Hieu Nguyen
GTID: 903185448
Email: hieu@gatech.edu
10/19/15

CS-6475 Computational Photography Assignment 8

For this assignment, I experimented with rotational and planar panorama. The input images were taken from the "backyard" of my apartment complex in Austin, TX. First, I tried my hand at rotational panorama by standing in one place and rotating the camera and making sure parts of the images would overlap. Then I took some sample images by keeping the camera angle stationary, but moving along a straight line (in the same plane).

The input images were scaled down to improve execution time. Once the files are placed in the right directory, the "assignment8_test.py" file is built to generate a panorama_1.jpg output image. This process utilizes the functions I wrote to find feature matches between the images, warp the images to align them together, and then blend them. These panoramas were generated with a number of matches equal to 20, as in the test code. I found that by increasing the number of matched features, the output is warped differently because there are more features to affect the warping and translation. There seems to be a sweet spot to making the first image the least warped (~50 for the example images), but this would change based on the input images and camera rotation. Also, the execution times are longer with additional matches because there is more data to sift through.

Overall, I am happy with the result as the sample images turned out nicely. My own panoramas, however, did not perform so well. I could have spent more time making sure that my own input files were of good quality to create a panorama. The planar panorama attempt failed horribly; I believe this is due to occlusions in features taken at different angles. I could probably fix this by taking images of a relatively "flat" subject with lots of features to match with.

I also wish I had more time to improve the blending and execution time of the program. In the future, I'd like to combine my Laplacian pyramid blending from assignment 6 to make the blended images look more seamless. I also think it'd be cool to experiment with the Brenizer method of panorama to create stunning portraits.

Rotational Panorama Input (left to right):

Rotational Panorama Result:

Cropped Rotational Panorama Result:

Planar Panorama Input (left to right):

Planar Panorama Result (fail):

Part 1: Programming the core of panoramas

1. getImageCorners()
   This function returns the 4 corners of an input image. The output is a numpy.ndarray of shape (4, 1, 2), which contains the corners in (X, Y) format.

```
def getImageCorners(image):
    corners = np.zeros((4, 1, 2), dtype=np.float32)
    corners[1] = [0, image.shape[0]]
    corners[2] = [image.shape[1], 0]
    corners[3] = [image.shape[1], image.shape[0]]
    return corners
```

2. findMatchesBetweenImages()
   This function returns the top list of matches between two input images. It takes in two grayscale images and a desired number of matches as arguments, then outputs the corresponding image keypoints and a list of the matches. For this function, I just recycled my code from Assignment 7 on Feature Detection. It essentially uses ORB to get keypoints and descriptors, then performs a brute force technique to find matches.

```
def findMatchesBetweenImages(image_1, image_2, num_matches):
    orb = cv2.ORB()
    image_1_kp, image_1_desc = orb.detectAndCompute(image_1, None)
    image_2_kp, image_2_desc = orb.detectAndCompute(image_2, None)
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(image_1_desc, image_2_desc)
    matches = sorted(matches, key=lambda x:x.distance)
    return image_1_kp, image_2_kp, matches
```

3. findHomography()
   This function returns the homography between the keypoints of image 1, image 2, and its matches. I iterate through each match to find the (X, Y) location of the keypoint for each match. Here, image 1 is the query image and image 2 is the train image. The cv2.findHomography() function is then used to generate a 3x3 homography matrix. This function uses RANSAC to eliminate outlier matches.

```
def findHomography(image_1_kp, image_2_kp, matches):
    image_1_points = np.zeros((len(matches), 1, 2), dtype=np.float32)
    image_2_points = np.zeros((len(matches), 1, 2), dtype=np.float32)
    i = 0
    for mat in matches:
        image_1_points[i] = np.float32(image_1_kp[mat.queryIdx].pt)
        image_2_points[i] = np.float32(image_2_kp[mat.trainIdx].pt)
        i += 1
    M_hom, mask = cv2.findHomography(image_1_points, image_2_points, \
                    method=cv2.RANSAC, ransacReprojThreshold=5.0)
    return M_hom
```

4. warpImagePair()
   This function warps image 1 so it can be blended with image 2 (stitched). First, the corners of both images are found. Next, the corners of image 1 are transformed using the calculated homography matrix. The min and max values for X and Y are calculated based off the

transformed image 1 corners and the image 2 corners. A translation matrix is then created using (-1*x_min) and (-1*y_min). These values are multiplied by negative 1 because of properties of the matrix dot product. The combined translation and homography matrix needs to be inversed for the output warped image pair to be oriented properly. These negative values help to remove the offsets of the x and y minimums. The function cv2.warpPerspective() is then called using the dot product of the homography and the translation matrix. The resulting warped image and image 2 are then stitched/blended together by calling blendImagePair().

```
def warpImagePair(image_1, image_2, homography):
    image_1_corners = getImageCorners(image_1)
    image_2_corners = getImageCorners(image_2)
    image_1_corners_t = cv2.perspectiveTransform(image_1_corners, homography)
    join_corners = np.append(image_1_corners_t, image_2_corners, axis=0)
    x_min = np.amin(join_corners[:,:,0])
    x_max = np.amax(join_corners[:,:,0])
    y_min = np.amin(join_corners[:,:,1])
    y_max = np.amax(join_corners[:,:,1])
    trans_mat = np.array([[1, 0, -1 * x_min], [0, 1, -1 * y_min], [0, 0, 1]])
    trans_hom_mat = np.dot(trans_mat, homography)
    warped_image = cv2.warpPerspective(image_1, trans_hom_mat, (x_max-x_min,
y_max-y_min))
    output_image = blendImagePair(warped_image, image_2, (-1*x_min, -1*y_min))
    return output_image
```

Part 2: Get creative with programming and photography

1. blendImagePair()
   This function takes an image that has been warped and an image that needs to be inserted into the warped image, at a specific point in the output image. I would have liked to utilize my code from assignment 6 on Image Blending (Laplacian pyramids, etc.), but I was unsure if it was within the scope of this assignment. Instead, I implemented a function to average the pixels where the images overlap. I incorporated a weight function from the start of the overlap point to create a “fade” effect and properly transition from image 1 to image 2. This fade effect was attempted both horizontally and vertically (within top and bottom thresholds).

   This blending function has a long execution time because it must iterate over every pixel in the output image and calculate the new pixel value, up to three times (~8 minutes to generate the sample panorama). These execution times would extend for larger amounts of high resolution input images. Also, it may not work well with an input warped image that contains a lot of black pixels, as a crude workaround was used to generate its mask.

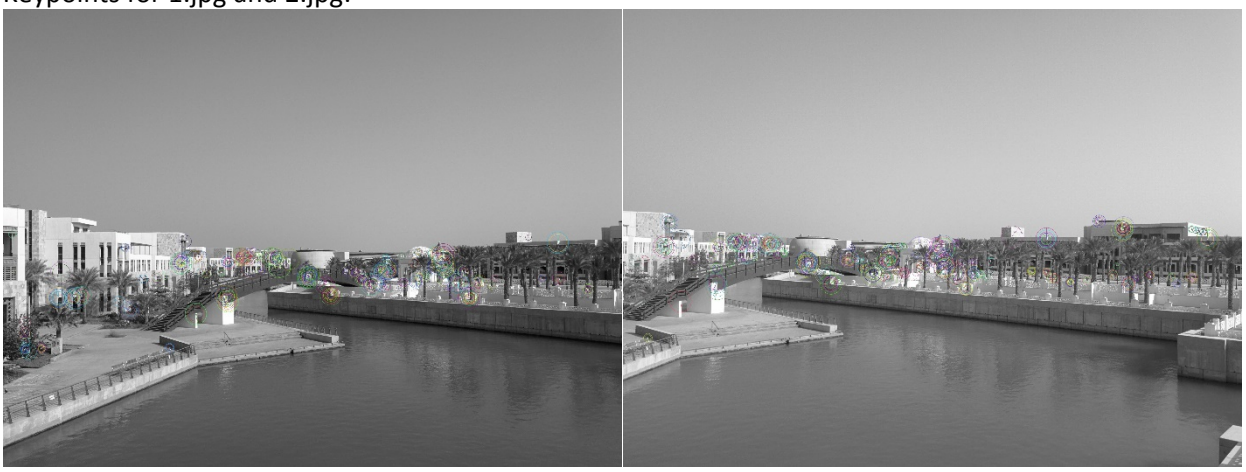
```
def blendImagePair(warped_image, image_2, point):
    output_image = np.copy(warped_image)
    output_image[point[1]:point[1] + image_2.shape[0],
                 point[0]:point[0] + image_2.shape[1]] = image_2

    # Create mask of overlapping region
    mask_1 = np.copy(warped_image)
    mask_2 = np.copy(warped_image)
    mask_1[mask_1 > 0] = 127
    mask_2[:,:,:] = 0
```

```
mask_overlap = np.copy(mask_2)
mask_2[point[1]:point[1]+image_2.shape[0],
       point[0]:point[0]+image_2.shape[1]] = 127
mask_overlap = mask_1 + mask_2
mask_overlap[mask_overlap > 127] = 255
mask_overlap[mask_overlap < 128] = 0

# Find corners of overlap mask
max_col = 0
min_col = 100000
max_row = 0
min_row = 100000
for row in xrange(mask_overlap.shape[0]):
    for col in xrange(mask_overlap.shape[1]):
        if(col>max_col):
            if(mask_overlap[row,col,0]==255):
                max_col = col
        if(col<min_col):
            if(mask_overlap[row,col,0]==255):
                min_col = col
        if(row>max_row):
            if(mask_overlap[row,col,0]==255):
                max_row = row
        if(row<min_row):
            if(mask_overlap[row,col,0]==255):
                min_row = row

# Iterate over output image and replace overlap with weighted average
row_top_threshold = 100
row_bot_threshold = 150
for row in xrange(output_image.shape[0]):
    for col in xrange(output_image.shape[1]):
        if (mask_overlap[row, col, 0] > 0):
            x_weight = (float(col)-min_col)/(max_col-min_col)
            output_image[row, col] = x_weight*output_image[row, col] +
                (1-x_weight)*warped_image[row, col]
            if (row>=min_row and row<=min_row+row_top_threshold):
                y_weight_top = (float(row)-min_row)/row_top_threshold
                output_image[row, col] = y_weight_top*
                      output_image[row, col] + (1-
                      y_weight_top)*warped_image[row, col]
            if (row<=max_row and row>=max_row-row_bot_threshold):
                y_weight_bot = 1.0 - ((max_row-float(row))/
                             row_bot_threshold)
                output_image[row, col] = y_weight_bot*image_2[row-point[1],
                col-point[0]] + (1-y_weight_bot)*output_image[row, col]

return output_image
```

Rotational Panorama tests with example images:

Input Images:

Keypoints for 1.jpg and 2.jpg:



Warped 1.jpg to be merged with 2.jpg:

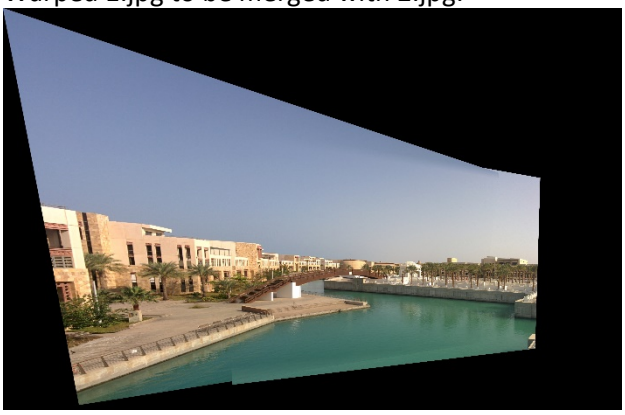
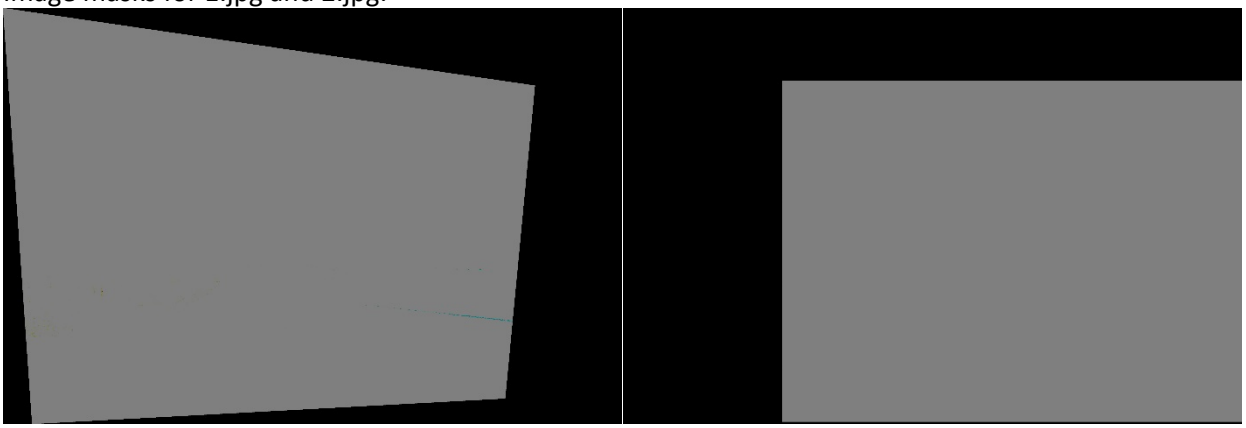
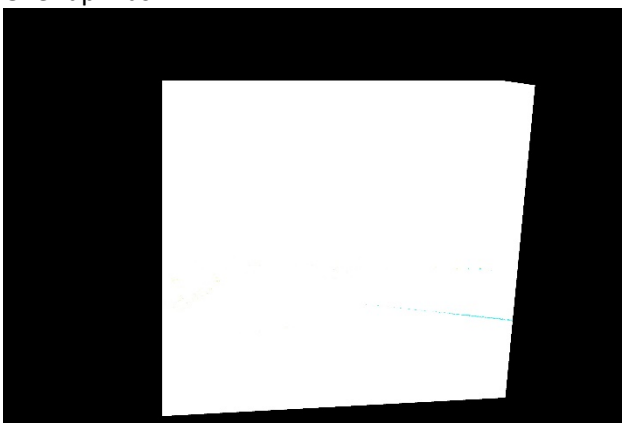


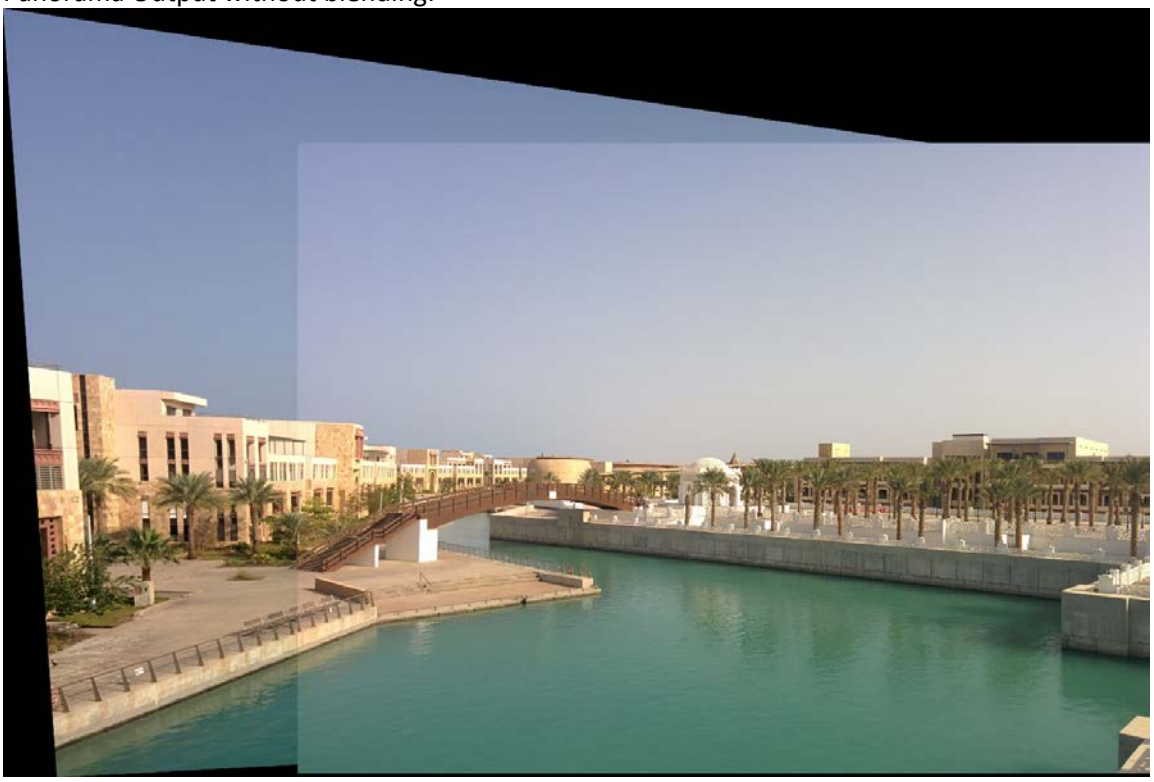
Image masks for 1.jpg and 2.jpg:



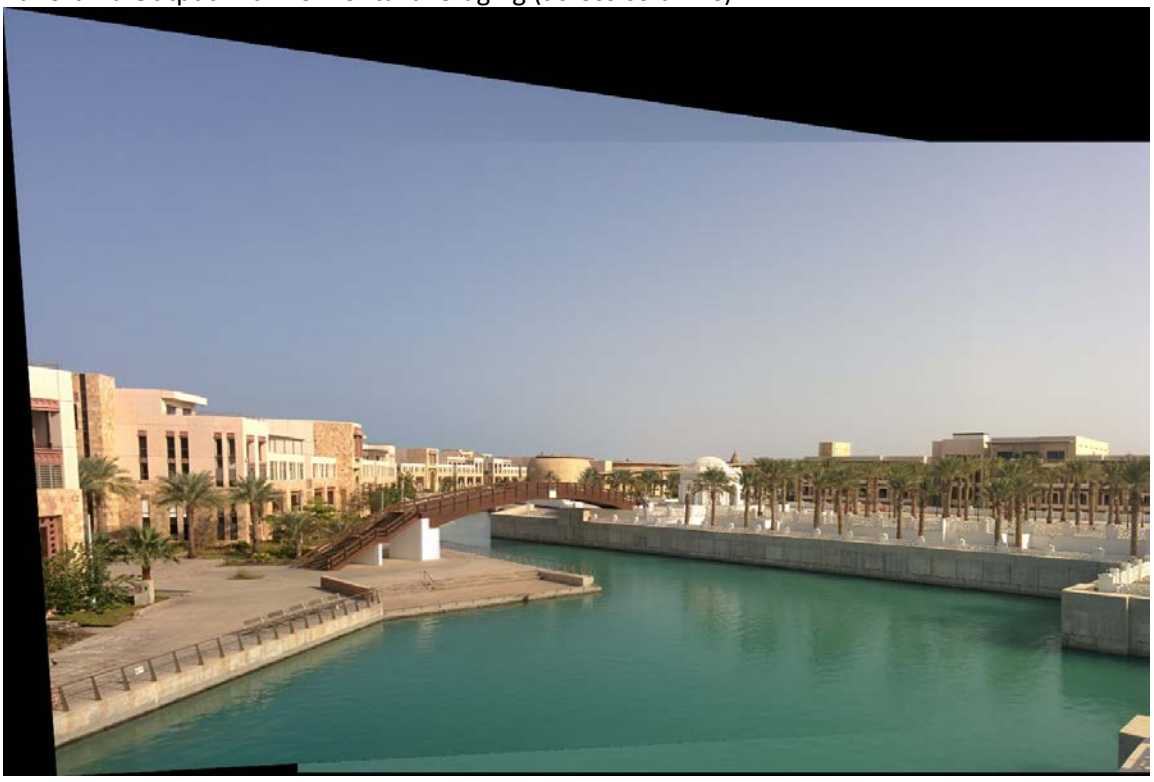
Overlap mask:



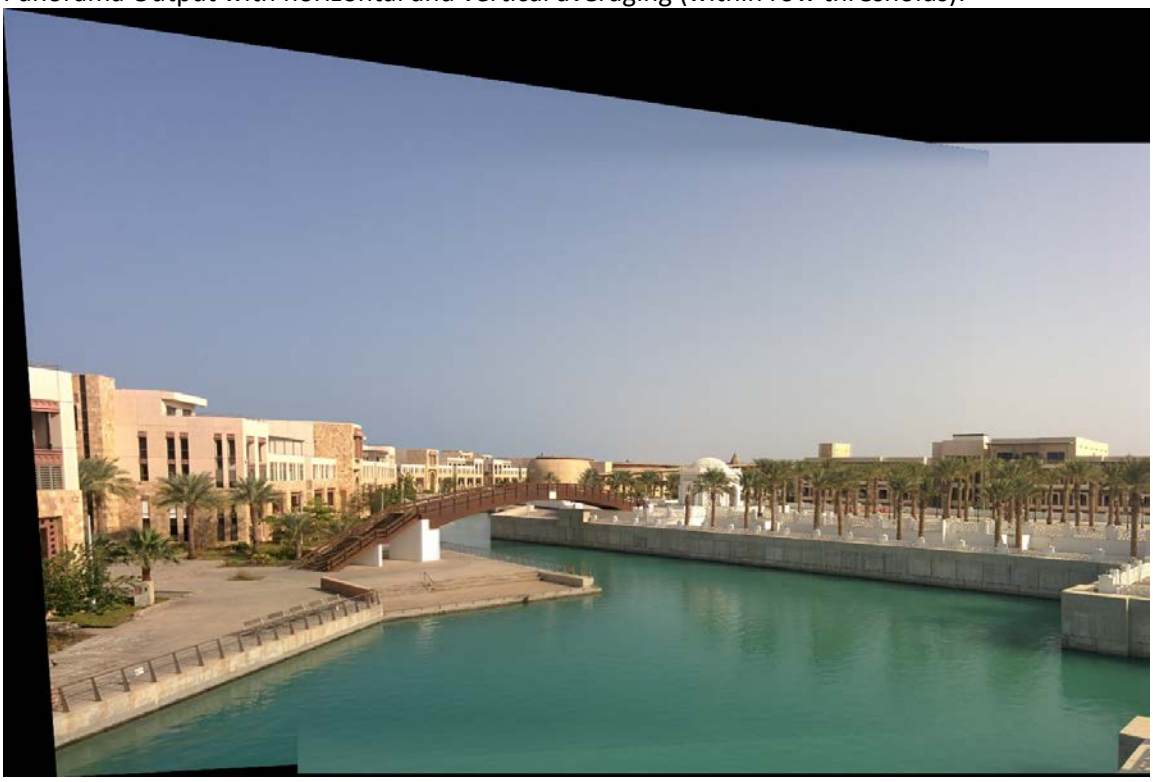
Panorama Output without blending:



Panorama Output with horizontal averaging (across columns):

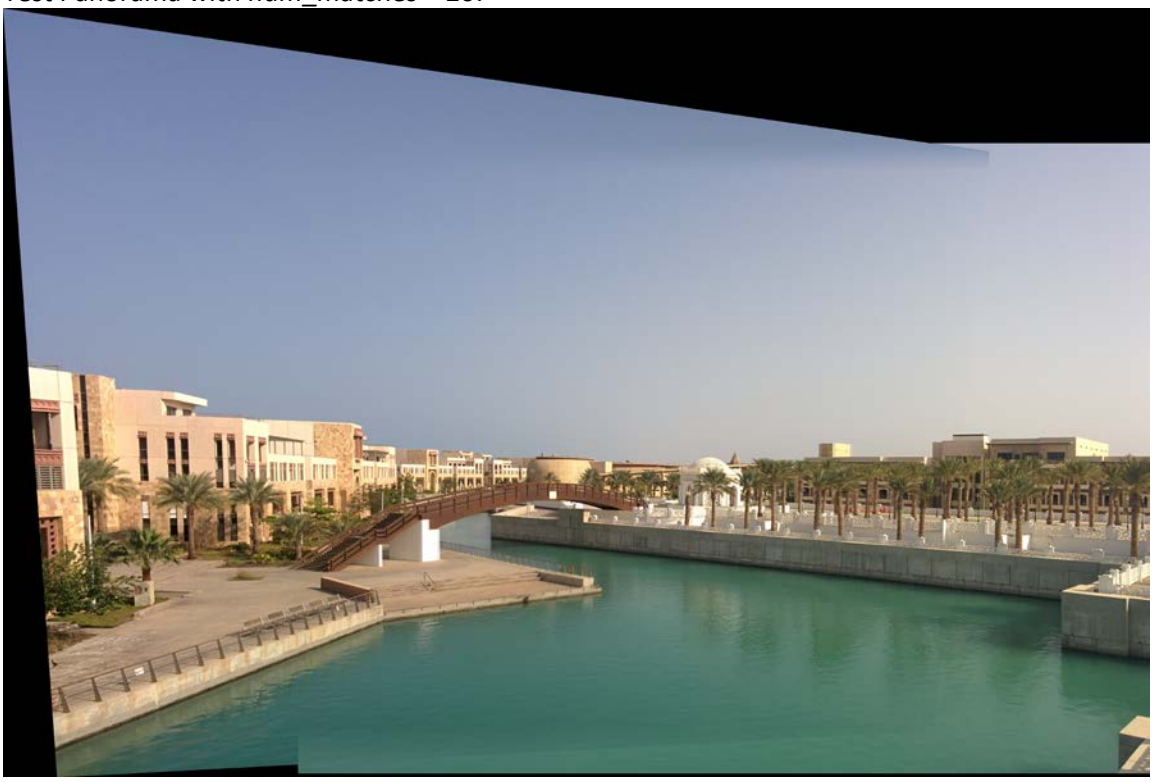


Panorama Output with horizontal and vertical averaging (within row thresholds):

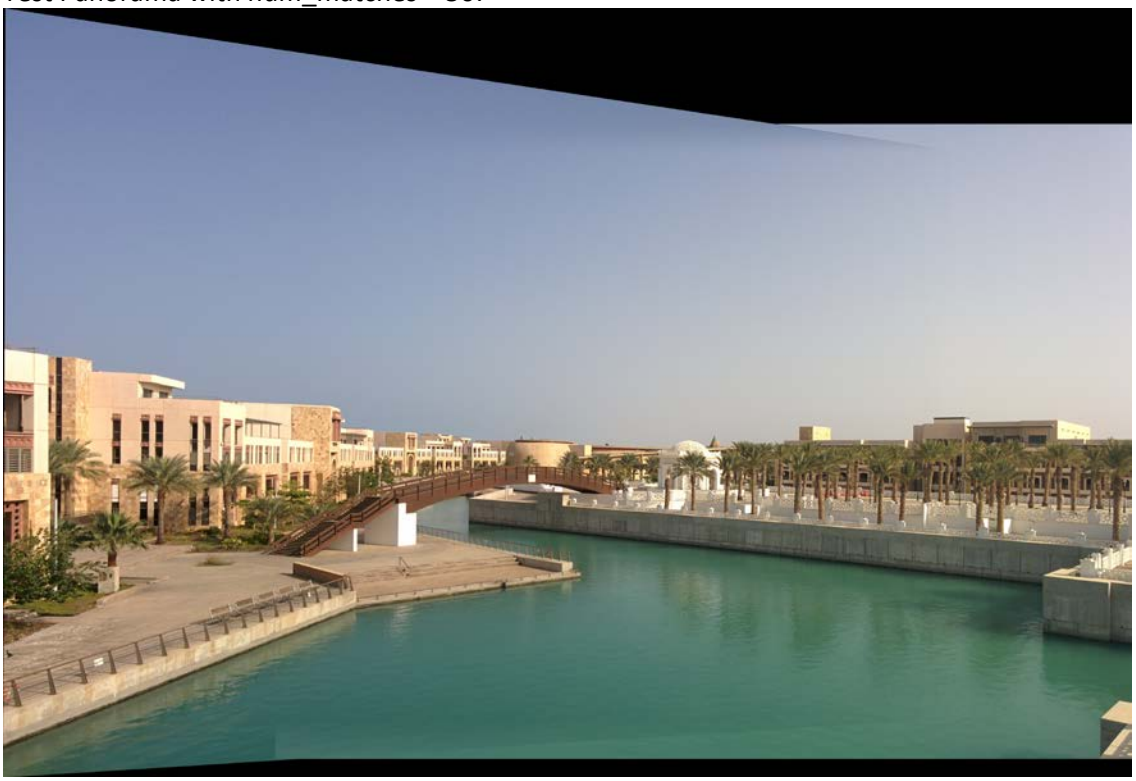


Cropped Panorama Output:

Test Panorama with num_matches = 20:



Test Panorama with num_matches = 50:



Test Panorama with num_matches = 100: