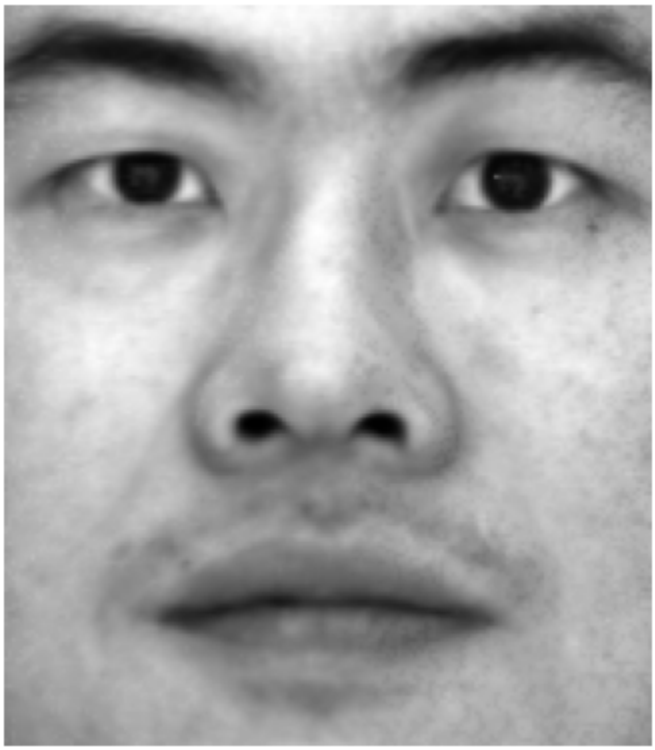
CS543/ECE549 Assignment 4

Your Name: Noel Mrowiec
Your NetId: mrowiec3

Part 1 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?
   Preprocessing: I took the difference between the image array and the ambient image. Photometric stereo: I reshaped the image array and found the least squared between the light_dirs and the image array. The norm of the least square resulted in the albedo

image. Dividing the least squared result by the albedo image resulted in the surface normals (with reshaping).
Get_surface:
For all the integration methods I divided the X and Y surface normals by the Z surface normal (see below). Then I used np.cumsum() to get the cumulative sum to get the x and y axis. The cum sum in x and y was used for the integration methods. Row integration method was just summing the cum sum of y with the first row of cum sum of x. Column integration method was just summing the cum sum of x with the first column of cum sum of y. Average integration method took the average of the row and column method. Finally, the random integration method goes over each pixel and iterates four times doing the following: creates random path with zeros and ones, do a path traversal from the path and accumulate the results. Then I average over the four iterations. The random method is by far the slowest and the fastest doing the row integration method, followed closely by the column method. Unfortunately, it looks like the quality isn't as good as I expected. It was a simple implementation method and quite fast. Future work can be to reduce the pixelated effect.

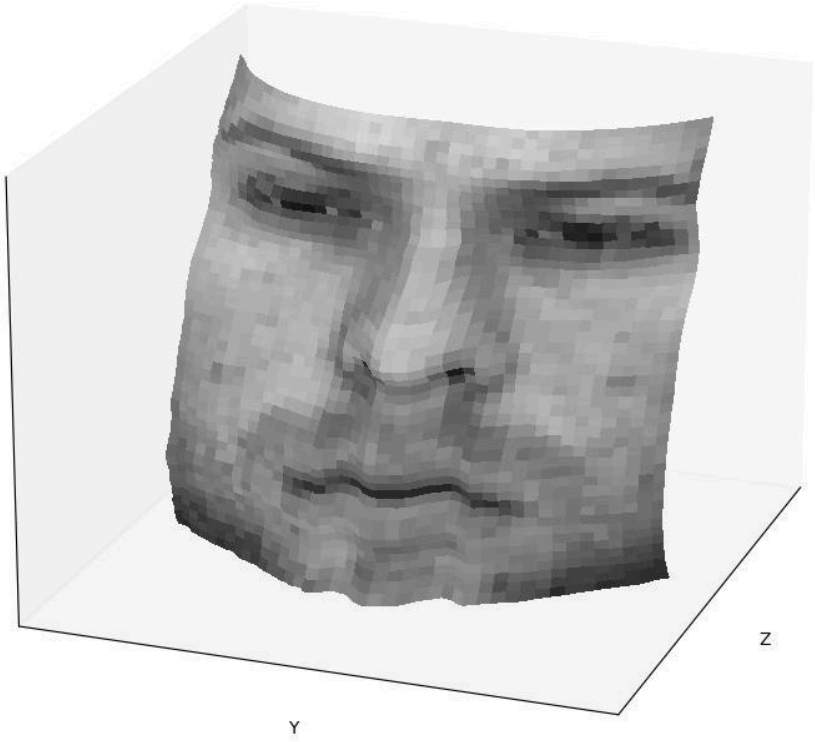
3) What are some artifacts and/or limitations of your implementation, and what are possible reasons for them?
   As discussed above, the resulting image is quite pixelated. Additionally, the row method creates too many vertical edges. There is correctly a ridge in the nose but also a ridge in the lips, which is incorrect. This is because I take only the first row of the cumulative sum across the X axis. This creates bigger differences as we move from column to column which creates the incorrect ridges in the lips. The column method flattens out the nose too much because I take only the first column of the cumulative sum across the Y axis. This creates bigger differences as we move from row to row but less when we move from column to column. Thus, the nose is too flat. The random method isn't good either because there are obvious artifacts at the chin and lips. This is due to the random method of summing; it has too many extraneous values. Averaging seems like the best method and the output looks better than the other because it is not flat as the column method, nor excessively bumpy going across the column. There is still an artifact in the lips of the average method. This is because the row method contributes too much difference across the columns. There is a bump out in the middle of the lips, which is incorrect.

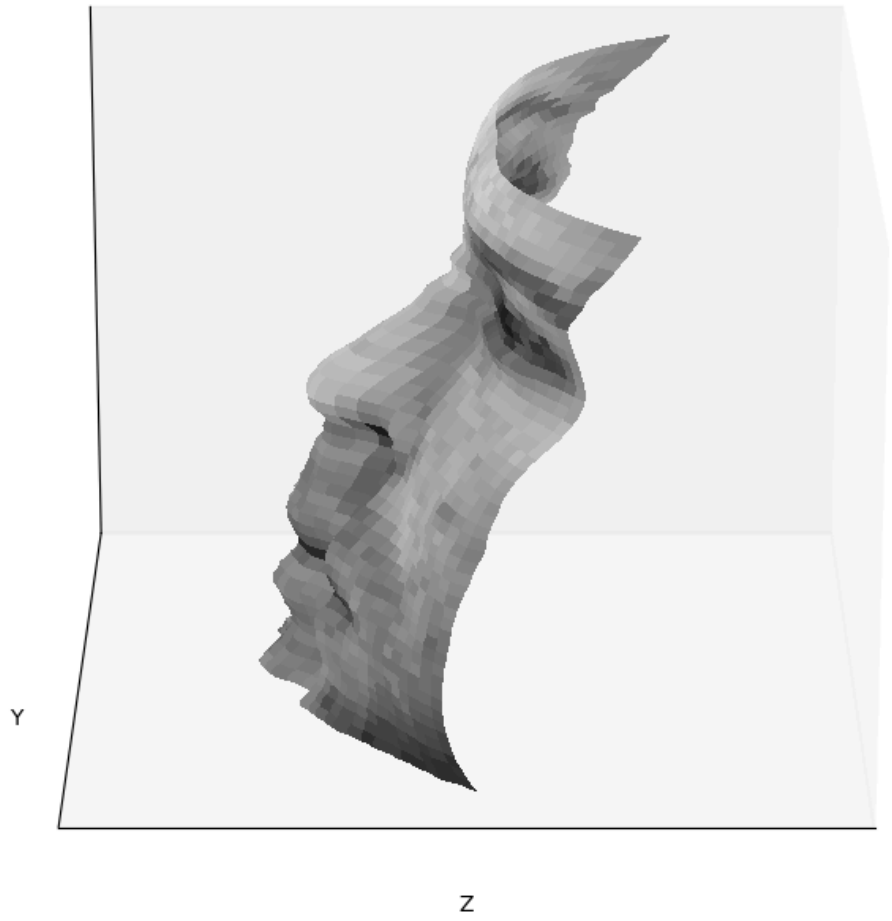
4) Display the surface normal estimation images below:

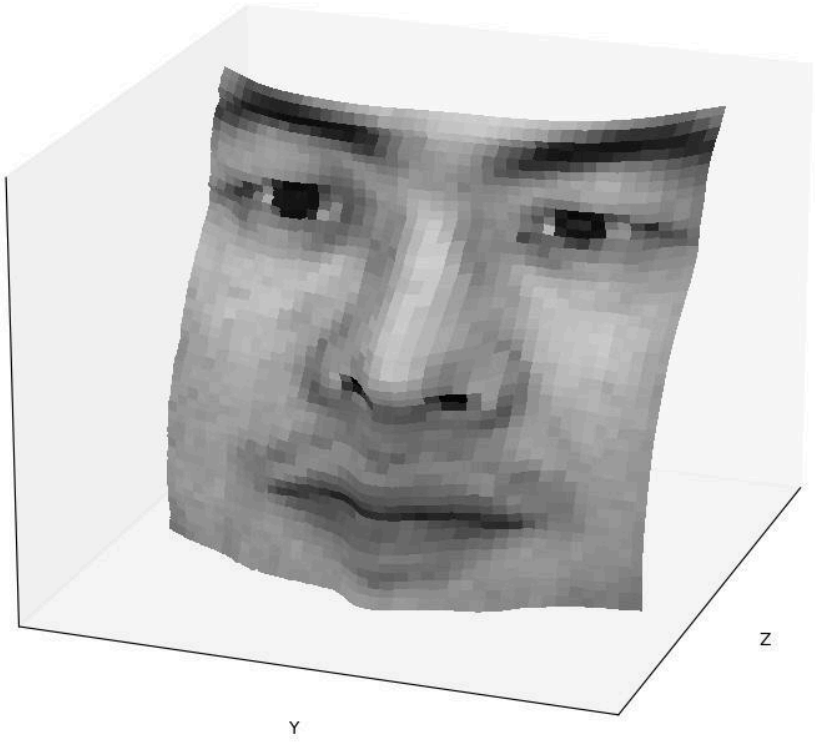
B: Compute Height Map

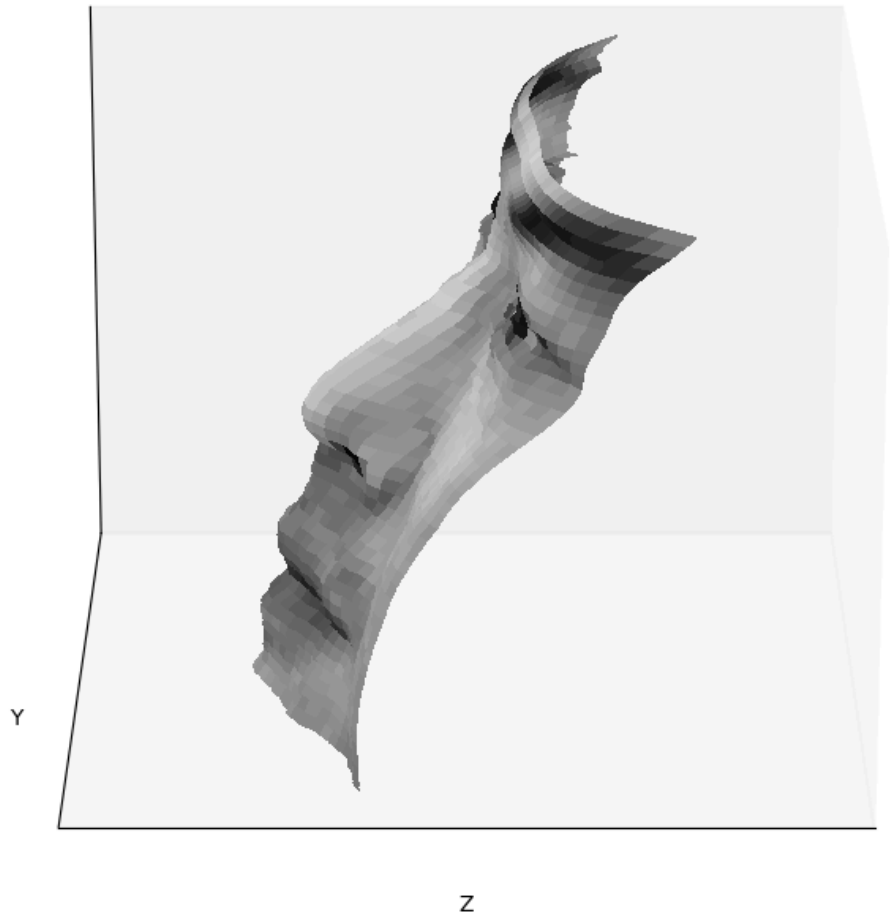
5) For every subject, display the surface height map by integration. Select one subject, list height map images computed using different integration methods and from different views; for other subjects, only from different views, using the method that you think performs best. When inserting results images 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.

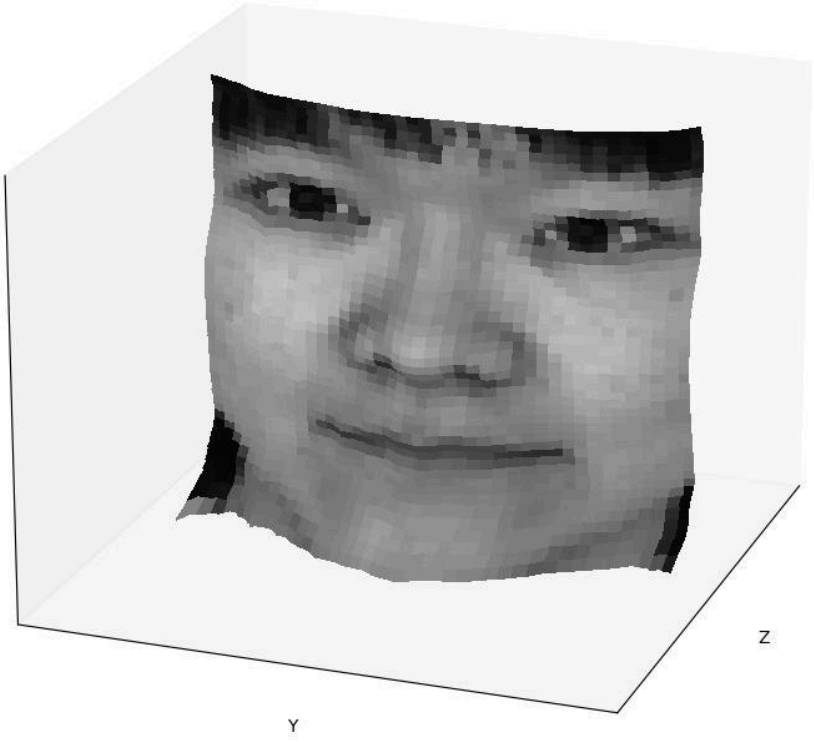
For every subject, display the surface height map by integration.
Using averaging.

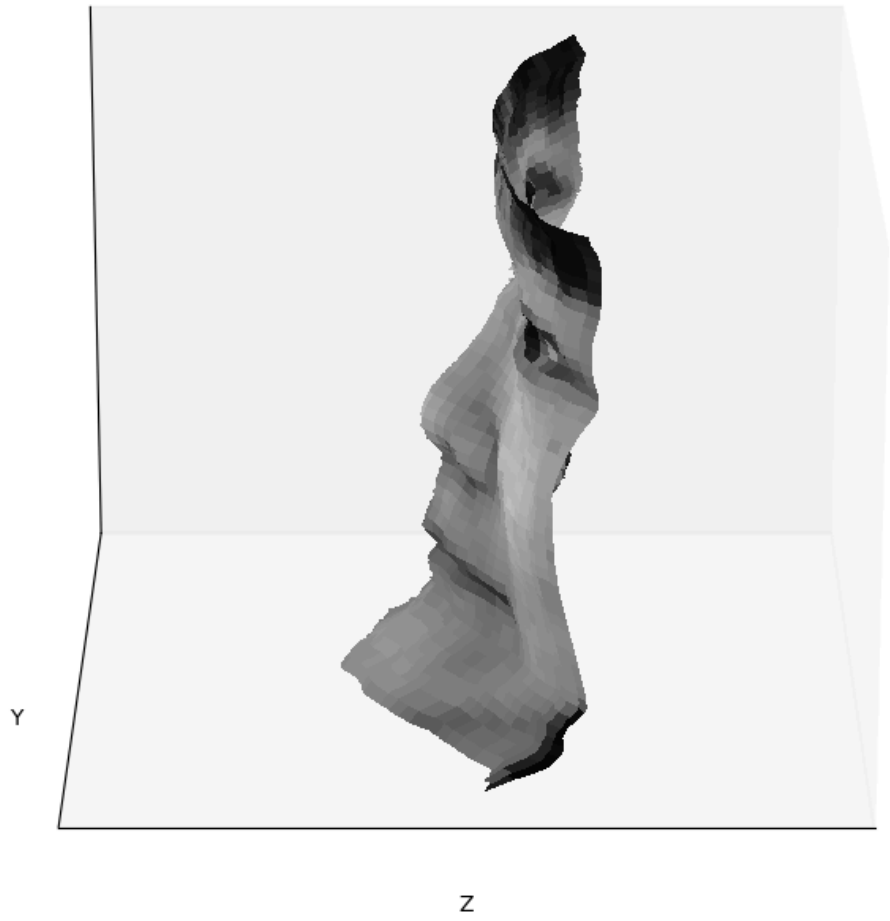


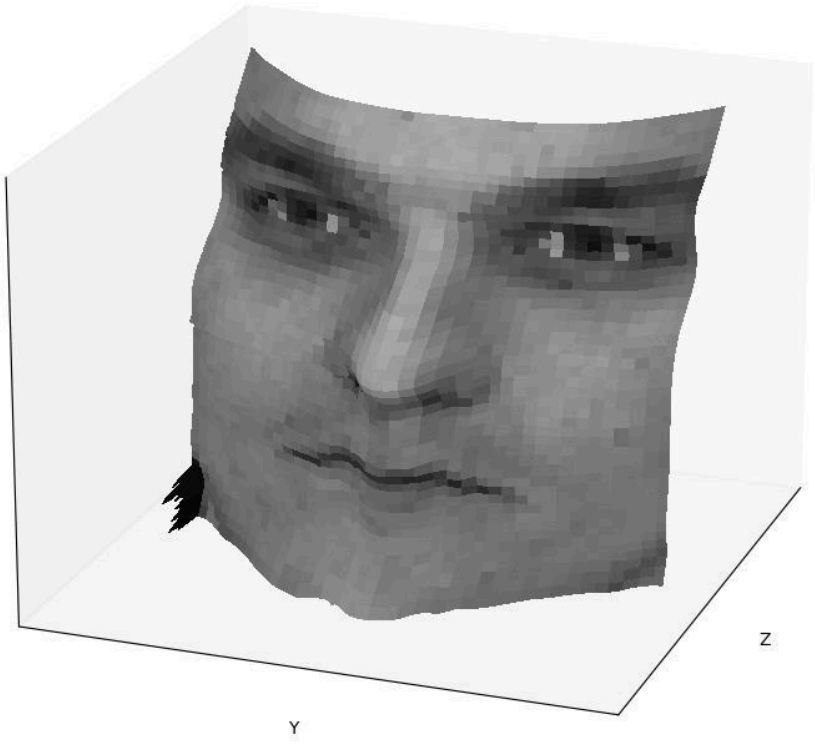

Y
Z

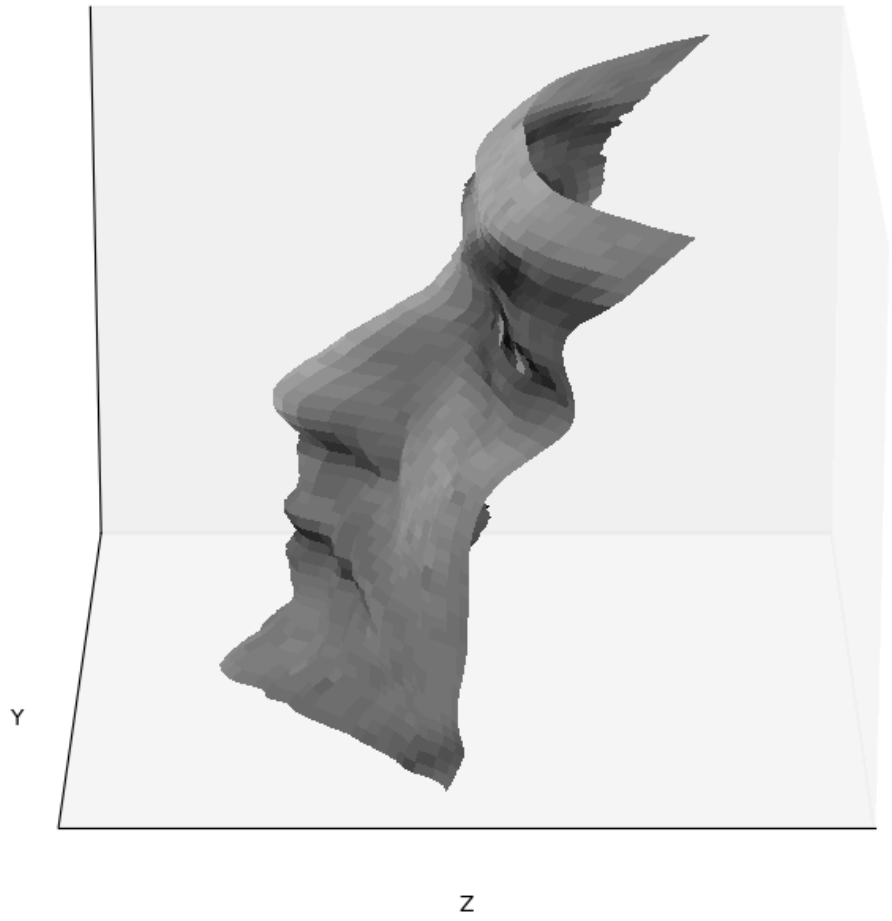

Z
Y


Y
Z


Z
Y


Y
Z

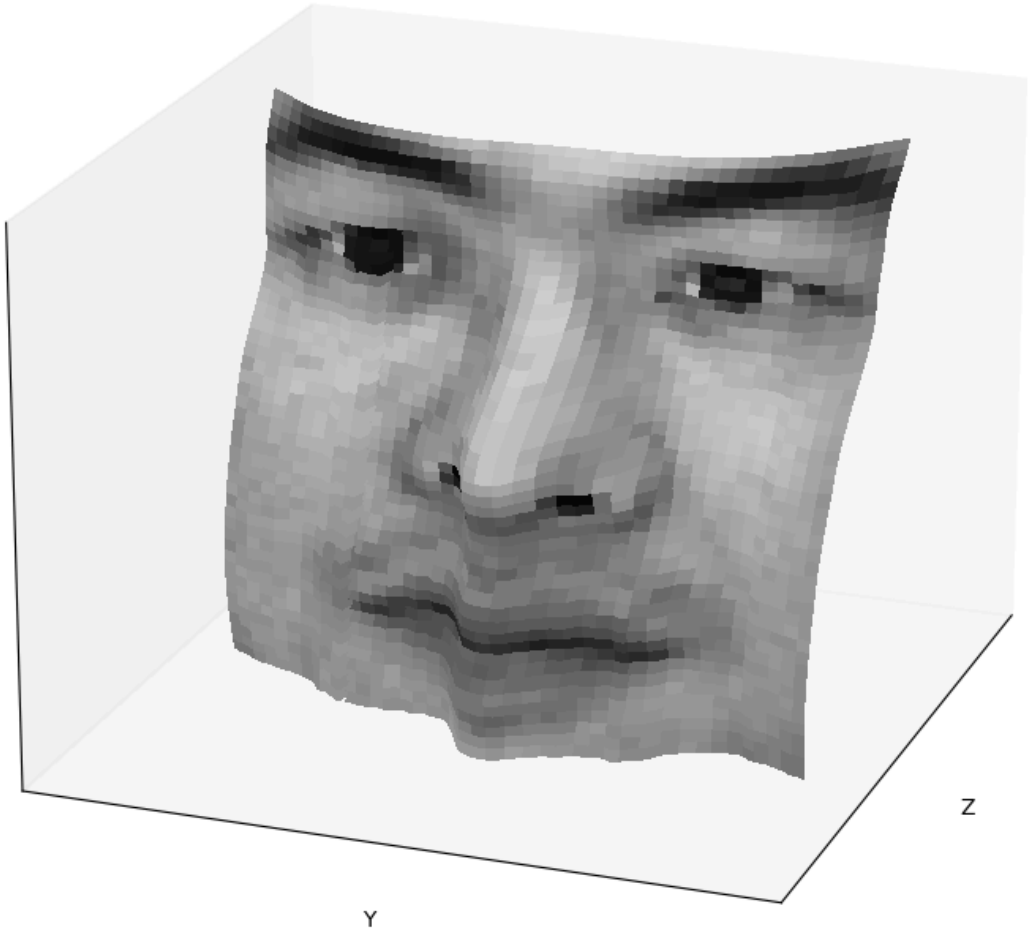

Z
Y

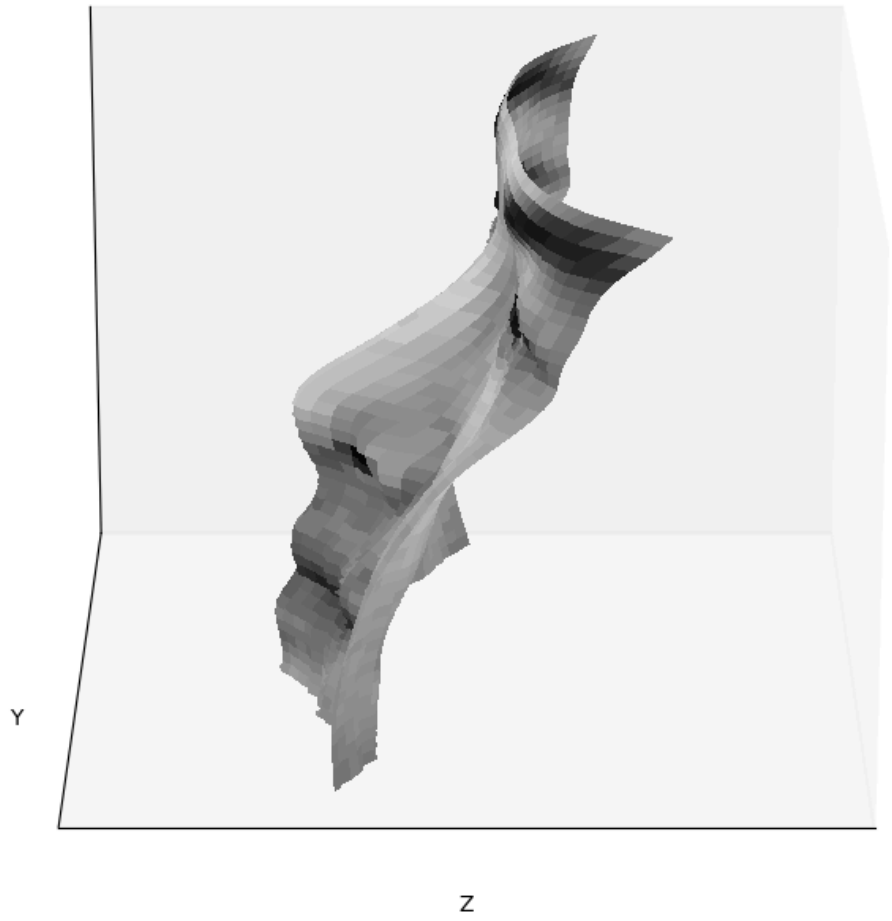


Select one subject, list height map images computed using different integration methods and from different views

subject_name = 'yaleB02'

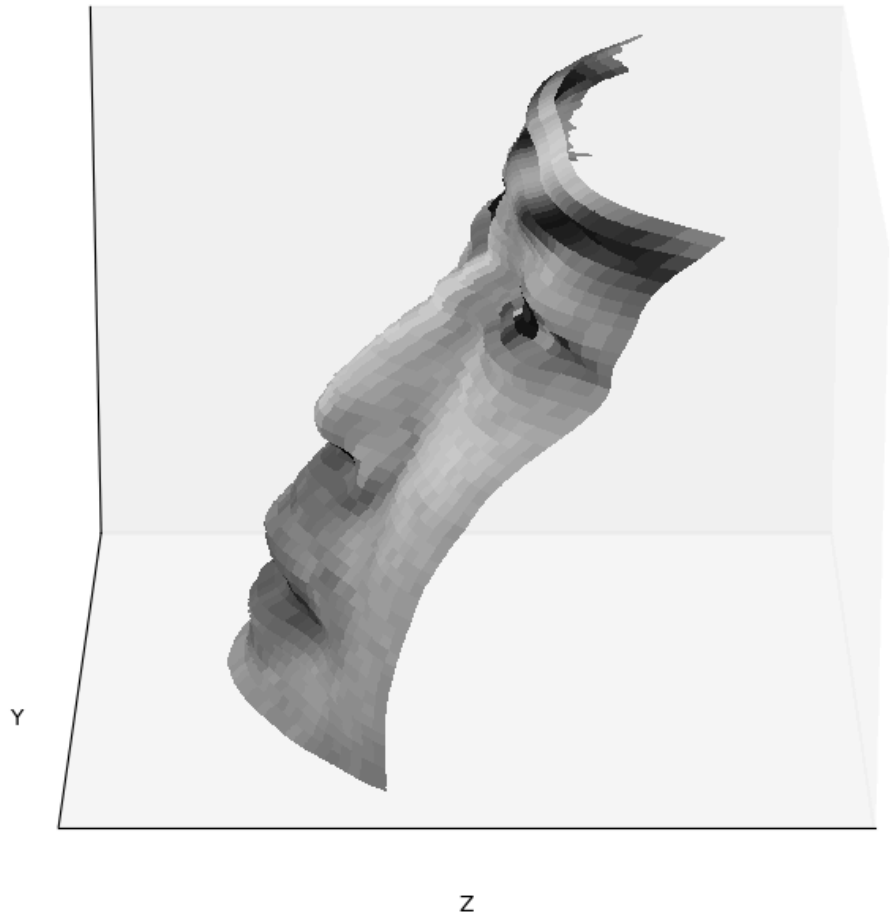
Integration_method = 'row'


Z
Y

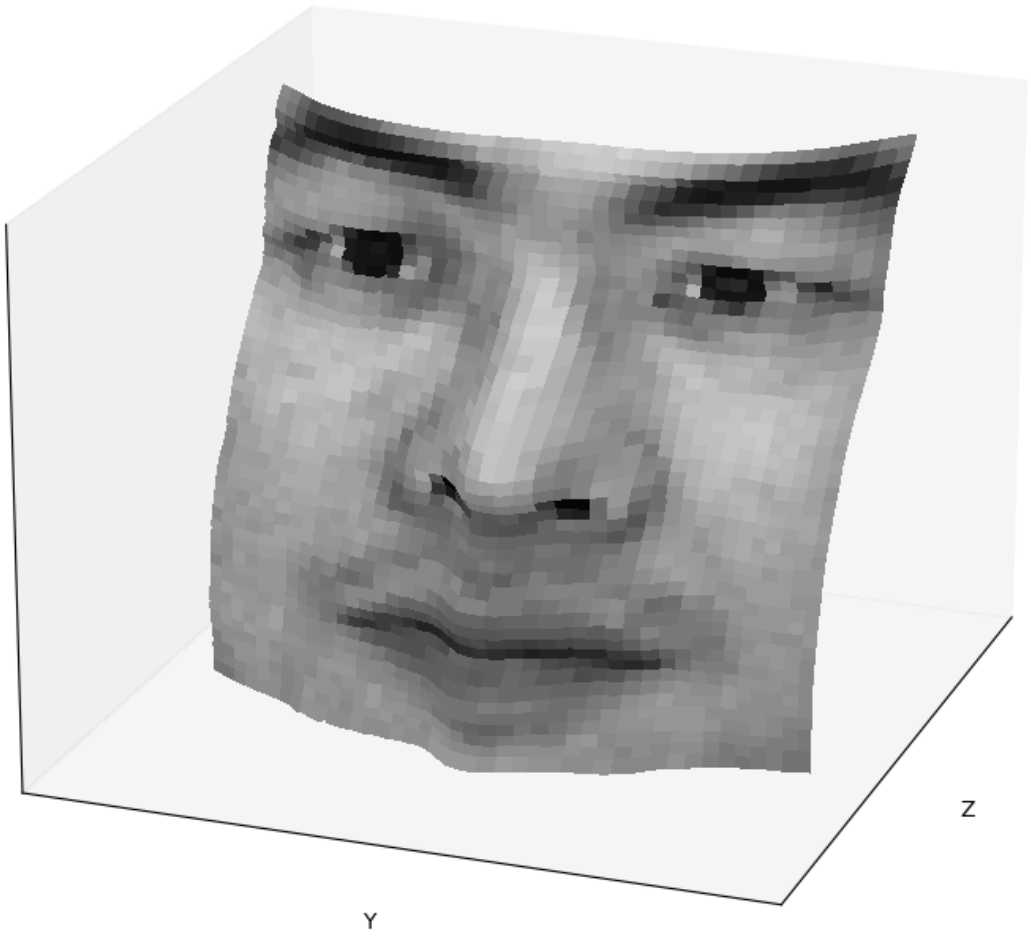


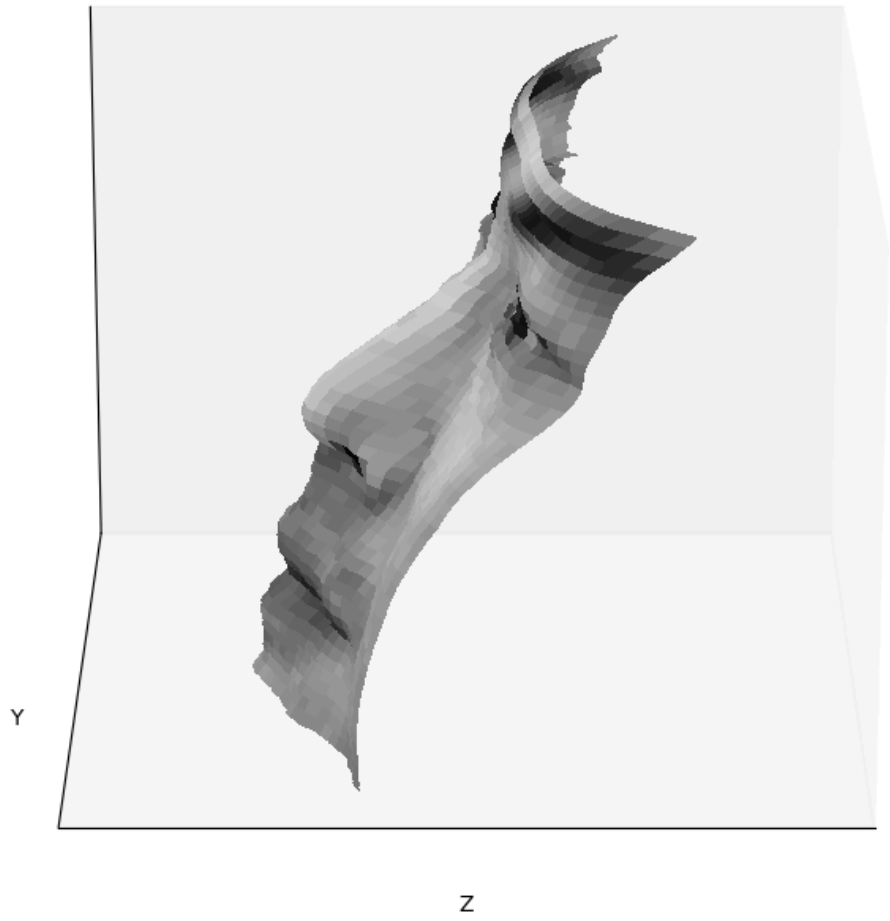
Integration_method = 'column'


Z
Y

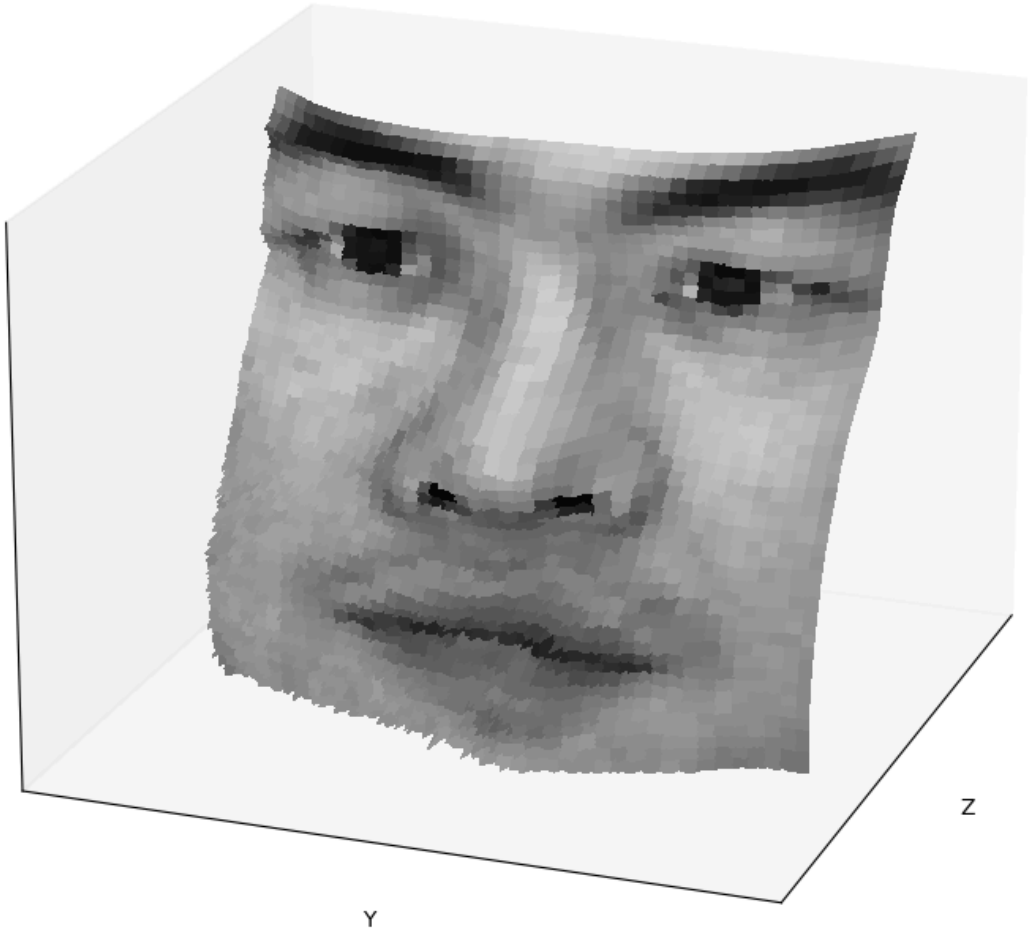


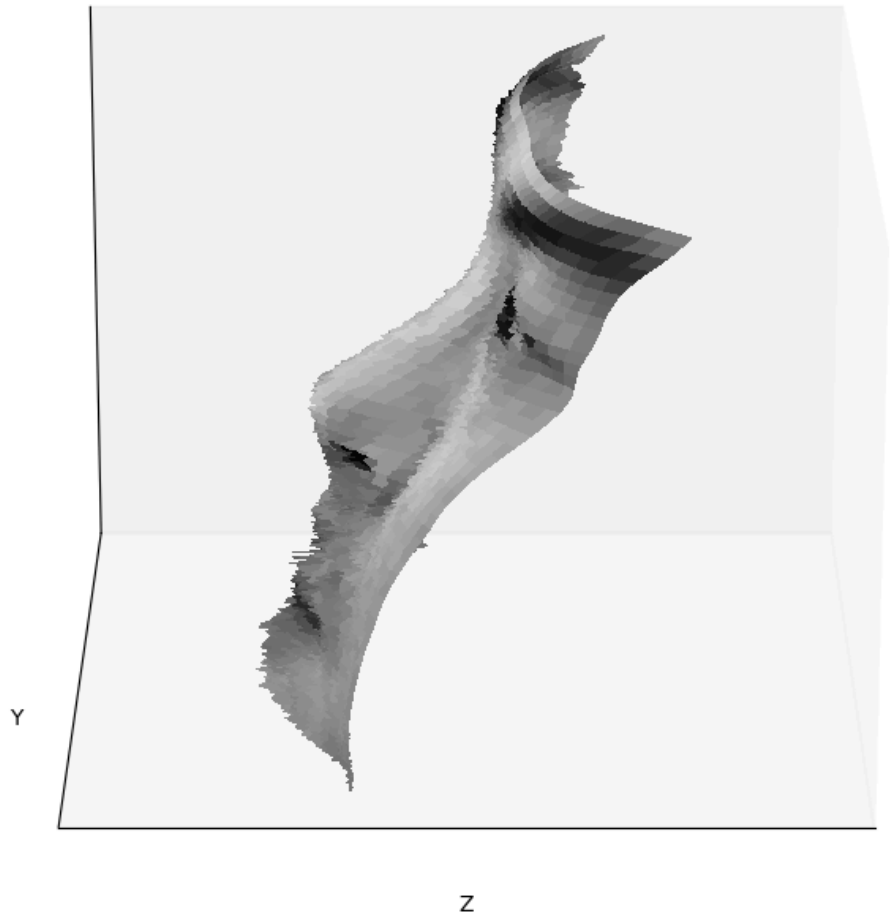
Integration_method = 'average'


Z
Y



Integration_method = 'random'


Z
Y



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

As already discussed above, the resulting image is quite pixelated. Additionally, the row method creates too many vertical edges. While there is correctly a ridge in the nose, there is also a ridge in the lips, which is incorrect. This happens because I take only the first row of the cumulative sum across the X axis. This creates bigger differences as we move from column to column, resulting in the incorrect ridges in the lips. The column method flattens out the nose too much because I take only the first column of the cumulative sum across the Y axis. This creates bigger differences as we move from row to row but fewer differences when we move from column to column, causing the nose to appear too flat. The random method isn't ideal either because there are obvious artifacts at the chin and lips. These artifacts are due to the random

method of summing, which introduces too many extraneous values. Averaging seems like the best method, and the output looks better than the others because it is not as flat as the column method, nor excessively bumpy across the columns. However, there is still an artifact in the lips with the averaging method. This is because the row method contributes too much difference across the columns, resulting in an incorrect bump in the middle of the lips.

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:

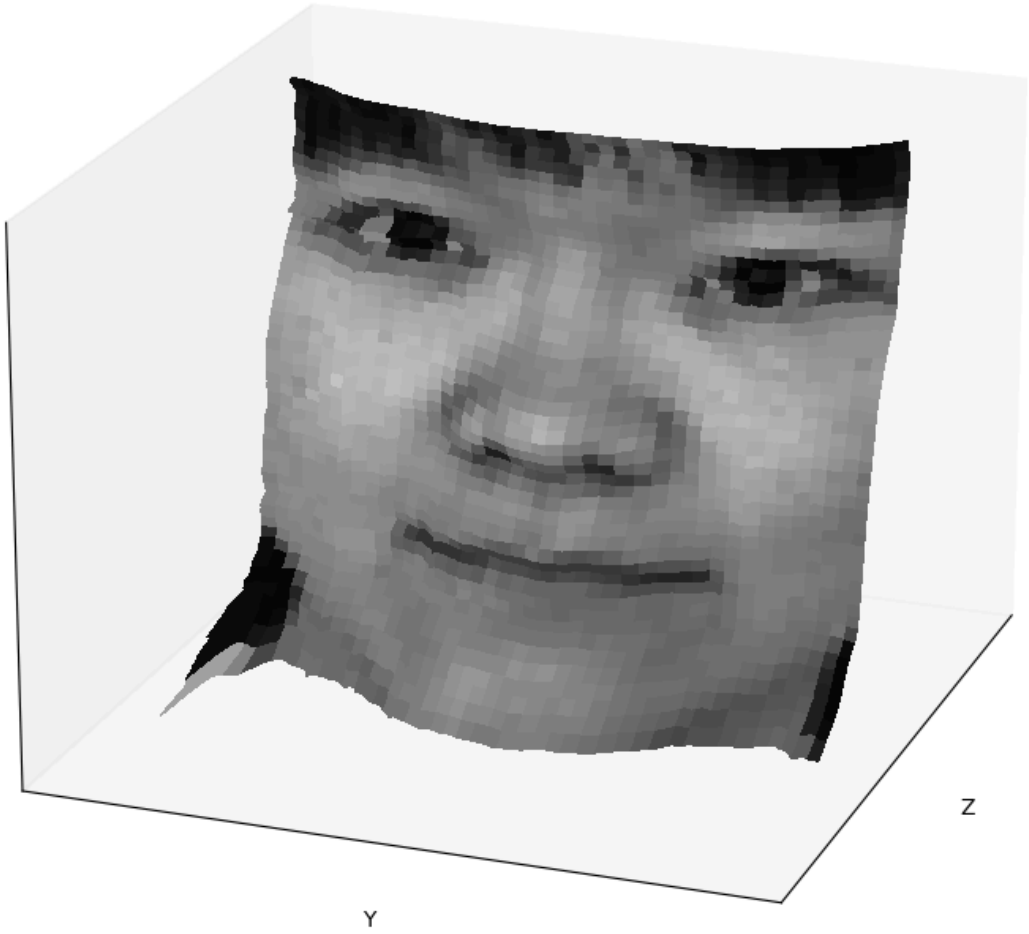
Integration method	Execution time
row	0.058 ms
column	0.048 ms
average	0.151 ms
random	10272.123 ms

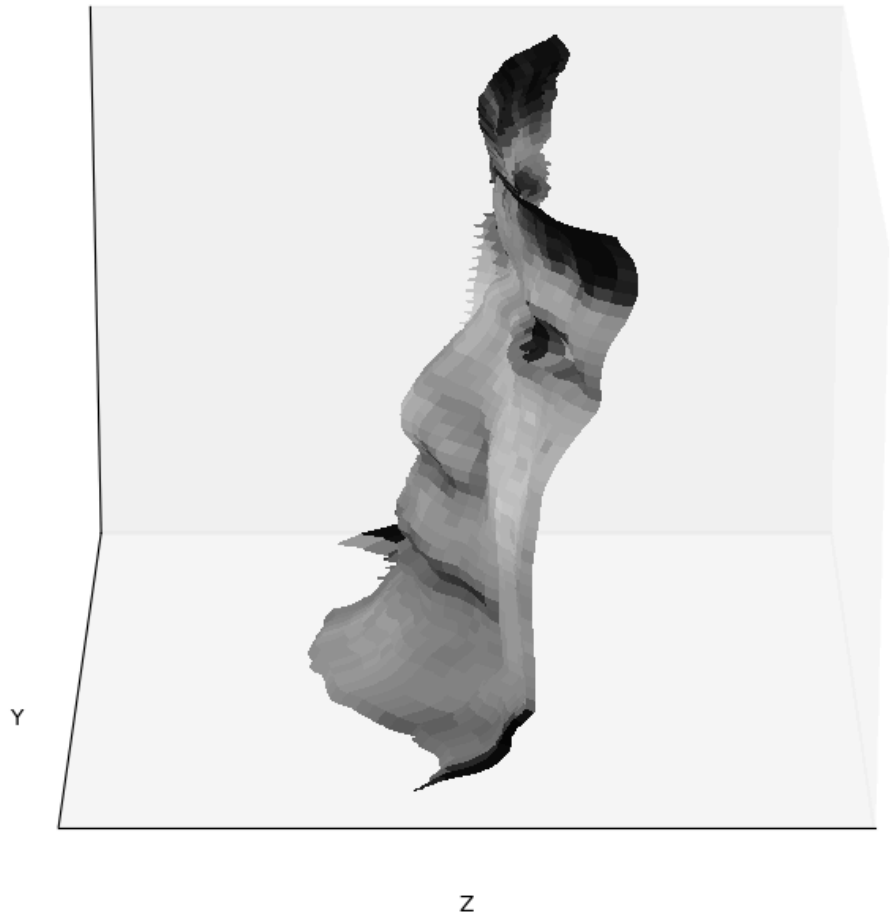
Obviously, the random method is a lot slow because we are looping over every pixel and taking 4 iterations for each pixel. This is slow. The averaging method is a bit slower than the row or column method because it is the average the two methods.

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.
It violates the slides because faces have complex reflection patterns and thus are Non-Lambertian surfaces. Also there aren't constant shadows because facial features and faces aren’t quite smooth. Faces have wrinkles, pores, etc. Also, faces aren’t flat geometry because the have varying depth and thus varying shadows.

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.
Since the subject 'yaleB05' looks weird above. We will try to improve it. But using a subset of images doesn’t help. See below for a subset of 32 images:


Z
Y



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

I tried using a subset of images from 30 to 63, but there isn't a big difference. As you can see in the above image, a subset of 32 is actually worse. I could find some, like a subset of 37 images, that slightly improved the chin from using all 64 images. But there isn't a big enough difference.

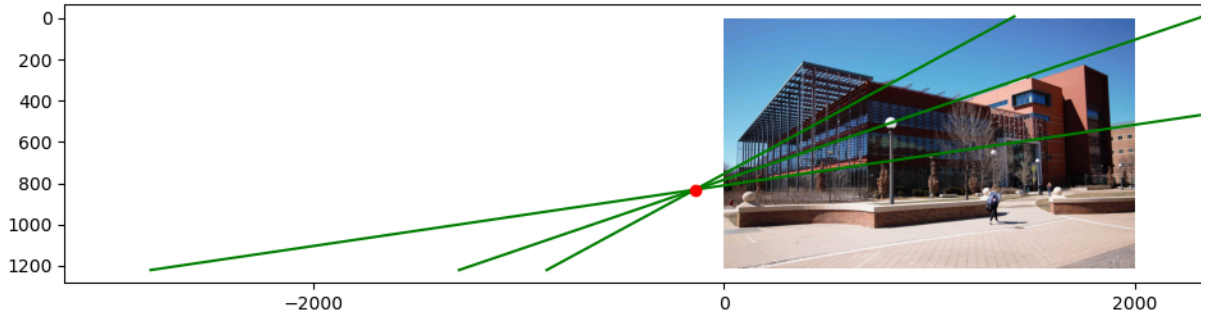
Part 1: Extra Credit (optional)

Clearly mark every item you're submitting for extra credit. Don't forget to include discussion of what you did, results, and any references. Refer to the assignment for suggested outputs.

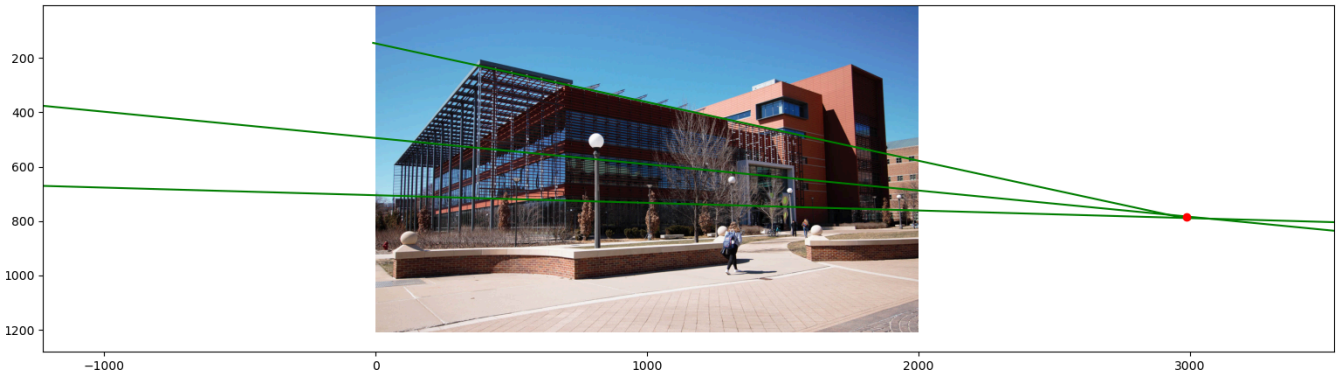
Part 2 Single-View Geometry:

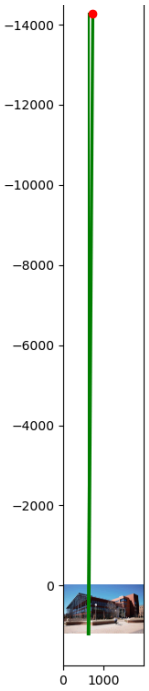
1. Estimate the three major vanishing points.

    a. Plot the VPs and the lines used to estimate them on the image plane using the provided code.




0
200
400
600
800
1000
1200
0
250
500
750
1000
1250
1500
1750


200
400
600
800
1000
1200
−1000
0
1000
2000
3000



b. Specify the VP pixel coordinates. Discuss how you estimated them.

VP 0 : (-142.5, 830.6, 1)

VP 1: (2988.8, 785.3, 1)

VP 2: (739.8, -14268.3, 1)

c. Plot the ground horizon line and specify its parameters in the form a * x + b * y + c = 0. Normalize the parameters so that: a^2 + b^2 = 1.

Horizontal line = [0.01, 0.1, -828]

horizon line normalized (horizon_line[0])**2 + (horizon_line[1])**2 = 0.99999999

2. Using the fact that the vanishing directions are orthogonal, solve for the focal length and optical center (principal point) of the camera. Show all your work.

focal length=1487.8, principal point = (955.8, 666.5)

```python
def get_camera_parameters(vpts):
    """
    Computes the camera parameters. Hint: The SymPy package is suitable for this.
    """
    vpt0 = vpts[:, 0][:, np.newaxis]
    vpt1 = vpts[:, 1][:, np.newaxis]
    vpt2 = vpts[:, 2][:, np.newaxis]

    #focal length and principal point
    focal_len, x_p, y_p= symbols('focal_len, x_p, y_p')
    CAM_MAT_T = Matrix([[1/focal_len, 0, 0], [0, 1/focal_len, 0], [-x_p/focal_len, -y_p/focal_len, 1]])
    CAM_MAT = Matrix([[1/focal_len, 0, -x_p/focal_len], [0, 1/focal_len, -y_p/focal_len], [0, 0, 1]])

    eq1 = vpt0.T * CAM_MAT_T * CAM_MAT * vpt1
    eq2 = vpt0.T * CAM_MAT_T * CAM_MAT * vpt2
    eq3 = vpt1.T * CAM_MAT_T * CAM_MAT * vpt2

    focal_len, x_p, y_p = solve([eq1[0], eq2[0], eq3[0]], (focal_len, x_p, y_p))[0]

    return abs(focal_len), x_p, y_p
```

3. Compute the rotation matrix for the camera.

```
Rotation matrix =
[[ 0.80609914 -0.0143874  -0.59160559]
 [ 0.04710769 -0.99497192  0.0883841 ]
 [ 0.58990257  0.09911552  0.80136825]]
```

4. Estimate the following heights and show all the lines and measurements used to perform the calculation. As a reference measurement, assume that the person in the picture is 5ft 6in tall.

a. The left side of the ECE building.

Height of left side = 69.4 feet

b. The right side of the ECE building.

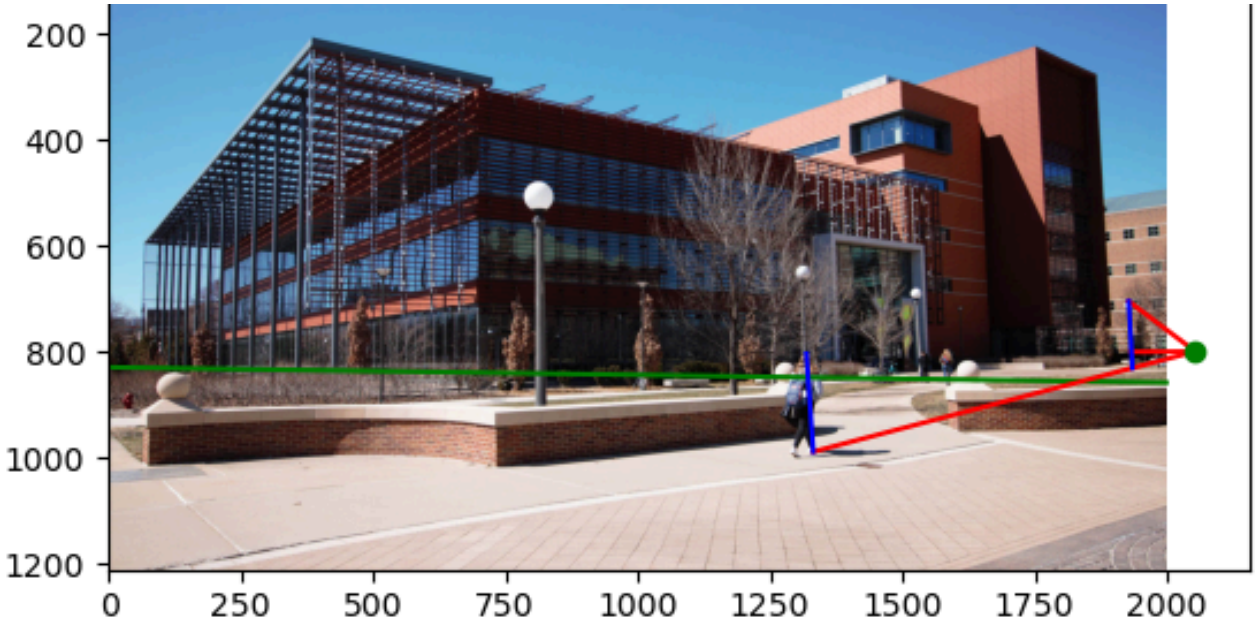
Height of right side = 245.2 feet

c. The lamp posts.

Door lamp post

Height of door lamp post = 20.99 feet

Right lamp post



Height of right lamp post = 22.89 feet

Average height of the lamp posts = 21.94 feet

5. Recompute the answers in a-c above assuming that the person is 6ft tall.

Height of left side = 75.7 feet

Height of right side = 267.5 feet

Height of door lamp post = 22.90 feet

Height of right lamp post = 24.97 feet

Average height of the lamp posts = 23.94

Part 2 Extra Credit (optional)

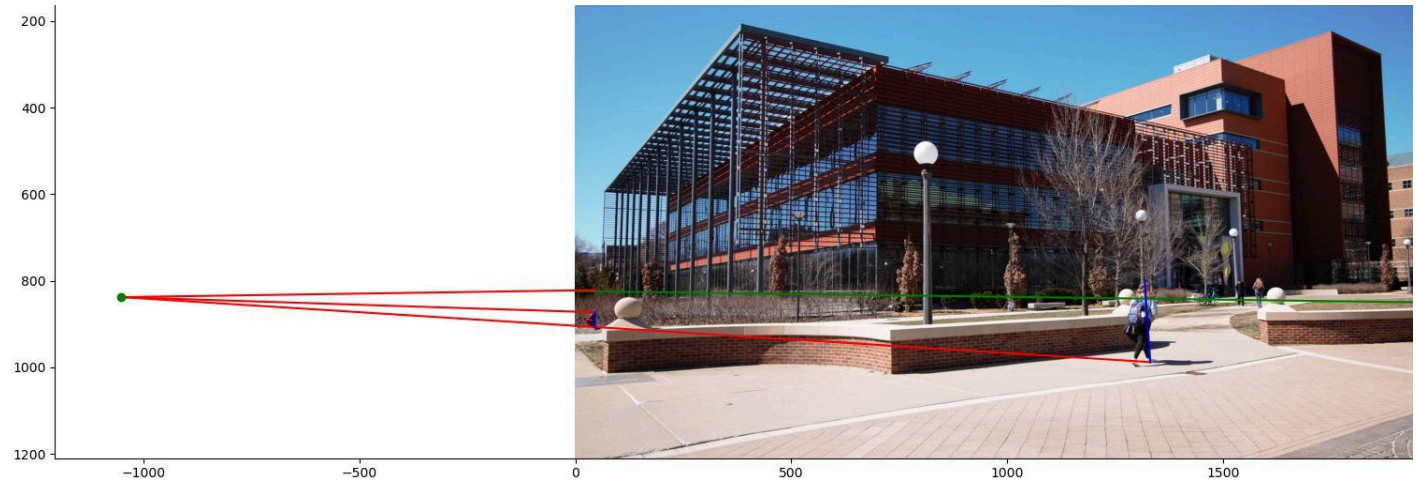
Clearly mark every item you're submitting for extra credit. Don't forget to include discussion of what you did, results, and any references. Refer to the assignment for suggested outputs.

Part 2 Extra Credit 1:

Perform additional measurements on the image, such as other parts of the ECE building, windows, trees, etc. Show all your work.

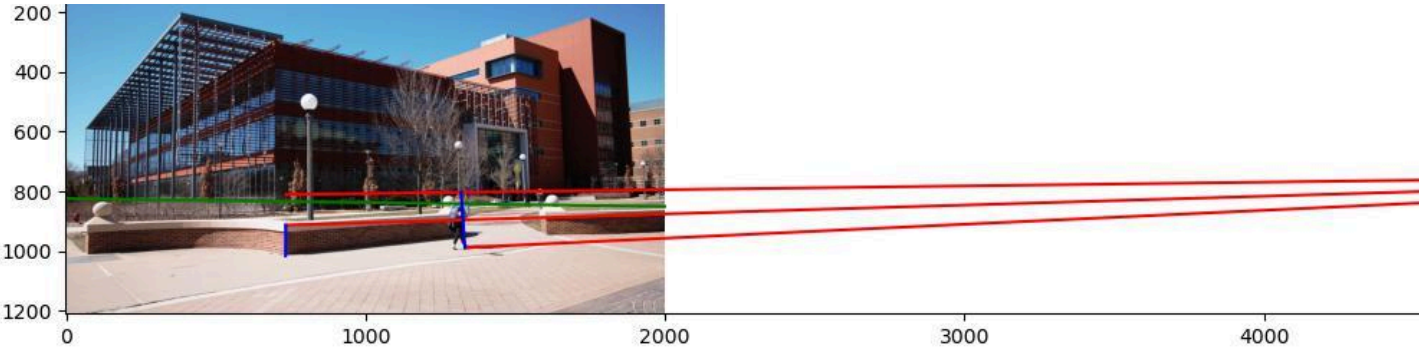
Height of Sculpture = 25.41 feet

Height of Fire hydrant = 2.16 feet



Height of Fur tree = 23.15 feet

Height of Wall = 2.66 feet



All of the heights seem fairly reasonable. Below is the pixel coordinates used for the objects:

```
# coordinates from image (x1 top and X2 bottom)
objects_EC = ('person', 'Sculpture', 'Fire hydrant', 'Fur tree', 'Wall')
coords_EC = dict()

x1, x2, y1, y2 = 1319, 1332, 803, 988
coords_EC['person'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1508, 1512, 697, 834
coords_EC['Sculpture'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 46, 45, 873, 907
coords_EC['Fire hydrant'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 1876, 1883, 713, 826
coords_EC['Fur tree'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))

x1, x2, y1, y2 = 734, 733, 913, 1016
coords_EC['Wall'] = np.array(([[x1, x2], [y1, y2], [1, 1]]))
```

Then we applied the height estimation using the 5.5 foot person:

```
# 5.5 foot person
print("Extra credit")
heights = dict()
for obj in objects_EC[1:]:
    print('Estimating height of %s' % obj)
    height = estimate_height(coords_EC, obj, coords_EC['person'], horizon_line, vpts, im, person_height=5.5)
    heights[obj] = height
    print(f"Height of {obj} = {height} feet")
```

```
def estimate_height(coords, obj, person_coords, horizon_line, vpts, im,
person_height):
    """
    Estimates height for a specific object using the recorded coordinates.
You might need to plot additional images here for
    your report.
    """
    horizon_line = horizon_line/np.linalg.norm([horizon_line[0],
horizon_line[1]])

    person = person_coords
    person_top = person[:,0]
    person_bottom = person[:,1]
```

```
    object = coords[obj]
    object_top = object[:,0]
    object_bottom = object[:,1]

    bottom_line = np.cross(person_bottom, object_bottom)

    vanishing_point = np.cross(bottom_line, horizon_line)
    vanishing_point = vanishing_point/vanishing_point[-1]

    object_line = np.cross(object_bottom, object_top)
    person_vanish = np.cross(person_top, vanishing_point)
    target_point = np.cross(person_vanish, object_line)
    target_point = target_point/target_point[-1]

    infinite_vpt = vpts[:,2]
    p1_p3 = np.linalg.norm(object_bottom-object_top)
    p2_p4 = np.linalg.norm(infinite_vpt-target_point)
    p3_p4 = np.linalg.norm(object_top-infinite_vpt)
    p1_p2 = np.linalg.norm(object_bottom-target_point)
    ratio = p1_p3*p2_p4 / (p1_p2*p3_p4)

    plt.figure()
    plt.imshow(im)
    col = im.shape[1]
    x_array = np.arange(0, col, 1)
    y_array = horizon_line[0]*x_array+horizon_line[2] / (-horizon_line[1])
    plt.plot(x_array, y_array, 'g')
    plt.plot([vanishing_point[0], person_bottom[0]], [vanishing_point[1],
person_bottom[1]], 'r')
    plt.plot([vanishing_point[0], target_point[0]], [vanishing_point[1],
target_point[1]], 'r')
    plt.plot([vanishing_point[0], object_top[0]], [vanishing_point[1],
object_top[1]], 'r')
    plt.plot([person_top[0], person_bottom[0]], [person_top[1],
person_bottom[1]], 'b')
    plt.plot([object_bottom[0], object_top[0]], [object_bottom[1],
object_top[1]], 'b')
    plt.plot(vanishing_point[0], vanishing_point[1], 'go')
    plt.show()
```

```
    obj_height = ratio * person_height
    return obj_height
```