Image Mosaicing

By Arvinder Singh and Uday Raghuvanshi

ABSTRACT

The aim of this project is to find corner features in multiple images using Harris corner detection, automatically find corresponding features using Normalized Cross-correlation, estimate homography using the best correspondences. Finally we warp one image into coordinate system of second one to produce a mosaic containing the union of all pixels in the two images.

For the project, we have used python as the programming language and used modules including cv2, numpy and os.

DESCRIPTION OF ALGORITHM

In order to implement Image Mosaicing, we started by forming an ImageMosaicing class which has several methods to perform the steps involved in forming a mosaic. The instance of the image mosaicing class takes 'array_of_images' as an array of grayscale images, 'type_of_derivative_filter' to choose from sobel or prewitt derivative filter, 'threshold' for extracting corners, 'hc_window_size' as the size of window used to obtain r_score for harris corners, 'ncc_window' as the window size of the template used while performing Normalized Cross-Correlation and 'ransac iterations' as the number of times we perform ransac to get the best Homography matrix as inputs.

The **ImageMosaicing** class is **initialized** as follows:

```
import os
import numpy as np
import cv2
import sys
np.set_printoptions(threshold=sys.maxsize)


class ImageMosaicing:
    def __init__(self, array_of_images: list, type_of_derivative_filter: str, hc_window_size: tuple,
                 ncc_threshold: float, ncc_window: tuple, ransac_iterations: int):
        self.array_of_images = array_of_images
        self.type_of_derivative_filter = type_of_derivative_filter
        self.window_size = hc_window_size
        self.threshold = ncc_threshold
        self.ncc_window = ncc_window
        self.iterations = ransac_iterations
```

Fig: ImageMosaicing class initialization

The description of each method of our algorithm is as follows:

1. derivative: We start by computing the derivative of the two images. The 'derivative' method reads all the gray scale images in the 'array_of_images' (two in our case), type of derivative filter- 'prewitt' or 'sobel' and finally returns their derivatives in x and y direction as i_x and i_y arrays.

```python
def derivative(self):
    i_x = []
    i_y = []
    for image in self.array_of_images:
        image = cv2.boxFilter(image, -1, (3, 3))
        if self.type_of_derivative_filter.lower() == "sobel":
            sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
            sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
            i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
            i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
        elif self.type_of_derivative_filter.lower() == "prewitt":
            prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
            prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
            i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
            i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
        else:
            print("Incorrect input")
            return
    return i_x, i_y
```

Fig: derivative method

2. harris_corner_detector: This method takes the derivatives i_x and i_y, forms a window for corner detection according to the hc_window_size. For each pixel, it iterates over all the neighbouring pixels in the window to obtain the r_score. Then it thresholds the r_score value to keep only relevant corner features, setting others to 0. We use the value of k (empirically determined constant as 0.06). This returns the array of of matrices of same size as image containing the r_scores

```python
def harris_corner_detector(self, k=0.06):
    i_x, i_y = self.derivative()
    harris_r_array = []
    for ind in range(len(i_x)):
        fx = i_x[ind]
        fy = i_y[ind]
        harris_r = np.zeros(np.shape(fx))
        rows, columns = np.shape(fx)[0], np.shape(fx)[1]
        for row in range(rows):
            for column in range(columns):
                i = row - self.window_size[0] // 2
                j = column - self.window_size[0] // 2
                i_x_2 = 0
                i_y_2 = 0
                i_x_y = 0
                while i <= row + self.window_size[0] // 2:
                    while j <= column + self.window_size[0] // 2:
                        if 0 < i < rows and 0 < j < columns:
                            i_x_2 += np.square(fx[i][j])
                            i_y_2 += np.square(fy[i][j])
                            i_x_y += fx[i][j] * fy[i][j]
                        j += 1
                    i += 1
                m = np.array([[i_x_2, i_x_y], [i_x_y, i_y_2]])
                r_score = np.linalg.det(m) - k * (np.square(m[0][0] + m[1][1]))
                if self.threshold < r_score:
                    harris_r[row][column] = r_score
        harris_r_array.append(harris_r)
    return harris_r_array
```

Fig: harris_corner_detector

3. non_max_suppression: We use this function to obtain a sparse set of harris corners. To perform NMS, we split each matrix returned by harris_corner_detector into tiles of size 64x64 pixels. For each tile, we only keep those r_score values which are maximum in a window of 5x5, keeping others as 0. So, we essentially have one corner in a window of 5x5. Then we sort all the r_score values in the tile and keep only the 10 largest values. Thus, each tile gives us at most 10 corner features. The value of window and the total number of corners to be retained in a tile can be changed. This returns an array of matrices (size 2 in our case) with a sparse set of corner features.

```
def non_max_supression(self):
    harris_r_array = self.harris_corner_detector()
    nms_harris_arr = []
    pix_dist = 5
    h, w = np.shape(harris_r_array[0])
    for ind in harris_r_array:
        nms_harris = np.zeros((h, w))
        for i in range(0, h, 64):
            for j in range(0, w, 64):
                r_array = []
                for m in range(i + pix_dist, i + 64 - pix_dist):
                    for n in range(j + pix_dist, j + 64 - pix_dist):
                        if m < h and n < w:
                            if ind[m, n] > 0 and ind[m, n] == np.max(
                                    ind[m - pix_dist:m + pix_dist + 1, n - pix_dist:n + pix_dist + 1]):
                                r_array.append((ind[m, n], m, n))
                if len(r_array) < 10:
                    for p in range(len(r_array)):
                        nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]
                else:
                    r_array = sorted(r_array, reverse=True)
                    for p in range(10):
                        nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]

        nms_harris_arr.append(nms_harris)
    return nms_harris_arr
```

Fig: non_max_suppression

4. disp_img: This method takes in the two rgb images where we detected corners and overlays the corner points detected with harris_detector or non_max_suppresion.

```
def disp_img(self, img1, img2):
    nms_harris = self.non_max_supression()
    rgb_imgs_arr = [img1, img2]
    corner_imgs = []
    for k in range(len(nms_harris)):
        rgb = np.copy(rgb_imgs_arr[k])
        r, c = np.shape(rgb)[0], np.shape(rgb)[1]
        for i in range(r):
            for j in range(c):
                if nms_harris[k][i, j] > 0:
                    rgb[i, j] = [0, 0, 255]
        corner_imgs.append(rgb)
    return corner_imgs
```

Fig: disp_image

5. find_correspondences: This method takes as input the sparse set of corner features of the two images and finds correspondences between them using Normalized Cross-Correlation (NCC). For one image, it forms a window of size 'ncc_window' and iterates over the image looking for potential corner features (corners_0[row][column]>0). Upon encountering a corner, it forms a template of size ncc_window around it. Then it slides that template in the second image, calculating the NCC scores for all corners in the second image. If the NCC score is above 0.90, it appends the location of pixels in the two images in two lists, correspondences_picture0 and correspondences_picture1. Finally, it returns us the corresponding features in the two images.

```
def find_correspondences(self):
    corners_array = self.non_max_supression()
    corners_0 = corners_array[0]
    corners_1 = corners_array[1]
    rows, columns = np.shape(corners_0)[0], np.shape(corners_0)[1]
    correspondences_picture0 = []
    correspondences_picture1 = []
    for row in range(rows):
        for column in range(columns):
            if corners_0[row][column] > 0:
                a = row - (self.ncc_window[0] // 2)
                b = row + (self.ncc_window[0] // 2) + 1
                c = column - (self.ncc_window[0] // 2)
                d = column + (self.ncc_window[0] // 2) + 1
                if 0 < a < rows - self.ncc_window[0] and 0 < c < columns - self.ncc_window[0]:
                    template = np.array(self.array_of_images[0][a:b, c:d])
                    max_ncc = 0
                    max_row = None
                    max_column = None
                    for row2 in range(rows):
                        for column2 in range(columns):
                            if corners_1[row2][column2] > 0:
                                a2 = row2 - (self.ncc_window[0] // 2)
                                b2 = row2 + (self.ncc_window[0] // 2) + 1
                                c2 = column2 - (self.ncc_window[0] // 2)
                                d2 = column2 + (self.ncc_window[0] // 2) + 1
                                if 0 < a2 < rows - self.ncc_window[0] and 0 < c2 < columns - self.ncc_window[0]:
                                    match_to = np.array(self.array_of_images[1][a2:b2, c2:d2])
                                    f = (match_to - match_to.mean())/(match_to.std() * np.sqrt(match_to.size))
                                    g = (template - template.mean())/(template.std() * np.sqrt(template.size))
                                    product = f * g
                                    stds = np.sum(product)
                                    if stds > max_ncc:
                                        max_ncc = stds
                                        max_row = row2
                                        max_column = column2
                    if max_row is not None:
                        if max_ncc > 0.90:
                            correspondences_picture0.append([row, column])
                            correspondences_picture1.append([max_row, max_column])
    return correspondences_picture0, correspondences_picture1
```

Fig: find_correspondences

6. **find_homography**: This is a static method inside ImageMosaicing class which is used to estimate the homography using correspondences. This method takes as input a list of lists which contains the pixel values of the correspondences for source and the destination image. We are calculating a least squares homography using Algebraic Distance, ||h||=1 condition in order to avoid any division by 0.

```
@staticmethod
def find_homography(pts_src, pts_dst):
    pts_src = np.array(pts_src)
    pts_dst = np.array(pts_dst)

    # forming A matrix
    a_matrix = np.zeros((2 * len(pts_src), 9))
    for i in range(len(pts_src)):
        a_matrix[2 * i, 0] = pts_src[i][0]
        a_matrix[2 * i, 1] = pts_src[i][1]
        a_matrix[2 * i, 2] = 1
        a_matrix[2 * i, 6] = -pts_src[i][0] * pts_dst[i][0]
        a_matrix[2 * i, 7] = -pts_src[i][1] * pts_dst[i][0]
        a_matrix[2 * i, 8] = -pts_dst[i][0]

        a_matrix[2 * i + 1, 3] = pts_src[i][0]
        a_matrix[2 * i + 1, 4] = pts_src[i][1]
        a_matrix[2 * i + 1, 5] = 1
        a_matrix[2 * i + 1, 6] = -pts_src[i][0] * pts_dst[i][1]
        a_matrix[2 * i + 1, 7] = -pts_src[i][1] * pts_dst[i][1]
        a_matrix[2 * i + 1, 8] = -pts_dst[i][1]

    # Transposing A
    a_transpose = np.transpose(a_matrix)
    final_a_matrix = np.dot(a_transpose, a_matrix)
    u, d, u_transpose = np.linalg.svd(final_a_matrix, full_matrices=True)

    # h is the column of U associated with smallest eigen in D, which is the last value in D
    h = (u[:, 8])
    h = np.reshape(h, (3, 3))
    return h
```

Fig: find_correspondences

7. **ransac:** The correspondences we found using find_correspondences have many outliers. So, we implement ransac to eliminate them. This method takes the correspondences we found and randomly samples 4 correspondences. It then computes a homography from these 4 and maps all the corresponding points in one image to the points in second. Since we know the observed locations of corresponding points in the second image, we compute the euclidean distance between predicted and observed locations and if the distance is less than 5, we keep the corresponding points as inliers. We keep doing this for all the iterations and the homography that gives us the highest number of inliers is the one we keep. As a last step, we find the homography using the set of all inliers to attain the best homography matrix.

```
def ransac(self):
    corres_0, corres_1 = self.find_correspondences()
    max_inliers = 0
    homography = None
    max_inliers_0 = []
    max_inliers_1 = []
    for it in range(self.iterations):
        random_sampled_indx = []
        pts_src = []
        pts_dst = []
        i = 0
        while i < 4:
            random_sample = int(np.random.randint(0, high=len(corres_0) - 1, size=1, dtype=int))
            if random_sample not in random_sampled_indx:
                pts_src.append(corres_0[random_sample])
                pts_dst.append(corres_1[random_sample])
                random_sampled_indx.append(random_sample)
                i += 1
        h = self.find_homography(pts_src, pts_dst)
        j = 0
        inliers = 0
        inliers_0 = []
        inliers_1 = []
        while j < len(corres_0):
            src = np.transpose(np.array([[corres_0[j][0], corres_0[j][1], 1]]))
            est = np.matmul(h, src)
            if est[2][0] == 0:
                j += 1
                continue
            est = est / est[2][0]
            dist = np.sqrt(np.power(est[0]-corres_1[j][0], 2)+np.power(est[1]-corres_1[j][1], 2))
            if dist < 5:
                inliers += 1
                inliers_0.append(corres_0[j])
                inliers_1.append(corres_1[j])
            j += 1

        if inliers > max_inliers:
            max_inliers = inliers
            max_inliers_0 = inliers_0
            max_inliers_1 = inliers_1
            if max_inliers > 3:
                homography = self.find_homography(max_inliers_0, max_inliers_1)
    return homography, max_inliers_0, max_inliers_1
```

Fig: ransac

8. feature_matching: This method takes as input the two images whose features need to be matched. It takes the correspondences found in find_correspondences, concatenates the two images and draws lines joining the two images. It returns an image showing potential corner feature location matches between the two images.

```
def feature_matching(self, img_l, img_r, matching_type: str):
    corres_0, corres_1 = self.find_correspondences()
    img_l = np.copy(img_l)
    img_r = np.copy(img_r)
    shift = np.shape(img_r)[1]
    full_img = np.concatenate((img_l, img_r), axis=1)
    if matching_type == 'before ransac':
        for i in range(len(corres_0)):
            color = list(np.random.random_sample(size=3) * 256)
            cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] + shift, corres_1[i][0]), color,
                     thickness=1)

    elif matching_type == 'after ransac':
        inliers_0, inliers_1 = self.ransac()[1], self.ransac()[2]
        for i in range(len(corres_0)):
            if corres_0[i] in inliers_0:
                cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] + shift, corres_1[i][0]),
                         (0, 255, 0), thickness=1)
            else:
                cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] + shift, corres_1[i][0]),
                         (0, 0, 255), thickness=1)
    return full_img
```

Fig: feature_matching

9. image_stitching: It takes as input the two images that need to be stitched together, then we compute the inverse of the homography matrix to find the homography from right to left image. Then we warp the 4 corners of the left image using the inverse homography matrix to estimate the size of the panorama. Then we implement backward warping to find the warped image on the source image without any gaps. There are two methods we tried to blend the pixels- the output from feathering was a lot better than simple averaging so we implemented feathering in our code. The function then finally returns the panorama.

```
def image_stitching(self, img_1, img_2):

    h = np.linalg.inv(self.ransac()[0])
    y, x = img_2.shape[:2]
    p_1 = [[0], [0], [1]]
    p_2 = [[0], [x - 1], [1]]
    p_3 = [[y - 1], [0], [1]]
    p_4 = [[y - 1], [x - 1], [1]]

    p_1 = np.matmul(h, p_1)
    p_2 = np.matmul(h, p_2)
    p_3 = np.matmul(h, p_3)
    p_4 = np.matmul(h, p_4)

    p_1 = p_1 / p_1[2][0]
    p_2 = p_2 / p_2[2][0]
    p_3 = p_3 / p_3[2][0]
    p_4 = p_4 / p_4[2][0]
```

Fig: image_stitching

```
y_min = round(min(p_1[0], p_2[0], p_3[0], p_4[0])[0])
x_min = round(min(p_1[1], p_2[1], p_3[1], p_4[1])[0])
y_max = round(max(p_1[0], p_2[0], p_3[0], p_4[0])[0])
x_max = round(max(p_1[1], p_2[1], p_3[1], p_4[1])[0])

pano = np.zeros((y_max - y_min + 10, x_max - x_min + round(np.shape(img_1)[1] * 0.3), 3), dtype=np.uint8)
pano[abs(y_min):np.shape(img_1)[0] + abs(y_min), :np.shape(img_1)[1], :] = img_1

for row in range(pano.shape[0]):
    for column in range(pano.shape[1]):
        est = np.matmul(np.linalg.inv(h), np.array([[row], [column], [1]]))
        est = est / est[2][0]
        est_y = round(est[0][0])
        est_x = round(est[1][0])
        if 0 <= est_x < img_2.shape[1] and 0 <= est_y + y_min < img_2.shape[0]:
            if pano[row][column][:].any() > 0:
                w_1 = min(row, column, img_1.shape[0] - row - y_min, img_1.shape[1] - column)
                w_2 = min(est_y + y_min, est_x, img_2.shape[0] - (est_y + y_min),
                          img_2.shape[1] - est_x)
                if w_1 + w_2 != 0:
                    feather_1 = np.array(pano[row][column][:]).astype(np.uint64) * w_1 / (w_1 + w_2)
                    feather_2 = np.array(img_2[est_y + y_min][est_x][:]).astype(np.uint64) * w_2 / (w_1 + w_2)
                    pano[row][column][:] = (feather_1 + feather_2).astype(np.uint8)

                else:
                    pano[row][column][:] = 0.5 * np.array(pano[row][column][:]) \
                                           + 0.5 * np.array(img_2[abs(est_y + y_min)][est_x][:])
            else:
                pano[row][column][:] = img_2[abs(est_y + y_min)][est_x][:]

return pano
```

Fig: image_stitching(continued)

Experiments:-

1. ***Detecting Harris corners:-*** We tried computing Harris corners using different derivative filters, Harris corner window size, non-max suppression thresholds and thresholding at different values of normalized cross-correlation.

Below are some of the observations and conclusions that we were able to make by performing different experiments:-

derivative_filter = prewitt
hc_window_size = (5, 5)
r_score_threshold = 100
ncc_value = 0.9
ncc_window = (7, 7)
k = 0.06
nms_threshold = 10

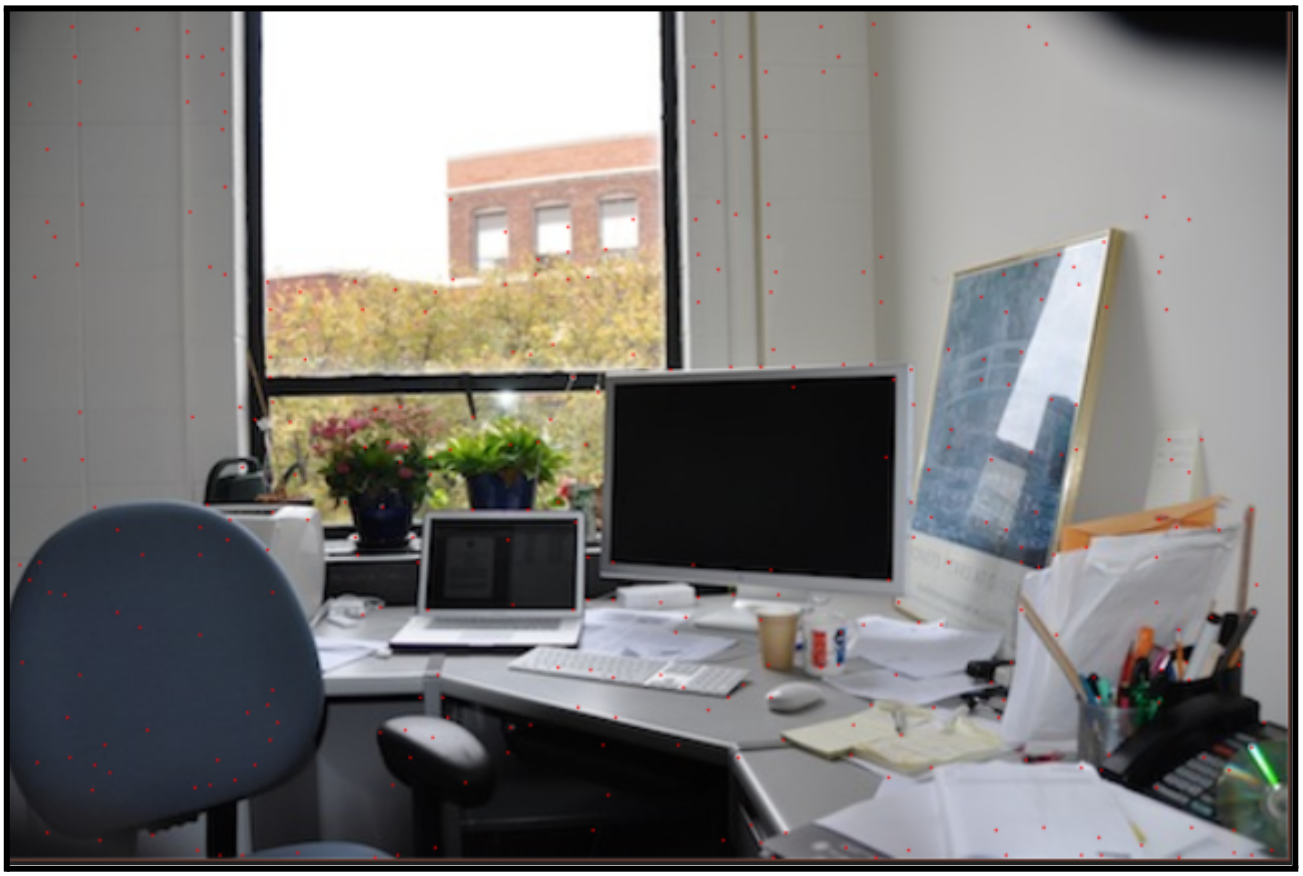


Fig: harris corner(1)

derivative_filter = sobel

hc_window_size = (3, 3)

r_score_threshold = 100000

ncc_value = 0.9

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

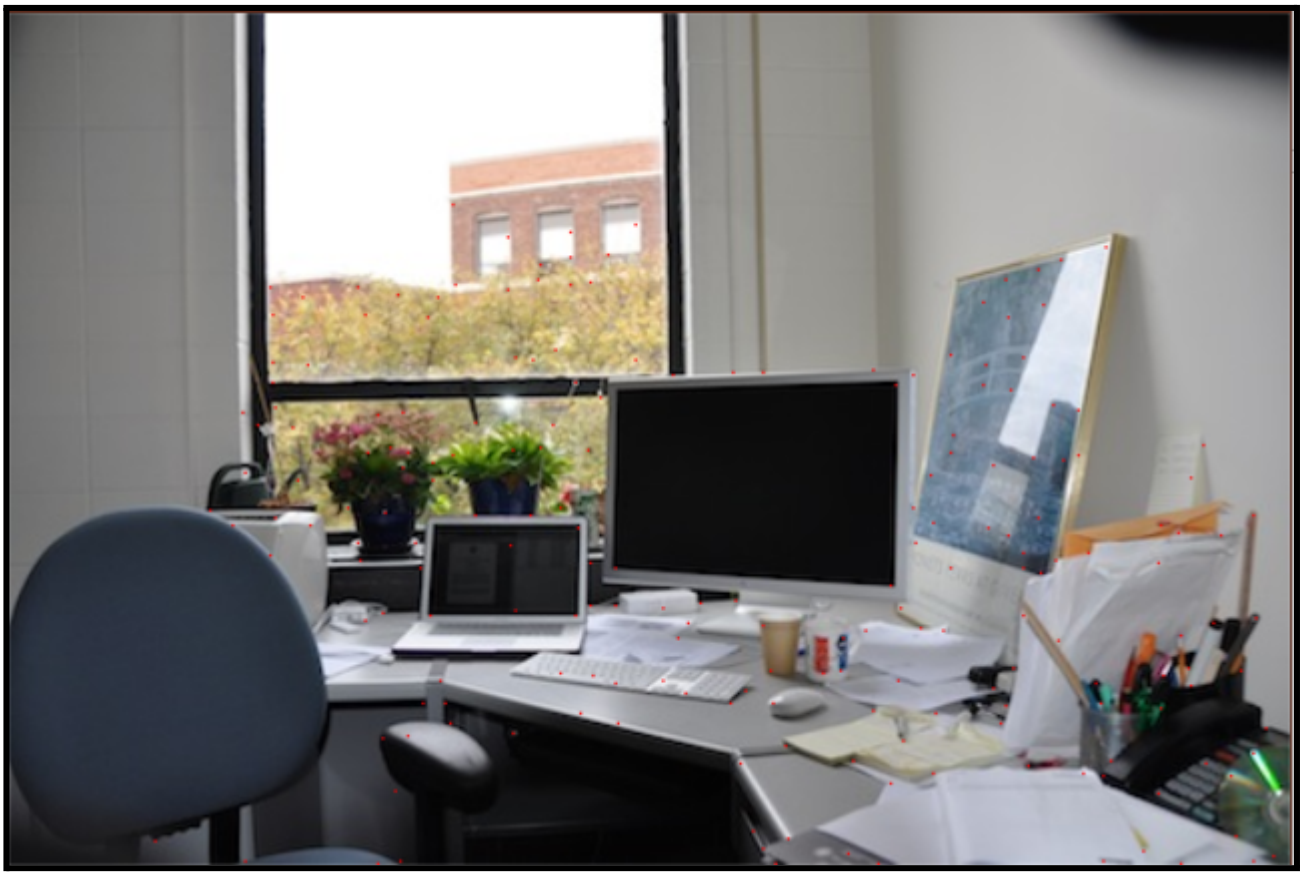


Fig: harris corner(2)

As we increase the r_score threshold for harris corners, we can see that we get more corners and very few edges. This is desirable as corners are great for feature matching. Also, the larger hc_window we use the more likely we are to get correct corners but increasing the window size too much will take away corners that were changed by intensity changes etc. so we don't want to increase our hc_window size or r_score_threshold too much. We also observed that increasing the nms threshold results in losing important corners, so we want to find a sweet spot that reduces the number of points but keeps the important corners for feature matching. Since there were many corners in the first case, we can use a higher nms window to suppress the features since the threshold is too low and we will have a lot of features that are not corners. But in the second case we don't want to have the same sized nms window as that will result in loss of important corners since we already have very few of them.

2. Feature Matching:- Now that we have found the Harris corners by thresholding and non max suppression, we want to now match features from one image to the other. For this we calculate the normalized cross correlation of image patches centered at each corner and then threshold the ncc_value to get rid of incorrect corner matches.

derivative_filter = prewitt

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.9

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

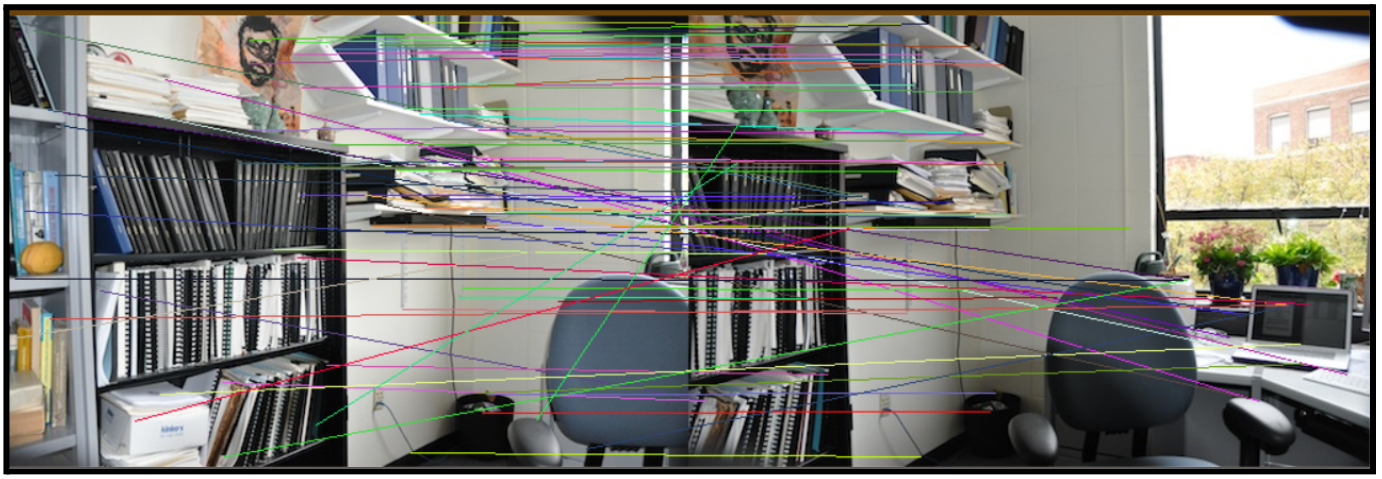


Fig: Feature matching(1)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.9

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

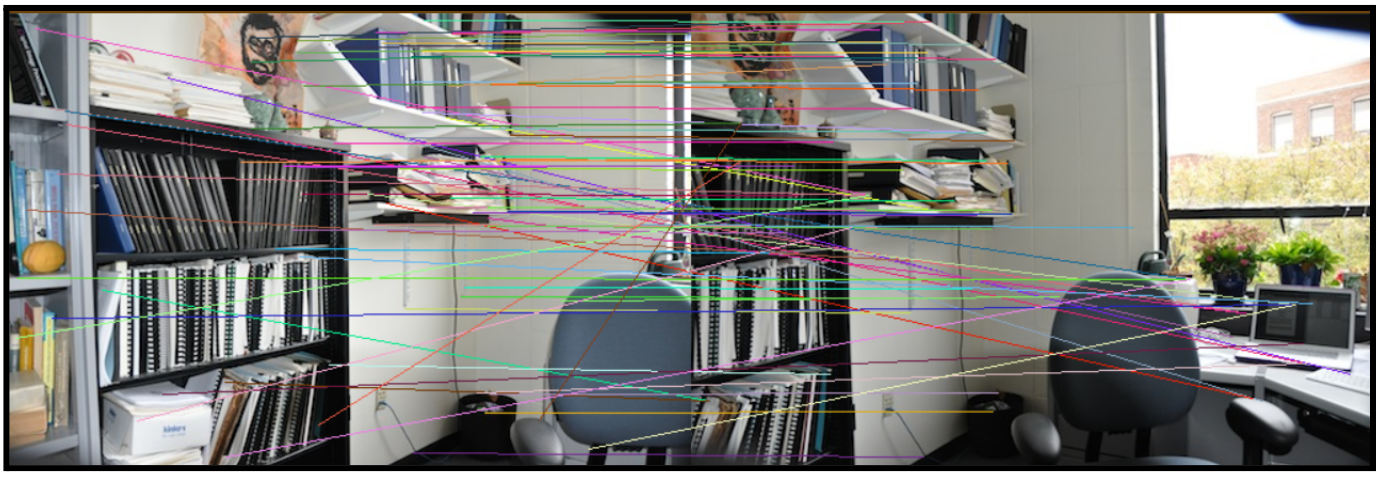


Fig: Feature matching(2)

derivative_filter = prewitt

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.95

ncc_window = (9, 9)

k = 0.06

nms_threshold = 5

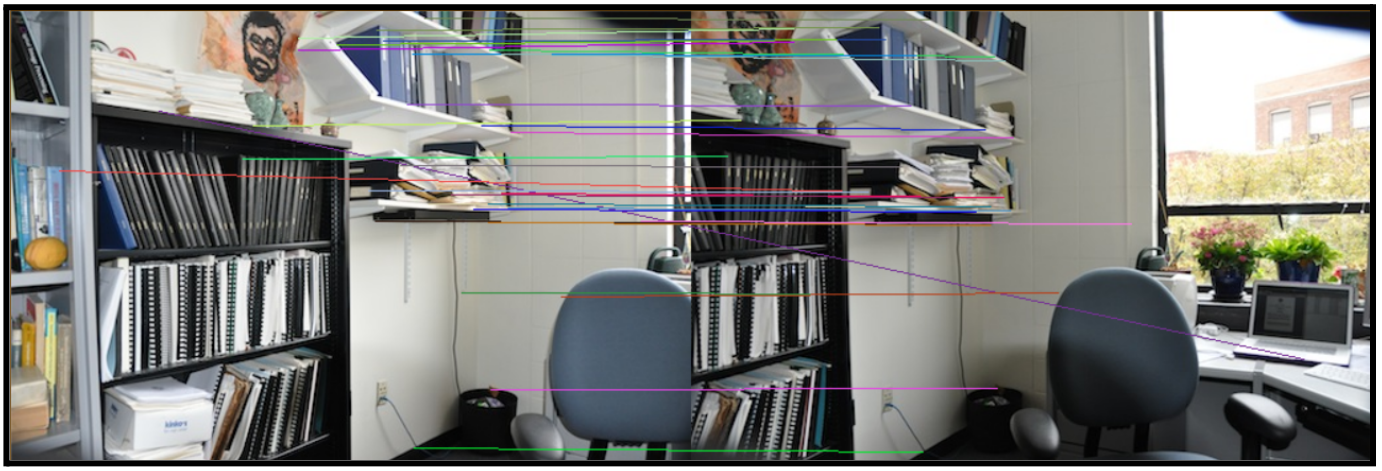


Fig: Feature matching(3)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.95

ncc_window = (3, 3)

k = 0.06

nms_threshold = 5

From the above experiments, we concluded that both Prewitt and Sobel derivative filters are equally good and output the same results. As we increase the size of the ncc_window and the ncc threshold value, we get fewer false positives but we lose a lot of important data which might be important to calculate homography. So we want to use ncc_window that is not very big in size and is neither very small, and we want to use ncc_threshold value of 0.9 which gives us fewer false positives without losing too much of the features. In the end we tested that a very small ncc_window gives us a lot of incorrect features even at a 0.95 ncc_threshold value so we should not use a very small ncc_window. For a large ncc_window and a large ncc threshold value we observe very few false positives but the features are very sparse so we want to use an intermediate threshold and window size.

3. RANSAC:- After getting the point correspondences, we now run RANSAC on these points to get rid of outliers. We sample 4 random points and then calculate the homography using those 4 points. Then we run RANSAC to compute the number of inliers and the outliers. Whatever homography gives us the maximum number of inliers we then finally use those inliers to calculate the final homography matrix using least squares method. Below are some experiments that we performed to estimate the inliers by changing the number of iterations and the minimum distance to be an inlier

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 10

Minimum distance to be an inlier = 5

Fig: RANSAC(1)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 100

Minimum distance to be an inlier = 5

Fig: RANSAC(2)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 10000

Minimum distance to be an inlier = 5

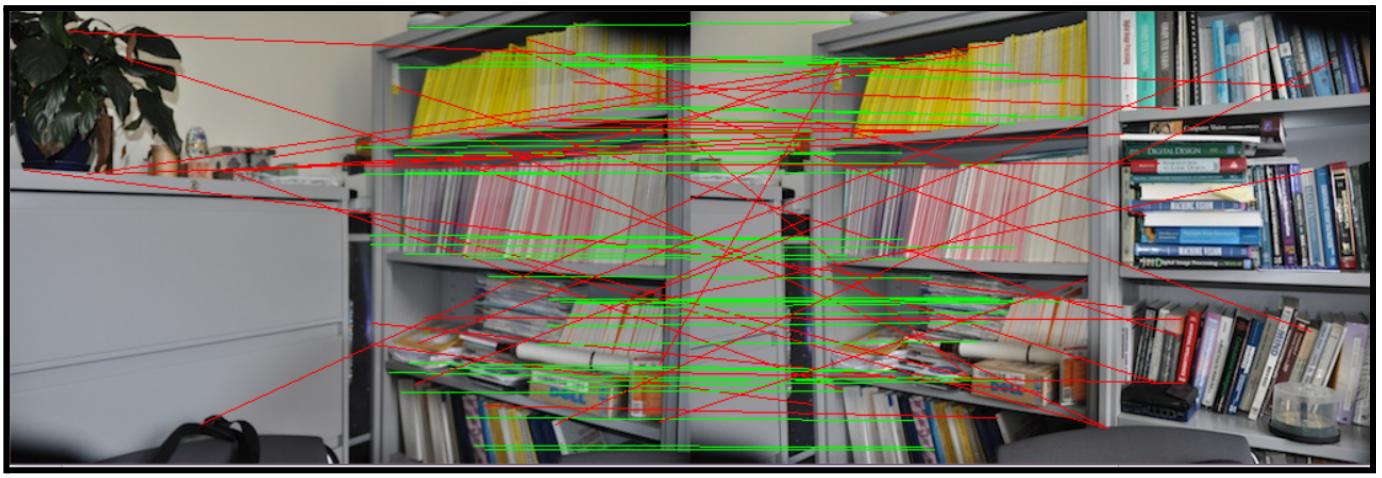


Fig: RANSAC(3)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 10000

Minimum distance to be an inlier = 1

Fig: RANSAC(4)

We performed RANSAC for different iterations and different minimum distance for an inlier and as we expected- increasing the number of iterations for RANSAC gave us more and more inliers and a better homography estimate but it increases the cost of computation and after a certain number of iterations the number of inliers increase by 1 or maybe 2. Similarly for a lower minimum distance for a point to be inlier, we get very few correspondences even after a large number of iterations. So we decided to set the minimum distance for a point to be an inlier as 5 and perform 1000 iterations as it gives us the required number of correspondences at an optimal computational cost.

4. Image Stitching:- After we get the homography using all the inliers, now we want to stitch the images to form a panorama. There are two types of image blending techniques that we discussed in class- 1) Averaging and 2) Feathering. Below are the results using both the methods:-

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 1000

Minimum distance to be an inlier = 5

Image Blending = Averaging

Fig: Panorama(1)

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.90

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 1000

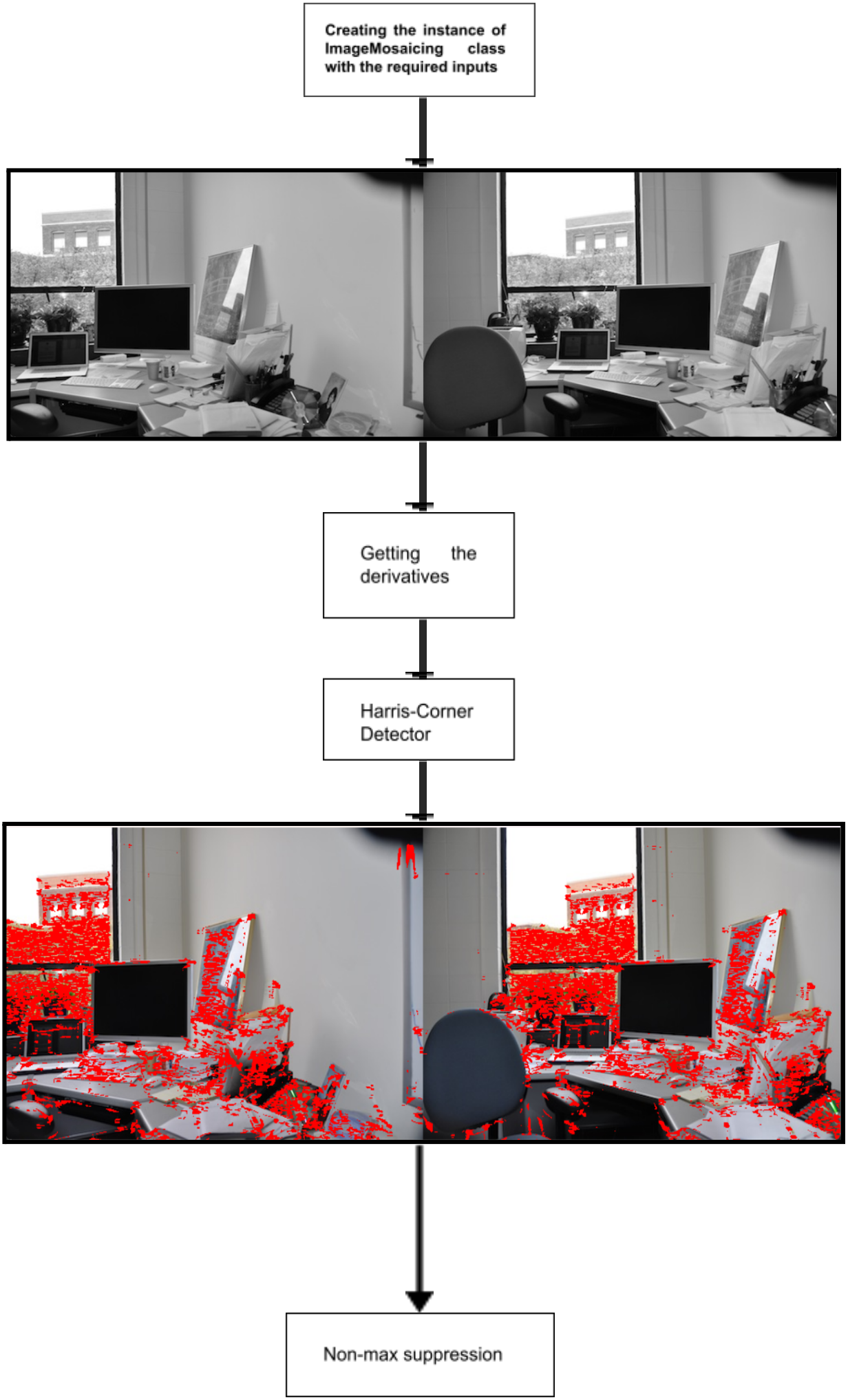
Minimum distance to be an inlier = 5

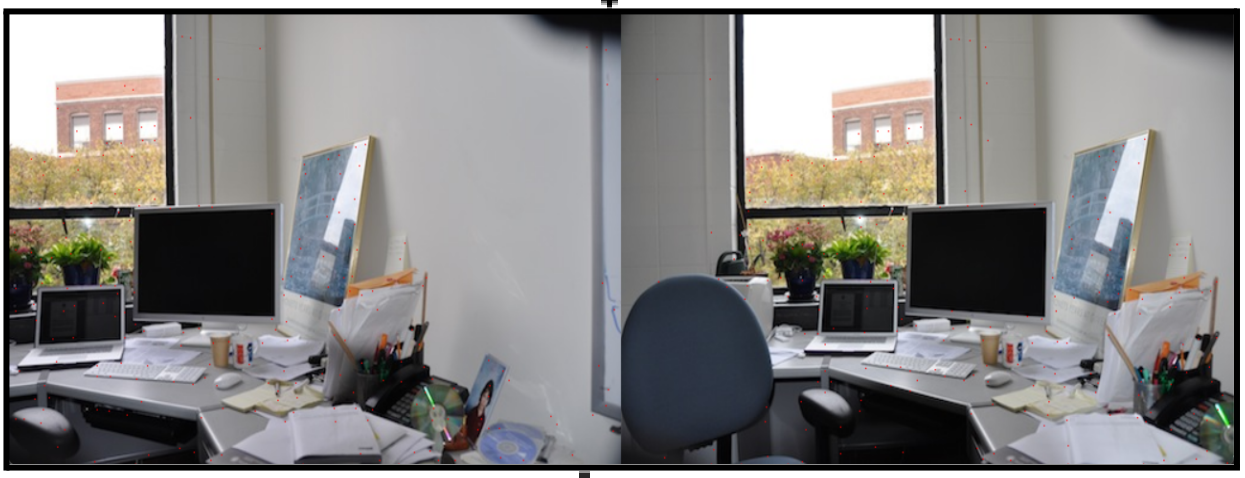
Image Blending = Feathering

Fig: Panorama(2)

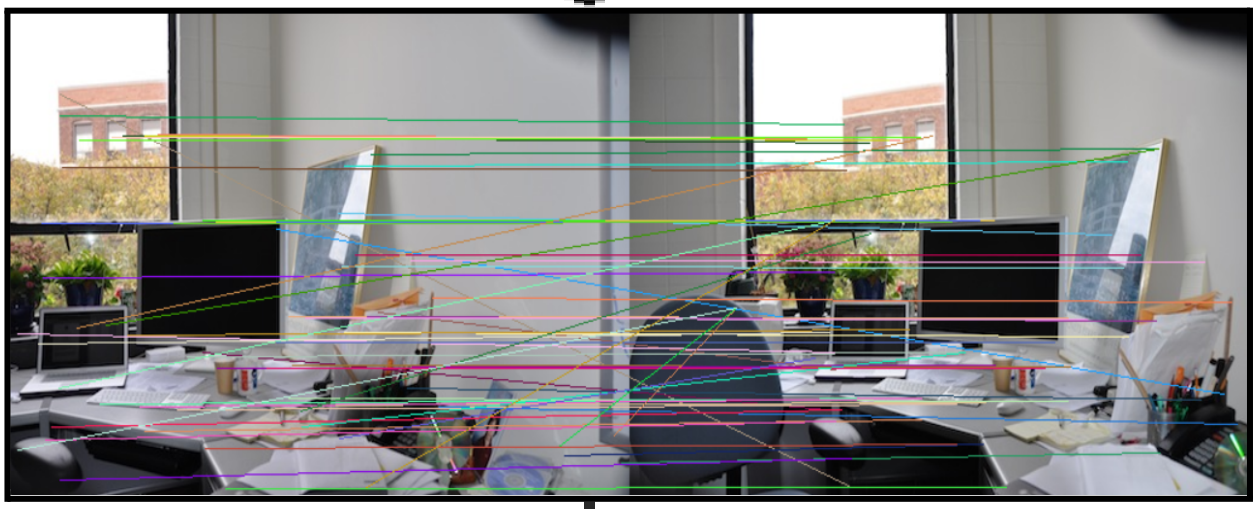
It is very visible from the above panorama images that using Feathering to blend the images is way better than averaging to blend the images. We can clearly see in the first panoramic image- the area of overlap is clearly visible but in the second image we get a very nice panoramic image with area of overlap almost negligibly visible.

FLOWCHART


Creating the instance of ImageMosaicing class with the required inputs
Getting the derivatives
Harris-Corner Detector
Non-max suppression



Feature matching
before ransac



Feature matching
after ransac

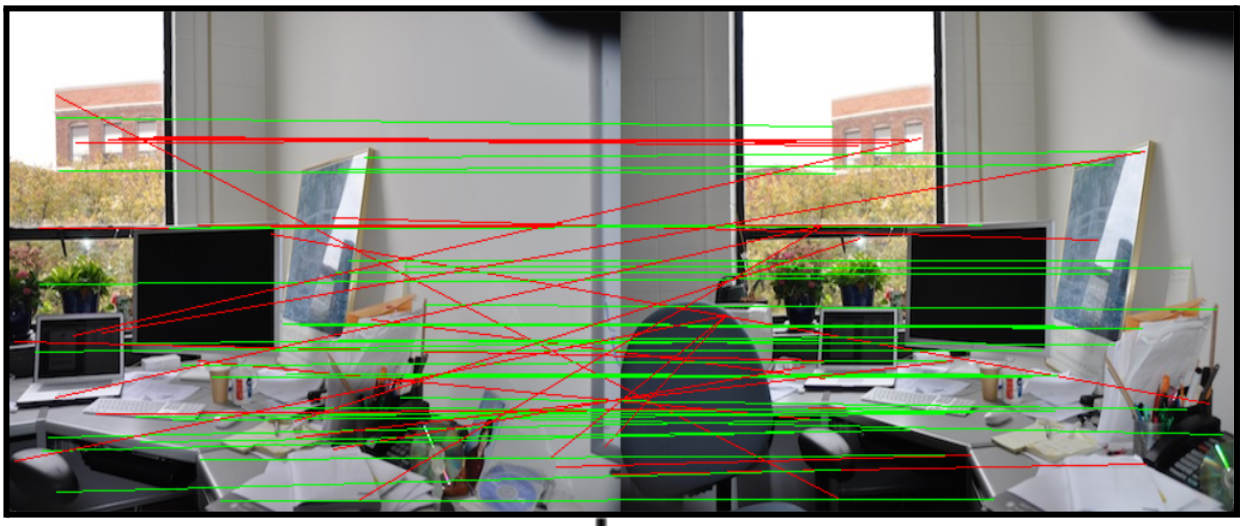


Image stitching

IMAGE SHOWING POTENTIAL CORNER MATCHES

1. **From DanaOffice dataset:**

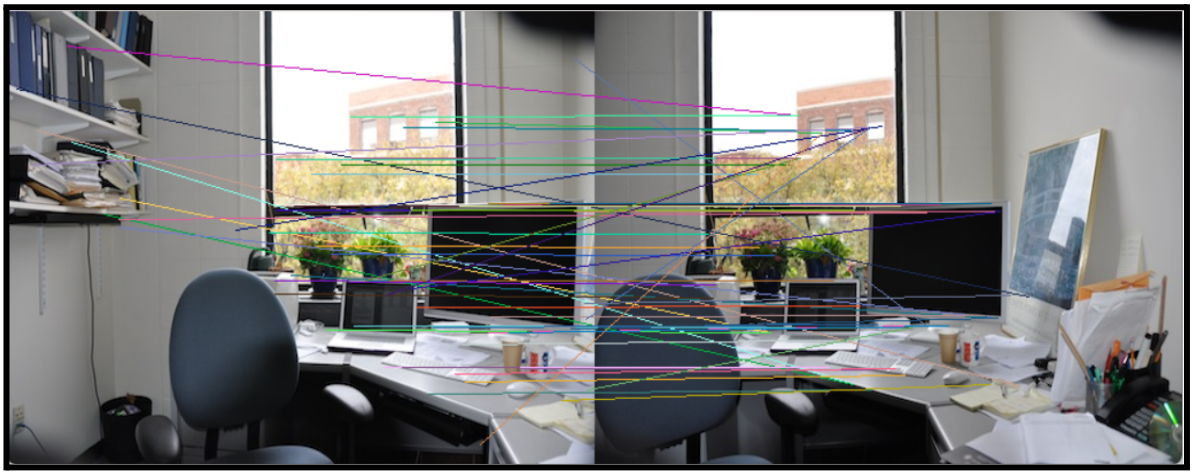


Fig: Before ransac

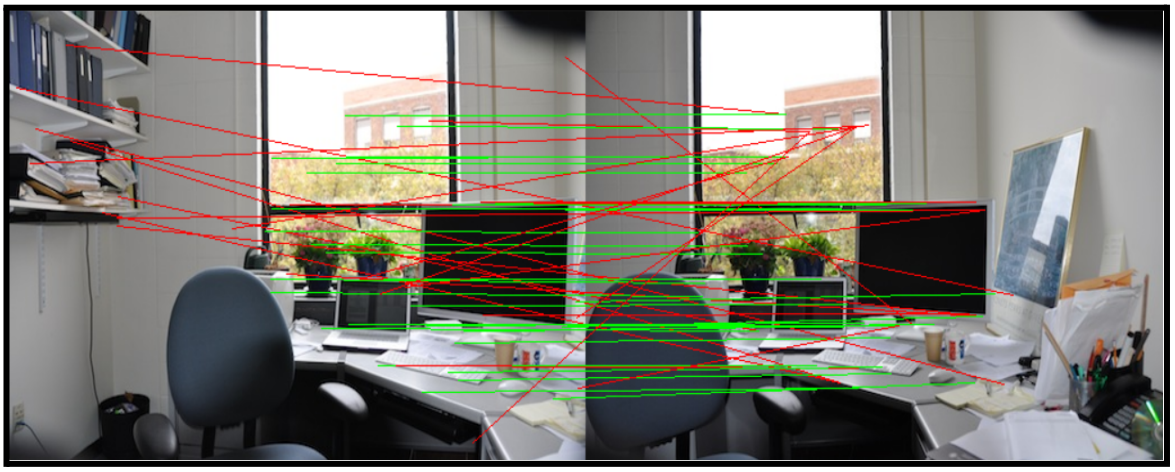


Fig: After RANSAC

2. **From DanahallWay1 dataset**

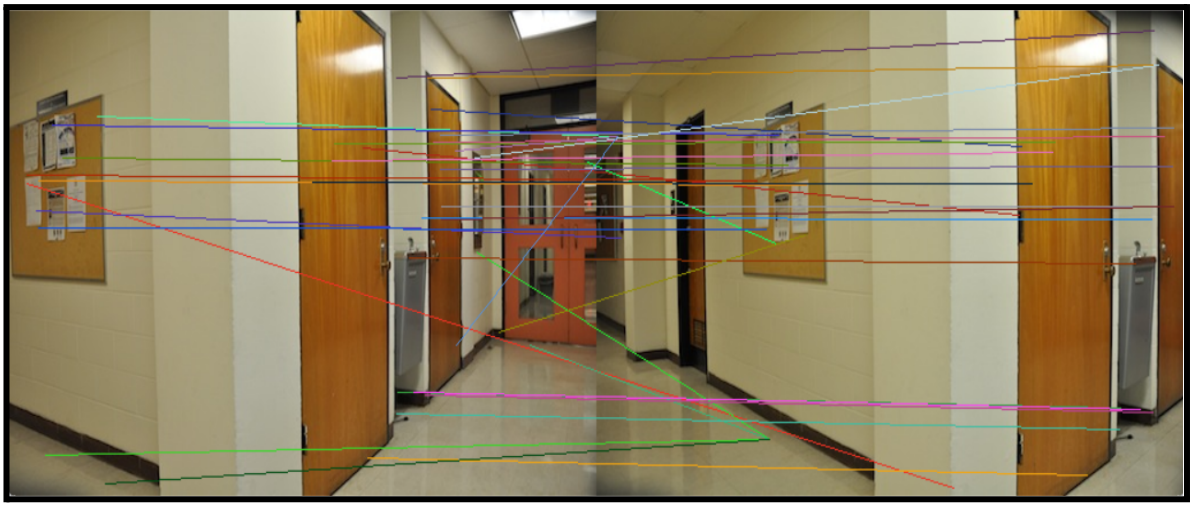


Fig: Before RANSAC

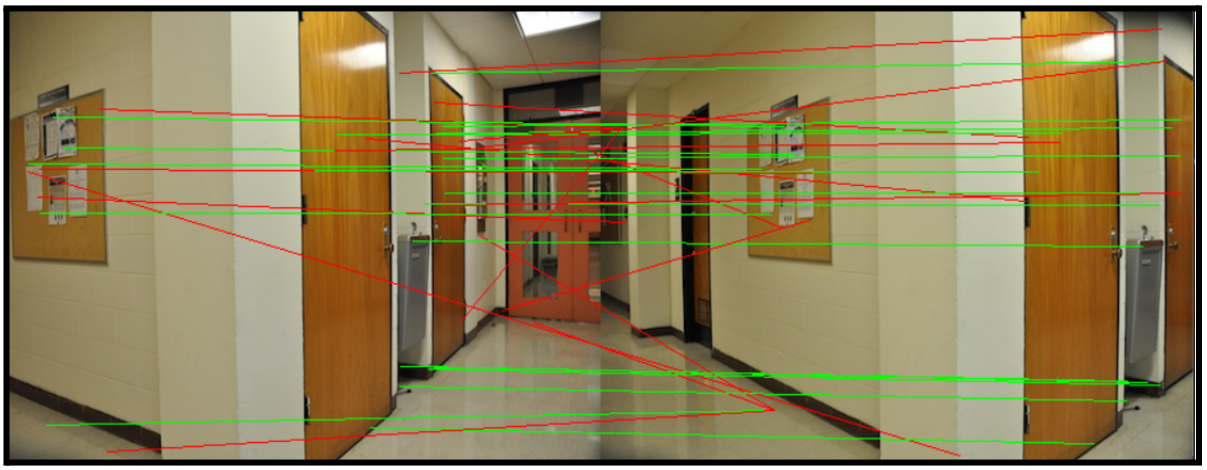


Fig: After RANSAC

FINAL MOSAICS

1. DanahallWay1 dataset:

Fig: Panorama(1)

Fig: Panorama(2)

2. DanaOffice dataset

Fig: Panorama(3)

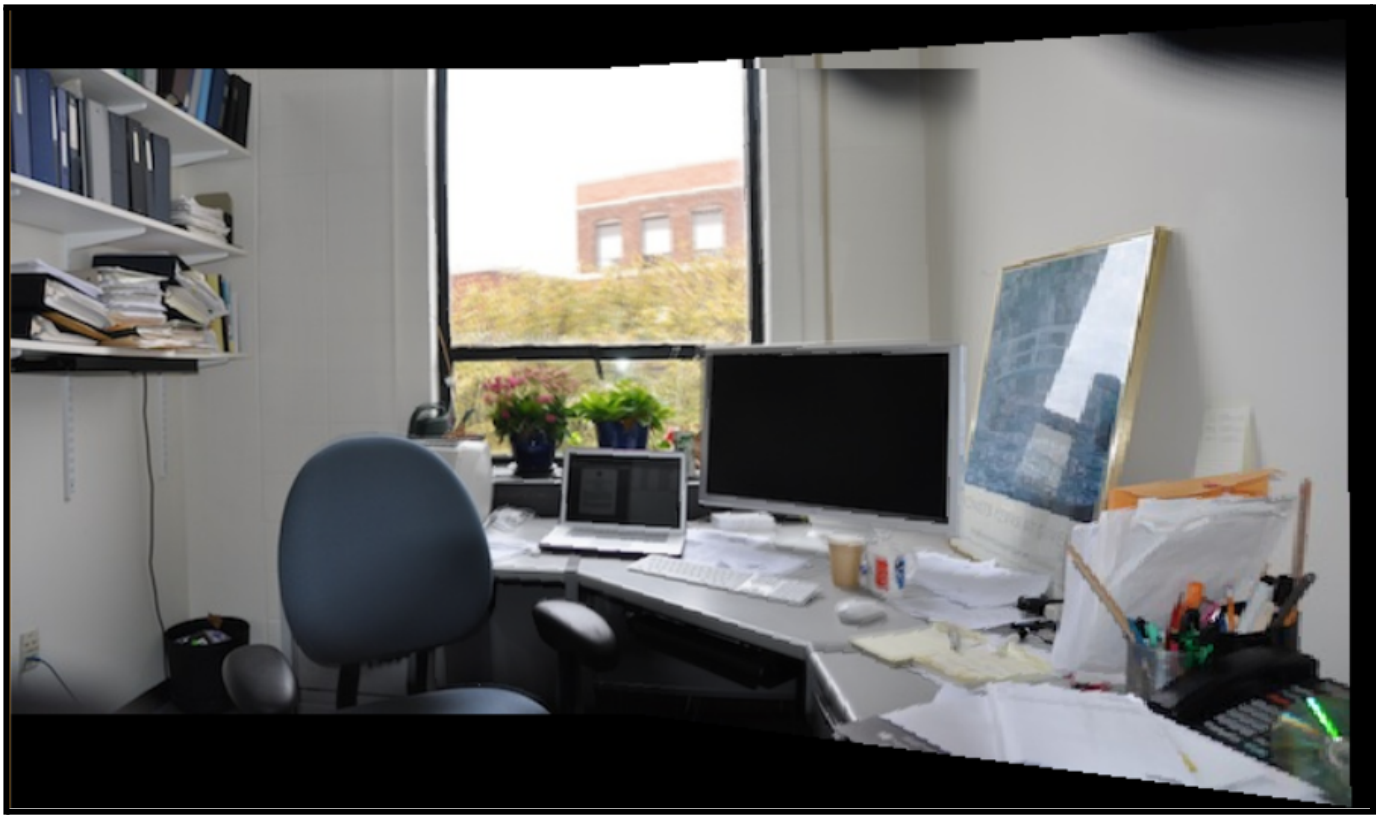


Fig: Panorama(4)

CONCLUSION

While performing our experiments, we made sure to check how our results vary upon changing the type and values of parameters used. We started with trying different derivative filters, namely Prewitt and Sobel and varied the window size for calculating Harris corners. We observed that a higher threshold for r-score gives us less false positives and increasing the window size by just the right amount will give us the desired corner features. We also talked about choosing a sweet spot for the nms_threshold so that we can retain the important corners for feature matching.

For feature matching, we used ncc. There we varied the window size for the template we use to find corresponding features. We observed that increasing the window size will lower the number of false positives since we match a bigger template but we might also lose on important corner points that would reduce our ability to get accurate homography. The ncc_threshold should not be too large or too small as it would affect the corner points we obtain.

While performing RANSAC, we saw that increasing the number of iterations gave us more inliers and accurate homography but it came at the cost of high computing power and time. Also, keeping the minimum distance between observed and predicted points too low resulted in very few points as inliers.

Finally, to obtain a panorama, we tried two different blending techniques, Averaging and Feathering. We saw that Feathering gave us better results and the visible area of overlap was negligible in this

APPENDIX

```
import os
import numpy as np
import cv2
import sys
np.set_printoptions(threshold=sys.maxsize)


class ImageMosaicing:
    def __init__(self, array_of_images: list, type_of_derivative_filter: str, hc_window_size:
tuple,
                 ncc_threshold: float, ncc_window: tuple, ransac_iterations: int):
        self.array_of_images = array_of_images
        self.type_of_derivative_filter = type_of_derivative_filter
        self.window_size = hc_window_size
        self.threshold = ncc_threshold
        self.ncc_window = ncc_window
        self.iterations = ransac_iterations

    def derivative(self):
        i_x = []
        i_y = []
        for image in self.array_of_images:
            image = cv2.boxFilter(image, -1, (3, 3))
            if self.type_of_derivative_filter.lower() == "sobel":
                sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
            elif self.type_of_derivative_filter.lower() == "prewitt":
                prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
            else:
                print("Incorrect input")
                return
        return i_x, i_y

    def harris_corner_detector(self, k=0.06):
        i_x, i_y = self.derivative()
        harris_r_array = []
        for ind in range(len(i_x)):
            fx = i_x[ind]
            fy = i_y[ind]
            harris_r = np.zeros(np.shape(fx))
            rows, columns = np.shape(fx)[0], np.shape(fx)[1]
            for row in range(rows):
                for column in range(columns):
                    i = row - self.window_size[0] // 2
                    j = column - self.window_size[0] // 2
                    i_x_2 = 0
                    i_y_2 = 0
                    i_x_y = 0
                    while i <= row + self.window_size[0] // 2:
                        while j <= column + self.window_size[0] // 2:
                            if 0 < i < rows and 0 < j < columns:
                                i_x_2 += np.square(fx[i][j])
                                i_y_2 += np.square(fy[i][j])
                                i_x_y += fx[i][j] * fy[i][j]
                            j += 1
                        i += 1
```

```
                    m = np.array([[i_x_2, i_x_y], [i_x_y, i_y_2]])
                    r_score = np.linalg.det(m) - k * (np.square(m[0][0] + m[1][1]))
                    if self.threshold < r_score:
                        harris_r[row][column] = r_score
            harris_r_array.append(harris_r)
        return harris_r_array

    def non_max_supression(self):
        harris_r_array = self.harris_corner_detector()
        nms_harris_arr = []
        pix_dist = 5
        h, w = np.shape(harris_r_array[0])
        for ind in harris_r_array:
            nms_harris = np.zeros((h, w))
            for i in range(0, h, 64):
                for j in range(0, w, 64):
                    r_array = []
                    for m in range(i + pix_dist, i + 64 - pix_dist):
                        for n in range(j + pix_dist, j + 64 - pix_dist):
                            if m < h and n < w:
                                if ind[m, n] > 0 and ind[m, n] == np.max(
                                        ind[m - pix_dist:m + pix_dist + 1, n - pix_dist:n +
pix_dist + 1]):
                                    r_array.append((ind[m, n], m, n))
                    if len(r_array) < 10:
                        for p in range(len(r_array)):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]
                    else:
                        r_array = sorted(r_array, reverse=True)
                        for p in range(10):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]

            nms_harris_arr.append(nms_harris)
        return nms_harris_arr

    def find_correspondences(self):
        corners_array = self.non_max_supression()
        corners_0 = corners_array[0]
        corners_1 = corners_array[1]
        rows, columns = np.shape(corners_0)[0], np.shape(corners_0)[1]
        correspondences_picture0 = []
        correspondences_picture1 = []
        for row in range(rows):
            for column in range(columns):
                if corners_0[row][column] > 0:
                    a = row - (self.ncc_window[0] // 2)
                    b = row + (self.ncc_window[0] // 2) + 1
                    c = column - (self.ncc_window[0] // 2)
                    d = column + (self.ncc_window[0] // 2) + 1
                            if 0 < a < rows - self.ncc_window[0] and 0 < c < columns -
self.ncc_window[0]:
                        template = np.array(self.array_of_images[0][a:b, c:d])
                        max_ncc = 0
                        max_row = None
                        max_column = None
                        for row2 in range(rows):
                            for column2 in range(columns):
                                if corners_1[row2][column2] > 0:
                                    a2 = row2 - (self.ncc_window[0] // 2)
                                    b2 = row2 + (self.ncc_window[0] // 2) + 1
                                    c2 = column2 - (self.ncc_window[0] // 2)
                                    d2 = column2 + (self.ncc_window[0] // 2) + 1
```

```
                                    if 0 < a2 < rows - self.ncc_window[0] and 0 < c2 < columns
- self.ncc_window[0]:
                                        match_to = np.array(self.array_of_images[1][a2:b2,
c2:d2])
                                        f = (match_to - match_to.mean())/(match_to.std() *
np.sqrt(match_to.size))
                                        g = (template - template.mean())/(template.std() *
np.sqrt(template.size))
                                    product = f * g
                                    stds = np.sum(product)
                                    if stds > max_ncc:
                                        max_ncc = stds
                                        max_row = row2
                                        max_column = column2
                        if max_row is not None:
                            if max_ncc > 0.90:
                                correspondences_picture0.append([row, column])
                                correspondences_picture1.append([max_row, max_column])
        return correspondences_picture0, correspondences_picture1

    def feature_matching(self, img_1, img_r, matching_type: str):
        corres_0, corres_1 = self.find_correspondences()
        img_1 = np.copy(img_1)
        img_r = np.copy(img_r)
        shift = np.shape(img_r)[1]
        full_img = np.concatenate((img_1, img_r), axis=1)
        if matching_type == 'before ransac':
            for i in range(len(corres_0)):
                color = list(np.random.random_sample(size=3) * 256)
                cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] + shift,
corres_1[i][0]), color,
                         thickness=1)

        elif matching_type == 'after ransac':
            inliers_0, inliers_1 = self.ransac()[1], self.ransac()[2]
            for i in range(len(corres_0)):
                if corres_0[i] in inliers_0:
                    cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] +
shift, corres_1[i][0]),
                             (0, 255, 0), thickness=1)
                else:
                    cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] +
shift, corres_1[i][0]),
                             (0, 0, 255), thickness=1)
        return full_img

    @staticmethod
    def find_homography(pts_src, pts_dst):
        pts_src = np.array(pts_src)
        pts_dst = np.array(pts_dst)

        # forming A matrix
        a_matrix = np.zeros((2 * len(pts_src), 9))
        for i in range(len(pts_src)):
            a_matrix[2 * i, 0] = pts_src[i][0]
            a_matrix[2 * i, 1] = pts_src[i][1]
            a_matrix[2 * i, 2] = 1
            a_matrix[2 * i, 6] = -pts_src[i][0] * pts_dst[i][0]
            a_matrix[2 * i, 7] = -pts_src[i][1] * pts_dst[i][0]
            a_matrix[2 * i, 8] = -pts_dst[i][0]

            a_matrix[2 * i + 1, 3] = pts_src[i][0]
            a_matrix[2 * i + 1, 4] = pts_src[i][1]
```

```
            a_matrix[2 * i + 1, 5] = 1
            a_matrix[2 * i + 1, 6] = -pts_src[i][0] * pts_dst[i][1]
            a_matrix[2 * i + 1, 7] = -pts_src[i][1] * pts_dst[i][1]
            a_matrix[2 * i + 1, 8] = -pts_dst[i][1]

        # Transposing A
        a_transpose = np.transpose(a_matrix)
        final_a_matrix = np.dot(a_transpose, a_matrix)
        u, d, u_transpose = np.linalg.svd(final_a_matrix, full_matrices=True)

         # h is the column of U associated with smallest eigen in D, which is the last value in
D
        h = (u[:, 8])
        h = np.reshape(h, (3, 3))
        return h

    def ransac(self):
        corres_0, corres_1 = self.find_correspondences()
        max_inliers = 0
        homography = None
        max_inliers_0 = []
        max_inliers_1 = []
        for it in range(self.iterations):
            random_sampled_indx = []
            pts_src = []
            pts_dst = []
            i = 0
            while i < 4:
                    random_sample = int(np.random.randint(0, high=len(corres_0) - 1, size=1,
dtype=int))
                if random_sample not in random_sampled_indx:
                    pts_src.append(corres_0[random_sample])
                    pts_dst.append(corres_1[random_sample])
                    random_sampled_indx.append(random_sample)
                    i += 1
            h = self.find_homography(pts_src, pts_dst)
            j = 0
            inliers = 0
            inliers_0 = []
            inliers_1 = []
            while j < len(corres_0):
                src = np.transpose(np.array([[corres_0[j][0], corres_0[j][1], 1]]))
                est = np.matmul(h, src)
                if est[2][0] == 0:
                    j += 1
                    continue
                est = est / est[2][0]
                                                dist    =    np.sqrt(np.power(est[0]-corres_1[j][0],
2)+np.power(est[1]-corres_1[j][1], 2))
                if dist < 5:
                    inliers += 1
                    inliers_0.append(corres_0[j])
                    inliers_1.append(corres_1[j])
                j += 1

            if inliers > max_inliers:
                max_inliers = inliers
                max_inliers_0 = inliers_0
                max_inliers_1 = inliers_1
                if max_inliers > 3:
                    homography = self.find_homography(max_inliers_0, max_inliers_1)
        return homography, max_inliers_0, max_inliers_1
```

```
    def disp_img(self, img1, img2):
        nms_harris = self.non_max_supression()
        rgb_imgs_arr = [img1, img2]
        corner_imgs = []
        for k in range(len(nms_harris)):
            rgb = np.copy(rgb_imgs_arr[k])
            r, c = np.shape(rgb)[0], np.shape(rgb)[1]
            for i in range(r):
                for j in range(c):
                    if nms_harris[k][i, j] > 0:
                        rgb[i, j] = [0, 0, 255]
            corner_imgs.append(rgb)
        return corner_imgs

    def image_stitching(self, img_1, img_2):

        h = np.linalg.inv(self.ransac()[0])
        y, x = img_2.shape[:2]
        p_1 = [[0], [0], [1]]
        p_2 = [[0], [x - 1], [1]]
        p_3 = [[y - 1], [0], [1]]
        p_4 = [[y - 1], [x - 1], [1]]

        p_1 = np.matmul(h, p_1)
        p_2 = np.matmul(h, p_2)
        p_3 = np.matmul(h, p_3)
        p_4 = np.matmul(h, p_4)

        p_1 = p_1 / p_1[2][0]
        p_2 = p_2 / p_2[2][0]
        p_3 = p_3 / p_3[2][0]
        p_4 = p_4 / p_4[2][0]

        y_min = round(min(p_1[0], p_2[0], p_3[0], p_4[0])[0])
        x_min = round(min(p_1[1], p_2[1], p_3[1], p_4[1])[0])
        y_max = round(max(p_1[0], p_2[0], p_3[0], p_4[0])[0])
        x_max = round(max(p_1[1], p_2[1], p_3[1], p_4[1])[0])

        pano = np.zeros((y_max - y_min + 10, x_max - x_min + round(np.shape(img_1)[1] * 0.3),
3), dtype=np.uint8)
        pano[abs(y_min):np.shape(img_1)[0] + abs(y_min), :np.shape(img_1)[1], :] = img_1

        for row in range(pano.shape[0]):
            for column in range(pano.shape[1]):
                est = np.matmul(np.linalg.inv(h), np.array([[row], [column], [1]]))
                est = est / est[2][0]
                est_y = round(est[0][0])
                est_x = round(est[1][0])
                if 0 <= est_x < img_2.shape[1] and 0 <= est_y + y_min < img_2.shape[0]:
                    if pano[row][column][:].any() > 0:
                        w_1 = min(row, column, img_1.shape[0] - row - y_min, img_1.shape[1] -
column)
                        w_2 = min(est_y + y_min, est_x, img_2.shape[0] - (est_y + y_min),
                                  img_2.shape[1] - est_x)
                        if w_1 + w_2 != 0:
                            feather_1 = np.array(pano[row][column][:]).astype(np.uint64) * w_1
/ (w_1 + w_2)
                            feather_2 = np.array(img_2[est_y + y_min][est_x][:]).astype(np.uint64) * w_2 / (w_1 + w_2)
                            pano[row][column][:] = (feather_1 + feather_2).astype(np.uint8)

                        else:
                            pano[row][column][:] = 0.5 * np.array(pano[row][column][:]) \
```

```
                                                    + 0.5 * np.array(img_2[abs(est_y +
y_min)][est_x][:])
                    else:
                        pano[row][column][:] = img_2[abs(est_y + y_min)][est_x][:]

        return pano


if __name__ == "__main__":
    path_to_images = r"D:\NEU\3 Sem\Computer Vision\Project 2\DanaHallWay1"
    imgs_arr = []
    imgs_arr2 = []
    for img_name in os.listdir(path_to_images):
        img = cv2.imread(path_to_images + '\\' + img_name, 0)  # Read and convert the image to
grayscale
        imgs_arr.append(np.asarray(img).astype(float))  # Create an array of all the images
    for img_name2 in os.listdir(path_to_images):
        img = cv2.imread(path_to_images + '\\' + img_name2)  # Read the image
        imgs_arr2.append(np.asarray(img))
    image_to_analyze = [0, 1]
    rgb_1 = imgs_arr2[image_to_analyze[0]]
    rgb_2 = imgs_arr2[image_to_analyze[1]]
    g_1 = imgs_arr[image_to_analyze[0]]
    g_2 = imgs_arr[image_to_analyze[1]]
    inst = ImageMosaicing([g_1, g_2], "sobel", (5, 5), 100000, (7, 7), 1000)
    canvas = inst.image_stitching(rgb_1, rgb_2)
    cv2.imshow('Mosaic', canvas)
    cv2.waitKey(0)
```