CMU Fall24 16820 Homework 5

Patrick Chen

November 10, 2024

Q1-a at page 3

Ans:

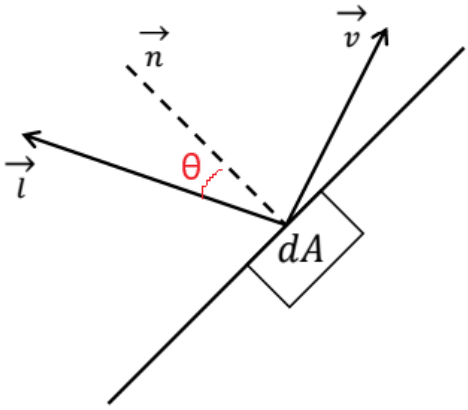
According to Lambertian's cosine law, the intensity of light observed from a Lambertian surface is propational to the cosine of the angle θ between the surface normal, $\vec{n}$, and the direction of the incoming light, $\vec{l}$.

$$\overrightarrow{n} \cdot \overrightarrow{l} = |\overrightarrow{n}||\overrightarrow{l}|\cos(\theta) \tag{1}$$

Typically in the context of n-dot-l lighting, both $\vec{n}$ and $\vec{l}$ are unit vectors, so we have:

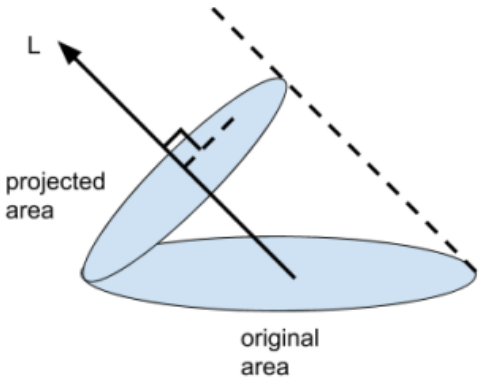
$$\overrightarrow{n} \cdot \overrightarrow{l} = \cos(\theta) \tag{2}$$

As shown in Fig. 2a as below, the θ is the dot product between $\vec{n}$ and $\vec{l}$. The dot product, θ, quantifies the amount of incident light that is effectively contributing to the brightness of the surface. As the angle increases, less light is effectively reaching the surface, as the angle decrease, more light hits the surface, reaching maximum incoming light when θ is 0, identical to the case that $\vec{l}$ is aligned with $\vec{n}$.



(a) Geometry of photometric stereo

Fig. 2a



(b) Projected area

Fig. 2b

The projected area, $\mathbf{dA_{projected}}$, in Fig. 2b is the original area, $\mathbf{dA}$, times $\cos(\theta)$:

$$\mathbf{dA_{projected}} = \mathbf{dA} \cdot \cos(\theta) \tag{3}$$

$$= \mathbf{dA} \cdot \overrightarrow{n} \cdot \overrightarrow{l} \tag{4}$$

$$\frac{\mathbf{dA_{projected}}}{\mathbf{dA}} = \cos(\theta) \tag{5}$$

$$= \overrightarrow{n} \cdot \overrightarrow{l} \tag{6}$$

As θ becomes larger, the effective area that light directly expose to becomes smaller, and on the opposite, as θ becomes smaller, the effective area becomes larger, reaching maximum $\mathbf{dA}$ as $\theta = 0$. Hence, the projected area comes into the equation as part of the Lambertian reflectance model through the $\cos(\theta)$ factor.

The reason that the viewing direction does not matter is that we assume the surface is Lambertial surface, which reflects light equally in all directions, $\vec{v}$. Specifically, the light reflected from each point on the surface has the same intensity value no matter where the observer is located. It also means that the intensity of light that the observer can observe is based solely on the $\vec{n} \cdot \vec{l} = \cos(\theta)$ term.

Q1-b at page 4

Ans:

Following Figure 3 - Figure 5 shows the rendered result, and the Figure 6 shows the code snippet in q1.py.

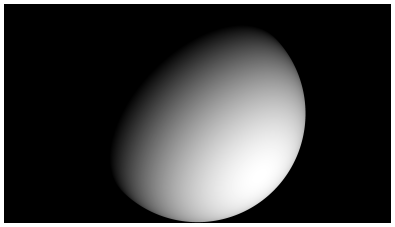


Figure 3: Rendered Result with $(1, 1, 1)/\sqrt{3}$ Light Vector

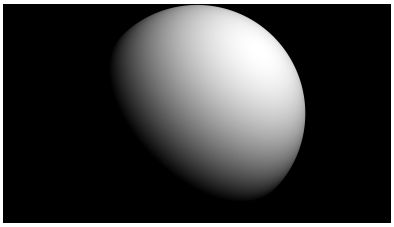


Figure 4: Rendered Result with $(1, -1, 1)/\sqrt{3}$ Light Vector

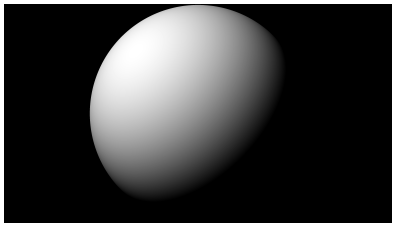


Figure 5: Rendered Result with $(-1, -1, 1)/\sqrt{3}$ Light Vector

```python
def renderNDotLSphere(center, rad, light, pxSize, res):
    """
    Question 1 (b)

    Render a hemispherical bowl with a given center and radius. Assume that
    the hollow end of the bowl faces in the positive z direction, and the
    camera looks towards the hollow end in the negative z direction. The
    camera's sensor axes are aligned with the x- and y-axes.

    Parameters
    ----------
    center : numpy.ndarray
        The center of the hemispherical bowl in an array of size (3,)

    rad : float
        The radius of the bowl

    light : numpy.ndarray
        The direction of incoming light

    pxSize : float
        Pixel size

    res : numpy.ndarray
        The resolution of the camera frame

    Returns
    -------
    image : numpy.ndarray
        The rendered image of the hemispherical bowl
    """

    [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
    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)
    image = None
    # Your code here
    # Identify the visible region on the hemisphere
    visible = Z > 0  # Only points within the bowl's radius in positive Z

    # Calculate surface normals for the visible points on the hemisphere
    Nx = X[visible] - center[0]
    Ny = Y[visible] - center[1]
    Nz = Z[visible] - center[2]
    normals = np.stack((Nx, Ny, Nz), axis=-1)
    normals = normals / np.linalg.norm(normals, axis=-1, keepdims=True)

    # Calculate the dot product (n dot l) for Lambertian lighting
    ndotl = np.dot(normals, light)
    ndotl[ndotl < 0] = 0  # Ignore negative values (light coming from behind)

    # Create the final image with lighting applied
    image = np.zeros((res[1], res[0]))
    image[visible] = ndotl/np.max(ndotl)*255


    return image
```

Figure 6: Code Snippet

Q1-c at page 4

Ans:

Following Figure 7 shows the conde snippet of loadData() in q1.py.

```python
def loadData(path="../data/"):
    """
    Question 1 (c)

    Load data from the path given. The images are stored as input_n.tif
    for n = {1...7}. The source lighting directions are stored in
    sources.mat.

    Parameters
    ---------
    path: str
        Path of the data directory

    Returns
    -------
    I : numpy.ndarray
        The 7 x P matrix of vectorized images

    L : numpy.ndarray
        The 3 x 7 matrix of lighting directions

    s: tuple
        Image shape

    """

    I = None
    L = None
    s = None
    # Your code here
    I = []

    # Load images and convert to luminance channel
    for n in range(1, 8):
        image = io.imread(f"{path}/input_{n}.tif").astype(np.uint16)
        assert image.dtype == np.uint16

        xyz_image = rgb2xyz(image)
        luminance = xyz_image[:, :, 1]  # Extract Y channel (luminance)
        #luminance = (luminance*65535).astype(np.uint16)
        if s is None:
            s = luminance.shape
        I.append(luminance.flatten())
    I = np.array(I)

    # Load the sources file
    L = np.load(f"{path}/sources.npy").T

    return I, L, s
```

Figure 7: Code Snippet

Q1-d at page 4

Ans:

L^T has the shape (7, 3), representing the transposed light directions. If L has rank of 3, it means 3 of the light directions in L can span the whole 3-dimensional space. Also, B has the shape of (3, P), representing the set of pseudonormals in the images. If B has rank 3, it means 3 of the pseudonormals in B could span the whole 3-dimensional space. If both L and B have rank of 3, then $I = L^T B$ would also have rank 3, meaning that the singular value of I should agree with rank 3.

However, we can see from below Figure 8, the singular value shows that I has rank of 7, not agreeing with rank-3 requirement. It is because that if the light directions in L or the pseudonormals in B are not linearly independent, due to the noise or measurement imprecision of light on camera intensity, then the rank of L and B might not be exactly 3, resulting in the disagreement here. The Figure 9 shows the conde snippet in q1.py.

```
Singular Values of I: [79.36348099 13.16260675  9.22148403  2.414729    1.61659626  1.26289066
  0.89368302]
```

Figure 8: Singular Values of I

```
293     # Part 1(d)
294     # Your code here
295     u_vec, singular, vt_vec = np.linalg.svd(I, full_matrices=False)
296     print("Singular Values of I:", singular)
297
```

Figure 9: Code Snippet of (d)

Q1-e at page 4

Ans:

The way I construct A and B is shown in the following derivation of estimated B:

$$\mathbf{I} = \mathbf{L^T B} \tag{7}$$

$$Set\ \mathbf{A} = L^T, \mathbf{x} = \mathbf{B}, \mathbf{y} = \mathbf{I} \tag{8}$$

$$\rightarrow \mathbf{Ax} = \mathbf{y} \tag{9}$$

$$\rightarrow \mathbf{x} = (\mathbf{A^T A})^{-1}\mathbf{A^T y} \tag{10}$$

$$\rightarrow \mathbf{B} = (\mathbf{L L^T})^{-1}\mathbf{L I} \tag{11}$$

Following Figure 10 shows the code snippet of (e) in q1.py.

```
130 def estimatePseudonormalsCalibrated(I, L):
131     """
132     Question 1 (e)
133 
134     In calibrated photometric stereo, estimate pseudonormals from the
135     light direction and image matrices
136 
137     Parameters
138     ----------
139     I : numpy.ndarray
140         The 7 x P array of vectorized images
141 
142     L : numpy.ndarray
143         The 3 x 7 array of lighting directions
144 
145     Returns
146     -------
147     B : numpy.ndarray
148         The 3 x P matrix of pesudonormals
149     """
150 
151     # Your code here
152     B = np.linalg.inv(L@L.T)@(L@I)
153     return B
```

Figure 10: Code Snippet of (e)

Q1-f at page 5

Ans:

The Figure 11 and Figure 12 show the result of estimated albedo and normals image respectively. The albedo image shows the reflectance of the face, and normal image shows the normal directions of the face. The forehead and cheek has smooth color changing, meaning there exist smooth change of normal direction among these areas, while the area like nose and lips show sharp change of color like boundaries, indicating the normal has sharp change among these areas. As of the albedo image, the forehead, cheek, nose, and ears show high intensity, indicating these areas have high reflectance property, aligning with my expectation because the skin of these area are usually shiny and oily. The Figure 13 and Figure 14 show the code snippets in q1.py.

Figure 11: Estimated albedo

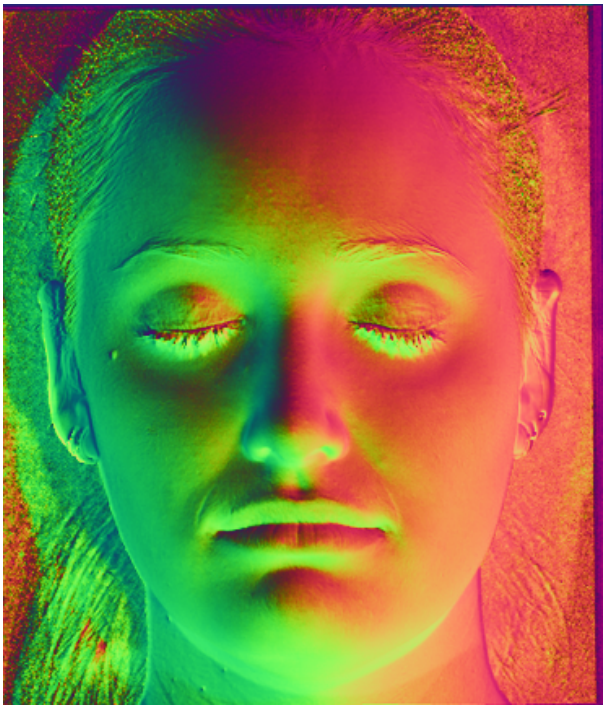
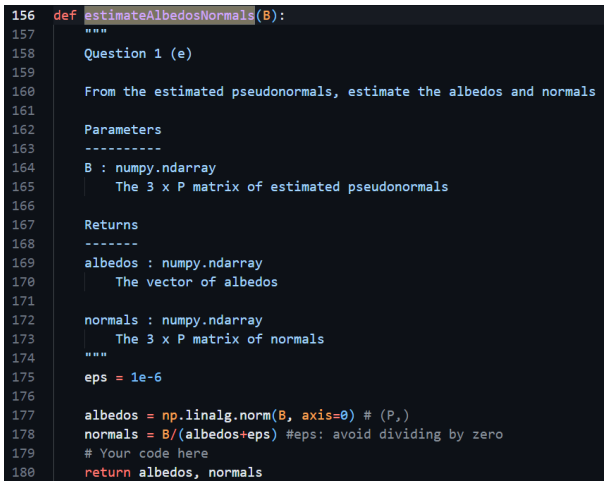
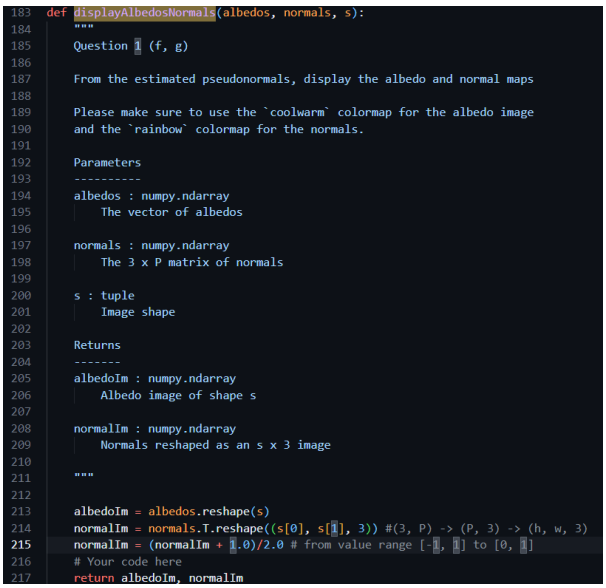


Figure 12: Estimated normals



```python
def estimateAlbedosNormals(B):
    """
    Question 1 (e)

    From the estimated pseudonormals, estimate the albedos and normals

    Parameters
    ----------
    B : numpy.ndarray
        The 3 x P matrix of estimated pseudonormals

    Returns
    -------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals
    """
    eps = 1e-6

    albedos = np.linalg.norm(B, axis=0) # (P,)
    normals = B/(albedos+eps) #eps: avoid dividing by zero
    # Your code here
    return albedos, normals
```

Figure 13: Code Snippet of estimateAlbedosNormals()



```python
def displayAlbedosNormals(albedos, normals, s):
    """
    Question 1 (f, g)

    From the estimated pseudonormals, display the albedo and normal maps

    Please make sure to use the `coolwarm` colormap for the albedo image
    and the `rainbow` colormap for the normals.

    Parameters
    ----------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    -------
    albedoIm : numpy.ndarray
        Albedo image of shape s

    normalIm : numpy.ndarray
        Normals reshaped as an s x 3 image

    """

    albedoIm = albedos.reshape(s)
    normalIm = normals.T.reshape((s[0], s[1], 3)) #(3, P) -> (P, 3) -> (h, w, 3)
    normalIm = (normalIm + 1.0)/2.0 # from value range [-1, 1] to [0, 1]
    # Your code here
    return albedoIm, normalIm
```

Figure 14: Code Snippet of displayAlbedosNormals()

Q1-g at page 5

Ans:

The surface normal at point(x, y) is n = (n_1, n_2, n_3), and 3D depth map function at this point is z = f(x, y). We can have the tangent along x direction as: (1, 0, $\frac{\partial f(x,y)}{\partial x}$), and tangent along y direction as (0, 1, $\frac{\partial f(x,y)}{\partial y}$). Then the tangent x has the following relationship with normal n vector:

$$\mathbf{tangent}_x \cdot \mathbf{n} = (1, 0, \frac{\partial f(x,y)}{\partial x}) \cdot (n_1, n_2, n_3) = 0 \tag{12}$$

$$\rightarrow n_1 + n_3 \frac{\partial f(x,y)}{\partial x} = 0 \tag{13}$$

$$\rightarrow -\frac{n_1}{n_3} = \frac{\partial f(x,y)}{\partial x} = f_x \tag{14}$$

We can do the similar thing on the tangent y vector with normal n vector:

$$\mathbf{tangent}_y \cdot \mathbf{n} = (0, 1, \frac{\partial f(x,y)}{\partial y}) \cdot (n_1, n_2, n_3) = 0 \tag{15}$$

$$\rightarrow n_2 + n_3 \frac{\partial f(x,y)}{\partial y} = 0 \tag{16}$$

$$\rightarrow -\frac{n_2}{n_3} = \frac{\partial f(x,y)}{\partial y} = f_y \tag{17}$$

As one can see, the equation holds as the statement in Q1-g in hw5.pdf.

Q1-h at page 6

Ans:

We have the following two gradient calculations:

$$g_x(x_i, y_j) = g(x_{i+1}, y_j) - g(x_i, y_j) \tag{18}$$
$$g_y(x_i, y_j) = g(x_i, y_{j+1}) - g(x_i, y_j) \tag{19}$$

When using equation (18), the gradient of x, g_x, is:

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

When using equation (19), the gradient of y, g_y, is:

$$g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

Method 1. Given g(0, 0) = 1, use g_x to construct the first row of g, then use g_y to construct the rest of g:
Step1. Use g_x to construct the first row: g(0, j) = g(0, j-1) + g_x(0, j-1), $1 \leq j \leq 3$, the order is from left to right.

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Step2. Use g_y for the rest: g(i, j) = g(i-1, j) + g_y(i-1, j), $1 \leq i \leq 3$, $0 \leq j \leq 3$, the order is from top to down. The reconstructed g is:

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

Method 2. Given g(0, 0) = 1, use g_y to construct the first column of g, then use g_x to construct the rest of g:
Step1. Use g_y to construct the first column: g(i, 0) = g(i-1, 0) + g_y(i-1, 0), $1 \leq i \leq 3$, the order is from top to down.

$$g = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \\ 13 & 0 & 0 & 0 \end{bmatrix}$$

Step2. Use g_x for the rest: g(i, j) = g(i, j-1) + g_x(i, j-1), $0 \leq i \leq 3$, $1 \leq j \leq 3$, the order is from left to right. The reconstructed g is:

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

As we can see above, both procedures produce the same reconstruction of g.
The integrability implies that:

$$\frac{\partial g_x}{\partial y} = \frac{\partial g_y}{\partial x} \tag{20}$$

So, we can modify the g_x and g_y as below to make them non-integrable:

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 30 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$g_y = \begin{bmatrix} 4 & 4 & 7 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

Then the reconstructed g using Method 1 above would be:

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 10 & 8 \\ 9 & 10 & 14 & 12 \\ 13 & 14 & 18 & 16 \end{bmatrix}$$

The reconstructed g using Method 2 above would be:

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 10 & 8 \\ 9 & 10 & 40 & 41 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

As we can see above, the reconstructed g are different in these two procedures, thus the g_x and g_y are now non-integrable. The reasons that the gradients estimated in the way of (g) may be non-integrable include the measurement noise in g_x or g_y, wrong assumptions on surface reflectance, or the existence of non-smooth surface.

Q1-i at page 6

Ans:

The following Figure 16 - Figure 18 show the results fo estimated shape in three different viewpoints. The code snippet is shown in Figure 15.

```python
def estimateShape(normals, s):
    """
    Question 1 (j)

    Integrate the estimated normals to get an estimate of the depth map
    of the surface.

    Parameters
    ----------
    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    ----------
    surface: numpy.ndarray
        The image, of size s, of estimated depths at each point

    """

    surface = None
    # Your code here
    # Reshape normals into the image shape
    n1 = normals[0, :].reshape(s)  # X-component of normals, (h, w, 1)
    n2 = normals[1, :].reshape(s)  # Y-component of normals, (h, w, 1)
    n3 = normals[2, :].reshape(s)  # Z-component of normals, (h, w, 1)

    # Compute gradients f_x and f_y
    fx = -n1 / n3
    fy = -n2 / n3

    # Use utils.integrateFrankot for integration
    surface = integrateFrankot(fx, fy)
    return surface
```

Figure 15: Code Snippet

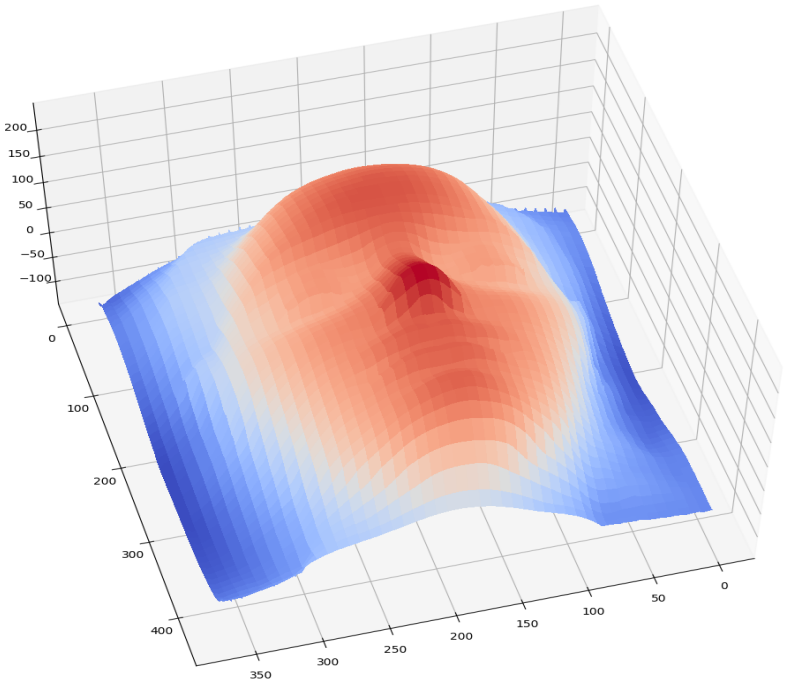


Figure 16: Result of Shape Estimation in View 1

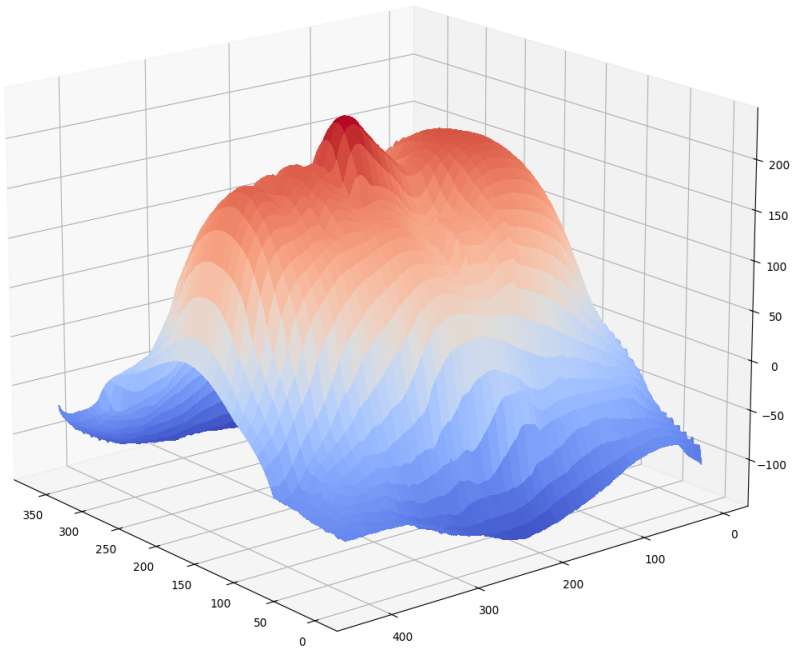


Figure 17: Result of Shape Estimation in View 2

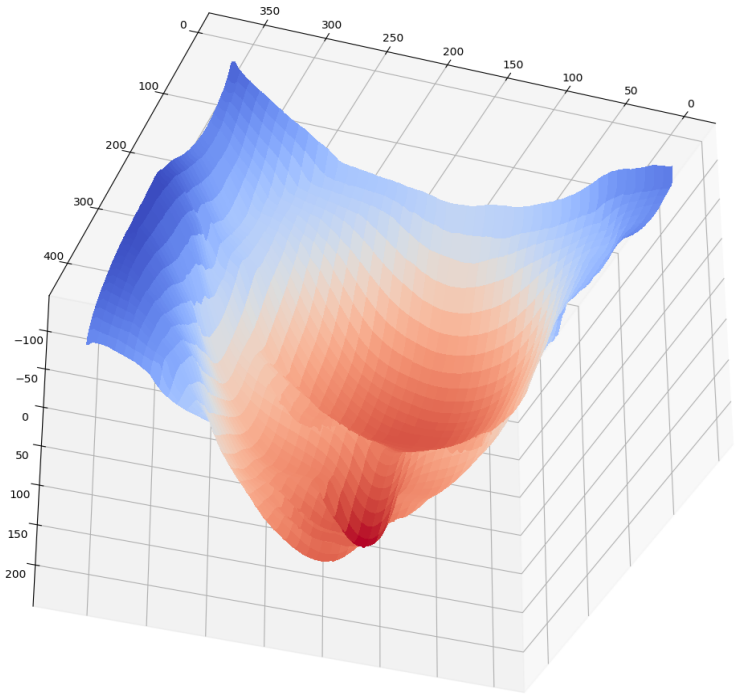


Figure 18: Result of Shape Estimation in View 3

Q2-a at page 7

Ans:

Since $\mathbf{I} = \mathbf{L^T B}$, and we do not know $\mathbf{L^T}$ and $\mathbf{B}$, we can directly do SVD on $\mathbf{I}$. This gives:

$$\mathbf{I} = \mathbf{U\Sigma V^T} \tag{21}$$

Since $\mathbf{I}$ is a 7xP matrix, so the $\mathbf{U}$ would be a 7x7 matrix, Σ would be a 7xP diagonal matrix, and $\mathbf{V^T}$ would be a PxP matrix, where P is the number of pixels.
To approximate the rank 3 of I, we first set all singular values except the top 3 from $\mathbf{\Sigma}$ to 0, and we get the diagonal 3x3 matrix $\hat{\mathbf{\Sigma}}$. Secondly, we choose the first 3 columns of $\mathbf{U}$ to get 7x3 matrix: $\hat{\mathbf{U}}$, choose the first 3 columns of $\mathbf{V}$ to get a Px3 matrix: $\hat{\mathbf{V}}$. Thus, we can get $\hat{\mathbf{I}}$ as below:

$$\hat{\mathbf{I}} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}^\mathbf{T}} \tag{22}$$

Where $\hat{\mathbf{I}}$ is the reconstructed rank 3 matrix with shape 7xP. Accordingly, we can use the result of equation (22) to approximate $\hat{\mathbf{L}}^T$ and $\hat{\mathbf{B}}$:

$$\hat{\mathbf{I}} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}^{1/2}\hat{\mathbf{\Sigma}}^{1/2}\hat{\mathbf{V}^\mathbf{T}} \tag{23}$$

$$= \hat{\mathbf{L}}^T\hat{\mathbf{B}} \tag{24}$$

Where $\hat{\mathbf{L}}^T = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}^{1/2}$ with shape 7x3, and $\hat{\mathbf{B}} = \hat{\mathbf{\Sigma}}^{1/2}\hat{\mathbf{V}^\mathbf{T}}$ with shape 3xP. So this is how we can use SVD to approximate $\hat{\mathbf{L}}^T$ and $\hat{\mathbf{B}}$ and construct a factorization of $\hat{\mathbf{I}}$ following the required constraints of rank 3.

Q2-b at page 7

Ans:

Below Figure 19 and Figure 20 shows the results generated by the method mentioned in Q2-a. Figure 21 and Figure 22 show the code snippets of this question in q2.py.

Figure 19: Estimated albedo

Figure 20: Estimated normals

```python
def estimatePseudonormalsUncalibrated(I):
    """
    Question 2 (b)

    Estimate pseudonormals without the help of light source directions.

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P matrix of loaded images

    Returns
    -------
    B : numpy.ndarray
        The 3 x P matrix of pseudonormals

    L : numpy.ndarray
        The 3 x 7 array of lighting directions

    """

    U, Sigma, Vt = np.linalg.svd(I, full_matrices=False)
    Sigma_hat = np.diag(Sigma[:3]) # shape(3, 3)
    U_hat = U[:, :3] #shape(7, 3)
    V_hat =Vt[:3, :] #shape(3, P)

    L_trans = U_hat @ np.sqrt(Sigma_hat) #shape(7, 3)
    B = np.sqrt(Sigma_hat) @ V_hat #shape(3, P)

    # Your code here
    return B, L_trans.T
```

Figure 21: Code Snippet 1

```python
if __name__ == "__main__":
    I, L, s = loadData("../data/")
    formatted_print('L', L)

    '''
    # test code
    print(f"L = {L}")
    print(f"I[0].dtype = {I[0].dtype}")

    num_images = I.shape[0]
    for i in range(num_images):
        image = I[i].reshape(s)  # Reshape to original shape
        plt.imshow(image, cmap='gray')
        plt.title(f"Image {i+1}")
        plt.axis('off')
        plt.show()
    '''

    # Part 2 (b)
    # Your code here
    B_hat, L_hat = estimatePseudonormalsUncalibrated(I)
    albedos, normals = estimateAlbedosNormals(B_hat)
    albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
    plt.imsave("2b-a.png", albedoIm, cmap="gray")
    plt.imsave("2b-b.png", normalIm, cmap="rainbow")
    formatted_print('L_hat', L_hat)
```

Figure 22: Code Snippet 2

Q2-c at page 8

Ans:

Below Figure 23 shows the original $\mathbf{L}$ and the reconstructed $\hat{\mathbf{L}} = \mathbf{L\_hat}$. As we can see, $\mathbf{L}$ and $\hat{\mathbf{L}}$ are not similar, but quite different. There are many ways to change the procedure in (a) to change $\hat{\mathbf{L}}$ and $\hat{\mathbf{B}}$, but keeps the rendered image using them the same. For example, we can set $\hat{\mathbf{L}}^T = \hat{\mathbf{U}}(\frac{1}{2}\hat{\mathbf{\Sigma}}^{1/2})$, $\hat{\mathbf{B}} = (2\hat{\mathbf{\Sigma}}^{1/2})\hat{\mathbf{V}^{\mathbf{T}}}$, and $\hat{\mathbf{L}}^T\hat{\mathbf{B}}$, the rendered image, is still the same.

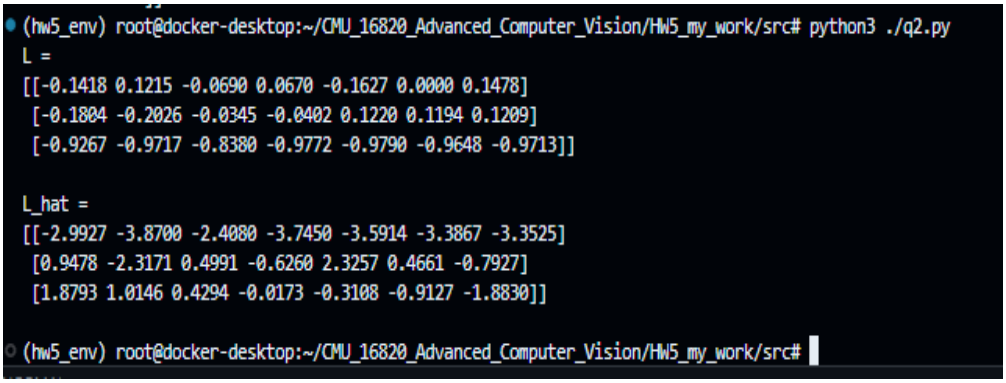


Figure 23: Comparison of $\mathbf{L}$ and $\hat{\mathbf{L}}$

Q2-d at page 8

Ans:

Figure 24 and Figure 25 show different views of the reconstructed 3D depth map shape using Franot-Chellappa algorithm. It does not look like a face.

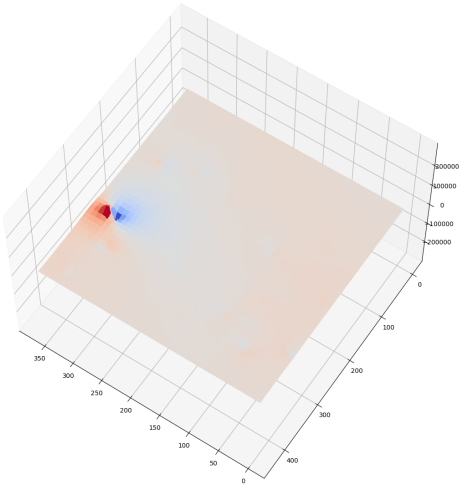


Figure 24: Reconstructed 3D Depth Map Shape View1

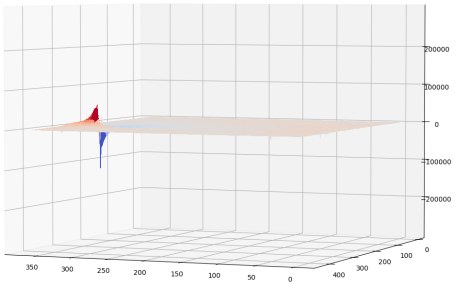


Figure 25: Reconstructed 3D Depth Map Shape View2

Q2-e at page 8

Ans:

The following Figure 26 - Figure 28 show the results of estimated shape in three different viewpoints using pseudonormals and enforceIntegrability() in utils.py. The surface does look like the one output by calibrated photometric stereo in Q1-i question above, but it seems to be a little more flattened, as one can see the height range of the results here are only in the range of [-50, 100], while the results in Q1-i are in the range of [-100, 200].

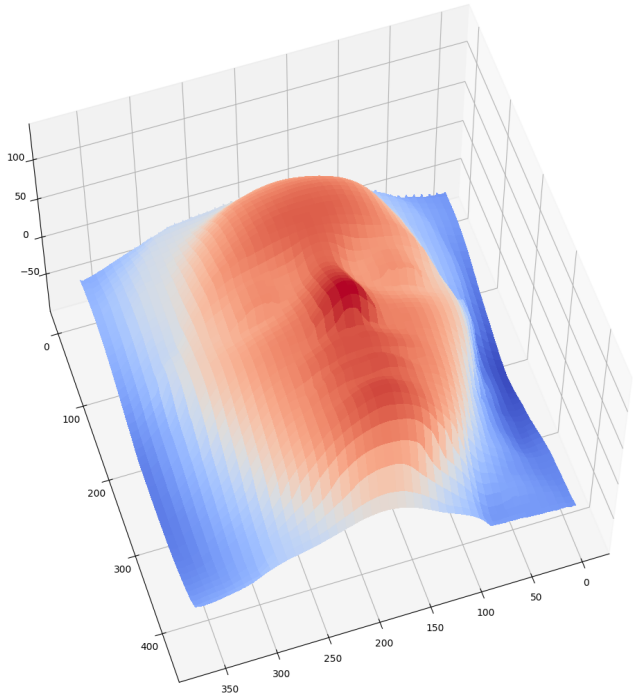


Figure 26: Result of Shape Estimation in View 1

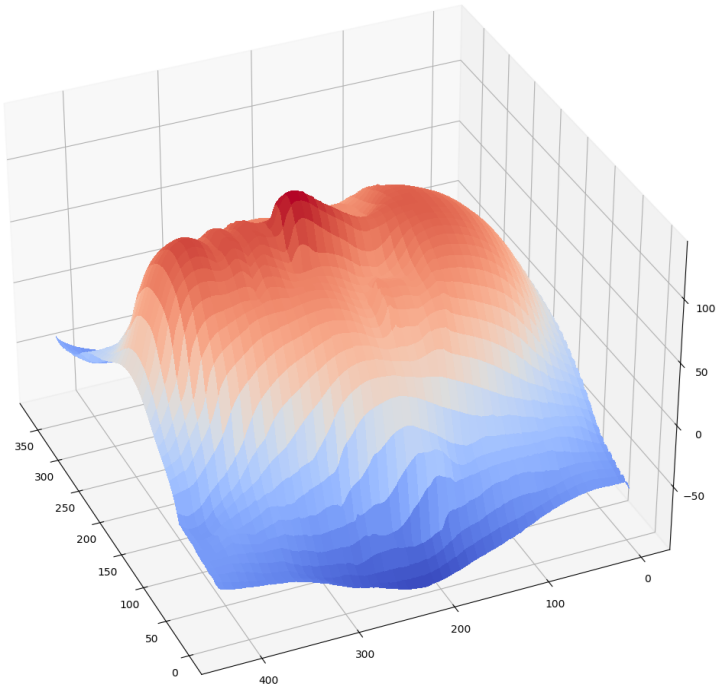


Figure 27: Result of Shape Estimation in View 2

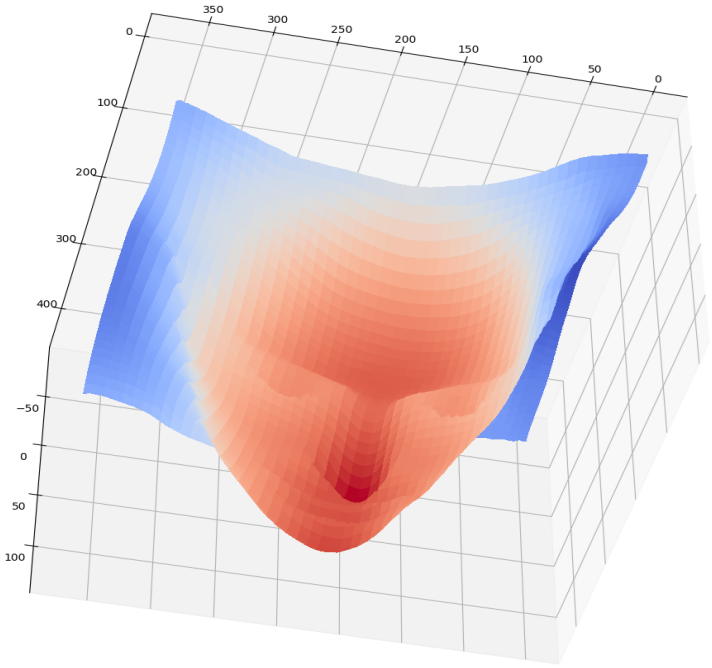
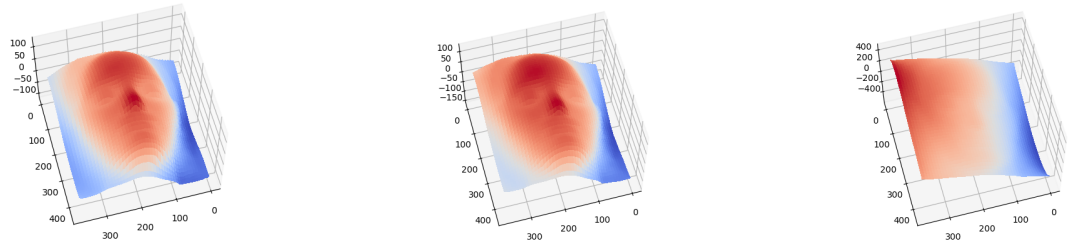


Figure 28: Result of Shape Estimation in View 3

Q2-f at page 9

Ans:

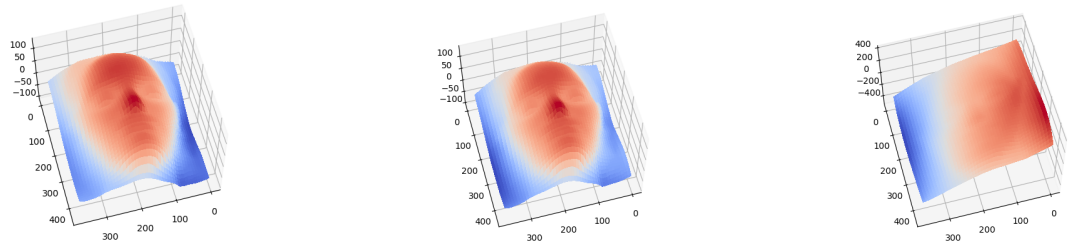
Let's discuss how the value of μ ν λ affect the reconstructed surface:
When μ = [-0.1, -1, -10], and ν = 1, λ = 1:



(a) μ=-0.1, ν=1, λ=1 (b) μ=-1, ν=1, λ=1 (c) μ=-10, ν=1, λ=1

Figure 29: Impact of Negative μ Values on Surface

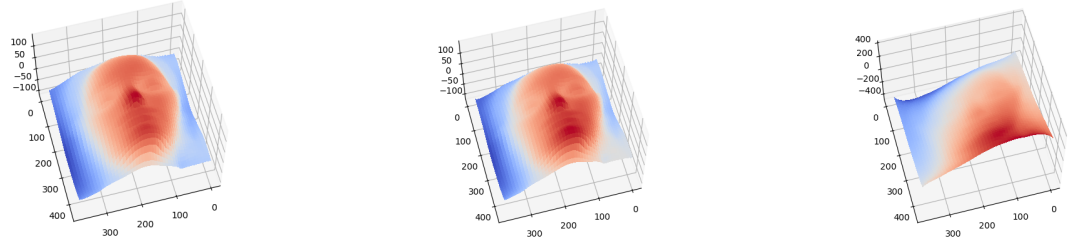
When μ = [0.1, 1, 10], and ν = 1, λ = 1:



(a) μ=0.1, ν=1, λ=1 (b) μ=1, ν=1, λ=1 (c) μ=10, ν=1, λ=1

Figure 30: Impact of Positive μ Values on Surface

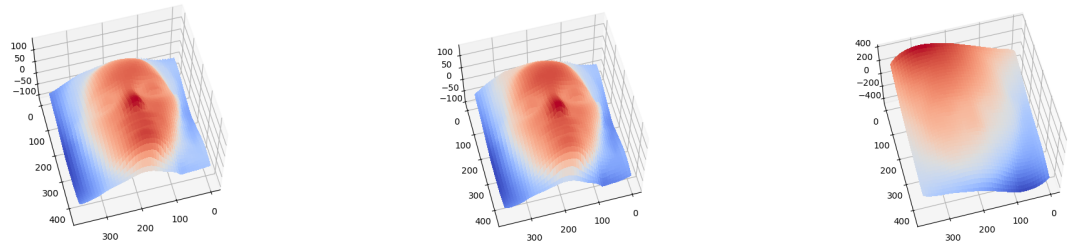
When ν = [-0.1, -1, -10], and μ = 1, λ = 1:



(a) μ=1, ν=-0.1, λ=1 (b) μ=1, ν=-1, λ=1 (c) μ=1, ν=-10, λ=1

Figure 31: Impact of Negative ν Values on Surface

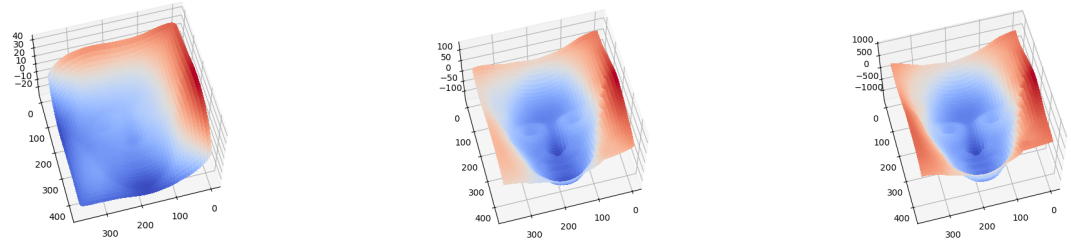
When ν = [0.1, 1, 10], and $\mu = 1$, $\lambda = 1$:



(a) μ=1, ν=0.1, λ=1 (b) μ=1, ν=1, λ=1 (c) μ=1, ν=10, λ=1

Figure 32: Impact of Positive ν Values on Surface

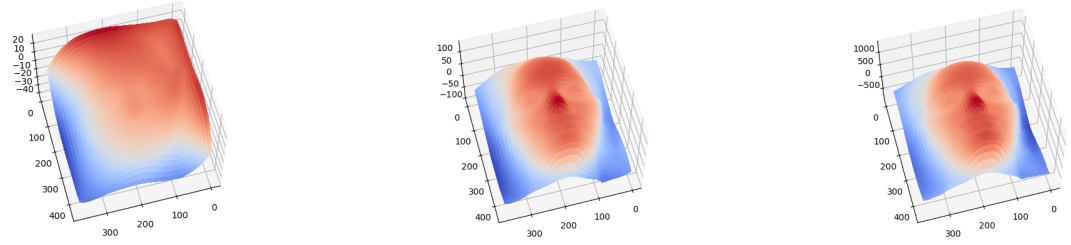
When λ = [-0.1, -1, -10], and $\mu = 1$, $\nu = 1$:



(a) μ=1, ν=1, λ=-0.1 (b) μ=1, ν=1, λ=-1 (c) μ=1, ν=1, λ=-10

Figure 33: Impact of Negative λ Values on Surface

When λ = [0.1, 1, 10], and $\mu = 1$, $\nu = 1$:



(a) μ=1, ν=1, λ=0.1 (b) μ=1, ν=1, λ=1 (c) μ=1, ν=1, λ=10

Figure 34: Impact of Positive λ Values on Surface

As we can see above, when μ changes from small values to large values, the surface gets more flattened, while the positive and negative values affect the surface to be more flattened toward opposite directions.

The ν changes in similar way, as it changes from small values to large values, the surface gets more flattened, while the positive and negative values affect the surface to be more flattened in opposite directions. The λ changes in another particular way: as its value change from small to large, the curve of the surface becomes sharper, while the positive and negative values affect the surface to be sharper in opposite directions.

The reason why bas-relief ambiguity is named so is that the GBR transform preserves the dot product of surface normal and light direction while producing different surface shape by tilting and scaling curved depth (the x and y components of B are not changed, only z component becomes the combination of x, y and z components after GBR transform), which can be seen from above Figure 29 to Figure 34. That is to say, $\mathbf{I}$, which is $\mathbf{L}^T\mathbf{B}$, would be equal to $\mathbf{I}' = \mathbf{L}^T\mathbf{B}'$, where $\mathbf{B}' = \mathbf{G}^{-\mathbf{T}}\mathbf{B}$, because the appearance $\mathbf{I}$ only depends on the $\cos\theta$ between $\mathbf{L}$ and $\mathbf{B}$. As a result, we cannot reconstruct a unique shape based only on appearance $\mathbf{I}$.

Q2-g at page 9

Ans:

As we can see the trend in Q2-f, the larger absolute value of μ or ν, not both, the more flattened the reconstructed shape is. Besides, the smaller absolute value of λ, the more flattened the reconstructed shape is. Thus, we can use the setting of $\mu = 10$, $\nu = 1$, and λ=0.001 to generate the flattened shape as below Figure 35.

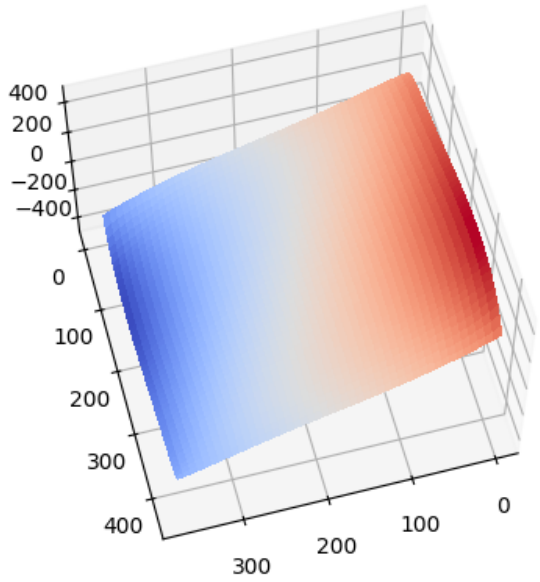
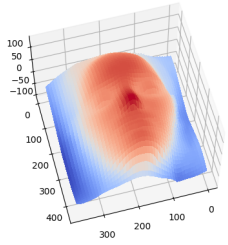
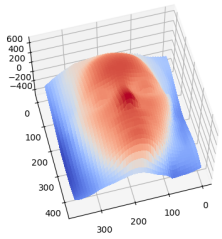


Figure 35: Result of Flattened Shape

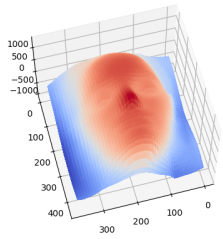
Another interesting fact is that when we set both μ and ν to be large, the reconstructed shape would not be flattened:



(a) μ=1, ν=1



(b) μ=5, ν=5



(c) μ=10, ν=10

Q2-h at page 9

Ans:

Acquiring more pictures does not help resolve the ambiguity, because with only known appearance **I**, we can only calculate the reconstructed **B** up to a matrix of the form in Eq. 2 above, the GBR transform, generating the same appearance as original **B** does. To resolve the ambiguity, we may need the light direction **L**, the depth information, or other features like the geometry of the object.

Collaborations

Ans:
Though I do not have collaborators, I found the following websites helpful on understanding the concepts in this homework.

1. https://www.ri.cmu.edu/pub_files/pub3/baker_simon_2003_3/baker_simon_2003_3.pdf.
2. https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/classes/cs294-appearance_models/sp2001/cache/belhumeur99.pdf
3. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imsave.html
4. https://https://en.wikipedia.org/wiki/Singular_value_decomposition
5. http://graphics.cs.cmu.edu/courses/15-463/2019_fall/lectures/lecture16.pdf