16-720A — Spring 2021 — Homework 6

Jen-Hung Ho
jenhungh@andrew.cmu.edu

May 7, 2021

1 Calibrated photometric stereo

(a) **Understanding n-dot-l lighting**.
The n-dot-l lighting model:

$$L = \frac{\rho_d}{\pi} A(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{l}}) = \frac{\rho_d}{\pi} Acos(\theta) \tag{1}$$

where θ is the angle between the lighting direction and the normal vector.

Since the normal vector $\hat{n}$ and the lighting direction $\hat{l}$ are both unit vector, their dot product is $cos(\theta)$. The projected area is A$cos(\theta)$, where A is the original area. The viewing direction doesn't matter because the brightness of the whole surface is equal.

(b) **Rendering n-dot-l lighting**.

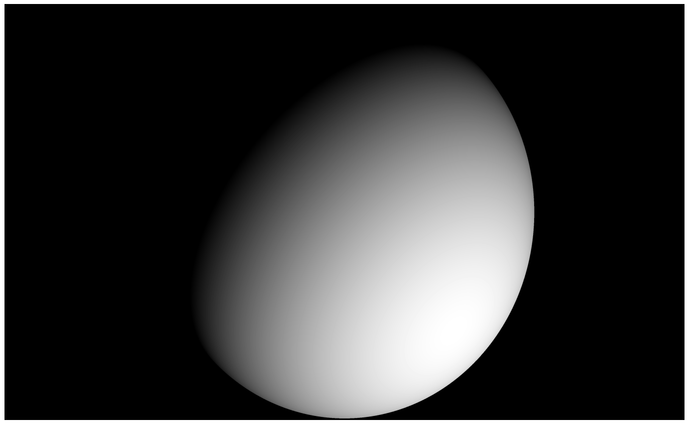

Figure 1: The appearance of the sphere with incoming lighting direction $(1, 1, 1)/\sqrt{3}$

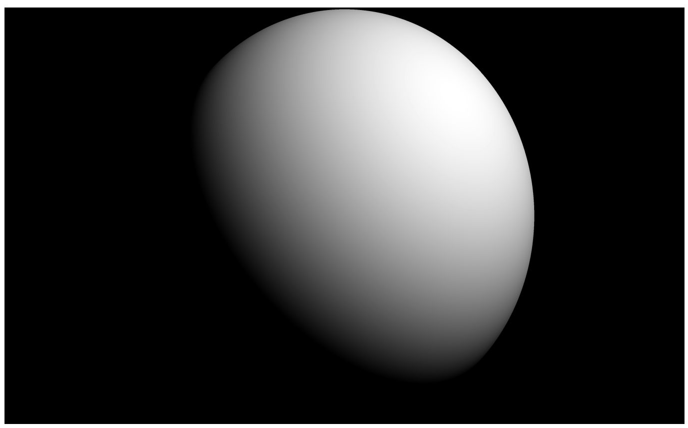

Figure 2: The appearance of the sphere with incoming lighting direction $(1, -1, 1)/\sqrt{3}$

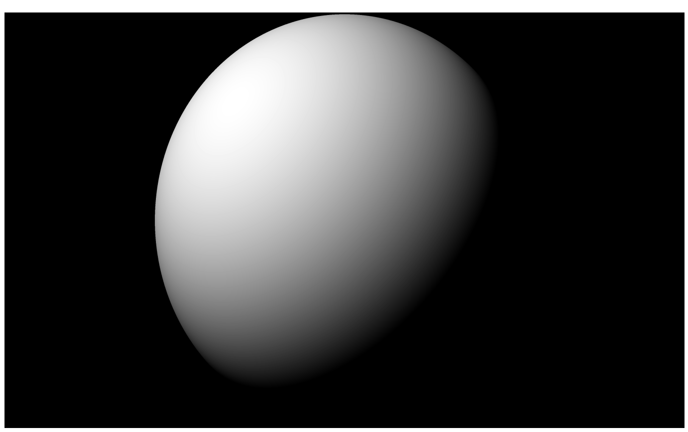

Figure 3: The appearance of the sphere with incoming lighting direction $(-1, -1, 1)/\sqrt{3}$

(d) **Initials**.

The rank of **I** should be 3 because the model is in 3D coordinates, and thus the minimum lighting directions for reconstruction is 3. The size of **I**, **L**, and **B** are (3, P), (3, 3), and (3, P). However, the singular values of **I** = [79.36348099 13.16260675 9.22148403 2.414729 1.61659626 1.26289066 0.89368302], which does NOT agree with the rank-3 requirement. This is probably because of the noises or blur of the real world images. Hence, we need more lighting directions (7 lighting directions for this model) to reconstruct the surface.

(e) **Estimating pseudonormals**.

In general, a Least-Squares Problem is formed as $\mathbf{Ax} = \mathbf{y}$, and the estimation is the pseudo inverse x = $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$. In this problem, we treat the relation $\mathbf{L}^T\mathbf{B} = \mathbf{I}$ as a Least-Squares and construct $\mathbf{L}^T$ as matrix **A** and **I** as vector **y**. Therefore, the estimation of the pseudonormals is **B** = $(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{L}\mathbf{I}$.

(f) **Albedos and normals**.

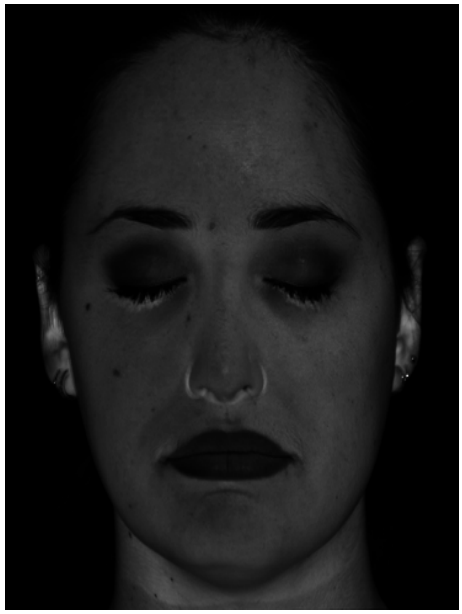

Figure 4: Visualization of the Albedos Image (gray colormap)

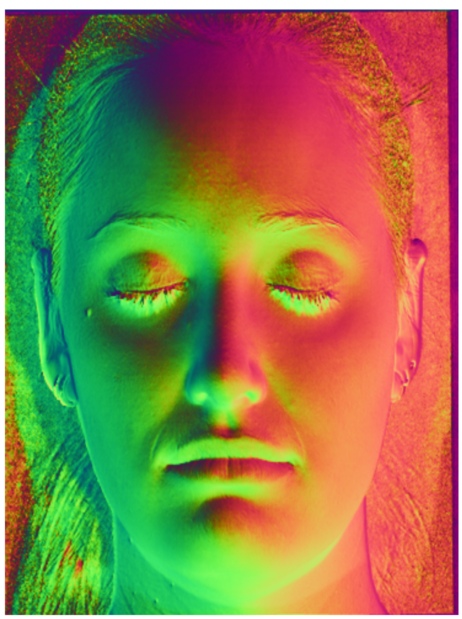

Figure 5: Visualization of the Normals Image (rainbow colormap)

From Figure 4, we can discover that the albedos image couldn't show the hair and performs poorly around the ears ans nose. Since we only consider 7 lighting directions and the n-dot-l algorithm couldn't handle shadows, we might lose some information if the area is under shadows.

(g) **Normals and depth**.
The equation of a surface is

$$n_1 * x + n_2 * y + n_3 * z + D = n_1 * x + n_2 * y + n_3 * f(x, y) + D = 0 \tag{2}$$

where the normal vector at point (x, y, z) is $\mathbf{n} = (n_1, n_2, n_3)$
Take partial derivatives of f at (x, y):

$$f(x, y) = -\frac{D}{n_3} - \frac{n_1}{n_3}x - \frac{n_2}{n_3}y \tag{3}$$

$$f_x = \frac{\partial f(x, y)}{\partial x} = -\frac{n_1}{n_3} \tag{4}$$

$$f_y = \frac{\partial f(x, y)}{\partial y} = -\frac{n_2}{n_3} \tag{5}$$

Therefore, we know that $\mathbf{n}$ is related to the partial derivatives of f at (x, y).

(h) **Understanding integrability of gradients**.
The gradients of the 2D and discrete function g are:

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{6}$$

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

Reconstruct function g using the two procedures, we can get the same reconstructed g:

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

If we modify the first row or column into a non-integrable function, then the estimated functions g in each procedure are not the same, which makes g_x and g_y non-integrable.

The gradients estimated in the way of (g) might be non-integrable because the partial derivatives with respect to x or y might not exist in some special functions.

(i) **Shape estimation**.

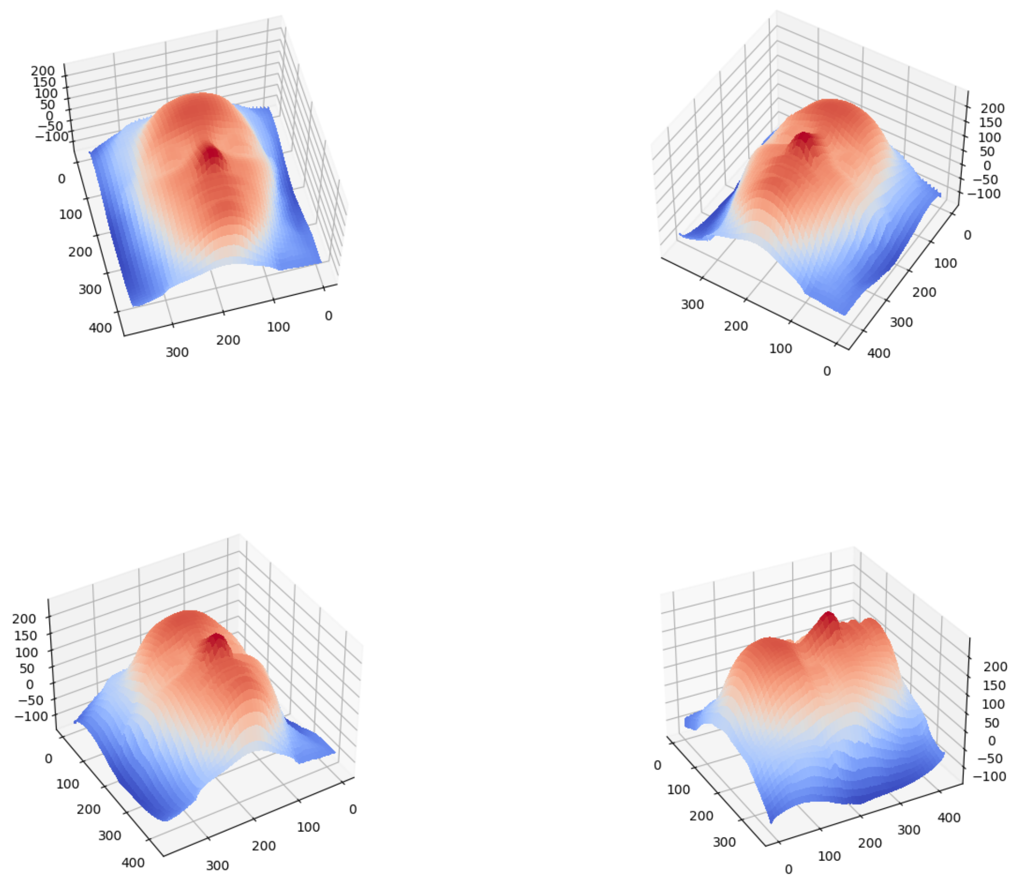

Figure 6: Visualization of the Reconstructed Surface (coolwarm colormap)

2 Uncalibrated photometric stereo

(a) **Uncalibrated normal estimation**.
We could compute the best rank-3 approximation of matrix **I**

$$\boldsymbol{I} = \boldsymbol{U\Sigma V^T} = (\boldsymbol{U\Sigma^{1/2}})(\boldsymbol{\Sigma^{1/2}V^T}) = \hat{\boldsymbol{L^T}}\hat{\boldsymbol{B}} \tag{9}$$

Set all singular values except the top 3 from Σ to 0 and extract the first 3 rows of $\hat{\boldsymbol{L}}$ and $\hat{\boldsymbol{B}}$. The shape of $\hat{\boldsymbol{L}}$ and $\hat{\boldsymbol{B}}$ are (3, 3) and (3, P).

(b) **Calculation and visualization**.

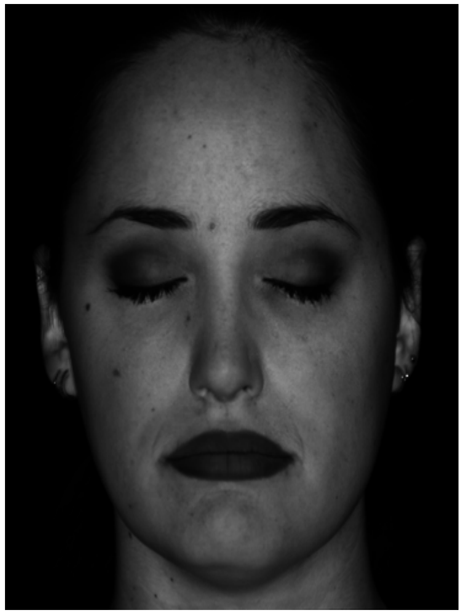

Figure 7: Visualization of the Albedos Image (gray colormap)

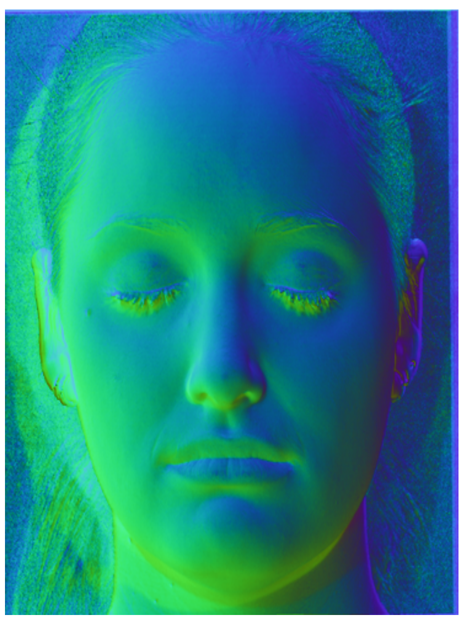

Figure 8: Visualization of the Normals Image (rainbow colormap)

(c) **Comparing to ground truth lighting**.
The ground truth lighting direction $\boldsymbol{L}_0 =$

$$\begin{bmatrix} -0.1418 & 0.1215 & -0.069 & 0.067 & -0.1627 & 0. & 0.1478 \\ -0.1804 & -0.2026 & -0.0345 & -0.0402 & 0.122 & 0.1194 & 0.1209 \\ -0.9267 & -0.9717 & -0.838 & -0.9772 & -0.979 & -0.9648 & -0.9713 \end{bmatrix} \tag{10}$$

The estimated lighting direction $\hat{\boldsymbol{L}} =$

$$\begin{bmatrix} -2.9927 & -3.87 & -2.408 & -3.745 & -3.5914 & -3.3867 & -3.3525 \\ 0.9478 & -2.3171 & 0.4991 & -0.626 & 2.3257 & 0.4661 & -0.7927 \\ 1.8793 & 1.0146 & 0.4294 & -0.0173 & -0.3108 & -0.9127 & -1.883 \end{bmatrix} \tag{11}$$

The ground truth and estimated lighting directions are NOT similar. There are multiple factorization methods to choose. For example, set $\boldsymbol{L}^{\boldsymbol{T}}$ as $\mathbf{U\Sigma}$ and $\boldsymbol{B}$ as $\boldsymbol{V}^{\boldsymbol{T}}$. As long as the relation $\mathbf{I} = \boldsymbol{L}^{\boldsymbol{T}}\mathbf{B}$ doesn't change, the output images are the same.

(d) **Reconstructing the shape, attempt 1**.

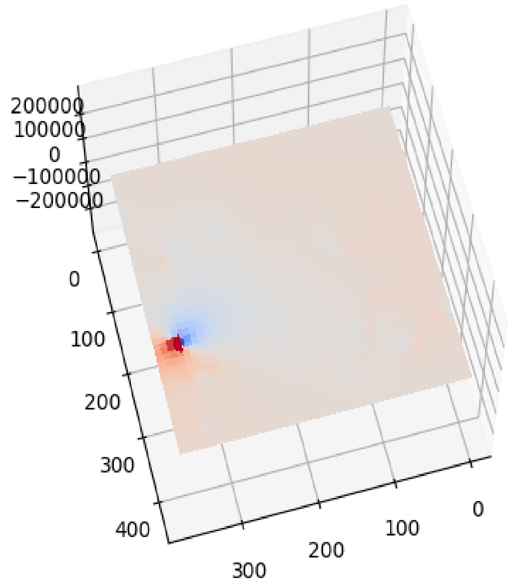

Figure 9: Visualization of the Reconstructed Surface (coolwarm colormap)

According to Figure 9, the reconstructed surface does NOT look like a face.

(e) **Reconstructing the shape, attempt 2**.

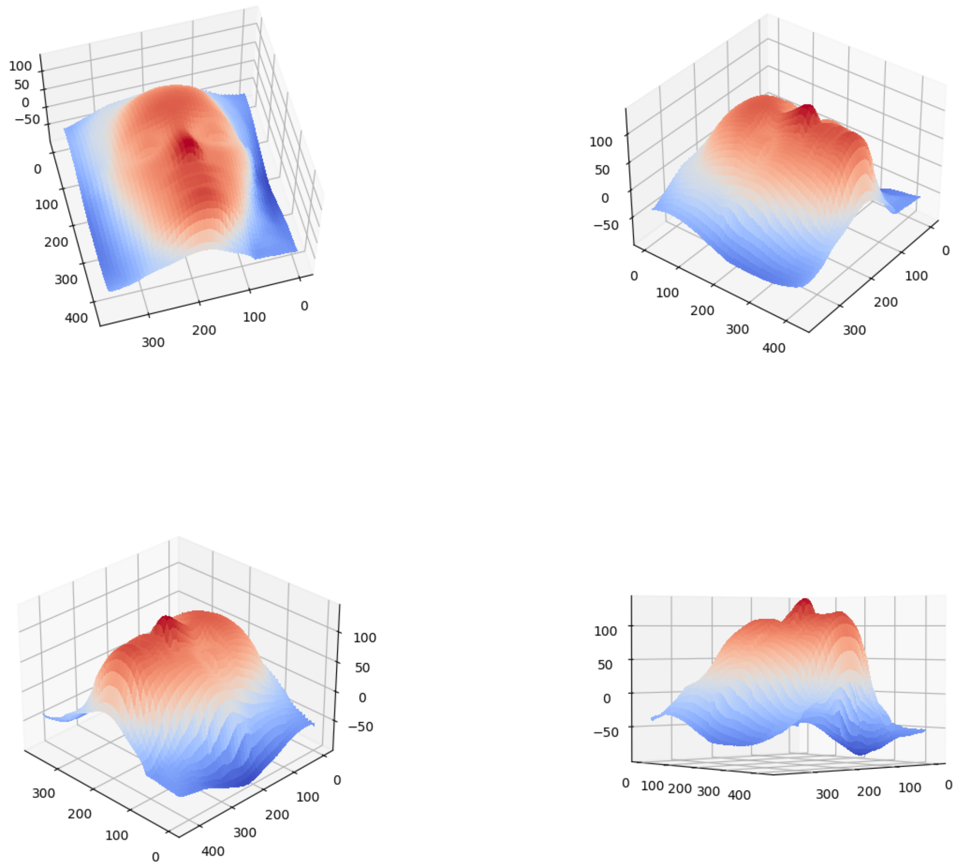

Figure 10: Visualization of the Reconstructed Surface (coolwarm colormap)

According to Figure 10, the reconstructed surface looks similar to the one output by calibrated photometric stereo.

(f) **Why low relief?**

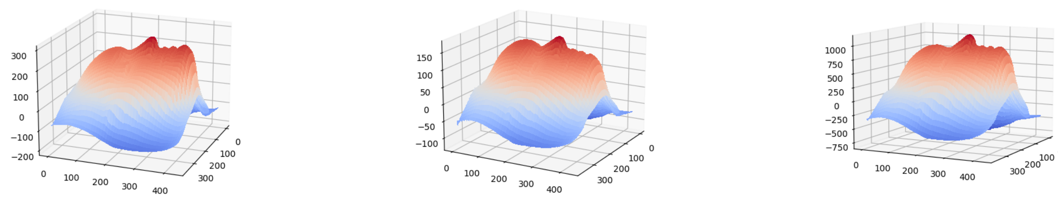

Figure 11: Vary the parameters μ in the bas-relief transformation (μ = -2, 1, 10)

From Figure 11, we could guess that parameter μ affects the gradients of the reconstructed surfaces since the range of the z axis increases with the values of μ.

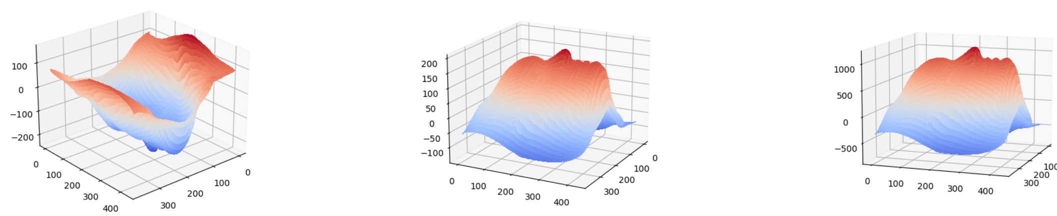

Figure 12: Vary the parameters ν in the bas-relief transformation (ν = -2, 1, 10)

From Figure 12, we could guess that parameter ν affects the gradients of the reconstructed surfaces since the range of the z axis increases with the values of ν. Also, ν affects the direction of the reconstructed surface. When ν is negative, the surface is upside down.

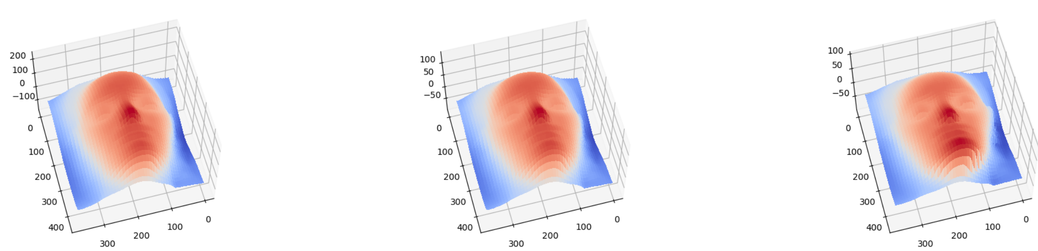

Figure 13: Vary the parameters λ in the bas-relief transformation (λ = 0.5, 1, 10)

From Figure 12, we could guess that parameter λ affects the flatness of the reconstructed surfaces since the surface is flatter when λ is large.

(g) **Flattest surface possible**.

If we want to make the estimated surface as flat as possible, I would increase λ (increase flatness) and set μ and ν to 0 (minimize the gradients).

(h) **More measurements**.

Acquiring more pictures from more lighting directions will help resolve the ambiguity since we have more information about the surface.

Code Appendix

2.1 q1.py

```
# ##################################################################### #
# 16720: Computer Vision Homework 6
# Carnegie Mellon University
# April 22, 2021
# ##################################################################### #

import numpy as np
from matplotlib import pyplot as plt
import skimage.io
from skimage.color import rgb2xyz
from utils import integrateFrankot, plotSurface

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.e-4
    Y = (Y - res[1]/2) * pxSize * 1.e-4
    Z = np.sqrt(rad**2 + 0j - X**2 - Y**2)
    Z = np.real(Z)

    image = None
    # Your code here
    # Build the normal vector
    n_vector = np.concatenate((X[:, :, np.newaxis], Y[:, :, np.newaxis],
                                                    Z[:, :, np.newaxis]), axis = 2)
    N = n_vector.reshape(res[0]*res[1], -1)
    # Normalization
    norm = np.linalg.norm(N, ord = 2, axis = 1)
    N = N / norm.reshape(-1, 1)

    # Implement NdotL Algorithm
    L = light
    image = (N @ L).reshape(res[1], res[0])
    image[np.real(Z) == 0] = 0

    return image


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.npy.

    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

```

```python
    s: tuple
        Image shape

    """

    I = None
    L = None
    s = None

    # Your code here
    # Load the image and Compute I and L
    num_img = 7
    for i in range(1, num_img+1):
        # Load the image and check the datatype
        input_img_rgb = skimage.io.imread(path + f'input_{i}.tif')
        input_img_rgb = input_img_rgb.astype(np.uint16)

        # Convert the RGB images into the XYZ color space
        input_img_xyz = rgb2xyz(input_img_rgb)

        # Compute s
        h, w, _ = input_img_rgb.shape
        s = (h, w)

        # Compute L
        if I is None:
            I = np.zeros((7, h*w))
        I[i-1, :] = input_img_xyz[:, :, 1].reshape(1, h*w)

    # Compute I
    L = np.load(path + 'sources.npy')
    L = L.T

    return I, L, s


def estimatePseudonormalsCalibrated(I, L):

    """
    Question 1 (e)

    In calibrated photometric stereo, estimate pseudonormals from the
    light direction and image matrices

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P array of vectorized images
```

```python

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

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

    B = None
    # Your code here
    # Least Square Problem : Ax = y
    # L.T @ B = I
    A = L.T
    y = I
    # Pseudo-inverse : x = inv(A.T @ A) @ A.T @ y
    B = np.linalg.inv(A.T @ A) @ A.T @ y

    return B


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
    '''

    albedos = None
    normals = None
    # Your code here
    # albedos = the magnitudes of the pseudonormals
```

```python
    albedos = np.linalg.norm(B, ord = 2, axis = 0)

    # normals = normalized normal vectors
    normals = B / albedos

    return albedos, normals


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 = None
    normalIm = None
    # Your code here
    # Reshape albedos
    albedoIm = albedos.reshape(s)

    # Rescale and Reshape normals
    normals = (normals + 1) / 2
    normalIm = normals.T.reshape(s[0], s[1], 3)
```

```

    return albedoIm, normalIm


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
    # Rescale and Reshape normals
    # normals = (normals + 1) / 2
    normalIm = normals.T.reshape(s[0], s[1], 3)

    # Compute the partial derivatives
    n1, n2, n3 = normalIm[:, :, 0], normalIm[:, :, 1], normalIm[:, :, 2]
    df_dx = -n1 / n3
    df_dy = -n2 / n3

    # Estimate the actual surface
    surface = integrateFrankot(df_dx, df_dy)

    return surface


if __name__ == '__main__':
    # Part 1(b)
    radius = 0.75 # cm
    center = np.asarray([0, 0, 0]) # cm
```

```
    pxSize = 7 # um
    res = (3840, 2160)

    light = np.asarray([1, 1, 1])/np.sqrt(3)
    image = renderNDotLSphere(center, radius, light, pxSize, res)
    plt.figure()
    plt.imshow(image, cmap = 'gray', vmin = 0, vmax = 1)
    plt.show()
    plt.imsave('1b-a.png', image, cmap = 'gray', vmin = 0, vmax = 1)

    light = np.asarray([1, -1, 1])/np.sqrt(3)
    image = renderNDotLSphere(center, radius, light, pxSize, res)
    plt.figure()
    plt.imshow(image, cmap = 'gray', vmin = 0, vmax = 1)
    plt.show()
    plt.imsave('1b-b.png', image, cmap = 'gray', vmin = 0, vmax= 1)

    light = np.asarray([-1, -1, 1])/np.sqrt(3)
    image = renderNDotLSphere(center, radius, light, pxSize, res)
    plt.figure()
    plt.imshow(image, cmap = 'gray', vmin = 0, vmax = 1)
    plt.show()
    plt.imsave('1b-c.png', image, cmap = 'gray', vmin = 0, vmax = 1)

    # Part 1(c)
    I, L, s = loadData('../data/')

    # Part 1(d)
    # Singular Value Decomposition of I
    u, sin, vh = np.linalg.svd(I, full_matrices = False)
    print(f"Singular Value of I = {sin}")

    # Part 1(e)
    B = estimatePseudonormalsCalibrated(I, L)

    # # Part 1(f)
    albedos, normals = estimateAlbedosNormals(B)
    albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
    plt.imsave('1f-a.png', albedoIm, cmap = 'gray')
    plt.imsave('1f-b.png', normalIm, cmap = 'rainbow')

    # # Part 1(i)
    surface = estimateShape(normals, s)
    plotSurface(surface, suffix = '1i')
```

2.2 q2.py

```
# ##################################################################### #
# 16720: Computer Vision Homework 6
# Carnegie Mellon University
# April 22, 2021
# ##################################################################### #

import numpy as np
import matplotlib.pyplot as plt
from q1 import loadData, estimateAlbedosNormals, displayAlbedosNormals,
    estimateShape
from q1 import estimateShape
from utils import enforceIntegrability, plotSurface

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

    """

    B = None
    L = None
    # Your code here
    # Singular Value Decomposition of I
    u, sin, vh = np.linalg.svd(I, full_matrices = False)

    # Estimate the best rank-3 approximation
    # Set all other singular values to 0
    sin[3:] = 0
    s_sqrt = np.sqrt(np.diag(sin))

```

```python
    # Estimate B and L : L.T @ B = I
    B = (s_sqrt @ vh)[0:3, :]
    L = (u @ s_sqrt).T[0:3, :]

    return B, L

def plotBasRelief(B, mu, nu, lam, suffix = ''):

    """
    Question 2 (f)

    Make a 3D plot of of a bas-relief transformation with the given parameters.

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

    mu : float
        bas-relief parameter

    nu : float
        bas-relief parameter

    lambda : float
        bas-relief parameter

    Returns
    -------
        None

    """

    # Your code here
    # Form the G matrix
    G = np.array([[1, 0, 0], [0, 1, 0], [mu, nu, lam]])

    # Generalized bas-relief ambiguity
    new_B = np.linalg.inv(G).T @ B

    # Visulaize the reconstructed 3D depth map
    integrable_normals = enforceIntegrability(new_B, s)
    surface = estimateShape(integrable_normals, s)
    plotSurface(surface, suffix = suffix)


if __name__ == "__main__":
```

```python

    # Part 2 (b)
    # Estimate B_hat and L_hat
    I, L0, s = loadData()
    B_hat, L_hat = estimatePseudonormalsUncalibrated(I)
    # Visualize the estimated albedos and normals
    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')

    # Part 2 (c)
    # Compare L0 and L_hat
    print(f"Ground truth lighting L0 =\n{L0}")
    print(f"Estimated lighting L_hat =\n{L_hat}")

    # Part 2 (d)
    # Visualize the reconstructed 3D depth map
    surface = estimateShape(normals, s)
    plotSurface(surface, suffix = '2d')

    # Part 2 (e)
    # Transform non-integrable pseudonormals into integrable pseudonormals
    integrable_normals = enforceIntegrability(B_hat, s)
    # Visualize the reconstructed 3D depth map
    surface = estimateShape(integrable_normals, s)
    plotSurface(surface, suffix = '2e')

    # Part 2 (f)
    # Visualize the corresponding surfaces
    vary_parameters = [-2, 1, 10]
    for i in vary_parameters:
        plotBasRelief(B_hat, i, 0, 1, suffix = f'_mu_{i}')
        plotBasRelief(B_hat, 0, i, 1, suffix = f'_nu_{i}')
        if i > 0:
            plotBasRelief(B_hat, 0, 0, i, suffix = f'_lam_{i}')
```

2.3 utils.py

```
# ##################################################################### #
# 16720: Computer Vision Homework 6
# Carnegie Mellon University
# April 22, 2021
# ##################################################################### #

import numpy as np
import warnings
from scipy.ndimage import gaussian_filter
from matplotlib import pyplot as plt
from matplotlib import cm

def integrateFrankot(zx, zy, pad = 512):

    """
    Question 1 (j)

    Implement the Frankot-Chellappa algorithm for enforcing integrability
    and normal integration

    Parameters
    ----------
    zx : numpy.ndarray
        The image of derivatives of the depth along the x image dimension

    zy : tuple
        The image of derivatives of the depth along the y image dimension

    pad : float
        The size of the full FFT used for the reconstruction

    Returns
    ----------
    z: numpy.ndarray
        The image, of the same size as the derivatives, of estimated depths
        at each point

    """

    # Raise error if the shapes of the gradients don't match
    if not zx.shape == zy.shape:
        raise ValueError('Sizes of both gradients must match!')

    # Pad the array FFT with a size we specify
    h, w = 512, 512

```

```
    # Fourier transform of gradients for projection
    Zx = np.fft.fftshift(np.fft.fft2(zx, (h, w)))
    Zy = np.fft.fftshift(np.fft.fft2(zy, (h, w)))
    j = 1j

    # Frequency grid
    [wx, wy] = np.meshgrid(np.linspace(-np.pi, np.pi, w),
                           np.linspace(-np.pi, np.pi, h))
    absFreq = wx**2 + wy**2

    # Perform the actual projection
    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        z = (-j*wx*Zx-j*wy*Zy)/absFreq

    # Set (undefined) mean value of the surface depth to 0
    z[0, 0] = 0.
    z = np.fft.ifftshift(z)

    # Invert the Fourier transform for the depth
    z = np.real(np.fft.ifft2(z))
    z = z[:zx.shape[0], :zx.shape[1]]

    return z


def enforceIntegrability(N, s, sig = 3):

    """
    Question 2 (e)

    Find a transform Q that makes the normals integrable and transform them
    by it

    Parameters
    ----------
    N : numpy.ndarray
        The 3 x P matrix of (possibly) non-integrable normals

    s : tuple
        Image shape

    Returns
    -------
    Nt : numpy.ndarray
        The 3 x P matrix of transformed, integrable normals
    """

```

```
    N1 = N[0, :].reshape(s)
    N2 = N[1, :].reshape(s)
    N3 = N[2, :].reshape(s)

    N1y, N1x = np.gradient(gaussian_filter(N1, sig), edge_order = 2)
    N2y, N2x = np.gradient(gaussian_filter(N2, sig), edge_order = 2)
    N3y, N3x = np.gradient(gaussian_filter(N3, sig), edge_order = 2)

    A1 = N1*N2x-N2*N1x
    A2 = N1*N3x-N3*N1x
    A3 = N2*N3x-N3*N2x
    A4 = N2*N1y-N1*N2y
    A5 = N3*N1y-N1*N3y
    A6 = N3*N2y-N2*N3y

    A = np.hstack((A1.reshape(-1, 1),
                   A2.reshape(-1, 1),
                   A3.reshape(-1, 1),
                   A4.reshape(-1, 1),
                   A5.reshape(-1, 1),
                   A6.reshape(-1, 1)))

    AtA = A.T.dot(A)
    W, V = np.linalg.eig(AtA)
    h = V[:, np.argmin(np.abs(W))]

    delta = np.asarray([[-h[2],  h[5], 1],
                        [ h[1], -h[4], 0],
                        [-h[0],  h[3], 0]])
    Nt = np.linalg.inv(delta).dot(N)

    return Nt

def plotSurface(surface, suffix=''):

    """
    Plot the depth map as a surface

    Parameters
    ----------
    surface : numpy.ndarray
        The depth map to be plotted

    suffix: str
        suffix for save file

    Returns
    -------
```

```
        None

    """
    x, y = np.meshgrid(np.arange(surface.shape[1]),
                       np.arange(surface.shape[0]))
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(x, y, -surface, cmap = cm.coolwarm,
                           linewidth = 0, antialiased = False)
    ax.view_init(elev = 60., azim = 75.)
    plt.savefig(f'faceCalibrated{suffix}.png')
    plt.show()
```