Homework 6: Photometric Stereo

Srividya Gandikota
sgandiko
sgandiko@andrew.cmu.edu

1 Calibrated Photometric Stereo

1.1 In your write-up, explain the geometry of the n-dot-l lighting model from Fig. 2a. Where does the dot product come from? Where does projected area (Fig. 2b) come into the equation? Why does the viewing direction not matter?

- If we are analyzing Figure 2a from the homework document, we see that the geometry of the n dot l lighting model is a light beam reflected off a surface and bounces off a surface at an angle. The surface radiance equation is $L = \frac{\rho d}{\pi} I cos\theta_i = \frac{\rho d}{\pi} I \overrightarrow{n} \cdot \overrightarrow{l}$ where n is the surface normal, l is the light direction, and I is the intensity of light.
- The dot product comes from the previously mentioned surface normal n and direction of light l. By finding the dot product between these two, we can find a value that is very similar to the cosine of the angle between the two vectors. This can also be seen through $n \cdot l = |n| \times |l| \times \cos(\theta)$.
- The projected area comes into the equation from the effective area of the surface from the perspective of the light source. We can see that the actual area A = projected area P $* \cos(\theta)$
- The viewing direction doesn't matter because the amount of irradiance that is seen by the viewer is the same from every single direction due to the fact that the angle gets changed every time the viewing angle. This is due to the cosine value accounting for the incident light's effectiveness based on the impact angle.

1.2 Simulate the appearance of the sphere under the n-dot-l model with directional light sources with incoming lighting directions (1, 1, 1)/sqrt(3), (1, -1, 1)/sqrt(3) and (-1, -1, 1)/ sqrt(3) in the function renderNDotLSphere (individual images for all three lighting directions). Note that your rendering isn't required to be absolutely radiometrically accurate: we need only evaluate the n-dot-l model. Include the 3 renderings in your writeup.

The implementation of the renderNDotLSphere function can be found in Figure 1 below.

```
# Calculate center coordinates for camera
c_x = res[0] / 2
c_y = res[1] / 2

# Find meshgrid and x/y in space
x, y = np.meshgrid(np.arange(res[0]), np.arange(res[1]))

x = pxSize * (x - c_x) + center[0]
y = pxSize * (y - c_y) + center[1]

# Find z for sphere
z = rad ** 2 - x ** 2 - y ** 2
mask = z < 0
z[mask] = 0.0

# Compute dot product with light direction
pts = np.stack((x, y, np.sqrt(z)), axis = 2).reshape((-1, 3))
pts = pts / np.linalg.norm(pts, axis = 1)[:, np.newaxis]
image = np.dot(pts, light).reshape(res[::-1])

# print(image.shape)
image[mask] = 0.0
return image
```

Figure 1: renderNDotLSphere Function

- $(1, 1, 1)/\sqrt{3}$

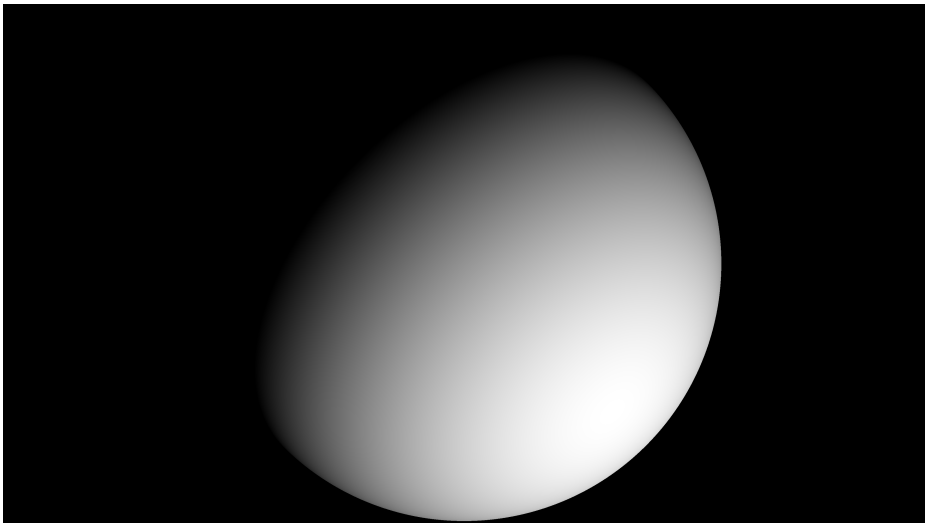


Figure 2: $(1, 1, 1)/\sqrt{3}$

- $(1, -1, 1)/\sqrt{3}$

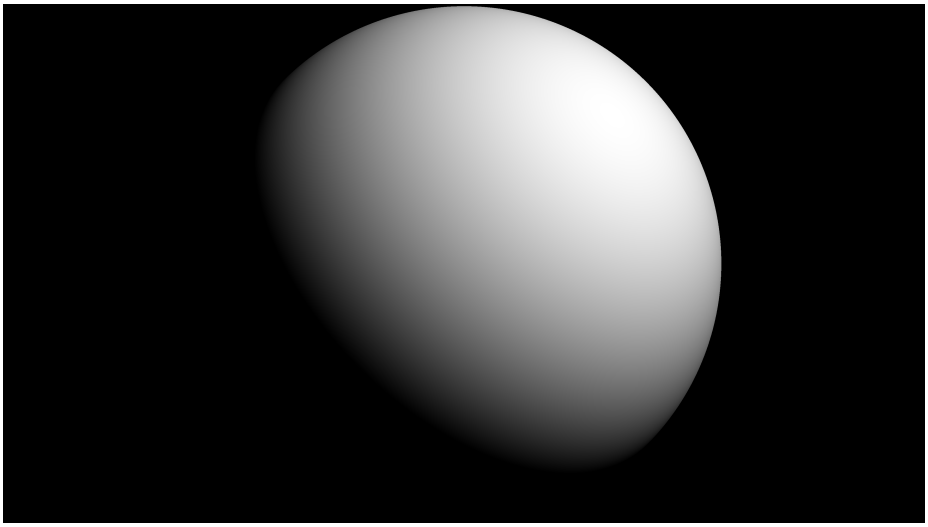


Figure 3: $(1, -1, 1)/\sqrt{3}$

- $(-1, -1, 1)/\sqrt{3}$

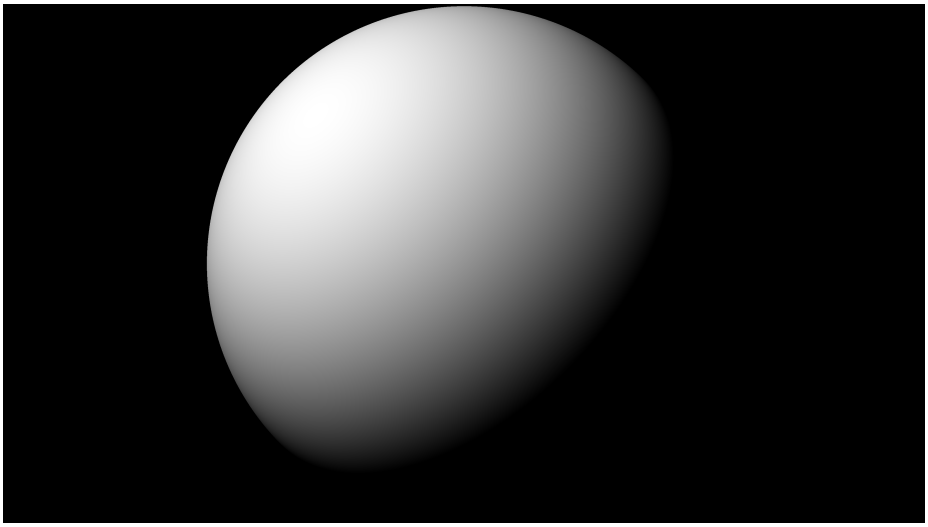


Figure 4: $(-1, -1, 1)/\sqrt{3}$

1.3 In the function loadData, read the images into Python. Convert the RGB images into the XYZ color space and extract the luminance channel. Vec- torize these luminance images and stack them in a $7 \times P$ matrix, where P is the number of pixels in each image. This is the matrix I, which is given to us by the camera. Next, load the sources file and convert it to a 3×7 matrix L. For this question, include a screenshot of your function loadData.

The implementation of the loadData function can be found in Figure 5 below.

```
I = []
L = None
P = 0

# Iterate through 7 images
for i in range(7):
    # Generate path for each image
    img_path = path + f"input_{i+1}.tif"
    gray_img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Store the pixel values in the I list
    I.append(gray_img.flatten())

    if P == 0:
        P = gray_img.size

# Convert to array, load, and get dimensions
I = np.array(I)
L = np.load(path + "sources.npy").T
s = gray_img.shape

return I, L, s
```

Figure 5: loadData Function

1.4 With this model, explain why the rank of I should be 3. Perform a singular value decomposition of I and report the singular values in your write-up. Do the singular values agree with the rank-3 requirement? Explain this result.

- The reason that the rank of I should be 3 is that we are operating in a 3d coordinate space so we require 3 different light source from various directions to find them. Due to these three different lights, we see that the intensity matrix is expected to have a rank of 3.
- After performing singular value decomposition on the intensity matrix I, the resulting singular values were found to be [66066.69 7845.778 5478.131 1666.1006 1265.8109 1000.85767 815.4014].
- The problem with this is that we see that the rank is 7 and not 3 as expected. This is due to the fact that there is alot of noise or blurring in the images. One suggestion to fix this would be to photograph the image from many different light directions in order to help reduce the noise.

1.5 Solve this linear system in the function estimatePseudonormalsCalibrated. Estimate per- pixel albedos and normals from this matrix in estimateAlbedosNormals. In your write-up, mention how you constructed the matrix A and the vector y.

The implementation of the two functions function can be found in Figure 6 below.

```
def estimatePseudonormalsCalibrated(I, L):

    L_new = np.linalg.inv(np.dot(L, L.T))
    B = L_new.dot(L).dot(I)
    return B


def estimateAlbedosNormals(B):

    albedos = np.linalg.norm(B, axis=0)
    normals = B / (albedos + 0.000001)
    return albedos, normals
```

Figure 6: Two Estimate Function Implementation

The construction of matrix A and vector y was much easier than expected. Given that we start with Ax=y, we can solve for this by using x = A^{-1}y. Since we know that x = B, A = L^T, and y = I, we can simplify the previous equation to B = $(L^T)^{-1}I$. One problem that we encounter here is that we cannot solve $(L^T)^{-1}I$ due to the lack of L being a square matrix so what needs to be done is turn $(L^T)^{-1}$ into $(LL^T)^{-1}L$. Therefore, we find B = $(LL^T)^{-1}LI$ where A = LL^T, and y = LI.

1.6 Calculate the albedos, reshape them into the original size of the images and display the resulting image in the function displayAlbedosNormals. Include the image in your write-up and comment on any unusual or unnatural features you may find in the albedo image, and on why they might be happening. Make sure to display in the gray colormap.

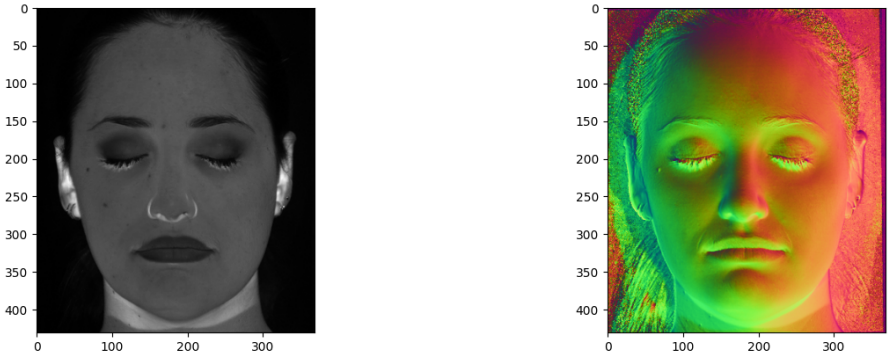
The implementation of the display function can be found in Figure 7 below. What we can observe from the

```
albedo = albedos/np.max(albedos)
albedoIm = np.reshape(albedo, s)

normal = (normals+1.0) / 2.0
normalIm = np.reshape((normal.T, (s[0], s[1], 3)))

return albedoIm, normalIm
```

Figure 7: Display Function Implementation



(a) Albedo Image (b) Normal Image

Figure 8: Albedos and Normals

albedo image is that it struggled with tight, shadowed components just as the ears, neck and bottom of the nose. This may be due to the fact that the model assumes that we have one light source with directional lighting which means if there are components that are shadowed, we lose the reconstruction there. On the other hand, the normals does seem to match between the left and right side of the image. Similar to the albedo, if there are parts in shadows, we will struggle to reconstruct normals.

1.7 Explain, in your write-up, why n is related to the partial derivatives of f at (x, y).

Given that the surface is F(x, y, z), it also is equal to z - f(x, y) = 0. Let the normal at point (x, y) be n = (n_1, n_2, n_3).

$$\frac{\partial F(x,y,z)}{\partial x} = 0, \frac{\partial F(x,y,z)}{\partial y} = 0, \text{ and } \frac{\partial F(x,y,z)}{\partial z} = 0$$

$$\downarrow$$

$$N = (\frac{-\partial f(x,y)}{\partial x}, \frac{-\partial f(x,y)}{\partial y}, 1)$$

$$\downarrow$$

$$N = (-\frac{n_1}{n_3}, -\frac{n_2}{n_3}, 1)$$

1.8 Given that g(0, 0) = 1, perform these two procedures: 1. Use gx to construct the first row of g, then use gy to construct the rest of g; 2. Use gy to construct the first column of g, then use gx to construct the rest of g. Are these the same?

If we set $g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ and $g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$, when we reconstruct matrix g using the two different methods, we see that both methods will result in g $= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$. A few ways to modify the gradients that we calculated above to make g_x and g_y non integrable are to using non consistant patterns, discontinuities, or non uniformities.The gradients estimates in the way previously would not be non integrable if the surfaces are not smooth and include discontinuities mentioned before.

- Method 1:

  Start with $g_x = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$ and if we use g_y to reconstruct the rest, we get g $= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$.

- Method 2:

  Start with $g_y = \begin{bmatrix} 1 \\ 5 \\ 9 \\ 13 \end{bmatrix}$ and if we use g_x to reconstruct the rest, we get g $= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$.

1.9 Write a function estimateShape to apply the Frankot- Chellappa algorithm to your estimated normals. Once you have the function f (x, y), plot it as a surface in the function plotSurface and include some significant viewpoints in your write-up.

The implementation of the estimateShape function can be found in Figure 9 below. Along with that, the surface plot can be found in Figure 10 as well.

```python
def estimateShape(normals, s):

    x, y = (normals[:2] / (-normals[2] + 0.000001)).reshape((2, *s))
    surface = integrateFrankot(x, y)
    return -surface


def plotSurface(surface):

    fig = plt.figure()
    X, Y = np.meshgrid(np.arange(surface.shape[1]), np.arange(surface.shape[0]))

    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, surface, edgecolor='none', cmap = matplotlib.cm.coolwarm)
    ax.set_title('Surface Plot')

    plt.show()
```

Figure 9: estimateShape Function Implementation

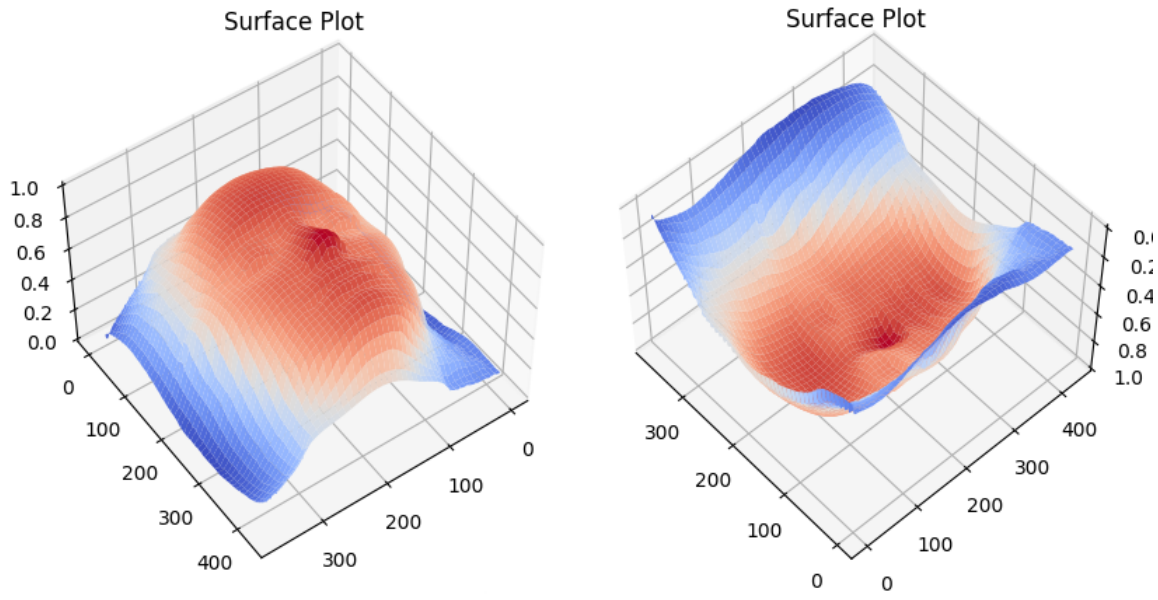


Figure 10: Surface Plot

2 Uncalibrated Photometric Stereo

2.1 It is well-known that the best rank-k approximation to a m × n matrix M, where k ¡= minm, n is calculated as the following: perform a singular value decomposition, set all singular values except the top k from sigma to 0 to get the matrix, and reconstitute. Explain in your write-up how this can be used to construct a factorization of the form detailed above following the required constraints.

Given that we start with I = $L^T B$, when we apply singular value decomposition to I, we get a result of $I = U\Sigma V^T$. This can be used to factorize if we utilize matrix dimensions for simplification. Matrix U is a 7 x 7 while matrix V is of size P x P and I has a dimension of 7 x P. Theoretically, we can also reduce the rank by setting all the singular values from Σ to 0 except the top k values. Similarly the u and v matrices would become matrices with just the first three columns of U and V.

2.2 With your method, estimate the pseudonor- mals ˆB in estimatePseudonormalsUncalibrated and visualize the resultant albedos and normals in the gray and rainbow colormaps respectively. Include these in the write-up.

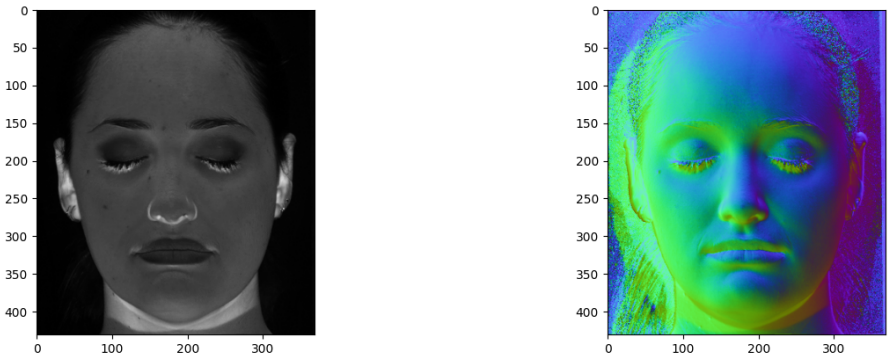
The implementation of the estimatePsuedonormalUncalibrated function can be found in Figure 11 below.

```
def estimatePseudonormalsUncalibrated(I):

    u, _, v = np.linalg.svd(I, full_matrices=False)
    B = v[0:3, :]
    L = u[0:3, :]

    return B, L
```

Figure 11: estimate Function Implementation



(a) Albedo Image

(b) Normal Image

Figure 12: Albedos and Normals

2.3 Describe a simple change to the procedure in (a) that changes the ˆL and ˆB, but keeps the images rendered using them the same using only the matrices you calculated during the singular value decomposition (U/S/V). (No need to code anything for this part, just describe the change in the write-up)

- The ground truth lighting directions were found to be:

$$\begin{bmatrix} -0.1418 & 0.1215 & -0.0690.067 & -0.1627 & 0. & 0.1478 \\ -0.1804 & -0.2026 & -0.0345 - 0.0402 & 0.122 & 0.1194 & 0.1209 \\ -0.9267 & -0.9717 & -0.838 - 0.9772 & -0.979 & -0.9648 & -0.9713 \end{bmatrix}$$

- The estimated lighting directions were found to be:

$$\begin{bmatrix} -0.3556 & 0.3548 & 0.6277 & -0.5442 & -0.1553 & -0.1479 & -0.1065 \\ -0.3983 & -0.6283 & 0.3991 & 0.4232 & -0.3140 & -0.0144 & -0.0953 \\ -0.3255 & 0.178 & 0.1064 & 0.2528 & 0.28774 & -0.1618 & 0.8232 \end{bmatrix}$$

Based on these two matrices, we can see that the ground truth lighting and the estimated lighting are actually quite different. One way that we can chance the procedure to chage L and B but keeps images rendered would be to implement a normalization approach. This can be achieved by normalizing L and B and then multiplying the new B matrix with $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mu & \nu & \lambda \end{bmatrix}$.

2.4 Use the given implementation of the Frankot-Chellappa algorithm from the previous question to reconstruct a 3D depth map and visualize it as a surface in the 'coolwarm' colormap as in the previous question. Does this look like a face?

The implementation of the Frankot Chellapa algorithm can be found in Figure 13 below. Along with that, the surface plot can be found in Figure 14 as well. However, the surface plot we see does not resemble a face at all, but rather looks like a frequency of depth map of sorts.

```
# Q2.4
surface = estimateShape(normals, s)
surface = (surface - np.min(surface)) / (np.max(surface) - np.min(surface))
plotSurface(surface)
```

Figure 13: Surface Plot Code

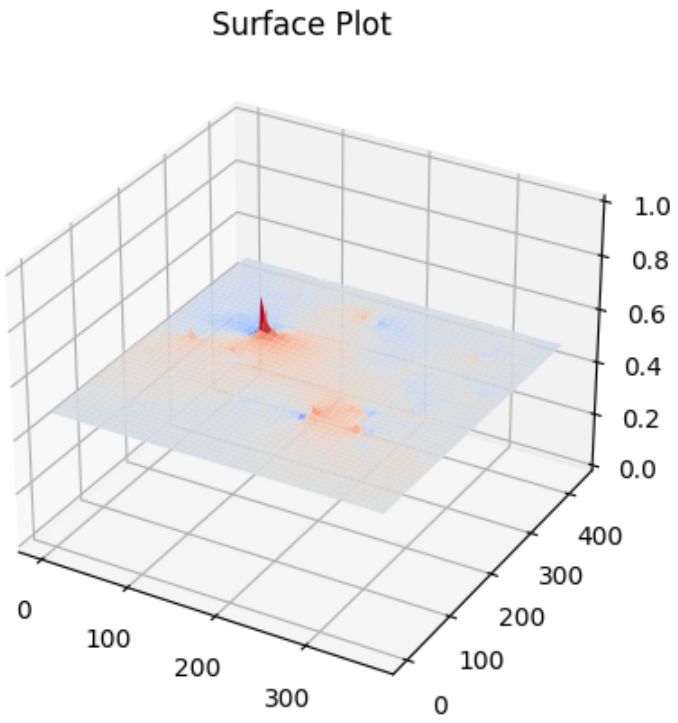


Figure 14: Surface Plot

2.5 Input your pseudonormals into the enforceIntegrability function, use the output pseudonormals to estimate the shape with the Frankot-Chellappa algorithm and plot a surface as in the previous questions. Does this surface look like the one output by calibrated photometric stereo? Include at least three viewpoints of the surface and your answers in your write-up.

The implementation of the enforceIntegrability algorithm can be found in Figure 15 below. Along with that, the surface plot in three different viewpoints can be found in Figure 16 as well. What we notice about the surface is that it actually looks like the face output from the stereo.

```
# Q2.5
albedos, normals = estimateAlbedosNormals(B)
normals = enforceIntegrability(normals, s)
surface = estimateShape(normals, s)
surface = (surface - np.min(surface)) / (np.max(surface) - np.min(surface))
plotSurface(-surface)
```

Figure 15: Surface Plot Code

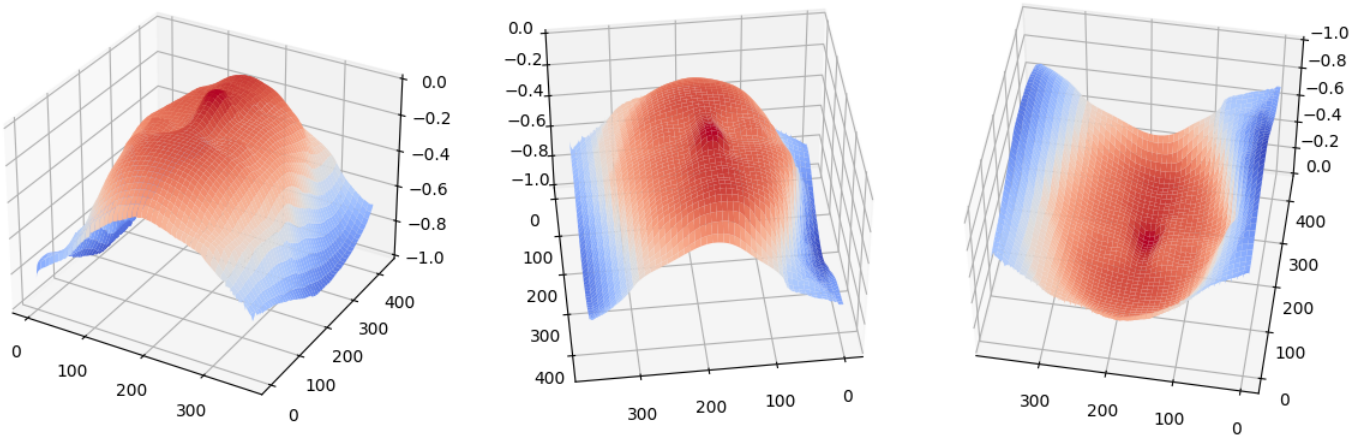


Figure 16: Surface Plot in Three Different Viewpoints

2.6 Vary the parameters u, v and sigma in the bas-relief transforma- tion and visualize the corresponding surfaces. Include at least six (two with each parameter varied) of the significant ones in your write-up. Looking at these, what is your guess for why the basrelief ambiguity is so named? In your writeup, describe how the three parameters affect the surface.

- Varying the parameter μ seems to do the inverse of the lambda where the gradient changes as it gets smaller.

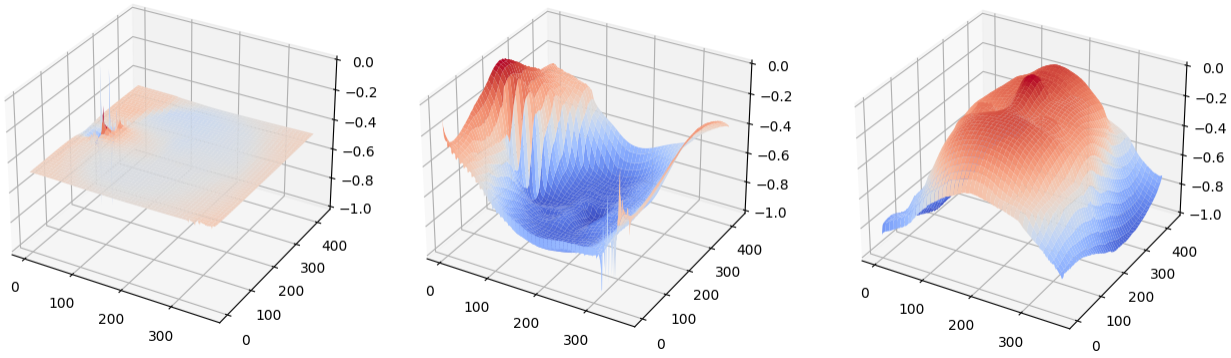


Figure 17: Surface Plot for μ = -1, 0, 1

- Varying the parameter ν seems to manipulate the gradient. As the nu vlau eincreases, the gradient also increases so we see that the face stretches out.

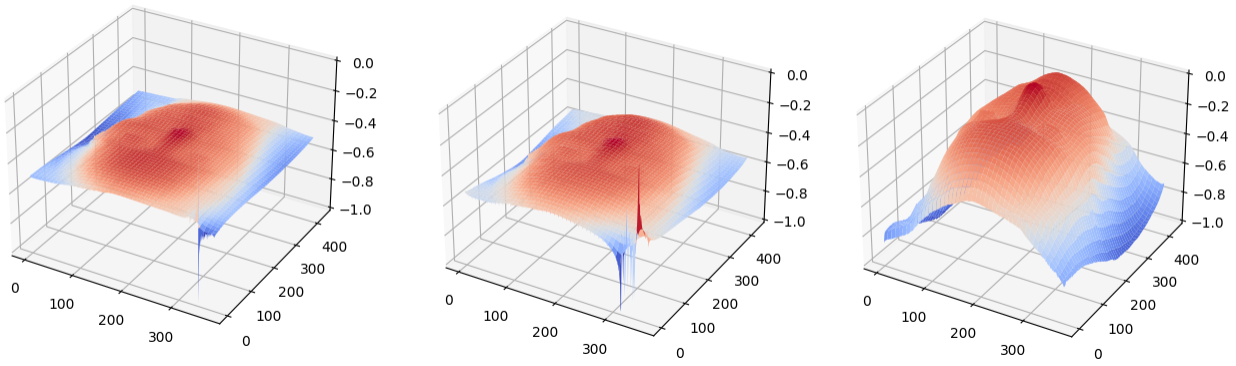


Figure 18: Surface Plot for ν = 0.1, 1, 2

- Varying the parameter λ seems to make the impact of the gradient change. As the value of lambda increases, we notice that the gradient disappears more.

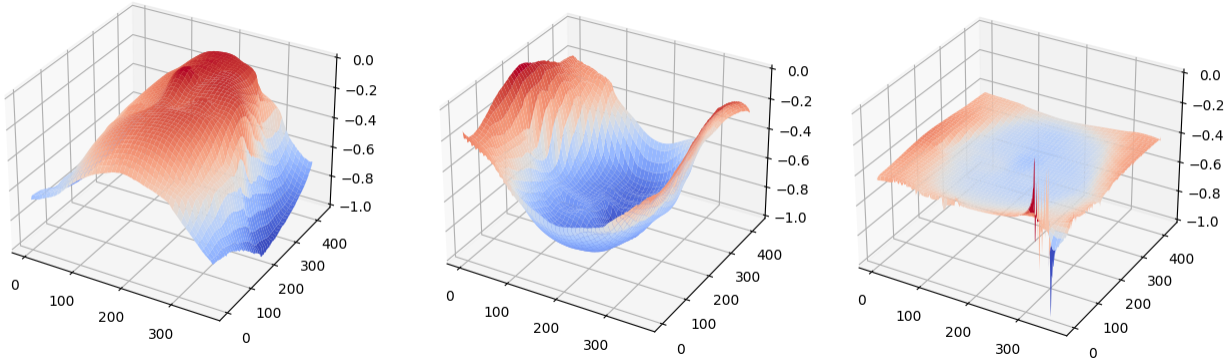


Figure 19: Surface Plot for λ = -1, 0.5, 3

2.7 With the bas-relief ambiguity, in Eq. 2, how would you go about designing a transformation that makes the estimated surface as flat as possible?

Given the bas-relief ambiguity, the best transformation, in my opinion, to make the estimate surface as flat as possible would be to choose a fairly large λ and set μ and ν to 0.

2.8 We solved the problem with 7 pictures of the face. Will acquiring more pictures from more lighting directions help resolve the ambiguity?

If we were to add more pictures with more lighting directions, it should help resolve some of the ambiguity. This is because we are increasing the amount of information present by adding more images and more lighting. Along with that, the larger set of images should make matrix I more comprehensive and reduce the ambiguity. However, one main drawback to consider is that the computational complexity will increase as the larger dataset will require more resources.