CV HW5

Lifan Yu (lifany)

November 2023

Question 1

a

n-dot-l model

For diffuse surface, the Lambertian Bidirectional Distribution Function is simply a constant $\frac{\rho_d}{\pi}$

Surface radiance (light going out, pixel value) is

$$L = \frac{\rho_d}{\pi} I_0 cos\theta_i = \frac{\rho_d}{\pi} I_0 \overrightarrow{n} \cdot \overrightarrow{l}$$

Where ρ_d is the albedo, I_0 is the intensity of incoming light, θ_i is the angle between the incoming light direction and the surface normal. $\overrightarrow{n}$ is the surface normal, and $\overrightarrow{l}$ is the direction of the incoming light.

projected area

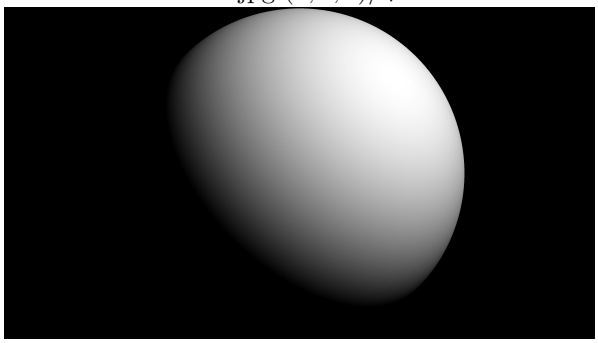
In a light cone, when light is hitting onto a surface at an angle, the area of light on the surface is greater, and the light intensity per unit area is smaller, scaled by $cos\theta_i$ above.

viewing direction does not matter

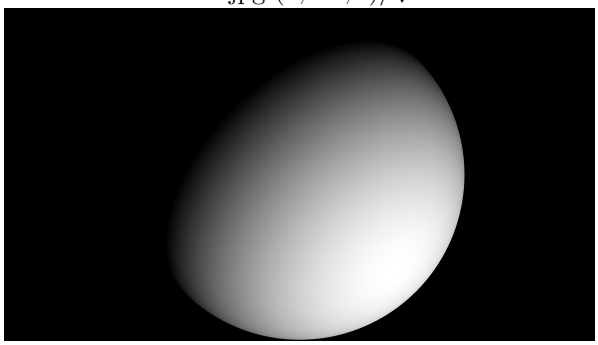
Because we are assuming an diffuse reflection surface, the surface appears equally bright in all directions.

b

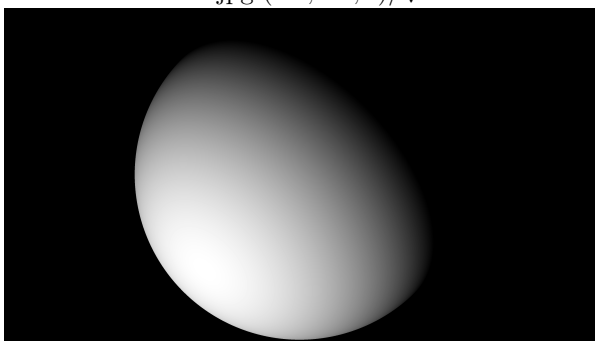
1b-a.jpg $(1, 1, 1)/\sqrt{3}$



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



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



```python
def renderNDotLSphere(center, rad, light, pxSize, res):
    # https://stackoverflow.com/questions/67867707/make-a-sphere-from-a-circle-of-dots
    [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)

    # Your code here
    image = X * light[0] - Y * light[1] + Z * light[2]
    image = np.clip(image, 0, None)

    return image
```

c

```
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

    s: tuple
        Image shape

    """
    # Your code here
    I = []
    for i in range(1, 8):
        im_name = "input_" + str(i) + ".tif"
        # preserve bit depth while reading, datatype should be uint16
        im0 = skimage.io.imread(fname = path + im_name).astype(np.uint16)
        # Y channel (channel 1) is the luminance
        im = skimage.color.rgb2xyz(im0)[:, :, 1].flatten()
        I.append(im)
    I = np.array(I)

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

    s = im0.shape[:2]

    return I, L, s
```

d

With $I = L^T B$, and L (3 x 7), lightsource directions, B (3 x P) surface pseudonormal stacked.

$$L^T = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \dots & \dots & \dots \\ x_7 & y_7 & z_7 \end{bmatrix}$$

The pseudonormals are unknowns

$$B = \begin{bmatrix} X_1 & X_2 & \dots & X_P \\ Y_1 & Y_2 & \dots & Y_P \\ Z_1 & Z_2 & \dots & Z_p \end{bmatrix}$$

$$I = L^T B = \begin{bmatrix} x_1X_1 + y_1Y_1 + z_1Z_1 & \dots & x_1X_P + y_1Y_Pz_1Z_P \\ \dots & & \\ x_7X_1 + y_7Y_1 + z_7Z_1 & \dots & x_7X_P + y_7Y_P + z_7Z_P \end{bmatrix}$$

Rank is the dimension of the vector space generated (or spanned) by its columns. Because for each pixel, we have 3 unknowns (X_i, Y_i, Z_i), perpendicular to each other, the Rank of matrix I should be 3.

This way, theoretically if we have accurate 3 measurements, we can solve for the i-th pixel's pseudonormal (X, Y, Z) using the i-th column in I matrix.

However, using numpy's SVD, we got the following singular values, and there are 7 intotal.

```
[142.4639219   23.56050042  16.49253875   4.32672882   2.92286927   2.30343506   1.63375072]
```

This is because we have errors in each measurement, the lights also reflect between the object surface and the environment. For each measurement we have a different result independent from the others, and cannot have for a completely accurate pseudonormal for each pixel. This way 7 Measurements gives rank 7.

```
# Part 1(d)
    # Your code here
    I, L, s = loadData()
    U, Sigma, VT = np.linalg.svd(I, full_matrices=False)
    print("1(d) Sigma: ", Sigma)
```

e

We solve B from the expression

$$L^T B = I$$

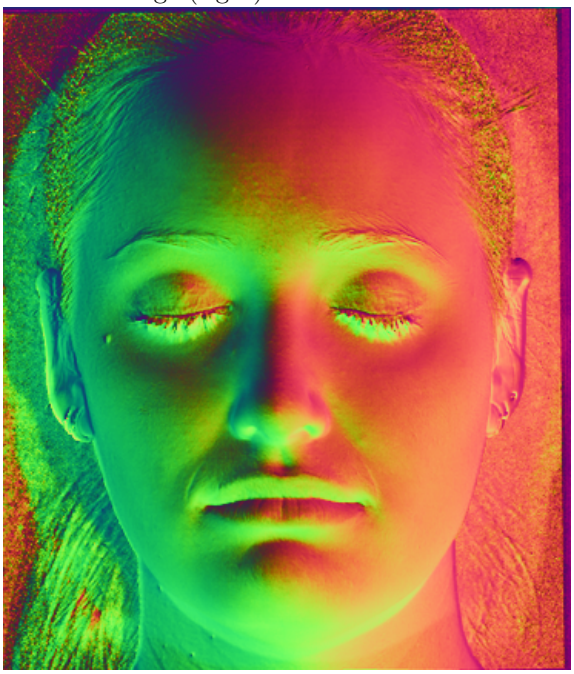
Let $A = L^T, x = B, y = I$
We solve $Ax = y$ using least-squares

```
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

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

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

    B = np.linalg.lstsq(L.T, I)[0]
    # Your code here
    return B
```

```
B =
B =  [[-8.21469909e-04 -9.56470429e-04 -9.13452434e-04 ... -1.44086039e-04
  -3.22785787e-04 -2.21082676e-04]
 [-7.60629095e-04 -8.84245543e-04 -8.49388028e-04 ... -2.30568653e-03
  -2.08288358e-03 -1.89259199e-03]
 [-6.91239871e-05 -7.98400935e-05 -7.63779155e-05 ... -2.59909474e-04
  -1.86110847e-04 -1.67896179e-04]]
```

f

Albedo image (left) and normal image (right)

The left albedo image does not actually match one's expectation of a face, because the regions with sharper changes in gradients, or more ups and downs on the surface, such as the ears, lower eye lid, and the lower part of the nose and the upper neck are too bright, while in real life they should have a darker shade. Lights are reflected from these areas onto their nearby surfaces more than other areas, which failed to be captured by the camera. That is part of the reason why the measurement is not accurately an n-dot-l model.

The right normal image matches one's expectation of a face's shape. The depth of the face is relatively accurate.

```
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 = np.linalg.norm(B, axis = 0)
    normals = B / albedos
    # Your code here
    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

    """
    # Your code here
    albedoIm = albedos.reshape(s)
    normalIm = normals.T.reshape(s[0], s[1], 3)
    scale = np.max(normalIm) - np.min(normalIm)
    normalIm = (normalIm - np.min(normalIm)) / scale
    #normalIm = np.clip(normalIm, 0, 1)
    return albedoIm, normalIm
```

g

why n is related to the partial derivatives of f at (x, y)

$$f_x = \frac{\partial f(x,y)}{\partial x}$$

$$f_y = \frac{\partial f(x,y)}{\partial y}$$

$$\partial z = \partial f(x,y) = f_x \partial x$$

$$\partial z = \partial f(x,y) = f_y \partial y$$

For simplicity, let $\partial x = \partial y = 1$
Therefore, we have tangent vectors to the surface

$$v_1 = (1, 0, f_x), v_2 = (0, 1, f_y)$$

We take the cross product of them and get the normal vector

$$\overrightarrow{N} = (-f_x, -f_y, 1)$$

Unit normal vector n

$$\overrightarrow{n} = \frac{(-f_x, -f_y, 1)}{\sqrt[3]{f_x^2 + f_y^2 + 1}}$$

$$\sqrt[3]{f_x^2 + f_y^2 + 1} * (n_1, n_2, n_3) = (-f_x, -f_y, 1)$$

We now have

$$\frac{n_1}{n_2} = -f_x, \frac{n_2}{n_2} = -f_y$$

Therefore,

$$f_x = -\frac{n_1}{n_2}$$

$$f_y = -\frac{n_2}{n_3}$$

h

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

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

1.

First row

$$row1 = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

Row 234

$$row234 = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Putting together the calculated rows we get the original g.

2.

First column

$$col1 = \begin{bmatrix} 1 \\ 5 \\ 9 \\ 13 \end{bmatrix}$$

Columns 234

$$col234 = \begin{bmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \\ 10 & 11 & 12 \\ 14 & 15 & 16 \end{bmatrix}$$

Putting together the columns we get the original g.

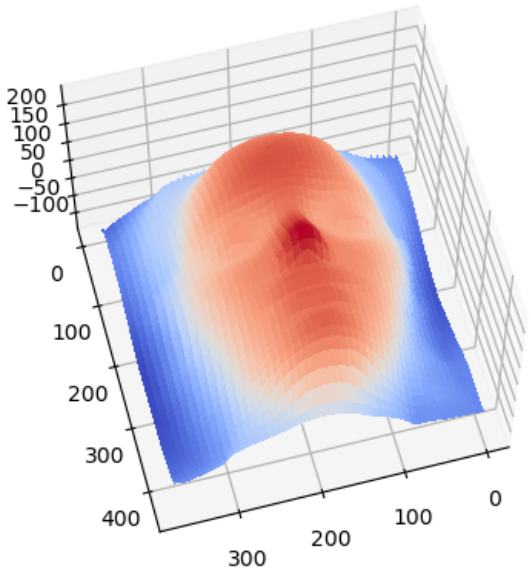
Both calculations gave the same result.

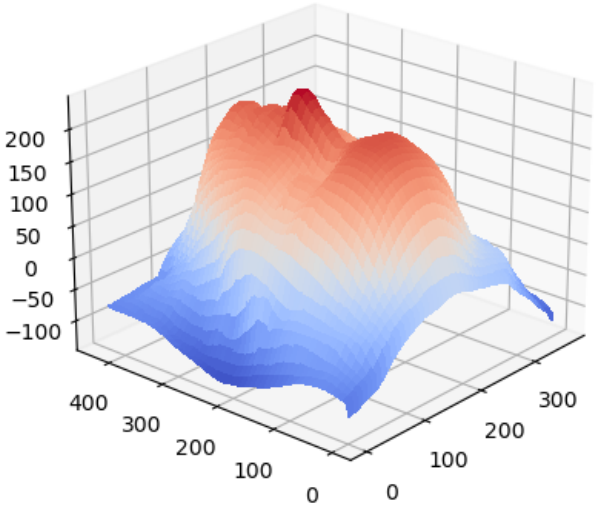
How can we modify the gradients you calculated above to make gx and gy non-integrable?

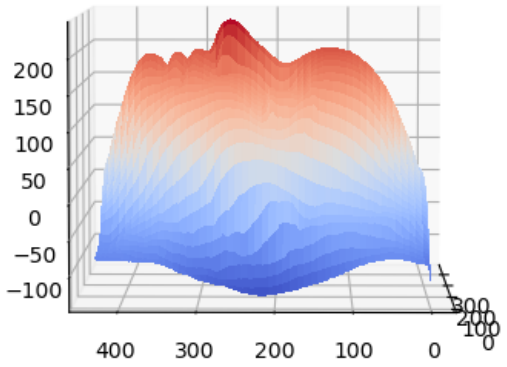
-

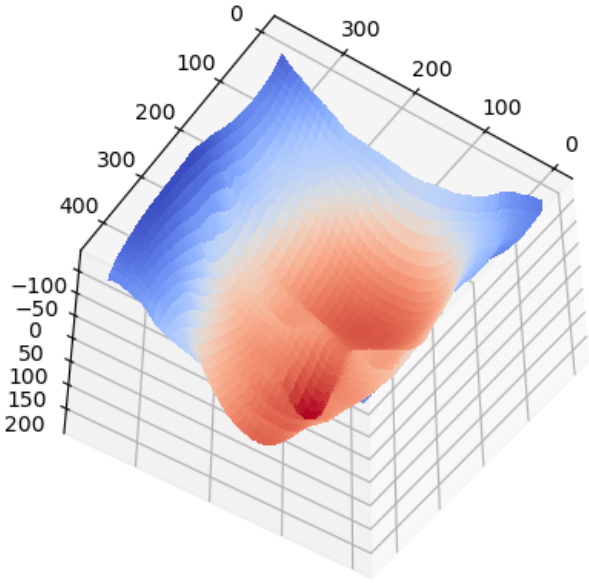
- change certain values in the gradient matrix so they are not all the same
- flip signs of the gradient matrix elements so addition and subtractions do not happen in the same way

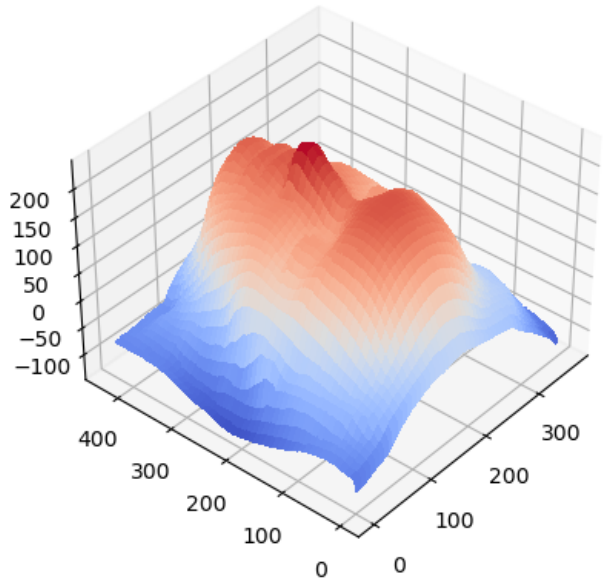
i


200
150
100
50
0
−50
−100
0
100
200
300
400
300
200
100
0


200
150
100
50
0
−50
−100
400
300
200
100
0
0
100
200
300


200
150
100
50
0
−50
−100
400
300
200
100
0
300
200
100
0


0
100
200
300
400
300
200
100
0
−100
−50
0
50
100
150
200




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

    # Your code here
    fx = - normals[0, :] / normals[2, :]
    fy = - normals[1, :] / normals[2, :]
    fx = fx.reshape(s)
    fy = fy.reshape(s)
    surface = integrateFrankot(fx, fy)

    return surface
```

Question 2

a

With SVD decomposition, we can decompose I as

$$I = U\Sigma V^T$$

Without knowing the exact value, we can only estimate $\hat{I}$ from $\hat{L^T}\hat{B}$

$$\hat{I} = \hat{L^T}\hat{B}$$

Because we want only the 3 largest singular value in order to construct an I with rank 3, we set all other singular values in Σ to zero and get $\hat{\Sigma}$

$$\hat{I} = U\hat{\Sigma}V^T = U\begin{bmatrix} s_1 & 0 & 0 & ... \\ 0 & s_2 & 0 & ... \\ 0 & 0 & s_3 & ... \\ ... & & & 0 \end{bmatrix} V^T = U\begin{bmatrix} \sqrt{s_1} & 0 & 0 & ... \\ 0 & \sqrt{s_2} & 0 & ... \\ 0 & 0 & \sqrt{s_3} & ... \\ ... & & & 0 \end{bmatrix}\begin{bmatrix} \sqrt{s_1} & 0 & 0 & ... \\ 0 & \sqrt{s_2} & 0 & ... \\ 0 & 0 & \sqrt{s_3} & ... \\ ... & & & 0 \end{bmatrix} V^T$$

We now have

$$\hat{L}^T = U\begin{bmatrix} \sqrt{s_1} & 0 & 0 & ... \\ 0 & \sqrt{s_2} & 0 & ... \\ 0 & 0 & \sqrt{s_3} & ... \\ ... & & & 0 \end{bmatrix} = U[:, :3]\begin{bmatrix} \sqrt{s_1} & 0 & 0 \\ 0 & \sqrt{s_2} & 0 \\ 0 & 0 & \sqrt{s_3} \end{bmatrix}$$

$$\hat{B} = \begin{bmatrix} \sqrt{s_1} & 0 & 0 & ... \\ 0 & \sqrt{s_2} & 0 & ... \\ 0 & 0 & \sqrt{s_3} & ... \\ ... & & & 0 \end{bmatrix} V^T = \begin{bmatrix} \sqrt{s_1} & 0 & 0 \\ 0 & \sqrt{s_2} & 0 \\ 0 & 0 & \sqrt{s_3} \end{bmatrix} V^T[:3, :]$$

(Take the first 3 solumns of U and first 3 rows of V^T)

b

```
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

    """

    # Your code here
    U, Sigma, VT = np.linalg.svd(I, full_matrices=False)
    U = U[:, :3]
    VT = VT[:3, :]
    Sigma_sqrt = np.diag(np.sqrt(Sigma[:3]))
    L = np.matmul(U, Sigma_sqrt).T
    B = np.matmul(Sigma_sqrt, VT)
    # L = np.matmul(U, np.diag(Sigma[:3])).T
    # B = VT
    return B, L
```

```
if __name__ == "__main__":

    # Part 2 (b)
    # Your code here
    I, L, s = loadData("../data/")
    B, L = estimatePseudonormalsUncalibrated(I)

    albedos, normals = estimateAlbedosNormals(B)
    albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
    plt.imsave("2a-a.png", albedoIm, cmap="gray")
    plt.imsave("2a-b.png", normalIm, cmap="rainbow")
```

c

Original L

```
[[-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]]
```

L estimated

```
[[-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 ]]
```

They are very different.

Instead of taking the square roots of values in Sigma matrix, we can directly use it from the SVD decomposition. This way, we have

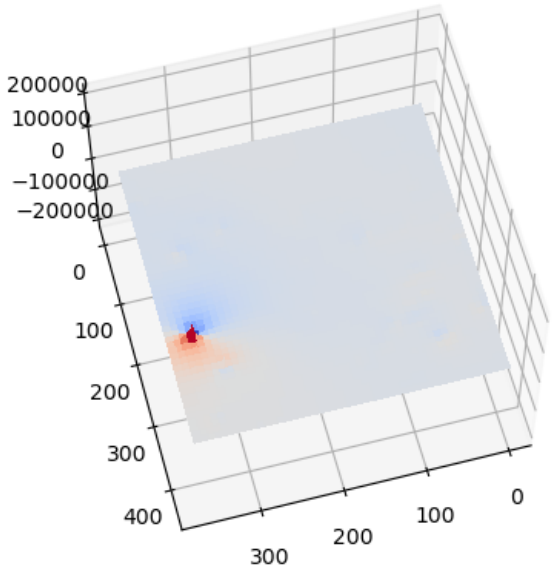
$$\hat{L}^T = U[:, :3]$$

$$\hat{B} = \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{bmatrix} V^T[:3, :]$$

This method keeps the images rendered using them the same using only the matrices you calculated during the singular value decomposition (U/S/V).

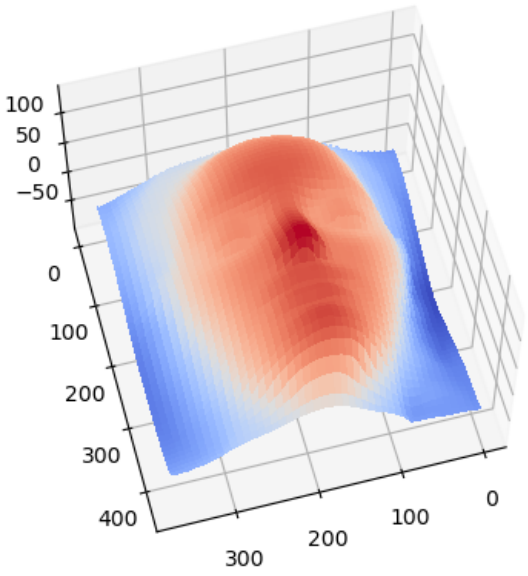
d

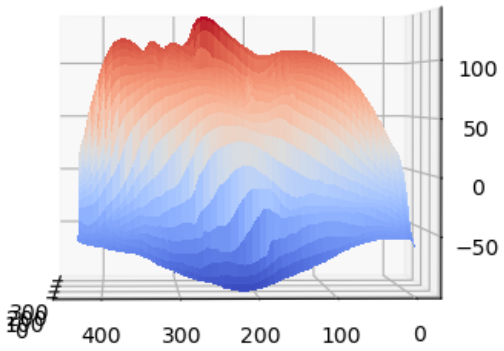
The result of this depth map from estimated pseudonormals does not look like a face.

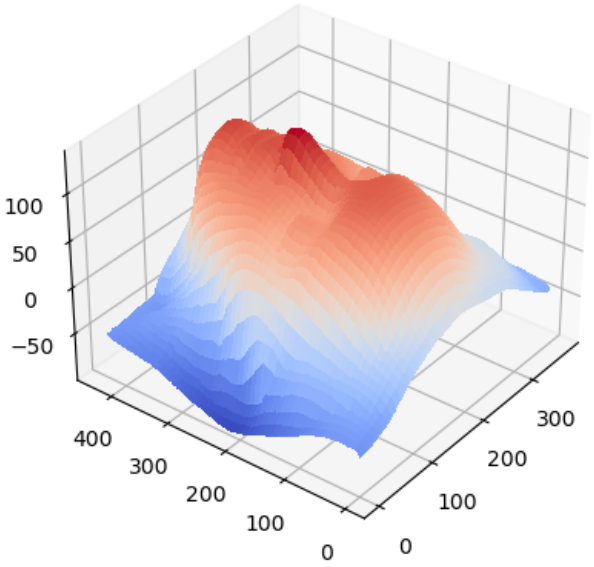


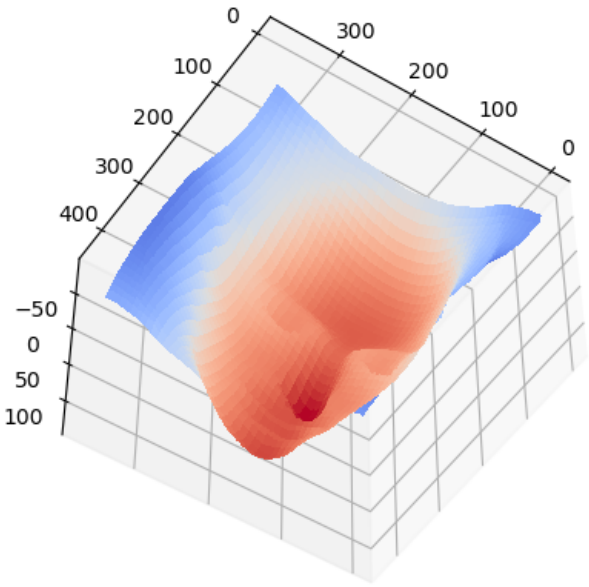

```
# Part 2 (d)
# Your code here
surface = estimateShape(normals, s)
plotSurface(surface)
```

e


100
50
0
−50
0
100
200
300
400
300
200
100
0


100
50
0
−50
400
300
200
100
0






```
# Part 2 (e)
# Your code here
Bt = enforceIntegrability(B1, s)
# repeat (b)
albedos1, normals1 = estimateAlbedosNormals(Bt)
# repeat (d)
surface_f = estimateShape(normals1, s)
plotSurface(surface_f)
```

f

Why bas-relief ambiguity is so named?

When calculating the depth, there is ambiguity on how much depth the object itself has, how much a side of the object is tilted. This is due to the limited number of viewing angles or limited measurement accuracy. We can adjust these shape magnitudes by setting some parameters.

How the three parameters affect the surface

λ: When > 0, the face is upward. When < 0, the face is downward. Cannot $= 0$ because the matrix needs to be inverted and diagonal entries should not be zero. When we scale λ, the overall depth of the face scales accordingly.

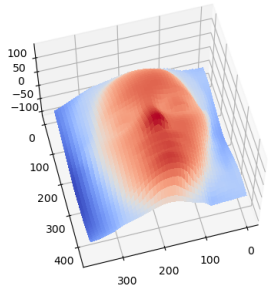
μ: It smoothens and stretches the left side of the face.How much and to which direction it magnifies the z depth values depend on its value and sign.

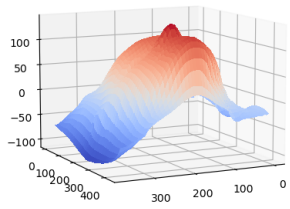
ν: It smoothens and stretches the lower right area of the face. How much and to which direction it magnifies the z depth values depend on its value and sign.

If μ, ν have different signs and similar magnitude, the surface is more smooth.

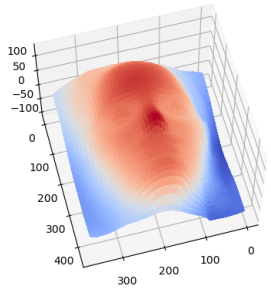
$\boxed{\lambda > 0}$

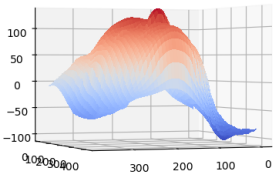
(1) $\lambda = 1, \mu = 1, \nu = 0$



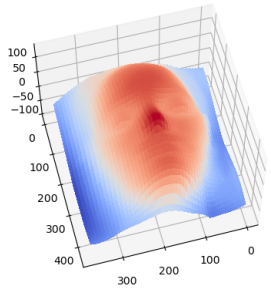


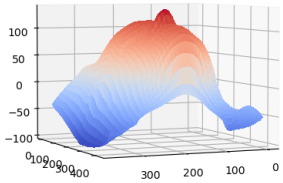
(2) $\lambda = 1, \mu = 0, \nu = 1$





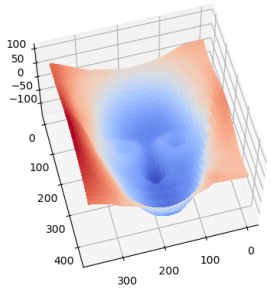
(3) $\lambda = 1, \mu = 1, \nu = 1$

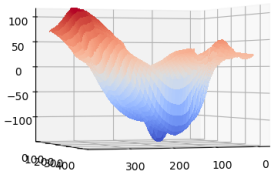




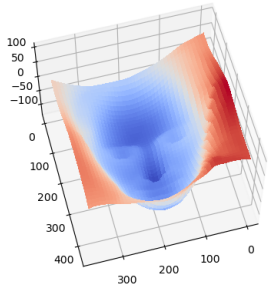
$\boxed{\lambda < 0}$

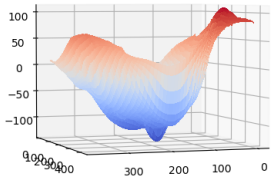
(4) $\lambda = -1, \mu = -1, \nu = 0$



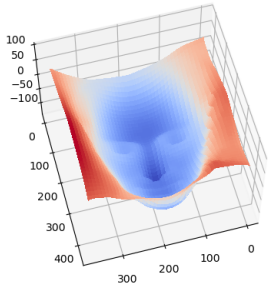


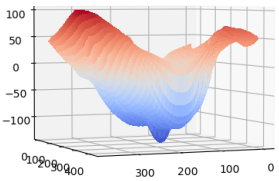
(5) $\lambda = -1, \mu = 0, \nu = -1$





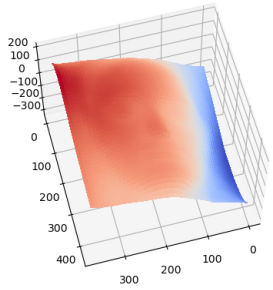
(6) $\lambda = -1, \mu = -1, \nu = -1$

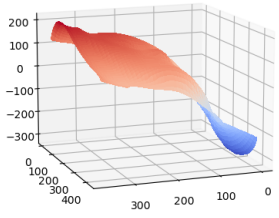




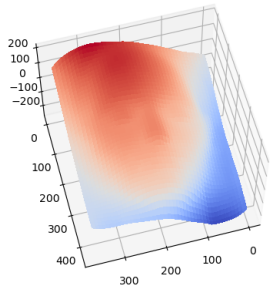
Scaling

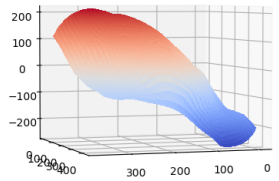
(7) $\lambda = 1, \mu = -5, \nu = 0$



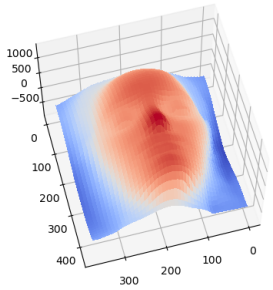


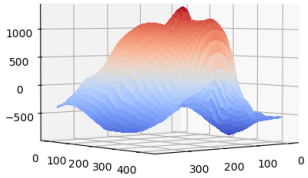
(8) $\lambda = 1, \mu = 0, \nu = 5$





(9) $\lambda = 10, \mu = 1, \nu = 1$, depth range of the face changed from (-100, 100) in previous examples to (-500, 1000).





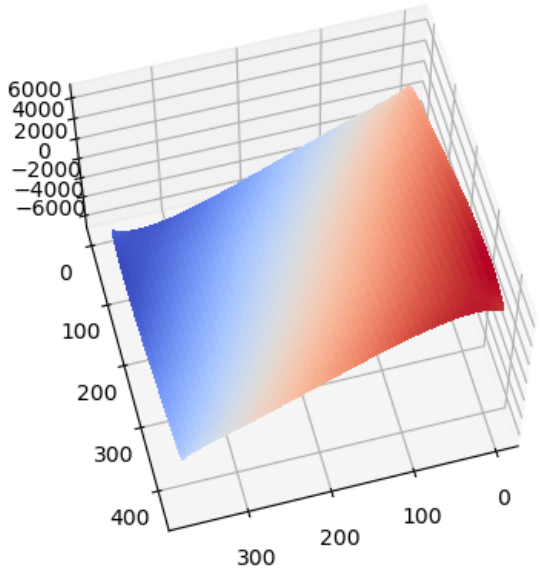
```
def plotBasRelief(B, mu, nu, lam, s):
    """
    Question 2 (f)

    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
    G = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [mu, nu, lam]])
    Bt = enforceIntegrability(B, s)
    B_bas = np.matmul(np.linalg.inv(G).T, Bt)
    albedos2, normals2 = estimateAlbedosNormals(B_bas)
    albedoIm, normalIm = displayAlbedosNormals(albedos2, normals2, s)
    # repeat (d)
    surface2 = estimateShape(normals2, s)
    plotSurface(surface2)
```

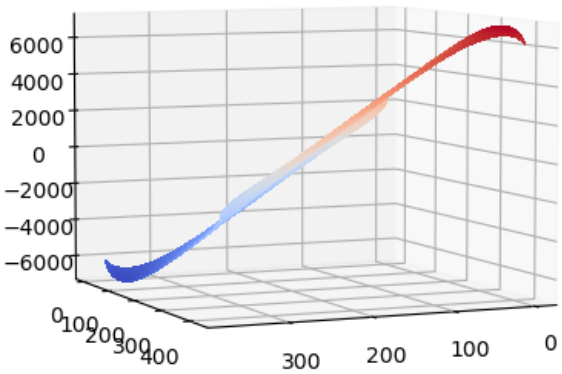
```
# Part 2 (f)
# Your code here
mus = [1, 0, 1, -1, 0, -1, 5, 0, 1]
nus = [0, 1, 1, 0, -1, -1, 0, 5, 1]
lams = [1, 1, 1, -1, -1, -1, 1, 1, 10]
for i in range(len(mus)):
    plotBasRelief(B1, mus[i], nus[i], lams[i], s)
```

g

The bas-relief equation changes the depth of different parts of the face. The final depth value of the pseudonormals x', y', z' are calculated from $z' = (1/\lambda) * z$, $x' = x + (1/\lambda) * \mu * z$, $y' = y + (1/\lambda) * \nu * z$. To smoothen the image, we want to increase the magnitude difference between depth z and the coordinates (x, y). To achieve this, we can choose a small λ with large μ, ν (μ and ν have opposite signs). With large (x, y) and small z, after some normalizing calculations, this huge difference in their magnitude will lead to very small difference in depth z across the depth image.

The below images show the results with $\lambda = 0.1, \mu = 100, \nu = -100$. In this case $G^{-T} = \begin{bmatrix} 1 & 0 & -1000 \\ 0 & 1 & 1000 \\ 0 & 0 & 10 \end{bmatrix}$.





h

No, increasing the observations might not help. With more measurements there will be more degrees of freedom in the B pseudonormal matrix to solve. This is unavoidable as we have noises from the lights reflecting back to the face and lights reflecting from the surrounding environment. We still need to estimate and take the 3 largest singular values from I using SVD. Also, too large matrices might be computationally expensive. Theoretically 3 measurements from different lightings are sufficient, but more measurements can still help to lower the chance of a really bad measurement affecting the whole calculation.