16-820 Advanced Computer Vision: Homework 5 (Fall 2024) Photometric Stereo

Ryan Wu (Andrew ID: weihuanw)

Wednesday November 20, 2024

Q1.a Understanding n-dot-l lighting.
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?

The $n \cdot l$ lighting model, also known as the Lambertian reflectance model, captures the behavior of light reflected from a rough surface. Here, $\vec{l}$ represents the light source vector, $\vec{n}$ represents the surface normal vector, and $\vec{v}$ represents the viewing direction vector. The term dA represents the projected area (or the differential area).

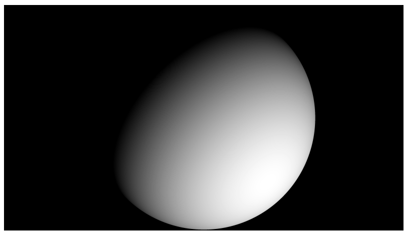
The dot product arises from the Lambert's Cosine Law, where the amount of light that a surface receives is directly proportional to the angle between the surface normal n and the light direction l. This can be modeled as:

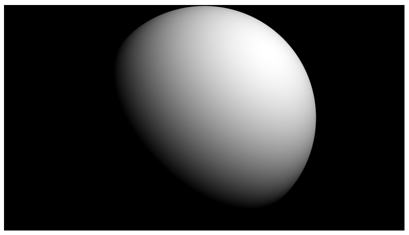
$$n \cdot l = \|\vec{n}\| \cdot \|\vec{l}\| \cdot \cos\theta$$

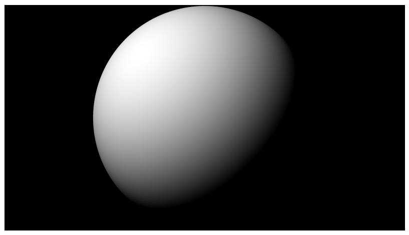
The projected area dA directly correlates with the $cos\theta$ angle term in the equation. The projected area dA receives more light energy when this angle is small and vice versa.

The viewing direction does not matter because the Lambertian reflectance mode distributes light uniformly in all directions. The surface's brightness is determined only by the angle between the surface normal and the incoming light source, not by the direction from which the observer is viewing the surface.

Q1.b Rendering n-dot-l lighting.
Consider a uniform fully reflective Lambertian sphere with its center at the origin and a radius of 0.75 cm (Fig. 3). An orthographic camera located at (0, 0, 10) cm looks towards the negative z-axis with its sensor axes aligned and centered on the x- and y-axes. The pixels on the camera are squares 7 μm in size, and the resolution of the camera is 3840 $\times$ 2160 pixels. Simulate the appearance of the sphere under the $n \cdot l$ 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 $n \cdot l$ model. Include the 3 renderings in your write-up. Include a snippet of the code you wrote for this part in your write-up.


(a) $(1, 1, 1)/\sqrt{3}$.


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


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

```python
def renderNDotLSphere(center, rad, light, pxSize, res):
    # Meshgrid for camera frame
    [X, Y] = np.meshgrid(np.arange(res[0]), np.arange(res[1]))
    X = (X - res[0] / 2) * pxSize * 1.0e-4
    Y = (Y - res[1] / 2) * pxSize * 1.0e-4

    # Calculate Z using the equation of a sphere
    Z = np.sqrt(rad**2 + 0j - X**2 - Y**2)
    X[np.real(Z) == 0] = 0
    Y[np.real(Z) == 0] = 0
    Z = np.real(Z)

    # Initialize image array
    image = np.zeros((res[1], res[0]))

    # Normalize the light vector
    light = light / np.linalg.norm(light)

    # Loop over each pixel
    for i in range(res[1]):
        for j in range(res[0]):
            # Omit the pixels that are outside the sphere
            if Z[i, j] > 0:
                # Calculate the normal at the point
                normal = np.array([X[i, j], Y[i, j], Z[i, j]])
                normal = normal / np.linalg.norm(normal)

                # Calculate the intensity of the pixel
                intensity = np.dot(normal, light)
                image[i, j] = max(0, intensity)

    return image
```

Q1.c Loading data.
In the function `loadData`, read the images into Python. Convert the RGB images into the XYZ color space and extract the luminance channel. Vectorize 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`.

```python
def loadData(path="../data/"):
    # Initialize variables
    s = None
    luminance_channel = []

    # Load the lighting source from sources.npy (3 x 7)
    L = np.load(path + "sources.npy").T

    # Loop over each image
    for i in range(1, 8):
        # Load the image
        image = plt.imread(path + "input_" + str(i) + ".tif")
        image = image.astype(np.uint16)  # 16-bit image

        # Convert the image to XYZ color space to get the luminance
        image = rgb2xyz(image)
        luminance = image[:, :, 1]

        # Vectorize the luminance values and store them in I
        luminance_channel.append(luminance.flatten())

        # Get the shape of the image
        if s is None:
            s = luminance.shape

    # Convert the list to a numpy array (7 x P)
    I = np.vstack(luminance_channel)

    # Print the shapes
    print('Matrix I:', I.shape)
    print('Matrix L:', L.shape)
    print('Image shape:', s)

    return I, L, s
```

Q1.d Initials.
Recall that in general, we also need to consider the reflectance, or albedo, of the surface we're reconstructing. We find it convenient to group the normals and albedos (both of which are a property of only the surface) into a pseudonormal b $= a \cdot$ n, where a is the scalar albedo. We then have the relation I $=$ L^{T}B, where the $3 \times P$ matrix B is the set of pseudonormals in the images. 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 and include the code snippet in the write-up.

From the relation $\mathbf{I} = \mathbf{L}^T\mathbf{B}$, matrix $\mathbf{L}^T$ has a dimension of 7×3 and a rank of at most 3, matrix $\mathbf{B}$ has a dimension of $3 \times P$ and also a rank of at most 3. In theory, the resulting matrix I should have a dimension of $7 \times P$ and a rank of at most 3. The rank 3 constraint reflects the 3-dimensional nature of the problem, where the intensity is determined by the 3D lighting direction and 3D surface properties.

After performing the singular value decomposition of $\mathbf{I}$, the singular values are:

$$\begin{bmatrix} 0.07576378 & 0.00906763 & 0.00635114 & 0.00194115 & 0.00146786 & 0.00115865 & 0.00094721 \end{bmatrix}$$

We can obverse that the first 3 singular values are significantly larger than the latter 4. It means that most of the meaningful information are captured by the first 3 dimensions, which is, to a degree, consistent with the rank 3 constraint. However, the singular values having a rank of 7 does not totally agree with the rank-3 requirement. This can be due to noise in a real-world dataset.

```python
# Part 1(d)
    U, S, Vt = np.linalg.svd(I, full_matrices=False)
    print('Singular Values of matrix I:', S)
```

Q1.e Estimating pseudonormals.
Since we have more measurements (7 per pixel) than variables (3 per pixel), we will estimate the pseudonormals in a least-squares sense. Note that there is a linear relation between I and B through L: therefore, we can write a linear system of the form Ax = y out of the relation $\mathbf{I} = \mathbf{L}^T\mathbf{B}$ and solve it to get B. Solve this linear system in the function `estimatePseudonormalsCalibrated`. In your write-up, mention how you constructed the matrix A and the vector y along with the code for the function you wrote.

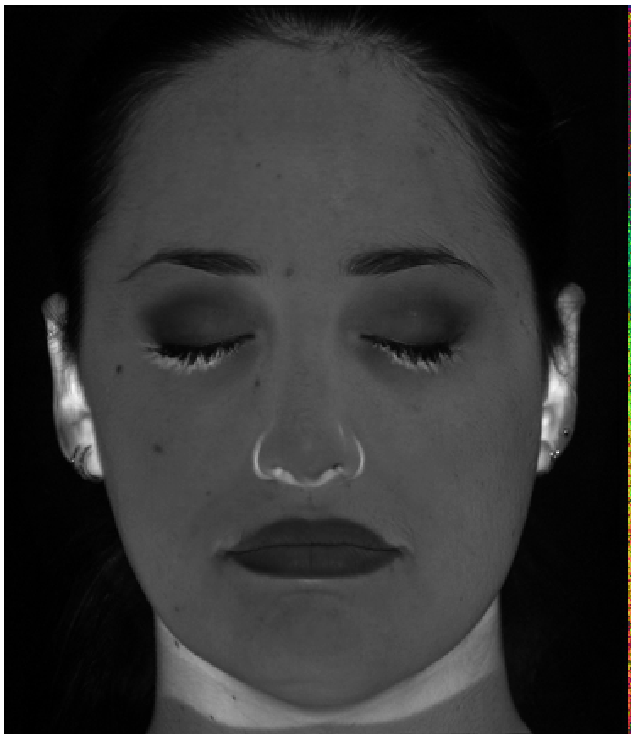
With the given relation of $\mathbf{A}x = \mathbf{y}$, $\mathbf{x}$ can solved by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$. In a least-squares formulation, $\mathbf{x} = (A^TA)^{-1}\mathbf{A}^T\mathbf{y}$. Similarly, with the relation $\mathbf{I} = \mathbf{L}^T\mathbf{B}$, $\mathbf{B}$ can solved by $\mathbf{B} = (LL^T)^{-1}\mathbf{LI}$ in a least-squares formulation. In our case, (LL^T) represents matrix A and $\mathbf{LI}$ represents vector y.

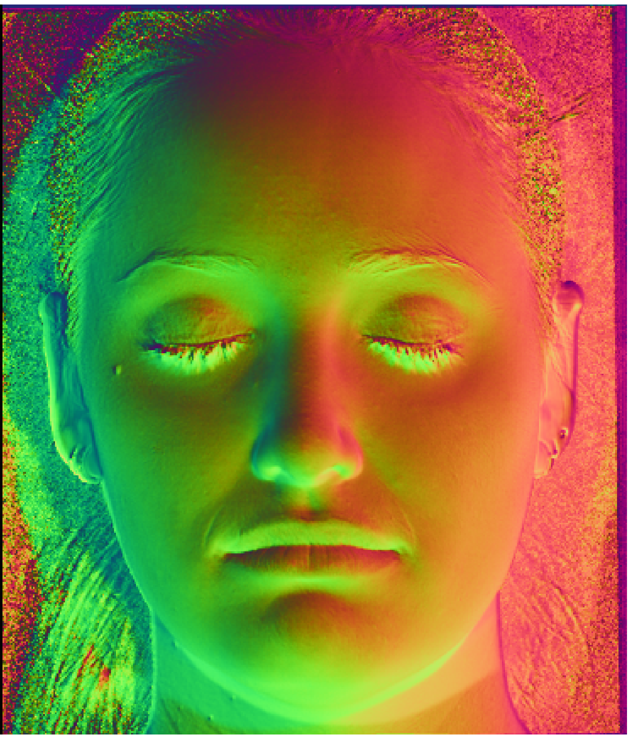
```python
def estimatePseudonormalsCalibrated(I, L):
    # Calculate the pseudonormals using the least squares method
    B = np.linalg.lstsq(L.T, I, rcond=None)[0]

    return B
```

Q1.f Albedos and normals.
Estimate per-pixel albedos and normals from matrix B in `estimateAlbedosNormals`. Note that the albedos are the magnitudes of the pseudonormals by definition. 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 per-pixel normals can be viewed as an RGB image. Reshape the estimated normals into an image with 3 channels and display it in the function `displayAlbedoNormals`. Note that the components of these normals will have values in the range $[-1, 1]$. You will need to rescale them so that they lie in $[0, 1]$ to display them properly as RGB images. Include this image in the write-up. Do the normals match your expectation of the curvature of the face? Make sure to display in the rainbow colormap. Your results should look like those in Fig. 4. Include the code snippets.

From the visualization below, we can observe some unusual and unnatural features around the nose, ears, and neck areas with light-gray features in the Albedo Image (a). The cause of these unusual features can be due to occlusions, shadow effects, or noise in the dataset. On the other hand, the normals image (b) matches my expectation of the curvature of the face.


(a) Albedo image.


(b) Normals image.

```python
def estimateAlbedosNormals(B):
    # Calculate the albedos
    albedos = np.linalg.norm(B, axis=0)

    # Normalize the pseudonormals
    normals = B / albedos

    return albedos, normals
```

```python
def displayAlbedosNormals(albedos, normals, s):
    # Reshape the albedos and normals
    albedoIm = albedos.reshape(s)
    normalIm = normals.T.reshape(s[0], s[1], 3)

    # Normalize the pseudonormals
    normalIm = (normalIm + 1) / 2

    return albedoIm, normalIm
```

Q1.g Normals and depth.
We will now estimate from the normals the actual shape of the face. Represent the shape of the face as a 3D depth map given by a function $z = f(x, y)$. Let the normal at the point (x, y) be $\mathbf{n} = (n_1, n_2, n_3)$. Explain, in your write-up, why $\mathbf{n}$ is related to the partial derivatives of f at (x, y): $f_x = \frac{\partial f(x,y)}{\partial x} = -\frac{n_1}{n_3}$ and $f_y = \frac{\partial f(x,y)}{\partial y} = -\frac{n_2}{n_3}$. You may consider the 1D case where $z = f(x)$.

Given function $z = f(x, y)$ and $\mathbf{n} = (n_1, n_2, n_3)$, we can represent the 3D surface as:

$$\mathbf{r}(x, y) = (x, y, f(x, y)),$$

where

$$\frac{\partial \mathbf{r}}{\partial x} = \left(1, 0, \frac{\partial f}{\partial x}\right) = (1, 0, f_x), \quad \frac{\partial \mathbf{r}}{\partial y} = \left(0, 1, \frac{\partial f}{\partial y}\right) = (0, 1, f_y).$$

Moreover,

$$\mathbf{n} = \frac{\partial \mathbf{r}}{\partial x} \times \frac{\partial \mathbf{r}}{\partial y} = (-f_x, -f_y, 1).$$

The normal function can be further represented as:

$$\mathbf{n} = (n_1, n_2, n_3) = \frac{\mathbf{n}}{\|\mathbf{n}\|} = \frac{1}{\sqrt{f_x^2 + f_y^2 + 1}}(-f_x, -f_y, 1),$$

where

$$n_1 = -\frac{f_x}{\sqrt{f_x^2 + f_y^2 + 1}}, \quad n_2 = -\frac{f_y}{\sqrt{f_x^2 + f_y^2 + 1}}, \quad n_3 = \frac{1}{\sqrt{f_x^2 + f_y^2 + 1}}.$$

Finally, we can proof that:

$$f_x = \frac{\partial f(x, y)}{\partial x} = -\frac{n_1}{n_3}, \quad f_y = \frac{\partial f(x, y)}{\partial y} = -\frac{n_2}{n_3}.$$

Q1.h Understanding integrability of gradients.
Consider the 2D, discrete function g on space given by the matrix below. Find its x and y gradient, given that gradients are calculated as $g_x(x_i, y_j) = g(x_{i+1}, y_j) - g(x_i, y_j)$ for all i, j (and similarly for y). Let us define $(0,0)$ as the top left, with x going in the horizontal direction and y in the vertical.

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Note that we can reconstruct the entirety of g given the values at its boundaries using g_x and g_y. Given that $g(0,0) = 1$, perform these two procedures:

1. **Use g_x to construct the first row of g, then use g_y to construct the rest of g.**
2. **Use g_y to construct the first column of g, then use g_x to construct the rest of g.**

Are these the same?

Note that these were two ways to reconstruct g from its gradients. Given arbitrary g_x and g_y, these two procedures will not give the same answer, and therefore this pair of gradients does not correspond to a true surface. Integrability implies that the value of g estimated in both these ways (or any other way you can think of) is the same. How can we modify the gradients you calculated above to make g_x and g_y non-integrable? Why may the gradients estimated in the way of (g) be non-integrable? Note all this down in your write-up.

With the given g matrix, we can define g_x and g_y as:

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}.$$

Following procedure 1, we get g matrix as:

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}.$$

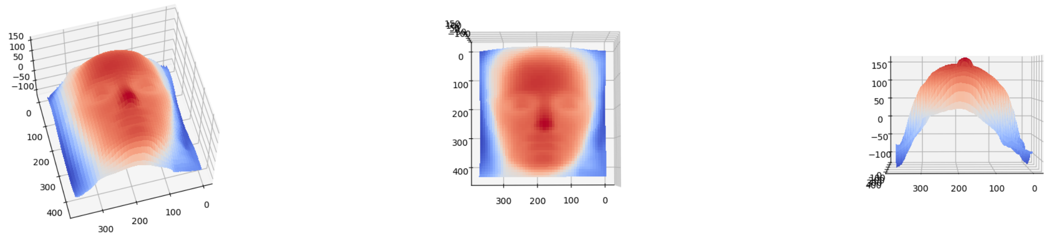
Following procedure 2, we get g matrix as:

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}.$$

We can observe that both procedures produced the same g matrix.

We can modify the gradients by introducing non-uniform differences between the rows and columns to make make g_x and g_y non-integrable. The gradients estimated may contain noise in real world datasets, which can lead to non-integrable (g).

Q1.i Shape estimation.
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 3D projection from mpl toolkits.mplot3D.Axes3D along with the function `plot_surface` might be of help. Make sure to plot this in the `coolwarm` colormap. The result we expect of you is shown in Fig.1. Include a snippet of the code you wrote for this part in your write-up.



3D reconstruction results from calibrated photometric stereo.

```python

def estimateShape(normals, s):
    # Initialize the surface
    surface = np.zeros(s)

    # Calculate the gradients of the normals
    zx = np.reshape(normals[0, :] / -normals[2, :], s)
    zy = np.reshape(normals[1, :] / -normals[2, :], s)

    # Apply the Frankot-Chellappa algorithm
    surface = integrateFrankot(zx, zy)

    return surface
```

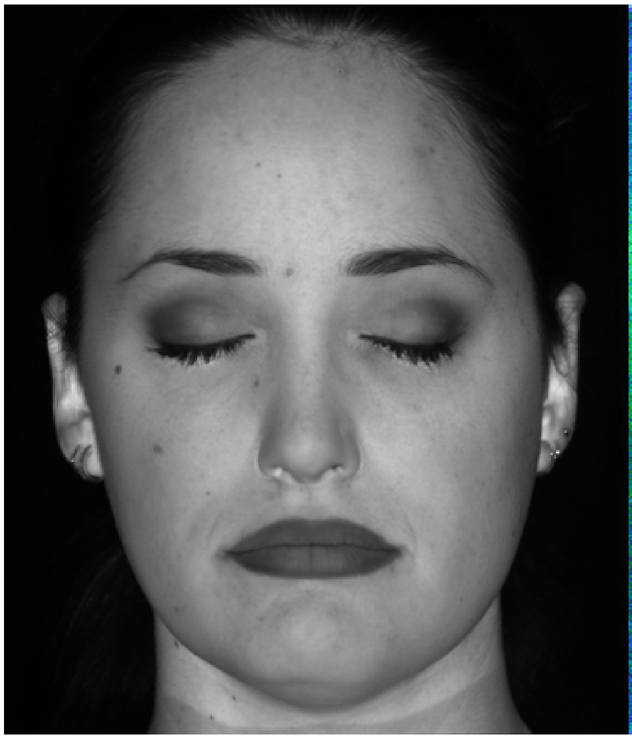
Q2.a Uncalibrated normal estimation.
Recall the relation $\mathbf{I} = \mathbf{L}^T\mathbf{B}$. Here, we know neither $\mathbf{L}$ nor $\mathbf{B}$. Therefore, this is a matrix factorization problem with the constraint that with the estimated $\hat{\mathbf{L}}$ and $\hat{\mathbf{B}}$, the rank of $\hat{\mathbf{I}} = \hat{\mathbf{L}}^T\hat{\mathbf{B}}$ be 3 (as you answered in the previous question), and the estimated $\hat{\mathbf{I}}$ and $\hat{\mathbf{L}}$ have appropriate dimensions.

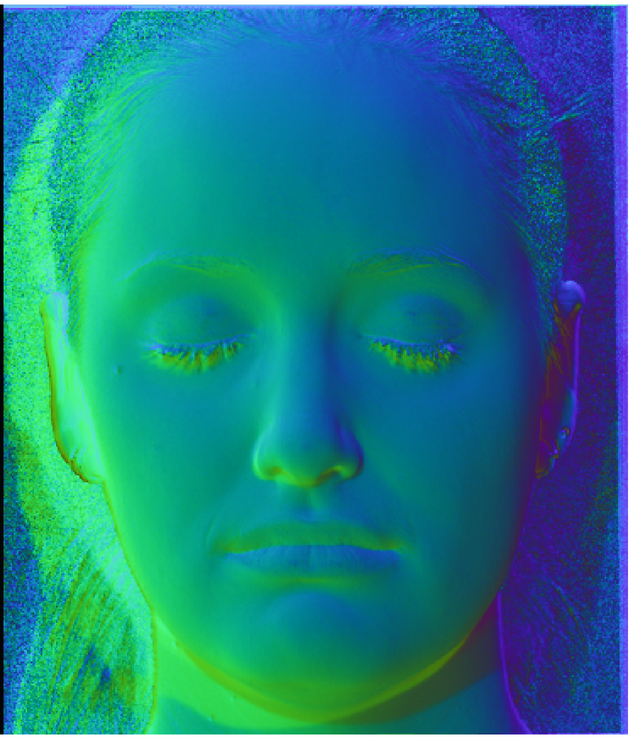
It is well-known that the best rank-k approximation to an $m \times n$ matrix $\mathbf{M}$, where $k \leq \min\{m, n\}$, is calculated as follows: perform a singular value decomposition (SVD) $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, set all singular values except the top k from $\mathbf{\Sigma}$ to 0 to get the matrix $\hat{\mathbf{\Sigma}}$, and reconstitute $\hat{\mathbf{M}} = \mathbf{U}\hat{\mathbf{\Sigma}}\mathbf{V}^T$. Explain in your write-up how this can be used to construct a factorization of the form detailed above following the required constraints.

To construct a factorization $\hat{\mathbf{I}} = \hat{\mathbf{L}}^T\hat{\mathbf{B}}$ with rank 3 constraint, we should perform the following steps:

1. Perform singular value decomposition on $\mathbf{I} = \mathbf{L}^T\mathbf{B}$.
2. Set all singular values except the top 3 from $\mathbf{\Sigma}$ to 0 to get the matrix $\hat{\mathbf{\Sigma}_3}$ as:
$$\hat{\mathbf{\Sigma}}_3 = \text{diag}(\sigma_1, \sigma_2, \sigma_3, 0, \ldots, 0).$$
3. Construct:
$$\hat{\mathbf{L}}^T = \mathbf{U}_3\hat{\mathbf{\Sigma}}_3^{1/2}, \quad \hat{\mathbf{B}} = \hat{\mathbf{\Sigma}}_3^{1/2}\mathbf{V}_3^T,$$
where:
   - $\mathbf{U}_3$ is the first 3 columns of $\mathbf{U}$,
   - $\hat{\mathbf{\Sigma}}_3^{1/2}$ is the square root of the top 3 singular values,
   - $\mathbf{V}_3^T$ is the first 3 rows of $\mathbf{V}^T$.

Q2.b Calculation and visualization.
With your method, estimate the pseudonormals $\hat{B}$ in `estimatePseudonormalsUncalibrated`, include a snippet of this function in your write-up and visualize the resultant albedos and normals in the gray and rainbow colormaps respectively. Include these too in the write-up. A sample result has been shown in Fig. 6.


(a) Albedo image.


(b) Normals image.

```python
def estimatePseudonormalsUncalibrated(I):
    # Perfrom SVD on I
    U, S, Vt = np.linalg.svd(I, full_matrices=False)

    # Truncate the top 3 singular values
    S3 = np.diag(S[:3])      # first 3 singular values
    U3 = U[:, :3]            # first 3 columns of U
    Vt3 = Vt[:3, :]          # first 3 rows of Vt

    # Calculate lighting directions (L) and pseudonormals (B)
    L = U3 @ np.sqrt(S3)
    B = np.sqrt(S3) @ Vt3  # B: 3 x P

    return B, L
```

Q2.c Comparing to ground truth lighting.
In your write-up, compare the $\hat{\mathbf{L}}$ estimated by the factorization above to the ground truth lighting directions given in $\mathbf{L}_0$. Are they similar? Unless a special choice of factorization is made, they will be different. Describe a simple change to the procedure in (a) that changes the $\hat{\mathbf{L}}$ and $\hat{\mathbf{B}}$, 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 $\mathbf{L}_0$ are:

$$\begin{bmatrix} -0.1418 & 0.1215 & -0.0690 & 0.0670 & -0.1627 & 0.0000 & 0.1478 \\ -0.1804 & -0.2026 & -0.0345 & -0.0402 & 0.1220 & 0.1194 & 0.1209 \\ -0.9267 & -0.9717 & -0.8380 & -0.9772 & -0.9790 & -0.9648 & -0.9713 \end{bmatrix}$$

The estimated lighting directions $\hat{\mathbf{L}}$ are:

$$\begin{bmatrix} -0.09775742 & 0.03365684 & 0.05015493 \\ -0.10976828 & -0.05996671 & 0.03168933 \\ -0.08944811 & 0.01702700 & 0.00845597 \\ -0.11062144 & -0.01870367 & -0.00109931 \\ -0.10833356 & 0.05369181 & -0.01652167 \\ -0.10587512 & 0.00834930 & -0.02641608 \\ -0.10475887 & -0.02732348 & -0.04228383 \end{bmatrix}$$

We can observe that they are not similar. A simple change to the procedure in (a) would be to introduce an invertible transformation matrix $\mathbf{R}$, such as a rotation matrix. For example:

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix $\mathbf{R}$ will modify $\hat{\mathbf{L}}$ and $\hat{\mathbf{B}}$ as:

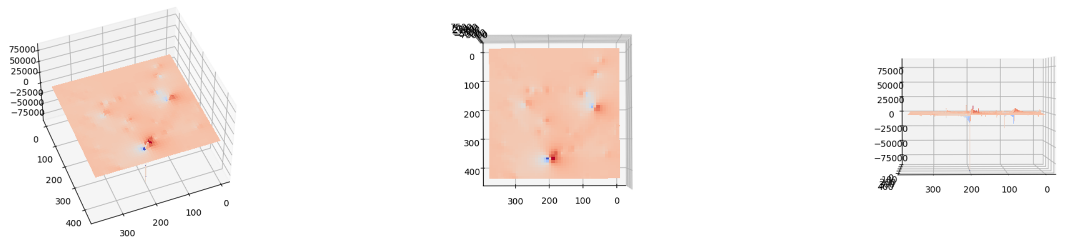
$$\hat{\mathbf{L}}' = \mathbf{R}\hat{\mathbf{L}}, \quad \hat{\mathbf{B}}' = \mathbf{R}^{-1}\hat{\mathbf{B}},$$

while preserving the rendered images $\hat{\mathbf{I}}$:

$$\hat{\mathbf{I}} = (\hat{\mathbf{L}}')^T(\hat{\mathbf{B}}') = \hat{\mathbf{L}}^T\hat{\mathbf{B}}.$$

Q2.d Reconstructing the shape, attempt 1.
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?

From the visualization below, we can observe that it does not look like a face.

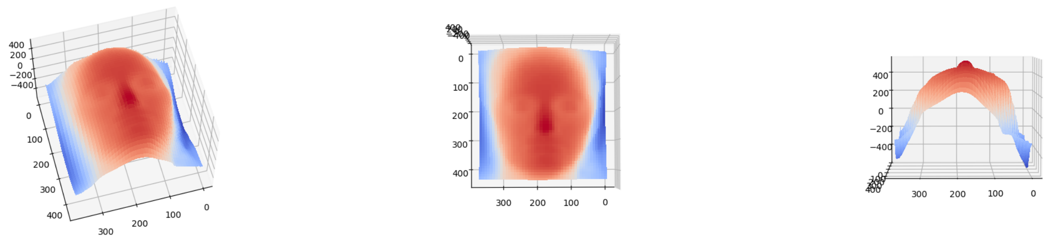


3D reconstruction results from uncalibrated photometric stereo (attempt 1).

Q2.e Reconstructing the shape, attempt 2.

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 (results shown in Fig. 5). 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.

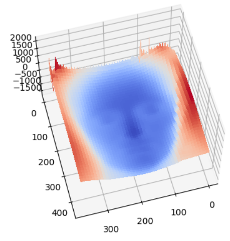
From the visualization below, we can observe that it does look like the one output by calibrated photometric stereo.

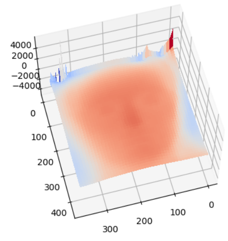


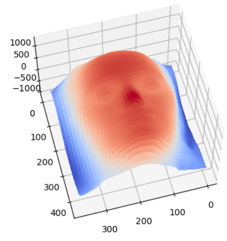
3D reconstruction results from uncalibrated photometric stereo (attempt 2).

Q2.f Why low relief?
Vary the parameters μ, ν, and λ in the bas-relief transformation 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 bas-relief ambiguity is so named? In your write-up, describe how the three parameters affect the surface.

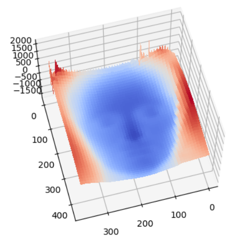
By varying μ with (0.5, 1, and 5) while keeping ν at 0.5 and λ at 1, we can observe that the surface tilts more along the x-axis and even inverts as the μ value increase.

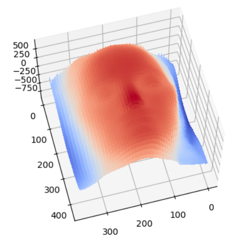

$\mu = 0.5$, $\nu = 0.5$, and $\lambda = 1$.

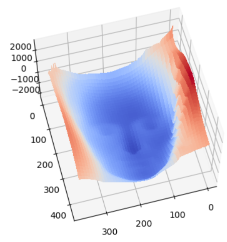

$\mu = 1$, $\nu = 0.5$, and $\lambda = 1$.


$\mu = 5$, $\nu = 0.5$, and $\lambda = 1$.

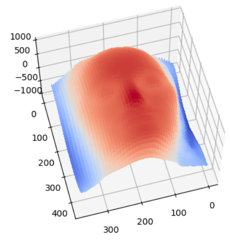
By varying ν with (0.5, 1, and 5) while keeping μ at 0.5 and λ at 1, we can observe that the surface tilts more along the y-axis and even inverts as the ν value approach to 1.

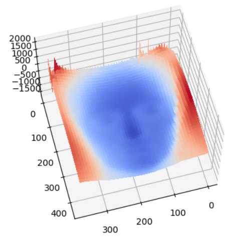

$\mu = 0.5$, $\nu = 0.5$, and $\lambda = 1$.

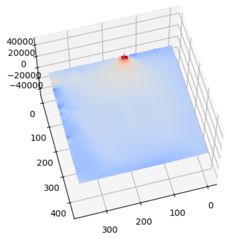

$\mu = 0.5$, $\nu = 1$, and $\lambda = 1$.


$\mu = 0.5$, $\nu = 5$, and $\lambda = 1$.

By varying λ with (0.5, 1, and 10) while keeping μ at 0.5 and ν at 0.5, we can observe that surface gets more and more extruded with more stretched face features as the λ value increase.


$\mu = 0.5$, $\nu = 0.5$, and $\lambda = 0.5$.

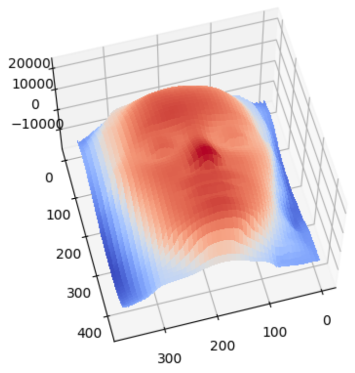

$\mu = 0.5$, $\nu = 0.5$, and $\lambda = 1$.


$\mu = 0.5$, $\nu = 0.5$, and $\lambda = 10$.

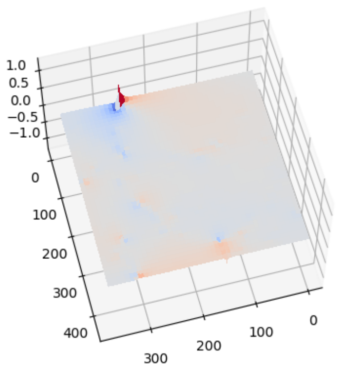
From the question's name hint, I believe that the bas-relief ambiguity is named because of the relationship it resembles with low-relief sculpturing technique (sculpting objects on a flat surface to create a 3D appearance).

Q2.g Flattest surface possible.
With the bas-relief ambiguity, in Eq.2, how would you design a transformation that makes the estimated surface as flat as possible?

With the bas-relief ambiguity, I would set both μ and ν to 0, and λ to a small value to estimate the surface as flat as possible.



Smallest λ (0.009) that still shows face features.



Even smaller λ (0.0001) that shows flatter surface.

Q2.h More measurements.
We solved the problem with 7 pictures of the face. Will acquiring more pictures from more lighting directions help resolve the ambiguity?

Acquiring more pictures from more lighting directions will not help resolve the ambiguity issue.