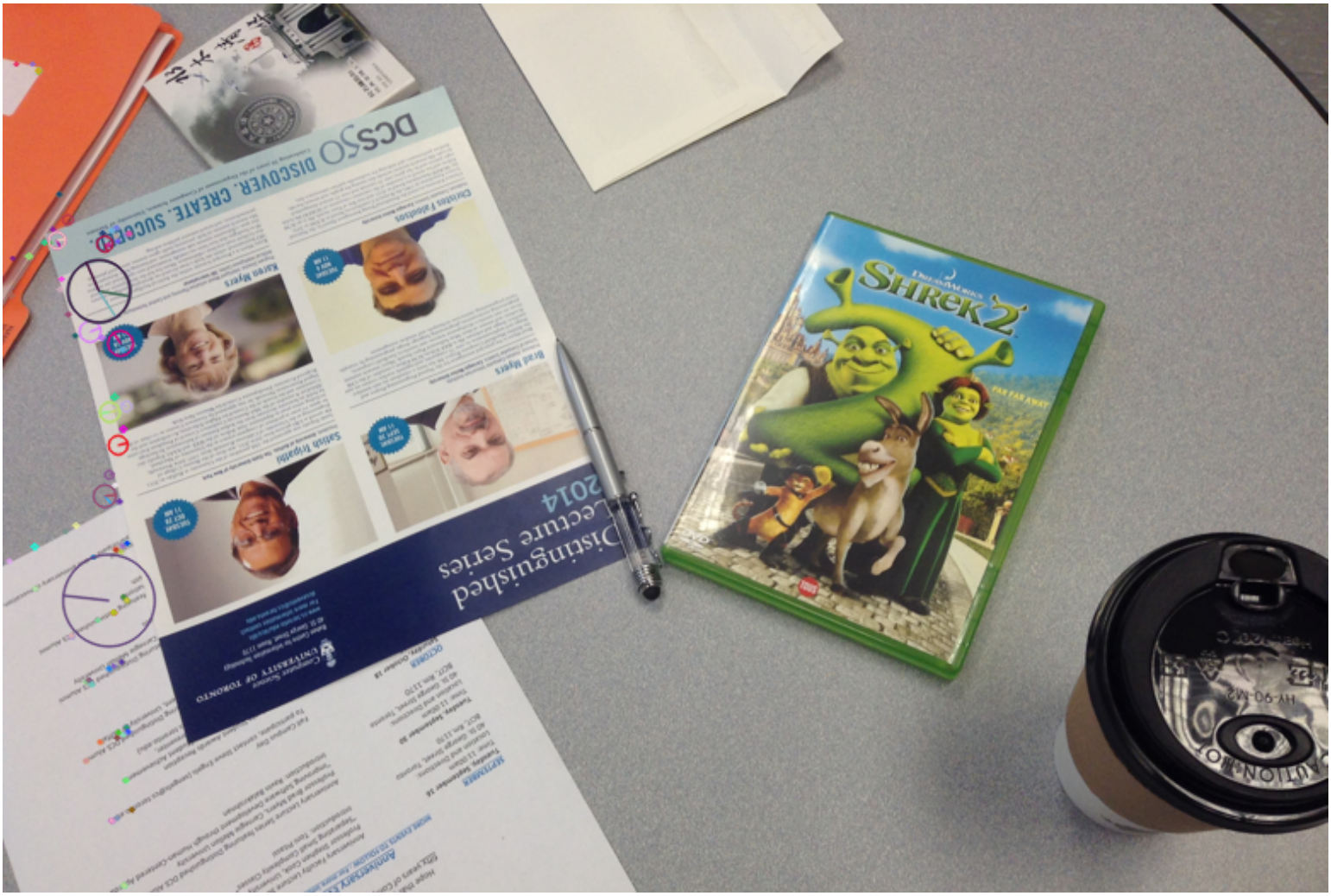
CSC420 - Assignment 3

1. a) Feature Extraction

* Code can be found in SIFT_matching.py — feature_extraction()

Visualization of the first 100 SIFT keypoints of reference.png, test.png, and test2.png

reference.png


DCSSO DISCOVER. CREATE. SUCCEED.
Distinguished Lecture Series
2014
SHREK 2

test.png


SHREK 2
LES INDESTRUCTIBLES
PAR LES CRÉATEURS DU MONDE DE NEMO

test2.png

1. b) Matching

* Code can be found in SIFT_matching.py — matching()

I define the ‘best’ match to be a match where the threshold ratio is the smallest. This means that for all matching keypoints i and j between images A and B, the best match would be the match that has the smallest threshold ratio.

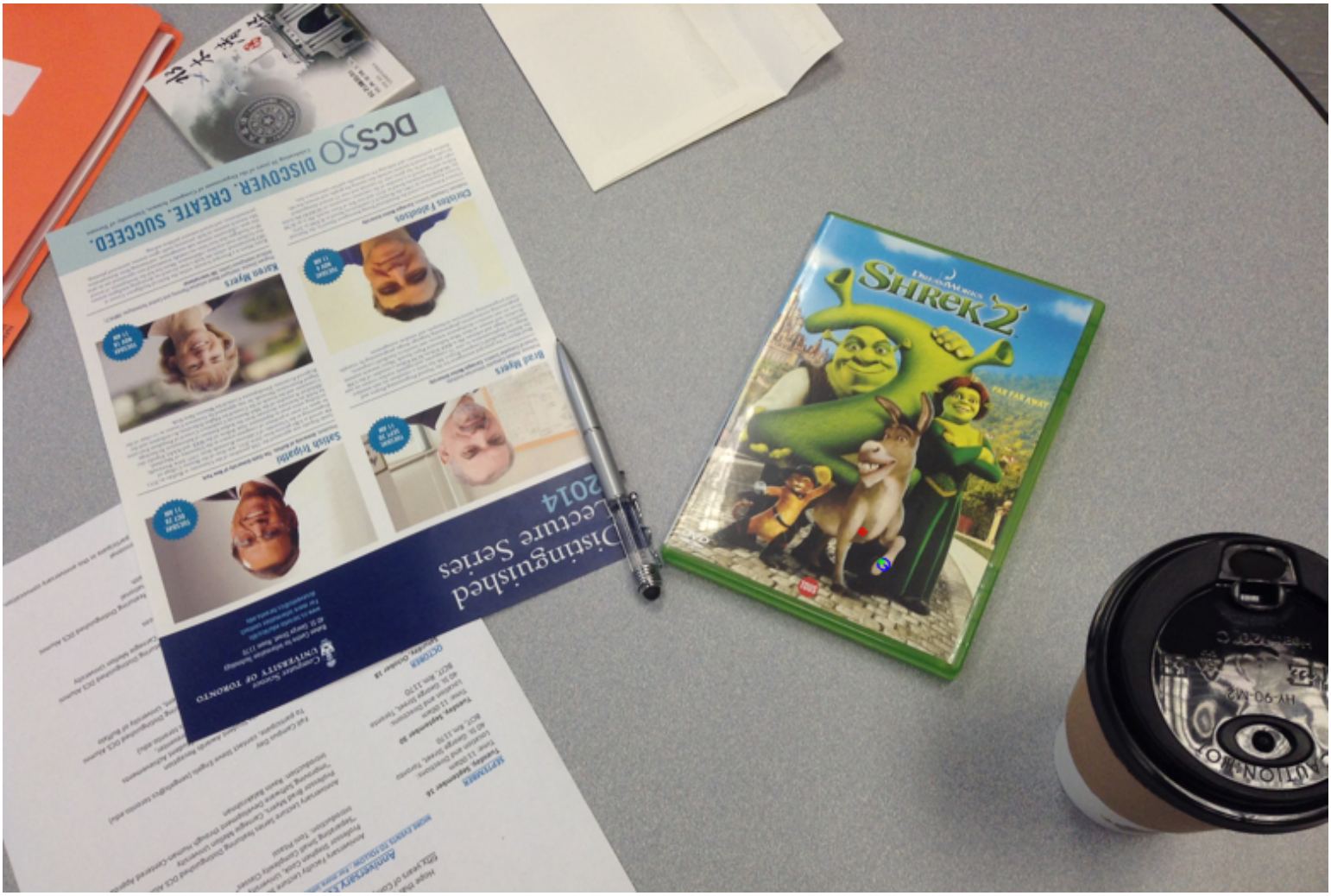
$$\phi_i = \frac{|f_i - f_j|}{|f_i - f_k|}$$

keypoints (i, j) is the ‘best’ match if psi is the smallest for all j, k on i

A matching algorithm is as follows:

- for each keypoint i in image A
  - find i’s matching keypoint j in image B
  - calculate the threshold ratio of match (i,j)
  - if threshold ratio is one the the 3 smallest ratios encountered
    - save the matching pair (i,j) as one of the 3 ‘best’ matches
- return the 3 best keypoint matches

3 best keypoint matches between reference and test



3 best keypoint matches between reference and test

3 best keypoint matches between reference and test2

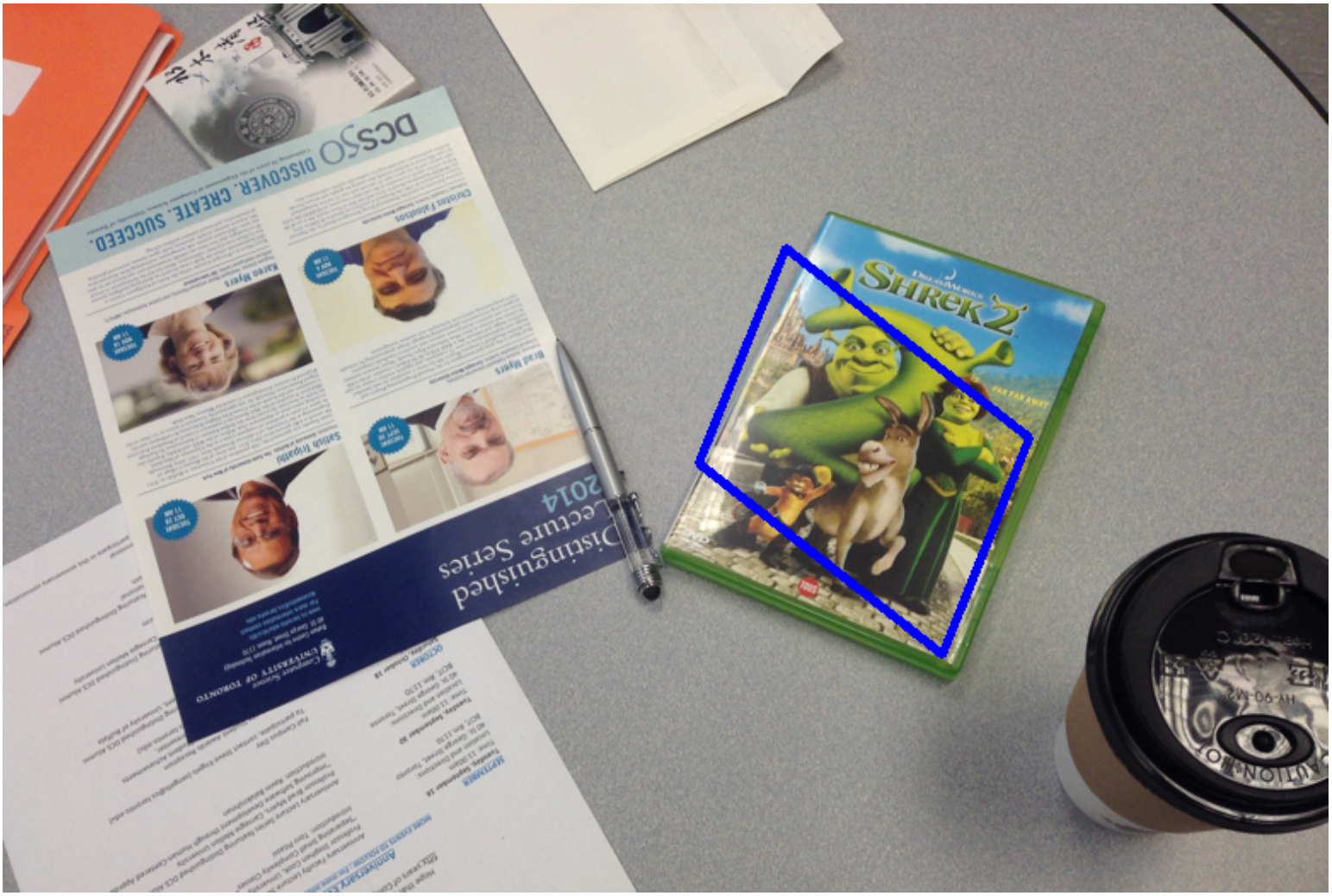
3 best keypoint matches between reference and test2

1. c) Calculating Affine Transformation

* Code can be found in affine_transform.py — calculate_affine_transform()

1. d) Visualizing Affine Transformation

* Code can be found in affine_transform.py — visualize_affine_transform()



Affine Transformation from reference to test

This affine transform is inaccurate. This may be because the object is close to the camera and so the affine transform is not a good approximation of the viewpoint change. Or maybe the 3 matching keypoints used to calculate the affine transform are not accurate of the transform of the viewpoint. A RANASAC method may be better.

Affine Transformation from reference to test2

2. Door Homography

* Code used to calculate homography H can be found in door.py

We can estimate the dimensions of the door by calculating a homography transform from from the pixel positions in the picture to the real life plane of the door. (The real life plane can use an arbitrarily determined coordinate system) After determining the homography between the picture and the plane of the wall we can transform the pixel coordinates of the wall to the coordinates in the plane of the door using the matrix H.

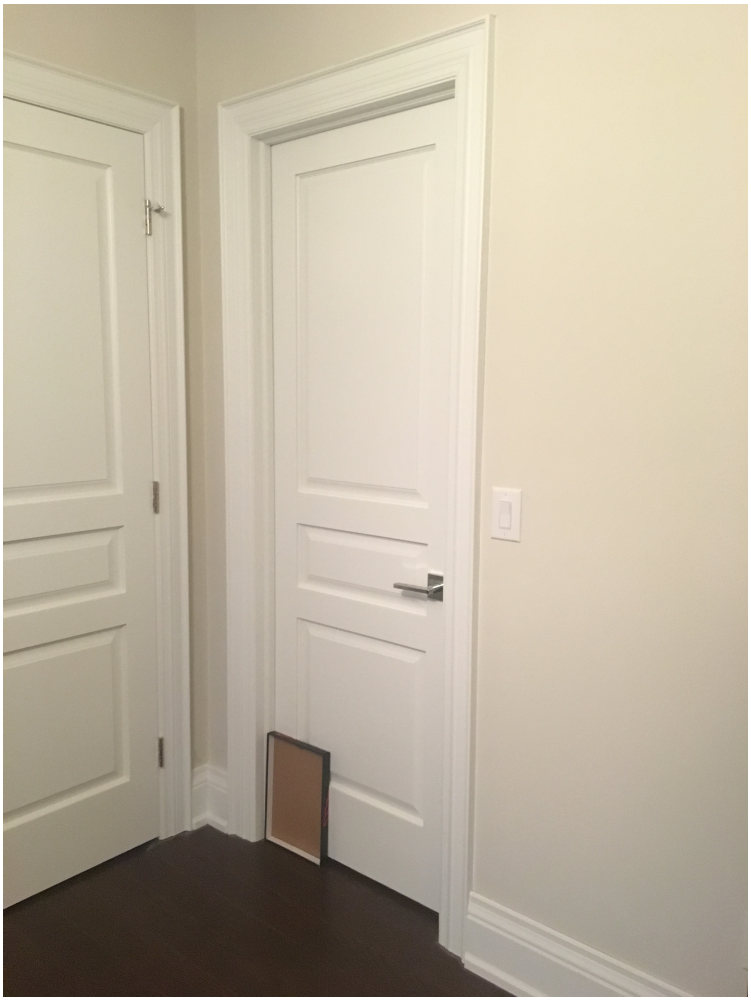
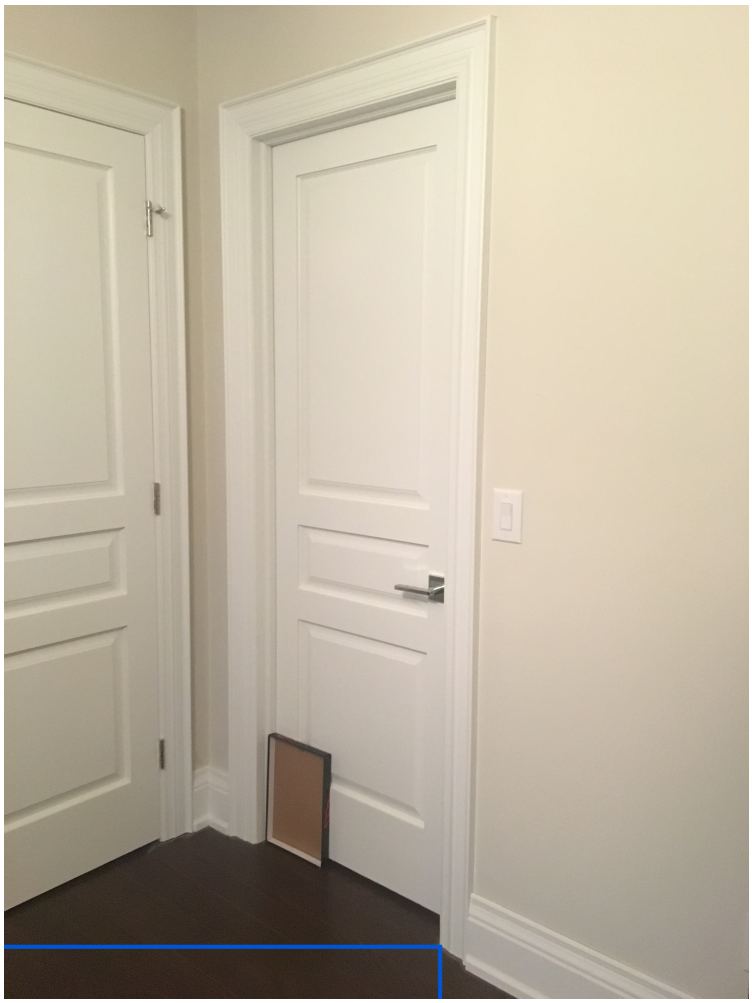


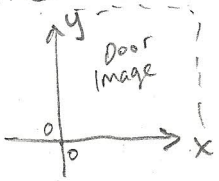
Image of door and shoe lid

The pictures on the right show how some pixel positions were measured

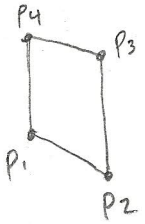


In 2D picture

The coordinates I use are



Coordinates of box-lid (in px)



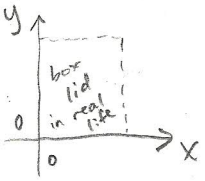
$P_1 = (1062, 635)$
$P_2 = (1285, 531)$
$P_3 = (1300, 981)$
$P_4 = (1072, 1072)$

Coordinates of door (in px)

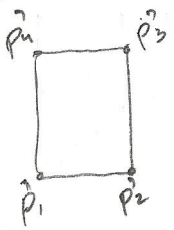
$q_1 = (1025, 630)$
$q_2 = (1768, 222)$
$q_3 = (1835, 3726)$
$q_4 = (1005, 3494)$

In Real Life Plane (Wall)

The coordinates I use are



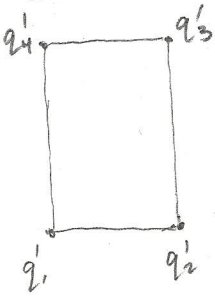
Coordinates of box-lid (in cm)



$P_1' = (0, 0)$
$P_2' = (24.2, 0)$
$P_3' = (24.2, 35.5)$
$P_4' = (0, 35.5)$

H

Coordinates of door (in cm)



$q_1' = ?$
$q_2' = ?$
$q_3' = ?$
$q_4' = ?$

* may include negative coordinates

The matrix A is then...

$$A = \begin{bmatrix}
1062 & 635 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1062 & 635 & 1 & 0 & 0 & 0 \\
1285 & 531 & 1 & 0 & 0 & 0 & -31097 & -12850.2 & -24.2 \\
0 & 0 & 0 & 1285 & 531 & 1 & 0 & 0 & 0 \\
1300 & 981 & 1 & 0 & 0 & 0 & -31460 & -23740.2 & -24.2 \\
0 & 0 & 0 & 1300 & 981 & 1 & -46150 & -34825.5 & -35.5 \\
1072 & 1072 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1072 & 1072 & 1 & -38056 & -38056 & -35.5
\end{bmatrix}$$

By finding the smallest eigenvalue of A^TA we get the eigenvector h

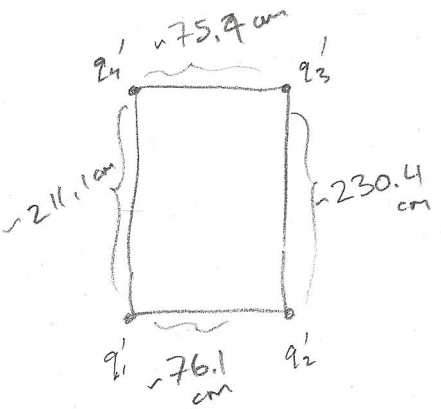
$$\Rightarrow \quad H = \begin{bmatrix}
-7.47\times10^{-4} & 1.71\times10^{-5} & 7.82\times10^{-1} \\
-2.57\times10^{-4} & -5.51\times10^{-4} & 6.23\times10^{-1} \\
-1.16\times10^{-6} & -2.84\times10^{-7} & -5.31\times10^{-3}
\end{bmatrix}$$

Next we transform each point of the door in the picture into a position on the real lif plane using H.

↳ * some transformed points may be negative.

The transformed points are

$$q_1' = (-4.1, -1.8)$$
$$q_2' = (71.9, -6.2)$$
$$q_3' = (61.7, 224.0)$$
$$q_4' = (-12.2, 209.1)$$



⇒ by calculating distances we can guess the width + height of the door.

$w \simeq 75.75$ cm
$h \simeq 220.75$ cm

inaccuracies may be due to incorrect coordinates of the door image due to measurement error.

3. Panoramic Stitching

* Code can be found in panorama.py — stitch_images() (helper functions in homography.py)

The algorithm is straightforward and described in the lecture slides. However I will elaborate on the ‘stitching’ portion of the algorithm.

- First I calculated the dimensions of the stitched image.
- Next I found an offset value such that all points would be mapped to the stitched image. This is necessary because some transformed positions are negative and can not be mapped to images
- Next I transformed each point from the first image using the homography H. These positions are then shifted by the offset to prevent negative positions.
    - The mapping from old to new positions is used to apply a forward warp of the image onto the stitched image
- The second image is then shifted by the same offset and mapped to the new stitched image

While transforming points using the homography, some points are transformed to float values. Since image positions must be described by integer values I needed to round the transformed positions. This caused some sampling loss.

In order to solve this problem I applied a simple linear interpolation on any skipped pixels in the stitched image. The resulting stitched image is below:

Appendix - Code

SIFT_matching.py

```
import cv2
import numpy as np
from skimage import io
from skimage.color import rgb2gray

def extract_SIFT_keypoints(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create()
    kp, des = sift.detectAndCompute(gray, None)
    # kp is list of keypoints
    # des is a 2D numpy array with rows as descriptors of keypoints
    return kp, des

def feature_extraction(image_path):
    image = cv2.imread(image_path)
    kp, des = extract_SIFT_keypoints(image)
    image_keypoints = np.copy(image)
    cv2.drawKeypoints(image, kp[:100], image_keypoints,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.imwrite('keypoints.png', image_keypoints)

# Calculate the best feature match for the keypoint with descriptor (f)
# in the list of descriptors (descriptors)
# Returns closest matching descriptor in (descriptors)
# and the threshold ratio of the closest match
def calculate_correspondance(f, descriptors):

    # Calculate the euclidean distance between target_descriptor (f)
    # and all the search descriptors (descriptors)
    similarity = np.linalg.norm(f - descriptors, axis=1)

    # Find the closest and 2nd closest descriptor matches
    i = np.argsort(similarity)[0]   # Closest Match
    f_i = descriptors[i]

    j = np.argsort(similarity)[1]   # 2nd Closest Match
    f_j = descriptors[j]

    # Calculate threshold ratio
    thresh_ratio = similarity[i] / similarity[j]

    # Return the descriptor with the best match, its descriptor
    # index (i), and its threshold ratio
    return f_i, i, thresh_ratio
```

```
# Parameters:   two images to find matching keypoints on
# Output:       a list of the 3 best matches and their threshold ratios
#               'best' is defined as the matches with lowest threshold ratios
def get_best_matches(ref, test):
    kp_ref, des_ref = extract_SIFT_keypoints(ref)
    kp_test, des_test = extract_SIFT_keypoints(test)

    s_index_i = np.array([0, 0, 0])     # stores index of ref keypoint
    s_index_j = np.array([0, 0, 0])     # stores index of test keypoint
    s_ratios = np.array([1., 1., 1.])   # stores ratios between the ref and test keypoints

    # Find best matches to keypoint/descriptor (i) in reference image
    for i in range(len(kp_ref)):
        # get the keypoint match in test image corresponding to keypoint i
        f_j, j, thresh_ratio = calculate_correspondance(des_ref[i], des_test)

        # If current threshold ratio is smaller than one of the
        # currently stored ratios then replace that keypoint match
        s = np.argmax(s_ratios)
        if thresh_ratio < s_ratios[s]:
            s_index_i[s] = i
            s_index_j[s] = j
            s_ratios[s] = thresh_ratio

    # create list of keypoint matches in ref and test images
    kp_matches_i = [kp_ref[s_index_i[0]], kp_ref[s_index_i[1]], kp_ref[s_index_i[2]]]
    kp_matches_j = [kp_test[s_index_j[0]], kp_test[s_index_j[1]], kp_test[s_index_j[2]]]
    return kp_matches_i, kp_matches_j, s_ratios


def matching(reference, test):
    ref = cv2.imread(reference)
    test = cv2.imread(test)

    # FIND CLOSEST 3 MATCHES BEWTEEN BOTH IMAGES
    match_i, match_j, ratios = get_best_matches(ref, test)

    # DRAW MATCHED KEYPOINTS
    cv2.drawKeypoints(ref, [match_i[0]], ref,
color=(255,0,0),flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.drawKeypoints(ref, [match_i[1]], ref,
color=(0,255,0) ,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.drawKeypoints(ref, [match_i[2]], ref, color=(0,0,255),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    cv2.drawKeypoints(test, [match_j[0]], test, color=(255,0,0),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.drawKeypoints(test, [match_j[1]], test, color=(0,255,0),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    cv2.drawKeypoints(test, [match_j[2]], test, color=(0,0,255),
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    # OUTPUT IMAGES TO FILE
    cv2.imwrite('ref_keypoints.png', ref)
    cv2.imwrite('test_keypoints.png', test)
```

```
def main():
    test_path = 'images/test.png'
    test2_path = 'images/test2.png'
    reference_path = 'images/reference.png'

    # Q1 A)
    feature_extraction(test_path)
    feature_extraction(test2_path)
    feature_extraction(reference_path)

    # Q1 B)
    matching(reference_path, test_path)
    matching(reference_path, test2_path)
```

affine_transform.py

```
from SIFT_matching import *

def calculate_affine_transform(img1, img2):
    # FIND CLOSEST 3 MATCHES BEWTEEN BOTH IMAGES
    match_i, match_j, ratios = get_best_matches(img1, img2)
    n_keypoints = ratios.shape[0]

    P = np.empty((0,6))
    P_prime = np.empty((0))

    for k in range(n_keypoints):
        # GET KEYPOINT POSITIONS
        kp_i = match_i[k]
        kp_j = match_j[k]
        x_i = kp_i.pt[0]; y_i = kp_i.pt[1]
        x_j = kp_j.pt[0]; y_j = kp_j.pt[1]

        # CREATE PARTIAL P MATRIX
        p = np.array([
            [x_i, y_i, 0, 0, 1, 0],
            [0, 0, x_i, y_i, 0, 1]
        ])
        p_prime = np.array([
            x_j, y_j
        ])

        # APPEND TO P
        P = np.append(P, p, axis=0)
        P_prime = np.append(P_prime, p_prime)

    # CALCULATE AFFINE TRANSFORM MATRIX
    a = np.linalg.inv(P.T @ P) @ P.T @ P_prime
    A = np.array([
        [a[0], a[1], a[4]],
        [a[2], a[3], a[5]]
    ])

    return A

def visualize_affine_transform(ref, test):

    # GET AFFINE TRANSFORM
    A = calculate_affine_transform(ref, test)

    # GET REFERENCE CORNER POINTS (using OpenCV Point coordinate system)
    # in OpenCV Point(x,y) is (column,row) ... very confusing...
    x_max = ref.shape[1]-1
    y_max = ref.shape[0]-1
    p1 = np.array([0, 0, 1])
    p2 = np.array([x_max, 0, 1])
    p3 = np.array([x_max, y_max, 1])
    p4 = np.array([0, y_max, 1])
```

```
    # APPLY AFFINE TRANSFORM
    t1 = A @ p1
    t2 = A @ p2
    t3 = A @ p3
    t4 = A @ p4
    # round and cast to int b/c cv2.line requires int datatypes
    t1 = np.around(t1).astype(int)
    t2 = np.around(t2).astype(int)
    t3 = np.around(t3).astype(int)
    t4 = np.around(t4).astype(int)

    # PLOT TRANSFORMED POINTS
    cv2.line(test, tuple(t1), tuple(t2), (255,0,0), 3)
    cv2.line(test, tuple(t2), tuple(t3), (255,0,0), 3)
    cv2.line(test, tuple(t3), tuple(t4), (255,0,0), 3)
    cv2.line(test, tuple(t4), tuple(t1), (255,0,0), 3)

    cv2.imwrite('affine_transform_test.png', test)


def main():
    test_path = 'images/test.png'
    test2_path = 'images/test2.png'
    reference_path = 'images/reference.png'

    ref = cv2.imread(reference_path)
    test = cv2.imread(test_path)
    test2 = cv2.imread(test2_path)

    # Q1 C)
    calculate_affine_transform(ref, test)

    # Q1 D)
    visualize_affine_transform(ref, test)
    visualize_affine_transform(ref, test2)
```

door.py

```
import numpy as np

A = np.array([
    [1062, 635, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1062, 635, 1, 0, 0, 0],
    [1285, 531, 1, 0, 0, 0, -31097, -12850.2, -24.2],
    [0, 0, 0, 1285, 531, 1, 0, 0, 0],
    [1300, 981, 1, 0, 0, 0, -31460, -23740.2, -24.2],
    [0, 0, 0, 1300, 981, 1, -46150, -34825.5, -35.5],
    [1072, 1072, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1072, 1072, 1, -38056, -38056, -35.5]
])

q1 = np.array([
    1025, 630, 1
])
q2 = np.array([
    1768, 222, 1
])
q3 = np.array([
    1835, 3726, 1
])
q4 = np.array([
    1005, 3494, 1
])

def main():
    # FIND EIGENVALUES AND EIGENVECTORS
    w, v = np.linalg.eig(A.T @ A)

    # CALCULATE HOMOGRAPHY MATRIX H
    i_min_eigenvalue = np.argmin(w)
    h = v[:, i_min_eigenvalue]
    H = np.reshape(h, (3,3))

    # WARP EACH POINT (DOOR CORNERS)
    y = H @ q1
    q1_ = y/y[2]

    y = H @ q2
    q2_ = y/y[2]

    y = H @ q3
    q3_ = y/y[2]

    y = H @ q4
    q4_ = y/y[2]

    # PRINT TRANSFORMED COORDINATES
    print(q1_)
    print(q2_)
    print(q3_)
    print(q4_)
```

homography.py

```
import numpy as np
import random
INLIER_THRESH = 3

# Parameters:   A list of matching keypoints from both images
#               elements of the list are tuples (keypoint, keypoint)
#               corresponding to a matched pair from both images
def calculate_homography(matched_keypoints):
    n_keypoints = len(matched_keypoints)
    A = np.empty((0,9))

    for t in range(n_keypoints):
        (keypoint_j, keypoint_k) = matched_keypoints[t]
        (x_j, y_j) = keypoint_j.pt
        (x_k, y_k) = keypoint_k.pt

        # CREATE PARTIAL A MATRIX
        a = np.array([
            [x_j, y_j, 1, 0, 0, 0, -x_k*x_j, -x_k*y_j, -x_k],
            [0, 0, 0, x_j, y_j, 1, -y_k*x_j, -y_k*y_j, -y_k]
        ])
        # APPEND PARTIAL TO A
        A = np.append(A, a, axis=0)

    # CALCULATE EIGENVALUES & EIGENVECTORS
    w, v = np.linalg.eig(A.T @ A)

    # CALCULATE HOMOGRAPHY MATRIX H
    i_min = np.argmin(w)
    h = v[:, i_min]
    H = np.reshape(h, (3,3))

    return H

# Parameters:   Homography matrix (H)
#               OpenCV point (pt) of point to transform
def transform_point(H, point):
    (x, y) = point  #.pt
    q = np.array([x, y, 1])

    wp = H @ q
    p = wp/wp[2]

    # Return an OpenCV position tuple (column, row)
    return (p[0], p[1])
```

```
def number_of_inliers(H, matched_keypoints):
    n = 0
    for i in range(len(matched_keypoints)):
        (keypoint_j, keypoint_k) = matched_keypoints[i]

        p = np.asarray(keypoint_k.pt)
        p_hat = np.asarray(transform_point(H, keypoint_j.pt))
        d = np.linalg.norm(p - p_hat)

        if d < INLIER_THRESH:
            n = n + 1

    return n

# Parameters:   A list of matched keypoints
# Output:       Best solution for the homography (H)
#               calculated via RANSAC
def RANSAC_homography(matched_keypoints):
    max_inliers = 0
    H_optimal = None

    # ITERATE ~3000 ROUNDS to guarentee 99% accuracy assuming 20% inliers
    for i in range(3000):
        # SELECT 4 RANDOM MATCHES TO CALCULATE HOMOGRAPHY
        matches = random.sample(matched_keypoints, 4)
        H = calculate_homography(matches)

        # CALCULATE NUMBER OF INLIERS
        n_inliers = number_of_inliers(H, matched_keypoints)

        # STORE THE HOMOGRAPHY WITH MOST INLIERS
        if n_inliers > max_inliers:
            max_inliers = n_inliers
            H_optimal = H

   return H_optimal
```

panorama.py

```
from SIFT_matching import *
from homography import *
from scipy import ndimage

# Parameters:   Two images img1, img2
# Output:       List of matching keypoints between the two images
#               elements of the list are in the form
#               (keypoint_1, keypoint_2) where keypoints 1 & 2 are matching
def get_matching_keypoints(img1, img2):
    keypts_1, des_1 = extract_SIFT_keypoints(img1)
    keypts_2, des_2 = extract_SIFT_keypoints(img2)

    matched_keypoints = []

    for i in range(len(keypts_1)):
        # GET THE KEYPOINT MATCH CORRESPONDING TO KEYPOINT i
        f_j, j, thresh_ratio = calculate_correspondance(des_1[i], des_2)

        # THRESHOLD FOR GOOD MATCHES
        if thresh_ratio < 0.8:
            match_tuple = (keypts_1[i], keypts_2[j])
            matched_keypoints.append(match_tuple)

    return matched_keypoints

def stitch_images(H, img1, img2):
    nrow = img1.shape[0]
    ncol = img1.shape[1]
    nrow_2 = img2.shape[0]
    ncol_2 = img2.shape[1]

    # TRANSFORM 4 CORNERS OF FIRST IMAGE
    (topleft_x, topleft_y) = transform_point(H, (0,0))
    (topright_x, topright_y) = transform_point(H, (ncol-1,0))
    (bottomright_x, bottomright_y) = transform_point(H, (ncol-1,nrow-1))
    (bottomleft_x, bottomleft_y) = transform_point(H, (0,nrow-1))

    # CALCULATE DIMENSIONS OF STITCHED IMAGE
    min_y = min(topleft_y, topright_y, 0)
    max_y = max(bottomleft_y, bottomright_y, nrow_2-1)
    min_x = min(topleft_x, bottomleft_x, 0)
    max_x = max(topright_x, bottomright_x, ncol_2-1)
    nrows_stitched = max_y - min_y + 1
    ncols_stitched = max_x - min_x + 1

    # CREATE BLANK STITCHED IMAGE
    shape = (int(nrows_stitched)+1, int(ncols_stitched)+1, 3)
    stitched_image = np.zeros(shape)

    # CALCULATE OFFSET
    offset_x = 0 - min_x
    offset_y = 0 - min_y
```

```
    # TRANSFORM ALL PIXELS IN THE FIRST IMAGE
    for x in range(ncol):
        for y in range(nrow):
            (x_t, y_t) = transform_point(H, (x,y))
            x_t = int(round(x_t + offset_x))
            y_t = int(round(y_t + offset_y))
            stitched_image[y_t, x_t, :] = img1[y, x, :]

    # SHIFT ALL PIXELS IN THE SECOND IMAGE
    for x in range(ncol_2):
        for y in range(nrow_2):
            x_t = int(round(x + offset_x))
            y_t = int(round(y + offset_y))
            stitched_image[y_t, x_t, :] = img2[y, x, :]

    # INTERPOLATE SKIPPED PIXELS
    # after applying H some pixels in the stitched image may be skipped
    # b/c float position is converted to int position (during formation of stitched image)
    for x in range(1, stitched_image.shape[1]-1):
        for y in range(1, stitched_image.shape[0]-1):
            # If point is black/skipped then linearly interpolate based on neighbours
            if np.all(stitched_image[y,x,:] == 0):
                interpolate_x = 0.5*(stitched_image[y,x-1,:] + stitched_image[y,x+1,:])
                interpolate_y = 0.5*(stitched_image[y-1,x,:] + stitched_image[y+1,x,:])
                if np.linalg.norm(interpolate_x) > np.linalg.norm(interpolate_y):
                    stitched_image[y,x,:] = interpolate_x
                else:
                    stitched_image[y,x,:] = interpolate_y

    return stitched_image

def main():
    landscape1_path = 'images/landscape_1.jpg'
    landscape2_path = 'images/landscape_2.jpg'
    img1 = cv2.imread(landscape1_path)
    img2 = cv2.imread(landscape2_path)

    # Q3
    matched_keypoints = get_matching_keypoints(img1, img2)
    H = RANSAC_homography(matched_keypoints)
    stitch = stitch_images(H, img1, img2)

    cv2.imwrite('stitch.jpg', stitch)
```