16-720 HW6: Photometric Stereo

For each question please refer to the handout for more details.

Programming questions begin at **Q1**. **Remember to run all cells** and save the notebook to your local machine as a pdf for gradescope submission.

Collaborators

List your collaborators for all questions here:

Utils and Imports

Importing all necessary libraries.

```
import numpy as np
from matplotlib import pyplot as plt
from skimage.color import rgb2xyz
import warnings
from scipy.ndimage import gaussian_filter
from matplotlib import cm
from skimage.io import imread
from scipy.sparse import kron as spkron
from scipy.sparse import eye as speye
from scipy.sparse.linalg import lsqr as splsqr
import os
import shutil
```

Downloading the data

```
if os.path.exists('/content/data'):
  shutil.rmtree('/content/data')

os.mkdir('/content/data')
!wget 'https://docs.google.com/uc?export=download&id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0' -O /content/data/data.zip
!unzip "/content/data/data.zip" -d "/content/"
os.system("rm /content/data/data.zip")
data_dir = '/content/data/'
```

```
--2024-04-25 18:10:46--  https://docs.google.com/uc?export=download&id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0
Resolving docs.google.com (docs.google.com)... 108.177.119.102, 108.177.119.100, 108.177.119.101, ...
Connecting to docs.google.com (docs.google.com)|108.177.119.102|:443... connected.
HTTP request sent, awaiting response... 303 See Other
Location: https://drive.usercontent.google.com/download?id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0&export=download [following]
--2024-04-25 18:10:46--  https://drive.usercontent.google.com/download?id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0&export=download
Resolving drive.usercontent.google.com (drive.usercontent.google.com)... 173.194.79.132, 2a00:1450:4013:c05::84
Connecting to drive.usercontent.google.com (drive.usercontent.google.com)|173.194.79.132|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6210854 (5.9M) [application/octet-stream]
Saving to: '/content/data/data.zip'

/content/data/data. 100%[===================>]   5.92M  --.-KB/s    in 0.08s

2024-04-25 18:10:49 (72.8 MB/s) - '/content/data/data.zip' saved [6210854/6210854]

Archive:  /content/data/data.zip
  inflating: /content/data/sources.npy
  inflating: /content/data/input_5.tif
  inflating: /content/data/input_7.tif
  inflating: /content/data/input_6.tif
  inflating: /content/data/input_4.tif
  inflating: /content/data/input_1.tif
  inflating: /content/data/input_2.tif
  inflating: /content/data/input_3.tif
```

Utils Functions.

```python
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')
    ax = fig.add_subplot(111, 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()

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.

    Paramters
    ---------
    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

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

    im = imread(path + 'input_1.tif')
    P = im[:, :, 0].size
    s = im[:, :, 0].shape

    I = np.zeros((7, P))
    for i in range(1, 8):
        im = imread(path + 'input_' + str(i) + '.tif')
        im = rgb2xyz(im)[:, :, 1]
        I[i-1, :] = im.reshape(-1,)

    return I, L, s

def displayAlbedosNormals(albedos, normals, s):

    """
    Question 1 (e)

    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

    albedoIm = albedos.reshape(s)
    normalIm = (normals.T.reshape((s[0], s[1], 3))+1)/2

    plt.figure()
    plt.imshow(albedoIm, cmap = 'gray')

    plt.figure()
    plt.imshow(normalIm, cmap = 'rainbow')

    plt.show()

    return albedoIm, normalIm
```

Q1: Calibrated photometric stereo (75 points)

Q 1 (a): Understanding n-dot-l lighting (5 points)

In the figure, the vector l represents the incident light, the vector v represents the reflected light from the surface, and the vector n represents the surface normal.

In n-dot-l lighting, the dot product between the two vectors n.l = |n|.|l|.cosθ (where θ is the angle between the two vectors).

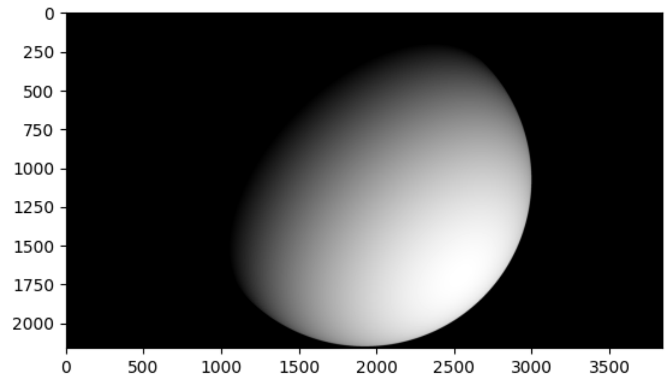
The dot product explains the amount of incident light that falls onto the surface. In the case of θ=0, it means that the light source is directly above the surface, indicating maximum intensity (cos0=1). In the case of θ=90, it means that the light source is parallel to the surface, meaning no loght is incident on the surface (cos90=0).

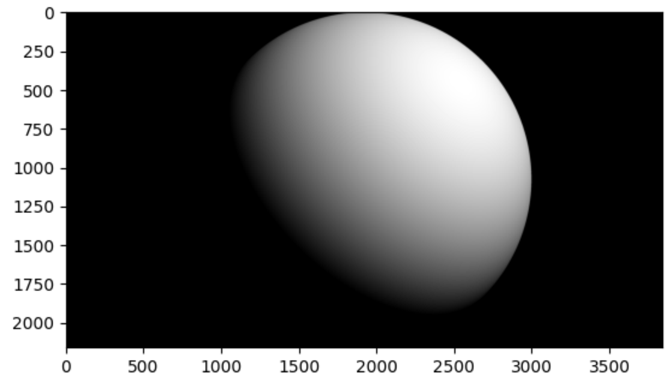
The projected area dA, comes into the equation to represent the amount of foreshortened area that actually receives the illumination from the light source. To calculate the projected area, we project our original area dA onto the perpendicular angle to the light vector l.

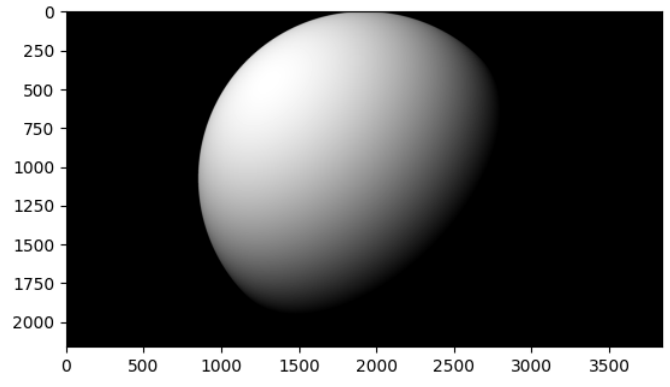
THe viewing direction doesn't matter since the reflected intensity only depends on the angle between the incident light and the surface normal. In the Lambertian model, we assume that light reflects equally in all directions. Therefore, the intensity of reflected light is only dependent on the angle between the normal and the direction of the incident light, not on the direction from which the surface is viewed.

Q 1 (b): Rendering the n-dot-l lighting (10 points)

```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.e-4
    Y = (Y - res[1]/2) * pxSize*1.e-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
    Nx = X / rad
    Ny = Y / rad
    Nz = Z / rad

    intensity = Nx * light[0] + Ny * light[1] + Nz * light[2]
    intensity[intensity < 0] = 0
    intensity[intensity > 1] = 1

    image = intensity
    ### END YOUR CODE

    return image

# 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')
plt.imsave('1b-a.png', image, cmap = 'gray')

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

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

I, L, s = loadData(data_dir)
```







Q 1 (c): Initials (10 points)

```
### YOUR CODE HERE
U, V, Vh = np.linalg.svd(I, full_matrices=False)
### END YOUR CODE
```

```
print(V)
```

```
[79.36348099 13.16260675  9.22148403  2.414729    1.61659626  1.26289066
  0.89368302]
```

Given a 3d coordinate system, we expect the rank of the Initials to be 3 (representing the 3 dimensional space of the pseudonormals)

Although we see 7 non-singular values, we can observe that 3 of the 7 singular values are much larger in magnitude in comparison to the other three. The remaining non-zero singular values may possibly represnt noise in terms of image capturing, as we have more measurements than variables per pixel.

Q 1 (d) Estimating pseudonormals (20 points)

```
def estimatePseudonormalsCalibrated(I, L):

    """
    Question 1 (d)

    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

    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
    B = np.linalg.lstsq(L.T, I, rcond=None)[0]
    ### END YOUR CODE

    return B

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

```
print(B.shape)
```

```
(3, 159039)
```

We can solve the equation I = L.T . B

This is of the form Ax=y, which we can solve through least squares.

Here, Matrix A is the tranpose of the inverse of the Lighting Matrix L. It is a P x 3 matrix as it represents the 3 lighting directions for each pixel of the image

Vector y is the Intensity vector I. It is a P x 1 vector that represents the intensity at each pixel P of the image

We revcover Matrix B, which is the matrix of pseudonormals, representing surface normals and albedos for each pixel. It is a 3 x P matrix

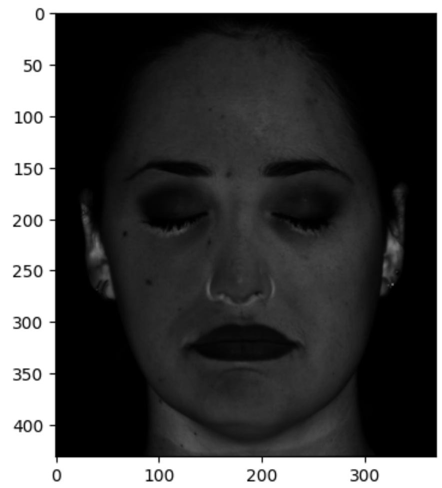
Q 1 (e) Albedos and normals (10 points)

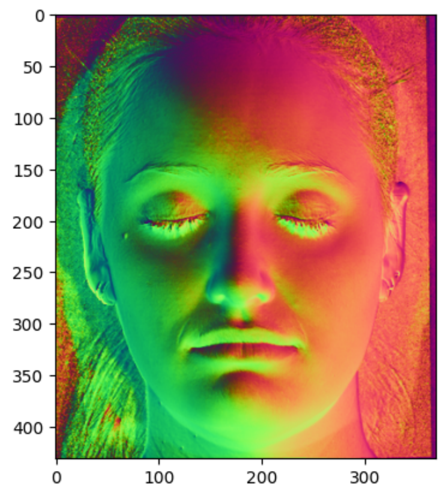
Some regions of the image seem unnaturally bright. This is probably due to the fact that our lighting model assumes constant illumination across the entire surface, but some regions like shadows and occlusions (like the ears and nose and undereyes) may not be receiving the same amount of lighting as compared to other parts of the face. Addiitonally, different regions may have different surface properties.

Thus, the albedos in these regions result in brighter spots compared to the rest of the face.

```
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 = np.linalg.norm(B, axis=0)
    normals = B / albedos
    ### END YOUR CODE

    return albedos, normals
```

```
# Part 1(e)
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')
```





Start coding or generate with AI.

Q 1 (f): Normals and depth (5 points)

Given depth map $z = f(x, y)$

Normal at the point (x,y) be $n = (n1, n2, n3)$

Normal of the surface is given by $(\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)$

Surface can be represented as : $s(x, y) = f(x, y) - z = 0$

So, unit surface normal $n(x, y)$ is given by $(\partial f/\partial z, \partial f/\partial z, -1)$

This is parallel to our normals $(n1, n2, n3)$

Thus,

$(n1, n2, n3) = (\partial f/\partial x, \partial f/\partial y - 1)$

divide by n3 to ensure unit normal

$(n1/n3, n2/n3, 1) = (\partial f/\partial x, \partial f/\partial y - 1)$

As the two are equivalent in magnitude (unit norm), we can conclude that:

$\partial f/\partial x = -n1/n3$

$\partial f/\partial y = -n2/n3$

Q 1 (g): Understanding integrability of gradients (5 points)

Y

gx = [[1 1 1], [1 1 1], [1 1 1], [1 1 1]]

gy = [[4 4 4 4], [4 4 4 4], [4 4 4 4]]

Given g(0,0) = 1

(i) g(1,0) = g(0,0) + gx(0,0) = 1+ 1 =2 g(2,0) = g(1,0) + gx(1,0) = 2+1 =3 g(3,0) = g(2,0) + gx(2,0) = 3+1 =4

g first row = [1 2 3 4]

g(0,1) = g(0,0) + gy(0,0) = 1+4=5

Similarly, we finally get:

g= [[1 2 3 4], [5 6 7 8], [9 10 11 12], [13 14 15 16]]

(i) g(0,1) = g(0,0) + gy(0,0) = 1 + 4 = 5 g(2,0) = g(1,0) + gy(1,0) = 5 + 4 = 9 g(3,0) = g(2,0) + gy(2,0) = 9 + 4 = 13

g first row = [1 5 9 13]

g(1,1) = g(0,0) + gx(0,0) = 1+1 =2

Similarly, we finally get:

g= [[1 2 3 4], [5 6 7 8], [9 10 11 12], [13 14 15 16]]

Yes. We see that they're the same.

We can modify g to ensure gx and gy are non-integrable by introducing noise along a particular axis (either x or y). By doing so, we observe that the depth map g becomes non-integrable. For example, changing g from:

g= [[1 2 3 4], [5 6 7 8], [9 10 11 12], [13 14 15 16]]

to

g= [[1 2 3 4], [1 2 3 4], [9 10 11 12], [13 14 15 16]]

We are only introducing noise along one direction ultimately leading to two different calculations of g when we test both methods.

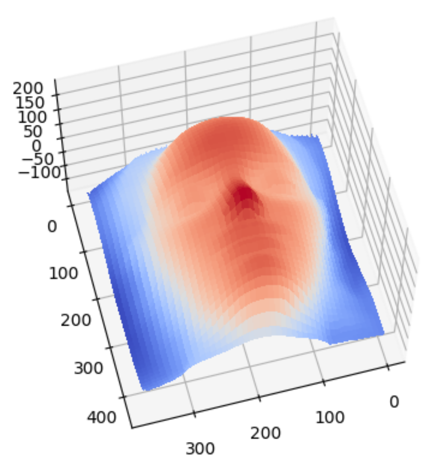
Additionally, we could introduce random noise into the data, or introduce discontinuity to a portion of our surface by setting the values of certain cells to 0.

The gradients estimates may be non-integrable due to various factors. Some of them include:

- Discontinuities in the surface
- Noisy estimates of surface gradients which can occur due to either:
    - Noise due to Image Intensity measurements
    - Incorrect reflectance assumptions
    - Issues such as interreflections

Q 1 (h): Shape estimation (10 points)

```
def estimateShape(normals, s):

    """
    Question 1 (h)

    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
    zx = np.reshape(normals[0, :]/(-normals[2, :]), s)
    zy = np.reshape(normals[1, :]/(-normals[2, :]), s)
    surface = integrateFrankot(zx, zy)
    ### END YOUR CODE

    return surface


# Part 1(h)
surface = estimateShape(normals, s)
plotSurface(surface)
```



Q2: Uncalibrated photometric stereo (50 points)

Q 2 (a): Uncalibrated normal estimation (10 points)

$I = L^T . B$

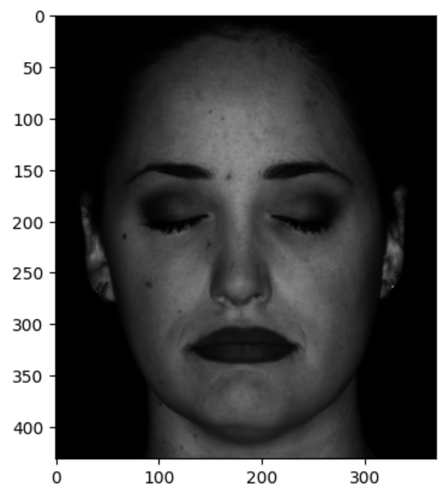
The rank of matrix I should be 3. However, it actually has a rank of 7. Thus, our estimated Î should have rank 3

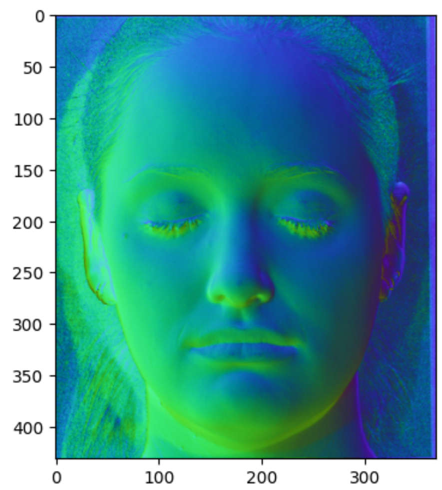
We can do this by:

- Performing SVD of recovered I to obtain U, V, S
- Take the first 3 columns of U and V, and the highest 3 singular values S.
- Recover B and L by:
    - B: multiply the $newV\ t$ and $singularvalues^{1/2}$
    - L: multiply the $newU$ and $singularvalues^{1/2}$

Q 2 (b): Calculation and visualization (10 points)

```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

    """

    B = None
    L = None

    ### YOUR CODE HERE

    u, s, vt = np.linalg.svd(I, full_matrices=False)
    s[3:] = 0
    s3 = np.diag(s[:3])
    vt3 =vt[:3,:]
    B = np.dot(np.sqrt(s3),vt3)
    L = np.dot(u[:,:3],np.sqrt(s3)).T
    ### END YOUR CODE

    return B, L


# Part 2 (b)
I, L, s = loadData(data_dir)
B, LEst = estimatePseudonormalsUncalibrated(I)
albedos, normals = estimateAlbedosNormals(B)
albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
plt.imsave('2b-a.png', albedoIm, cmap = 'gray')
plt.imsave('2b-b.png', normalIm, cmap = 'rainbow')
```





Q 2 (c): Comparing to ground truth lighting

No, we see that the Lighting matrices are indeed different.

A simple change to part (a) that we can make is:

- B: multiply the $newVt$ and $singularvalues$
- L: recover from the $newU$

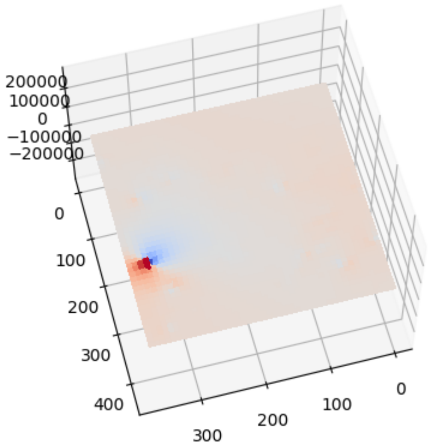
```python
print(L)
print(LEst)
print("--")
print(L-LEst)
```

```
[[-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]]
[[-2.99267472 -3.86998525 -2.40803005 -3.74500806 -3.59135539 -3.38666635
  -3.3525448 ]
 [ 0.94780484 -2.31708946  0.49911094 -0.62599426  2.32568155  0.46605103
  -0.79271078]
 [ 1.87934697  1.01461663  0.42942606 -0.01730299 -0.3107729  -0.91273581
  -1.8830081 ]]
--
[[ 2.85087472  3.99148525  2.33903005  3.81200806  3.42865539  3.38666635
   3.5003448 ]
 [-1.12820484  2.11448946 -0.53361094  0.58579426 -2.20368155 -0.34665103
   0.91361078]
 [-2.80604697 -1.98631663 -1.26742606 -0.95989701 -0.6682271  -0.05206419
   0.9117081 ]]
```

Q 2 (d): Reconstructing the shape, attempt 1 (5 points)

No, it does not look like a face

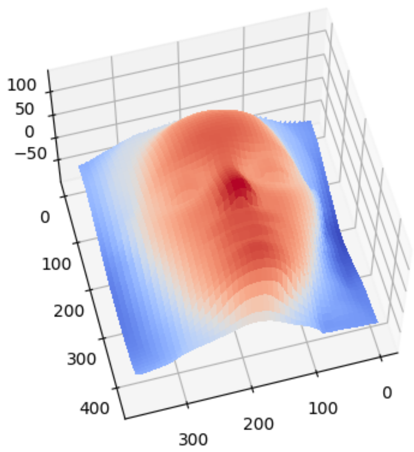
```
# Part 2 (d)
### YOUR CODE HERE
surface = estimateShape(normals, s)
plotSurface(surface)
### END YOUR CODE
```



Q 2 (e): Reconstructing the shape, attempt 2 (5 points)

Yes, this looks a lot closer to the output from calibrated photometric stereo

```
# Part 2 (e)
# Your code here
### YOUR CODE HERE
Bi = enforceIntegrability(B, s)
albedos, normals = estimateAlbedosNormals(Bi)
surface = estimateShape(normals, s)
plotSurface(surface)
### END YOUR CODE
```



Q 2 (f): Why low relief? (5 points)

It is called low-relief to represent the ambiguity in depth or relief that we face when reconstructing the scene from a 2D image. It means that we relatively have very ambiguous and minimal depth information when we are given no information regarding light direction and a single viewpoint only

From my observations, I beleibe these are the effects of the three parameters:

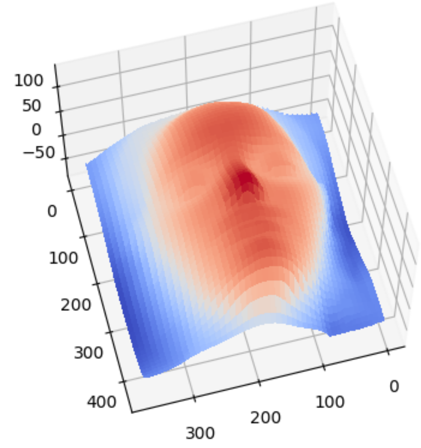
1) mu: this seems to affect the overall extent of the tilt and orientation of the surface along the x axis

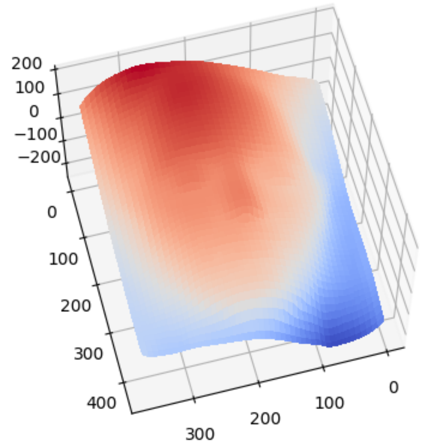
2) nu: this affects the tilt and orientation of the surface but along the y axis

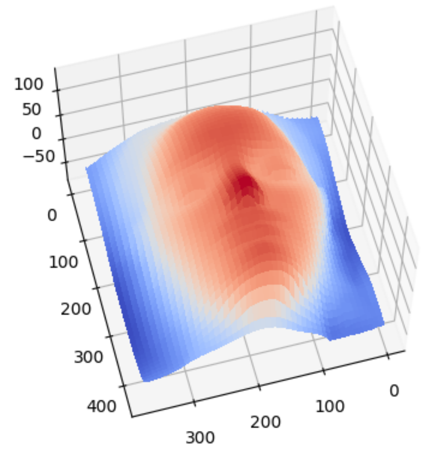
3) lambda: it seems to capture the extent of detail that is able to be captured. as lambda increases, the scale of the relief increases. When lambda is negative, the relief is captured in the opposite direction

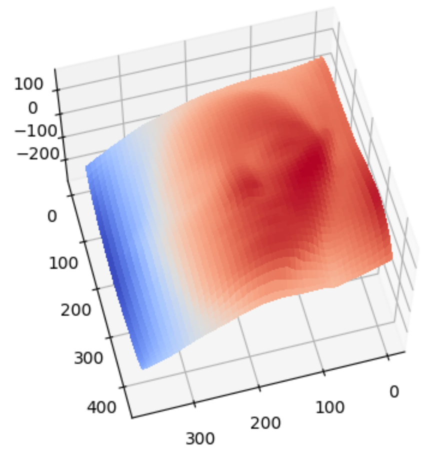
```
def plotBasRelief(B, mu, nu, lam):

    """
    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

    """
    G = np.asarray([[1, 0, -mu/lam],
            [0, 1, -nu/lam],
            [0, 0,   1/lam]])
    Bp = G.dot(B)
    surface = estimateShape(Bp, s)
    plotSurface(surface, suffix=f'br_{mu}_{nu}_{lam}')

# keep all outputs visible
from IPython.display import Javascript
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 5000})'''))

# Part 2 (f)
### YOUR CODE HERE\

#plotBasRelief(Bi, 0.5, -10, 1)
#plotBasRelief(Bi, 0.5, -5, 1)
#plotBasRelief(Bi, 0.5, -0.5, 1)
plotBasRelief(Bi, 0.5, 0.5, 1)
plotBasRelief(Bi, 0.5, 5, 1)
#plotBasRelief(Bi, 0.5, 10, 1)

#plotBasRelief(Bi, -10, 0.5, 1)
#plotBasRelief(Bi, -5, 0.55, 1)
#plotBasRelief(Bi, -0.5, 0.5, 1)
plotBasRelief(Bi, 0.5, 0.5, 1)
```

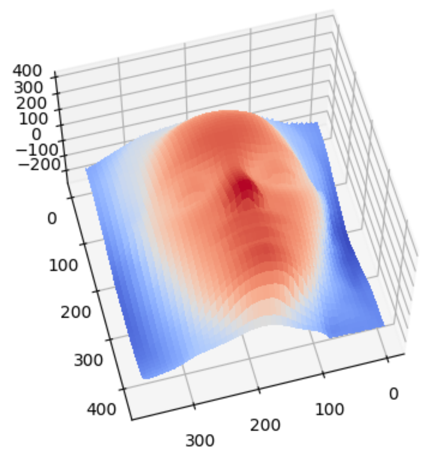
```
plotBasRelief(Bi, 5, 0.5, 1)
#plotBasRelief(Bi, 10, 0.5, 1)

plotBasRelief(Bi, 0.5, 0.5, 3)
#plotBasRelief(Bi, 0.5, 0.5, 9)
plotBasRelief(Bi, 0.5, 0.5, 12)
#plotBasRelief(Bi, 0.5, 0.5, 20)
#plotBasRelief(Bi, 0.5, 0.5, 50)
#plotBasRelief(Bi, 0.5, 0.5, -1)
#plotBasRelief(Bi, 0.5, 0.5, -5)

### END YOUR CODE
```

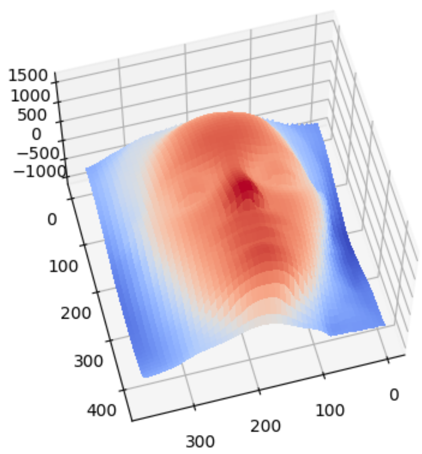










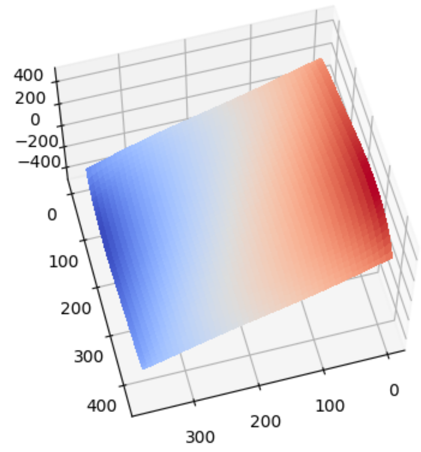


Q 2 (g): Flattest surface possible (5 points)

To get the flattest surface possible, we would have to set lambda to be very close to 0 to ensure as little relief is captured as possible. We would also want to set either mu or nu to be high along and the other one to be low so as to tilt the surface along one axis to minimize the perception of depth

---

```
plotBasRelief(Bi, 10, 0.1, 0.01)
```



Q 2 (h): More measurements

Increasing the number of can potentially help us estimate a more accuracate psuedonormal matrix B and lighting matrix L. However, this can come at a cost of increased noise in the image, leading to noisier estimates.

Additionally, there is still ambuguity in factorization of I into B and L which doesn't get relieved through more images necessarily.

There remains a fundamental ambiguity for uncalibrated photometric stereo that cannot be fixed through increasing the number of pictures.

It would be extremely useful to have images from more lighting sources in the case of calibrated sterei.

```
def plotBasRelief(B, mu, nu, lam):

    """
    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
```