    image1_points , image2_points = get_kp(matches)
    A =constructA(image1_points,image2_points,start=0,end=15)
    H = get_homography(A)
    final_frame = overlay(crop_center(movie_frames[i][44:317],cover),original_frames[i],H)
  else:
    final_frame = original_frames[i]
  frames_stack.append(final_frame)
```

```
100%|██████████| 641/641 [30:22<00:00,  2.84s/it]
```

```
write_video("/content/drive/MyDrive/asg2_cv/output_with_homography_cropped.mov",frames_stack,fps1)
```

Links:

Folder having the output videos:
https://drive.google.com/drive/folders/1oev3rrYDGfnDaJkT60e3E_nYuLBoXG0x?usp=share_link

Colab:
https://colab.research.google.com/drive/19VX7kDDZFRoi-qqfUHvah7-PXt4c3-jA?usp=sharing

Part 2:Image Mosaics

First we calculate keypoints ,descriptors using sift and we plot them just like the last part. After that we calculated the correspondences using brute force matcher from opencv and drew 50 correspondences. Next step we calculated the homography matrix by getting keypoints and calculating A matrix. We checked our homography function with the built-in function.

All of the functions mentioned were the same functions as the ones used in part 1.

After that we do padding to the image to be apple to plot the 2 images in 1 image

Code:

```
def padding(image1,image2,H):
  indices = [[i,j] for i in range(image1.shape[1]) for j in range(image1.shape[0])]
  new_points = get_new_points(indices,H)
  ptop = math.ceil(abs(min(new_points[:,1])))
  pbottom = math.ceil(max(max(new_points[:,1])-image2.shape[0],0))
  pright = math.ceil(max(max(new_points[:,0])-image1.shape[1],0))
  padding_right = np.zeros((image2.shape[0],pright,3))
  extended_image = image2.copy()
  extended_image = np.hstack((extended_image, padding_right))
  padding_top = np.zeros((ptop,extended_image.shape[1],3))
  extended_image = np.vstack((padding_top,extended_image))
  padding_bottom = np.zeros((pbottom,extended_image.shape[1],3))
  extended_image = np.vstack((extended_image,padding_bottom))
  return extended_image, ptop
```

Output:

After that we perform forward wrapping. This is done by multiplying the pixel positions by the homography matrix just like in part 1. However, this time around, we don't round. We split the values between two pixels and if a pixel already has value we average both values.

Code:

```
def overlay(overlay_image, original_image, H,ptop):
  indices = [[i,j] for i in range(overlay_image.shape[1]) for j in range(overlay_image.shape[0])]
  new_points = get_new_points(indices,H)
  copy_image = original_image.copy()
  for i,(x,y) in enumerate(new_points):
    [x2,y2] = indices[i]
    combinations = [#[math.ceil(x),math.ceil(y)],
                    [math.ceil(x),math.floor(y)],
                    #[math.floor(x),math.ceil(y)],
                    [math.floor(x),math.floor(y)]]
    for u,v in combinations:
      if (u < original_image.shape[1] and v+ptop < original_image.shape[0] and u > 0 and v+ptop > 0):
        if sum(copy_image[v+ptop,u]) == 0 and sum(original_image[v+ptop,u])==0:
          copy_image[v+ptop,u] = overlay_image[y2,x2]
        elif sum(original_image[v+ptop,u])==0:
          copy_image[v+ptop,u] = (copy_image[v+ptop,u]+overlay_image[y2,x2])//2
  return copy_image
```

Output:

After that we perform backward wrapping. This is done by multplying the black pixels indices by the inverse of the homography function and then we get the four possible candidates and apply bilinear interpolation to calculate the pixel new value. The weight of each pixel will be inversely proportional to the distance between the x and y coordinates of the pixel and the x and y resulted from the homography matrix.

Code:

```
def backward_wrap(original_image, new_image, H,ptop):
  final_image = new_image.copy()
  indices = [[i,j] for i in range(0,final_image.shape[1]) for j in range(final_image.shape[0])]
  copy_indices = np.array(indices.copy())
  copy_indices[:,1] = np.array(indices)[:,1] - ptop
  new_points = get_new_points(copy_indices.tolist(),np.linalg.inv(H))
  for i,(x,y) in enumerate(new_points):
    [x0,y0] = indices[i]
    if sum(final_image[y0,x0])==0 and sum(new_image[y0,x0])==0:
      combinations = [[math.ceil(x),math.ceil(y)],
                  [math.ceil(x),math.floor(y)],
                  [math.floor(x),math.ceil(y)],
                  [math.floor(x),math.floor(y)]]
      for u, v in combinations:
        if u < original_image.shape[1] and v < original_image.shape[0] and u > 0 and v > 0:
          final_image[y0,x0]  = final_image[y0,x0] + abs( (1-abs(u-x)) *(1-abs(v-y)) ) *original_image[v,u]
    final_image[y0,x0] = np.round(final_image[y0,x0]).astype('int64')
  return final_image
```

Output:

Another Example:

Forward wrapping

Code:

```
image1 = cv2.imread("/content/drive/MyDrive/asg2_cv/extra2.jpg")
image2 = cv2.imread("/content/drive/MyDrive/asg2_cv/extra1.jpg")
sift_image1, keypoints1, descriptors1 = sift(image1)
sift_image2, keypoints2, descriptors2 = sift(image2)
n_points = 20
matches = key_matching(descriptors1, descriptors2,nkp=300)[0:n_points]
image1_points , image2_points = get_kp(matches)
A =constructA(image1_points,image2_points,start=0,end=n_points)
H = get_homography(A)
extended_image,ptop = padding(image1,image2,H)
forward_frame = overlay(image1,extended_image,H,ptop)
cv2_imshow(forward_frame)
```

Output:

Next step we do backward wrapping

Output:

Bonus Part:

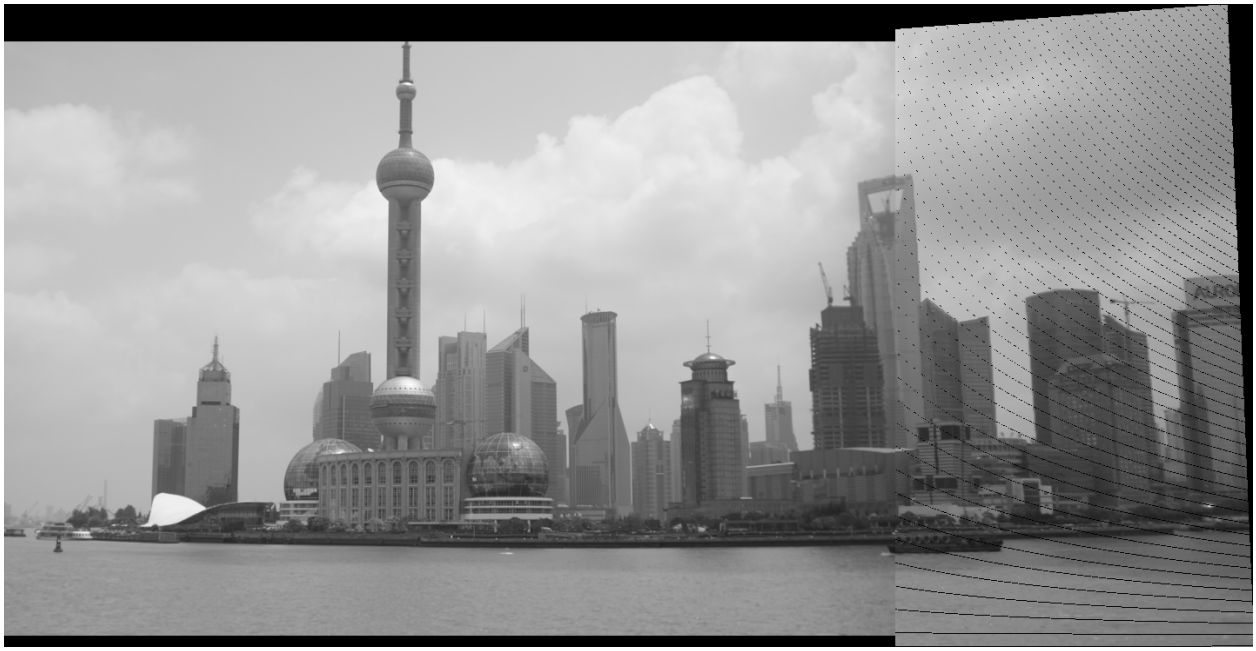
We do the same as the above part to combine 2 images. Then we do overlay to the 3rd image.

Code:

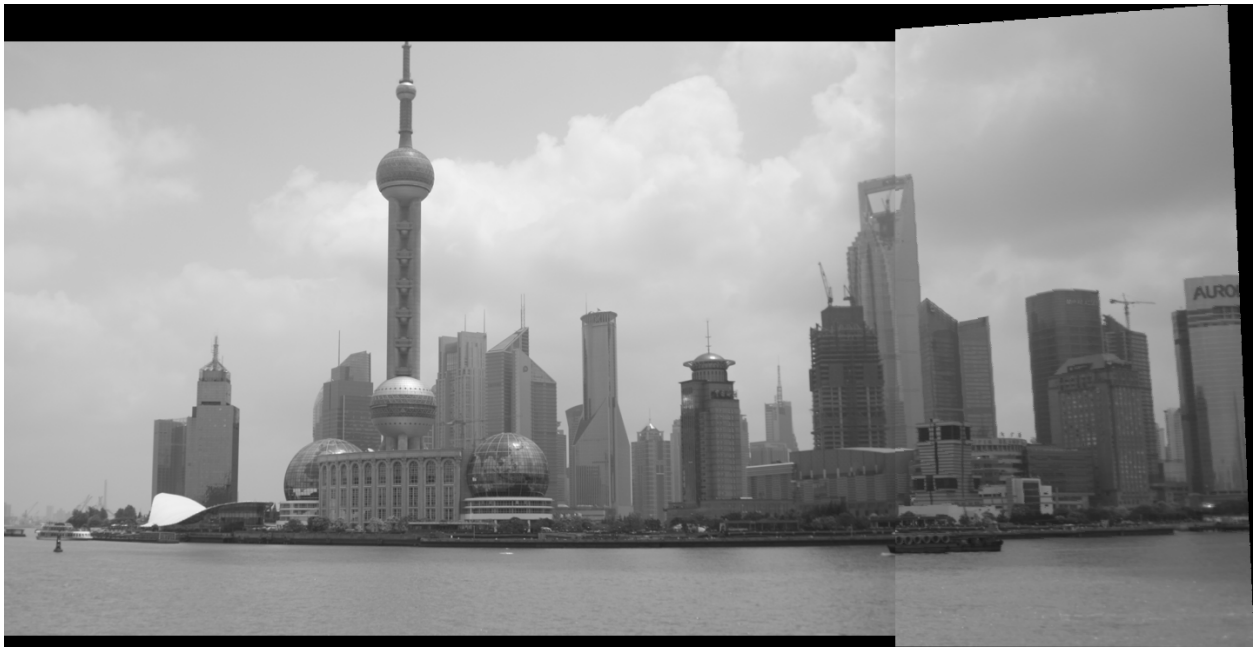
```
#for bonus example
def overlay2(overlay_image, original_image, H,border,ptop):
  indices = [[i,j] for i in range(overlay_image.shape[1]) for j in range(overlay_image.shape[0])]
  new_points = get_new_points(indices,H)
  copy_image = original_image.copy()
  for i,(x,y) in enumerate(new_points):
    [x2,y2] = indices[i]
    combinations = [#[math.ceil(x),math.ceil(y)],
                    [math.ceil(x),math.floor(y)],
                    #[math.floor(x),math.ceil(y)],
                    [math.floor(x),math.floor(y)]]
    for u,v in combinations:
      if (u < original_image.shape[1] and v+ptop < original_image.shape[0] and u > 0 and v+ptop > 0):
        if sum(copy_image[v+ptop,u]) == 0 and u >= border:
          copy_image[v+ptop,u] = overlay_image[y2,x2]
        elif u>=border:
          copy_image[v+ptop,u] = (copy_image[v+ptop,u]+overlay_image[y2,x2])//2
  return copy_image
```

Outputs:

2 images forward wrapping:



2 images after backward wrapping:



Final Result (putting the 3 images together):

Link :
https://colab.research.google.com/drive/1adCGAZZCP34pg7GjkLlbToAdvTQXsB3M?usp=sharing