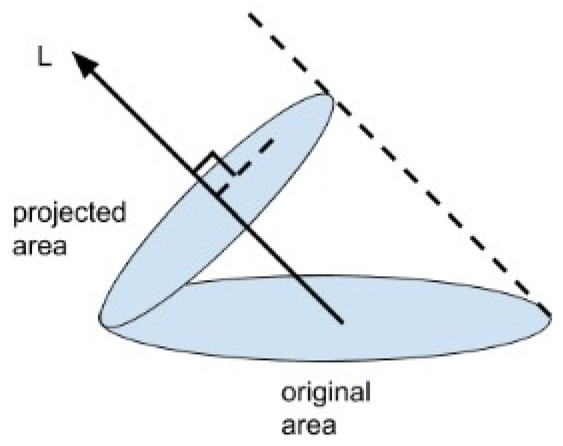
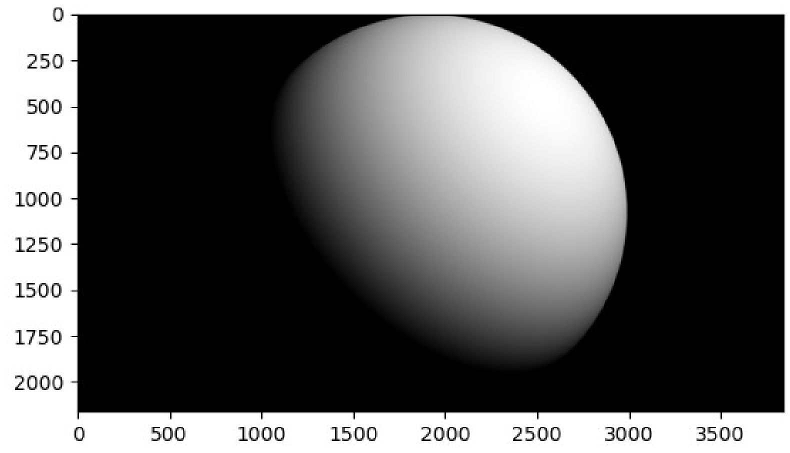
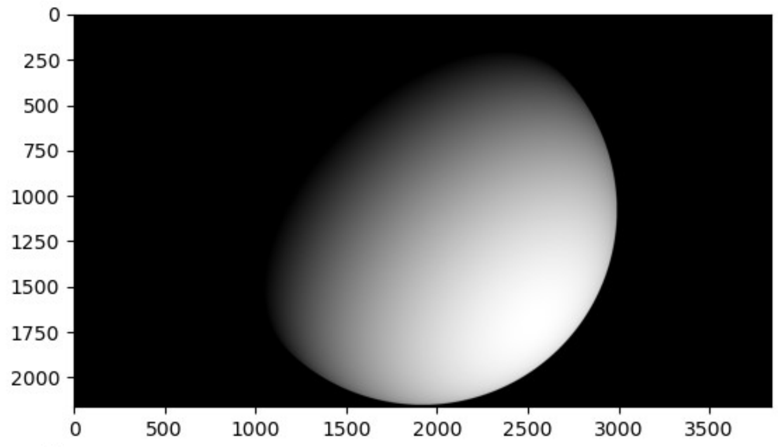
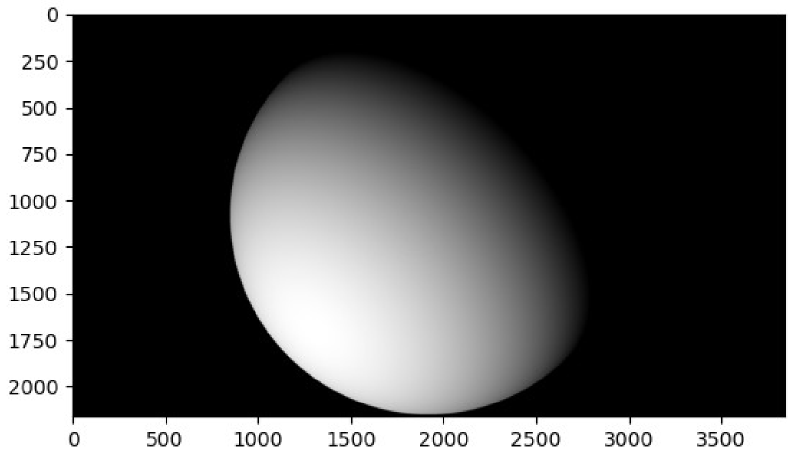


1.a. dA is a small area on the surface, n is the normal vector to that surface, l is the incident light ray that strikes the surface, and v is the light ray that goes from the surface to the camera / viewer. As the incident light ray l gets closer to normal vector n in orientation, the more light is apparent on the small area. This is described by Lambert's cosine law, which states that the cosine of the angle (between l and n) is proportionate to the surface radiance. This, in turn, leads to the dot product, as the dot product of l and n (which are unit vectors) is equal to the cosine of the angle between these vectors.
The original area is flat, and the project area is normal to L. As L gets closer to the normal of the original area, the projected area gets larger; as L gets closer to a tangent of the original area, the projected area gets smaller. This is a visualization of Lambert's cosine law: the projected area shows the effective area receiving light. As the projected area gets larger (cos of angle approaches 1), we can think that more L is interacting with the surface, making that surface brighter. As the projected area gets smaller (cos of angle approaches 0), we can think less L is interacting with the surface, making it dimmer. Viewing angle does not matter because we are assuming the surface has isotropic BRDF. We are assuming a Lambertian surface, where the BRDF is simply a constant.

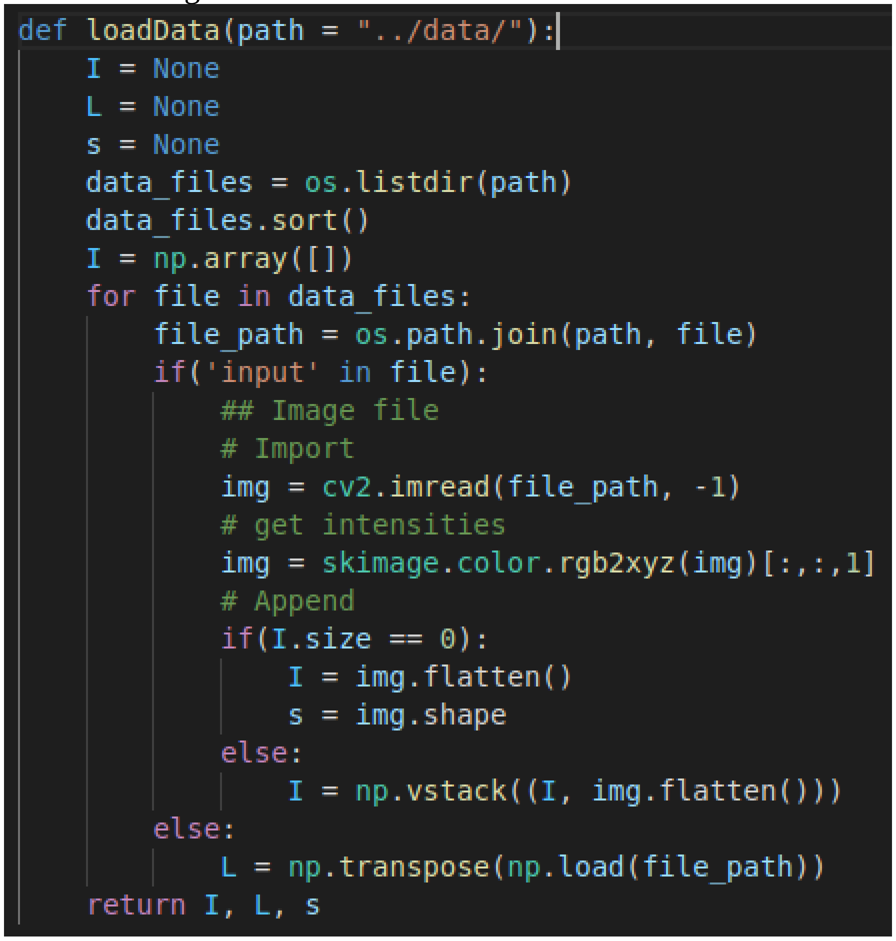
1.b.


0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500


0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500


0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500

1.c.
Comments removed so I could get this screenshot.



```python
def loadData(path = "../data/"):
    I = None
    L = None
    s = None
    data_files = os.listdir(path)
    data_files.sort()
    I = np.array([])
    for file in data_files:
        file_path = os.path.join(path, file)
        if('input' in file):
            ## Image file
            # Import
            img = cv2.imread(file_path, -1)
            # get intensities
            img = skimage.color.rgb2xyz(img)[:,:,1]
            # Append
            if(I.size == 0):
                I = img.flatten()
                s = img.shape
            else:
                I = np.vstack((I, img.flatten()))
        else:
            L = np.transpose(np.load(file_path))
    return I, L, s
```

1.d.
$I=L^TB$, where L is 3 x 7, B is 3 x P, and I is 7 x P matrix. Since B and L only have 3 rows, they are at most rank 3; since I is the product of these two matrices, it will also have at most rank 3. Generally speaking, B and L have rank 3, so I will have rank 3. Performing SVD, we find the the following singular values:

```
[72.40617702, 12.00738171, 8.42621836, 2.23003141, 1.51029184, 1.17968677, 0.84463311]
```

I has 7 singular values, so it actually has rank 7. This is due to noise; the underlying matrix is rank 3, but noise turns linearly dependent vectors into linearly independent ones.

1.e. We start with $I = L^T B$. We want to solve for B, and the easiest way to do that would be to apply the inverse of L^T to both sides of the equation. Unfortunately, L^T is not a square matrix, so the inverse does not exist. We can still solve this problem using a least squares approach by using a pseudoinverse.
In this case, L^T is 7 x 3, which is a tall matrix. For this case, we need the left pseudo-inverse:

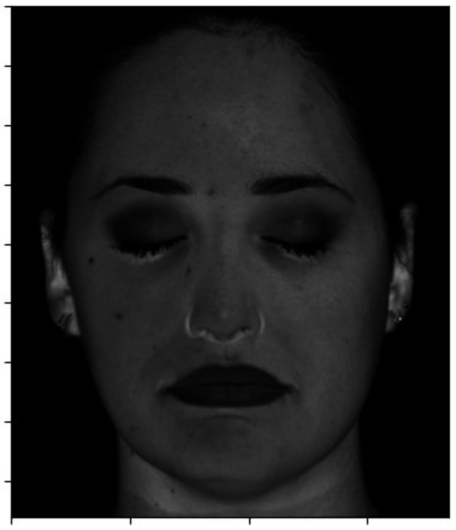
$$B = (LL^T)^{-1}LI$$

We can compare this to the rearranged equation $y = Ax$:

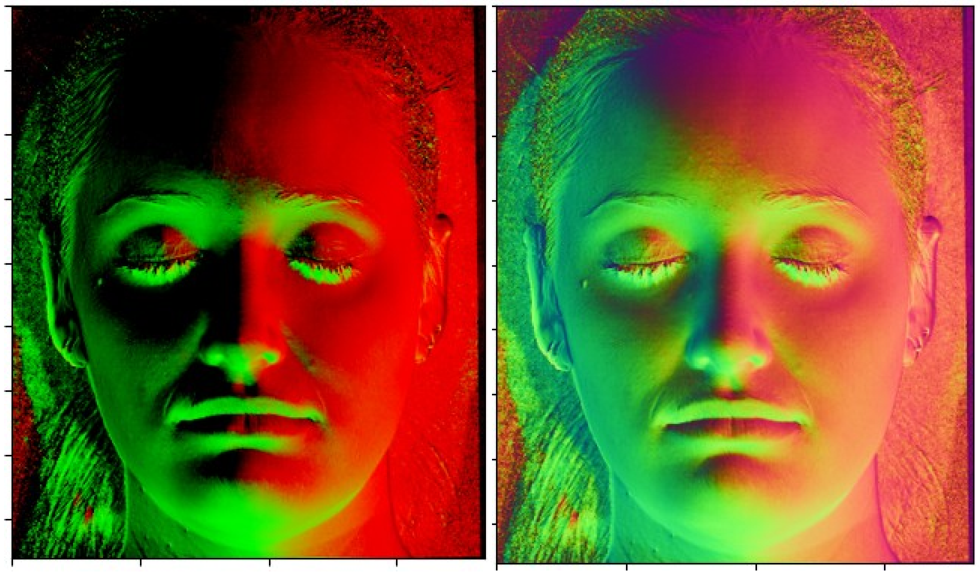
$$x = A^{-1}y$$

By inspection, we see that $A = LL^T$, and $y = LI$

1.f.

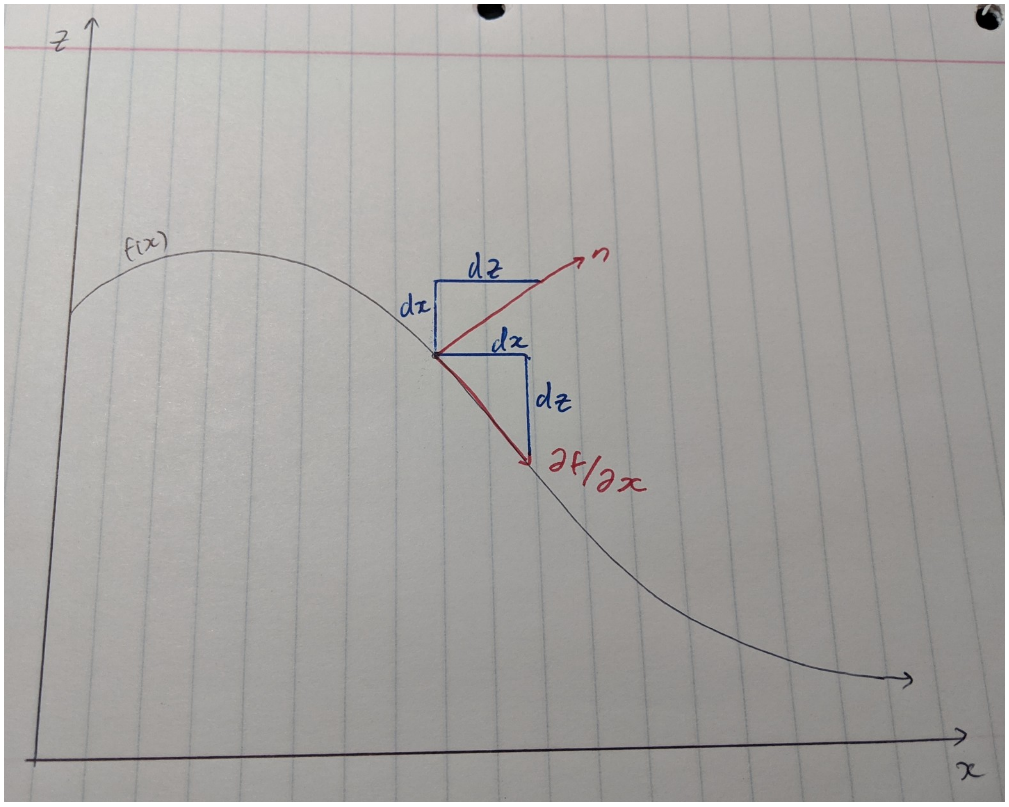


Above is the albedos image. Note the unusual highlighting around the bottom of the nose and eyes, inside of the ear, and near the top of the neck. This is caused by the skin there being unusually bright, typically due to smoothness, moisture, and/or oil. Also note the earring also has unusual brightness, again caused by its shiny surface.



Above shows the normals constructed from the image. On the left is the normals plotted using 'rainbow'. However, I noticed that 'rainbow' does not handle vectors with negative components well (negatives are read as 0), resulting in parts of the image showing up as all zeros. This was corrected in the right image, where shifting and scaling was introduced. Here, you see the normals displayed properly; the left (from our perspective) side of the face is green, the right side is orange, and the top of the face is purple. The normals image is in line with my expectations (after correction); the normals are about the same within each cheek, and the normals are about the same in the forehead area, with a gradient / natural transition.

1.g.



The image above shows a function f(x). Drawn at a point on the function is a normal vector, n, and the derivative vector at that point. The derivative has a horizontal component dx, and a vertical component dz. This forms a blue triangle, where one of the vertices is the point on the function. If we rotate this triangle by 90 degrees about this point, we see that dx and dz also form a triangle with the normal vector. This means that the normal vector has a slope of -dx/dz (negative takes into account the sign change caused by rotation: dz points in the -z direction before the rotation, and +x direction after rotation), and f_x has slope dz/dx. Note that the normal vector is (n_1,n_3), so it has slope n_3/n_1. Equating the two slopes for n, we get $-dx/dz = n_3/n_1$. Moving the minus sign and flipping both fractions, we get:

$$dz/dx = -n_1/n_3 = f_x$$

Repeating the same analysis using f(y) yields $f_y = -n_2/n_3$.

1.h. g_x is a 4x3 matrix of all ones, and g_y is a 3x4 matrix of all fours. Using g(0,0) = 1, we can use g_x and g_y for reconstruction:

- g_x gives a first row: [1 2 3 4]. Using g_y, we get the rest of the rows: [5 6 7 8], [9 10 11 12], [13, 14, 15, 16]
- g_y gives the first column: $[1\ 5\ 9\ 13]^T$. Using g_x, we get the rest of the columns: $[2\ 6\ 10\ 14]^T$, $[3\ 7\ 11\ 15]^T$, $[4\ 8\ 12\ 16]^T$

Note that both approaches above yield the same result.

We can modify the g_x so the two methods give different results. Modifying g_x so that the top left element is 0 instead of 1, the two methods give different results:

- Construct 1 row, then the rest of the rows:

$$\begin{bmatrix} 1 & 1 & 2 & 3 \\ 5 & 5 & 6 & 7 \\ 9 & 9 & 10 & 11 \\ 13 & 13 & 14 & 15 \end{bmatrix}$$

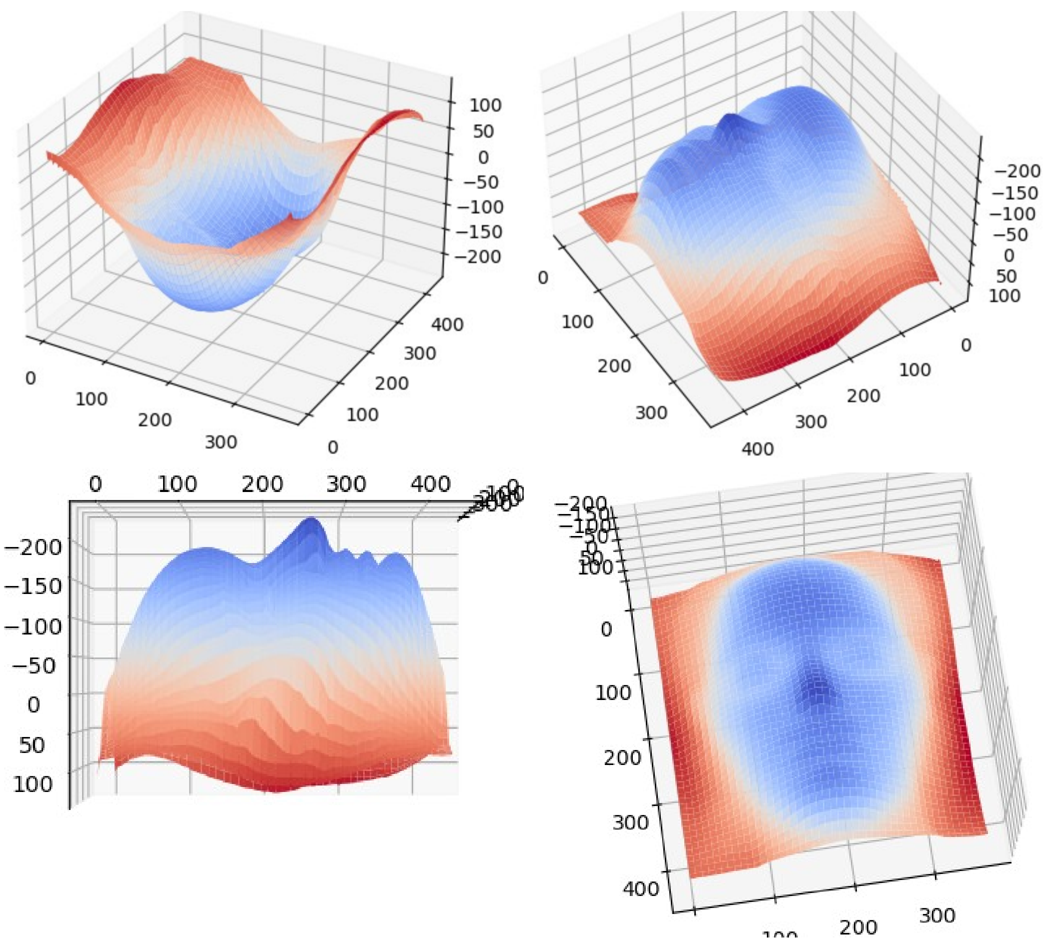
- Construct 1 column, then the rest of the columns:

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

Clearly, the two results are different.

The gradients computed in (g) may be non-integrable due to noise. As we saw, a small change in a single element of the gradient matrix may cause the non-integrability; in a real system, such as photos of the real world, the results will be impacted by noise, which will change many values. The end result are gradients that are not-integrable.

1.i.



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

2.a. Performing SVD on I, we get:

$$I = U*S*V^T$$

S is a diagonal matrix full of singular values. Setting all singular values except the first three to zero to get S_{hat}, we compute I_{hat}:

$$I_{hat}=U*S_{hat}*V^T$$

I_{hat} is like I, but the rank 3 constraint is now enforced. Next, we compute SVD once again on I_{hat} to perform decomposition / factorization:

$$I_{hat} = U_2*S_2*V_2^T=L_{hat}^TB_{hat}$$

Now, we can group S_2 with U_2 or V_2^T; I chose to group it with U_2. This gives:

$$L_{hat}^T = U_2*S_2$$

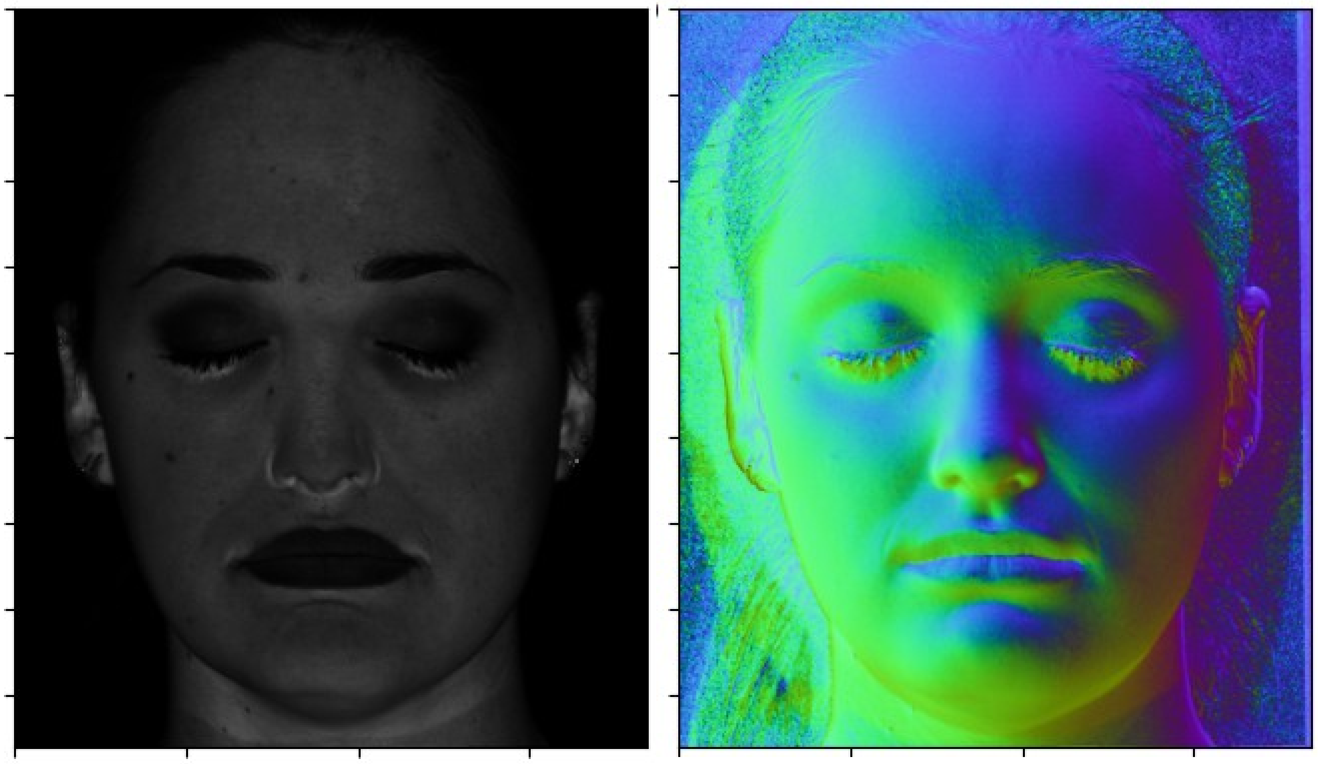
$$B_{hat} = V_2^T$$

Note L_{hat} is a 3x7 matrix, and B_{hat} is a 3xP matrix. To account for this, we adjust the above equations:

$$L_{hat} = \text{transpose of first 3 columns of } U_2*S_2$$

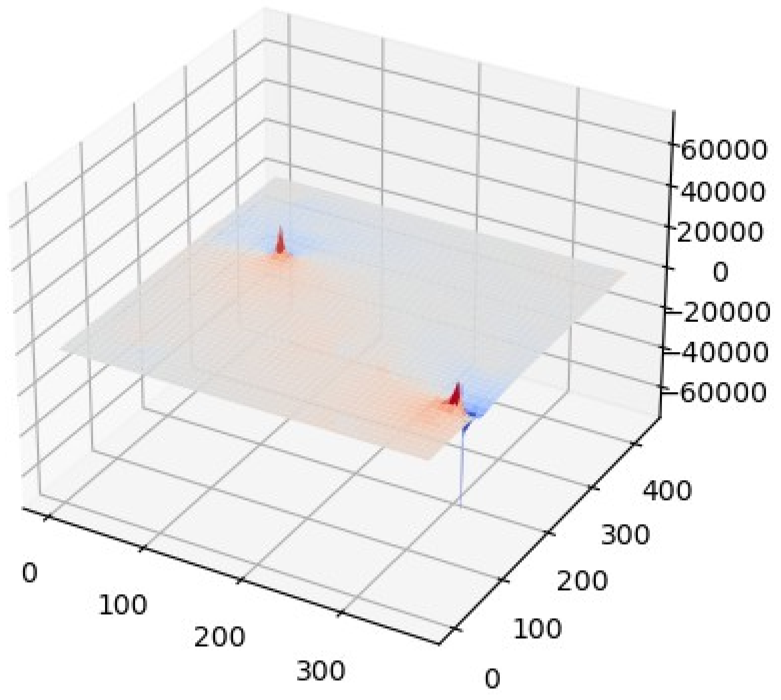
$$B_{hat} = \text{first 3 rows of } V_2^T$$

2.b.

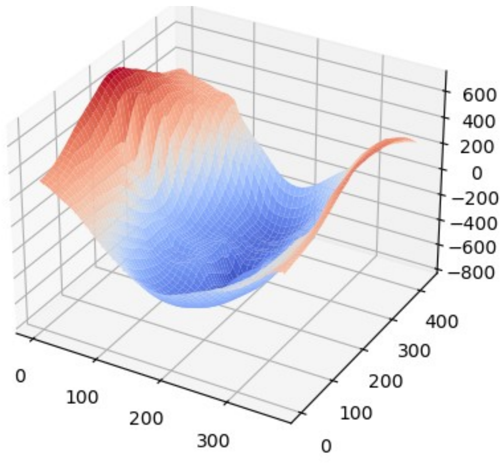


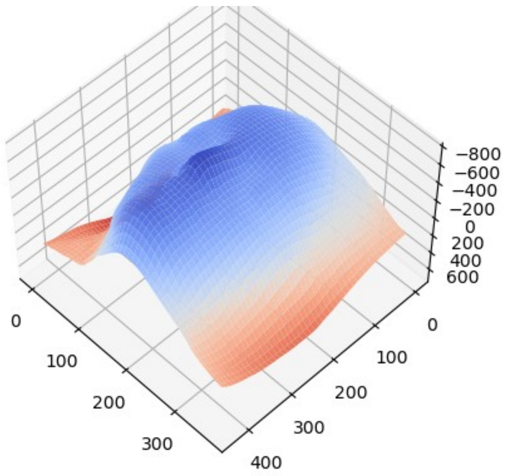
2.c. L and L_{hat} are not the same:
L =
```
[[-0.1418  0.1215 -0.069   0.067  -0.1627  0.      0.1478]
 [-0.1804 -0.2026 -0.0345 -0.0402  0.122   0.1194  0.1209]
 [-0.9267 -0.9717 -0.838  -0.9772 -0.979  -0.9648 -0.9713]]
```
Lhat =
```
[[-24.26835522 -31.50311608 -19.5703665  -30.47642999 -29.16624593
  -27.5249563  -27.22406898]
 [  3.11842287  -7.68972895   1.65045887  -2.05759441   7.69612734
    1.56082151  -2.58775461]
 [  5.22072085   2.80201767   1.19050877  -0.06108961  -0.83284533
   -2.53210904  -5.23141244]]
```
Note that L_{hat} and B_{hat} defined in 2.a. is not unique. As mentioned in my answer there, S_2 didn't have to be grouped with U_2; it could be grouped with V_2^T, or it could be split into two evenly and shared (each one is square root of S_2), or it could even be split unevenly and shared ($S_2^{1/3}$ and $S_2^{2/3}$, for example). Any of these approaches would change L_{hat} and B_{hat} without changing the rendered images.

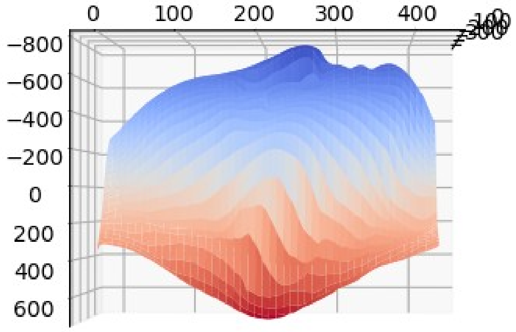
2.d. This doesn’t look anything like a face:

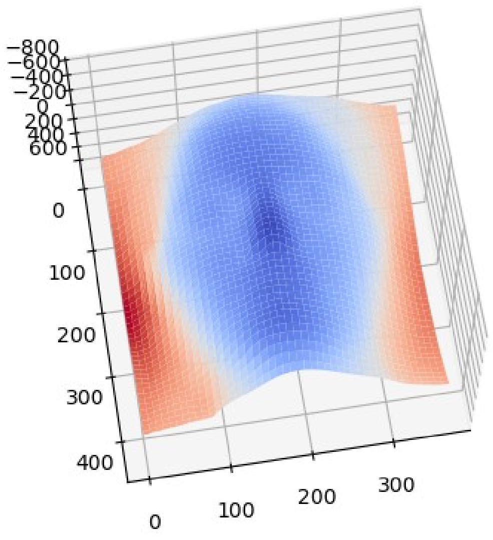


2.e. After enforcing integrability, the result is much closer to resembling a human face, but the results aren’t exactly the same as when we used calibrated lighting:

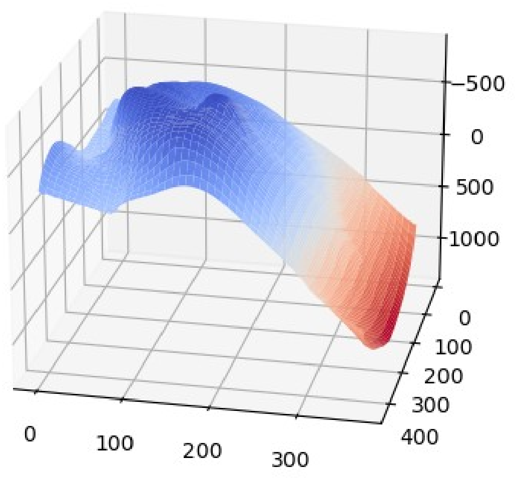


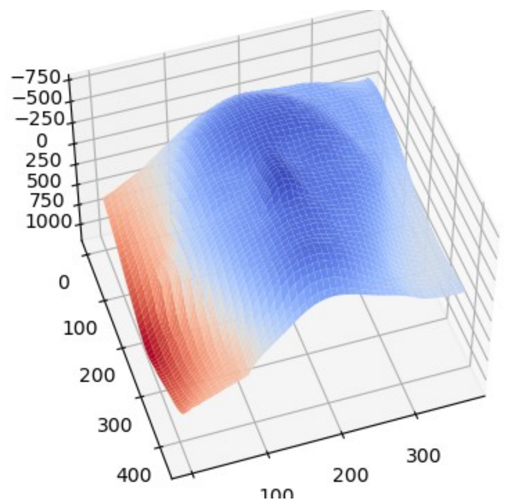




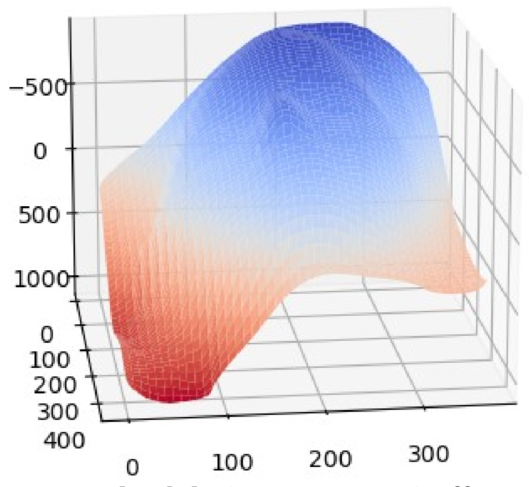


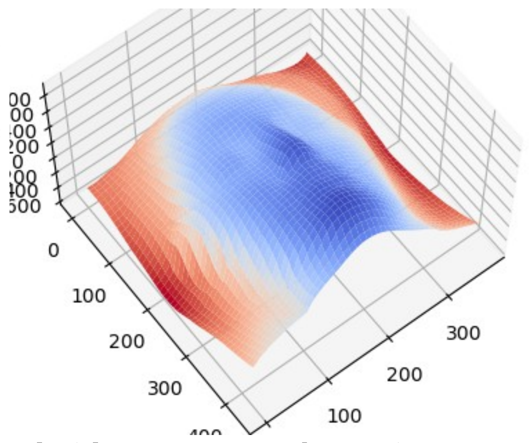
2.f.
Varying mu (20 and -10) squishes and stretches the face:



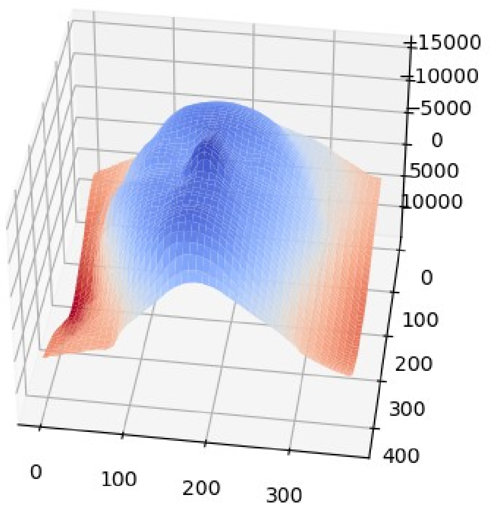


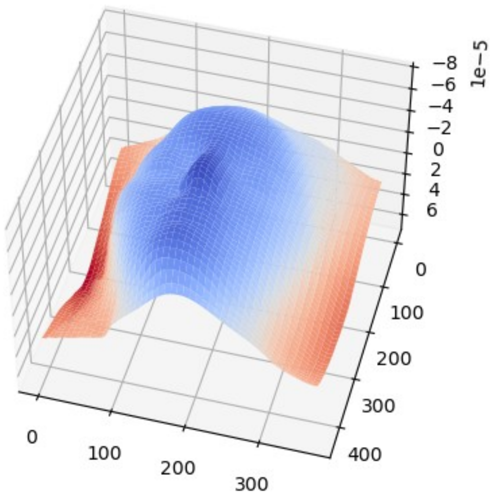
Varying v (20, -10) causes the face to tilt and change orientation:





Varying lambda (20, 0.0000001) effects scale (elongation; note the axes):





Bas-relief is so named because bas means “low” in French. That is, when a statue is made flat or “low”, such as when a statue is built into a wall, the statues are in truth distorted, but lighting can hide this distortion. This is similar to the ambiguity here, where the object shape can be distorted.

2.g. Make lambda as small as possible. In 2.f., we saw that setting lambda to a very small value made the depth of the image extremely small, so the smaller lambda is, the flatter the surface will be.

2.h. More measurements will help the quality of the reconstruction, by virtue of having more images to aid in the least-squares solution. However, the ambiguity cannot be eliminated entirely, even with the use of more images. Without knowing the light source direction, we cannot determine with certainty the shape of object.