Project #2

Image Mosaicing using Harris Corner Detection and Estimation of Homography

EECE 5639

Instructor: Octavia Camps

Authors: Juan Pablo Bernal Medina, Shivam Sharma

Abstract

In this project, we implemented our own Harris corner detector and non-max suppression algorithm to generate a sparse set of corners for an image. The corners between the two images are then correlated with a normalized cross-correlation to generate pairs of corners with the highest NCC score. We then use RANSAC to estimate a homography between the two perspectives by randomly sampling four pairs to produce the best fit. With the homography, we then warp the second image onto the first and merge the overlapping pixels to create the mosaic.

Algorithm

Harris corner detector: We began by computing the gradient of the image by using a 3x3 Sobel operator. We then computed the C matrix for each pixel given by:

$$C = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Where I_x and I_y are the gradients at (x, y). The output of the function gave us a Harris corner response at each pixel which we were then able to use to pinpoint corners on the image.

Non-max suppression: The output of the corner detector is then fed into a non-max suppression algorithm to find local maxima and therefore generate a sparse set of corners. To accomplish this we used a slightly different route and utilized mean shift to find every local maximum. The algorithm would be given a window size and would start searching the image at coordinates half the window size. It would start at that pixel and use the mean shift algorithm to find the nearest maxima. It would look in a window around that pixel and find the maximum. If the maximum was greater than the threshold it would move to that pixel location and repeat the process. This algorithm allowed us to find the corners with the greatest response and limit the number that we would find. The window size, in this case, limited the closest distance between observed corners.

Normalized cross-correlation (NCC): After running the Harris detector and non-max suppression on both images, we were ready to find correlations between the two. For every pair of corners across both images, an NCC score was assigned:

$$NCC\ score = \sum_{i,j}^{N} \frac{A_{i,j} - \mu_A}{\sigma_A} \cdot \frac{B_{i,j} - \mu_B}{\sigma_B}$$

Where $A_{i,j}$ and $B_{i,j}$ are the pixel values for each image respectively in the window and μ and σ are the mean and standard deviation respectively of the images in the window N.
The NCC pairs were then sorted and filter based on a threshold score. But we did not want to lose too many corners so we had a minimum length as well for output pairings so that in the case the threshold was too high, we would still have enough corners.

Homography estimation: With pairs of corners sorted by NCC pairs we could then use them to estimate a homography with RANSAC. Four pairs were picked randomly from the set of NCC sorted pairs and used to estimate a homography.
The generated homography was then used to compute projections for every other corner and find the error in distance between the projected and real location. The projection of a point (x, y) by a homography was calculated by:

$$x' = \frac{H_{11} \cdot x + H_{12} \cdot y + H_{13}}{H_{31} \cdot x + H_{32} \cdot y + H_{33}} \qquad y' = \frac{H_{21} \cdot x + H_{22} \cdot y + H_{23}}{H_{31} \cdot x + H_{32} \cdot y + H_{33}}$$

With a threshold in the distance, we were able to identify outliers and inliers and in the end, the homography that produced the most inliers was kept. Then we took all the inliers and used least squares to find the best fitting homography for the set of corners.

Warping and merging: With the homography, we were then able to warp one image into the perspective of the other. The new size of the mosaic could be calculated by finding the transformation for each of the 4 corners of the image. Both images were placed appropriately into the final mosaic with the overlapping pixels being merged. This was done by iterating through the overlapping window and finding the distance of each point to the center of both images. This distance was then normalized by dividing it by the max distance, being from the center to a corner. This factor was then subtracted from one to give the final weights for each image respectively:

$$M_{i,j} = \frac{r_A A_{i,j} + r_B B_{i,j}}{r_A + r_B}$$

Where $M_{i,j}$ is the output merged pixel, $A_{i,j}$ and $B_{i,j}$ are the pixels from the two images.

Experiments

Determine the probability of outliers: To do RANSAC to determine the inliers correspondences, we have to choose 4 random pairs of corners, compute the homography matrix and see how many of the other corresponding corners match this matrix. However, we don't want to do this process for every single combination of 4 corresponding corners. To determine how many times we have to repeat the process to get a result where we have a probability of 0.99 of getting all the inliers we use the equation $N = \frac{ln(1-p)}{ln(1-(1-e)^s)}$ where N is the number of times we have to do the process, p is the probability of success (0.99), e is the probability of outlier and s the number of correspondences. However, we don't have a value for e. This probability is directly related to the robustness of the NCC and Harris corner detector algorithms and therefore, we can assume that for our algorithm this probability will be similar for different images. To find this value, we ran some tests with different values of e and different input images. Then, we took the maximum number of inliers detected off all the different values of e and divided it by the total number of corners.

DanaHallWay1 = 1 - (15 corners/19 corners) = 0.21

DanaHallWay2 = 1 - (26 corners/30 corners) = 0.13

DanaOffice = 1 - (17 corners/19 corners) = 0.1

On average the probability of having an outlier is 0.14 just to be safe we increased the value of e to 0.3. However, if we have too many corners this function might go to infinity therefore, we put a limit of N = 1 million.

Determine the window size of RANSAC: When doing RANSAC we have to determine what it means to be inlier or outliner. For this, we use an error window that will decide if a corner is in fact an inlier or not. This was faster to implement than to compute the distance between expected and projected points every time. However, if this window is too big then there can be false positives and if it is too small it may reject an inlier just because of noise. Therefore, to get a good estimate we started with a window size of 10 pixels and start reducing it until we start getting the same number of inliers for different window sizes. This would determine what was a good estimate of inliers (15 in the case of DanaHallWay1). Finally, we kept reducing it until we lost 1 inlier. We saw that with a 3-pixel window size we still had the same amount of inliers but for size 2 we lost one. Therefore, we decided to go with a pixel window of 3 pixels.

After having decided on the parameters, 3 tests were run selecting two pictures of each scene to check how well the program worked. In Figures 1 to 3, the output of the NCC function can be observed. The colored lines represent a correlation between corners from image one and image two. Here, we can see that some of the correlations are correct but there are some corners from the first image that has multiple possible correlations in the second image. In spite of this, we got a good amount of correlated corners where at least some of them were correct. To get rid of these false correlations, the RANSAC algorithm was applied using the homology matrix to determine which correlations were in fact correct. In Figures 4 to 6, the output of the RANSAC function can be observed. This time, there are fewer correlations but all of the corners in image one only have one correlated corner in image two. Moreover, as seen in the pictures, the correlated corners are very accurate. Now with this information, we were able to go ahead and wrap one image onto the other.

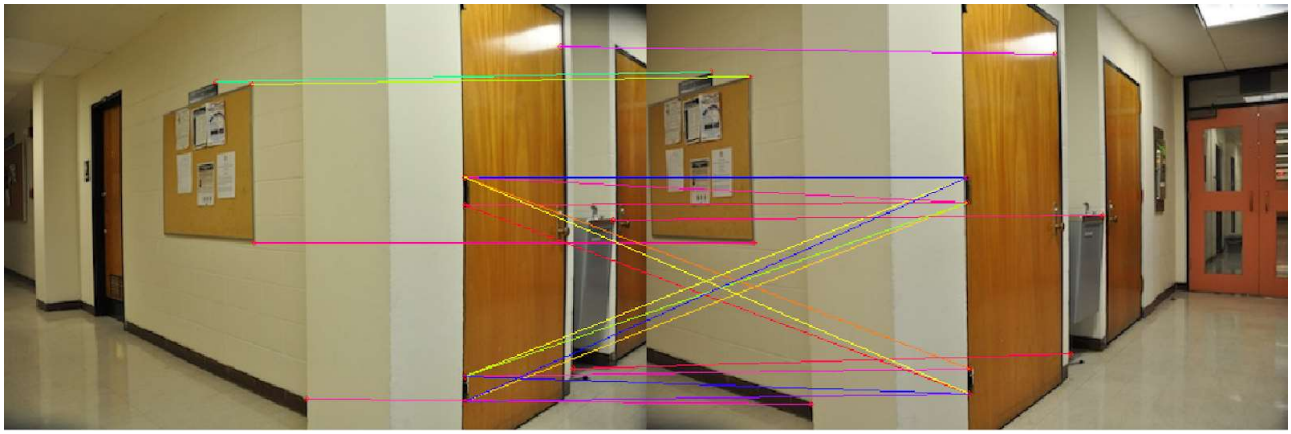


Fig.1 Corners pairings for top NCC pairs - Set 1

Fig.2 Corners pairings for top NCC pairs - Set 2

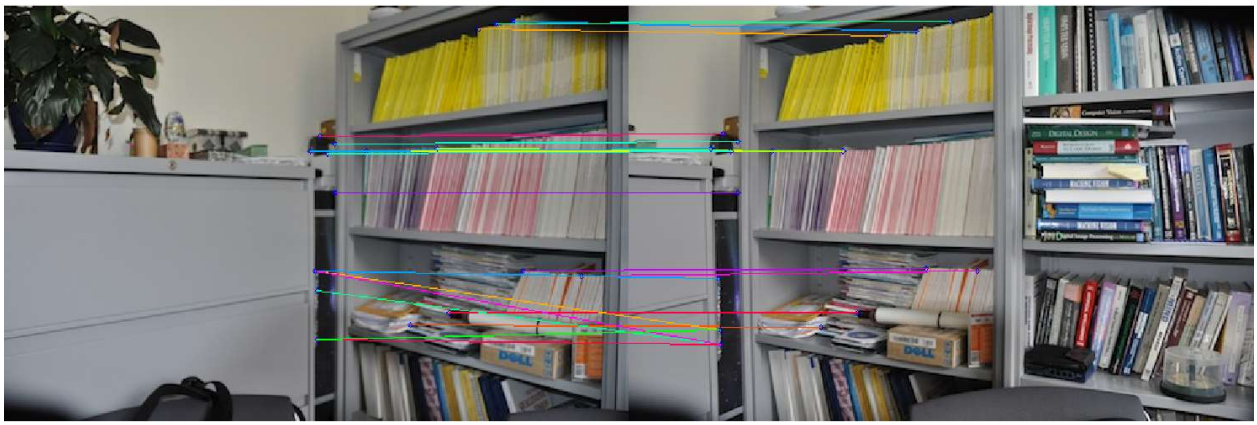


Fig.3 Corners pairings for top NCC pairs - Set 3

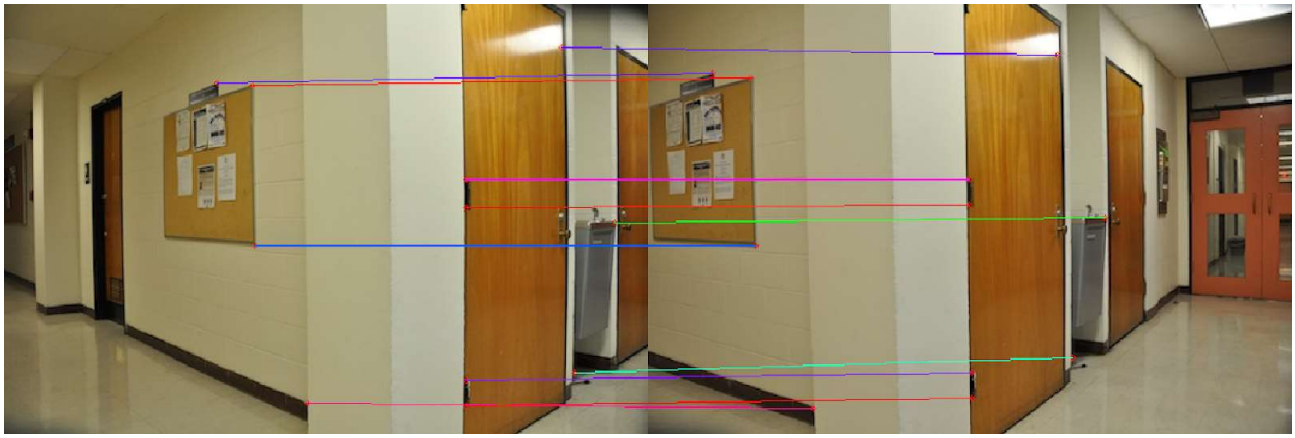


Fig.4 Corners pairings for inliers after RANSAC - Set 1

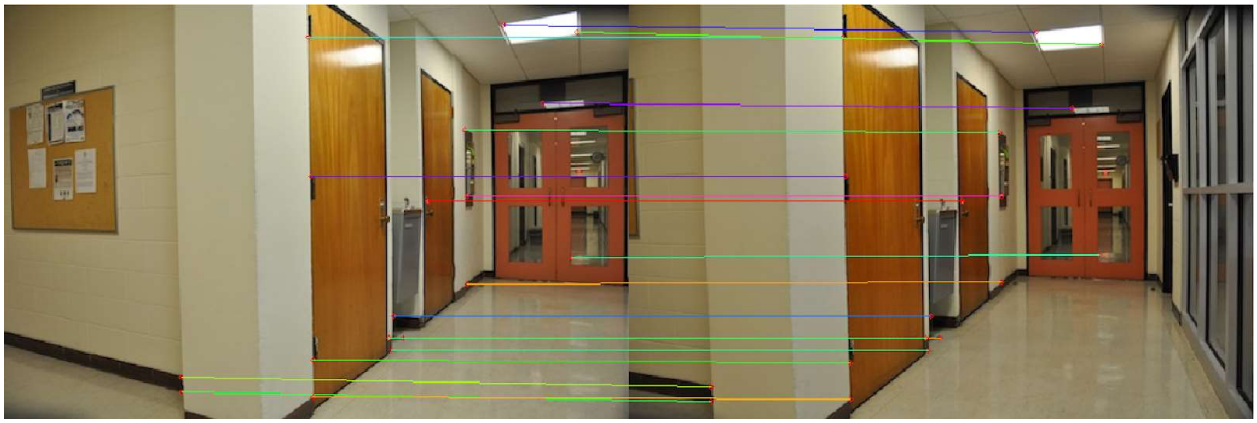


Fig.5 Corners pairings for inliers after RANSAC - Set 2

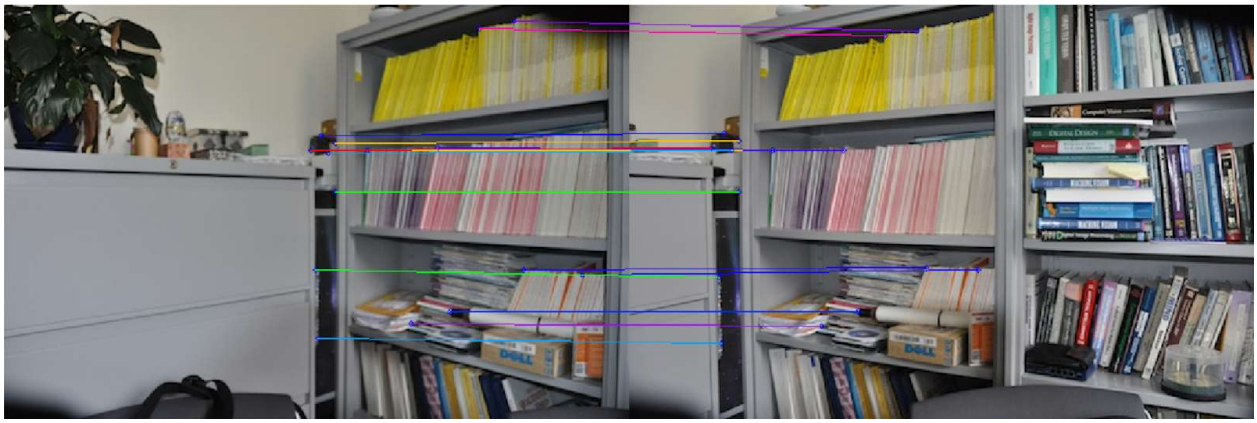


Fig.6 Corners pairings for inliers after RANSAC - Set 3

Values of parameters used

Corner detection:

Harris and Sobel window size were chosen to be 3. This was mainly due to the immensely larger runtime if the harris window size was any bigger. The program works well enough with a window size of 3 so we kept it at that.

Harris scale - 2. This was once again chosen as the runtime of computing the C matrix for every single was too long - > 15 seconds per image. By introducing a scalar of 2, we basically were only calculating the response for every other pixel with reduced the runtime by four times and only resulted in a loss of resolution of 1 pixel.

Harris corners threshold: 2.5 - arbitrarily chosen to get a good amount of corners. Main purpose is to allow negative Harris pixels to be ignored.

The non-max suppression window size was chosen to be 15. This is important as the window size determines the minimum distance between the observed corner (in our implementation). So for a window size of 15, the minimum lateral distance between corners is 8 pixels.

NCC:
NCC score threshold: 0.8, or a minimum of 20 correspondences

RANSAC:
Probability p = 0.99
Probability e = 0.3
Inlier window size = 3 pixels

Finally, using these parameters described above, the program was used to merge 3 sets of images (same images used for the experiments previously mentioned). Figures 7 to 9 are the final output of the program for the 3 sets of images. These results are what we expected to get from the program.

Fig.7 Final mosaic - Set 1

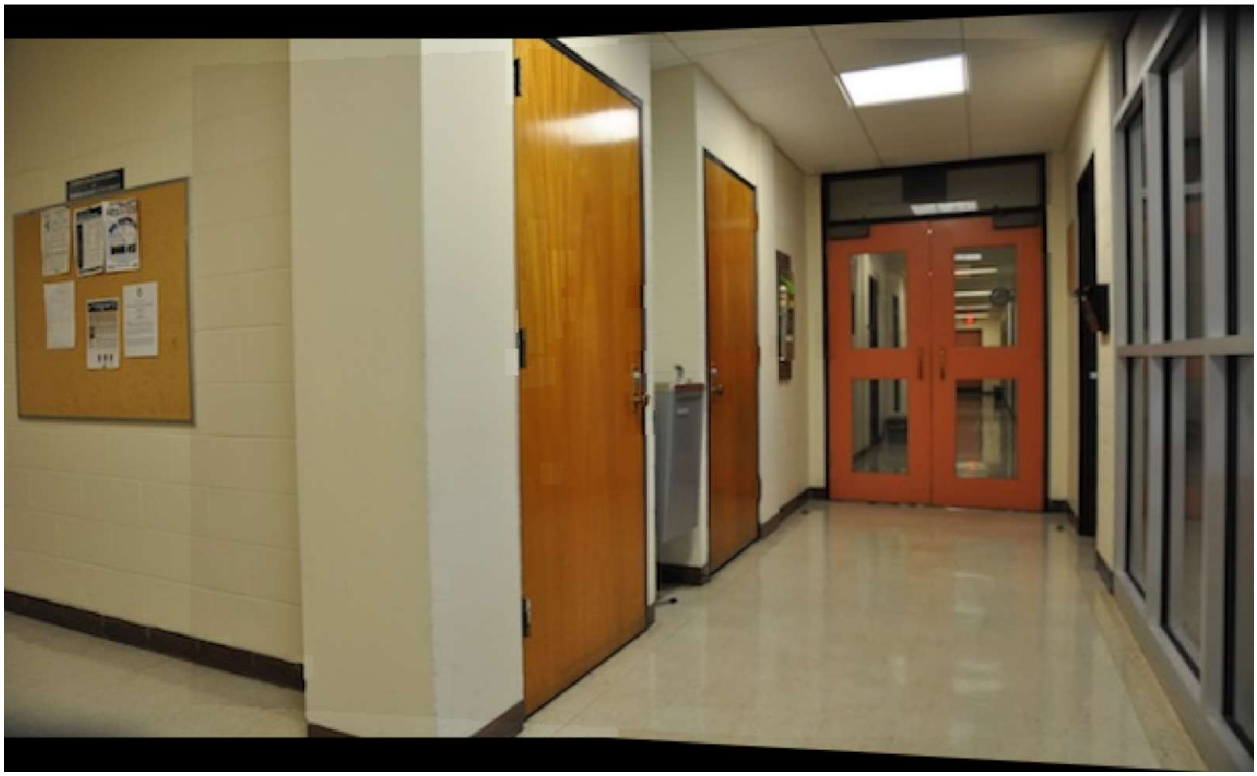


Fig.8 Final mosaic - Set 2

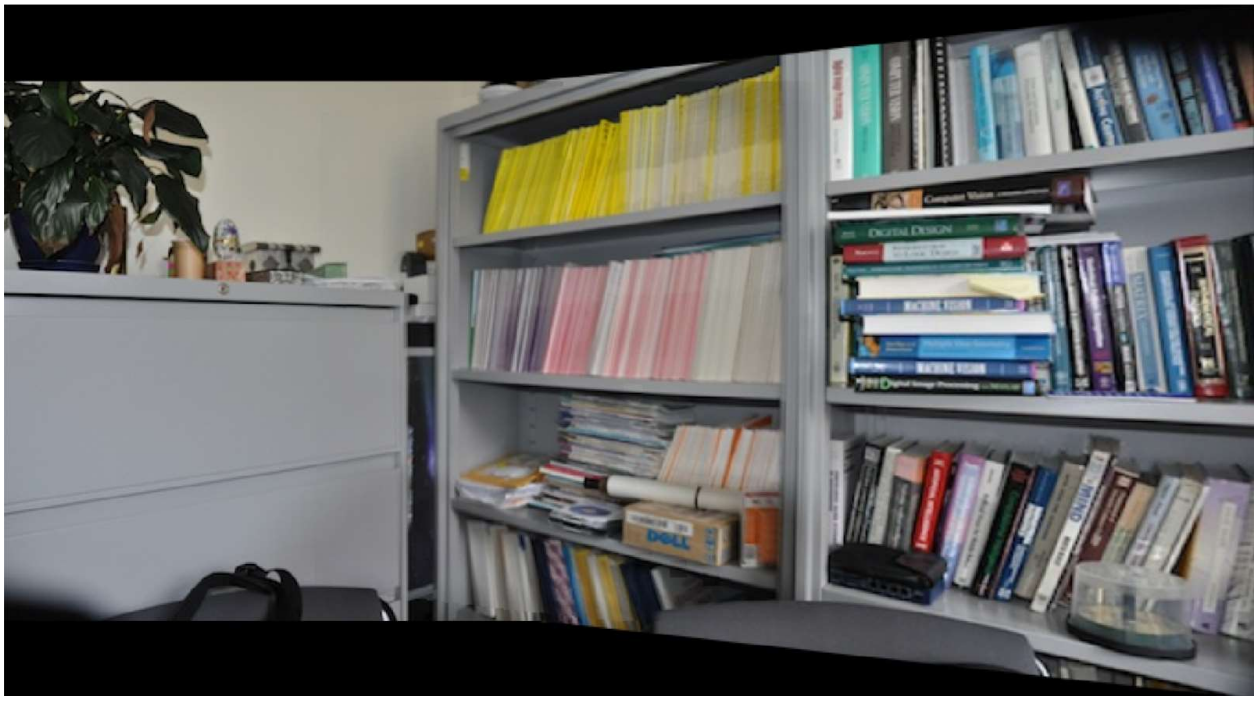
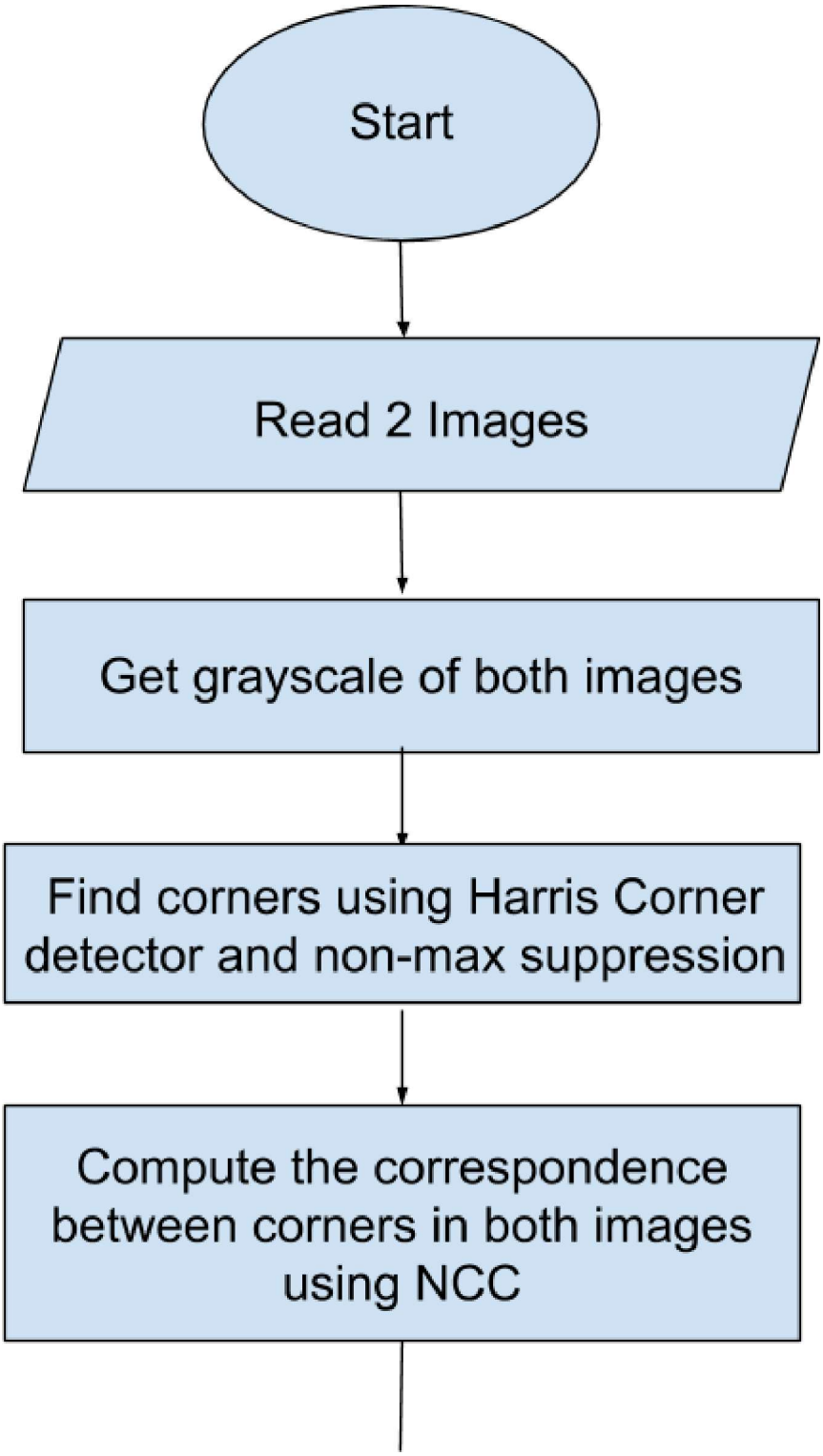
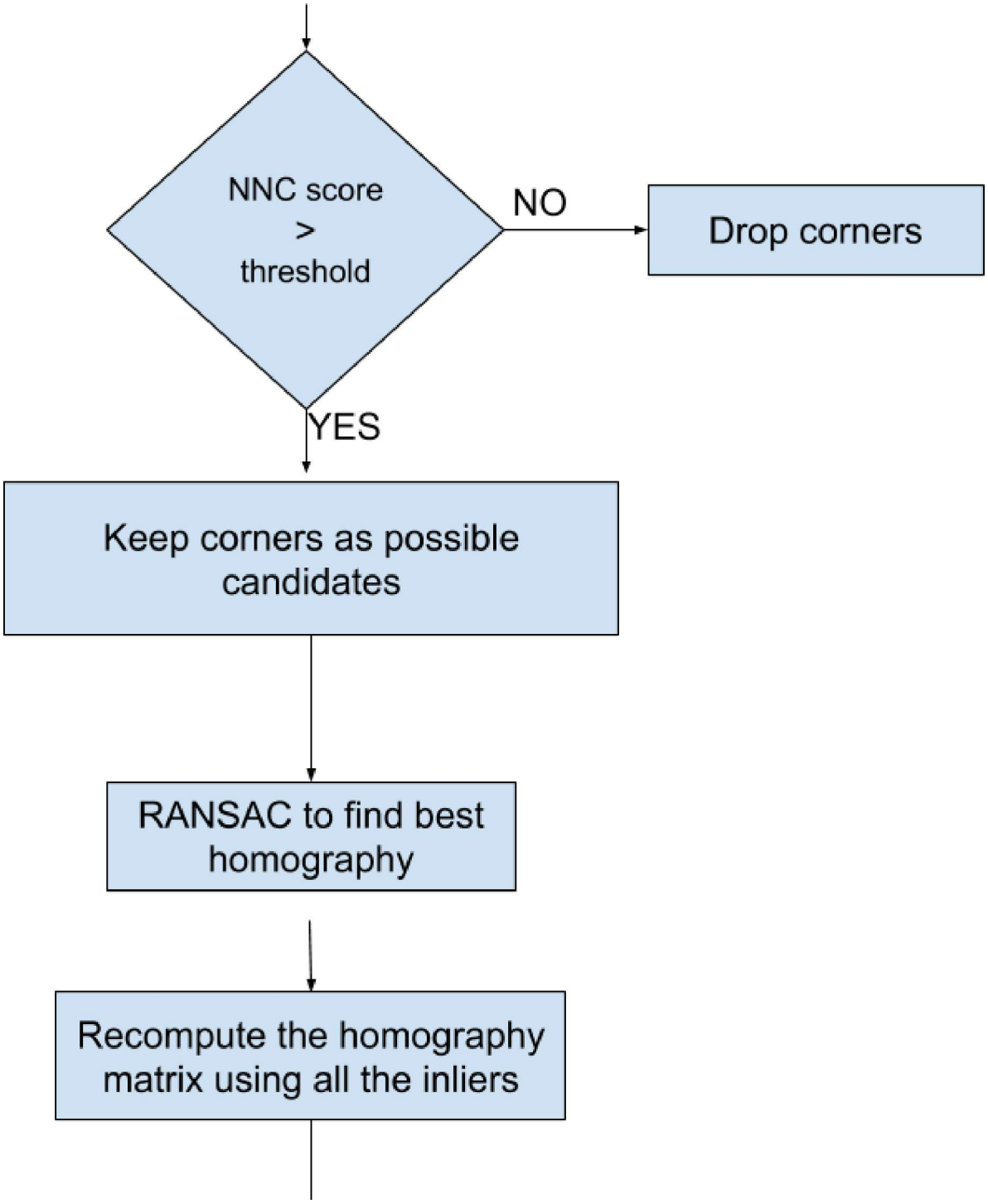
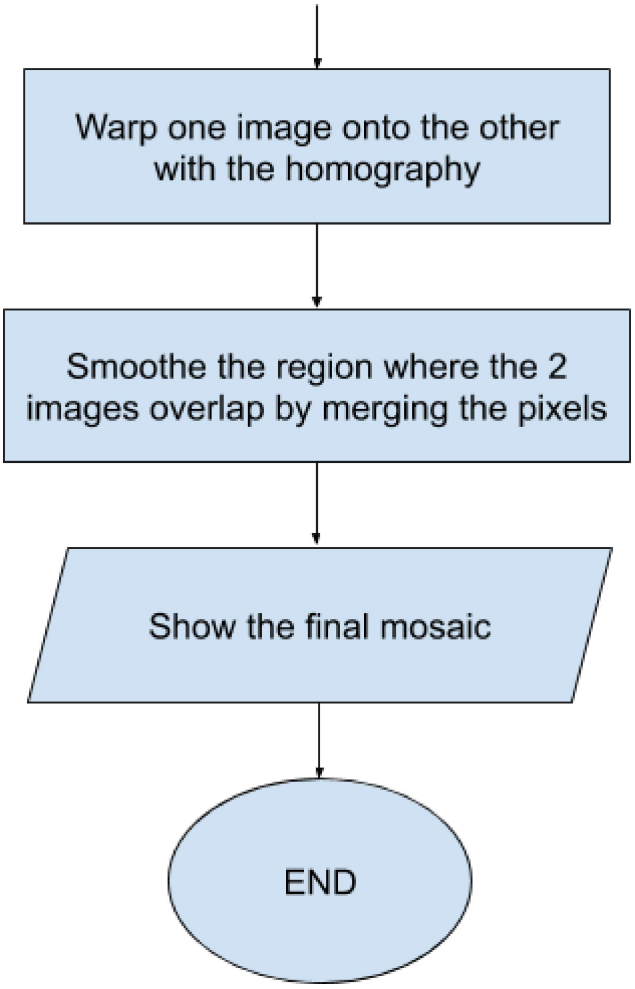


Fig.9 Final mosaic - Set 3

Flowchart:


Start
Read 2 Images
Get grayscale of both images
Find corners using Harris Corner detector and non-max suppression
Compute the correspondence between corners in both images using NCC


NNC score
>
threshold
NO
Drop corners
YES
Keep corners as possible candidates
RANSAC to find best homography
Recompute the homography matrix using all the inliers



Conclusions

In conclusion, 2 images of the same scene taken by rotating a camera were merged together. First, by finding the corners using a harris corner detection algorithm, looking for correlations between corners in both images using the NCC algorithm, removing false correlations with RANSAC and finally wrapping the images with the homography matrix. To maximize the results of these algorithms some parameters were chosen arbitrarily (trial and error) and some were derived after performing different experiments. The final result of the mosaic was as expected. The images are perfectly aligned, with any ghosting and the places where the images overlap is almost undetectable.

Appendix

Code:

```
import numpy as np
from matplotlib import pyplot as plt
import math
import cv2
import os
import gc
import time
import colorsys


def Harris(img, window_size=3, sobel_size=3, k=0.04, step_size=1):
    if len(img.shape) > 2:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    h, w = img.shape[0], img.shape[1]

    sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=sobel_size)
    sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=sobel_size)

    Ixx = sobelx ** 2
    Iyy = sobely ** 2
    Ixy = sobelx * sobely

    R = np.zeros_like(img, dtype=np.float64)

    offset = np.floor(window_size / 2).astype(np.int8)
    for y in range(offset, h-offset, step_size):
        for x in range(offset, w-offset, step_size):
            Sxx = np.sum(Ixx[y-offset:y+1+offset, x-offset:x+1+offset])
            Syy = np.sum(Iyy[y-offset:y+1+offset, x-offset:x+1+offset])
            Sxy = np.sum(Ixy[y-offset:y+1+offset, x-offset:x+1+offset])
            r = (Sxx*Syy)-(Sxy**2) - k*(Sxx+Syy)**2
            if r < 0:
                r = 0
            R[y][x] = r

    R /= np.max(R)
    R *= 255.0
    # _, R = cv2.threshold(R, 0.01 * np.max(R), 255, 0)
    return R


def non_max_suppression(img, window_size=5, thresh=2.5):
    h, w = img.shape[0], img.shape[1]

    offset = np.floor(window_size / 2).astype(np.int8)

    temp = np.zeros_like(img)
    temp[offset:h-offset, offset:w-offset] = img[offset:h-offset, offset:w-offset]
    img = temp
    del temp

    index = np.zeros((h, w, 3), dtype=np.int16)

    index[:, :, 0] = img

    for y in range(h):
        for x in range(w):
            index[y][x][1] = x
```

```
            index[y][x][2] = y

    corners = []

    global NMS_index
    NMS_index = np.zeros((h, w))

    for y in range(offset, h - offset, offset):
        for x in range(offset, w - offset, offset):
            ret = mean_shift_converge(index, (x, y), window_size, thresh)
            if ret:
                corners.append(ret)

    corners = list(set(corners))
    corners.sort()
    return corners


def mean_shift_converge(index, point, window_size=5, thresh=2.5):
    x, y = point
    offset = np.floor(window_size / 2).astype(np.int8)

    if NMS_index[y][x] == 1:
        return None

    window = index[y - offset:y + 1 + offset, x - offset:x + 1 + offset]
    if np.max(window[:, :, 0]) > thresh:
        window = np.reshape(window, (window_size ** 2, 3))
        window[::-1] = window[window[:, 0].argsort()]
        if window[0][1] == x and window[0][2] == y:
            NMS_index[y][x] = 1
            return x, y
        else:
            if mean_shift_converge(index, (window[0][1], window[0][2]), window_size, thresh):
                NMS_index[y][x] = 1
                return window[0][1], window[0][2]
            else:
                return None
    else:
        return None


def NCC(img1, img2, p1, p2, window_size=3):
    if len(img1.shape) > 2:
        img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)

    if len(img2.shape) > 2:
        img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    h, w = img1.shape[0], img1.shape[1]

    offset = np.floor(window_size / 2).astype(np.int8)

    (x1, y1) = p1
    (x2, y2) = p2

    if x1 <= offset or x1 >= (w-offset) or y1 <= offset or y1 >= (h-offset) or\
            x2 <= offset or x2 >= (w - offset) or y2 <= offset or y2 >= (h - offset):
        return 0

    window1 = img1[y1-offset:y1+1+offset, x1-offset:x1+1+offset]
    window1 = window1.astype(np.float64)

    window2 = img2[y2-offset:y2+1+offset, x2-offset:x2+1+offset]
    window2 = window2.astype(np.float64)
```

```
    mean1 = np.mean(window1)
    std1 = np.std(window1)

    mean2 = np.mean(window2)
    std2 = np.std(window2)

    s = 0
    for i in range(0, window_size):
        for j in range(0, window_size):
            s += (window1[i][j] - mean1) / std1 * (window2[i][j] - mean2) / std2
    s /= (window_size ** 2)

    return s


def remove_values_from_list(the_list, val):
    return [value for value in the_list if value[1] != val]


def filter_NCC(NCC_list, len_thresh=0, score_thresh=0):
    NCC_list.sort(key=lambda x: x[0], reverse=True)

    if score_thresh > 0:
        NCC_list = [NCC_list[i] for i in range(len(NCC_list)) if NCC_list[i][0] > score_thresh]

    if 0 < len_thresh <= len(NCC_list):
        NCC_list = NCC_list[:len_thresh]

    return NCC_list


def est_homography(points1, points2, thresh=3, N_max=10000):
    p = 0.9
    e = 0.3
    s = len(points1)
    N = np.min((np.log(1 - p) / (np.log(1 - (1 - e) ** s)), N_max))
    N = np.int32(N)

    inliers = []
    inlier_max = 0
    print("Corner size:", s, "N: ", N)
    for x in range(0, N):
        four_points1 = []
        four_points2 = []
        c = 0
        while c < 4:

            r = np.random.randint(low=0, high=s)
            if points1[r] not in four_points1 and points2[r] not in four_points2:
                four_points1.append(points1[r])
                four_points2.append(points2[r])
                c += 1

        four_points1 = np.array(four_points1, ndmin=2)
        four_points2 = np.array(four_points2, ndmin=2)

        h, status = cv2.findHomography(four_points2, four_points1)

        if status[0][0] != 0:
            counter = 0
            temp = []
            for i in range(len(points1)):
                x1, y1 = get_point_from_homography(points2[i], h)
                if x1 is None or y1 is None:
```

```
                    break

                if points1[i][0]-thresh <= x1 <= points1[i][0]+thresh and points1[i][1]-thresh <= y1 <= points1[i][1]+thresh:
                    counter += 1
                    temp.append((points1[i], points2[i]))

            if counter > inlier_max:
                inliers = temp
                inlier_max = counter

    p1 = np.array([(k[0][0], k[0][1]) for k in inliers])
    p2 = np.array([(k[1][0], k[1][1]) for k in inliers])
    h, _ = cv2.findHomography(p2, p1)
    return h, inliers


def get_point_from_homography(p, H):
    den = H[2][0]*p[0] + H[2][1]*p[1] + 1
    if den == 0.0 or math.isnan(den):
        return None, None
    x = np.round((H[0][0]*p[0] + H[0][1]*p[1] + H[0][2]) / den)
    y = np.round((H[1][0]*p[0] + H[1][1]*p[1] + H[1][2]) / den)

    if np.fabs(x) > 16000 or np.fabs(y) > 16000:
        return None, None

    return np.array([x, y], dtype=np.int16)


def display_corner_pairings(img1, img2, lines, y1_shift=0, y2_shift=0):
    w1, h1 = img1.shape[1], img1.shape[0]
    w2, h2 = img2.shape[1], img2.shape[0]

    w_max, h_max = np.max((w1, w2)), np.max((h1, h2))

    out1 = np.zeros((h_max, w_max, 3), dtype=np.uint8)
    out1[y1_shift:h1+y1_shift, 0:w1] = img1
    out2 = np.zeros((h_max, w_max, 3), dtype=np.uint8)
    out2[y2_shift:h2+y2_shift, 0:w2] = img2
    combine = np.concatenate((out1, out2), axis=1)

    for p in lines:
        cv2.circle(combine, (p[0][0], p[0][1] + y1_shift), 2, (0, 0, 255), 1)
        cv2.circle(combine, (p[1][0] + w1, p[1][1] + y2_shift), 2, (0, 0, 255), 1)

        hsv = np.array((np.random.rand(), 1, 1))
        rgb = list((np.array(colorsys.hsv_to_rgb(hsv[0], hsv[1], hsv[2])) * 255.0).astype(np.uint16))
        rgb = [int(rgb[0]), int(rgb[1]), int(rgb[2])]
        cv2.line(combine, (p[0][0], p[0][1]+y1_shift), (p[1][0]+w1, p[1][1]+y2_shift), rgb, 1)

    plt.imshow(combine)
    plt.axis('off')
    plt.show()


def main():
    dataset_path = ["DanaHallWay1", "DanaHallWay2", "DanaOffice"]

    for path in dataset_path:
        print(path)

        # Open first 2 images in the specified folder and save them in imgset
        imgset = []
        for i in range(2):
            imgset.append(cv2.cvtColor(cv2.imread(path + '/' + os.listdir(path + '/')[i]), cv2.COLOR_BGR2RGB))
```

```
w, h = imgset[0].shape[1], imgset[0].shape[0]

# Get corners for the first image
a = time.time()                     # Timing Statements
ret = Harris(imgset[0], 3, 3, 0.04, step_size=2)
print("Harris: ", time.time() - a)  # Timing Statements
a = time.time()                     # Timing Statements
centroids = non_max_suppression(ret, 15)
print("NMS: ", time.time() - a)  # Timing Statements
a = time.time()                     # Timing Statements
corners = [centroids]

# Get corners for the second image
ret = Harris(imgset[1], 3, 3, 0.04, step_size=2)
print("Harris: ", time.time() - a)  # Timing Statements
a = time.time()                     # Timing Statements
centroids = non_max_suppression(ret, 15)
print("NMS: ", time.time() - a)
a = time.time()
corners.append(centroids)

# Calculate NCC score for each corner pairing
NCC_score = []
for i in range(len(corners[0])):
    for j in range(len(corners[1])):
        score = NCC(imgset[0], imgset[1], corners[0][i], corners[1][j], 15)
        NCC_score.append((score, corners[0][i], corners[1][j]))

# Filters the NCC list based on some parameters - sort by score with a threshold, min length
NCC_score = filter_NCC(NCC_score, len_thresh=20)
print("NCC: ", time.time() - a)     # Timing Statements
a = time.time()                     # Timing Statements

# Get homography
p1 = [p[1] for p in NCC_score]
p2 = [p[2] for p in NCC_score]
H, inliers = est_homography(p1, p2)

# Make sure the images mover left to right
if inliers[0][0][0] < inliers[0][1][0]:
    # Swap images and homography if they flow right to left
    NCC_score = [(p[0], p[2], p[1]) for p in NCC_score]
    p1 = [p[1] for p in NCC_score]
    p2 = [p[2] for p in NCC_score]
    H, inliers = est_homography(p1, p2)
    imgset[0], imgset[1] = imgset[1], imgset[0]
H_I = np.linalg.inv(H)

print("Homography: ", time.time() - a)

# Display corner pairings before RANSAC
p1 = [(p[1], p[2]) for p in NCC_score]
display_corner_pairings(imgset[0], imgset[1], p1)
# Display corner pairings after RANSAC
display_corner_pairings(imgset[0], imgset[1], inliers)

# Calculate the bounding edges after warping the second image with H
# c1-c4 are the new corners of the image after warping
c1 = get_point_from_homography((0, 0), H)
c2 = get_point_from_homography((w-1, 0), H)
c3 = get_point_from_homography((0, h-1), H)
c4 = get_point_from_homography((w-1, h-1), H)
# w_new and h_new are the bounding dimensions of the warped image
w_new = np.max([c4[0]-c1[0], c4[0]-c3[0], c2[0]-c1[0], c2[0]-c3[0]])
```

```
        h_new = np.max([c4[1]-c1[1], c4[1]-c2[1], c3[1]-c1[1], c3[1]-c2[1]])
        # x_offset and y_offset are the minimum x, y coordinates of the warped image
        x_offset = np.min([c1[0], c3[0]])
        y_offset = np.min([c1[1], c2[1]])
        # Determine necessary offsets to shift the images so that they take up the entire image
        # This is done because the warp function ignores negative values produced by homography transformation
        # So if the offsets computed earlier are negative, the images need to be shifted
        if y_offset < 0:
            y_shift = -y_offset
        else:
            y_shift = 0

        if x_offset < 0:
            x_shift = -x_offset
        else:
            x_shift = 0

        # Offset matrix - since the homography may give negative values after warping
        # The image needs to be shifted accordingly to keep the coordinates positive
        T = np.array([[1, 0, x_shift],
                      [0, 1, y_shift],
                      [0, 0, 1]])

        # Warp the second image based on the Homography and offset matrix with the new image size
        # @ is the notation for matrix multiplication
        warped_image = cv2.warpPerspective(imgset[1], T @ H, (max(w_new+x_offset, w), max(h_new, h)))

        mosaic = warped_image.copy()

        # Paste the first image onto the warped one but limit the overlap
        # instad of going the full width w, only got to the point between w and where the warped image begins
        mosaic[y_shift: h + y_shift, x_shift: int((w + c1[0]) / 2)] = imgset[0][0: h, 0: int((w + c1[0]) / 2)]

        # Blend pixels
        # Easy method being used to check whether the are is in the overlap:
        # If that pixel is black, then it has not been filled so put the old image there
        r_max = ((h / 2) ** 2 + (w / 2) ** 2) ** 0.5
        for i in range(0, h):
            for j in range(x_offset-1, w):
                if (warped_image[i+y_shift][j+x_shift] == np.array((0, 0, 0))).all() == 0:
                    thresh = 5
                    window = warped_image[i+y_shift-thresh: i+y_shift+thresh+1, j+x_shift-thresh: j+x_shift+thresh+1]
                    flag = any(np.array((0, 0, 0)) in sublist for sublist in window)
                    if flag is False:
                        p2 = get_point_from_homography((j + x_shift, i + y_shift), H_I)

                        r1 = 1 - np.sqrt((i - h / 2) ** 2 + (j - w / 2) ** 2) / r_max
                        r2 = 1 - np.sqrt((p2[1] - h / 2) ** 2 + (p2[0] - w / 2) ** 2) / r_max

                        temp = (r1 * imgset[0][i][j].astype(np.float32).astype(np.float32) + r2*warped_image[i + y_shift][j +
x_shift].astype(np.float32)) / (r1 + r2)
                        mosaic[i + y_shift][j + x_shift] = temp

        # Show the final mosaic
        plt.figure()
        plt.imshow(mosaic)
        plt.axis('off')
        plt.show()


if __name__ == "__main__":
    main()
```