CS543/ECE549 Assignment 3

Name: Derek Yang
NetId: mcy3

Part 1: Homography estimation

A: Describe your solution, including any interesting parameters or implementation choices for feature extraction, putative matching, RANSAC, etc.

For feature extraction I used `cv2.SIFT_create().detectAndCompute` returning the key point and descriptor for the two images.

```
kp1,dsp1 = cv2.SIFT_create().detectAndCompute(img1_g,None)
kp2,dsp2 = cv2.SIFT_create().detectAndCompute(img2_g,None)
```

For distance calculation, I used `scipy.spatial.distance.cdist(X,Y,'sqeuclidean')` to calculate distance between every other descriptor.

```
dist = scipy.spatial.distance.cdist(dsp1, dsp2, 'sqeuclidean')
```

After testing different threshold, I picked the threshold of 9000 and keep anything that has a lower distance than it.

```
idx1 = np.where(dist < threshold)[0]
idx2 = np.where(dist < threshold)[1]
```

For the steps in RANSAC, I first compute the homography with SVD and use V[len(V)-1] for the smallest singular value.

```
def compute_homo(subset):
    A = []

    for i in range(subset.shape[0]):

        p1 = np.append(subset[i][0:2], 1)
        p2 = np.append(subset[i][2:4], 1)

        row1 = [0, 0, 0, p1[0], p1[1], p1[2], -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]*p1[2]]
        row2 = [p1[0], p1[1], p1[2], 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]*p1[2]]

        A.append(row1)
        A.append(row2)

    U, s, V = np.linalg.svd(np.array(A))

    H = V[len(V)-1].reshape(3, 3)

    H = H / H[2, 2]

    return H
```

I applied the homography in *compute_error* which returns the squared difference of the transformed point.

```
def compute_error(H,matches):
    num_pairs = len(matches)

    p1 = np.concatenate((matches[:,0:2],np.ones((1,num_pairs)).T),axis = 1)
    p2 = matches[:,2:4]

    t_p1 = np.zeros((num_pairs,2))

    for i in range(num_pairs):
        t_p1[i] = (np.matmul(H, p1[i]) / np.matmul(H, p1[i])[-1])[0:2]

    return np.linalg.norm(p2 - t_p1, axis=1) ** 2
```

For RANSAC algorithm, I did 1000 iterations with the randomly sampled 4 points

```
for i in range(1000):
    idx = rd.sample(range(matches.shape[0]),4)
    subset = matches[idx]
```

If the inlier surpasses the current best ones, the best inlier would be replaced by the current ones, the homography and matches would be stored and the average residual would be calculated.

```
error = compute_error(H,matches)
mask = np.where(error < threshold)[0]
inliers_pt  = matches[mask]

inliers = len(inliers_pt)
if inliers > best_inliers:
    best_inliers = inliers
    best_H = H.copy()
    best_inliers_pt = inliers_pt.copy()

    avg_residual = sum(compute_error(best_H,best_inliers_pt)) / best_inliers
```

After all the iteration the best one would be printed and returned

```
print("Inliers:",best_inliers,"Average residuals:",avg_residual)

fig,ax = plt.subplots(figsize=(20,10))
#plot_inlier_matches(ax, img1, img2, best_inliers_pt)
#plt.savefig("plot_basic.jpg")
```

For the image stitching process I used `skimage.transform.ProjectiveTransform` and `skimage.transform.warp`.

I determine the output image shape by finding the maximum difference after warping with the size of the original image.

```
c_temp = np.array([[0, 0],
                   [0, h],
                   [w, 0],
                   [w, h]])

c_w = transform(c_temp)
corner = np.vstack((c_w, c_temp))

c_min = np.min(corner, axis=0)
c_max = np.max(corner, axis=0)

output_shape = (c_max - c_min)
output_shape = np.ceil(output_shape[::-1])
```

And warp the image to the bigger image with offsets

```
img1_w = warp(img1, (transform + offset).inverse, output_shape=output_shape, cval=0)
img2_w = warp(img2, offset.inverse, output_shape=output_shape, cval=0)
```

For the overlapping part I used the pixel value of one of the images.

```
img2_w[img1_w > 0] = 0
result = img1_w + img2_w
result = cv2.convertScaleAbs(result, alpha=(255.0))
return result
```

B: For the image pair provided, report the number of homography inliers and the average residual for the inliers. Also, display the locations of inlier matches in both images.

```
KeyboardInterrupt
^C
C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageSticking_shape_from_shading>python Yang_Derek_a3_p1.py
Inliers: 18 Average residuals: 0.445717759778923

C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageSticking_shape_from_shading>
```

C: Display the final result of your stitching.

Part 2: Shape from shading

A: Estimate the albedo and surface normals

1) Insert the albedo image of your test image here:

B01

B02

B05

B07

2) What implementation choices did you make? How did it affect the quality and speed of your solution?

I implemented two methods in *photometric stereo*:

(a) through all the pixel (“each”)

```
for x in range(h):
    for y in range(w):

        g = np.linalg.lstsq(light_dirs, imarray[x, y, :], rcond = -1)[0]

        albedo_image[x][y] = np.linalg.norm(g, axis=0)

        surface_normals[x, y, :] = g / albedo_image[x][y]
```

(b) reshape the image so that solver can get the full solution in one go (“single”)

```
    imarray = imarray.reshape(h*w, n).transpose()


    results = np.linalg.lstsq(light_dirs, imarray,rcond = -1)
    g = results[0]


    albedo_image = np.linalg.norm(g, axis=0)
    surface_normals = g / albedo_image


    surface_normals = surface_normals.transpose().reshape(h, w, 3)
    albedo_image = albedo_image.reshape(h, w)

    return albedo_image, surface_normals
```

The “single” method yields significantly better speed performance wise.

```
single : 0.02602553367614746
each : 0.8129920959472656
```

For get_surface(), row and column method is pretty straightforward, “average” is a an average of both the row and column method

```
h, w, n = surface_normals.shape

fx = surface_normals[:, :, 0] / surface_normals[:, :, 2]
fy = surface_normals[:, :, 1] / surface_normals[:, :, 2]

row_sum_x = np.cumsum(fx, axis=1)
col_sum_y = np.cumsum(fy, axis=0)

if integration_method == 'row':
    return row_sum_x[0] + col_sum_y
elif integration_method == 'column':
    return col_sum_y[:, 0][:, np.newaxis] + row_sum_x
elif integration_method == 'average':
    return (col_sum_y[:, 0][:, np.newaxis] + row_sum_x + row_sum_x[0] + col_sum_y) / 2
else:
    return random_path(surface_normals,25,fx,fy)
```

For the random path method, I used np.random.shuffle() to do a pseudo random shuffle of a stream of bits representing the movements of the path to the desired location and average the total derivative pixel values.

```
for _ in range(loops):
    zeros = [0] * x
    ones = [1] * y
    bit_stream = np.array(zeros + ones)

    np.random.shuffle(bit_stream)

    current_x = 0
    current_y = 0
    cumsum = 0

    for step in bit_stream:
        if step == 0:
            cumsum += fx[current_y, current_x]
            current_x += 1
        else:
            cumsum += fy[current_y, current_x]
            current_y += 1

    height_map[y, x] += cumsum

height_map[y, x] = height_map[y, x]/loops
```

I increase the row and col size to 100 for a better quality of 3D height map

```
surf = ax.plot_surface(
    H, X, Y, rcount=100, ccount=100, cmap='gray', facecolors=A, linewidth=0, antialiased=False)
set_aspect_equal_3d(ax)
```

3) What are some artifacts and/or limitations of your implementation, and what are possible reasons for them?

   There are some artifacts on the edges for some of the albedo map, for the height maps there are also some artifacts on the mouth part for B01, B02, B05.

4) Display the surface normal estimation images below:

B01

X Y Z

B02

X Y Z

B05

X Y Z

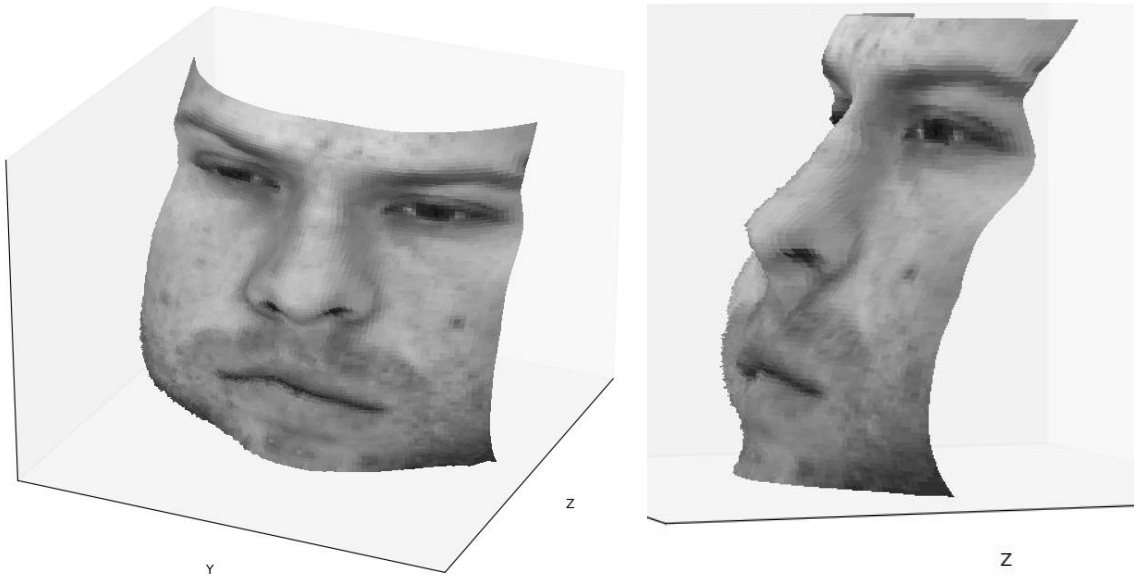
B07

X **Y** **Z**

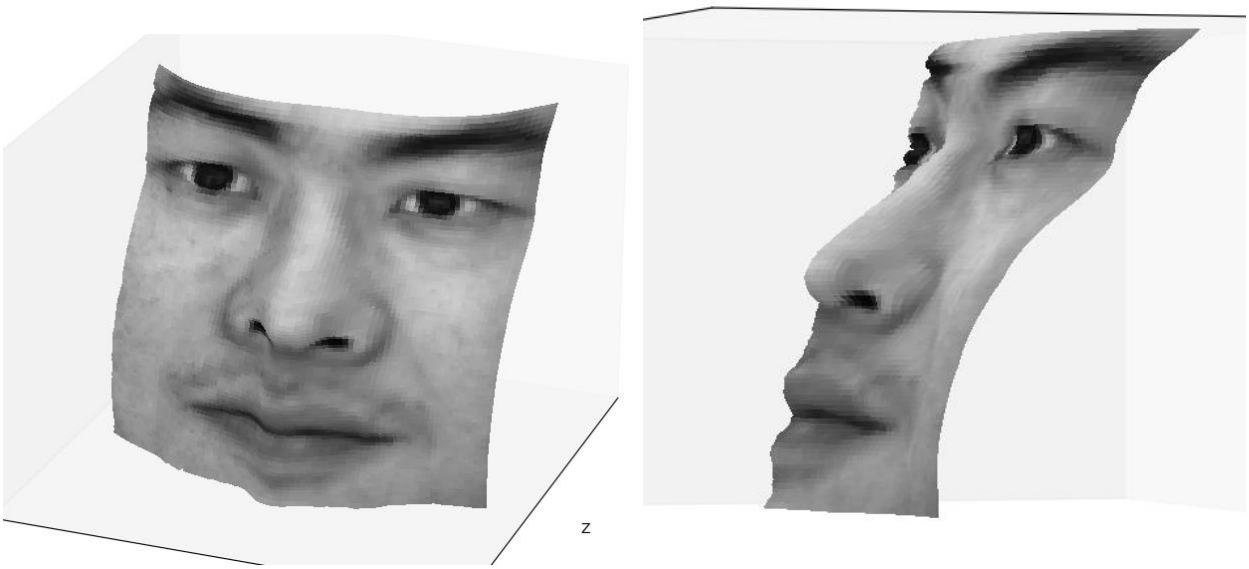
B: Compute Height Map

5) For every subject, display the surface height map by integration. Select one subject, list height map images computed using different integration method and from different views; for other subjects, only from different views, using the method that you think performs best. When inserting results images into your report, you should resize/compress them appropriately to keep the file size manageable -- but make sure that the correctness and quality of your output can be clearly and easily judged.

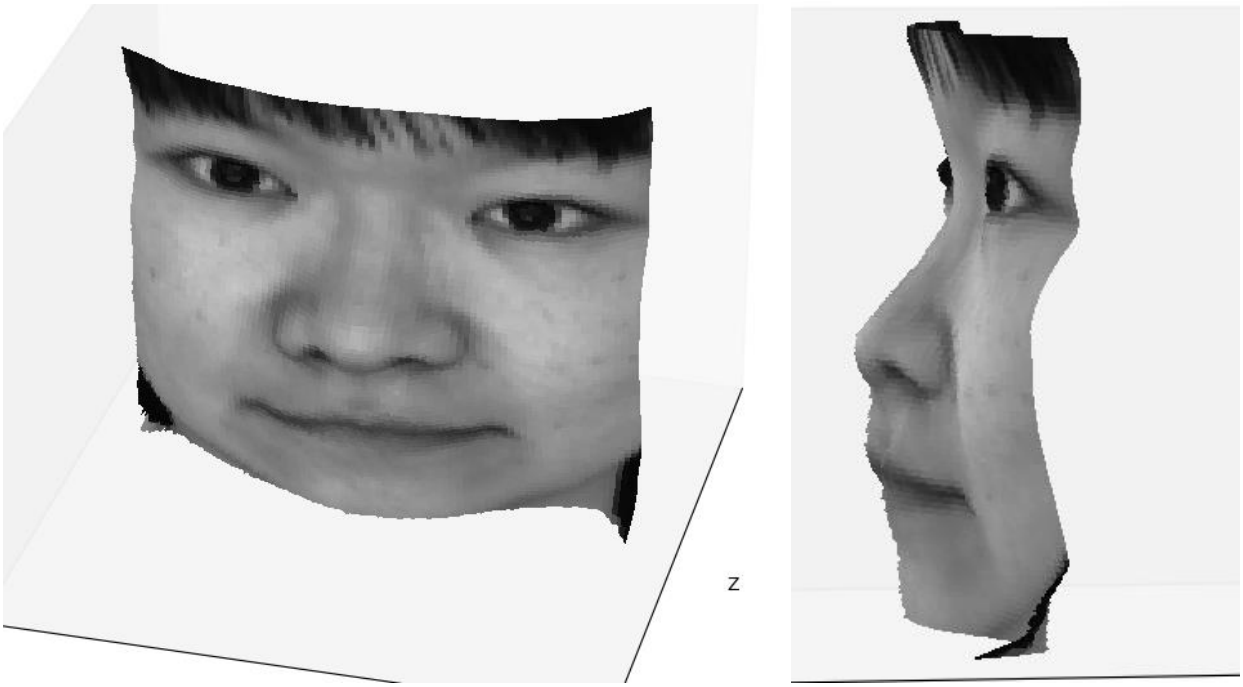
B01



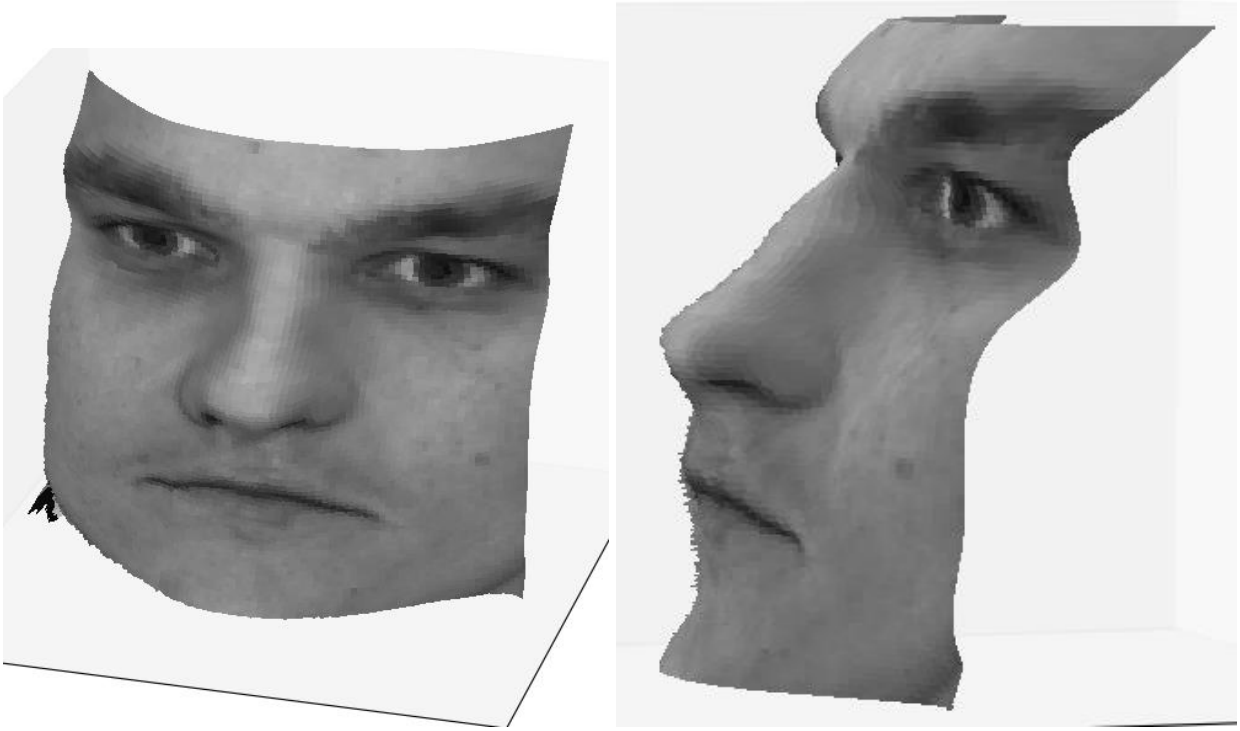
B02



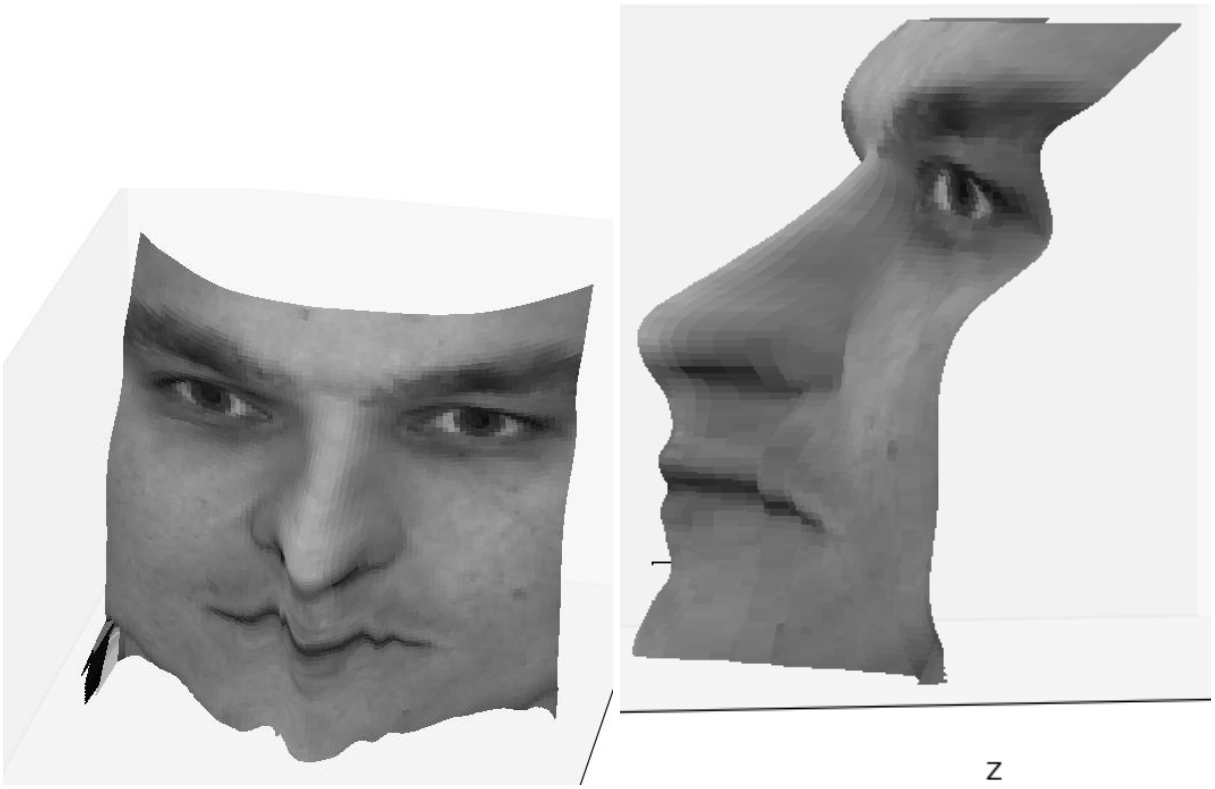
B05



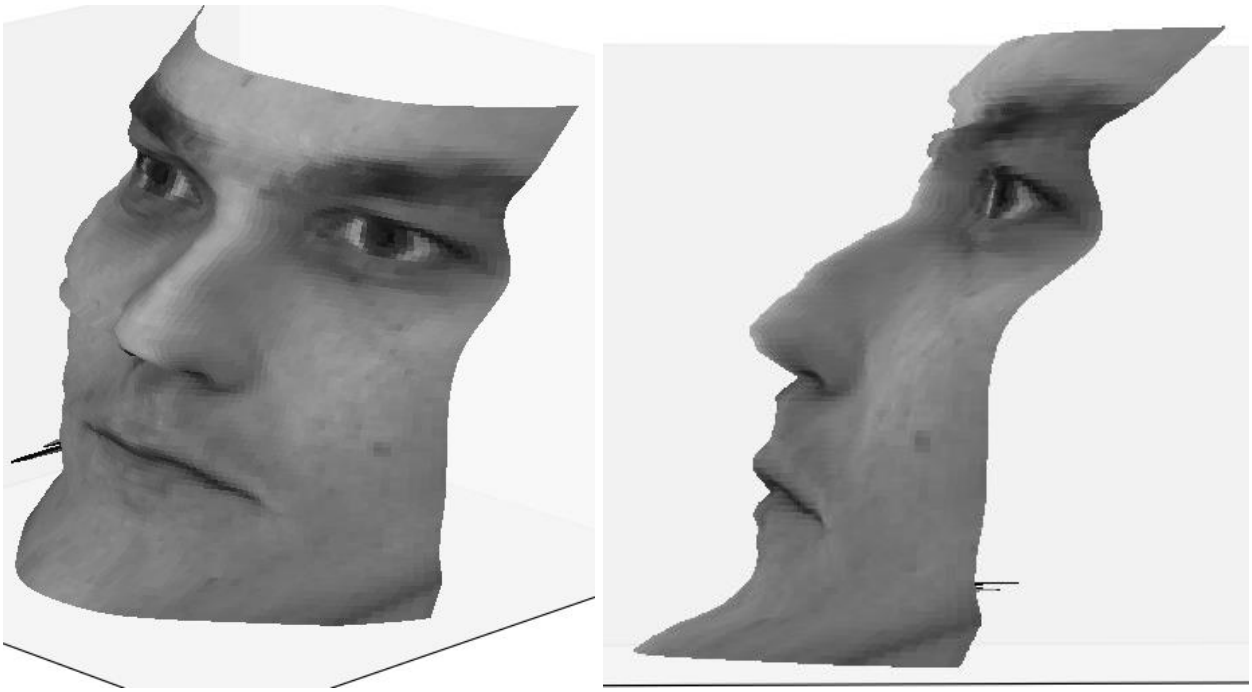
B07 - Random



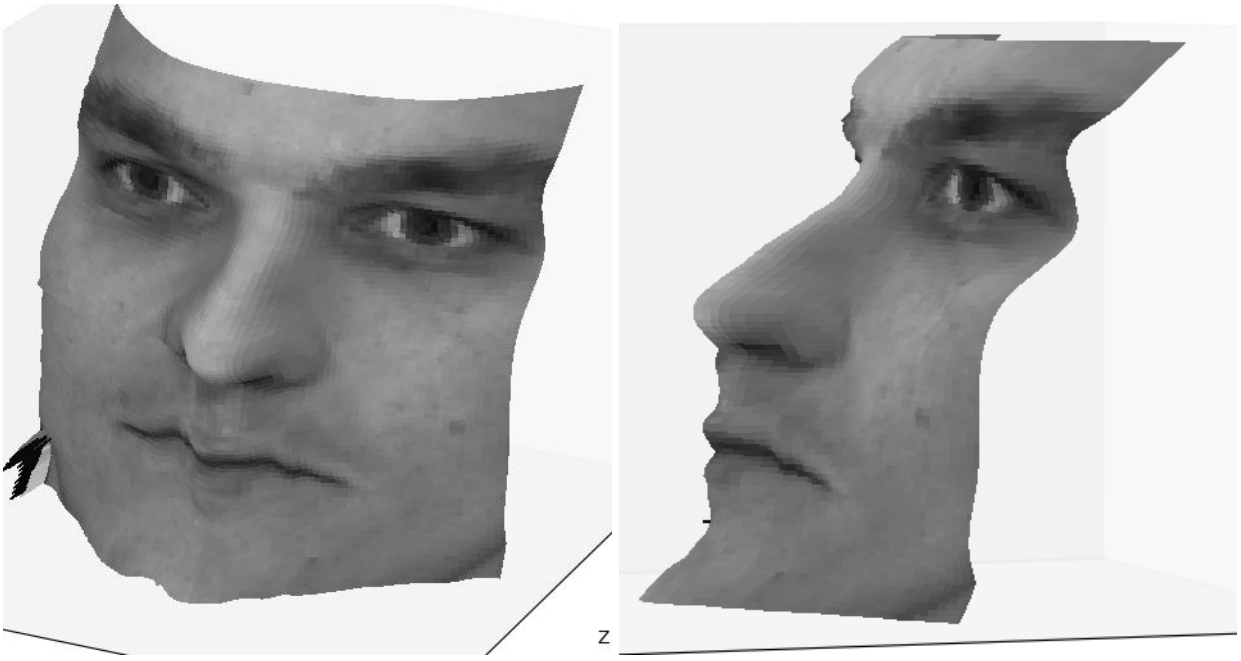
B07 – Row



B07 – Column



B07 - Average



6) Which integration method produces the best result and why?

The random method produces the best result on all of the samples. Because of the way random paths are generated, the method randomly includes the information of each row or column above the destination.

7) Compare the average execution time (only on your selected subject, “average” here means you should repeat the execution for several times to reduce random error) with each integration method, and analyze the cause of what you’ve observed:

   There is no distinct difference between average, row, and column. Random method takes approximately a minute because it needs to loop through each pixel with 25 iterations and get the average for the height map.

Integration method	Execution time
random	53.664
average	0.001016
row	0.001002
column	0.001007

C: Violation of the assumptions

8) Discuss how the Yale Face data violate the assumptions of the shape-from-shading method covered in the slides.

(1) Lack of perfect local shading

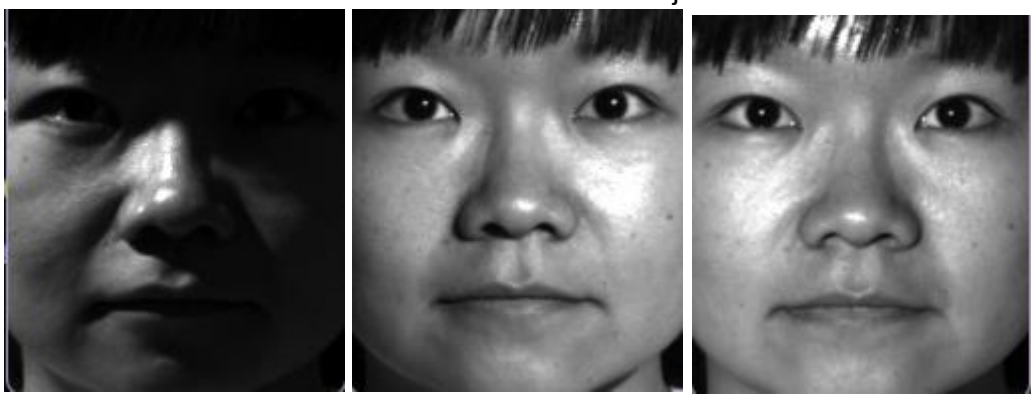
Perfect local shading model requires each point on the service only receive light from the light source, which is almost impossible to be done in the real life.

(2) Not having the same object configuration

The posture of the person or their facial expression would change between images and thus the image would not be fully aligned.

(3) Lambertian Objects too hard to realize

Images in the sample like the ones below has obvious specular reflection and that would not be an idea Lambertian object



Part 3: Extra Credit

Post any extra credit for parts 1 or 2 here. Don't forget to include references, an explanation, and outputs to receive credit. Refer to the assignment for suggested outputs.

Part 1 Multiple image stitching

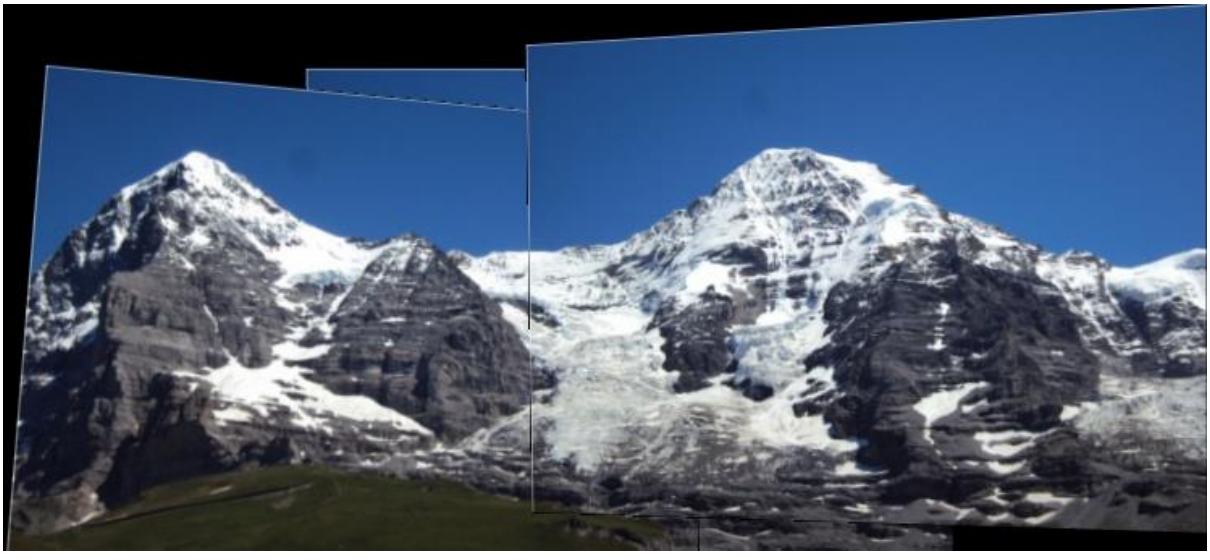
The multi-image stitch is done by reading and saving all the image in a list. Every time we stitch two of the images and push the stitched image back into the list. The loop will stop until there is only one image in the list. (The inlier code provided will fail if you feed in two images with different shape, hence I didn't provide the inlier matches for the images)

```
img_name = ['3.jpg','2.jpg','1.jpg']
img_list = []

for name in img_name:
    img_list.append(cv2.imread(img_dir + name))

while len(img_list) > 1 :
    img1 = img_list.pop()
    img2 = img_list.pop()
    stitched_img = stitch_image(img1,img2)
    img_list.insert(0,stitched_img)
```

Result:

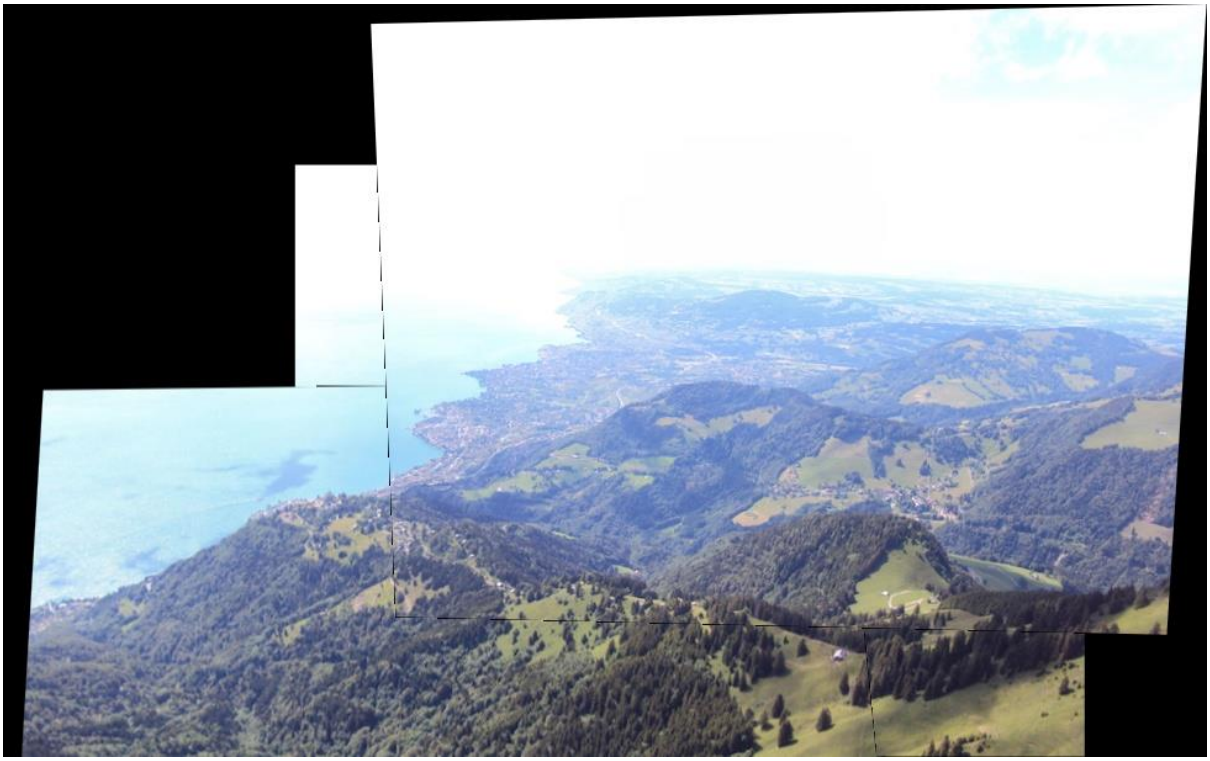
Hill

```
C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageSticking_shape_from_shading>python Yang_Derek_a3_p1.py
Inliers: 396 Average residuals: 0.17337322594834947
Inliers: 477 Average residuals: 0.23631926174597298
```



Ledge

```
C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageStiching_shape_from_shading>python Yang_Derek_a3_p1.py
Inliers: 386 Average residuals: 0.3594969818721268
Inliers: 256 Average residuals: 0.5993512501153715
```



Pier

```
C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageStiching_shape_from_shading>python Yang_Derek_a3_p1.py
Inliers: 259 Average residuals: 0.058832361368935805
Inliers: 193 Average residuals: 0.08995863445771668

C:\Users\derek\Documents\GitHub\myCode\CS543_CV\Lab3_imageStiching_shape_from_shading>
```