CSE 252A Computer Vision I Fall 2021 - Assignment 1

Instructor: Ben Ochoa

- Assignment Published On: **Wed, October 6, 2021**.
- Due On: **Wed, October 20, 2021 11:59 PM (Pacific Time)**.

Instructions

Please answer the questions below using Python in the attached Jupyter notebook and follow the guidelines below:

- This assignment must be completed **individually**. For more details, please follow the Academic Integrity Policy and Collaboration Policy on Canvas.
- All the solutions must be written in this Jupyter notebook.
- After finishing the assignment in the notebook, please export the notebook as a PDF and submit both the notebook and the PDF (i.e. the `.ipynb` and the `.pdf` files) on Gradescope.
- You may use basic algebra packages (e.g. `NumPy`, `SciPy`, etc) but you are not allowed to use the packages that directly solve the problems. Feel free to ask the instructor and the teaching assistants if you are unsure about the packages to use.
- 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.

Problem 1: Geometry (15 points)

Consider a line in the 2D plane, whose equation is given by $a\tilde{x} + b\tilde{y} + c = 0$, where $\boldsymbol{l} = (a, b, c)^\top$ and $\boldsymbol{x} = (\tilde{x}, \tilde{y}, 1)^\top$. Noticing that $\boldsymbol{x}$ is a homogeneous representation of $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y})^\top$, we can view $\boldsymbol{l}$ as a homogeneous representation of the line $a\tilde{x} + b\tilde{y} + c = 0$. We see that the line is also defined up to a scale since $(a, b, c)^\top$ and $k(a, b, c)^\top$ with $k \neq 0$ represents the same line.

1. [6 points] Prove $x^T l + l^T x = 0$, if a point x in homogeneous coordinates lies on the homogeneous line l.

2. [2 points] What is the line, in homogenous coordinates, joining the inhomogeneous points $(1, 7)$ and $(5, -8)$.

3. [2 points] Find a vector that is a homogeneous representation of the line that passes through the points (1, 5) and (–3, 2).

4. [5 points] Consider the intersection of two lines l_1 and l_2. Prove that the homogeneous point of intersection, x, of two homogeneous lines l_1 and l_2 is $x = l_1 \times l_2$, where × stands for the vector (or cross) product.

Answers Problem 1: Geometry

Question 1. We have a point $\boldsymbol{x} = (\tilde{x}, \tilde{y}, 1)^\top$ and the homogeneous line $\boldsymbol{l} = (a, b, c)^\top$. The line in the 2D plane can be written as $a\tilde{x} + b\tilde{y} + c = 0$. This equation can be found by computing $\boldsymbol{x}^T\boldsymbol{l} = (\tilde{x}, \tilde{y}, 1)(a, b, c)^\top = a\tilde{x} + b\tilde{y} + c = 0$ for the point $\boldsymbol{x} = (\tilde{x}, \tilde{y}, 1)^\top$. Similarly, we compute $\boldsymbol{l}^T\boldsymbol{x} = (a, b, c)(\tilde{x}, \tilde{y}, 1)^T = a\tilde{x} + b\tilde{y} + c = 0$.

A line $\boldsymbol{l}$ can be scaled with $k \neq 0$ to represent the same line. Thus, by using $k = -1$ on the second of the equations for a point on a line, we have $\boldsymbol{x}^T\boldsymbol{l} = \boldsymbol{l}^T\boldsymbol{x} = -\boldsymbol{l}^T\boldsymbol{x} = 0$. Note that this is just to show that a line can be scaled in any way and that k = -1 is convenient to show the proof but is strictly not necessary.

Hence, we have $\boldsymbol{x}^T\boldsymbol{l} = -\boldsymbol{l}^T\boldsymbol{x} \implies \boldsymbol{x}^T\boldsymbol{l} + \boldsymbol{l}^T\boldsymbol{x} = 0$. QED.

Question 2. We let $\boldsymbol{x_1} = (x_1, y_1)^T = (1, 7)^T$ and $\boldsymbol{x_2} = (x_2, y_2)^T = (5, -8)^T$. To find the equation joining the two points, we will represent the two inhomogeneous points as homogeneous points, and then take the cross product between these two to find the inhomogeneous line passing through the two points.

$\tilde{\boldsymbol{x}}_\mathbf{1} = (x_1, y_1, 1)^T = (1, 7, 1)^T$ and $\tilde{\boldsymbol{x}}_\mathbf{2} = (x_2, y_2, 1)^T = (5, -8, 1)^T$.

$\boldsymbol{l} = \tilde{\boldsymbol{x}}_\mathbf{1} \times \tilde{\boldsymbol{x}}_\mathbf{2} = (15, 4, -43)^T$ is the homogeneous line vector, making the line joining the two points $15\tilde{x} + 4\tilde{y} - 43 = 0$.

Question 3. We use the same approach as in the last question. We let $\boldsymbol{x_1} = (x_1, y_1)^T = (1, 5)^T$ and $\boldsymbol{x_2} = (x_2, y_2)^T = (-3, 2)^T$.

$\tilde{\boldsymbol{x}}_\mathbf{1} = (x_1, y_1, 1)^T = (1, 5, 1)^T$ and $\tilde{\boldsymbol{x}}_\mathbf{2} = (x_2, y_2, 1)^T = (-3, 2, 1)^T$.

$\boldsymbol{l} = \tilde{\boldsymbol{x}}_\mathbf{1} \times \tilde{\boldsymbol{x}}_\mathbf{2} = (3, -4, 17)^T$ are the coordinates of the homogeneous line between two points.

Question 4. As we know, a line can be represented by $a\tilde{x} + b\tilde{y} + c = 0$, where $\boldsymbol{x} = (\tilde{x}, \tilde{y}, 1)^\top$ is a point on the 2D line. Thus, as we know that for an intersection the two lines cross, the point $\boldsymbol{x}$ must be on both lines. Therefore, we know that we have $\boldsymbol{x}^T\boldsymbol{l}_1 = 0$ and $\boldsymbol{x}^T\boldsymbol{l}_2 = 0$. This means that the point $\boldsymbol{x}$ must be ortogonal to both the lines, as the dot product between the point and the line is zero. Therefore, we need to find a vector that is orthogonal to both lines at the same time. In high school, we learned that this can be found by taking the cross product between the two lines, namely $\boldsymbol{l}_1 \times \boldsymbol{l}_2$.

Thus, by using $\boldsymbol{x} = \boldsymbol{l}_1 \times \boldsymbol{l}_2$, we have found the homogeneous point of intersection $\boldsymbol{x}$ that is orthogonal to both lines at the same time. QED.

Problem 2: Image Formation and Rigid Body

Transformations (17 points)

In this problem we will practice rigid body transformations and image formations through the projective camera model. The goal will be to photograph the following four points $\widetilde{\boldsymbol{X}}_1 = [\text{-3 -6 } 4]^T$, $\widetilde{\boldsymbol{X}}_2 = [3 \text{ -6 } 4]^T$, $\widetilde{\boldsymbol{X}}_3 = [3 \text{ } 6 \text{ } 4]^T$, $\widetilde{\boldsymbol{X}}_4 = [\text{-3 } 6 \text{ } 4]^T$ in the world coordinate frame. First, recall the following formula for rigid body transformation

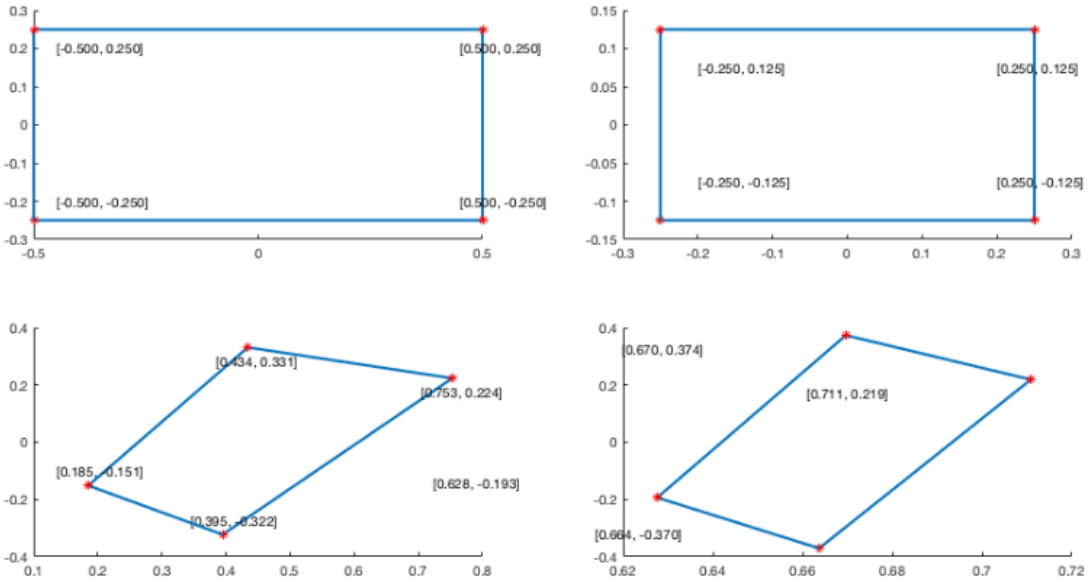
$$\widetilde{\boldsymbol{X}}_{cam} = R\widetilde{\boldsymbol{X}} + \boldsymbol{t}$$

Where $\widetilde{\boldsymbol{X}}_{cam}$ is the point coordinate in the camera coordinate system. $\widetilde{\boldsymbol{X}}$ is a point in the world coordinate frame, and R and $\boldsymbol{t}$ are the rotation and translation that transform points from the world coordinate frame to the camera coordinate frame. Together, R and $\boldsymbol{t}$ are the $extrinsic$ camera parameters. Once transformed to the camera coordinate frame, the points can be photographed using the 3×3 camera calibration matrix K, which embodies the $intrinsic$ camera parameters, and the canonical projection matrix $[\mathrm{I}|\mathbf{0}]$. Given K, R, and $\boldsymbol{t}$, the image of a point $\widetilde{\boldsymbol{X}}$ is $\boldsymbol{x} = \mathrm{K}[\mathrm{I}|\mathbf{0}]\boldsymbol{X}_{\mathrm{Cam}} = \mathrm{K}[\mathrm{R}|\boldsymbol{t}]\boldsymbol{X}$, where the homogeneous points $\boldsymbol{X}_{\mathrm{Cam}} = (\widetilde{\boldsymbol{X}}_{\mathrm{Cam}}^{\top}, 1)^{\top}$ and $\boldsymbol{X} = (\widetilde{\boldsymbol{X}}^{\top}, 1)^{\top}$. We will consider four different settings of focal length, viewing angles and camera positions below.

a). The extrinsic transformation matrix,

b). Intrinsic camera matrix under the perspective camera assumption.

c). Calculate the image of the four vertices and plot using the supplied **plot_points** function (see e.g. output in figure below).



1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
2. [Translation]. Focal length = 1. $\boldsymbol{t} = [0\ 0\ 2]^T$. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. R encodes a 60 degrees around the z-axis and then 30 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ 2]^T$.
4. [Translation and Rotation, long distance]. Focal length = 5. R encodes a 60 degrees around the z-axis and then 30 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ 7]^T$.

We will not use a full intrinsic camera matrix (e.g. that maps centimeters to pixels, and defines the coordinates of the center of the image), but only parameterize this with f, the focal length. In other words: the only parameter in the intrinsic camera matrix under the perspective assumption is f.

For all the four cases, include a image like above. Note that the axis are the same for each row, to facilitate comparison between the two camera models. Note: the angles and offsets used to generate these plots may be different from those in the problem statement, it's just to illustrate how to report your results.

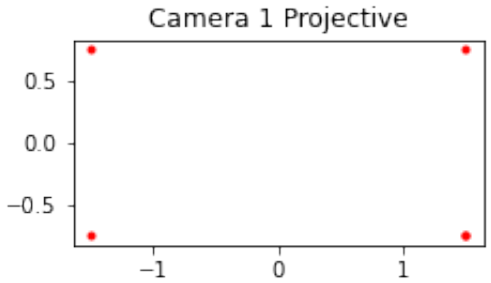
Also, Explain why you observe any distortions in the projection, if any, under this model.

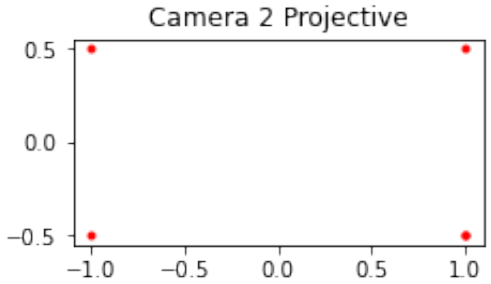
In [219…

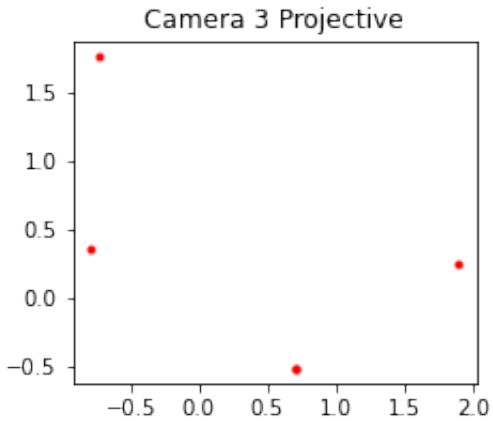
```
import numpy as np
import matplotlib.pyplot as plt
import math as m

# convert points from euclidean to homogeneous
def to_homog(points): #here always remember that points is a 3x4 matrix
    # write your code here
    homo_points = np.vstack((points, np.ones(4)))
    return homo_points

# convert points from homogeneous to euclidean
def from_homog(points_homog):
    # write your code here
    # Storing and removing the last row of the homogenious point matrix
    # to obtain inhomogeneous point matrix
    w = points_homog[-1]
    euclid_points = np.delete(points_homog, -1, 0)
    euclid_points = euclid_points / w
    return euclid_points

# project 3D euclidean points to 2D euclidean
def project_points(P_int, P_ext, pts):
    # make 3D euclidean points homogeneous
    homog_points = to_homog(pts)

    # make the 3D homo points 2D homo points
    homog_2d = P_int @ P_ext @ homog_points

    # go back to 2D euclidean points
```

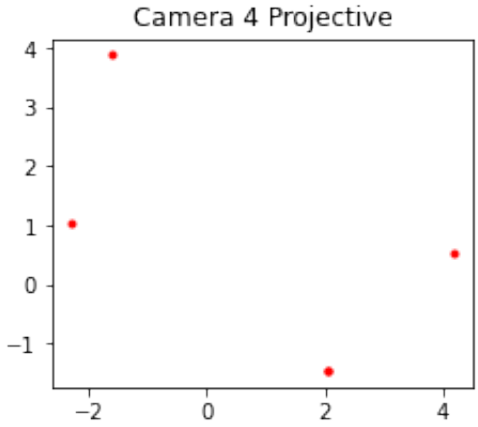
```
    pts_2d = from_homog(homog_2d)
    #print(pts_2d)
    #return the 2d euclidean points
    return pts_2d

#w = homog_2d[-1]
    #w_matrix = np.vstack((w,w))
    #homog_2d = np.delete(homog_2d,-1,0)
    #pts_2d = homog_2d / w



# Change the three matrices for the four cases as described in the problem
# in the four camera functions given below. Make sure that we can see the i
# (if one exists) being used to fill in the matrices. Feel free to document
# comments any thing you feel the need to explain.

# given the focal length, compute the intrinsic camera matrix

def intrinsic_cam_mat(f):
    #"""
    #K = [f 0 0
    #     0 f 0
    #     0 0 1]
    #"""
    # write your code here
    int_cam_mat = np.diag([f,f,1.0])
    return int_cam_mat

# Compute the extrinsic camera matrix
def extrinsic_cam_mat(angles, t):
    #"""
    #    ext_cam_mat = [R|t]
    #"""
    #
    x_rot = angles[0]
    y_rot = angles[1]
    z_rot = angles[2]
    r_x = np.matrix([[1, 0, 0],[0, m.cos(x_rot), -m.sin(x_rot)], [0, m.sin(
    r_y = np.matrix([[m.cos(y_rot), 0, m.sin(y_rot)],[0, 1, 0], [-m.sin(y_r
    r_z = np.matrix([[m.cos(z_rot), -m.sin(z_rot), 0],[m.sin(z_rot), m.cos(
    R = r_z @ r_y @ r_x
    ext_cam_mat = np.concatenate((R,t[:,None]),axis=1)

    # write your code here
    return ext_cam_mat

def camera1():
    P_int_proj = intrinsic_cam_mat(1.0)
    P_ext = extrinsic_cam_mat(np.array([0,0,np.pi/2]), np.array([0,0,0]))
    return P_int_proj, P_ext

def camera2():
    P_int_proj = intrinsic_cam_mat(1.0)
    P_ext = extrinsic_cam_mat(np.array([0,0,np.pi/2]), np.array([0,0,2]))
    return P_int_proj, P_ext
```

```
def camera3():
    P_int_proj = intrinsic_cam_mat(1.0)
    P_ext = extrinsic_cam_mat(np.array([0,np.pi/6,np.pi/3]), np.array([0,0,
    return P_int_proj, P_ext

def camera4():
    P_int_proj = intrinsic_cam_mat(5.0)
    P_ext = extrinsic_cam_mat(np.array([0,np.pi/6,np.pi/3]), np.array([0,0,
    return P_int_proj, P_ext


##########################################################
# test code. Do not modify
##########################################################

def plot_points(points, title='', style='.-r', axis=[]):
    inds = list(range(points.shape[1]))+[0]
    plt.plot(points[0,inds], points[1,inds],style)
    if title:
        plt.title(title)
    if axis:
        plt.axis('scaled')
        #plt.axis(axis)

def main():
    point1 = np.array([[-3,-6,4]]).T
    point2 = np.array([[3,-6,4]]).T
    point3 = np.array([[3,6,4]]).T
    point4 = np.array([[-3,6,4]]).T
    points = np.hstack((point1,point2,point3,point4))

    for i, camera in enumerate([camera1, camera2, camera3, camera4]):
        P_int_proj, P_ext = camera()
        plt.subplot(1, 2, 1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camer
        plt.show()

main()
```


Camera 1 Projective
0.5
0.0
−0.5
−1
0
1


Camera 2 Projective
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0


Camera 3 Projective
1.5
1.0
0.5
0.0
−0.5
−0.5
0.0
0.5
1.0
1.5
2.0


Camera 4 Projective
4
3
2
1
0
−1
−2
0
2
4

Problem 3: Surface Rendering (18 pts)

In this portion of the assignment we will be exploring different methods of approximating local reflectance of objects in a scene. This last section of the homework will be an exercise in rendering surfaces. Here, you need use the surface normals and the masks from the provided pickle files, with various light sources, different materials, and using a number of reflectance models. For the sake of simplicity, multiple reflections of light rays, and occlusion of light rays due to object/scene can be ignored.

Data

The surface normals and masks are to be loaded from the respective pickle files. For comparison, You should display the rendering results for both normals calculated from the original image and the diffuse components. There are 2 images that we will be playing with namely one of a sphere and the other of a pear.

Assume that the albedo map is uniform.

Lambertian Reflectance

One of the simplest models available to render 3D objections with reflectance is the Lambertian model. This model finds the apparent brightness to an observer using the direction of the light source $\mathbf{L}$ and the normal vector on the surface of the object $\mathbf{N}$. The brightness intensity at a given point on an object's surface, $\mathbf{I_d}$, with a single light source is found using the following relationship:

$$\mathbf{I_d} = \mathbf{L} \cdot \mathbf{N}(I_l \mathbf{C})$$

where, $\mathbf{C}$ and I_l are the the color and intensity of the light source respectively.

Phong Reflectance

One major drawback of Lambertian reflectance is that it only considers the diffuse light in its calculation of brightness intensity. One other major component to reflectance rendering is the specular component. The specular reflectance is the component of light that is reflected in a single direction, as opposed to all directions, which is the case in diffuse reflectance. One of the most used models to compute surface brightness with specular components is the Phong reflectance model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Phong shading also considers the material in the scene which is characterized by four values: the ambient reflection constant (k_a), the diffuse reflection constant (k_d), the specular reflection constant (k_s) and α the Phong constant, which is the 'shininess' of an object. Furthermore, since the specular component produces 'rays', only some of

which would be observed by a single observer, the observer's viewing direction ($\mathbf{V}$) must also be known. For some scene with known material parameters with M light sources the light intensity $\mathbf{I}_{phong}$ on a surface with normal vector $\mathbf{N}$ seen from viewing direction $\mathbf{V}$ can be computed by:

$$\mathbf{I}_{phong} = k_a\mathbf{I}_a + \sum_{m\in M}\left\{k_d(\mathbf{L}_m\cdot\mathbf{N})\mathbf{I}_{m,d} + k_s(\mathbf{R}_m\cdot\mathbf{V})^\alpha\mathbf{I}_{m,s}\right\},$$

$$\mathbf{R}_m = 2\mathbf{N}(\mathbf{L}_m\cdot\mathbf{N}) - \mathbf{L}_m,$$

where $\mathbf{I}_a$, is the color and intensity of the ambient lighting, $\mathbf{I}_{m,d}$ and $\mathbf{I}_{m,s}$ are the color values for the diffuse and specular light of the mth light source.

Rendering

Please complete the following:

1. Write the function `lambertian()` that calculates the Lambertian light intensity given the light direction $\mathbf{L}$ with color and intensity $\mathbf{C}$ and $I_l = 1$, and normal vector $\mathbf{N}$. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the two lighting sources found in Table 1. *Note: You do not need to worry about material coefficients in this model.*

2. Write the function `phong()` that calculates the Phong light intensity given the material constants (k_a, k_d, k_s, α), $\mathbf{V} = (0, 0, 1)^\top$, $\mathbf{N}$ and some number of M light sources. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the sets of coefficients found in Table 2 with each light source individually, and both light sources combined.

Hint: To avoid artifacts due to shadows, ensure that any negative intensities found are set to zero.

Table 1: Light Sources

m	Location	Color (RGB)
1	$(-\frac{1}{2}, \frac{1}{2}, \frac{1}{2})^\top$	$(1, 1, 1)$
2	$(1, 0, 0)^\top$	$(1, .5, 1)$

Table 2: Material Coefficients

Mat.	k_a	k_d	k_s	α
1	0	0.1	0.5	5
2	0	0.5	0.1	5
3	0	0.5	0.5	10

Part 1. Loading pickle files and plotting the normals [4 pts] (Sphere - 2pts, Pear - 2pts)

In this first part, you are required to work with 2 images, one of a sphere and the other one of a pear. The pickle file normals.pickle is a list consisting of 4 numpy matrices which are

1) Normal Vectors for the sphere with specularities removed (Diffuse component)
2) Normal Vector for the sphere
3) Normal Vectors for the pear with specularities removed (Diffuse component)
4) Normal vectors for the pear

Please load the normals and plot them using the function plot_normals which is provided.

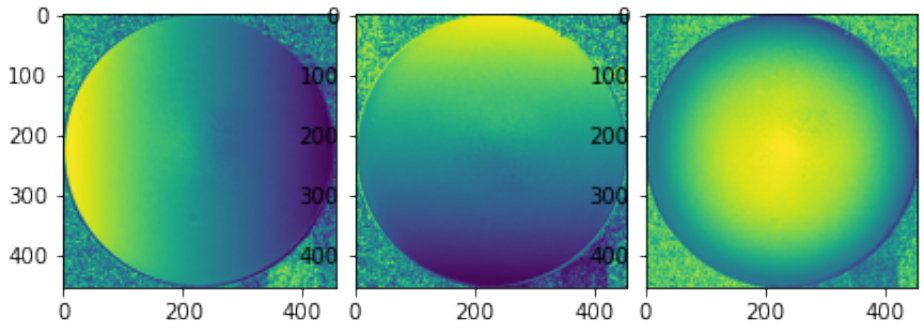
In [220…

```
def plot_normals(diffuse_normals, original_normals):
    # Stride in the plot, you may want to adjust it to different images
    stride = 5

    normalss = diffuse_normals
    normalss1 = original_normals

    print("Normals:")
    print("Diffuse")
    # showing normals as three separate channels
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normalss[..., 0])
    ax2 = figure.add_subplot(132)
    ax2.imshow(normalss[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normalss[..., 2])
    plt.show()
    print("Original")
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normalss1[..., 0])
    ax2 = figure.add_subplot(132)
    ax2.imshow(normalss1[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normalss1[..., 2])
    plt.show()
```

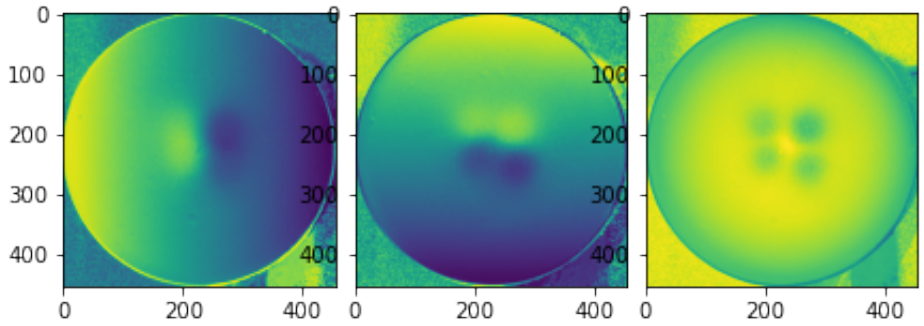
In [221…

```
#Plot the normals for the sphere and pear for both the normal and diffuse c
#1 : Load the different normals
# LOAD HERE
import pickle

with open('normals.pkl', 'rb') as f:
    normals = pickle.load(f)

sphere_diffuse = normals[0]
sphere_original = normals[1]
pear_diffuse = normals[2]
pear_original = normals[3]
#2 : Plot the normals using plot_normals
#What do you observe? What are the differences between the diffuse componer

#PLOT HERE
#Plot sphere
plot_normals(sphere_diffuse, sphere_original)

#Plot pear
plot_normals(pear_diffuse, pear_original)


print('We observe that the surfaces are not as clear and do not show the ir
```

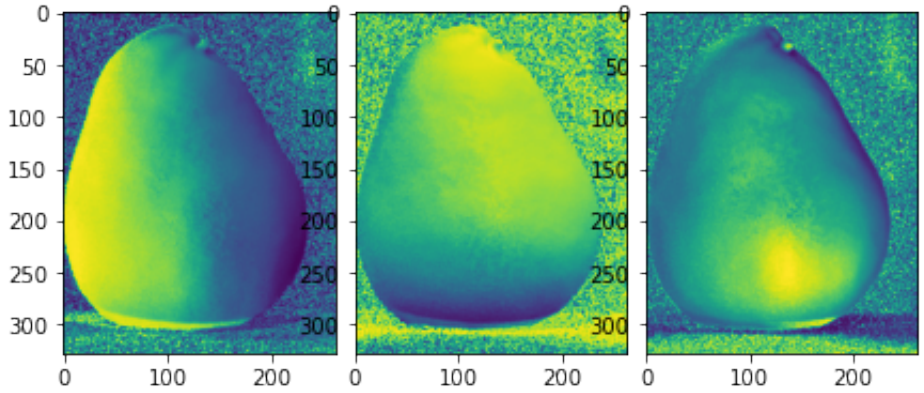
Normals:
Diffuse



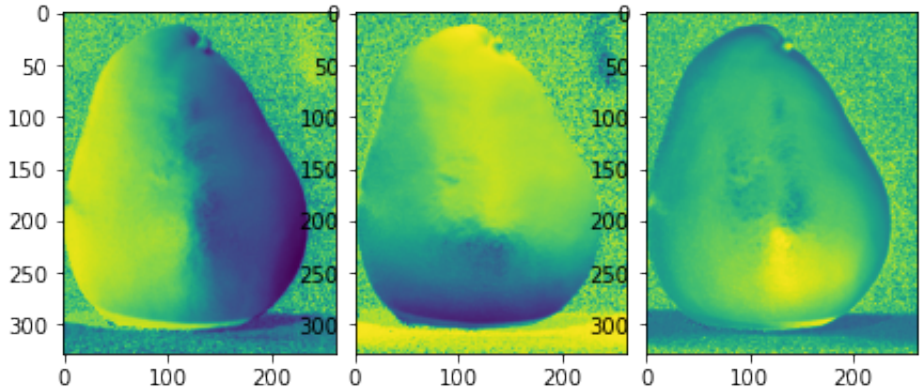
Original



Normals:
Diffuse



Original



We observe that the surfaces are not as clear and do not show the irregularities in the diffuse photo, as opposed to in the original one. The edges are also more distinct and more shadowy, as the object darkens when the normals are further from the light.

Part 2. Lambertian model [6 pts]

Fill in your implementation for the rendered image using the lambertian model.

In [222…

```
def normalize(img):
    #assert img.shape[2] == 3
    maxi = img.max()
    mini = img.min()
    return (img - mini)/(maxi-mini)
```

In [223…

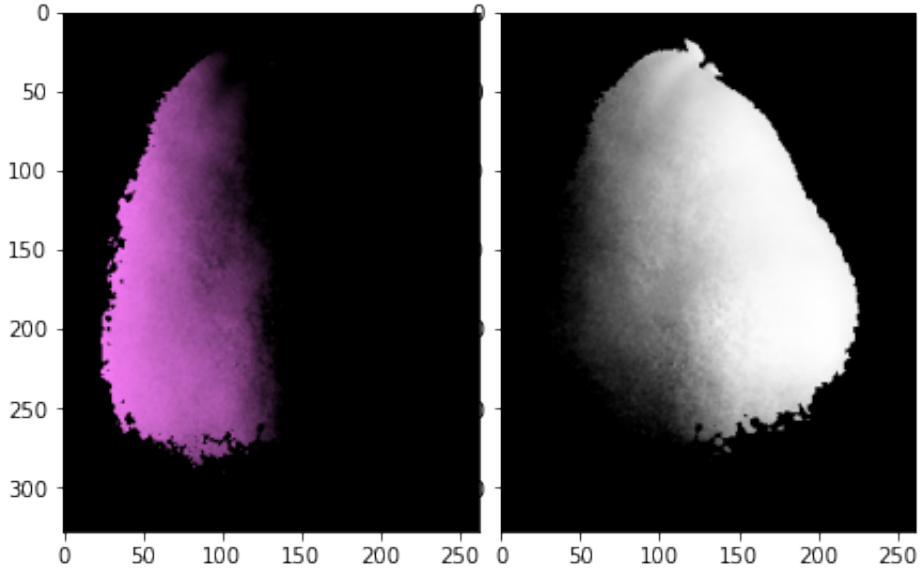
```
def lambertian(normals, lights, color, intensity, mask):
    # Lambertian function
    image = np.dot(normals, lights)*(color*intensity)

    # Ensure negative values are set to zero
    image = np.where(image<0, 0, image)

    # Multiply each layer of the image with the mask
    image[:,:,0] = mask*image[:,:,0]
    image[:,:,1] = mask*image[:,:,1]
    image[:,:,2] = mask*image[:,:,2]
    return (image)
```

Plot the rendered results for both the sphere and the pear for both the original and the diffuse components. Remember to first load the masks from the masks.pkl file. The masks.pkl file is a list consisting of 2 numpy arrays-
1)Mask for the sphere
2)Mask for the pear
Remember to plot the normalized image using the function normalize which is provided.

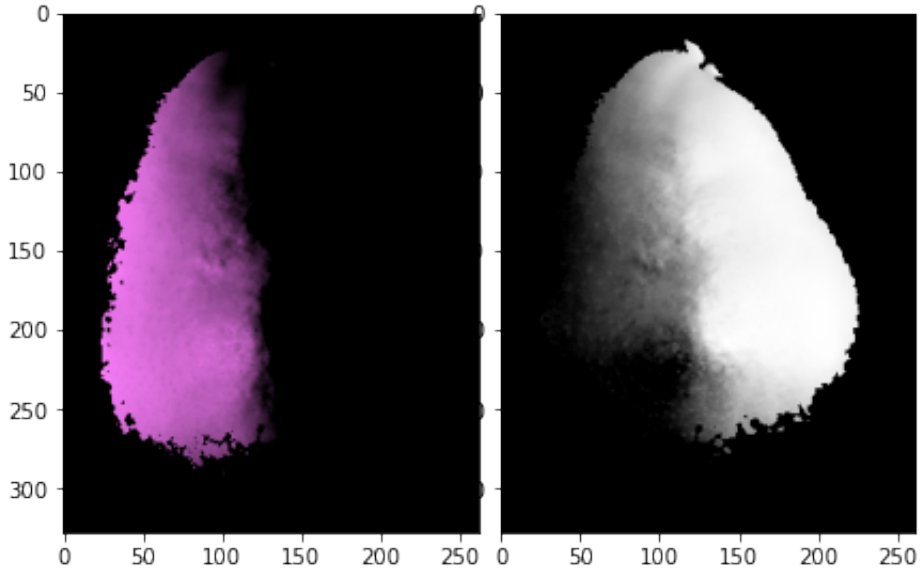
In [224…

```
# Load the masks for the sphere and pear
# LOAD HERE
with open('masks.pkl', 'rb') as f:
    masks = pickle.load(f)

sphere_mask = masks[0]
pear_mask = masks[1]

# Output the rendering results for Pear
dirn1 = np.array([[1.0],[0],[0]])
color1 = np.array([[1,.5,1]])
dirn2 = np.array([[-1.0/2],[1.0/2],[1.0/2]])
color2 = np.array([[1,1,1]])

#Display the rendering results for pear for both diffuse and for both the
print("Diffuse pear")
figure = plt.figure()
ax1 = figure.add_subplot(121)
ax1.imshow(normalize(lambertian(pear_diffuse,dirn1,color1,1,pear_mask)))
ax2 = figure.add_subplot(122)
ax2.imshow(normalize(lambertian(pear_diffuse,dirn2,color2,1,pear_mask)))
plt.show()

print("Original pear")
figure = plt.figure()
ax1 = figure.add_subplot(121)
ax1.imshow(normalize(lambertian(pear_original,dirn1,color1,1,pear_mask)))
ax2 = figure.add_subplot(122)
ax2.imshow(normalize(lambertian(pear_original,dirn2,color2,1,pear_mask)))
plt.show()
```

Diffuse pear



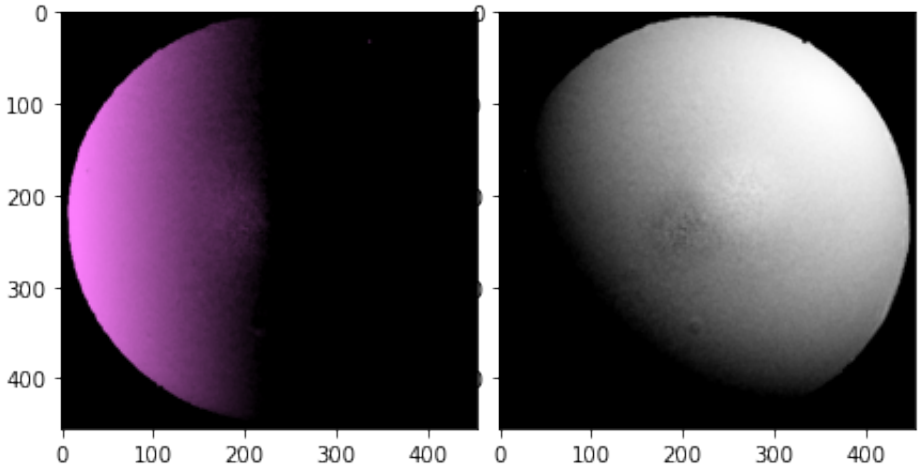
Original pear



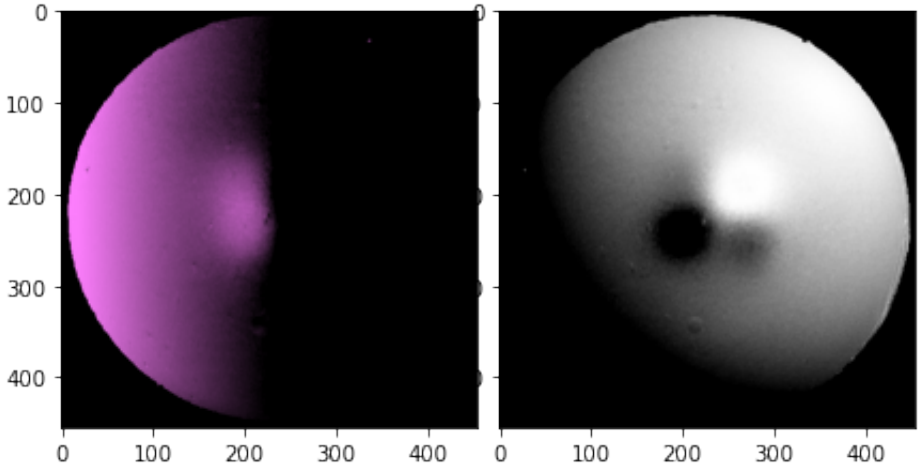
In [225…

```
# Output the rendering results for Sphere
dirn1 = np.array([[1.0],[0],[0]])
color1 = np.array([[1,.5,1]])
dirn2 = np.array([[-1.0/2],[1.0/2],[1.0/2]])
color2 = np.array([[1,1,1]])
#Display the rendering results for sphere for both diffuse and for both the
print("Diffuse sphere")
figure = plt.figure()
ax1 = figure.add_subplot(121)
ax1.imshow(normalize(lambertian(sphere_diffuse,dirn1,color1,1,sphere_mask))
ax2 = figure.add_subplot(122)
ax2.imshow(normalize(lambertian(sphere_diffuse,dirn2,color2,1,sphere_mask))
plt.show()

print("Original sphere")
figure = plt.figure()
ax1 = figure.add_subplot(121)
ax1.imshow(normalize(lambertian(sphere_original,dirn1,color1,1,sphere_mask)
ax2 = figure.add_subplot(122)
ax2.imshow(normalize(lambertian(sphere_original,dirn2,color2,1,sphere_mask)
plt.show()
```

```
Diffuse sphere
```




```
Original sphere
```



Part 3. Phong model [8 pts]

Please fill in your implementation for the Phong model below.

In [226…

```
def phong(normals, lights, color, material, view, mask):
    kd = material[0]
    ks = material[1]
    a = material[2]
    image = np.zeros((normals.shape[0],normals.shape[1],3))
    for i, light in enumerate(lights):
        R = 2*normals*(light*normals)-light
        image_i = kd*(normals*light)*color[i] + ks*np.power((np.dot(R,view)
        image_i = np.where(image_i<0, 0, image_i)
        image += image_i

    image[:,:,0] = mask*image[:,:,0]
    image[:,:,1] = mask*image[:,:,1]
    image[:,:,2] = mask*image[:,:,2]
    return (image)
```

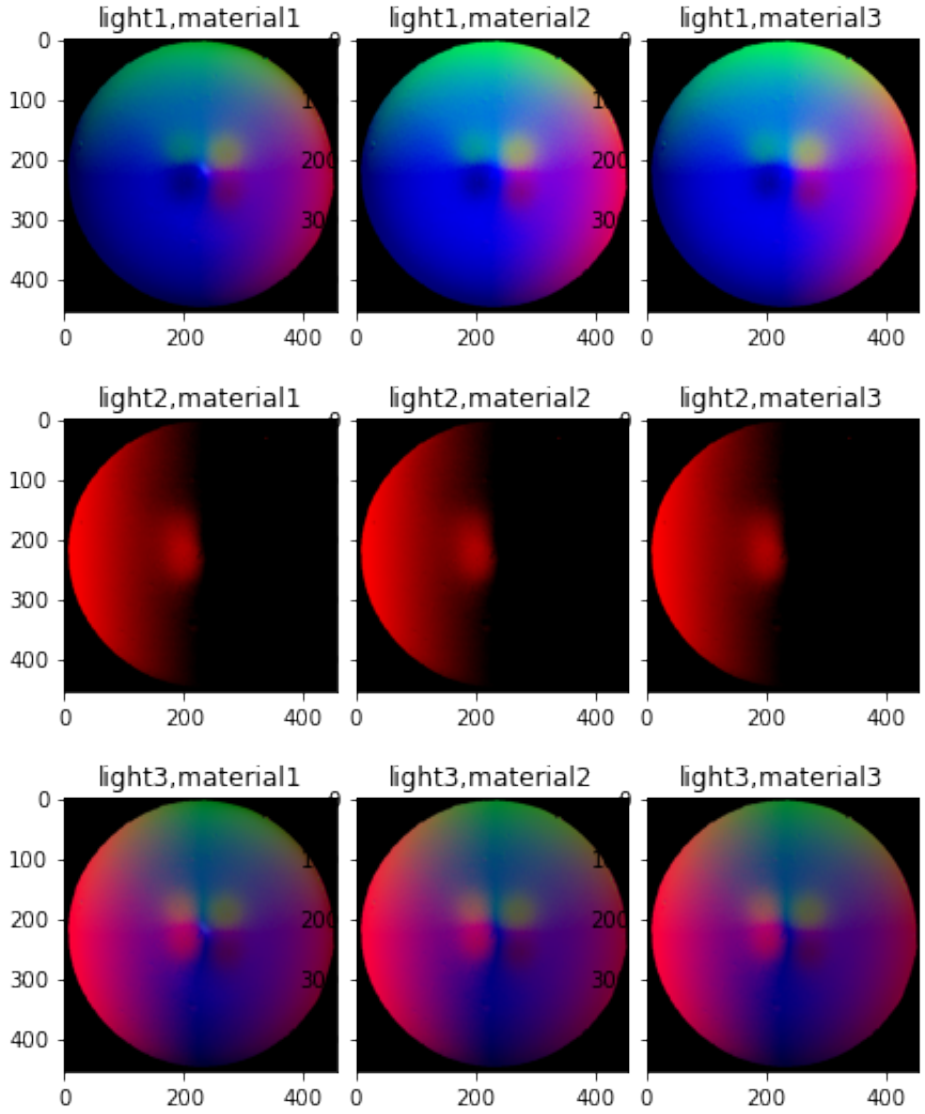
With the function completed, plot the rendering results for the sphere and pear (both diffuse and original compnents) for all the materials and light sources and also with the combination of both the light sources.

In [227…

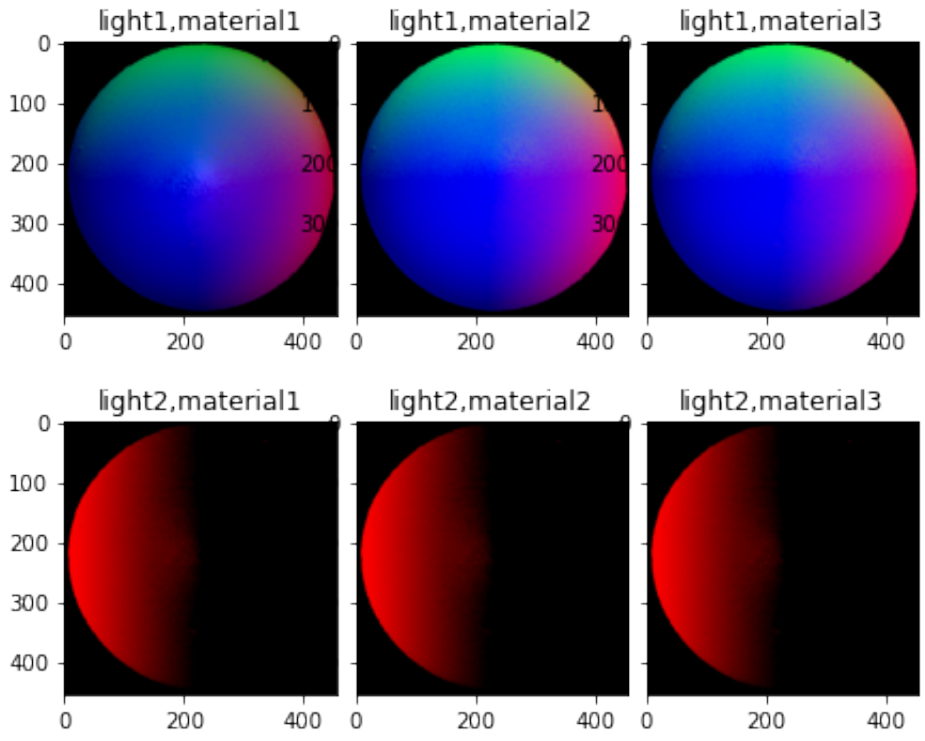
```
# Output the rendering results for sphere
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.5,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 =  np.array([[-1.0/2,1],[1.0/2,1],[1.0/2,1]])
lightcol2 = np.array([[1,1],[0,0.5],[0,1]])
#Display rendered results for sphere for all materials and light sources ar
plt.rcParams['figure.constrained_layout.use'] = True
print("Original sphere")
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.title.set_text('light1,material1')
ax1.imshow(normalize(phong(sphere_original, [lightcol1[:,0]], [lightcol1[:,
ax2 = figure.add_subplot(132)
ax2.title.set_text('light1,material2')
ax2.imshow(normalize(phong(sphere_original, [lightcol1[:,0]], [lightcol1[:,
ax3 = figure.add_subplot(133)
ax3.title.set_text('light1,material3')
ax3.imshow(normalize(phong(sphere_original, [lightcol1[:,0]], [lightcol1[:,
plt.show()
figure = plt.figure()
ax4 = figure.add_subplot(131)
ax4.title.set_text('light2,material1')
ax4.imshow(normalize(phong(sphere_original, [lightcol2[:,0]], [lightcol2[:,
ax5 = figure.add_subplot(132)
ax5.title.set_text('light2,material2')
ax5.imshow(normalize(phong(sphere_original, [lightcol2[:,0]], [lightcol2[:,
ax6 = figure.add_subplot(133)
ax6.title.set_text('light2,material3')
ax6.imshow(normalize(phong(sphere_original, [lightcol2[:,0]], [lightcol2[:,
plt.show()
figure = plt.figure()
ax7 = figure.add_subplot(131)
```

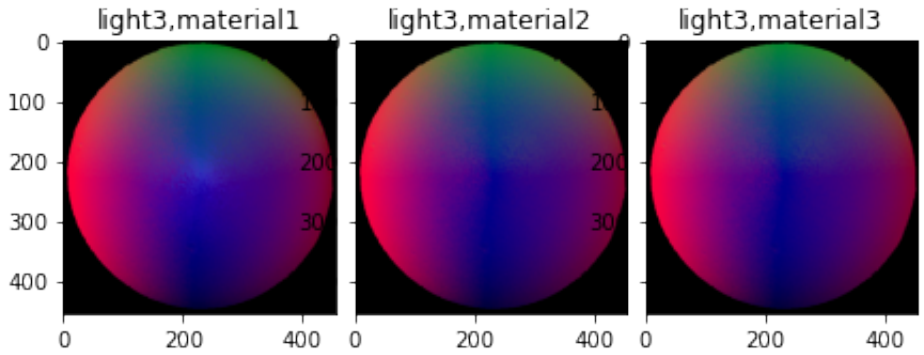
```
ax7.title.set_text('light3,material1')
ax7.imshow(normalize(phong(sphere_original, [lightcol1[:,0],lightcol2[:,0]]
ax8 = figure.add_subplot(132)
ax8.title.set_text('light3,material2')
ax8.imshow(normalize(phong(sphere_original, [lightcol1[:,0],lightcol2[:,0]]
ax9 = figure.add_subplot(133)
ax9.title.set_text('light3,material3')
ax9.imshow(normalize(phong(sphere_original, [lightcol1[:,0],lightcol2[:,0]]
plt.show()

print("Diffuse sphere")
figure = plt.figure()
plt.rcParams['figure.constrained_layout.use'] = True
ax1 = figure.add_subplot(131)
ax1.title.set_text('light1,material1')
ax1.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0]], [lightcol1[:,1
ax2 = figure.add_subplot(132)
ax2.title.set_text('light1,material2')
ax2.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0]], [lightcol1[:,1
ax3 = figure.add_subplot(133)
ax3.title.set_text('light1,material3')
ax3.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0]], [lightcol1[:,1
plt.show()
figure = plt.figure()
ax4 = figure.add_subplot(131)
ax4.title.set_text('light2,material1')
ax4.imshow(normalize(phong(sphere_diffuse, [lightcol2[:,0]], [lightcol2[:,1
ax5 = figure.add_subplot(132)
ax5.title.set_text('light2,material2')
ax5.imshow(normalize(phong(sphere_diffuse, [lightcol2[:,0]], [lightcol2[:,1
ax6 = figure.add_subplot(133)
ax6.title.set_text('light2,material3')
ax6.imshow(normalize(phong(sphere_diffuse, [lightcol2[:,0]], [lightcol2[:,1
plt.show()
figure = plt.figure()
ax7 = figure.add_subplot(131)
ax7.title.set_text('light3,material1')
ax7.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0],lightcol2[:,0]],
ax8 = figure.add_subplot(132)
ax8.title.set_text('light3,material2')
ax8.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0],lightcol2[:,0]],
ax9 = figure.add_subplot(133)
ax9.title.set_text('light3,material3')
ax9.imshow(normalize(phong(sphere_diffuse, [lightcol1[:,0],lightcol2[:,0]],
plt.show()
```

```
Original sphere
```



Diffuse sphere



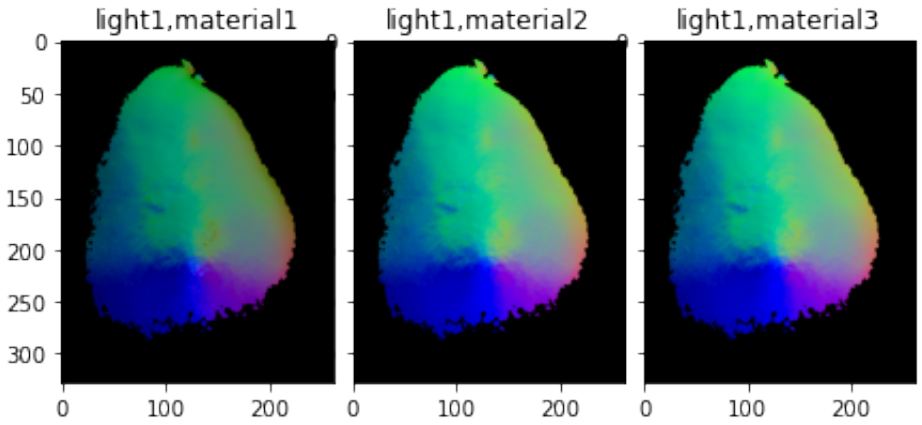


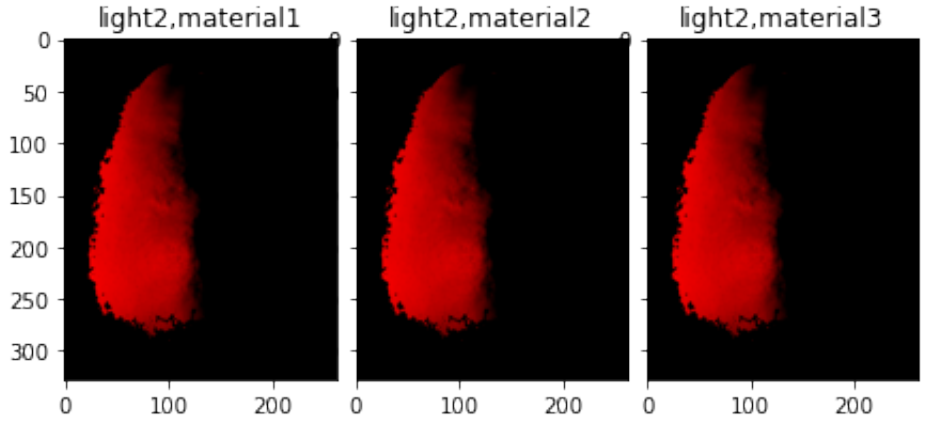
In [228…

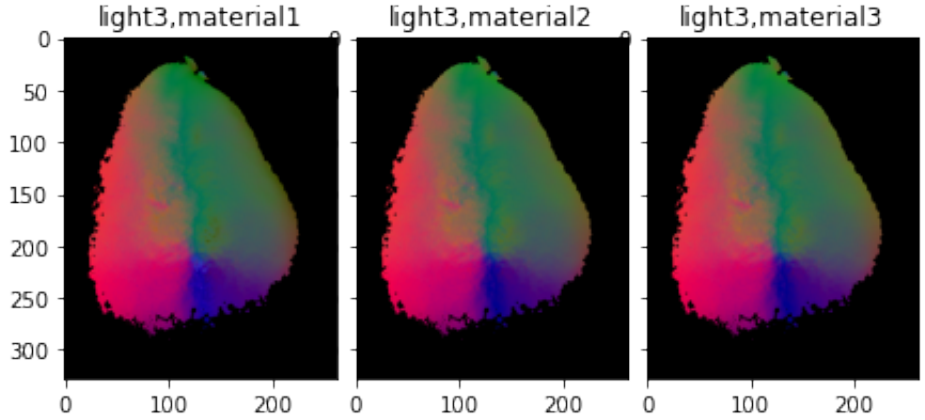
```
# Output the rendering results for the pear.
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.5,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 =  np.array([[-1.0/2,1],[1.0/2,1],[1.0/2,1]])
lightcol2 = np.array([[1,1],[0,0.5],[0,1]])
#Display rendered results for pear for all materials and light sources and
plt.rcParams['figure.constrained_layout.use'] = True
print("Original pear")
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.title.set_text('light1,material1')
ax1.imshow(normalize(phong(pear_original, [lightcol1[:,0]], [lightcol1[:,1
ax2 = figure.add_subplot(132)
ax2.title.set_text('light1,material2')
ax2.imshow(normalize(phong(pear_original, [lightcol1[:,0]], [lightcol1[:,1
ax3 = figure.add_subplot(133)
ax3.title.set_text('light1,material3')
ax3.imshow(normalize(phong(pear_original, [lightcol1[:,0]], [lightcol1[:,1
plt.show()
figure = plt.figure()
ax4 = figure.add_subplot(131)
ax4.title.set_text('light2,material1')
ax4.imshow(normalize(phong(pear_original, [lightcol2[:,0]], [lightcol2[:,1
ax5 = figure.add_subplot(132)
ax5.title.set_text('light2,material2')
ax5.imshow(normalize(phong(pear_original, [lightcol2[:,0]], [lightcol2[:,1
ax6 = figure.add_subplot(133)
ax6.title.set_text('light2,material3')
ax6.imshow(normalize(phong(pear_original, [lightcol2[:,0]], [lightcol2[:,1
plt.show()
figure = plt.figure()
ax7 = figure.add_subplot(131)
ax7.title.set_text('light3,material1')
ax7.imshow(normalize(phong(pear_original, [lightcol1[:,0],lightcol2[:,0]],
ax8 = figure.add_subplot(132)
ax8.title.set_text('light3,material2')
ax8.imshow(normalize(phong(pear_original, [lightcol1[:,0],lightcol2[:,0]],
ax9 = figure.add_subplot(133)
ax9.title.set_text('light3,material3')
ax9.imshow(normalize(phong(pear_original, [lightcol1[:,0],lightcol2[:,0]],
plt.show()

plt.rcParams['figure.constrained_layout.use'] = True
print("Diffuse pear")
```

```
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.title.set_text('light1,material1')
ax1.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0]], [lightcol1[:,1]]
ax2 = figure.add_subplot(132)
ax2.title.set_text('light1,material2')
ax2.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0]], [lightcol1[:,1]]
ax3 = figure.add_subplot(133)
ax3.title.set_text('light1,material3')
ax3.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0]], [lightcol1[:,1]]
plt.show()
figure = plt.figure()
ax4 = figure.add_subplot(131)
ax4.title.set_text('light2,material1')
ax4.imshow(normalize(phong(pear_diffuse, [lightcol2[:,0]], [lightcol2[:,1]]
ax5 = figure.add_subplot(132)
ax5.title.set_text('light2,material2')
ax5.imshow(normalize(phong(pear_diffuse, [lightcol2[:,0]], [lightcol2[:,1]]
ax6 = figure.add_subplot(133)
ax6.title.set_text('light2,material3')
ax6.imshow(normalize(phong(pear_diffuse, [lightcol2[:,0]], [lightcol2[:,1]]
plt.show()
figure = plt.figure()
ax7 = figure.add_subplot(131)
ax7.title.set_text('light3,material1')
ax7.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0],lightcol2[:,0]], [
ax8 = figure.add_subplot(132)
ax8.title.set_text('light3,material2')
ax8.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0],lightcol2[:,0]], [
ax9 = figure.add_subplot(133)
ax9.title.set_text('light3,material3')
ax9.imshow(normalize(phong(pear_diffuse, [lightcol1[:,0],lightcol2[:,0]], [
plt.show()
```

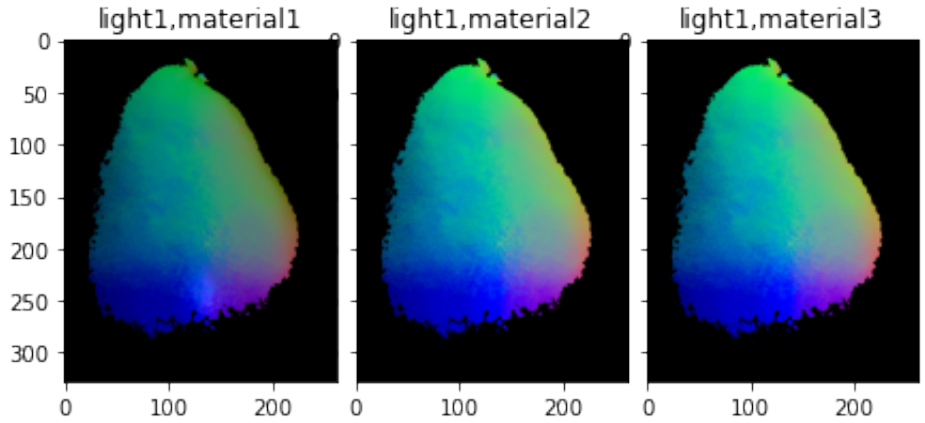
```
Original pear
```

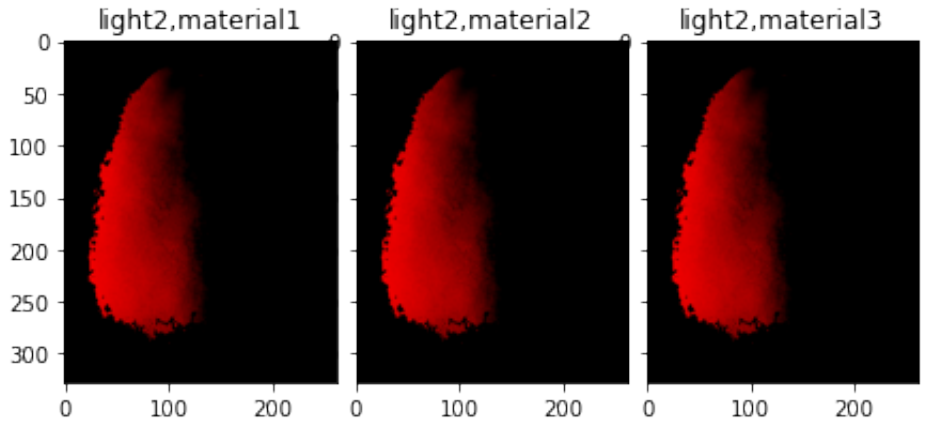


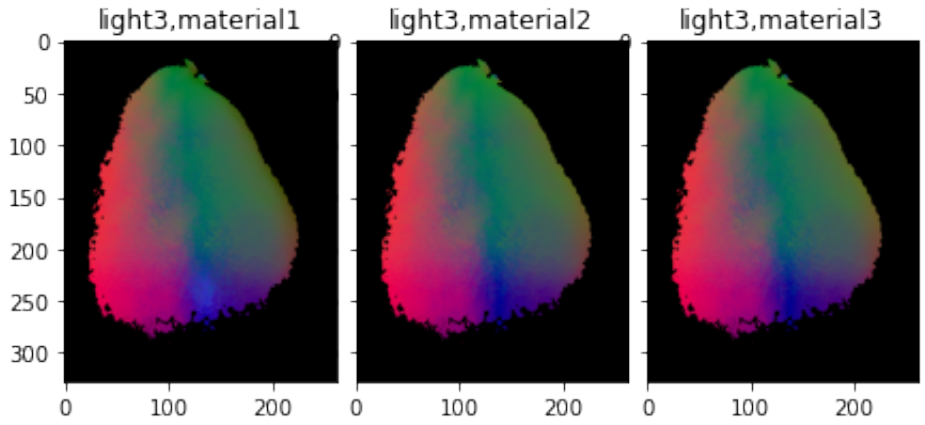




Diffuse pear







Problem 4: 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.

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.

Part 1: Lambertian Photometric Stereo [8 pts]

Implement the photometric stereo technique described in the lecture. 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 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.

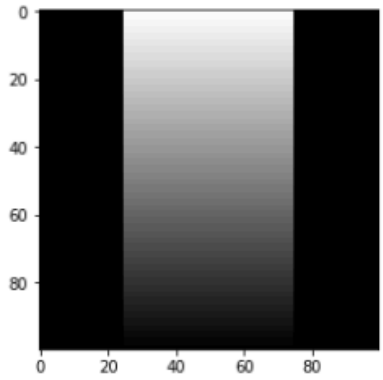
Then use all four images (most accurate).

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

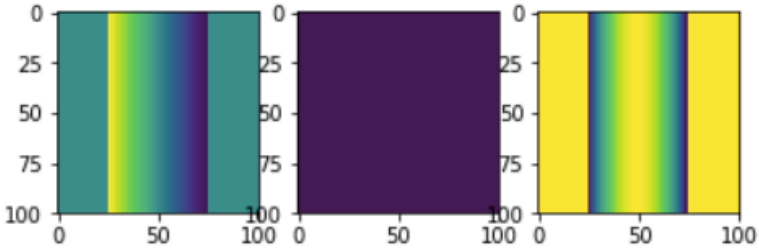
1. The estimated albedo map.
2. The estimated surface normals by showing both
    A. Needle map, and
    B. Three images showing each of the surface normal components.
3. A wireframe of the depth map given by the scanline method.
4. A wireframe of the depth map given by Horn integration.

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

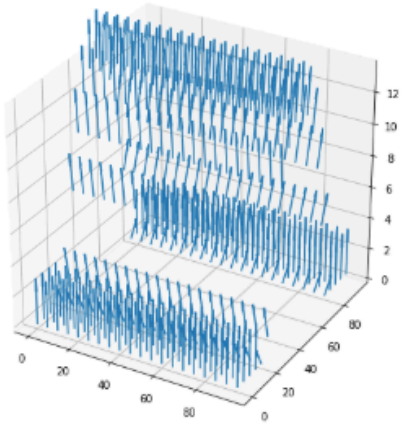
An example of outputs is shown in the figure below. (The example outputs only include one depth map, although we expect two – see above.)



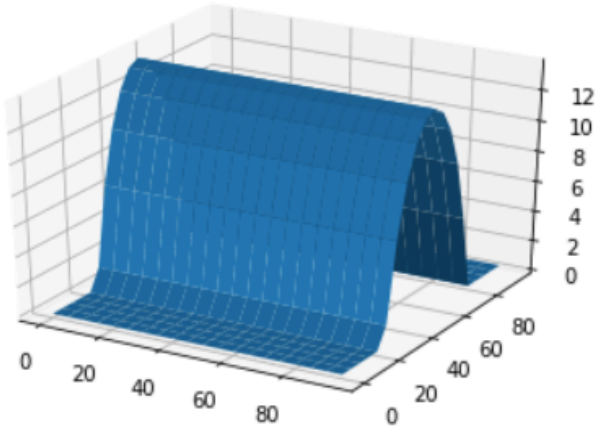
Albedo map



Normals as three separate channels



Needle map



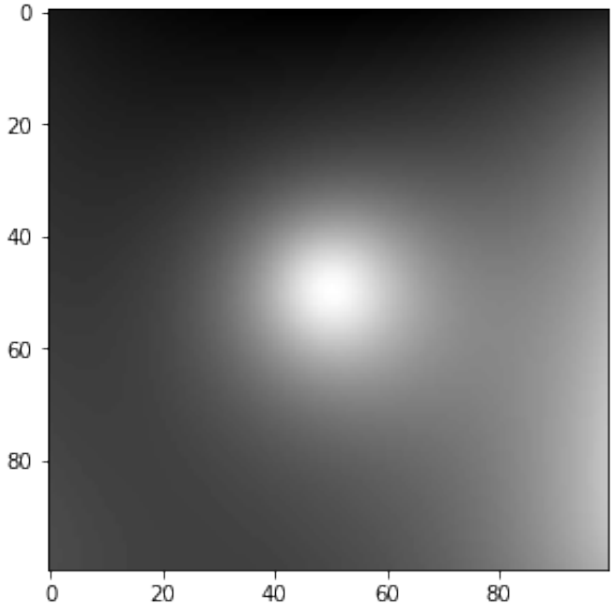
Wireframe of depth map

In [229…

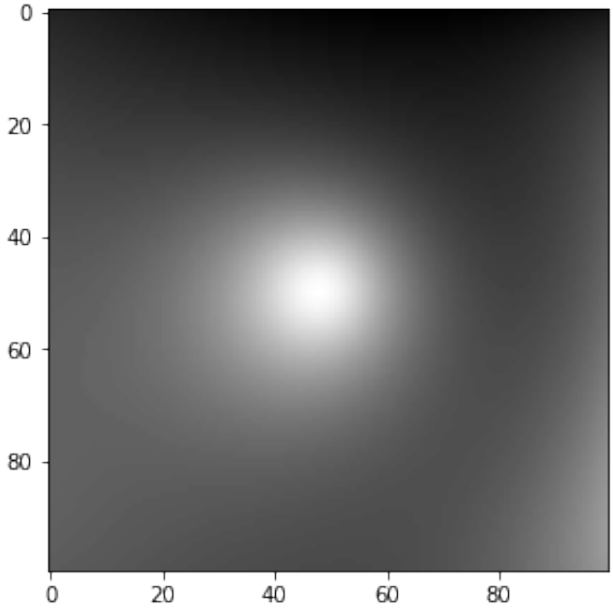
```
# Setup
import pickle
import numpy as np
from time import time
from skimage import io
%matplotlib inline
import matplotlib.pyplot as plt

### 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']
```

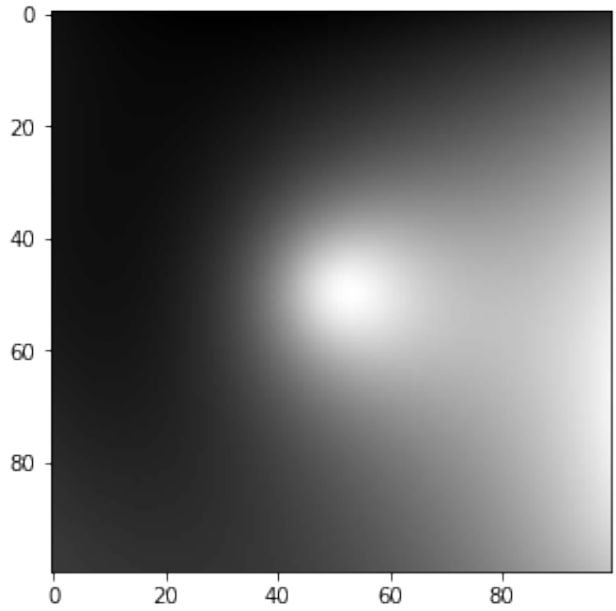
```
Image 1:
```




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

Image 2:
```

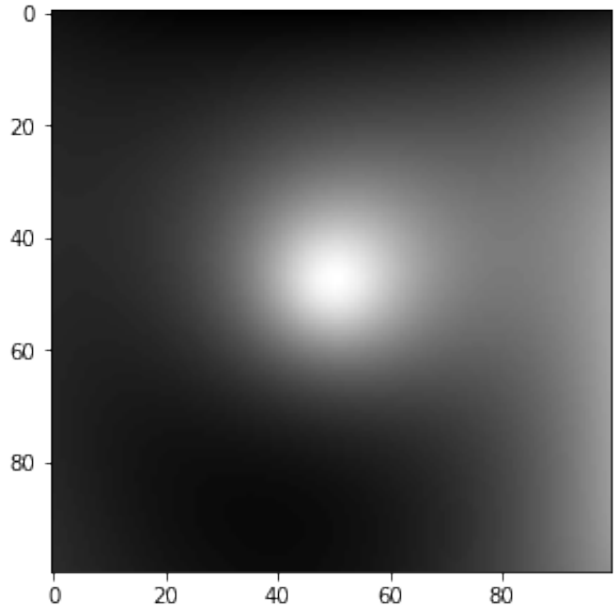



```
Light source direction: [[0.2 0.  1. ]]

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.

In [230…

```
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_rig

    return g
```

In [231…

```
def photometric_stereo(images, lights, mask, horn_niter=175000):

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

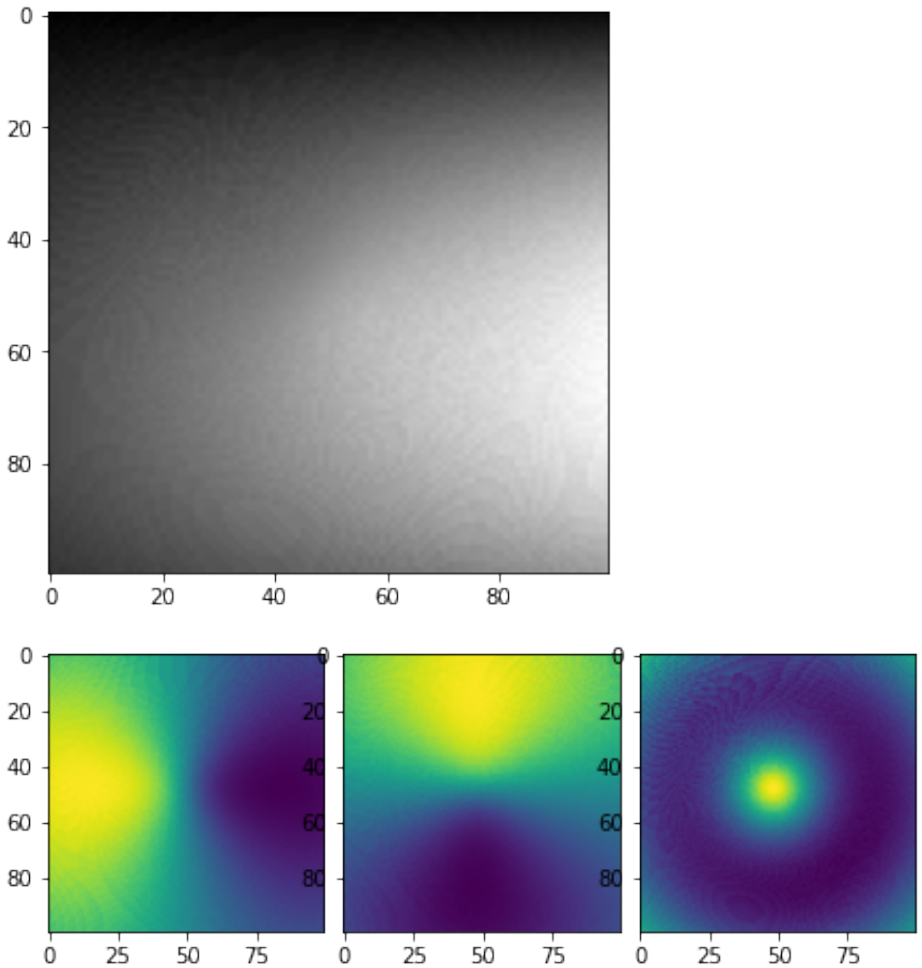
```
    The choice of threshold is something you can experiment with,
    but in practice something like 0.05 or 0.1 tends to work well.

    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 dat
    """

    # note:
    # images : (n_ims, h, w)
    # lights : (n_ims, 3)
    # mask   : (h, w)
    n_ims, h, w = images.shape
    albedo = np.zeros(images[0].shape)
    normals = np.dstack((np.zeros(images[0].shape),
                         np.zeros(images[0].shape),
                         np.ones(images[0].shape)))

    S = lights
    S_trans = np.transpose(S)
    STS = np.matmul(S_trans,S)
    STS_inv = np.linalg.inv(STS)
    b_ = np.matmul(STS_inv,S_trans)

    for x in range(h):
        for y in range(w):
            #if mask[x,y] != 0:
            img = images[:,x,y]
            b = np.dot(b_,img)
            albedo[x,y] = np.linalg.norm(b)
            normals[x,y,:] = b / np.linalg.norm(b)

    H = np.ones(images[0].shape)
    gx = np.zeros((h,w))
    gy = np.zeros((h,w))
    sum = 0
    for y in range(w):
        if mask[0,y] != 0:
            p = normals[0,y,0]/normals[0,y,2]
            sum += p
            H[0,y] = sum

    for y in range(w):
        sum = H[0,y]
        for x in range(1,h):
            if mask[x,y] != 0:
                q = normals[x,y,1]/normals[x,y,2]
                sum += q
                H[x,y] = sum
                gx[x,y] = normals[x,y,0]/normals[x,y,2]
                gy[x,y] = normals[x,y,1]/normals[x,y,2]

    H_horn = horn_integrate(gx, gy, mask, horn_niter)
    return albedo, normals, H, H_horn
```

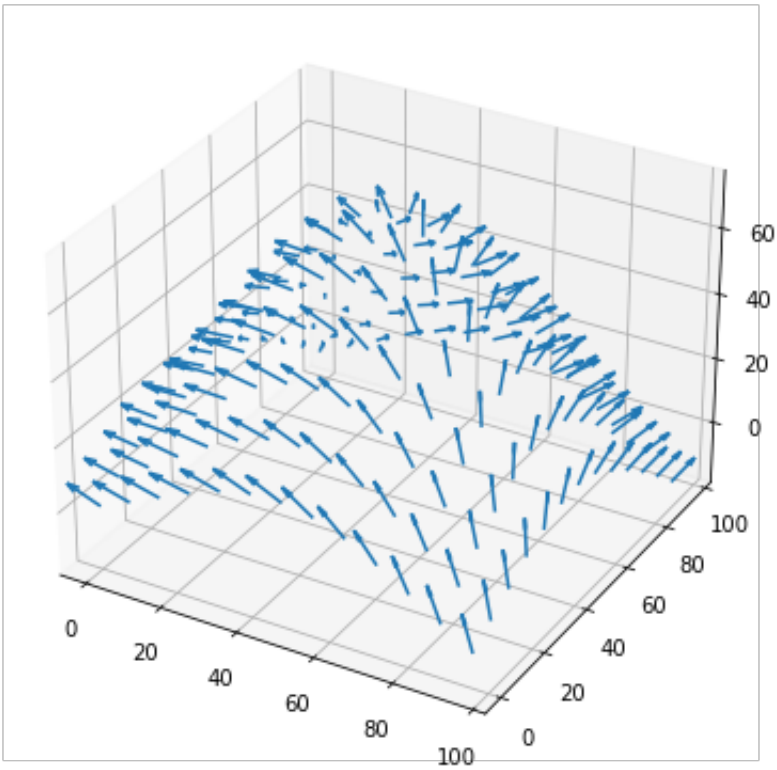
In [232…

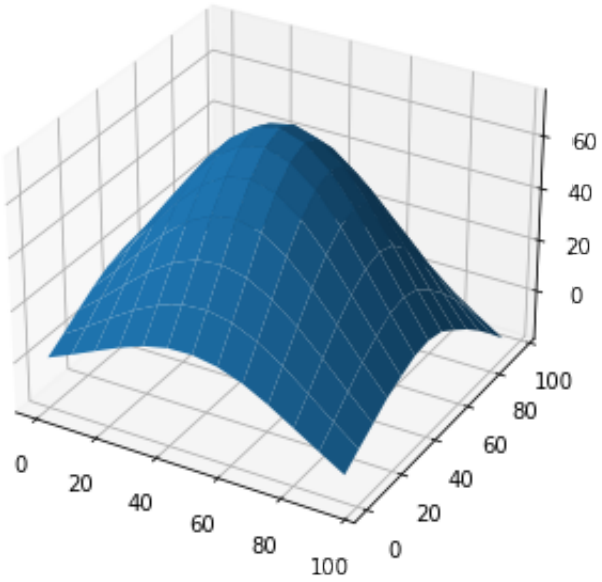
```
from mpl_toolkits.mplot3d import Axes3D
```

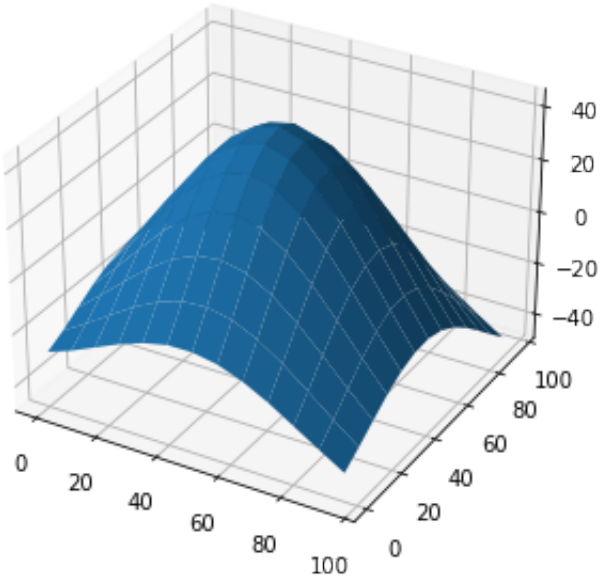
```
pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

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
# 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 = 8

    # 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
    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
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 8),
                          np.arange(0,np.shape(normals)[1], 8),
                          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
    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)
```




60
40
20
0
0
20
40
60
80
100
0
20
40
60
80
100

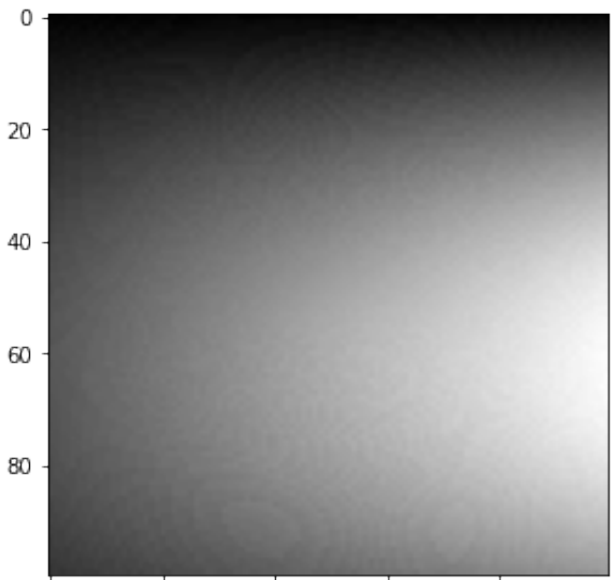

60
40
20
0
0
20
40
60
80
100
0
20
40
60
80
100

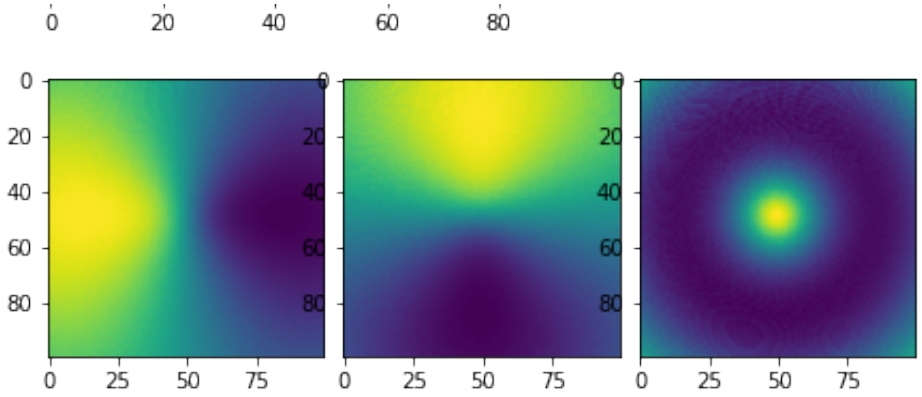


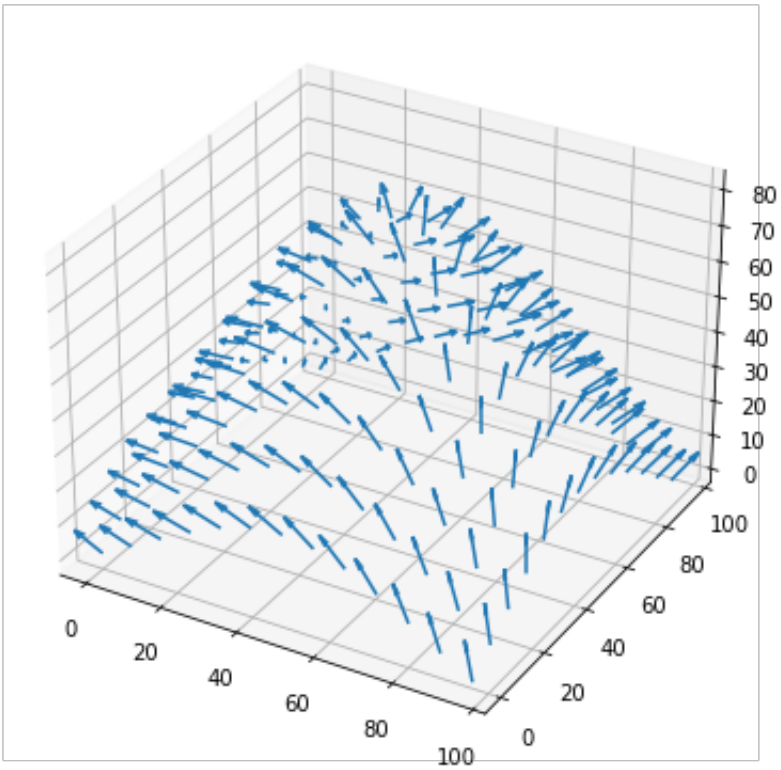
In [233…

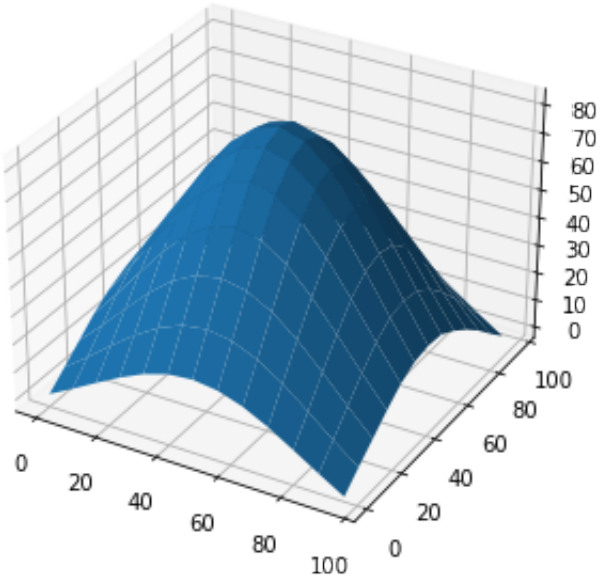
```
from mpl_toolkits.mplot3d import Axes3D

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

lights = np.vstack((data["l1"], data["l2"], data["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
# 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 visualize1(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 8

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

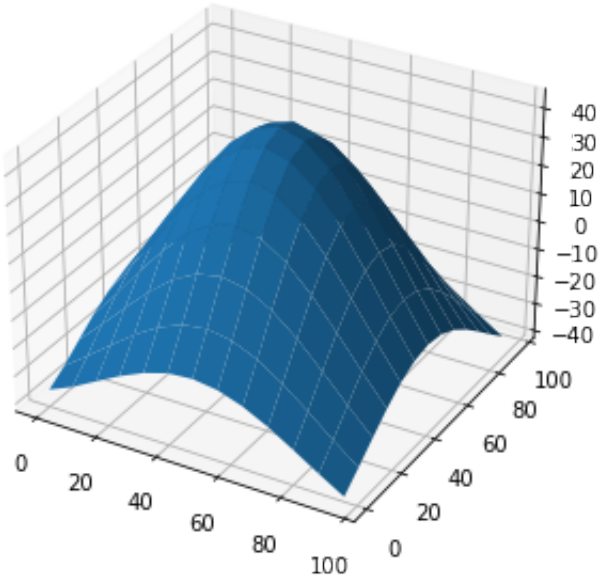
```
    # showing normals as three separate channels
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normals[..., 0])
    ax2 = figure.add_subplot(132)
    ax2.imshow(normals[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normals[..., 2])
    plt.show()

    # showing normals as quiver
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 8),
                          np.arange(0,np.shape(normals)[1], 8),
                          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
    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()

visualize1(albedo, normals, depth, horn)
```




0
20
40
60
80
0
20
40
60
80
0
25
50
75
0
25
50
75
0
25
50
75


0
20
40
60
80
100
0
20
40
60
80
100
0
10
20
30
40
50
60
70
80


0
20
40
60
80
100
0
20
40
60
80
100
0
10
20
30
40
50
60
70
80



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 $D = \sqrt{U^2 + V^2}$ to represent the diffuse part.

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

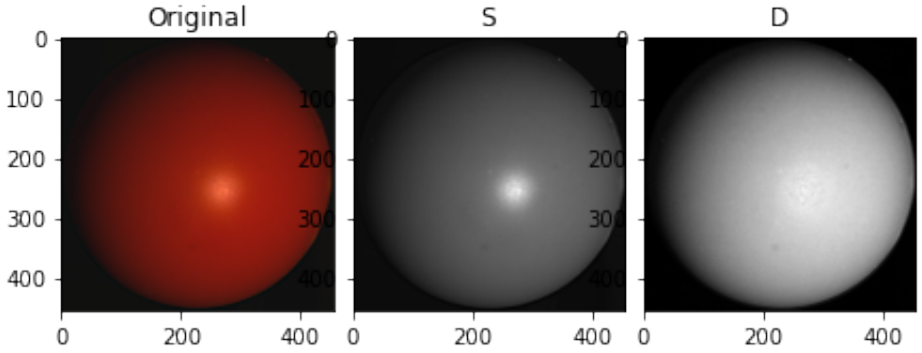
Note: You will find all the data for this part in `specular_sphere.pickle` and `specular_pear.pickle` .

In [234…

```
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:
```

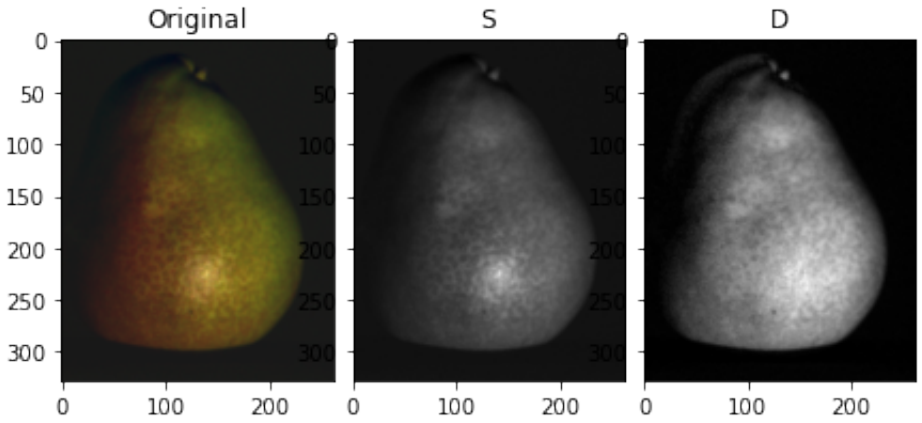
```
        unit = [1.0, 0.0, 0.0]

    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])
    D = np.ones(I_rgb.shape[:2])
    R = get_rot_mat(rot_vec)
    SUV = np.zeros(I_rgb.shape)
    for x in range(I_rgb.shape[0]):
       for y in range(I_rgb.shape[1]):
           SUV[x,y,:] = R@I_rgb[x,y,:]
    S = SUV[:,:,0]
    D = np.sqrt(np.power(SUV[:,:,1],2)+np.power(SUV[:,:,2],2))
    return S, D

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

# Sphere output
S, D = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                        data["c"][1][0],
                                        data["c"][2][0])))
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.title.set_text('Original')
ax1.imshow(normalize(data["im1"]))
ax2 = figure.add_subplot(132)
ax2.title.set_text('S')
ax2.imshow(S,cmap="gray")
ax3 = figure.add_subplot(133)
```

```
ax3.title.set_text('D')
ax3.imshow(D,cmap="gray")
plt.show()
```



In [235…

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

S, D = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                        data["c"][1][0],
                                        data["c"][2][0])))

figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.title.set_text('Original')
ax1.imshow(normalize(data["im1"]))
ax2 = figure.add_subplot(132)
ax2.title.set_text('S')
ax2.imshow(S,cmap="gray")
ax3 = figure.add_subplot(133)
ax3.title.set_text('D')
ax3.imshow(D,cmap="gray")
plt.show()
```



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.

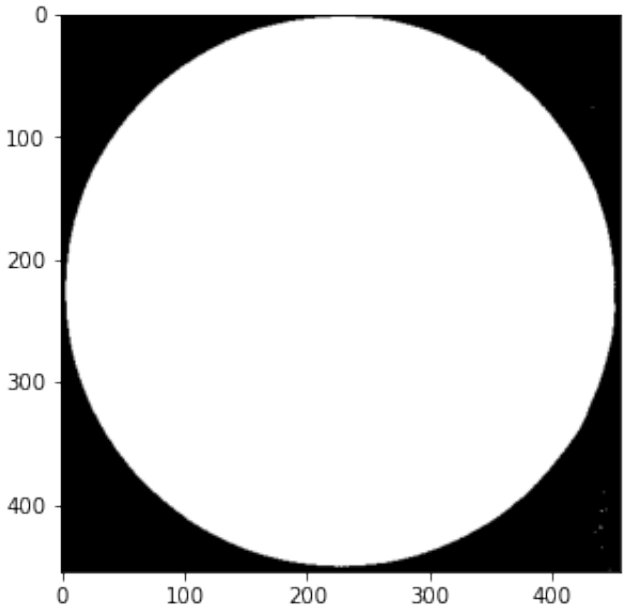
For the specular sphere image set in `specular_sphere.pickle`, using all of the four images (again, be sure to convert them to grayscale and normalize them so that their values go from 0 to 1), include:

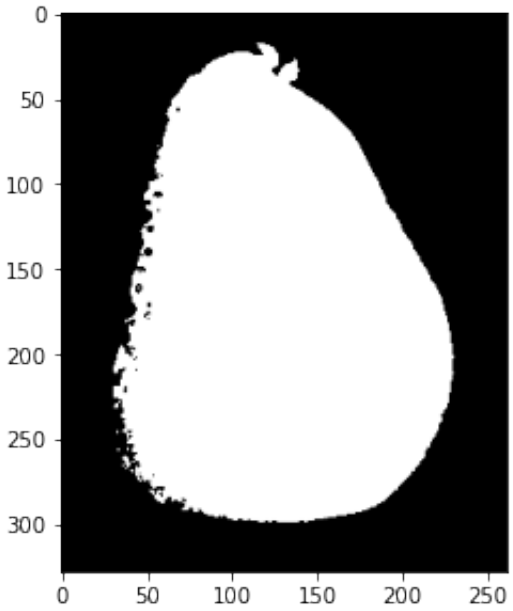
1. The estimated albedo map (original and diffuse).
2. The estimated surface normals (original and diffuse) by showing both
    A. Needle map, and
    B. Three images showing each of the surface normal components.
3. A wireframe of depth map (original and diffuse).
4. A wireframe of the depth map given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

In [236…

```
# MASK
"""mask is an optional parameter which you are encouraged to use.
    It can be used e.g. to ignore the background when integrating the norma
    It should be created by converting the images to grayscale, averaging t
    normalizing to [0, 1] and thresholding (only using locations for which
    pixel value is above some threshold).
"""
def to_grayscale(img):
    r, g, b = img[:,:,0], img[:,:,1], img[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray

def mask(data, threshold):
    gray = []
    im1_gray = to_grayscale(normalize(data["im1"]))
    gray.append(im1_gray)
    im2_gray = to_grayscale(normalize(data["im2"]))
    gray.append(im2_gray)
    im3_gray = to_grayscale(normalize(data["im3"]))
    gray.append(im3_gray)
    im4_gray = to_grayscale(normalize(data["im4"]))
    gray.append(im4_gray)
    average_image = (gray[0]+gray[1]+gray[2]+gray[3])/4
    avg_img_normal = normalize(average_image)
    h, w, _ = data["im1"].shape
    mask = np.zeros((h,w))
    for x in range(h):
        for y in range(w):
            if avg_img_normal[x,y] < threshold:
                mask[x,y] = 0
            else:
                mask[x,y] = 1
    return mask

# Create masks for sphere and pear
pickle_in = open("specular_sphere.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")
sphere_mask = mask(data,0.1)
plt.imshow(sphere_mask,cmap="gray")
plt.show()

pickle_in = open("specular_pear.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")
pear_mask = mask(data,0.13)
plt.imshow(pear_mask,cmap="gray")
plt.show()
```



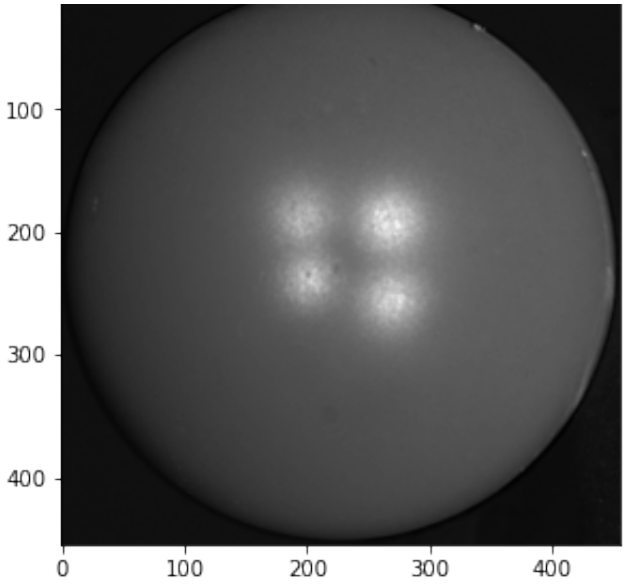


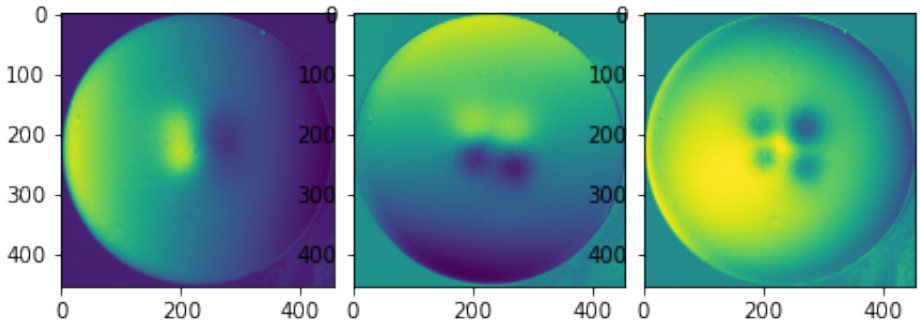
In [237…

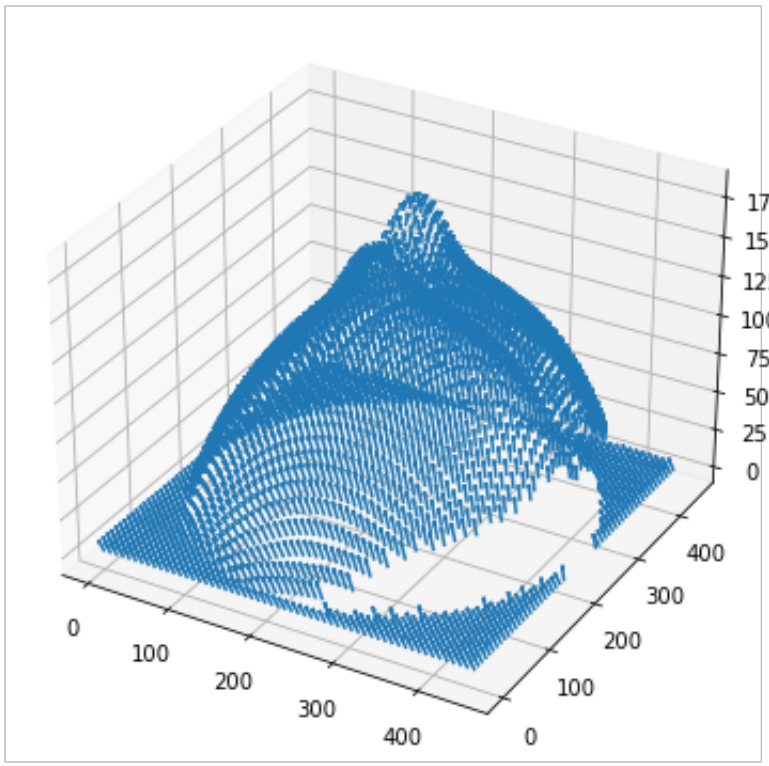
```
# SPHERE SPECULAR
from mpl_toolkits.mplot3d import Axes3D
pickle_in = open("specular_sphere.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

gray = []
im1_gray = to_grayscale(normalize(data["im1"]))
gray.append(im1_gray)
im2_gray = to_grayscale(normalize(data["im2"]))
gray.append(im2_gray)
im3_gray = to_grayscale(normalize(data["im3"]))
gray.append(im3_gray)
im4_gray = to_grayscale(normalize(data["im4"]))
gray.append(im4_gray)
gray = np.array(gray)


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

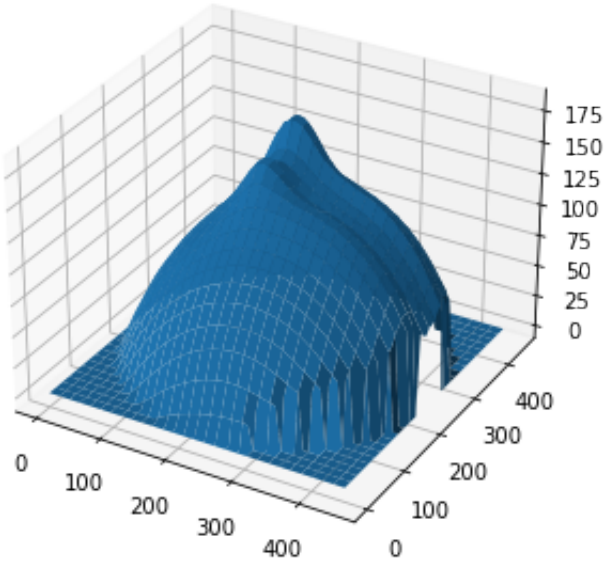
```
mask = sphere_mask

albedo_origs, normals_origs, depth_origs, horn_origs = photometric_stereo(g

def visualize2(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 8

    # 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
    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
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 8),
                          np.arange(0,np.shape(normals)[1], 8),
                          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
    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()

visualize2(albedo_origs, normals_origs, depth_origs, horn_origs)
```

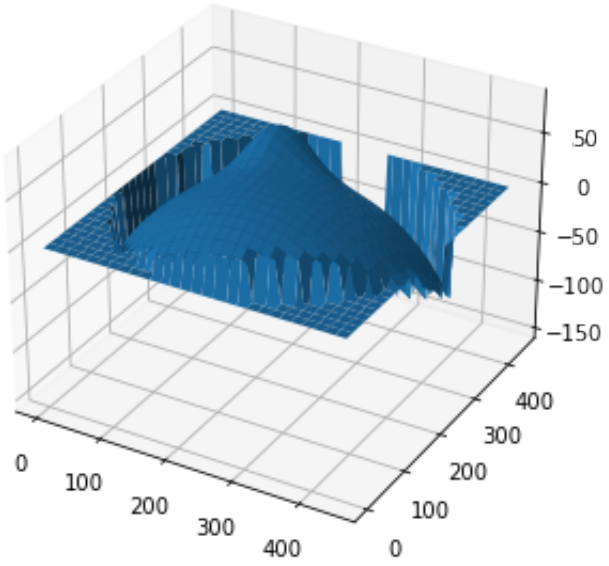
0


100
200
300
400
0
100
200
300
400


0
100
200
300
400
0
200
400


175
150
125
100
75
50
25
0
0
100
200
300
400
0
100
200
300
400



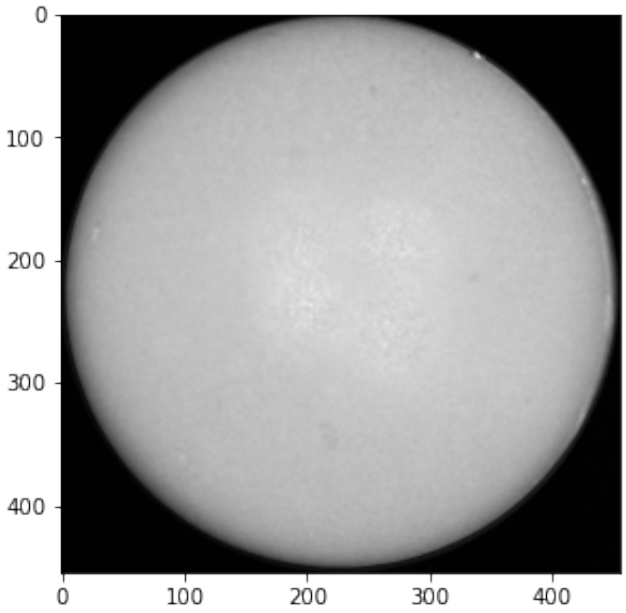


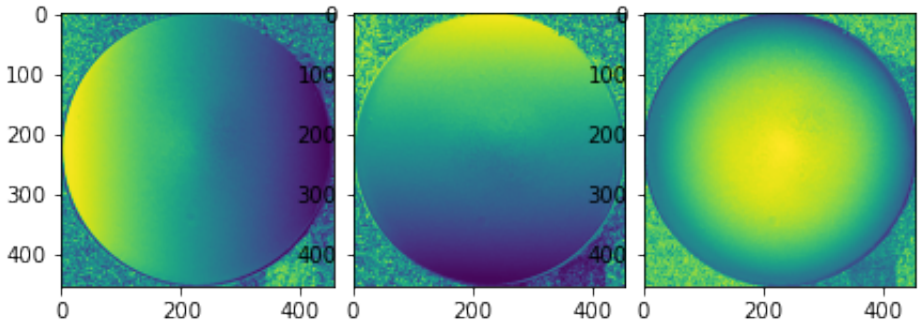
In [238…

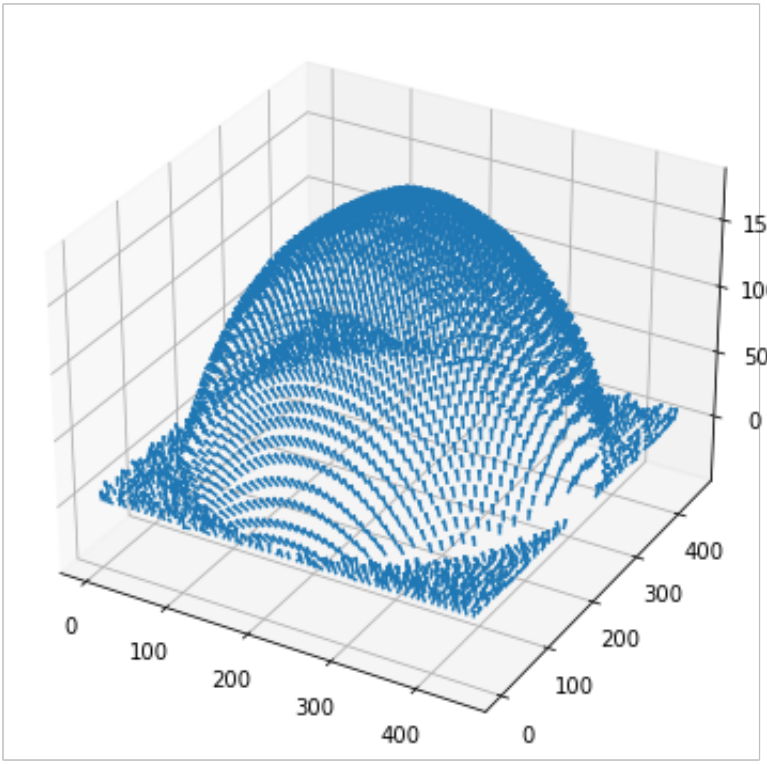
```
# SPHERE DIFFUSE
pickle_in = open("specular_sphere.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# Extracting Diffuse photo from photos
_, D1 = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D2 = RGBToSUV(data["im2"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D3 = RGBToSUV(data["im3"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D4 = RGBToSUV(data["im4"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
```

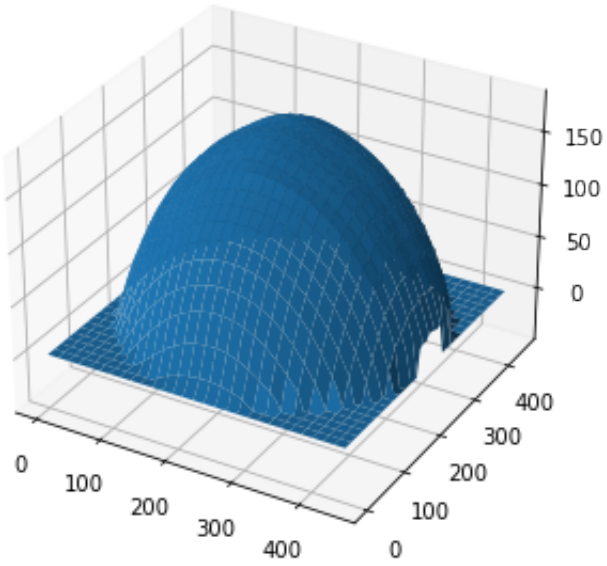
```
diff = []
diff.append(D1)
diff.append(D2)
diff.append(D3)
diff.append(D4)
diff = np.array(diff)

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

mask = sphere_mask

albedo_diffs, normals_diffs, depth_diffs, horn_diffs = photometric_stereo(d

def visualize3(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 8

    # 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
    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
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 8),
                          np.arange(0,np.shape(normals)[1], 8),
                          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
    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]
```

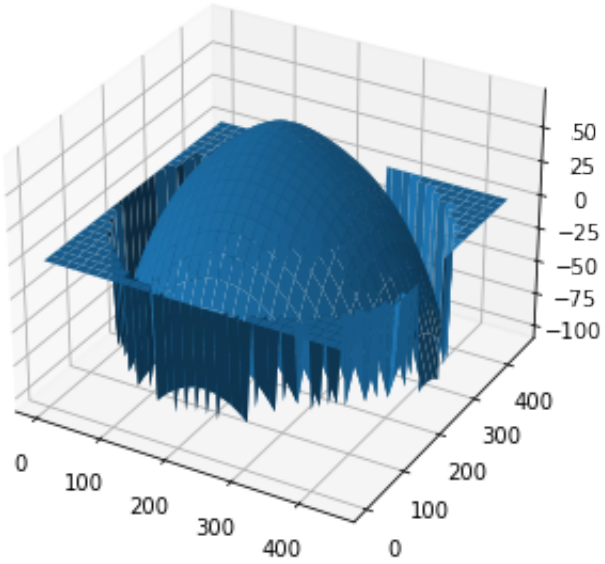
```
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

visualize3(albedo_diffs, normals_diffs, depth_diffs, horn_diffs)
```






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

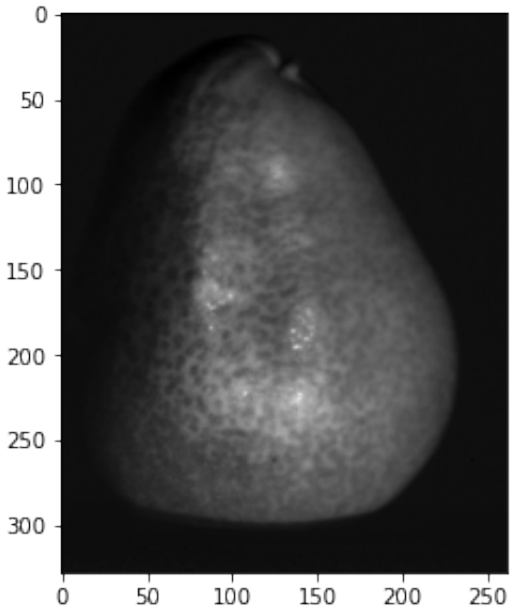

150
100
50
0
0
100
200
300
400
0
100
200
300
400

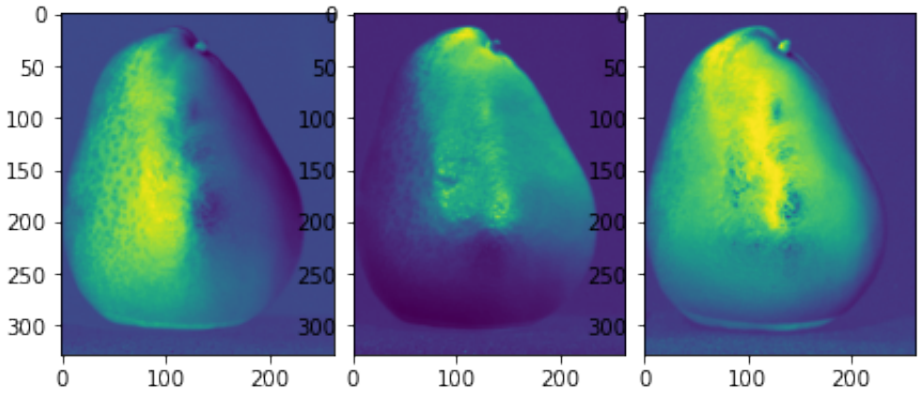


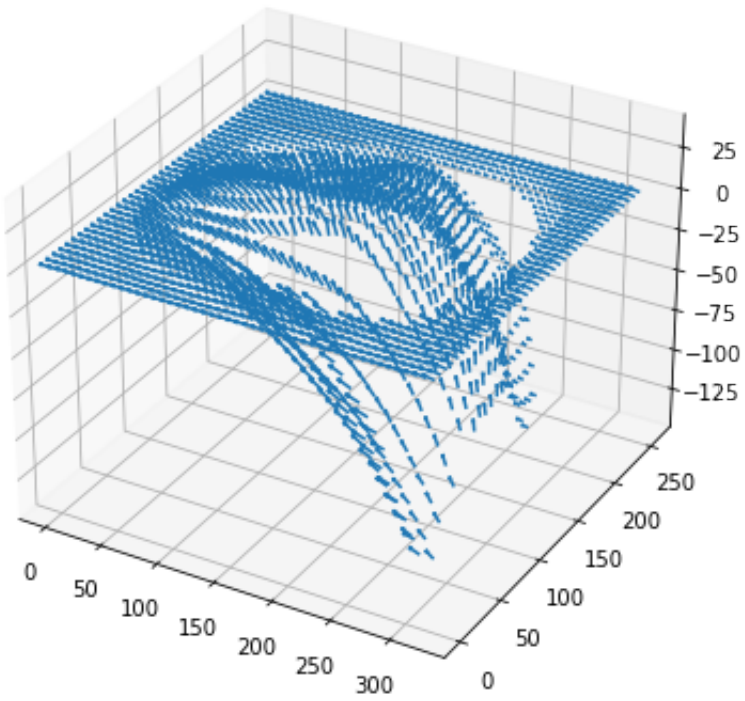
In [239…

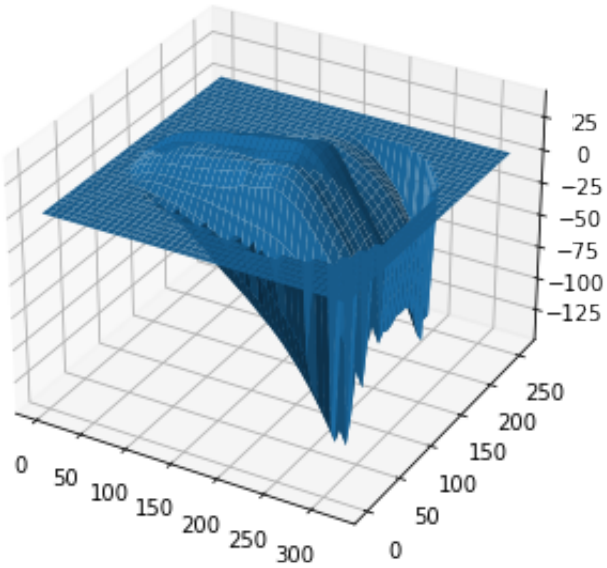
```
# PEAR SPECULAR
from mpl_toolkits.mplot3d import Axes3D
pickle_in = open("specular_pear.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

gray = []
im1_gray = to_grayscale(normalize(data["im1"]))
gray.append(im1_gray)
im2_gray = to_grayscale(normalize(data["im2"]))
gray.append(im2_gray)
im3_gray = to_grayscale(normalize(data["im3"]))
gray.append(im3_gray)
im4_gray = to_grayscale(normalize(data["im4"]))
gray.append(im4_gray)
gray = np.array(gray)

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

mask = pear_mask

albedo_origp, normals_origp, depth_origp, horn_origp = photometric_stereo(g

def visualize4(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 7

    # 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
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normals[..., 0])
```

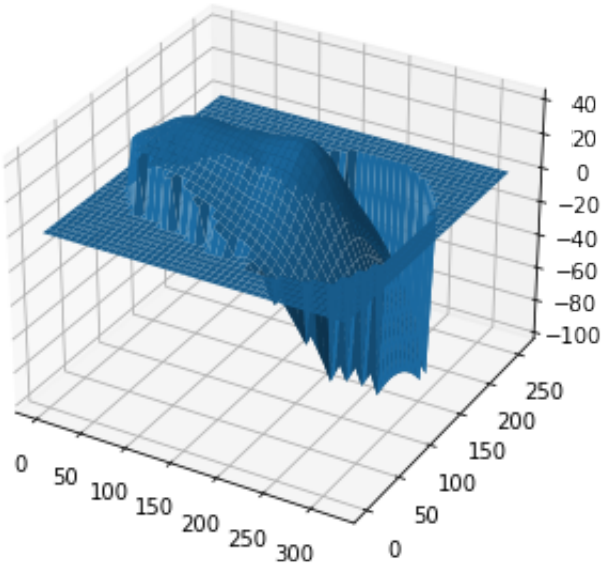
```
    ax2 = figure.add_subplot(132)
    ax2.imshow(normals[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normals[..., 2])
    plt.show()

    # showing normals as quiver
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 7),
                          np.arange(0,np.shape(normals)[1], 7),
                          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
    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()

visualize4(albedo_origp, normals_origp, depth_origp, horn_origp)
```




0
50
100
150
200
250
300
0
100
200


25
0
−25
−50
−75
−100
−125
0
50
100
150
200
250
300

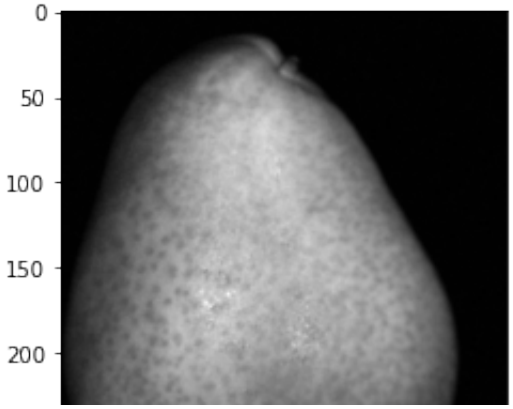

25
0
−25
−50
−75
−100
−125
0
50
100
150
200
250
300

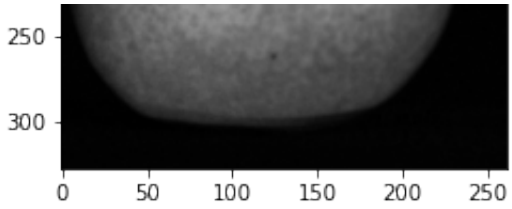


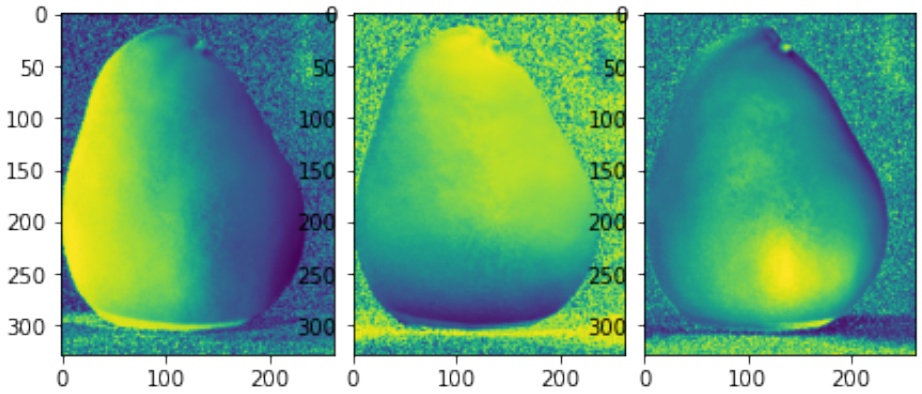
In [240…

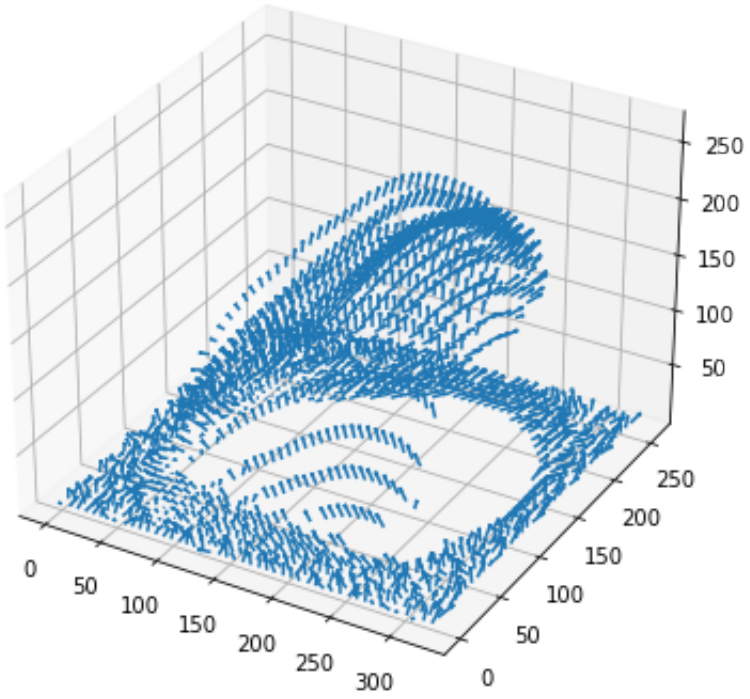
```
# PEAR DIFFUSE
pickle_in = open("specular_pear.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# Extracting Diffuse photo from photos
_, D1 = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D2 = RGBToSUV(data["im2"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D3 = RGBToSUV(data["im3"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))
_, D4 = RGBToSUV(data["im4"], np.hstack((data["c"][0][0],
                                         data["c"][1][0],
                                         data["c"][2][0])))

diff = []
diff.append(D1)
diff.append(D2)
diff.append(D3)
diff.append(D4)
diff = np.array(diff)

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

mask = pear_mask

albedo_diffp, normals_diffp, depth_diffp, horn_diffp = photometric_stereo(d

def visualize5(albedo, normals, depth, horn):
    # Stride in the plot, you may want to adjust it to different images
    stride = 7

    # showing albedo map
    fig = plt.figure()
```

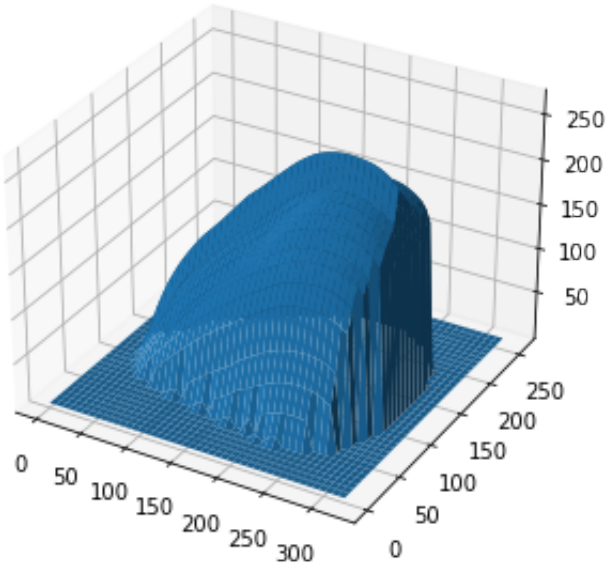
```
    albedo_max = albedo.max()
    albedo = albedo / albedo_max
    plt.imshow(albedo, cmap="gray")
    plt.show()

    # showing normals as three separate channels
    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
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 7),
                          np.arange(0,np.shape(normals)[1], 7),
                          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
    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()

visualize5(albedo_diffp, normals_diffp, depth_diffp, horn_diffp)
```

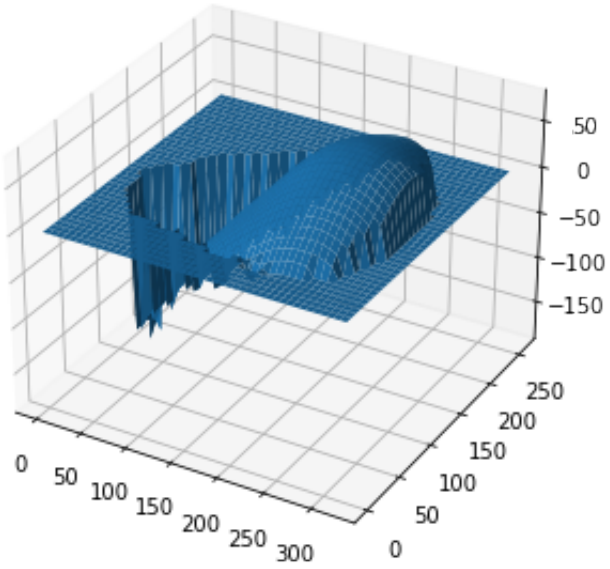



250
300
0
50
100
150
200
250


0
50
100
150
200
250
300
0
100
200


250
200
150
100
50
0
50
100
150
200
250
300


250
200
150
100
50
0
50
100
150
200
250
300
0
50
100
150
200
250


50
0
−50
−100
−150
0
50
100
150
200
250
300
0
50
100
150
200
250

In [ ]: