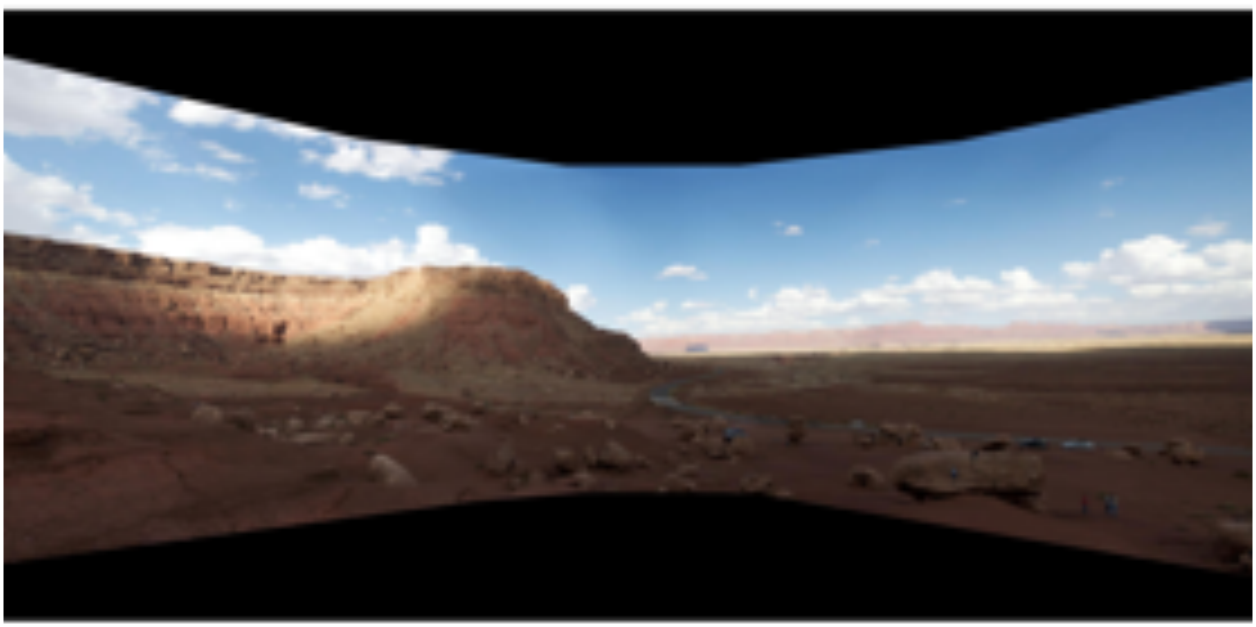
Mohammed Kamran Syed
50289322
msyed3@buffalo.edu
April 7, 2019

CSE573: Project 2 - Image Stitching



The goal of this task was to experiment with image stitching methods. Given a set of photos, the program should be able to stitch them into a panoramic photo.

Algorithm for Image Stitching:

- Extract key points and descriptors from both the images.
- Find matches between the two sets of key points based on hamming distance of their descriptors.
- Using RANSAC algorithm eliminate all the outliers from the set of matching points and generate the best homography matrix.
- Finally warp one image to another using the tomography matrix obtained and stitch them.

Program Steps:

- The main function checks and adds all the image files from the specified source directory to the image queue.

```
def main():
    # Check if source directory is specified while running the script
    if len(sys.argv) < 2:
        raise ValueError("Insufficient number of arguments")

    img_dir = sys.argv[1]
    image_queue = list()

    # Only add images from source directory to the queue
    for image in os.listdir(os.path.abspath(img_dir)):
        if (image.lower().endswith('jpg') or image.lower().endswith('png') or image.lower().endswith('jpeg')) and image.lower(
            image_queue.append(os.path.join(os.path.abspath(img_dir), image))
```

- Two random images are picked up from the image queue and checked for a possible overlap (for images with overlap, minimum hamming distance between the descriptor bits should be less than 10).
- If they have, then use these images to proceed further else pick the next two.

```
    # Select two images that can be stitched
    for i in range(2):
        image1 = image_queue[i]
        image2 = image_queue[i+1]
        img1, gray1 = read_image(image1, img_dir)
        img2, gray2 = read_image(image2, img_dir)

        result_image, result_gray = stitcher(img1, img2, gray1, gray2)

        if result_image is not None:
            image_queue.remove(image1)
            image_queue.remove(image2)
            break
```

- Using ORB, key points and descriptors of images are extracted.

```
def get_keypoint_descriptors(img):
    ''' Uses ORB to extract keypoints and descriptors from an image '''
    return orb.detectAndCompute(img, None)
```

- Using find_matches function, we create a set of matching pairs from both the images using hamming distance calculated over bits of descriptors.

```
def find_matches(keypoints1, descriptors1, keypoints2, descriptors2):
    ''' Used to find matches between the descriptors from image1 to image 2 using hamming distance on bits '''
    results = list()
    descriptor_size = len(descriptors1[0])
    overall_min = sys.maxsize

    for i in range(len(descriptors1)):
        min_diff = sys.maxsize
        min_right_index = -1
        for j in range(len(descriptors2)):
            total_diff_bits = 0
            for k in range(descriptor_size):
                x = int('{0:08b}'.format(descriptors1[i][k]))
                y = int('{0:08b}'.format(descriptors2[j][k]))
                total_diff_bits += np.count_nonzero(x != y)

            if total_diff_bits < min_diff:
                min_diff = total_diff_bits
                min_right_index = j

                if min_diff < overall_min:
                    overall_min = min_diff
        results.append([list(keypoints1[i].pt), list(keypoints2[min_right_index].pt)])

    if overall_min > 10:
        return None

    return results
```

- Once we have the matches, ransac function is used to eliminate all the outliers from the matches set and compute the homography matrix.

```
def ransac(source_points, destination_points, inlier_threshold):
    ''' Used to calculate the best homography matrix based on 4 randomnly chosen points and counting inliers '''
    max_inliers = - sys.maxsize - 1
    best_h = None

    for _ in range(500):
        random_pts = np.random.choice(range(0, len(source_points) - 1), 4, replace=False)

        src = []
        dst = []
        for i in random_pts:
            src.append(source_points[i])
            dst.append(destination_points[i])

        h = find_homograph(src, dst)

        count = 0
        for index in range(len(source_points)):
            src_pt = np.append(source_points[index], 1)

            dest_pt = np.dot(h, src_pt.T)

            dest_pt = np.true_divide(dest_pt, dest_pt[2])[0: 2]

            if distance.euclidean(destination_points[index], dest_pt) <= inlier_threshold:
                count += 1

        if count > max_inliers:
            max_inliers = count
            best_h = h

    return best_h, max_inliers
```

- find_homograph and create_P_matrix are the helper functions to create the homography matrix.

```python
def create_P_matrix(source_points, destination_points):
    ''' Helper matrix for creating homography matrix '''
    sub_matrices = []
    for i in range(len(source_points)):
        sub_matrix = np.zeros((2,9))
        sub_matrix[0][0] = -1 * source_points[i][0]
        sub_matrix[0][1] = -1 * source_points[i][1]
        sub_matrix[0][2] = -1
        sub_matrix[0][6] = source_points[i][0] * destination_points[i][0]
        sub_matrix[0][7] = source_points[i][1] * destination_points[i][0]
        sub_matrix[0][8] = destination_points[i][0]

        sub_matrix[1][3] = -1 * source_points[i][0]
        sub_matrix[1][4] = -1 * source_points[i][1]
        sub_matrix[1][5] = -1
        sub_matrix[1][6] = source_points[i][0] * destination_points[i][1]
        sub_matrix[1][7] = source_points[i][1] * destination_points[i][1]
        sub_matrix[1][8] = destination_points[i][1]
        sub_matrices.append(sub_matrix)

def stitcher(img1, img2, gray1, gray2):
    ''' Used to stitch two images if they are stitchable '''
    keypoints1, descriptors1 = get_keypoint_descriptors(gray1)
    keypoints2, descriptors2 = get_keypoint_descriptors(gray2)
    matches = find_matches(keypoints1, descriptors1, keypoints2, descriptors2)

    if matches is None:
        return None, None

    src = np.float32([match[0] for match in matches]).reshape(-1,2)
    dst = np.float32([match[1] for match in matches]).reshape(-1,2)

    homograph = None
    i = 0
    threshold = min(len(descriptors1), len(descriptors2))

    while True:
        homograph, max_inliers = ransac(src, dst, i)

        if 0.1 * threshold <= max_inliers < 0.3 * threshold:
            break
        elif max_inliers < 0.1 * threshold:
            i += 1
        else: i -= 1

    return warp_images(img2, img1, homograph), warp_images(gray2, gray1, homograph)
```

- The process of running the RANSAC algorithm is done until we get the inliers in the range of 10% to 30% the size of descriptors to avoid under fitting and over fitting of the images.
- This is done by increasing the inlier threshold distance by 1 starting from 1.

```
def stitcher(img1, img2, gray1, gray2):
    ''' Used to stitch two images if they are stitchable '''
    keypoints1, descriptors1 = get_keypoint_descriptors(gray1)
    keypoints2, descriptors2 = get_keypoint_descriptors(gray2)
    matches = find_matches(keypoints1, descriptors1, keypoints2, descriptors2)

    if matches is None:
        return None, None

    src = np.float32([match[0] for match in matches]).reshape(-1,2)
    dst = np.float32([match[1] for match in matches]).reshape(-1,2)

    homograph = None
    i = 0
    threshold = min(len(descriptors1), len(descriptors2))

    while True:
        homograph, max_inliers = ransac(src, dst, i)

        if 0.1 * threshold <= max_inliers < 0.3 * threshold:
            break
        elif max_inliers < 0.1 * threshold:
            i += 1
        else: i -= 1

    return warp_images(img2, img1, homograph), warp_images(gray2, gray1, homograph)
```

- Once we have the correct set of inliers and homography matrix, warp_images is called to warp the second image over the first one.

```
def warp_images(img1, img2, H):
    ''' Warps image 2 and stitches it to image 1 '''
    h1, w1 = img1.shape[ :2]
    h2, w2 = img2.shape[ :2]
    pts1 = np.float32([[0, 0], [0, h1], [w1, h1], [w1, 0]]).reshape(-1,1,2)
    pts2 = np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1,1,2)
    pts2_ = cv2.perspectiveTransform(pts2, H)
    pts = np.concatenate((pts1, pts2_), axis=0)
    [xmin, ymin] = np.int32(pts.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(pts.max(axis=0).ravel() + 0.5)
    t = [-xmin, -ymin]
    Ht = np.array([[1, 0, t[0]],[0, 1, t[1]],[0, 0, 1]])
    result = cv2.warpPerspective(img2, Ht.dot(H), (xmax-xmin, ymax-ymin))
    result[t[1]:h1+t[1],t[0]:w1+t[0]] = img1
    return result
```

- To stitch a third image, we use the stitched two images and repeat the process with the third image.

```
    # If there are more than two images then stitch this third image to the panorama obtained earlier
    if len(image_queue) > 0:
        image = image_queue[-1]
        img, gray = read_image(image, img_dir)

        result_image, result_gray = stitcher(result_image, img, result_gray, gray)

    # Finally write the panorama to disk
    write_output_image(result_image, img_dir, 'panorama')
```

Results:

The following were the results of the stitcher script when ran on two sets of test images placed in “data” and “ubdata” directories.

- Nevada Images:

The above image is the result of stitching the three Nevada images into panorama style.

- UB Images:

The below image is the panorama stitching performed on images from the UB Capen library. This is the place I value the most at UB and spend most of my time learning and doing projects.