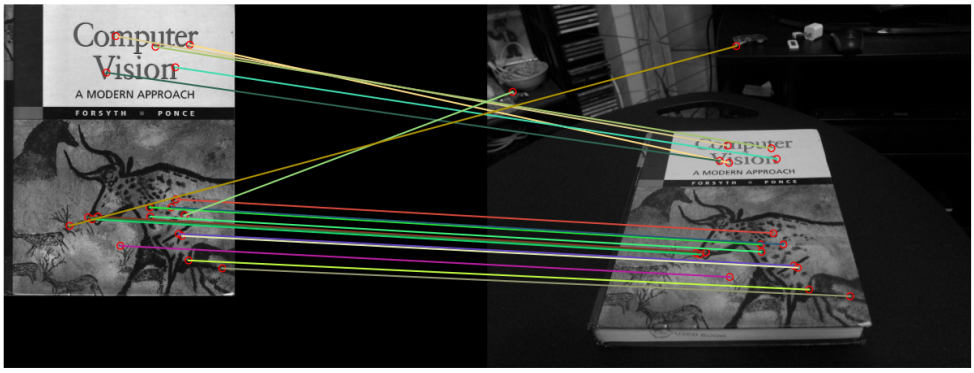
4.1



4.2

```
def computeH(x1, x2):
 #Section 4.2
 #Compute the homography between two sets of points

 # create A
 A = []
 for i in range(x1.shape[0]):
   current_x1 = x1[i]
   first_x, first_y = current_x1
   current_x2 = x2[i]
   second_x, second_y = current_x2

   A.append([-second_x, -second_y, -1, 0, 0, 0, second_x*first_x,
second_y*first_x, first_x])
   A.append([0, 0, 0, -second_x, -second_y, -1, second_x*first_y,
second_y*first_y, first_y])

 # calculate svd
 u, s, vh = np.linalg.svd(A, False)

 H = vh[-1, :] / vh[-1, -1]
```

```
 H = np.reshape(H, (3, 3))
 return H
```

The homography is computed using Ax=0, and selecting the smallest eigenvector to be chosen as the homography. The homography is then normalize to make the last entry (3, 3) to be 1 since the homography matrix only has 8 degrees of freedom and the last entry can be treated as 1.

4.3

```
def computeH_norm(x1, x2):
 #Section 4.3
 #Compute the centroid of the points
 mean_x1 = np.mean(x1, 0)
 mean_x2 = np.mean(x2, 0)

 # Shift the origin of the points to the centroid
 centered_x1 = x1 - mean_x1
 centered_x2 = x2 - mean_x2

 # Normalize the points so that the average distance from the origin is
equal to sqrt(2)
 scale_x1 = np.sqrt(2) / np.mean(np.sqrt(np.sum(centered_x1**2, 1)))
 scale_x2 = np.sqrt(2) / np.mean(np.sqrt(np.sum(centered_x2**2, 1)))

 # Similarity transform 1
 t1 = scale_x1 * np.array([[1, 0, -mean_x1[0]], [0, 1, -mean_x1[1]], [0,
0, 1/scale_x1]])
 normalized_x1 = scale_x1 * centered_x1

 # Similarity transform 2
 t2 = scale_x2 * np.array([[1, 0, -mean_x2[0]], [0, 1, -mean_x2[1]], [0,
0, 1/scale_x2]])
 normalized_x2 = scale_x2 * centered_x2

 # Compute homography
 H2to1 = computeH(normalized_x1, normalized_x2)

 # get the denormalized H
 H2to1 = np.linalg.inv(t1) @ H2to1 @ t2
 return H2to1
```

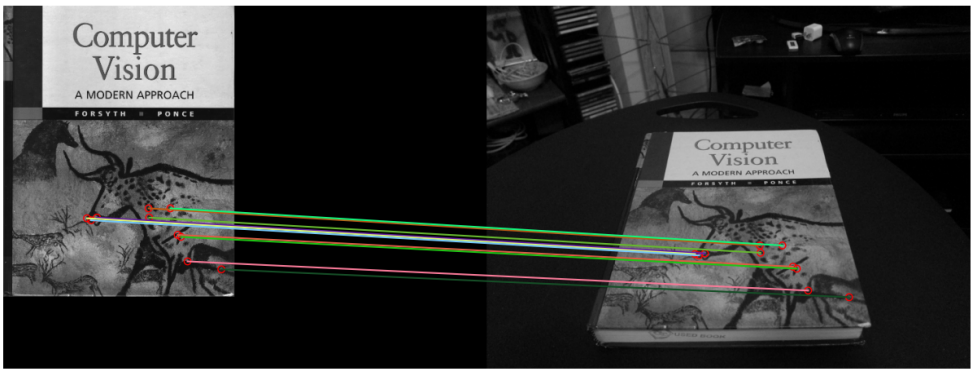
The points are normalized to have a 0 mean and average distance to the origin is $\sqrt{2}$. The transformation for the coordinates in image 1 and image 2 are kept so that after calculating the homography of the normalized coordinates, we can denormalize the normalized coordinates back to the image coordinates by using the equation: $T_1^{-1} H T_2$.

4.4

```
def computeH_ransac(locs1, locs2):
 #Section 4.4
 # RANSAC loop
 threshold = 5
 inliers = []
 bestH2to1 = None
 for i in range(4000):
   rand_points = np.random.choice(range(locs1.shape[0]), 4, replace=False)
   selected_loc1 = locs1[rand_points]
   selected_loc2 = locs2[rand_points]

   H2to1 = computeH(selected_loc1, selected_loc2)
   h_locs2 = np.ones((locs2.shape[0], 3))
   h_locs2[:, 0:2] = locs2
   result1 = np.matmul(H2to1, h_locs2.T)
   result1 = result1.T
   result1 = result1[:, 0:2] / result1[:, 2:3]

   error = np.linalg.norm(result1 - locs1, axis=1)

   current_inliers_indexes = np.where(error < threshold)[0]
   if current_inliers_indexes.shape[0] > len(inliers):
     inliers = current_inliers_indexes
     bestH2to1 = H2to1

 # recompute
 h_locs2 = np.ones((locs2.shape[0], 3))
 h_locs2[:, 0:2] = locs2
 result1 = np.matmul(bestH2to1, h_locs2.T)
 result1 = result1.T
 result1 = result1[:, 0:2] / result1[:, 2:3]

 error = np.linalg.norm(result1 - locs1, axis=1)
```

```
 inliers = [locs1[np.where(error < threshold)[0]], locs2[np.where(error <
threshold)[0]]]

 bestH2to1 = computeH_norm(inliers[0], inliers[1])
 return bestH2to1, inliers
```

The RANSAC is ran with 4000 loops, and for each iteration 4 correspondences are randomly selected and used to calculate the homography to transform the book from image 2 to image 1. There is a threshold to determine if an inlier is close to the actual point in the image 1, we will keep the inlier.
After the loop is done, there should be enough inliers to be used to calculate a more accurate homography to transform the book from image 2 to image 1. We will use the normalized homography to compute the all the inliers because there are more than 4 correspondence points. This is to improve the stabilization of the calculation of the homography. The resulting inliers can be seen below:

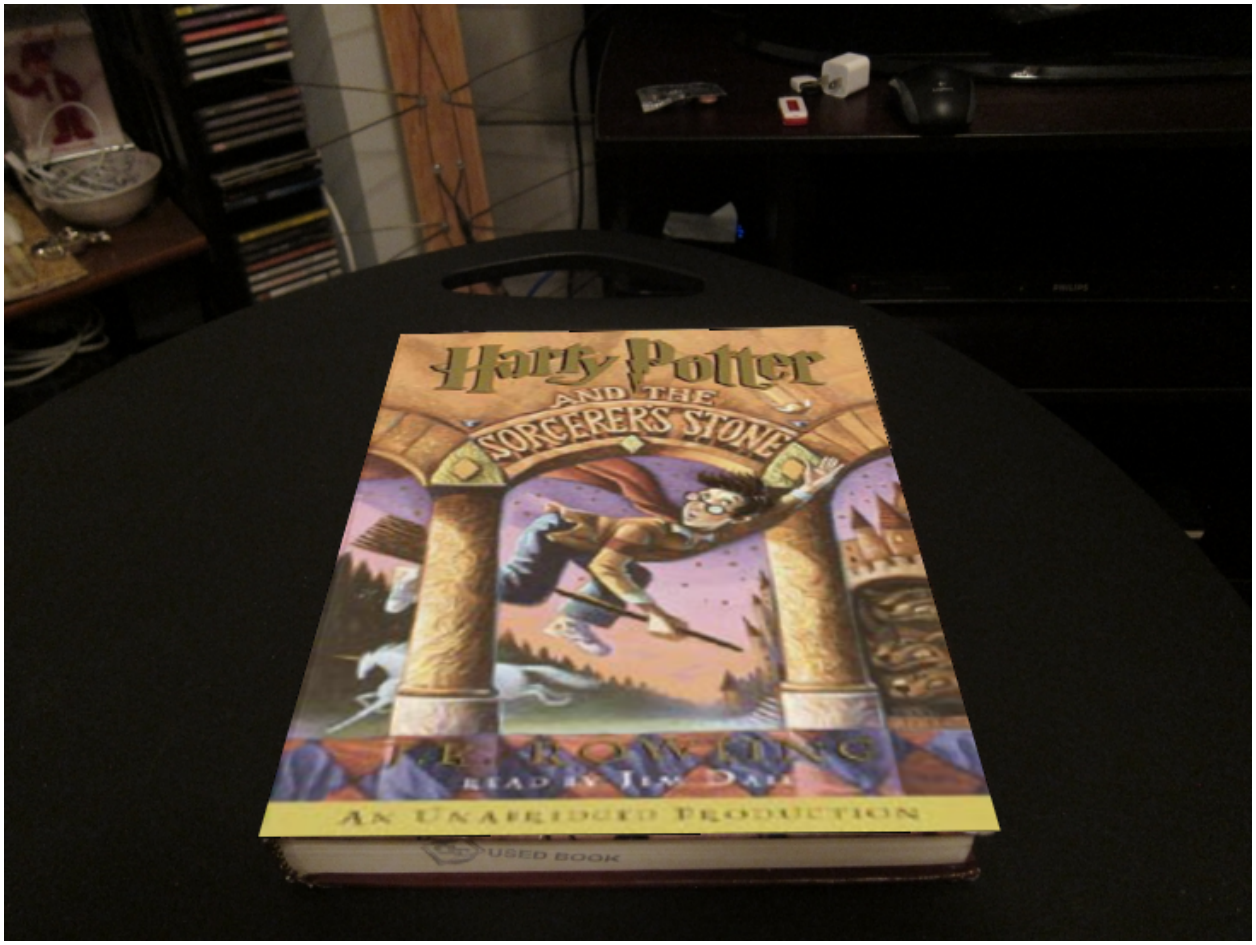


4.5

```
def compositeH(H2to1, template, img):
 #Note that the homography we compute is from the image to the template;
 #x_template = H2to1*x_photo
 #For warping the template to the image, we need to invert it using
np.linalg.inv
 inv_H2to1 = np.linalg.inv(H2to1)

 #Warp template by appropriate homography
 # you can use cv2.warpPerspective
```

```
 warped_template = cv2.warpPerspective(template, H2to1, (img.shape[1],
img.shape[0]))

 #Use mask to combine the warped template and the image
 composite_img = img.copy()
 for i in range(warped_template.shape[0]):
   for j in range(warped_template.shape[1]):
     if (warped_template[i][j] == [255, 255, 255]).all() or
(warped_template[i][j] == [0, 0, 0]).all():
       continue
     composite_img[i, j] = warped_template[i][j]
 return composite_img
```

The composite image is calculated by replacing all black pixels as the cv_desk.png image. For none black pixels, they are the transformed harry potter book so we will not replace them.

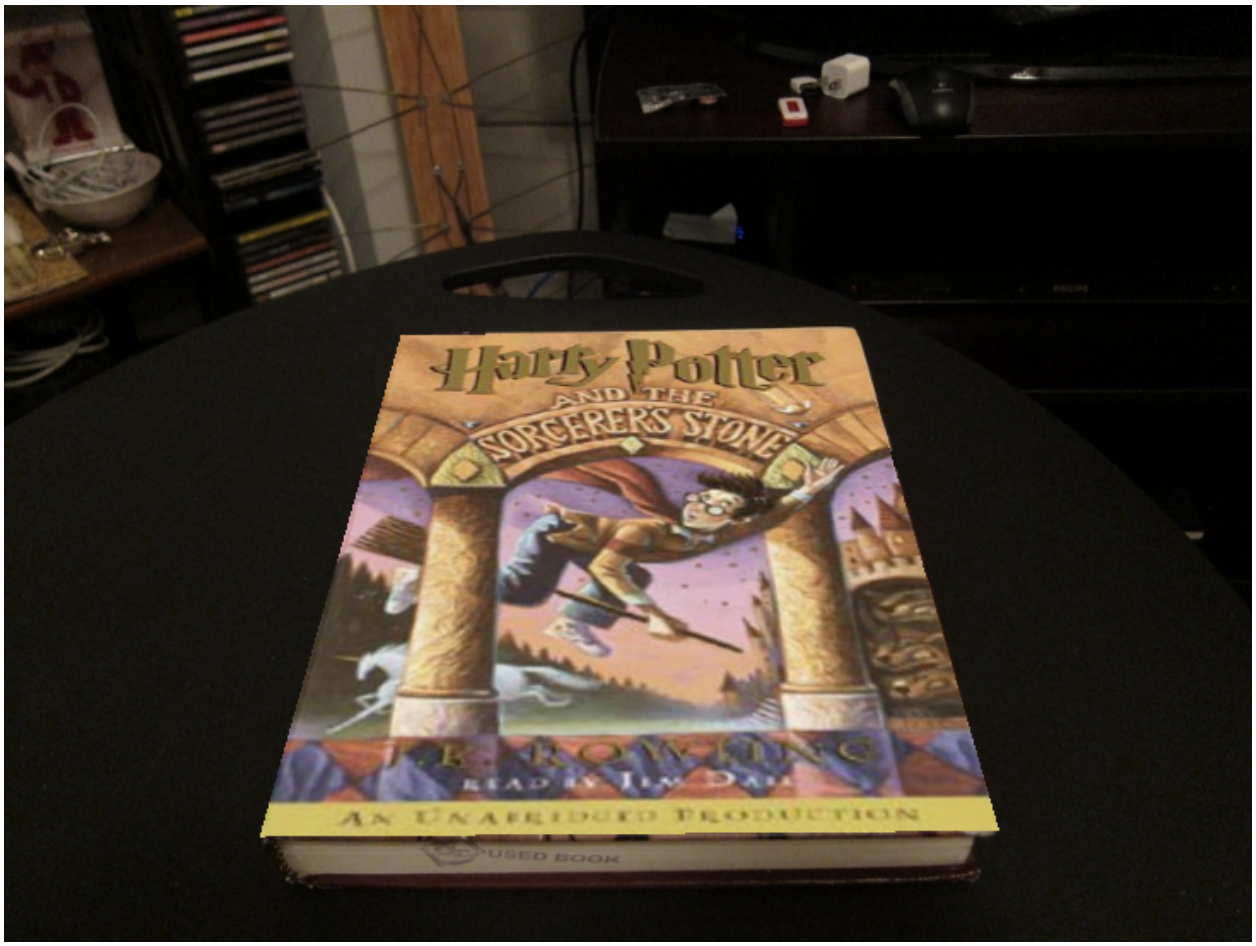


4.6

```
def warpingH(src, h, output_size):
 # implement your own warping function to replace cv2.warpPerspective in
compositeH

 # since warpPerspective receive upside down output_size, we convert the
rows and cols first
 height, width, channel = src.shape
 mtr = np.zeros((width, height, channel), dtype='int')
 for i in range(src.shape[0]):
   mtr[:, i] = src[i]

 row, col = output_size
 dst = np.zeros((row, col, channel))

 # to do proper bilinear interpolation, we will invert H and calculate the
mapping from destination picture to source
 h = np.linalg.inv(h)
 for i in range(dst.shape[0]):
   for j in range(dst.shape[1]):
     res = np.dot(h, [i, j, 1])
     i2, j2, _ = res / res[2]

     if np.floor(i2) < 0 or np.floor(j2) < 0:
       continue

     # do bilinear interpolation
     if 0 <= np.ceil(i2) < width:
       if 0 <= np.ceil(j2) < height:
         i2_decimal = i2 % 1
         j2_decimal = j2 % 1
         i2_lower = np.floor(i2).astype(np.int)
         i2_upper = np.ceil(i2).astype(np.int)
         j2_lower = np.floor(j2).astype(np.int)
         j2_upper = np.ceil(j2).astype(np.int)

         dst[i, j] = (1-i2_decimal) * (1-j2_decimal) * mtr[i2_lower,
j2_lower] + \
```

```
        (i2_decimal) * (1-j2_decimal) * mtr[i2_upper, j2_lower] + \
        (i2_decimal) * (j2_decimal) * mtr[i2_upper, j2_upper] + \
              (1-i2_decimal) * (j2_decimal) * mtr[i2_lower, j2_upper]

# convert from matrix back to image format
width, height, channel = dst.shape
img = np.zeros((height, width, channel), dtype='int')
for i in range(dst.shape[0]):
  img[:, i] = dst[i]

return img
```

Since the homography maps the harry potter book to the desk, to do proper bilinear interpolation, we would need to get the pixel from the desk image to map to the harry potter book. The homography is inverted in order to achieve this. The result comparison between the implementation of warpingH and OpenCV warpPerspective is quite similar, there is just some minor horizontal line that was not straight in the implementation of warpingH if we see from the top of the horizontal line of the harry potter book and the bottom of the horizontal line of the book.


Harry Potter
AND THE
SORCERER'S STONE
AN UNABRIDGED PRODUCTION
USED BOOK