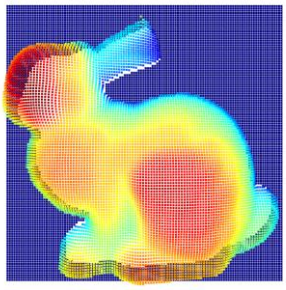
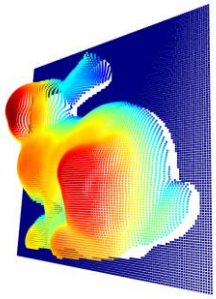
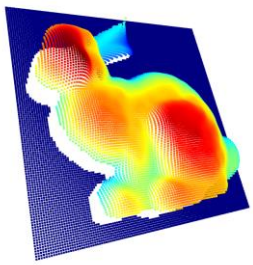
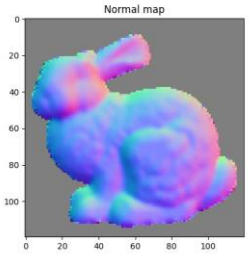
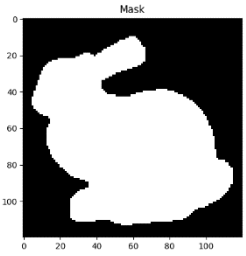
Homework 1 : Photometric Stereo

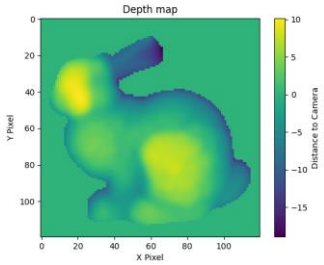
109550135　范恩宇

Results :

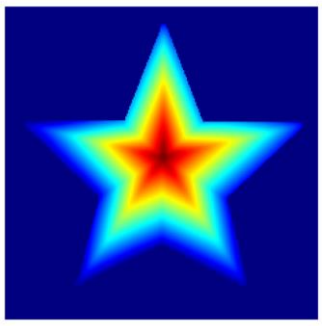
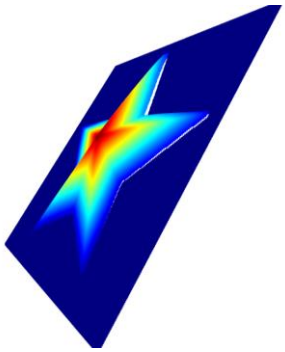
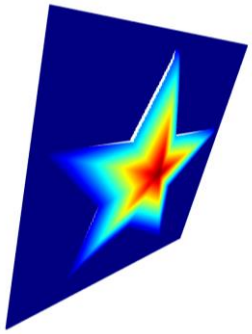
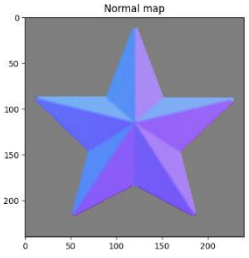
Bunny

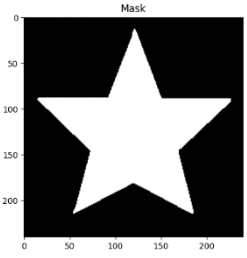





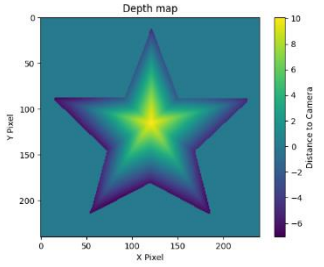




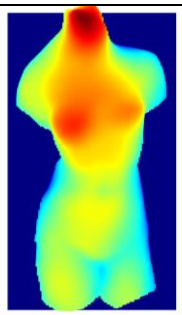
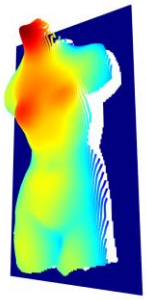
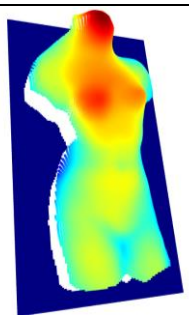
Star

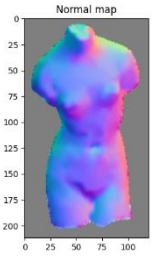





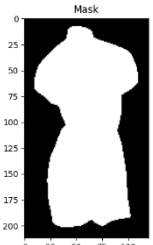


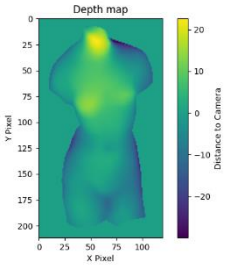


Venus

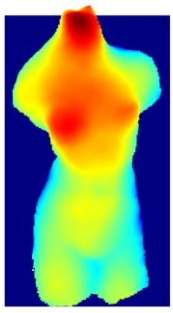
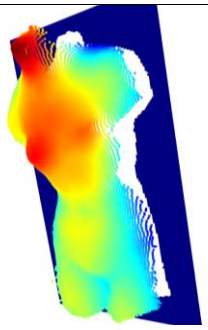
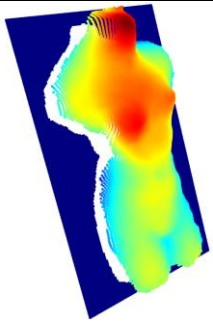




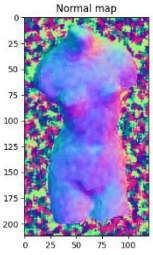


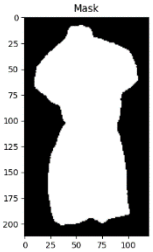


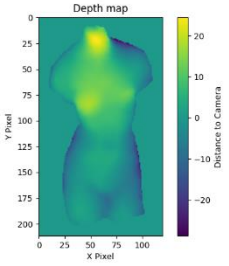


Noisy Venus







Implementation :

A. Mask

```
_, mask = cv2.threshold( mask, threshold_value, max_value, cv2.THRESH_BINARY )
```

To reconstruct the object, we first need to find a mask for locating the object, and I use the threshold function of cv2 with "THRESH_BINARY" to transform the grayscale image through binarization.

B. Normal

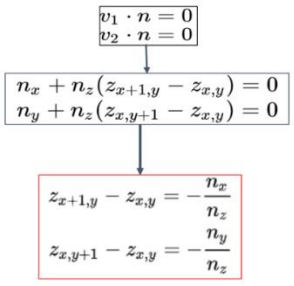
Base on normal estimation of Photometric Stereo, we use the below equation, while "I" represents intensity, "L" represents unit vector of light source, and "N" represents normal at the surface point.

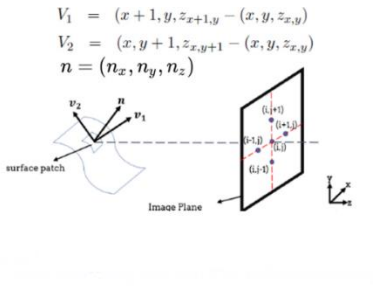
Here we perform pseudo-inverse by multiplying L^T on both sides of the original equation, then get K_dN through function "np.linalg.solve(L.T @ L, L.T @ I)" and normalize it through normalize( ) from sklearn.preprocessing

$$i_{x,y}^m = l_m K_d n \;\rightarrow\; I = LK_dN$$

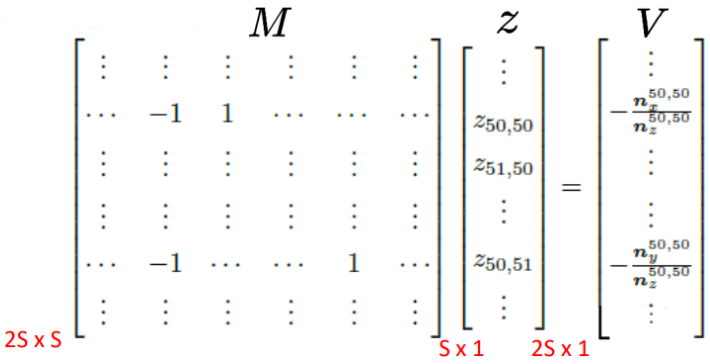
$$I = LK_dN \;\rightarrow\; L^TI = L^TLK_dN \;\rightarrow\; K_dN = (L^TL)^{-1}L^TIN = \frac{K_dN}{||K_dN||}$$

C. Depth & Surface Reconstruction





Through the above given formula for surface reconstruction, we know that normal vector "n" is orthogonal to vector v1 & v2. Then we can define a linear system, $Mz = v$, while setting "S" as the image size :



Basically through the following formula, I check whether the input mask and vectors have points from the object that surround them, which somehow accelerates the process of filling "M"'s element values.

$$z_{x+1,y} - z_{x,y} = -\frac{n_x}{n_z}$$

$$z_{x,y+1} - z_{x,y} = -\frac{n_y}{n_z}$$

```
# compute index numbers for M
idx_arr = np.zeros((image_row, image_col)).astype(np.int16)
for i in range(pixel_num):
    idx_arr[nonzero_h[i], nonzero_w[i]] = i

for i in range(pixel_num):
    h = nonzero_h[i]
    w = nonzero_w[i]
    # normal vecs
    n_x = N[h, w, 0]
    n_y = N[h, w, 1]
    n_z = N[h, w, 2]

    # - nx / nz -> z(x+1, y) - z(x, y)
    j = i*2
    if mask[h, w+1]: # right
        k = idx_arr[h, w+1]
        M[j, i] = -1
        M[j, k] = 1
        v[j] = -n_x/n_z
    elif mask[h, w-1]: # left
        k = idx_arr[h, w-1]
        M[j, k] = -1
        M[j, i] = 1
        v[j] = -n_x/n_z
```

```
    # - nx / nz -> z(x+1, y) - z(x, y)
    j = i*2
    if mask[h, w+1]: # right
        k = idx_arr[h, w+1]
        M[j, i] = -1
        M[j, k] = 1
        v[j] = -n_x/n_z
    elif mask[h, w-1]: # left
        k = idx_arr[h, w-1]
        M[j, k] = -1
        M[j, i] = 1
        v[j] = -n_x/n_z

    # -ny / nz -> z(x, y+1) - z(x, y)
    j = i*2+1
    if mask[h+1, w]:
        k = idx_arr[h+1, w] # up
        M[j, i] = 1
        M[j, k] = -1
        v[j] = -n_y/n_z
    elif mask[h-1, w]:
        k = idx_arr[h, w-1] # down
        M[j, k] = 1
        M[j, i] = -1
        v[j] = -n_y/n_z
```

After getting "M", then compute the "z" through "sparse.linalg.spsolve(a,b)" from scipy. Here I perform pseudo-inverse method again by multiplying M^T on both sides of $Mz = v$

```
# M.T * M * z = M.T * v
z = scipy.sparse.linalg.spsolve(M.T @ M, M.T @ v)
```

D. Surface Optimizing

Since we have some extreme normals in venus and noisy_venus, there exists some outliers having values that are far from most of points. To deal with this, I set values of points that are above(below) 4 standard deviations to maximum(minimum) normalized value. This resulting in a new point set, and making the surface look smoother while the objects remains not so flat.

```
# filter extreme points in z
z_normed = (z - np.mean(z)) / np.std(z)
extreme_idx = np.abs(z_normed) > 4
z_max = np.max(z[~extreme_idx])
z_min = np.min(z[~extreme_idx])

new_depth = mask.astype(np.float32)

for i in range(pixel_num):
    if z[i] > z_max:
        new_depth[nonzero_h[i], nonzero_w[i]] = z_max
    elif z[i] < z_min:
        new_depth[nonzero_h[i], nonzero_w[i]] = z_min
    else:
        new_depth[nonzero_h[i], nonzero_w[i]] = z[i]

return new_depth
```

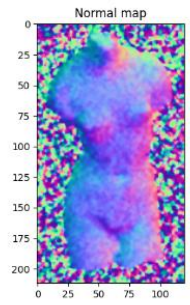
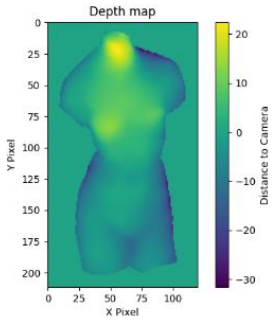
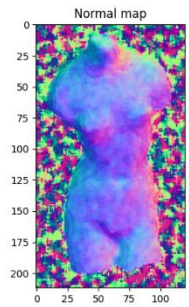
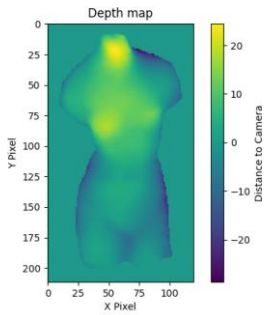
E. Measures for Gaussian noise

```
# read the .bmp file & denoise input
def read_bmp(filepath):
    global image_row
    global image_col
    image = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
    image_row, image_col = image.shape
    #denoised_img = cv2.GaussianBlur(image, (5, 5), 0)# 10
    denoised_img = cv2.medianBlur(image, 5)# 10
    return denoised_img
```

Eventually, I decided to denoise the image before processing it. For this task, GaussianBlur and Median Filter first came up to my mind, while the former denoises image by convolving it with a Gaussian kernel then resulting in smoother appearance, and the latter replaces each pixel's value with median value of its neighboring pixels.

At the beginning, I set 10 as the kernel size and find it miss too much detail, comparing with the “not-noisy venus”. Finally, I choose 5 to be the size by finding out that it’s the minimum size leading to a qualified result to me.

After comparing how these two methods perform, I think Median Filter did a better job, since it not only detects the edge of Venus better than GaussianBlur, but also lost less detail.

<table>
<tr><th colspan="4">Kernel = 5</th></tr>
<tr><th colspan="2">GaussianBlur</th><th colspan="2">Median Filter</th></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
</table>