1.a

$\vec{l}$ = incident direction

$\vec{n}$ = normal

$\vec{v}$ = viewing direction

dA = surface element

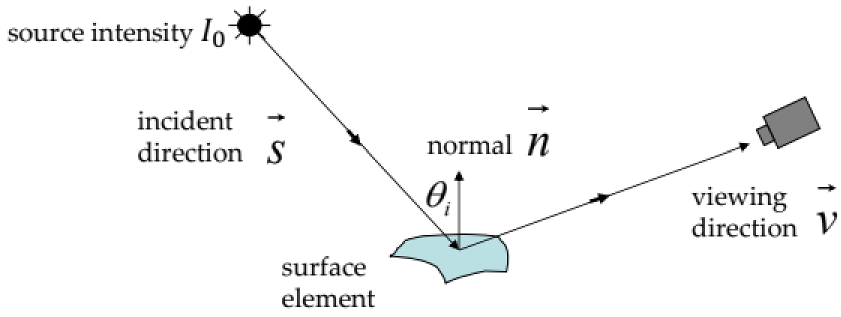


Fig (a)

For n-dot-l lighting model, Surface Radiance is:

$$L = \frac{\rho_d}{\pi} I_0 \cos \theta_i$$

where

$$\cos \theta_i = \vec{n}.\vec{s}$$

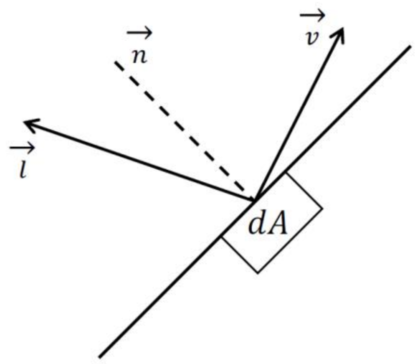
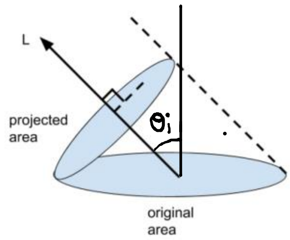


Fig (b)

$\vec{s}$ from fig (a) corresponds to $\vec{l}$ from fig (b). Therefore,

$$\cos \theta_i = \vec{n}.\vec{l}$$

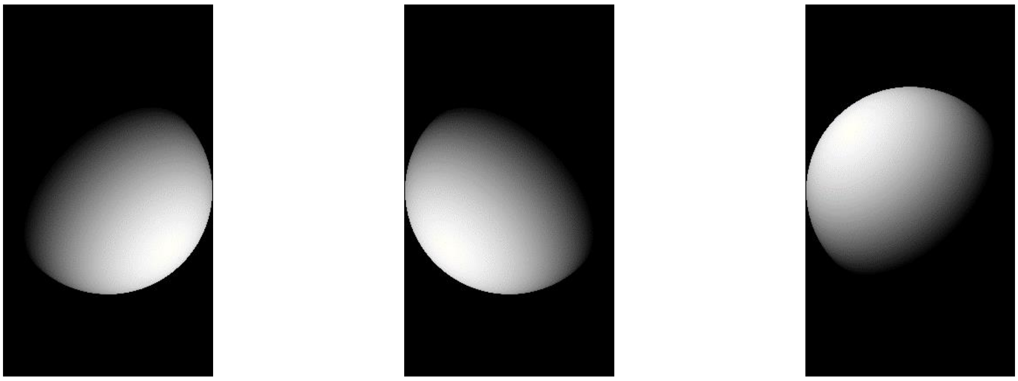


I_0 is source intensity and it depends on the projected area. The relationship between projected area and original area can be given by

$$\text{Original area} = \text{Projected area X } cos\ \theta_i$$

Viewing direction does not matter as surface appears equally bright from all directions, lambertian surface

1.b



```python
def renderNDotLSphere(center, rad, light, pxSize, res):

    [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
    image = np.zeros((res[1], res[0]))
    for i in range(res[1]):
        for j in range(res[0]):
            normal = np.array([X[i, j], Y[i, j], Z[i, j]])
            normal /= (np.linalg.norm(normal) + 1.0e-10)
            image[i, j] = np.dot(normal, light)
    image[image < 0] = 0
    image = (image - np.min(image)) / (np.max(image) - np.min(image))
    return image
```

1.c

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

    I = None
    L = None
    s = None
    # Your code here
    for i in range(7):
        image = plt.imread(path + "input_" + str(i + 1) + ".tif")

        # make sure datatype is uint16
        image = image.astype(np.uint16)

        # convert to xyz
        image = rgb2xyz(image)

        # extract luminance
        luminance = image[:, :, 1]

        if I is None:
            I = np.zeros((7, luminance.size))
            s = luminance.shape

        # stack luminance
        I[i, :] = luminance.reshape((1, luminance.size))

    # load sources and convert to 3x7
    L = np.load(path + "sources.npy").T
    return I, L, s
```

1.d

In 3d, to determine $\vec{n}$ we need 3 light sources from different directions. **I** should be rank 3.

Singular values = [0.07576378 0.00906763 0.00635114 0.00194115 0.00146786 0.00115865 0.00094721]

No, they do not agree with rank-3 requirement. From the SVD of I, we get rank 7. This may be because of the disturbance in real world. Capturing more images can resolve the problem

```
    U, S, V = np.linalg.svd(I, full_matrices=False)
    print(S)
```

1.e

$L^T B = I$ (**A**x=y)

$B = (L^T)^{-1} I$ (x=A^{-1}y)

But L is not square, therefore

$B = (L^T)^{-1}\, L^{-1}\, L\, I$

$B = (L\, L^T)^{-1}\, L\, I$

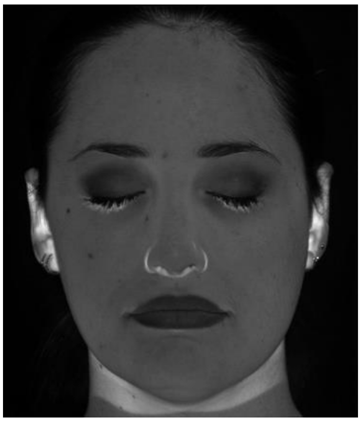
L L^T is square, hence inverse is possible

A = L L^T

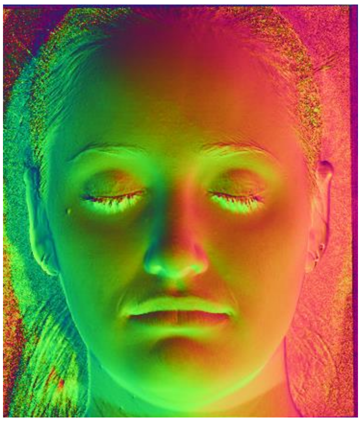
y = L I

```
def estimatePseudonormalsCalibrated(I, L):
    B = None
    # Your code here
    B = np.linalg.inv(L @ L.T) @ L @ I
    return B
```

1.f



Albedo



Normal

Unusual features found around ears, neck and nostrils. This is because n-dot-l lighting model does not account for shadows. Therefore, parts covered in shadows lose information in reconstruction.

```
def estimateAlbedosNormals(B):
    albedos = None
    normals = None
    # Your code here
    albedos = np.linalg.norm(B, axis=0)
    normals = B / (albedos + 1.0e-10)
    return albedos, normals
```

```
def displayAlbedosNormals(albedos, normals, s):
    albedoIm = None
    normalIm = None
    # Your code here
    albedoIm = albedos.reshape(s)
    normalIm = normals.T.reshape((s[0], s[1], 3))

    # normalize albedo
    albedoIm = (albedoIm - np.min(albedoIm)) / (np.max(albedoIm) -
np.min(albedoIm))

    # normalize normals
    normalIm = (normalIm - np.min(normalIm)) / (np.max(normalIm) -
np.min(normalIm))
    return albedoIm, normalIm
```

1.g

$V_1 = \left(x + 1, y, z_{x+1,y}\right) - \left(x, y, z_{xy}\right)$

$V_1 = \left(1{,}0, z_{x+1,y} - z_{xy}\right)$

$O \ = N\, V_1$

$O = (n_1 n_2 n_3)\left(1{,}0, z_{x+1,y} - z_{xy}\right)$

$O = n_1 + n_3\left(z_{x+1,y} - z_{xy}\right)$

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

Similarly,

$V_2 = \left(x, y + 1, z_{x,y+1}\right) - \left(x, y, z_{xy}\right)$

$V_2 = \left(0{,}1, z_{x,y+1} - z_{xy}\right)$

$O \ = N\, V_2$

$O = (n_1 n_2 n_3)\left(0{,}1, z_{x,y+1} - z_{xy}\right)$

$O = n_2 + n_3\left(z_{x+1,y} - z_{xy}\right)$

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

1.h

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

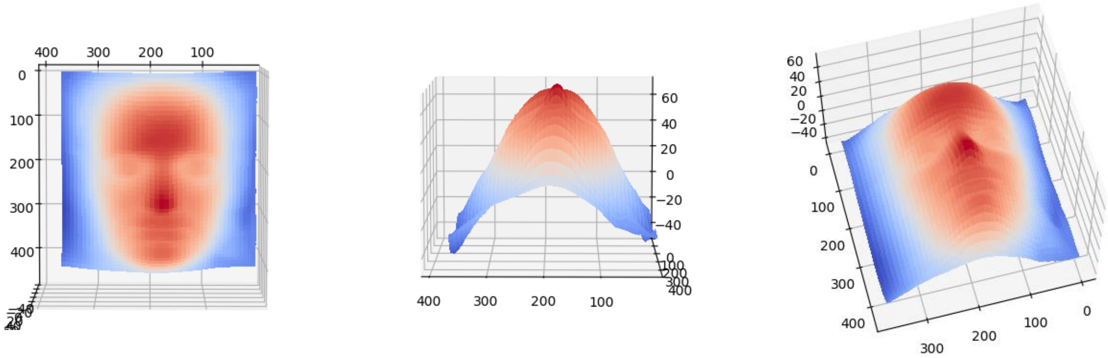
$$g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

Yes, they are same.

Following modifications make gx and gy non-integrable, leading to different g:

1. Presence of noise can make the gradients estimated in g non-integrable
2. When values in gradient matrix are not equal
3. When values in gradient are negative, then gradient addition in x and y differs

1.i



```python
def estimateShape(normals, s):
    surface = None
    # Your code here
    surface = integrateFrankot((-normals[0, :]/(normals[2, :] + 1.0e-10)).reshape(s), (-normals[1, :]/(normals[2, :] + 1.0e-10)).reshape(s), s)
    return surface
```

2.a

SVD of M is given by

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

For I

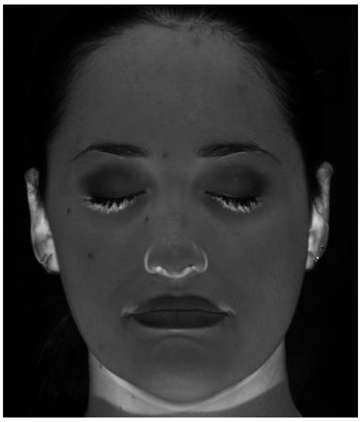
$$I = U \Sigma V^T$$

I has dimension 7 x P, U 7 x 7 and V P x P. Set all singular values except the top k from Σ to 0

1. U’ = Choose top 3 from U
2. V’ = Choose top 3 from V
3. Σ’ = Choose top 3x3 from Σ
4. L^T = U’ $\Sigma'^{1/2}$, B = $\Sigma'^{1/2}$V’

2.b

```
def estimatePseudonormalsUncalibrated(I):
    B = None
    L = None
    # Your code here
    U, S, V = np.linalg.svd(I, full_matrices=False)
    B = V[:3, :]
    L = U[:3, :]
    return B, L
```



2.c

L_0:

[-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]

$\hat{L}$:

[-0.35515581 0.35344873 0.62934359 -0.54308987 -0.15912136 0.14570881   0.1066195]

[-0.39879164 -0.62974303 0.39763736 0.42433508 -0.3114633   0.00970098   0.09544487]

[-0.32496781 0.17880977 0.10610544 0.25192986 0.28640674 0.16966527 -0.82272768]

No, they are not similar

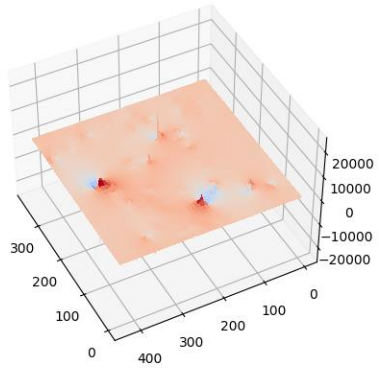
Albedos and normal are already normalized.

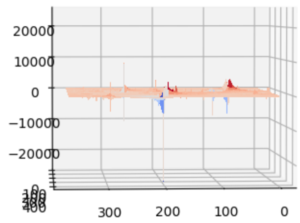
Multiplying B with G:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mu & \nu & \lambda \end{bmatrix}$$

does not change rendering

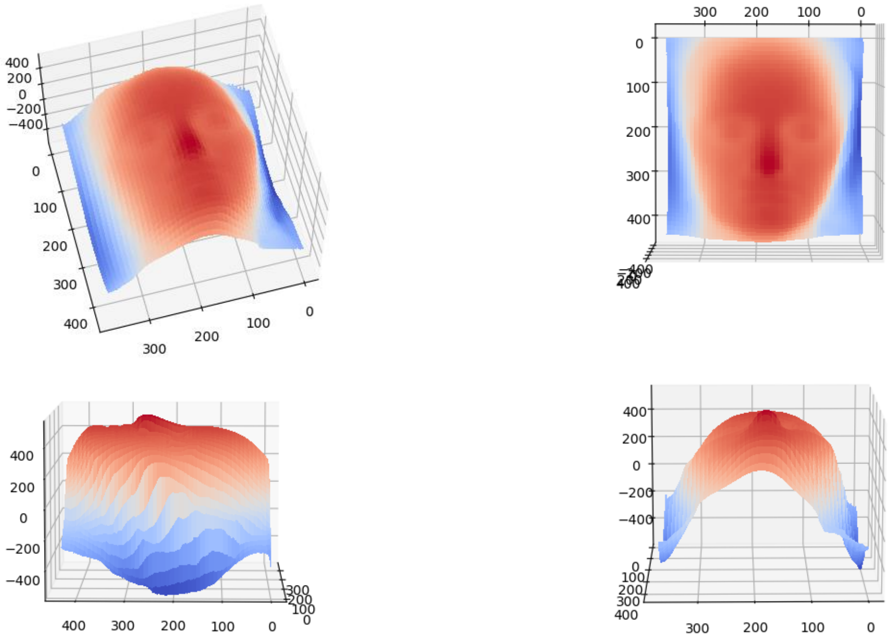
2.d





No, it does not look like face

2.e

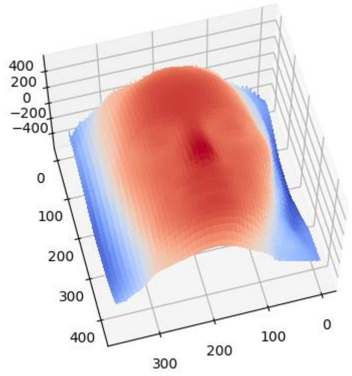


Yes, they look very similar to the output by calibrated photometric stereo

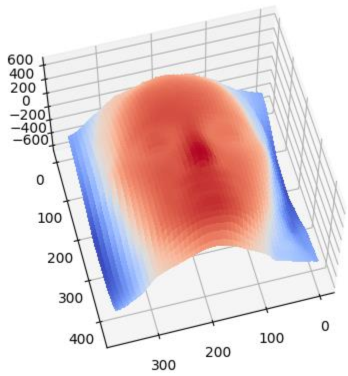
2.f

Varying $\boldsymbol{\mu}$

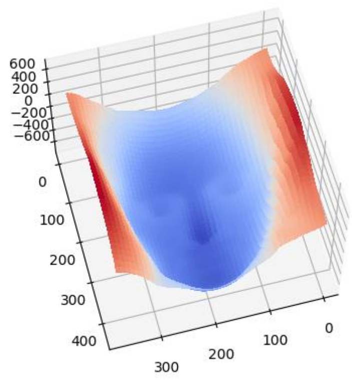
$\mu = -0.5$ $\nu = 0.5$ $\lambda = -1$



$\mu = 0.5$ $\nu = 0.5$ $\lambda = -1$



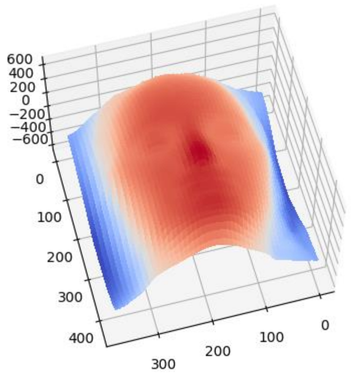
$\mu = 1$ $\nu = 0.5$ $\lambda = -1$



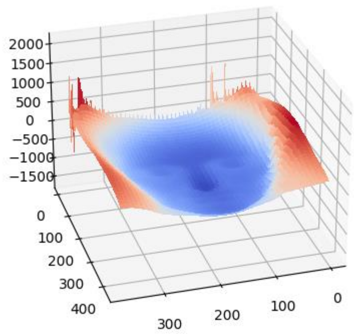
Changing μ does not affect the reconstruction except inversion taking place

Varying $\boldsymbol{\lambda}$

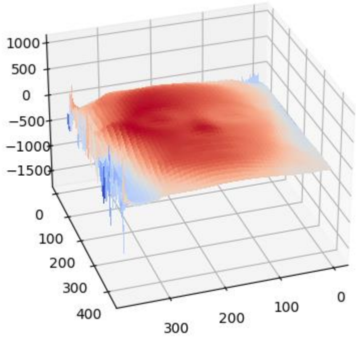
$\mu = 0.5$ $\nu = 0.5$ $\lambda = -1$



$\mu = 0.5$ $\nu = 0.5$ $\lambda = 1$



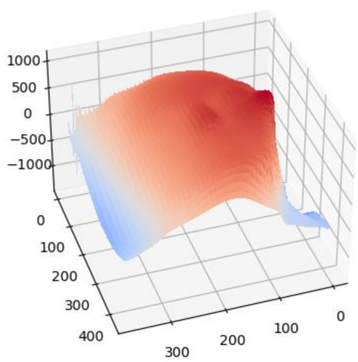
$\mu = 0.5$ $\nu = 0.5$ $\lambda = 5$



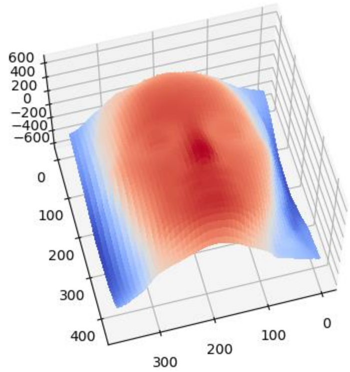
Increasing λ flattened reconstruction

Varying $\boldsymbol{\nu}$

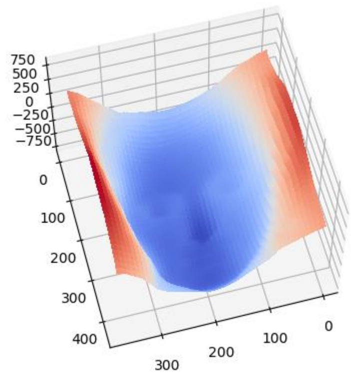
$\mu = 0.5$ $\nu = -1$ $\lambda = -1$



$\mu = 0.5$ $\nu = 0.5$ $\lambda = -1$

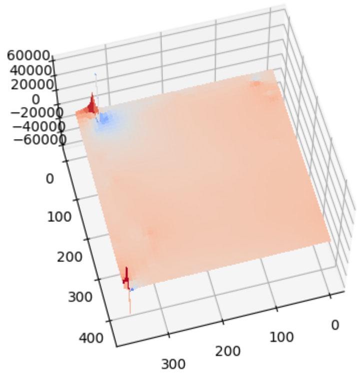


$\mu = 0.5$ $\nu = 1$ $\lambda = -1$



Changing ν caused stretching at some places

2.g



$\mu = 0$ $\nu = 0$ $\lambda = 10$

Making λ very large and $\mu = 0$ $\nu = 0$ makes the estimated surface as flat as possible

```
def plotBasRelief(B, mu, nu, lam):
    # Your code here
    G = np.array([[1, 0, 0], [0, 1, 0], [mu, nu, lam]])
    _, normals = estimateAlbedosNormals(np.linalg.inv(G.T) @ B)
    normals = enforceIntegrability(normals, s)
    surface = estimateShape(normals, s)
    plotSurface(surface)
```

It is named because of the ambiguity involved in reconstruction through lit images having shadows. Changing the parameters of G can alter the reconstruction creating ambiguity.

2.h

Yes, it does help resolving the ambiguity since pictures with more directions are fed.