CS543/ECE549 Assignment 3

Name: fanmin shi
NetId: fshi5

Part 1: Homography estimation

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

For putative matching

I use `cdist` to compute the 'euclidean' distance between two descriptors. And then find the two closest matches from right descriptors for a given left descriptor. If those two distances are within 80%, then I deduced that the (left_descriptor, best_right_match) passes the ratio test.

```
dist = scipy.spatial.distance.cdist(left_des, right_des,'euclidean')
left_pts = []
right_pts = []
for l_idx in range(len(dist)):
    d = dist[l_idx]
    m, n = np.argsort(d)[:2]
    if d[m] < 0.8 * d[n]:
        left_pts.append(left_kp[l_idx].pt)
        right_pts.append(right_kp[m].pt)

left_pts = np.float32(left_pts)
right_pts = np.float32(right_pts)
```

For RANSAC

My RANSAC is based on https://en.wikipedia.org/wiki/Random_sample_consensus instead of the one from the lecture slide. The difference is that I find the inliers given len(inliers) > D with the minimum errors.

I have the following interface definition

```
# Initial number of points S
# D Number of close data points required to assert that a model fits well
to data.
# Distance threshold T for inliers
# Number of steps
def ransack(S, D, T, N, left_pts, right_pts):
```

I used the following parameters

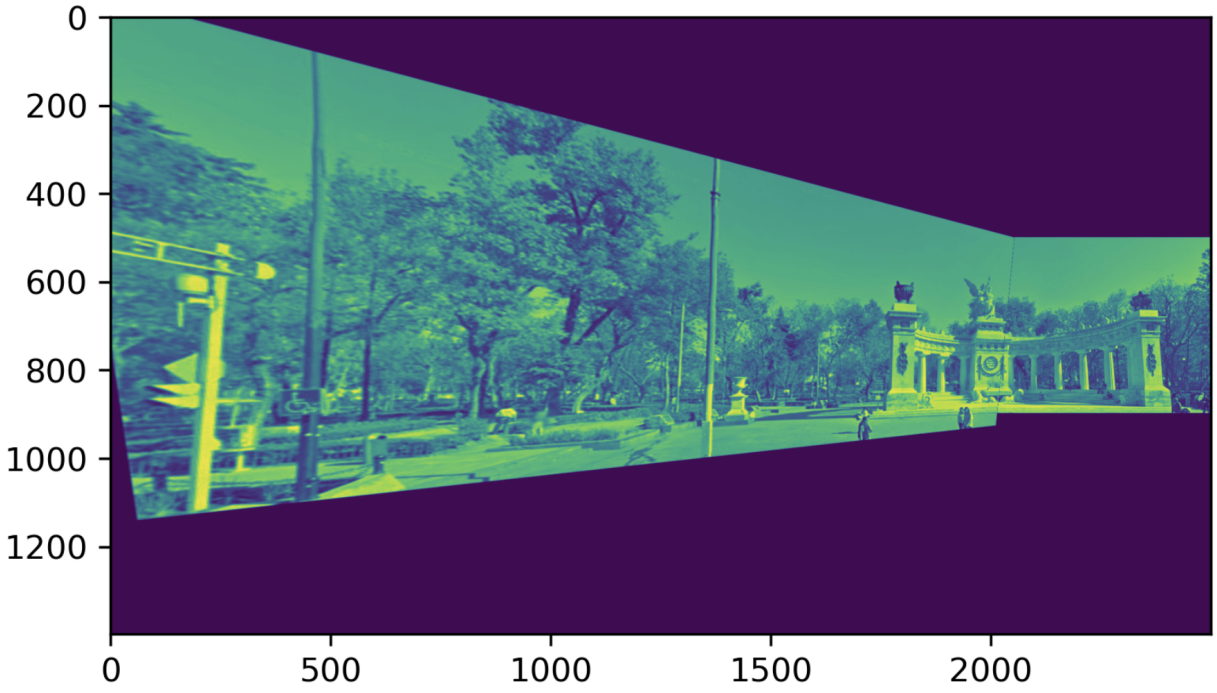
```
S = 4
T = 5
# assume 30% percent inlier ratio
D = np.floor(len(left_pts) *.30)
N = 500
```

I noticed that T is too small e.g .5, then my `ransack` won't find any inliers. So I need to increase it to 5. Then the average error will be 0.6735791696899763 and return inliers will be around 76. Also I played around D and found out that if D is too large e.g 95% inlier ratio, then my ransac won't find inliners that size. If D is too small e.g 5% inlier ratio, then the computed homography isn't good enough. So I found 30% - 50% a good ratio for the best 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.

Iniers = 79
Average residual = 0.7722913092674456

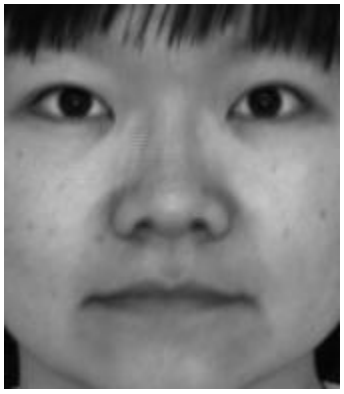
C: Display the final result of your stitching.



Part 2: Shape from shading

A: Estimate the albedo and surface normals

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



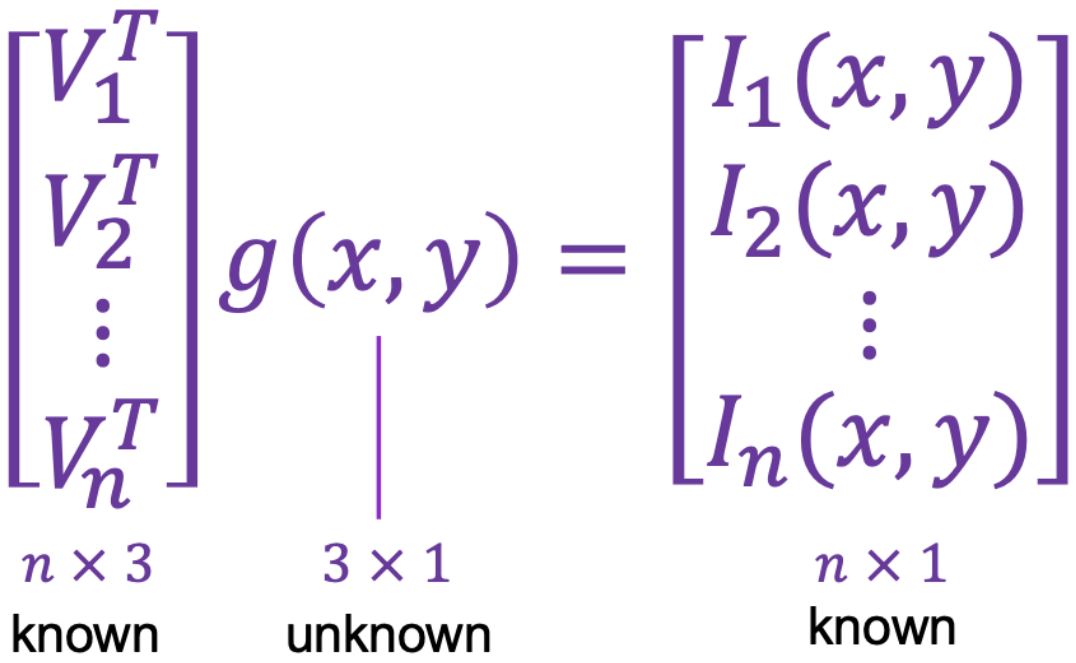
2) What implementation choices did you make? How did it affect the quality and speed of your solution?

   For calculating `photometric_stereo`,

I noticed that I need to produce **g** that's 3 x M x N instead of the slide shown below. So I create an I matrix where each row is 1D of pixel value.

```
 I = np.zeros(shape=(imarray.shape[-1], imarray.shape[0] *
imarray.shape[1]))
    for idx in range(imarray.shape[-1]):
        I[idx] = imarray[:,:,idx].flatten()
```

Then I called `np.linalg.lstsq(light_dirs, I)` to solve for g.



For memory optimization:

I learned that a numpy array is a continuous block of memory. So I pre-allocated the numpy array with the desired dimension and then assigned the value to the array as shown below

```
I = np.zeros(shape=(imarray.shape[-1], imarray.shape[0] *
imarray.shape[1]))
    for idx in range(imarray.shape[-1]):
        I[idx] = imarray[:,:,idx].flatten()
…
albedo_image_1D = np.zeros(imarray.shape[0] * imarray.shape[1])
surface_normals_2d = np.zeros((3, imarray.shape[0] *
imarray.shape[1]))
```

By doing that, I avoid using `np.vstack` and `np.concatenate` to create the desired array which will create a new array each time. This is quite slow. Hence, pre-allocating arrays are significantly faster.

For random path:

I implemented DP for random paths where you don't need to visit a path that you have seen before. The Idea is that we want to find the average random path at DP[Y][X] by looking at DP[Y-1][X] Or DP[Y][X-1] that we have computed before. Suppose DP is a 2D array that holds the average path value for location (Y, X). If the path is from the top, then DP[Y][X] = DP[Y-1][X] + dY[Y][X] where dY[Y][X] is the Y partial derivative at (Y,X). If the path is from the left, the DP[Y][X] = DP[Y][X-1] + dX[Y][X] where dX[Y][X] is the X partial derivative at (Y,X). Hence, if we want to random 20 paths. Then we just randomly pick a path from either top or left 20 times and then average its value. This is much faster because it has a runtime of O(MXN). If we don't do DP, then the runtime will be(MXNX(M+N)) where (M+N) is the length of the path.

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

   The given pictures violate the Lambertian assumption where there are shadows overcast part of the face. Hence, recovering **albedo** won't be exact. Also when taking the picture, the person might not be still. So the pixel of the exact face location between pictures might not align. Therefore, albedo and surface normal might not be exact.

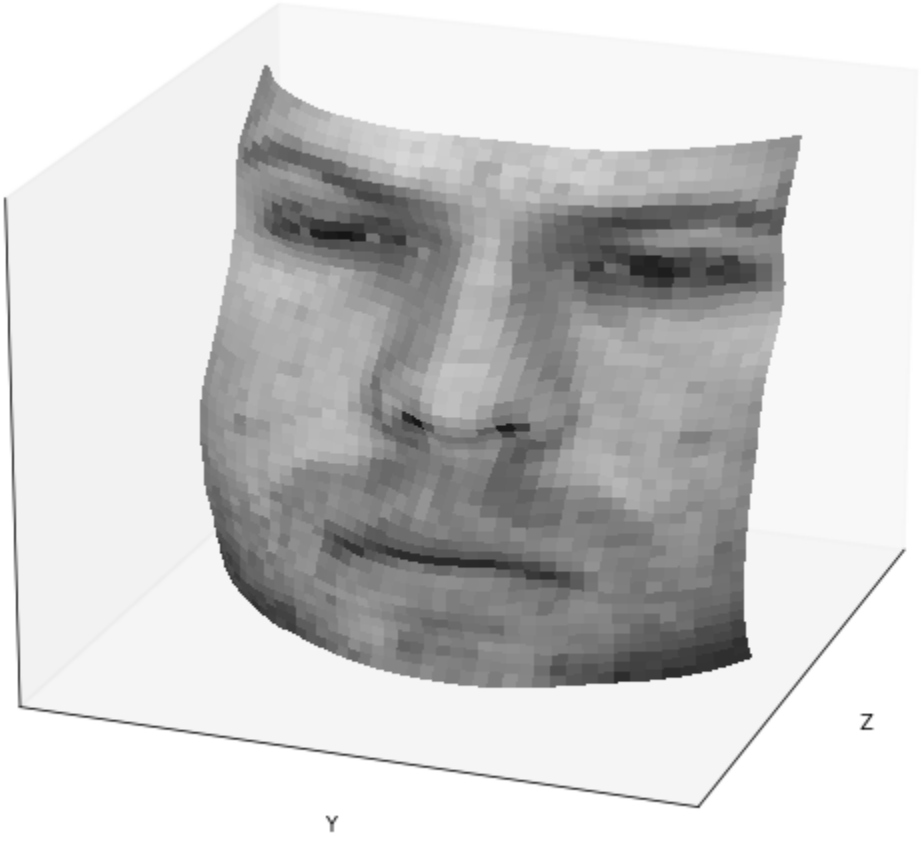
4) Display the surface normal estimation images below:

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.

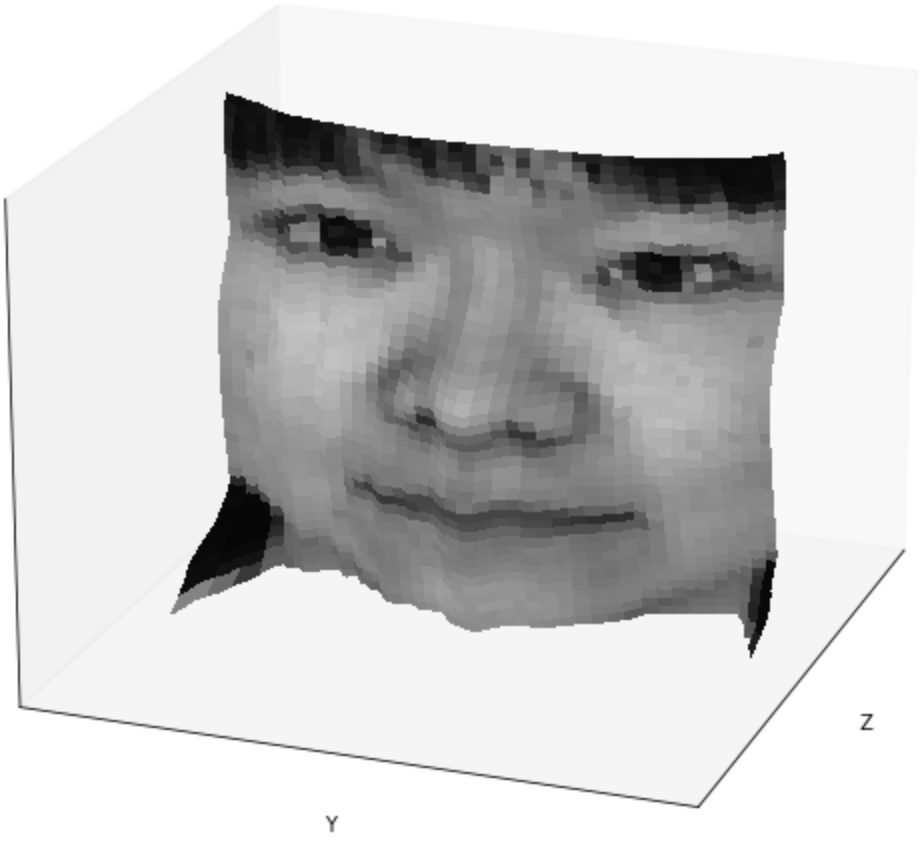
yaleB01



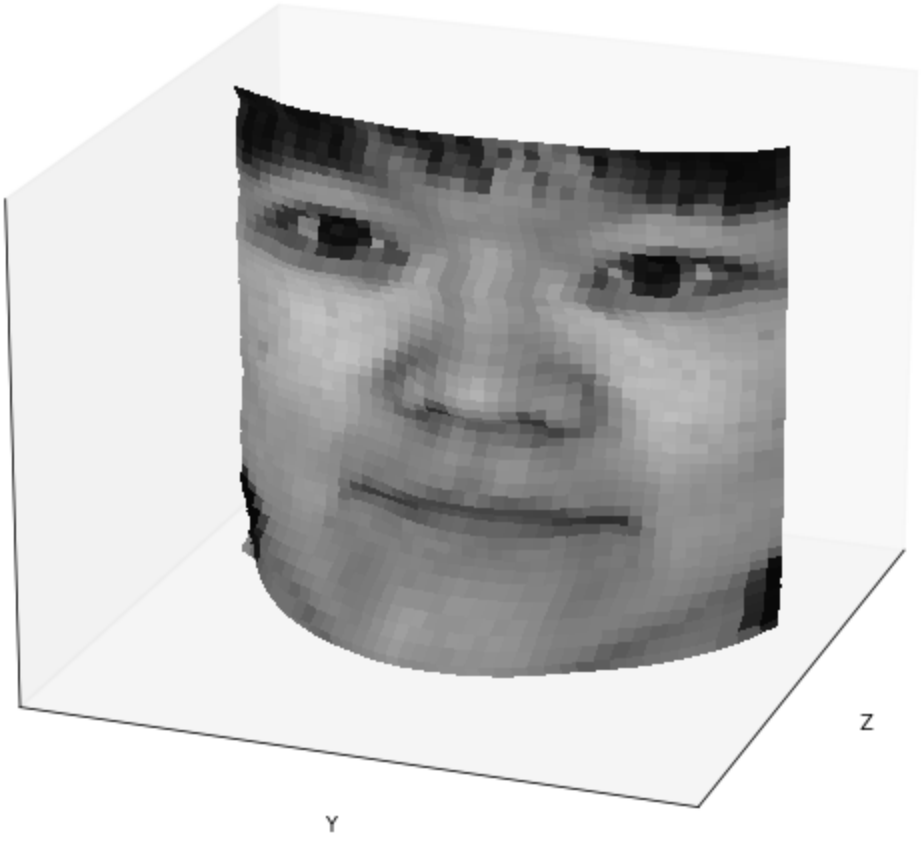
yaleB02

yaleB05

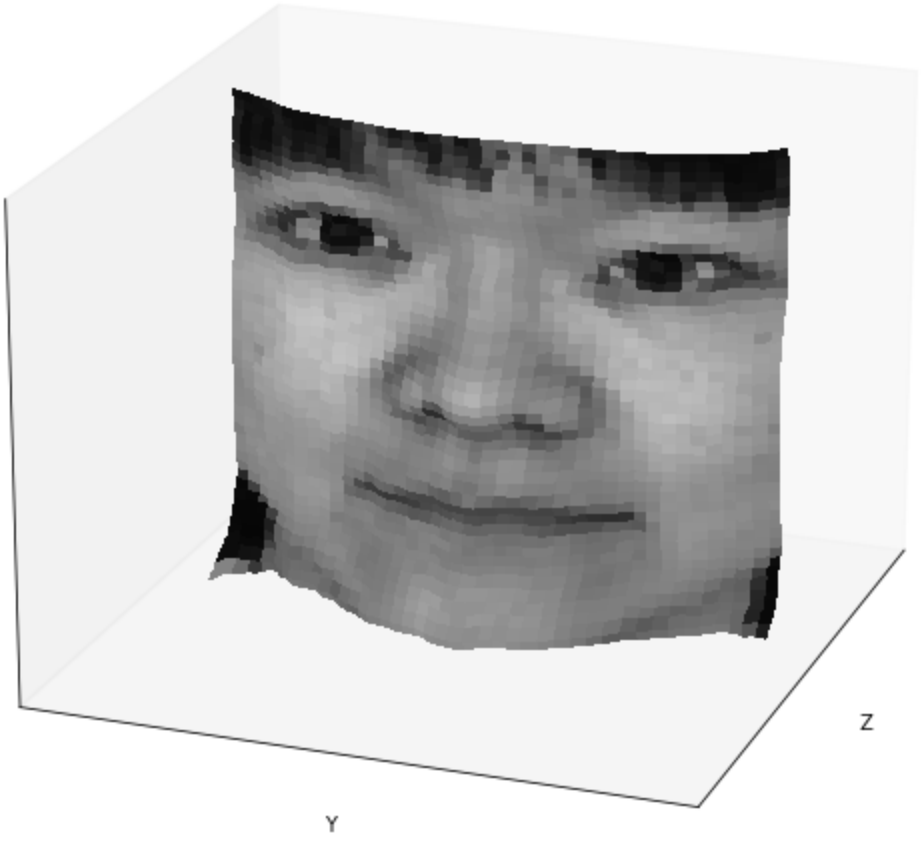
Row



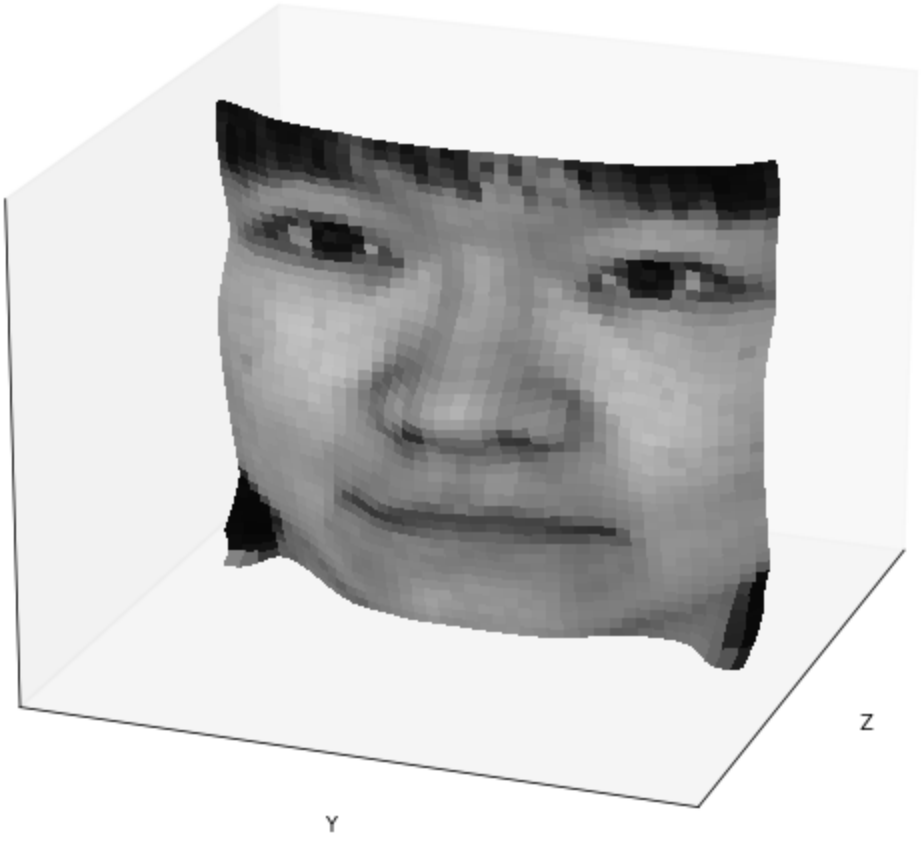
Column



Average



Random



yaleB07

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

   Random images appear to produce the best result. The reason is that the computed obedo and surface normal might not be perfect due to shadows overcast and non-still shots from each image. Hence, partial derivatives might be abrupt as a result. We can see clunky final images computed using Column and Row methods. The random method essentially achieves a “smoothing” factor by averaging derivatives integrated from different paths. Hence the result looks more natural and erases the abruptness.

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:

I only measure runtime for method `get_surface` because the preprocessing is constant for all integration methods. Adn I measure 5 run times and compute the average of those.

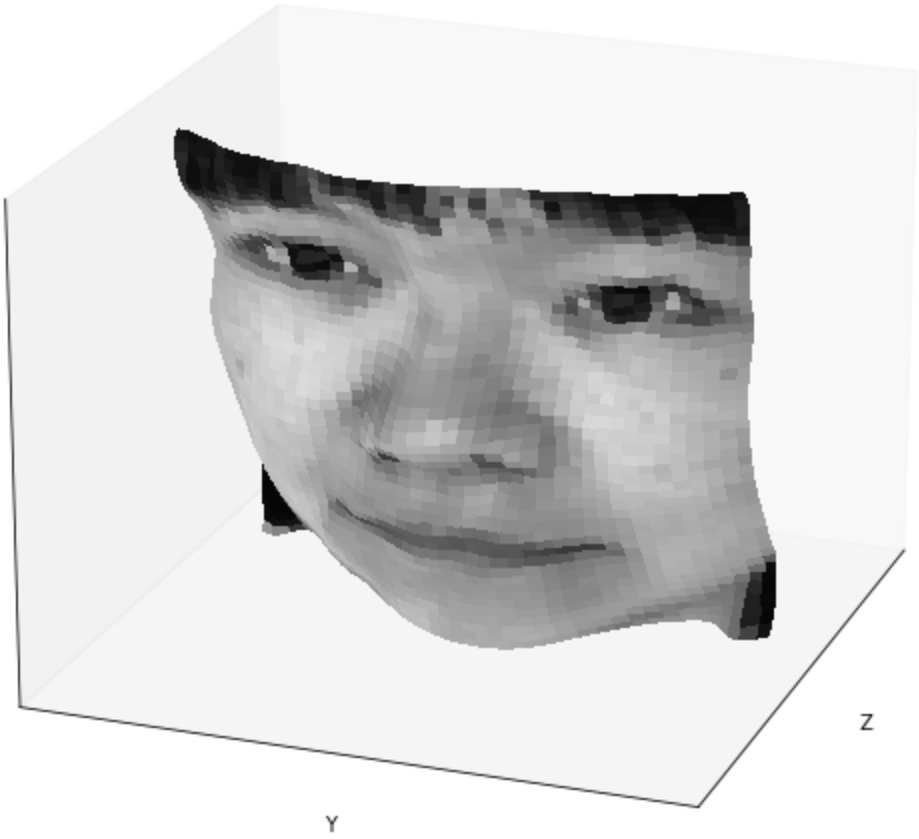
Integration method	Execution time
random	0.5611537456512451
average	0.0010996341705322265
row	0.0008494377136230469
column	0.0006197452545166016

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.

   First of all, the face is not a Lambertian object because skin is not perfectly diffusive because of microfacets that scatter incoming light randomly.
   Second of all, there are shadows that are casting over part of the face. Hence, we don't have the perfect information about the covered regions.
   Third of all, A set of pictures of an object, obtained in exactly the same camera/object Configuration can not be true because the person can't sit exactly still when taking different pictures.

9) Choose one subject and attempt to select a subset of all viewpoints that better match the assumptions of the method. Show your results for that subset.



10) Discuss whether you were able to get any improvement over a reconstruction computed from all the viewpoints.

    It seems like the face is more 3d compared to the original constructions. In Particular, you can see the forehead. So I would say there is an improvement.

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 Extra credit:

I use `cdist` to compute the 'euclidean' distance between two descriptors. And then find the two closest matches from right descriptors for a given left descriptor. If those two distances are within 80%, then I deduced that the (left_descriptor, best_right_match) passes the ratio test.

```
dist = scipy.spatial.distance.cdist(left_des, right_des,'euclidean')
left_pts = []
right_pts = []
for l_idx in range(len(dist)):
    d = dist[l_idx]
    m, n = np.argsort(d)[:2]
    if d[m] < 0.8 * d[n]:
        left_pts.append(left_kp[l_idx].pt)
        right_pts.append(right_kp[m].pt)

left_pts = np.float32(left_pts)
right_pts = np.float32(right_pts)
```

Part 2 Extra credit.

I implemented DP for random paths where you don't need to visit a path that you have seen before. The Idea is that we want to find the average random path at DP[Y][X] by looking at DP[Y-1][X] Or DP[Y][X-1] that we have computed before. Suppose DP is a 2D array that holds the average path value for location (Y, X). If the path is from the top, then DP[Y][X] = DP[Y-1][X] + dY[Y][X] where dY[Y][X] is the Y partial derivative at (Y,X). If the path is from the left, the DP[Y][X] = DP[Y][X-1] + dX[Y][X] where dX[Y][X] is the X partial derivative at (Y,X). Hence, if we want to random 20 paths. Then we just randomly pick a path from either top or left 20 times and then average its value. This is much faster because it has a runtime of O(MXN). If we don't do DP, then the runtime will be(MXNX(M+N)) where (M+N) is the length of the path.