HW3

November 6, 2019

1 CSE 252A Computer Vision I Fall 2019 - Homework 3

1.1 Instructor: Ben Ochoa

1.1.1 Assignment published on: Tuesday, October 22, 2019

1.1.2 Due on: Tuesday, November 5, 2019 11:59 pm

1.2 Instructions

- Review the academic integrity and collaboration policies on the course website.
  - This assignment must be completed individually.
- All solutions must be written in this notebook.
  - This includes the theoretical problems, for which you **must** write your answers in Markdown cells (using LaTeX when appropriate).
  - Programming aspects of the assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you may do so. It has only been provided as a framework for your solution.
- You may use Python packages (such as NumPy and SciPy) for basic linear algebra, but you may not use packages that directly solve the problem.
  - If you are unsure about using a specific package or function, then ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF. You must also submit this notebook as `.ipynb` file.
  - Submit both files (`.pdf` and `.ipynb`) on Gradescope.
  - **You must mark the PDF pages associated with each question in Gradescope. If you fail to do so, we may dock points.**
- It is highly recommended that you begin working on this assignment early.
- **Late policy: assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.**

1.3 Problem 1: Photometric Stereo, Specularity Removal [20 pts]

The goal of this problem is to implement a couple of different algorithms that reconstruct a surface using the concept of Lambertian photometric stereo. Additionally, you will implement the specular removal technique of Mallick et al., which enables photometric stereo to be performed on certain non-Lambertian materials.

You can assume a Lambertian reflectance function once specularities are removed. However, note that the albedo is unknown and non-constant in the images you will use.

As input, your program should take in multiple images along with the light source direction for each image. Each image is associated with only a single light, and hence a single direction.

1.3.1 Data

You will use synthetic images and specular sphere images as data. These images are stored in `.pickle` files which have been graciously provided by Satya Mallick. Each `.pickle` file contains

- `im1`, `im2`, `im3`, `im4`, ... images.
- `l1`, `l2`, `l3`, `l4`, ... light source directions.

1.3.2 Part 1: Lambertian Photometric Stereo [8 pts]

Implement the photometric stereo technique described in the lecture slides and in Forsyth and Ponce 2.2.4 (*Photometric Stereo: Shape from Multiple Shaded Images*). Your program should have two parts:

1. Read in the images and corresponding light source directions, and estimate the surface normals and albedo map.

2. Reconstruct the depth map from the surface normals. You should first try the naive scanline-based "shape by integration" method described in the book and in lecture. (You are required to implement this.) For comparison, you should also integrate using the Horn technique which is already implemented for you in the `horn_integrate` function. Note that for good results you will often want to run the `horn_integrate` function with 10000-100000 iterations, which will take a while. For your final submission, we will require that you run Horn integration for 1000 (one thousand) iterations or more in each case. But for debugging, it is suggested that you keep the number of iterations low.

You will find all the data for this part in `synthetic_data.pickle`. Try using only `im1`, `im2` and `im4` first. Display your outputs as mentioned below.

Then use all four images (most accurate).

For **each** of the **two above cases** you must output:

1. The estimated albedo map.

2. The estimated surface normals by showing both

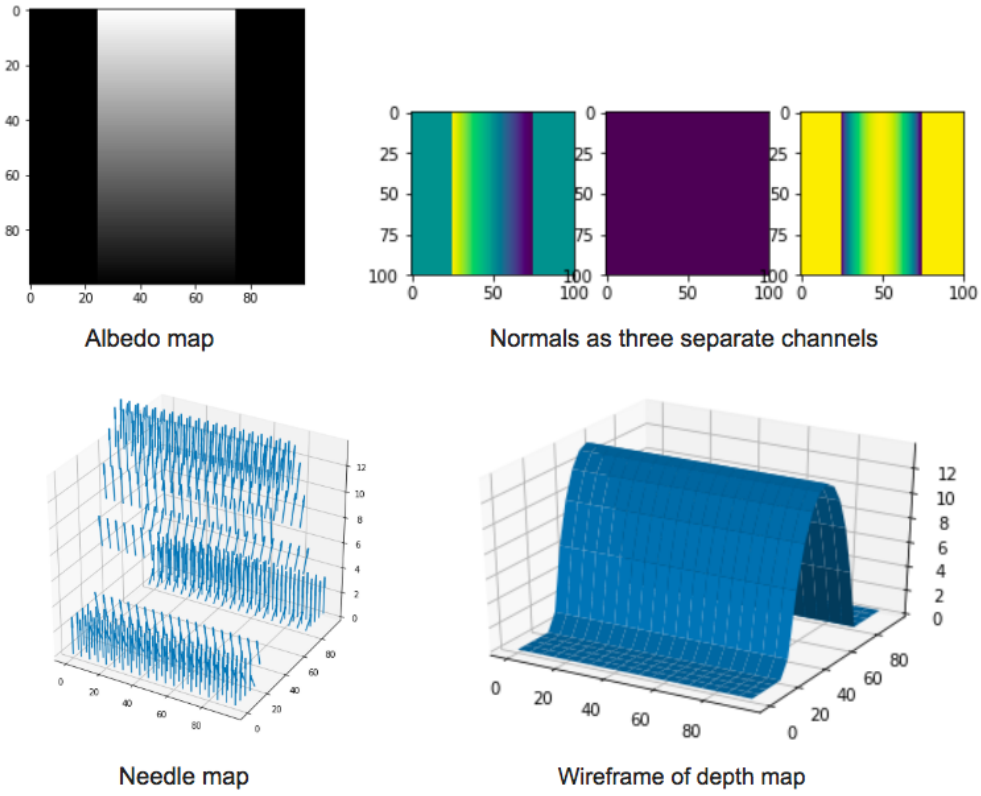
    1. Needle map, and
    2. Three images showing each of the surface normal components.

3. A wireframe of the depth map given by the scanline method.

4. A wireframe of the depth map given by Horn integration.

In total, we expect 2 * 7 = 14 images for this part.

An example of outputs is shown in the figure below. (The example outputs only include one depth map, although we expect two – see above.)



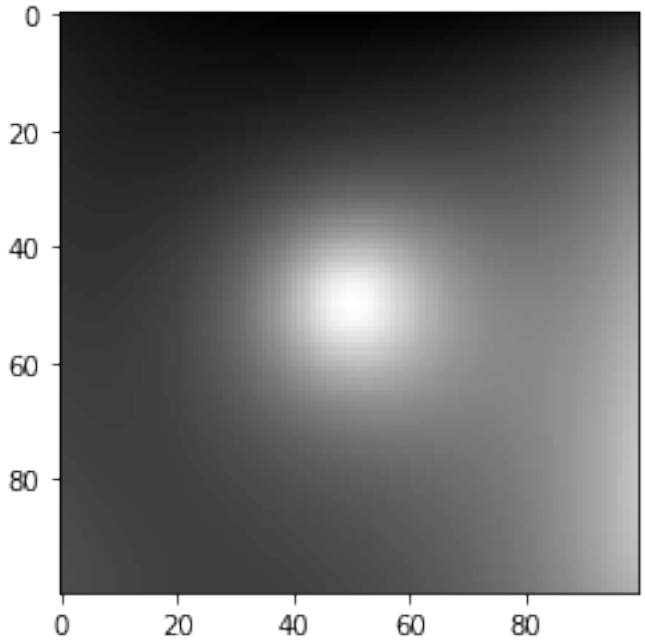
Problem 1.1 example outputs

[1]:
```
# Setup
import pickle
import numpy as np
from time import time
from skimage import io
%matplotlib inline
import matplotlib.pyplot as plt

### Example: how to read and access data from a .pickle file
pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# data is a dict which stores each element as a key-value pair.
print("Keys: ", list(data.keys()))

# To access the value of an entity, refer to it by its key.
```

```
for i in range(1, 5):
    print("\nImage %d:" % i)
    plt.imshow(data["im%d" % i], cmap="gray")
    plt.show()
    print("Light source direction: " + str(data["l%d" % i]))
```

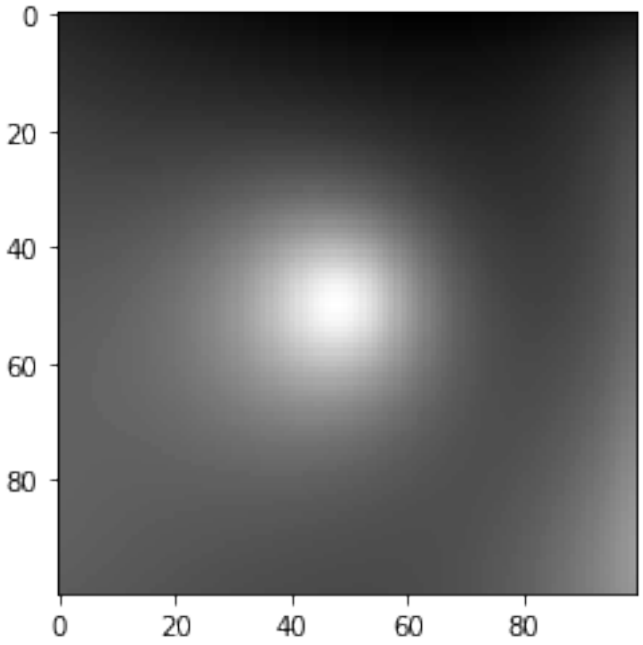
```
Keys:  ['__version__', 'l4', '__header__', 'im1', 'im3', 'im2', 'l2', 'im4',
'l1', '__globals__', 'l3']
```

```
Image 1:
```



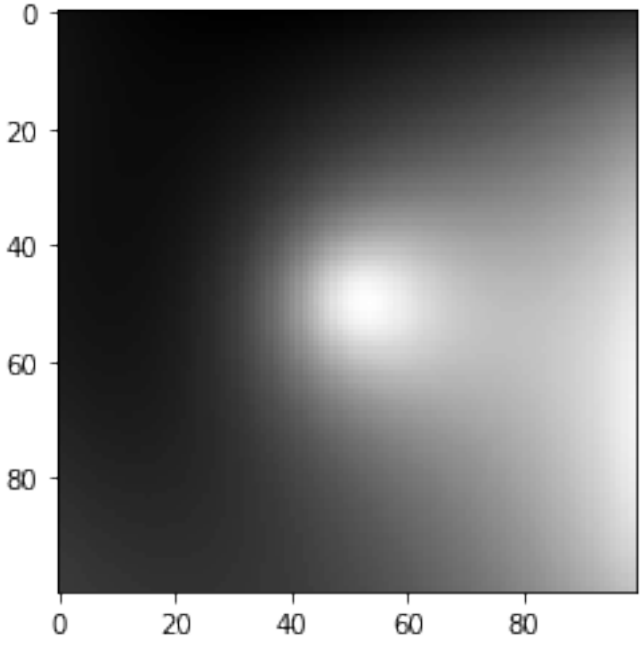

```
Light source direction: [[0 0 1]]
```

```
Image 2:
```

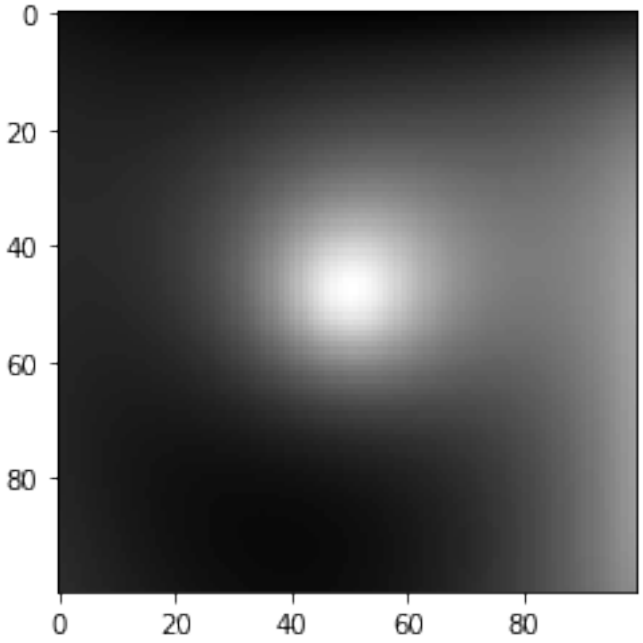



```
Light source direction: [[0.2 0.  1. ]]
```

Image 3:



```
Light source direction: [[-0.2  0.   1. ]]

Image 4:
```




```
Light source direction: [[0.  0.2 1. ]]
```

Based on the above images, can you interpret the orientation of the coordinate frame? If we label the axes in order as x, y, z, then the x-axis points left, the y-axis points up, and the z-axis points out of the screen in our direction. (That means this is a left-handed coordinate system. How will this affect the scanline integration algorithm? Hint: if you integrate rightward along the x-axis and downward along the y-axis, you will be doing in opposite directions to the axes, and the partial derivatives you compute may need to be modified.)

Note: as clarification, no direct response is needed for this cell.

[2]:
```python
import numpy as np
from scipy.signal import convolve

def horn_integrate(gx, gy, mask, niter):
    """
    horn_integrate recovers the function g from its partial
    derivatives gx and gy.
    mask is a binary image which tells which pixels are
    involved in integration.
    niter is the number of iterations.
    typically 100,000 or 200,000,
    although the trend can be seen even after 1000 iterations.
```

```
    """
    g = np.ones(np.shape(gx))

    gx = np.multiply(gx, mask)
    gy = np.multiply(gy, mask)

    A = np.array([[0,1,0],[0,0,0],[0,0,0]]) #y-1
    B = np.array([[0,0,0],[1,0,0],[0,0,0]]) #x-1
    C = np.array([[0,0,0],[0,0,1],[0,0,0]]) #x+1
    D = np.array([[0,0,0],[0,0,0],[0,1,0]]) #y+1

    d_mask = A + B + C + D

    den = np.multiply(convolve(mask,d_mask,mode="same"),mask)
    den[den == 0] = 1
    rden = 1.0 / den
    mask2 = np.multiply(rden, mask)

    m_a = convolve(mask, A, mode="same")
    m_b = convolve(mask, B, mode="same")
    m_c = convolve(mask, C, mode="same")
    m_d = convolve(mask, D, mode="same")

    term_right = np.multiply(m_c, gx) + np.multiply(m_d, gy)
    t_a = -1.0 * convolve(gx, B, mode="same")
    t_b = -1.0 * convolve(gy, A, mode="same")
    term_right = term_right + t_a + t_b
    term_right = np.multiply(mask2, term_right)

    for k in range(niter):
        g = np.multiply(mask2, convolve(g, d_mask, mode="same")) + term_right

    return g
```

[3]:
```
def photometric_stereo(images, lights, mask, horn_niter=25000):

    """mask is an optional parameter which you are encouraged to use.
    It can be used e.g. to ignore the background when integrating the normals.
    It should be created by converting the images to grayscale, averaging them,
    normalizing to [0, 1] and thresholding (only using locations for which the
    pixel value is above some threshold).

    The choice of threshold is something you can experiment with,
    but in practice something like 0.05 or 0.1 tends to work well.

    You do not need to use the mask for 1a (it shouldn't matter),
```

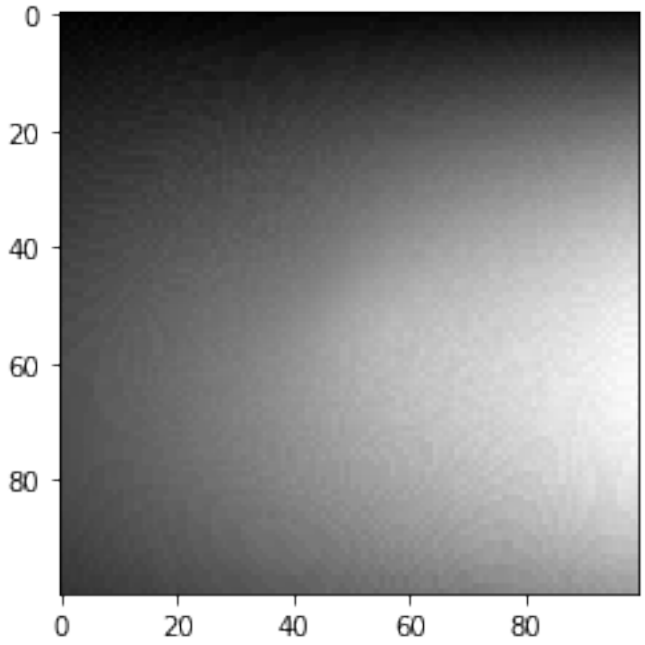
```
    but you SHOULD use it to filter out the background for the specular data␣
↪(1c).
    """
    # note:
    # images : (n_ims, h, w)
    # lights : (n_ims, 3)
    # mask   : (h, w)

    albedo = np.ones(images[0].shape)
    normals = np.dstack((np.zeros(images[0].shape),np.zeros(images[0].shape),np.
↪ones(images[0].shape)))
    H = np.ones(images[0].shape)
    H_horn = np.ones(images[0].shape)

    # Create arrays for albedo, normal
    p = np.zeros(images[0].shape)
    q = np.zeros(images[0].shape)

    I = np.zeros([images[0].shape[0],images[0].shape[1],lights.shape[0]])
    g = np.zeros([images[0].shape[0],images[0].shape[1],3])

    # for each point in the image array
    for h in range(images[0].shape[0]):
        for w in range(images[0].shape[1]):
            for n_ims in range(lights.shape[0]):
                # Stack image values into a vector i
                I[h,w,n_ims] = images[n_ims,h,w]
            # Solve Vg = i to obtain g for this point
            g[h,w,:] = np.dot(np.dot(np.linalg.inv(np.dot(lights.
↪T,lights)),lights.T),I[h,w,:])
            # Albedo at this point is |g|
            albedo[h,w] = np.linalg.norm(g[h,w,:])
            # Normal at this point is g/|g|
            normals[h,w] = g[h,w]/albedo[h,w]

            p[h,w] = normals[h,w,0]/normals[h,w,2]
            q[h,w] = normals[h,w,1]/normals[h,w,2]

    p = p * mask
    q = q * mask

    for col in range(images[0].shape[0]-1):
        H[col+1,0] = H[col,0] + q[col,0]
    for col in range(images[0].shape[0]):
        for row in range(images[0].shape[1]-1):
            H[col,row+1] = H[col,row] + p[col,row]
```

```
    H_horn = horn_integrate(p,q,mask,horn_niter)

    return albedo, normals, H, H_horn
```

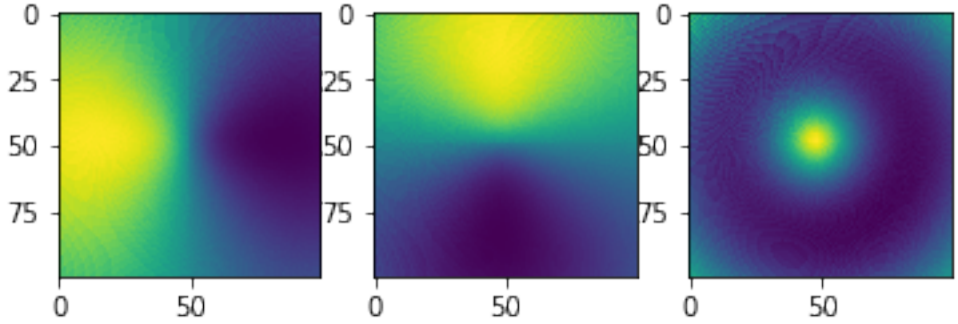
[4]:
```
from mpl_toolkits.mplot3d import Axes3D

pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

lights = np.vstack((data["l1"], data["l2"], data["l4"]))
# lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))

images = []
images.append(data["im1"])
images.append(data["im2"])
# images.append(data["im3"])
images.append(data["im4"])
images = np.array(images)

mask = np.ones(data["im1"].shape)

albedo, normals, depth, horn = photometric_stereo(images, lights, mask)

# ---------------------------------------------------------------------------
# The following code is just a working example so you don't get stuck with any
# of the graphs required. You may want to write your own code to align the
# results in a better layout. You are also free to change the function
# however you wish; just make sure you get all of the required outputs.
# ---------------------------------------------------------------------------

def visualize(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 15

    # showing albedo map
    print("showing albedo map:")
    fig = plt.figure()
    albedo_max = albedo.max()
    albedo = albedo / albedo_max
    plt.imshow(albedo, cmap="gray")
    plt.show()

    # showing normals as three separate channels
    print("showing normals as three seperate channels:")
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normals[..., 0])
```

```
    ax2 = figure.add_subplot(132)
    ax2.imshow(normals[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normals[..., 2])
    plt.show()

    # showing normals as quiver
    print("showing normals as quiver:")
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 15),
                          np.arange(0,np.shape(normals)[1], 15),
                          np.arange(1))
    X = X[..., 0]
    Y = Y[..., 0]
    Z = depth[::stride,::stride].T
    NX = normals[..., 0][::stride,::-stride].T
    NY = normals[..., 1][::-stride,::stride].T
    NZ = normals[..., 2][::stride,::stride].T
    fig = plt.figure(figsize=(5, 5))
    ax = fig.gca(projection='3d')
    plt.quiver(X,Y,Z,NX,NY,NZ, length=10)
    plt.show()

    # plotting wireframe depth map
    print("plotting wirefram depth map:")
    H = depth[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

    H = horn[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

visualize(albedo, normals, depth, horn)
```

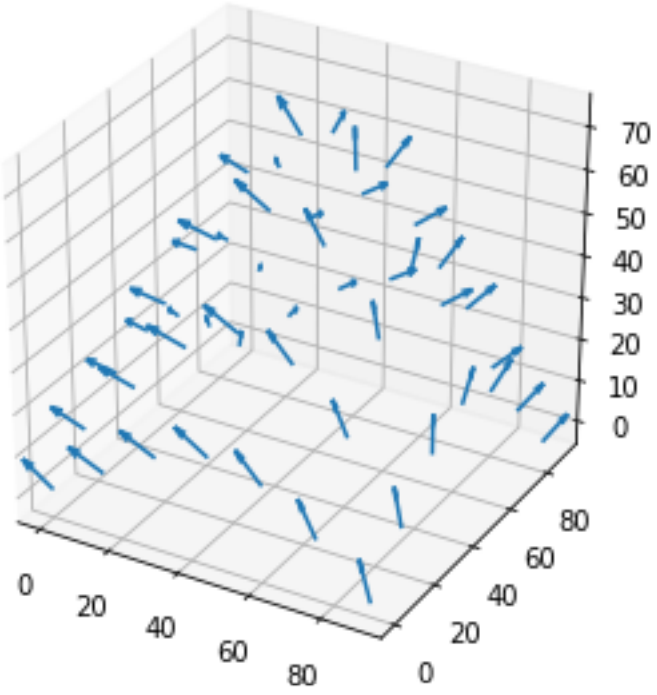
```
showing albedo map:
```



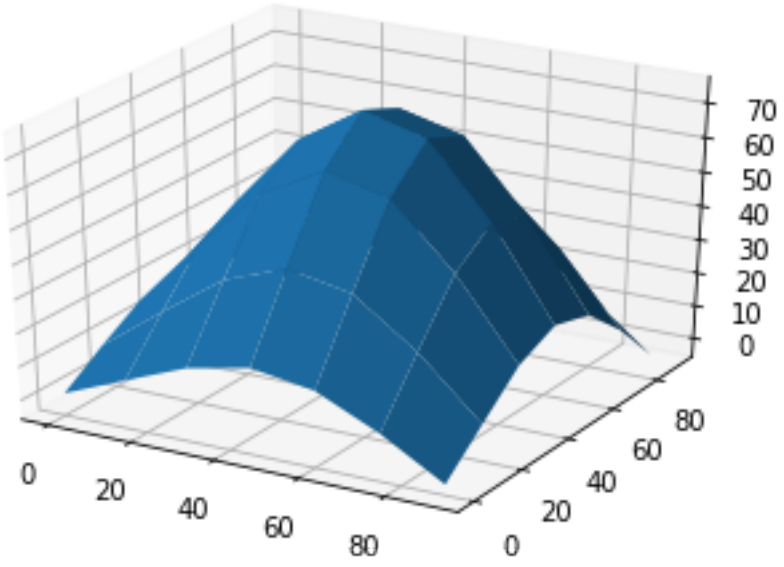
showing normals as three seperate channels:

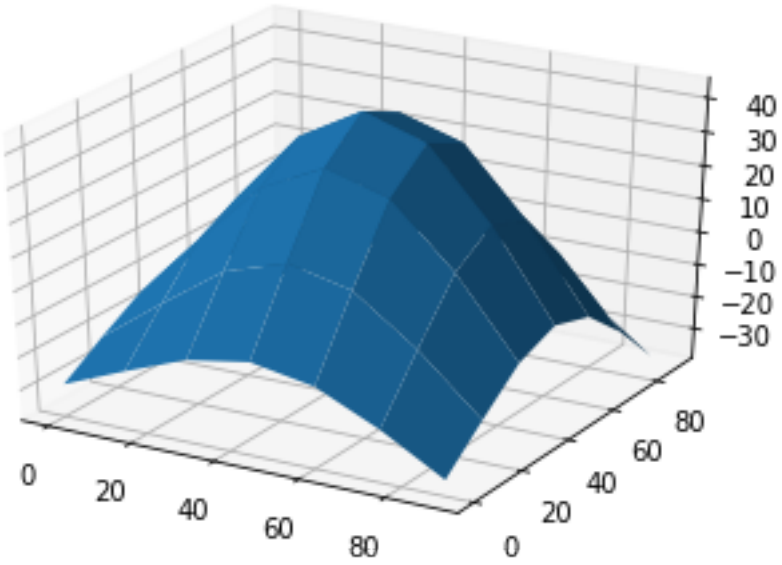


showing normals as quiver:



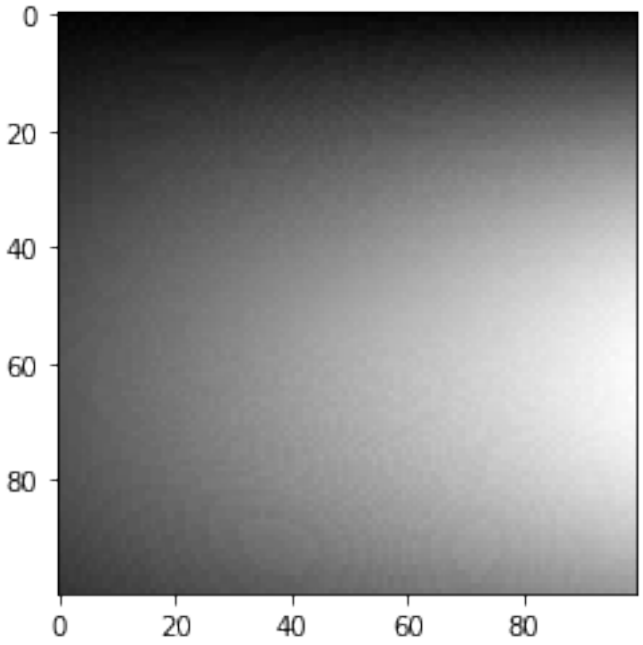
plotting wirefram depth map:



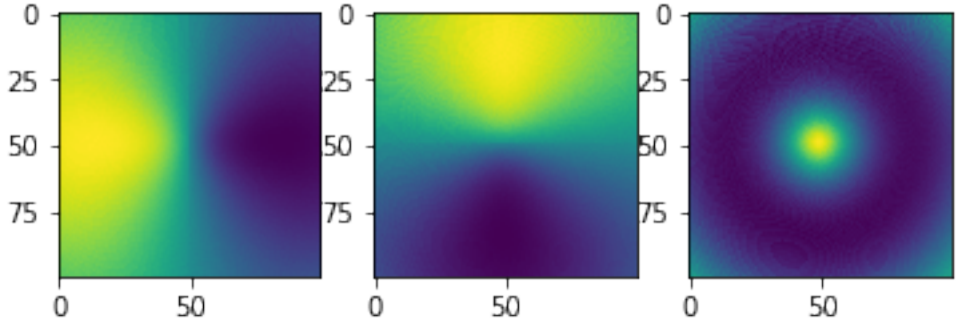


[5]:
```
# Don't forget to run your photometric stereo code on TWO sets of images!
# (One being {im1, im2, im4}, and the other being {im1, im2, im3, im4}.)

lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))

images = []
images.append(data["im1"])
images.append(data["im2"])
images.append(data["im3"])
images.append(data["im4"])
images = np.array(images)

mask = np.ones(data["im1"].shape)

albedo, normals, depth, horn = photometric_stereo(images, lights, mask)

visualize(albedo, normals, depth, horn)
```

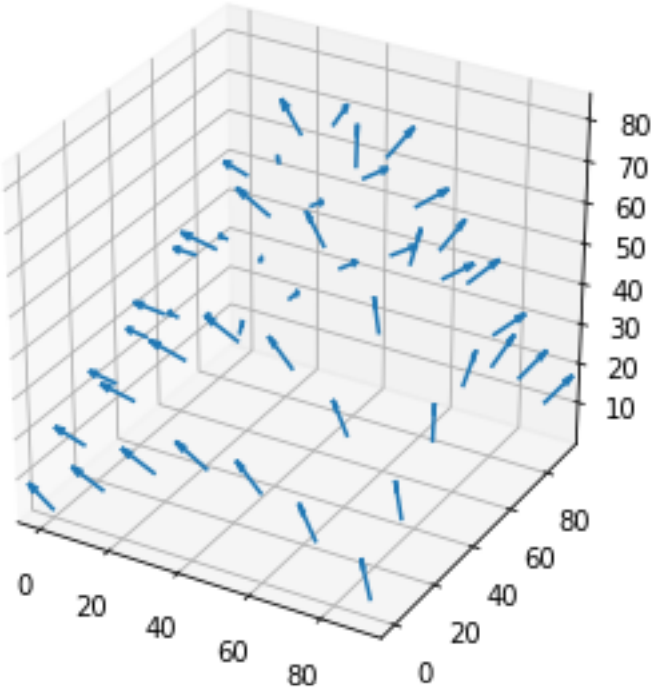
showing albedo map:



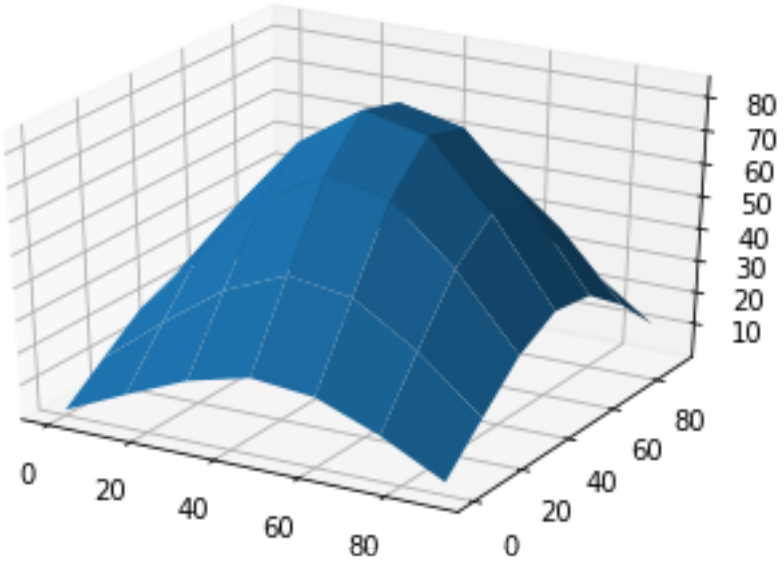
showing normals as three seperate channels:

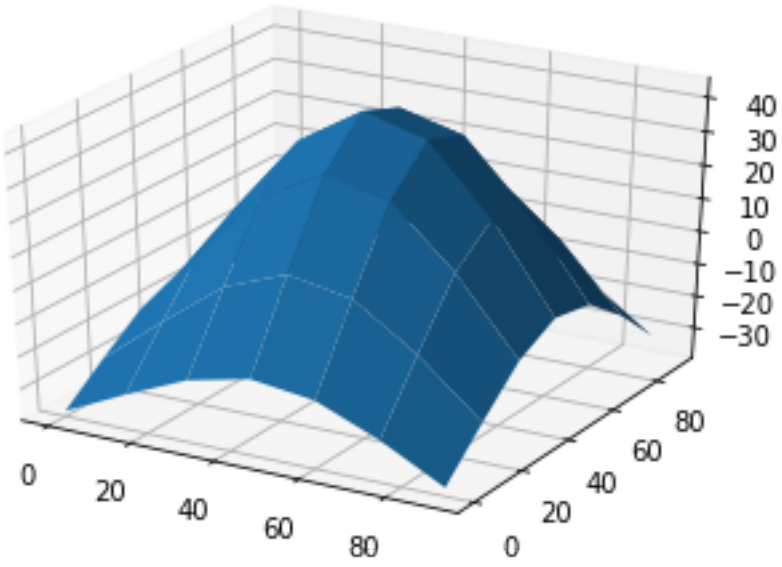


showing normals as quiver:



plotting wirefram depth map:





1.3.3 Part 2: Specularity Removal [6 pts]

Implement the specularity removal technique described in *Beyond Lambert: Reconstructing Specular Surfaces Using Color* (by Mallick, Zickler, Kriegman, and Belhumeur; CVPR 2005).

Your program should input an RGB image and light source color and output the corresponding SUV image.

Try this out first with the specular sphere images and then with the pear images.

For each of the specular sphere and pear images, include

1. The original image (in RGB colorspace).
2. The recovered S channel of the image.
3. The recovered diffuse part of the image. Use $G = \sqrt{U^2 + V^2}$ to represent the diffuse part.

In total, we expect 2 * 3 = 6 images as outputs for this problem.

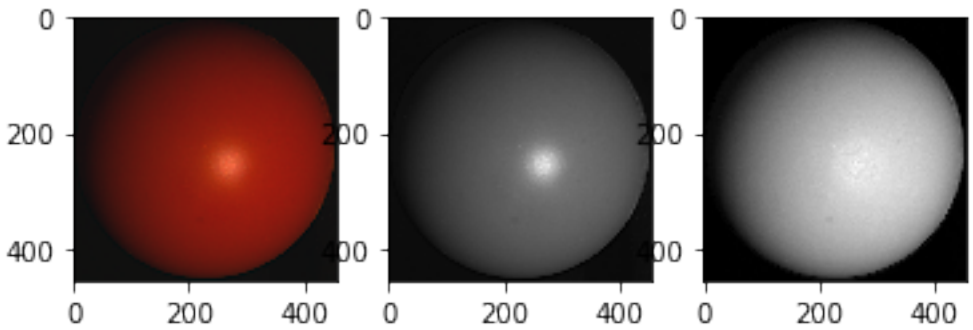
Note: You will find all the data for this part in `specular_sphere.pickle` and `specular_pear.pickle`.

[6]:
```
def get_rot_mat(rot_v, unit=None):
    '''
    Takes a vector and returns the rotation matrix required to align the
    unit vector(2nd arg) to it.
    '''
    if unit is None:
        unit = [1.0, 0.0, 0.0]

    rot_v = rot_v/np.linalg.norm(rot_v)
    uvw = np.cross(rot_v, unit) # axis of rotation
```

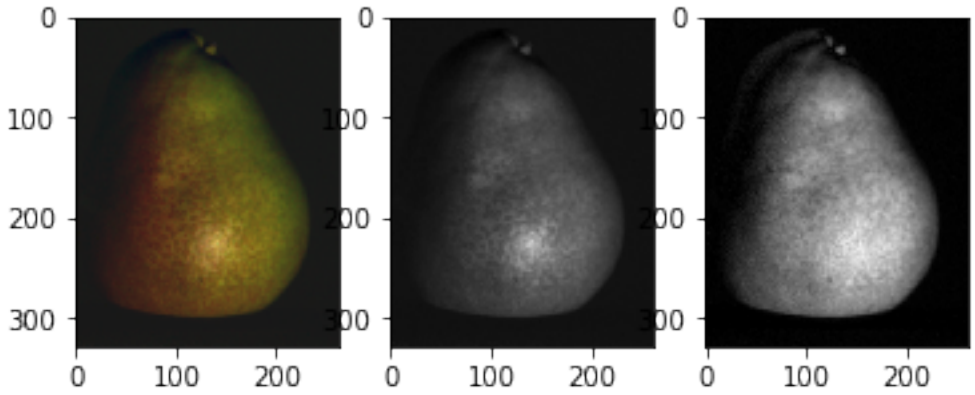
```
    rcos = np.dot(rot_v, unit) # cos by dot product
    rsin = np.linalg.norm(uvw) # sin by magnitude of cross product

    # normalize and unpack axis
    if not np.isclose(rsin, 0):
        uvw = uvw/rsin
    u, v, w = uvw

    # compute rotation matrix
    R = (
        rcos * np.eye(3) +
        rsin * np.array([
            [ 0, -w,  v],
            [ w,  0, -u],
            [-v,  u,  0]
        ]) +
        (1.0 - rcos) * uvw[:,None] * uvw[None,:]
    )
    return R

def RGBToSUV(I_rgb, rot_vec):
    '''
    Your implementation which takes an RGB image and a vector encoding
    the orientation of the S channel w.r.t. to RGB.
    '''
    S = np.ones(I_rgb.shape[:2])
    G = np.ones(I_rgb.shape[:2])

    R = get_rot_mat(rot_vec)

    for i in range(I_rgb.shape[0]):
        for j in range(I_rgb.shape[1]):
            I_suv = np.dot(R,I_rgb[i,j,:])
            S[i,j] = I_suv[0]
            G[i,j] = np.linalg.norm(I_suv[1:])

    return S, G

pickle_in_sphere = open("specular_sphere.pickle", "rb")
sphere_data = pickle.load(pickle_in_sphere, encoding="latin1")

pickle_in_pear = open("specular_pear.pickle","rb")
pear_data = pickle.load(pickle_in_pear,encoding="latin1")

# sample input
#S, G = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
```

```
#                                          data["c"][1][0],
#                                          data["c"][2][0])))

S_sphere1, G_sphere1 = RGBToSUV(sphere_data["im1"],np.
 ↪hstack((sphere_data["c"][0][0],
                                                          ␣
 ↪sphere_data["c"][1][0],
                                                          ␣
 ↪sphere_data["c"][2][0])))

print("The original, S channel and Diffuse part of Sphere img1:")
plt.subplot(131)
plt.imshow((sphere_data["im1"]-np.min(sphere_data["im1"]))/(np.
 ↪max(sphere_data["im1"])-np.min(sphere_data["im1"])))
plt.subplot(132)
plt.imshow(S_sphere1,cmap='gray')
plt.subplot(133)
plt.imshow(G_sphere1,cmap='gray')
plt.show()

print("The original, S channel and Diffuse part of Pear img1:")
S_pear1, G_pear1 = RGBToSUV(pear_data["im1"],np.hstack((pear_data["c"][0][0],
                                                        pear_data["c"][1][0],
                                                        pear_data["c"][2][0])))

plt.subplot(131)
plt.imshow((pear_data["im1"]-np.min(pear_data["im1"]))/(np.
 ↪max(pear_data["im1"])-np.min(pear_data["im1"])))
plt.subplot(132)
plt.imshow(S_pear1,cmap='gray')
plt.subplot(133)
plt.imshow(G_pear1,cmap='gray')
plt.show()
```

The original, S channel and Diffuse part of Sphere img1:



`The original, S channel and Diffuse part of Pear img1:`



1.3.4 Part 3: Robust Photometric Stereo [6 pts]

Now we will perform photometric stereo on our sphere/pear images which include specularities. First, for comparison, run your photometric stereo code from 1a on the original images (converted to grayscale and rescaled/shifted to be in the range [0, 1]). You should notice erroneous "bumps" in the resulting reconstructions, as a result of violating the Lambertian assumption. For this, show the same outputs as in 1a.

Next, combine parts 1 and 2 by removing the specularities (using your code from 1b) and then running photometric stereo on the diffuse components of the specular sphere/pear images. Our goal will be to remove the bumps/sharp parts in the reconstruction.

For the specular sphere image set in `specular_sphere.pickle`, using all of the four images (again, be sure to convert them to grayscale and normalize them so that their values go from 0 to 1), include:

1. The estimated albedo map (original and diffuse).
2. The estimated surface normals (original and diffuse) by showing both
    1. Needle map, and
    2. Three images showing each of the surface normal components.
3. A wireframe of depth map (original and diffuse).
4. A wireframe of the depth map given by Horn integration (original and diffuse).

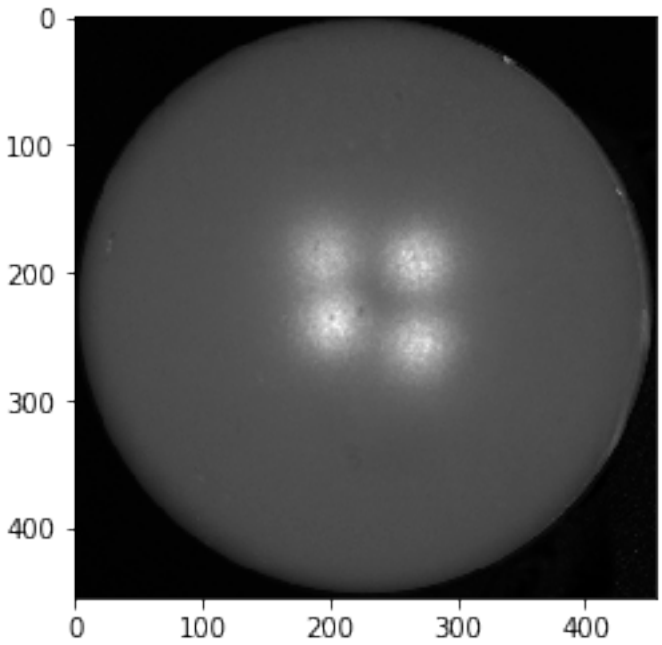
In total, we expect 2 * 7 = 14 images for the 1a comparison, plus 2 * 7 = 14 images for the outputs after specularity removal has been performed. (Thus 28 output images overall.)

[7]:
```
# ------------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals, and depth maps.
# ------------------------------------------------------------------------

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

def normalize(img):
    maxi = img.max()
    mini = img.min()
    return (img - mini)/(maxi-mini)

def calculate_mask(img,threshold = 0.05):
    img_mean = (img[0] + img[1] + img[2] + img[3])/4
    img_norm = normalize(img_mean)
    mask = np.ones(img_norm.shape)
    for i in range(img_mean.shape[0]):
        for j in range(img_mean.shape[1]):
            if(img_norm[i,j] < threshold):
                mask[i,j] = 0

    return mask

#sphere_data
lights_sphere = np.vstack((sphere_data["l1"], sphere_data["l2"],␣
 ↪sphere_data["l3"], sphere_data["l4"]))

im1_s_gray = rgb2gray(sphere_data["im1"])
im2_s_gray = rgb2gray(sphere_data["im2"])
im3_s_gray = rgb2gray(sphere_data["im3"])
im4_s_gray = rgb2gray(sphere_data["im4"])

images_s = []
images_s.append(im1_s_gray)
images_s.append(im2_s_gray)
images_s.append(im3_s_gray)
images_s.append(im4_s_gray)
images_s = np.array(images_s,dtype=float)

mask_s = calculate_mask(images_s,0.1)

albedo_s, normals_s, depth_s, horn_s = photometric_stereo(images_s,␣
 ↪lights_sphere, mask_s,1000)
visualize(albedo_s, normals_s, depth_s, horn_s)
```

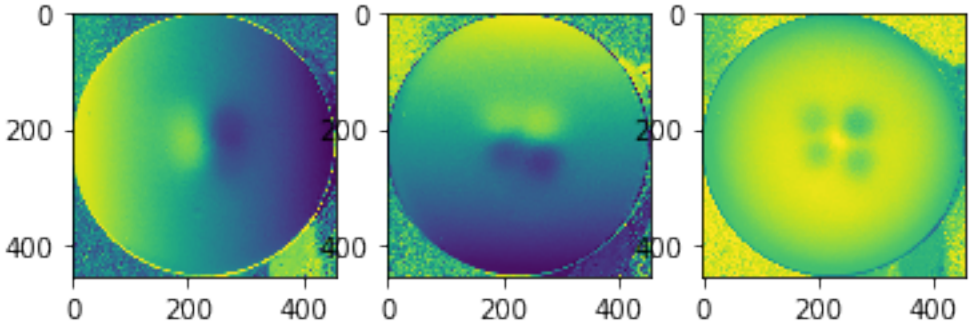
```
S_sphere2, G_sphere2 = RGBToSUV(sphere_data["im2"],np.
 ↪hstack((sphere_data["c"][0][0],sphere_data["c"][1][0],sphere_data["c"][2][0])))
S_sphere3, G_sphere3 = RGBToSUV(sphere_data["im3"],np.
 ↪hstack((sphere_data["c"][0][0],sphere_data["c"][1][0],sphere_data["c"][2][0])))
S_sphere4, G_sphere4 = RGBToSUV(sphere_data["im4"],np.
 ↪hstack((sphere_data["c"][0][0],sphere_data["c"][1][0],sphere_data["c"][2][0])))

images_sg = []
images_sg.append(np.ndarray.tolist(G_sphere1))
images_sg.append(np.ndarray.tolist(G_sphere2))
images_sg.append(np.ndarray.tolist(G_sphere3))
images_sg.append(np.ndarray.tolist(G_sphere4))
images_sg = np.array(images_sg,dtype=float)

albedo_sg,normals_sg,depth_sg,horn_sg =␣
 ↪photometric_stereo(images_sg,lights_sphere,mask_s,1000)
visualize(albedo_sg,normals_sg,depth_sg,horn_sg)

#pear_data
lights_pear = np.vstack((pear_data["l1"], pear_data["l2"], pear_data["l3"],␣
 ↪pear_data["l4"]))

im1_p_gray = rgb2gray(pear_data["im1"])
im2_p_gray = rgb2gray(pear_data["im2"])
im3_p_gray = rgb2gray(pear_data["im3"])
im4_p_gray = rgb2gray(pear_data["im4"])

images_p = []
images_p.append(im1_p_gray)
images_p.append(im2_p_gray)
images_p.append(im3_p_gray)
images_p.append(im4_p_gray)
images_p = np.array(images_p,dtype=float)

mask_p = calculate_mask(images_p,0.1)

albedo_p, normals_p, depth_p, horn_p = photometric_stereo(images_p, lights_pear,␣
 ↪mask_p,1000)
visualize(albedo_p, normals_p, depth_p, horn_p)

S_pear2, G_pear2 = RGBToSUV(pear_data["im2"],np.
 ↪hstack((pear_data["c"][0][0],pear_data["c"][1][0],pear_data["c"][2][0])))
S_pear3, G_pear3 = RGBToSUV(pear_data["im3"],np.
 ↪hstack((pear_data["c"][0][0],pear_data["c"][1][0],pear_data["c"][2][0])))
S_pear4, G_pear4 = RGBToSUV(pear_data["im4"],np.
 ↪hstack((pear_data["c"][0][0],pear_data["c"][1][0],pear_data["c"][2][0])))
```

```
images_pg = []
images_pg.append(np.ndarray.tolist(G_pear1))
images_pg.append(np.ndarray.tolist(G_pear2))
images_pg.append(np.ndarray.tolist(G_pear3))
images_pg.append(np.ndarray.tolist(G_pear4))
images_pg = np.array(images_pg,dtype=float)

albedo_pg,normals_pg,depth_pg,horn_pg =␣
 ↪photometric_stereo(images_pg,lights_pear,mask_p,1000)
visualize(albedo_pg,normals_pg,depth_pg,horn_pg)
```

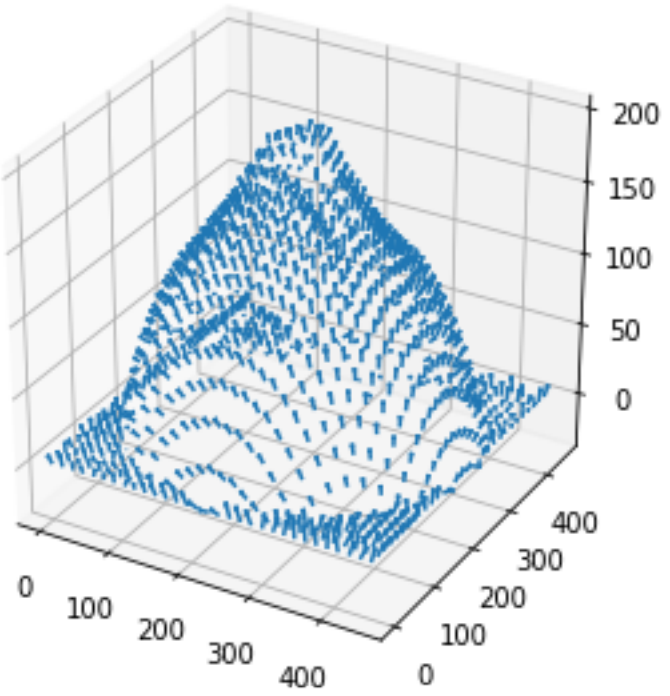
showing albedo map:



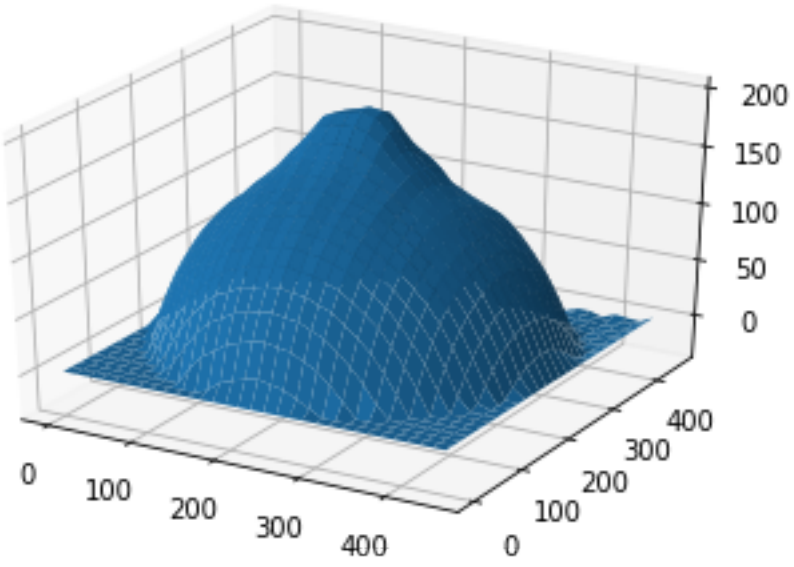
showing normals as three seperate channels:

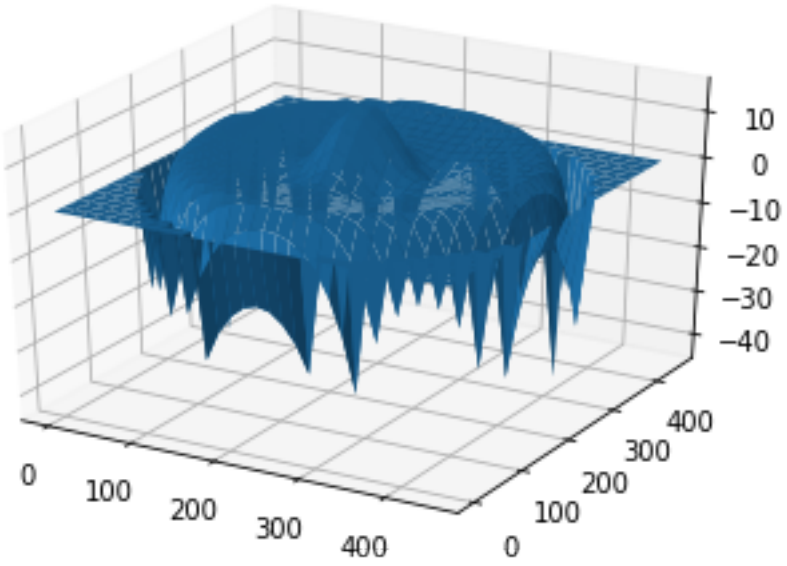


showing normals as quiver:

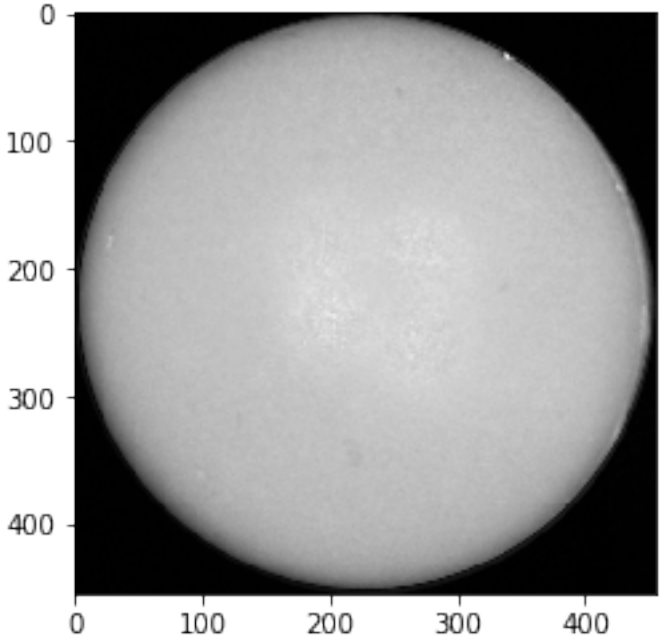


plotting wirefram depth map:

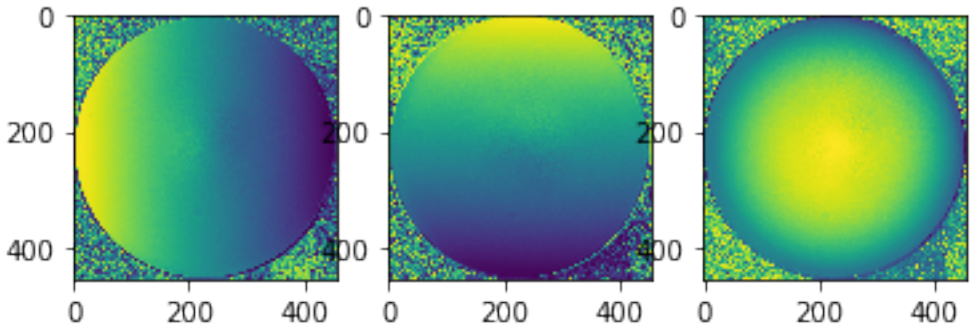




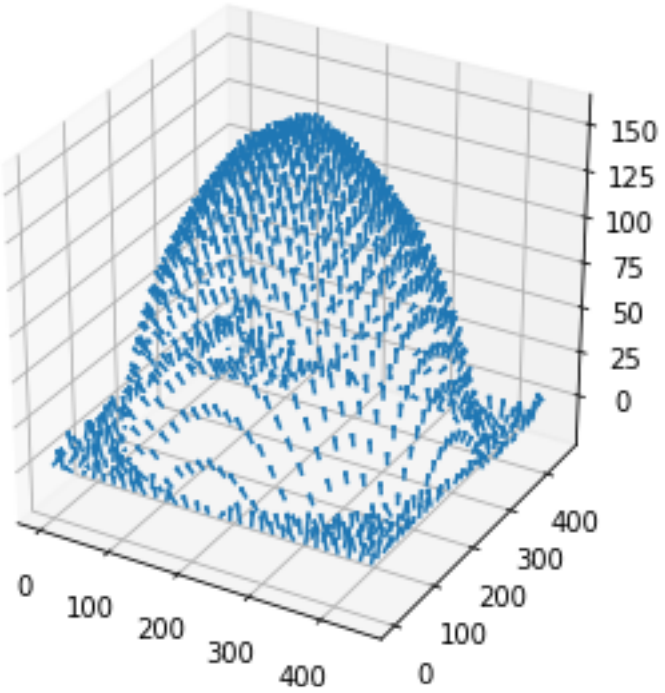
showing albedo map:



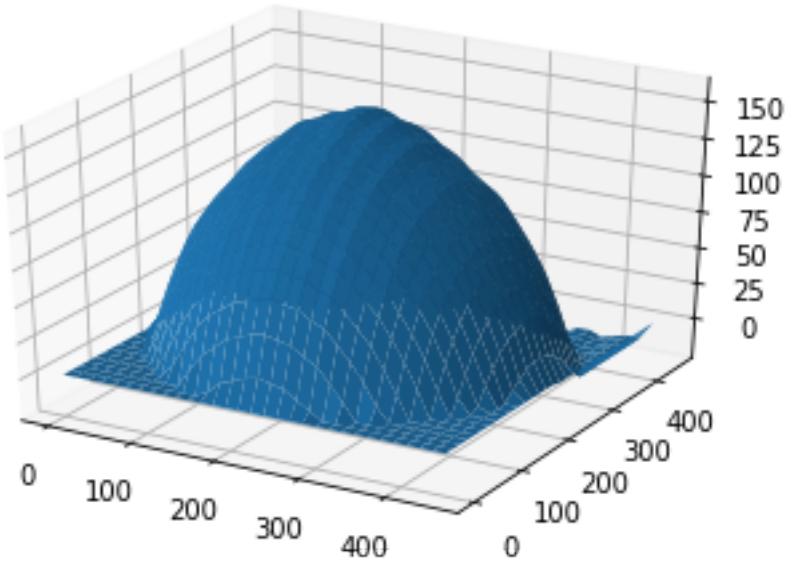
showing normals as three seperate channels:

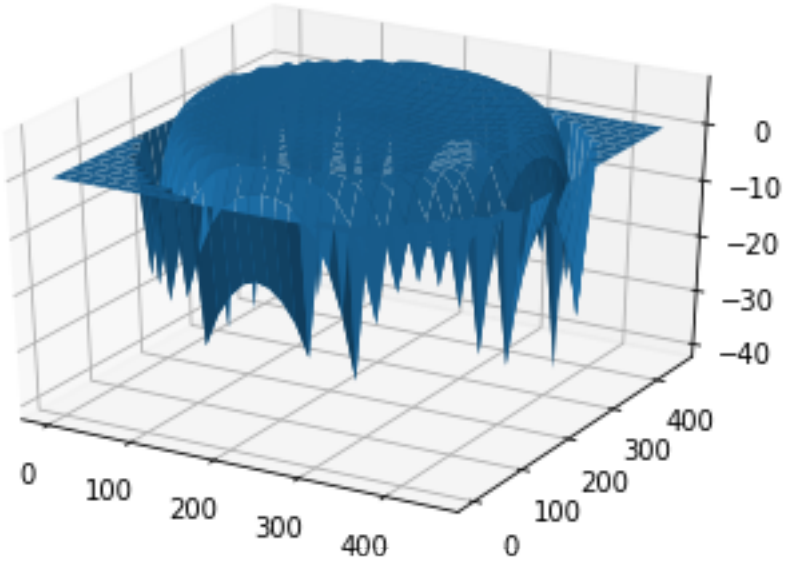


showing normals as quiver:

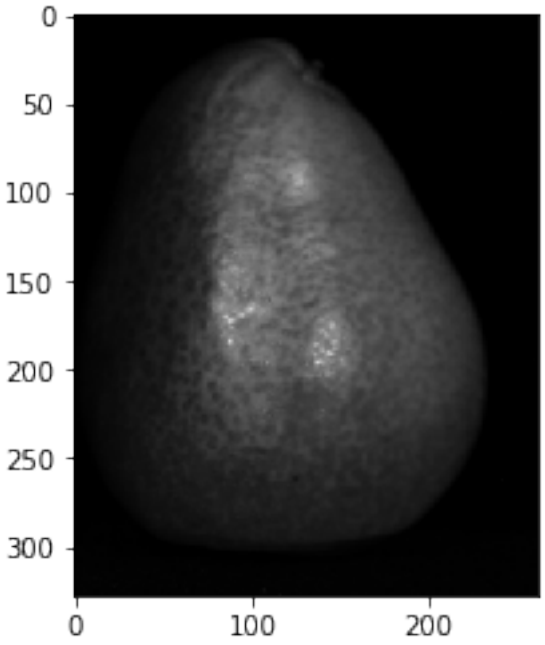


plotting wirefram depth map:

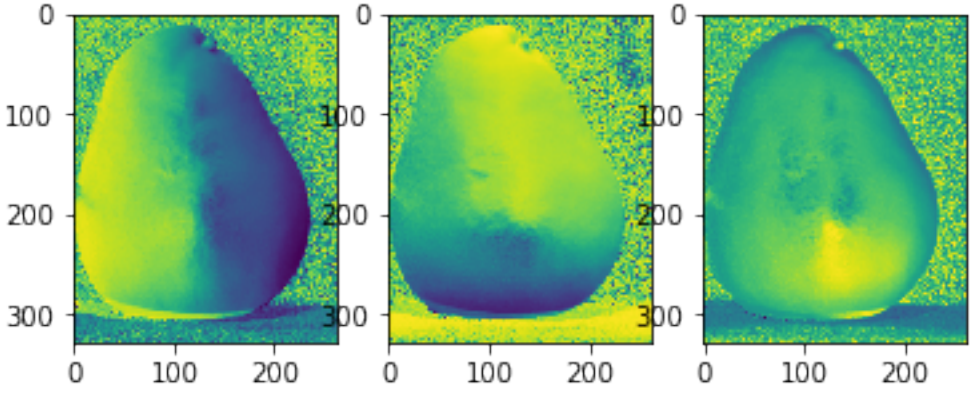




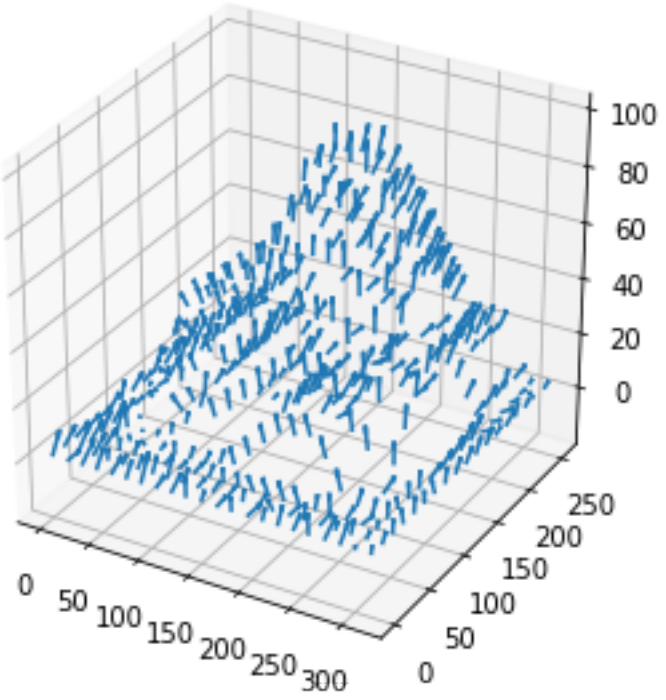
showing albedo map:



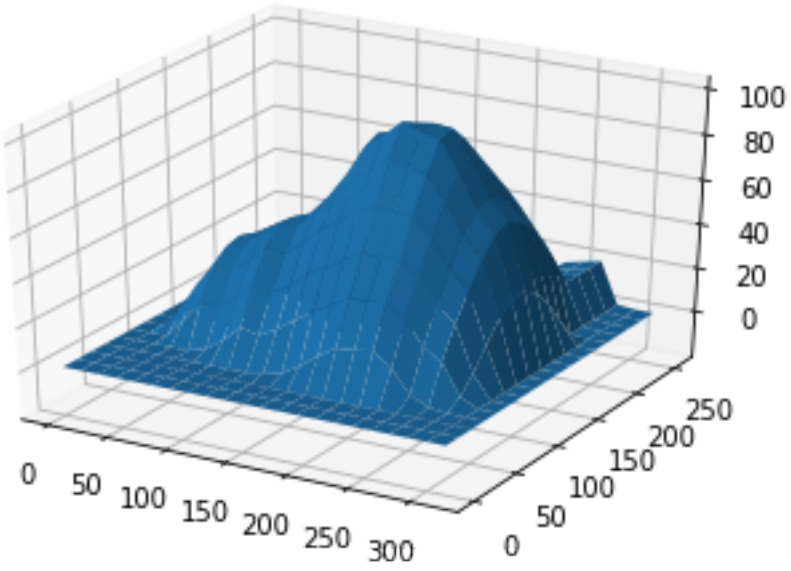
showing normals as three seperate channels:

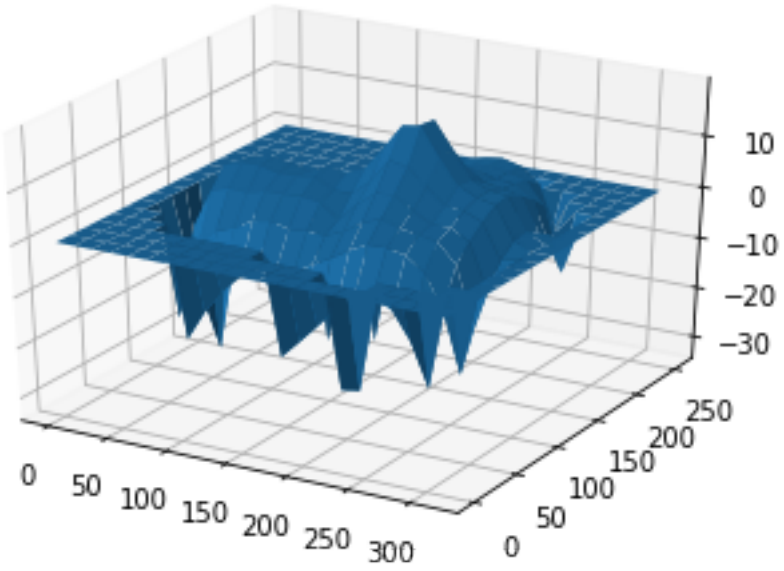


showing normals as quiver:

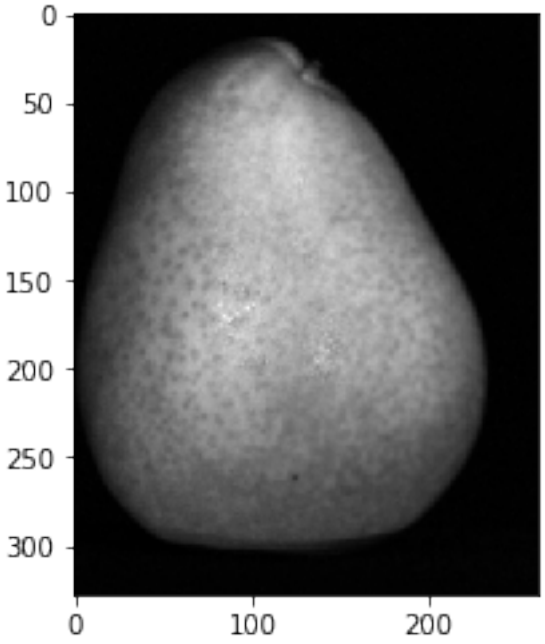


plotting wirefram depth map:

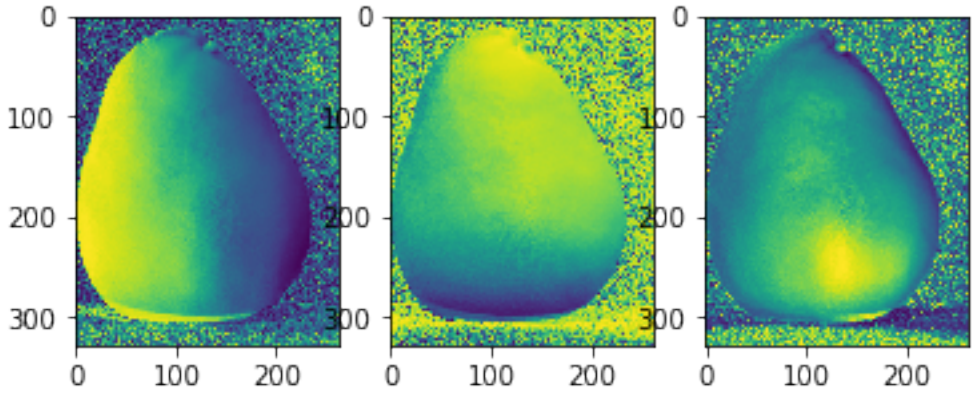




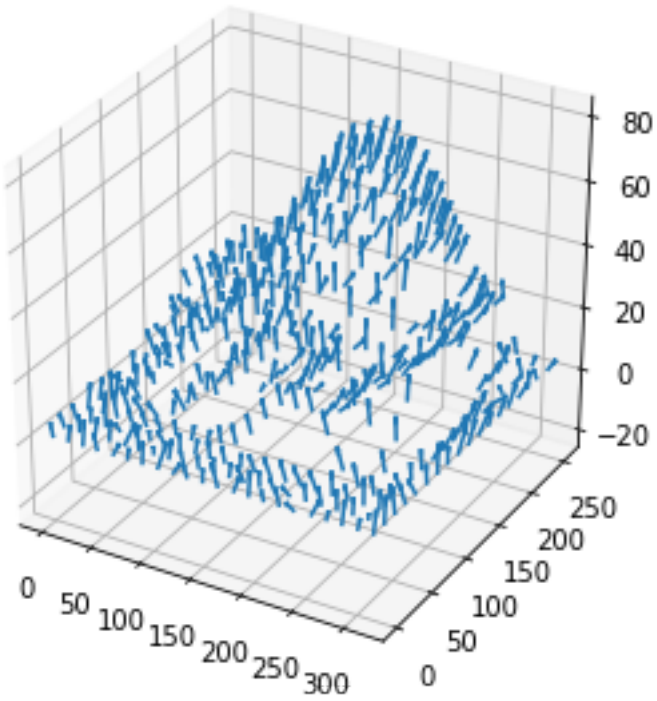
showing albedo map:



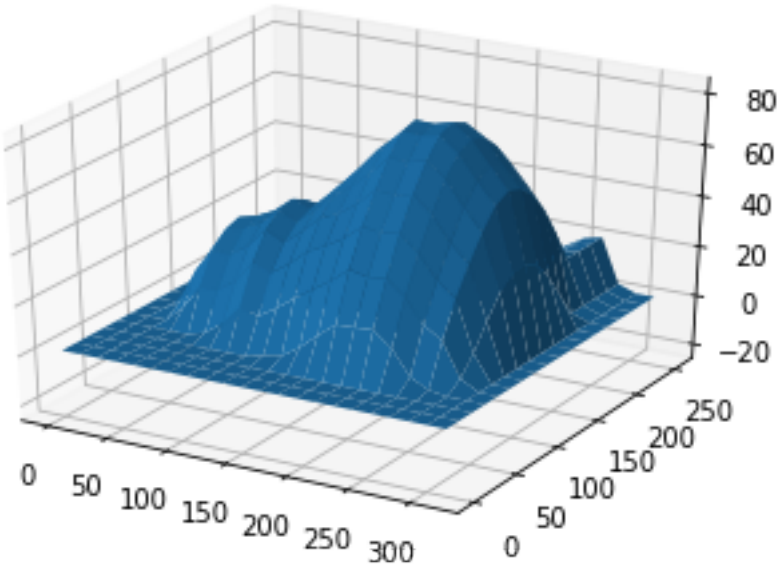
showing normals as three seperate channels:

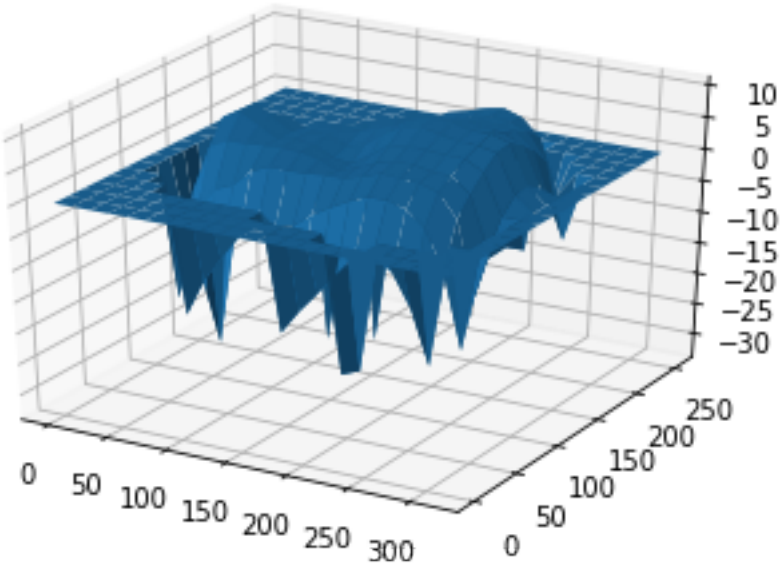


showing normals as quiver:



plotting wirefram depth map:





1.4 Problem 2: Image Filtering [13 pts]

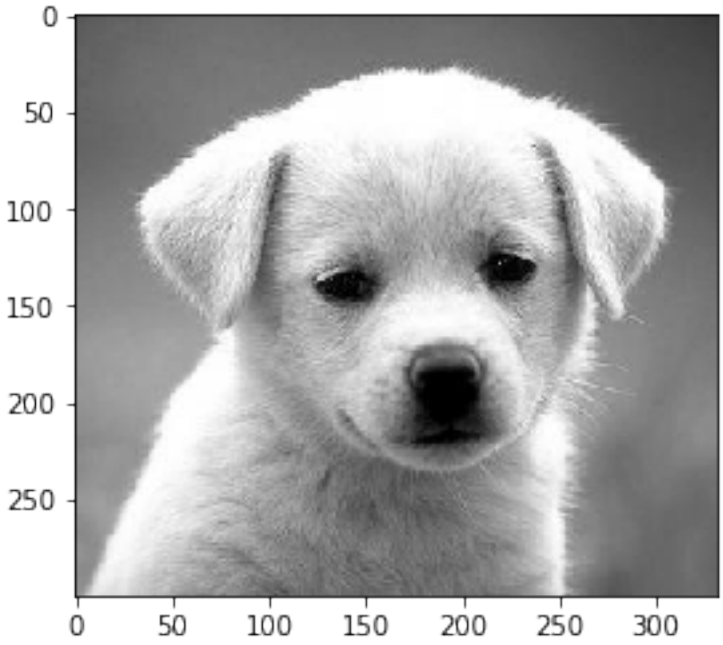
1.4.1 Part 1: Warmup [1.5 pts]

In this problem, we expect you to use convolution to filter the provided image with three different types of kernels:

1. A 5x5 Gaussian filter with $\sigma = 5$.
2. A 31x31 Gaussian filter with $\sigma = 5$.
3. A sharpening filter.

This is the image you will be using:

[8]:
```
# Open image as grayscale
dog_img = io.imread('dog.jpg', as_gray=True)

# Show image
plt.imshow(dog_img, cmap='gray')
plt.show()
```



For convenience, we have provided a helper function for creating a square isotropic Gaussian kernel. We have also provided the sharpening kernel that you should use. Finally, we have provided a function to help you plot the original and filtered results side-by-side. Take a look at each of these before you move on.

[9]:
```
def gaussian2d(filter_size=5, sig=1.0):
    """Creates a 2D Gaussian kernel with
    side length `filter_size` and a sigma of `sig`."""
    ax = np.arange(-filter_size // 2 + 1., filter_size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sig))

    return kernel / np.sum(kernel)

sharpening_kernel = np.array([
```

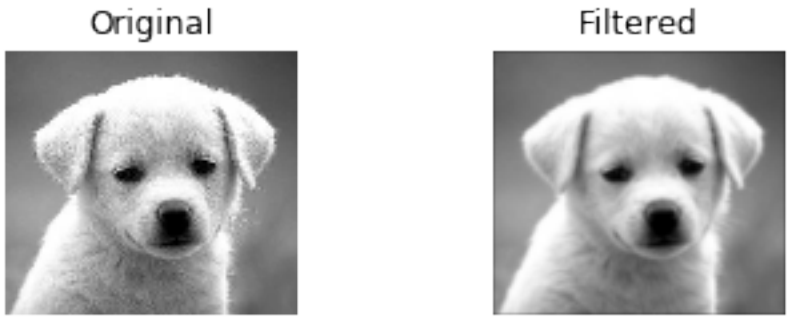
```
    [1, 4,     6,  4, 1],
    [4, 16,   24, 16, 4],
    [6, 24, -476, 24, 6],
    [4, 16,   24, 16, 4],
    [1,  4,    6,  4, 1],
]) * -1.0 / 256.0

def plot_results(original, filtered):
    # Plot original image
    plt.subplot(2,2,1)
    plt.imshow(original, cmap='gray',vmin=0.0, vmax=1.0)
    plt.title('Original')
    plt.axis('off')

    # Plot filtered image
    plt.subplot(2,2,2)
    plt.imshow(filtered, cmap='gray',vmin=0.0, vmax=1.0)
    plt.title('Filtered')
    plt.axis('off')

    plt.show()
```

Now fill in the functions below and display outputs for each of the filtering results. There should be three sets of (original, filtered) outputs in total. You are allowed to use the imported `convolve` function.

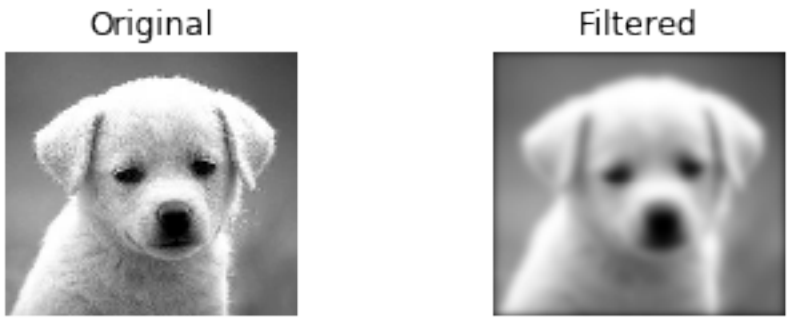
[10]:
```
from scipy.signal import convolve

def filter1(img):
    """Convolve the image with a 5x5 Gaussian filter with sigma=5."""
    kernel = gaussian2d(5,5.)
    img_f = convolve(img,kernel,mode="same")

    return img_f

def filter2(img):
    """Convolve the image with a 31x31 Gaussian filter with sigma=5."""
    kernel = gaussian2d(31,5.)
    img_f = convolve(img,kernel,mode="same")

    return img_f

def filter3(img):
    """Convolve the image with the provided sharpening filter."""
    img_f = convolve(img,sharpening_kernel,mode="same")

    return img_f
```

```
for filter_name, filter_fn in [
    ('5x5 Gaussian filter, sigma=5', filter1),
    ('31x31 Gaussian filter, sigma=5', filter2),
    ('sharpening filter', filter3),
]:
    filtered = filter_fn(dog_img)
    print(filter_name)
    plot_results(dog_img, filtered)
```

```
5x5 Gaussian filter, sigma=5
```




```
31x31 Gaussian filter, sigma=5
```




```
sharpening filter
```

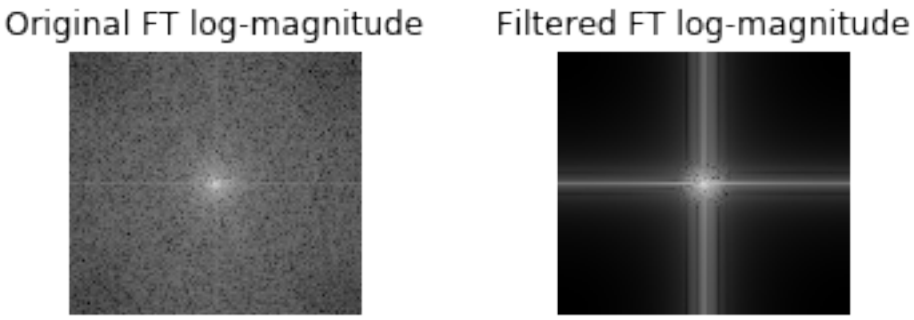
1.4.2 Part 2 [2 pts]

Display the Fourier log-magnitude transform image for the (original image, 31x31 Gaussian-filtered image) pair. (No need to include the others.) We have provided the code to compute the Fourier log-magnitude image.

Then, as a text answer, explain the differences you see between the original frequency domain image and the 31x31 Gaussian-filtered frequency domain image. In particular, be sure to address the following points: - Why is most of the frequency visualization dark after applying the Gaussian filter, and what does this signify? - What is an example of one of these dark images in the spatial domain (original image)? - What do the remaining bright regions in the magnitude image represent? - What is an example of one of these bright regions in the spatial domain (original image)?

[11]:
```
# Visualize the frequency domain images

def plot_ft_results(img1, img2):
    plt.subplot(2,2,1)
    plt.imshow(img1, cmap='gray')
    plt.title('Original FT log-magnitude')
    plt.axis('off')

    plt.subplot(2,2,2)
    plt.imshow(img2, cmap='gray')
    plt.title('Filtered FT log-magnitude')
    plt.axis('off')

    plt.show()

def ft_log_magnitude(img_gray):
    return np.log(np.abs(np.fft.fftshift(np.fft.fft2(img_gray))))

print('31x31 Gaussian filter, sigma=5')
plot_ft_results(ft_log_magnitude(dog_img), ft_log_magnitude(filter2(dog_img)))
```

```
31x31 Gaussian filter, sigma=5
```



1.4.3 Your answer to Problem 2.2:

- The darker area of 'Filtered FT log-magnitude' image, with respect to the oringinal image, represents the edges (high frequency parts) of the dog figure. While the 'Gaussian filter' is a smoothing filter, which blurs the sharp edge. Thus, the darker the outskirt of the filtered image is, the smoother the edge of the oringinal image would be.
- The outskrit of the dog can be an example that represents the dark area of the FT transformed image.
- The remaining bright regions ,which contain the low frequency features of the FT image, implies that the coefficient of the low frequency item is relatively large. And the low frequency items usually represents the main features (e.g. color, texture and etc.) of the oringinal image.
- The forehead and the top of the head remains relatively the same after operated with Gaussian filter, thus it may somehow correspond to the bright region of the FT image.

1.4.4 Part 3 [3 pts]

Consider (1) smoothing an image with a 3x3 box filter and then computing the derivative in the y-direction (use the derivative filter from Lecture 7). Also consider (2) computing the derivative first, then smoothing. What is a single convolution kernel that will simultaneously implement both (1) and (2)?

1.4.5 Your answer to Problem 2.3:

Suppose 1) a 3x3 box filter is:

$$k_s = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

and the derivative in the y-direction is:

$$\frac{d}{dy} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}^T.$$

The derivative operation can be considered as a convolution kernel. Thus, apply the commutative law and the associative law of convolution,(1) and (2) are equivalent. In other words,

$$Gauss(\frac{\partial}{\partial y}Image) = \frac{\partial}{\partial y}Gauss(Image),$$

and a single convolution kernel can be a matrix, whose each element is the product of both corresponding derivative kernel and smoothing kernel.

1.4.6 Part 4 [3 pts]

Give an example of a 3x3 separable filter and compare the number of arithmetic operations it takes to convolve using that filter on an n $\times$ n image before and after separation.

1.4.7 Your answer to Problem 2.4:

Given a 3x3 separable filter:

$$F = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix} \times \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix}$$

- convolve before separation: $(9+8) \times (n-2)^2 = 17n^2 - 68n + 68$
- convolve after separation: $2 \times ((3+2) \times (n-2) \times n = 10n^2 - 20n$

1.4.8 Part 5: Filters as Templates [3.5 pts]

Suppose that you are a clerk at a grocery store. One of your responsibilites is to check the shelves periodically and stock them up whenever there are sold-out items. You got tired of this laborious task and decided to build a computer vision system that keeps track of the items on the shelf.

Luckily, you have learned in CSE 252A (or are learning right now) that convolution can be used for template matching: a template g is multiplied with regions of a larger image f to measure how similar each region is to the template. Note that you will want to flip the filter before giving it to your convolution function, so that it is overall not flipped when making comparisons. You will also want to subtract off the mean value of the image or template (whichever you choose, subtract the same value from both the image and template) so that your solution is not biased toward higher-intensity (white) regions.

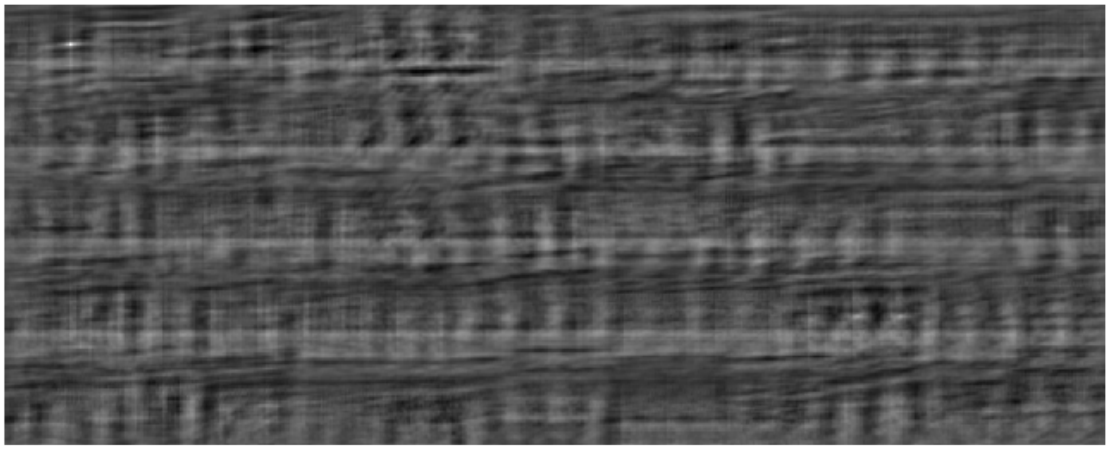
The template of a product (template.jpg) and the image of the shelf (shelf.jpg) is provided. We will use convolution to find the product in the shelf.

[12]:
```
import numpy as np
from skimage import io
import matplotlib.pyplot as plt
from scipy.signal import convolve
%matplotlib inline

# Load template and image in grayscale
img = io.imread('shelf.jpg')
img_gray = io.imread('shelf.jpg', as_gray=True)
temp = io.imread('template.jpg')
```

```
temp_gray = io.imread('template.jpg', as_gray=True)

# Perform a convolution between the image and the template
""" ==========
YOUR CODE HERE
========== """
out = np.zeros_like(img_gray)
img_gray_mean = np.mean(img_gray,dtype=np.float32)
temp_gray_mean = np.mean(temp_gray,dtype=np.float32)
img_gray_new = img_gray - img_gray_mean
kernel = np.rot90(temp_gray,2)-temp_gray_mean
out = convolve(img_gray_new,kernel,mode='same')

# Find the location with maximum similarity
y, x = (np.unravel_index(out.argmax(), out.shape))

# Display product template
plt.figure(figsize=(20,16))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display convolution output
plt.subplot(3, 1, 2)
plt.imshow(out,cmap = 'gray')
plt.title('Convolution output (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

Template

Convolution output (white means more correlated)



Result (blue marker on the detected location)

1.5 Problem 3: Edge Detection [7 pts]

In this problem, you will write a function to perform edge detection. The following steps need to be implemented.

- **Smoothing [1 pt]:** First, we need to smooth the images to prevent noise from being considered edges. For this problem, use a 9x9 Gaussian kernel filter with $\sigma = 1.4$ to smooth the images.
- **Gradient Computation [2 pts]:** After you have finished smoothing, find the image gradient in the horizontal and vertical directions. Compute the gradient magnitude image as $|G| = \sqrt{G_x^2 + G_y^2}$.
- **Non-Maximum Suppression [4 pts]:** We would like our edges to be sharp, unlike the ones in the gradient magnitude image from above. Use local non-maximum suppression on the gradient magnitude image to suppress all local non-maximum gradient magnitude values. To see how it affects the results, try using two different window sizes: 5x5 and 21x21.

Compute the images after each step. Show each of the intermediate steps and label your images accordingly.

In total, there should be five output images (original, smoothed, gradient magnitude, NMS result with 5x5 window, NMS result with 21x21 window).

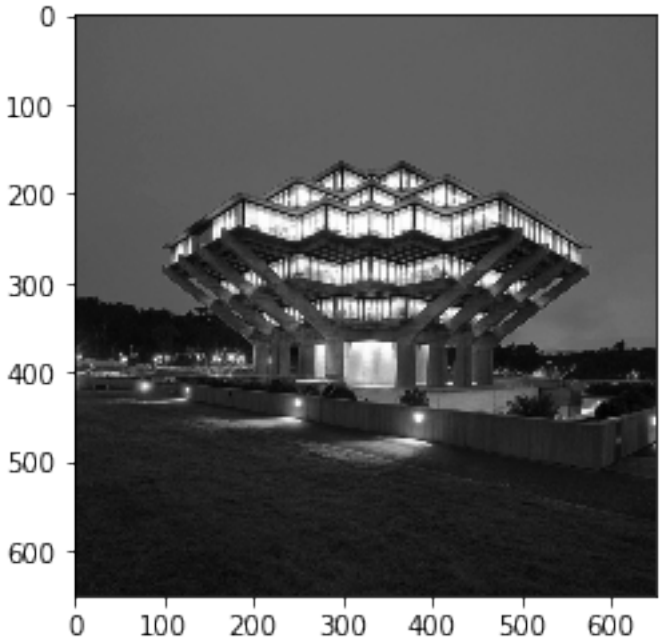
For this question, use the image `geisel.jpeg`**.**

[13]:
```
import numpy as np
from skimage import io
import matplotlib.pyplot as plt
%matplotlib inline

def smooth(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    image_tmp = convolve(image,gaussian2d(9,1.4),mode='same')

    return image_tmp

def gradient(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    g_mag = np.zeros_like(image)
    g_theta = np.zeros_like(image)
    x_kernel = np.array([[-0.5,0,0.5],
                         [-0.5,0,0.5],
                         [-0.5,0,0.5]])
    y_kernel = x_kernel.T

    Gx = convolve(image,x_kernel,mode='same')
    Gy = convolve(image,y_kernel,mode='same')
```

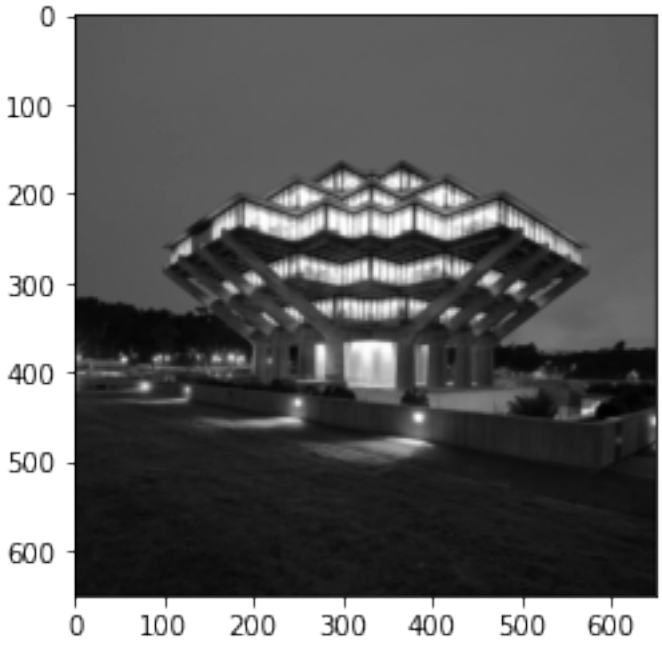
```
    g_mag = np.sqrt(Gx*Gx+Gy*Gy)
    g_theta = np.arctan(Gx/Gy)

    return g_mag, g_theta

def nms(g_mag, g_theta, window_size=5):
    """ ==========
    YOUR CODE HERE
    ========== """
    pad_width = int(0.5*window_size-0.5)
    image_tool =g_mag.copy()
    image_pad = np.pad(g_mag,(pad_width,␣
 ↪pad_width),'constant',constant_values=(0,0))

    for h in range(image_pad.shape[0]-window_size):
        for w in range(image_pad.shape[1]-window_size):
            window = image_pad[h:h+window_size,w:w+window_size]
            if (np.max(window)!=g_mag[h,w]):
                image_tool[h,w] = 0

    return image_tool

def edge_detect(image):
    """Perform edge detection on the image."""
    smoothed = smooth(image)
    g_mag, g_theta = gradient(smoothed)
    nms_image_5x5 = nms(g_mag, g_theta, window_size=5)
    nms_image_21x21 = nms(g_mag, g_theta, window_size=21)
    return smoothed, g_mag, nms_image_5x5, nms_image_21x21

# Load image in grayscale
image = io.imread('geisel.jpeg', as_gray=True)
smoothed, g_mag, nms_image_5x5, nms_image_21x21 = edge_detect(image)

print('Original:')
plt.imshow(image,cmap = 'gray')
plt.show()

print('Smoothed:')
plt.imshow(smoothed,cmap = 'gray')
plt.show()

print('Gradient magnitude:')
plt.imshow(g_mag,cmap = 'gray')
plt.show()

plt.figure(figsize = (15,15))
```

```
print('NMS with 5x5 window:')
plt.imshow(nms_image_5x5,cmap = 'gray')
plt.show()

plt.figure(figsize = (15,15))
print('NMS with 21x21 window:')
plt.imshow(nms_image_21x21,cmap = 'gray')
plt.show()
```

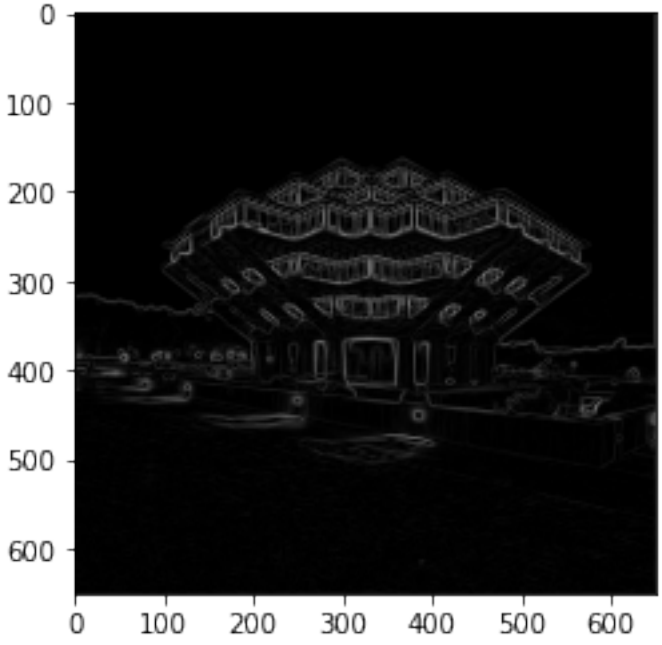
Original:



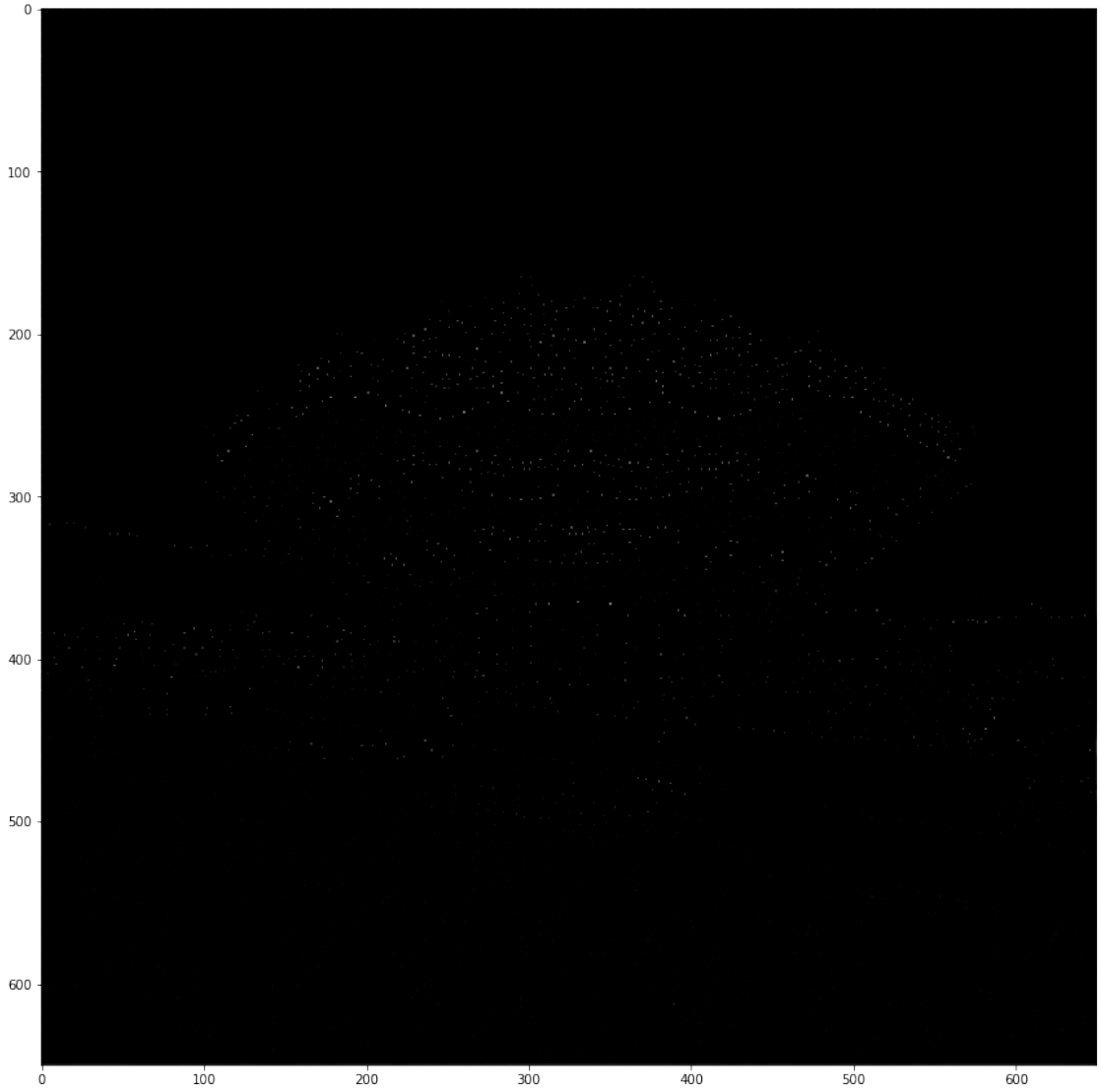
Smoothed:



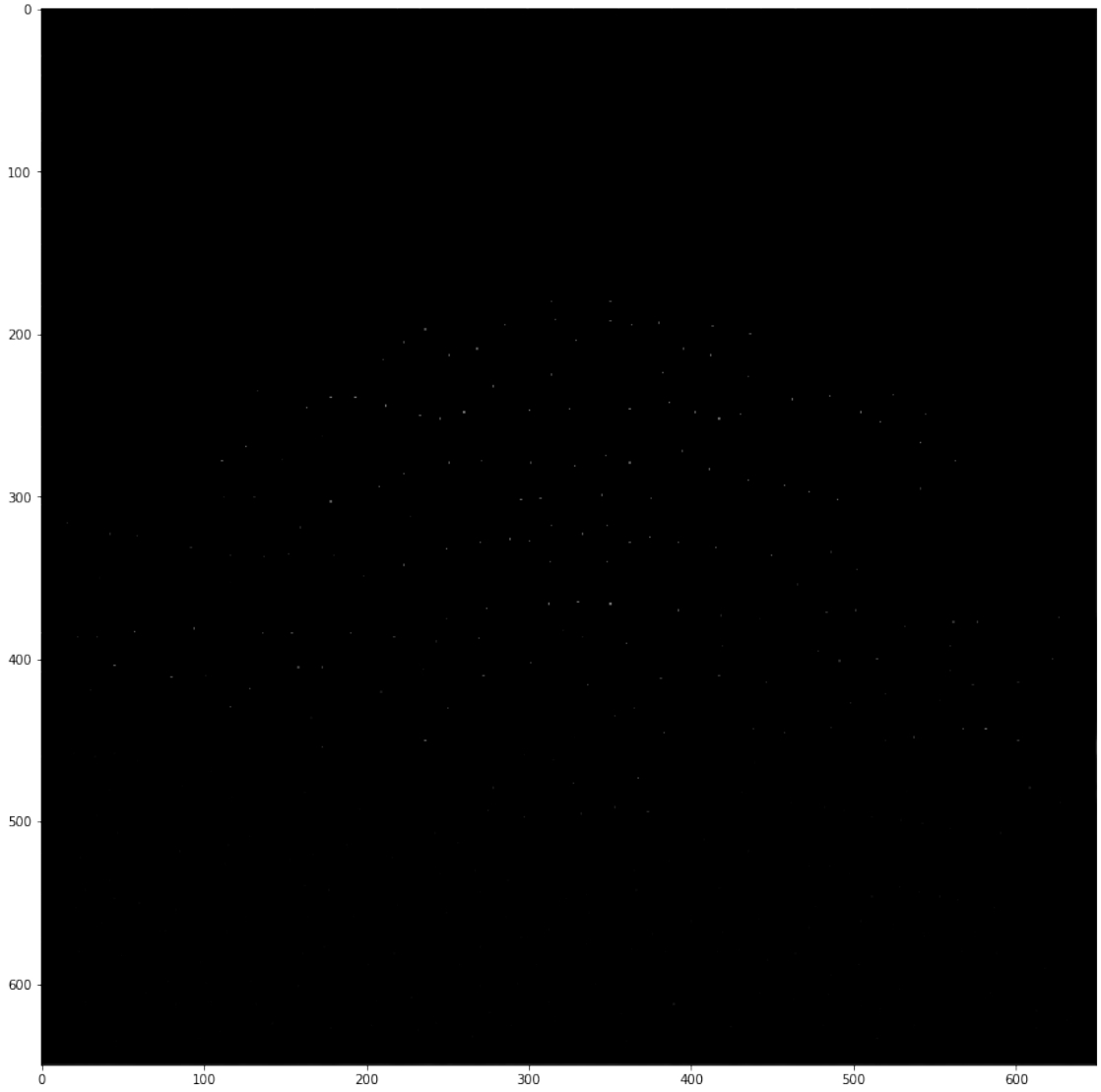
Gradient magnitude:



NMS with 5x5 window:



NMS with 21x21 window:



1.6 Submission Instructions

Remember to submit a PDF version of this notebook to Gradescope. Please make sure the contents of each cell are clearly shown in your final PDF file. **If they are not, we may dock points.**

There are multiple options for converting the notebook to PDF: 1. You can export using LaTeX (File $\rightarrow$ Download as $\rightarrow$ PDF via LaTeX). 2. You can first export as HTML and then save it as a PDF.