CSE252A_FA23_assignment_1

October 11, 2023

1 CSE 252A Computer Vision I Fall 2023 - Assignment 1

Instructor: Ben Ochoa

Due: Wed, Oct 25, 11:59 PM

1.1 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.

- 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.

- **You must submit 3 files on Gradescope - .pdf , .ipynb and .py file where the .py file is the conversion of your .ipynb to .py file . You must mark each problem on Gradescope in the pdf.** To convert the notebook to PDF, you can choose one way below:

  1. You can print the web page and save as PDF (e.g., Chrome: Right click the web page → Print… → Choose "Destination: Save as PDF" and click "Save").

  2. You can find the export option in the header: File → Download as → "PDF via LaTeX"

  To convert the notebook (.ipynb) to .py file use the following command:

jupyter nbconvert –to script filename.ipynb –output output_filename.py

- Please make sure the content in each cell (e.g., code, output images, printed results, etc.) are clearly visible and are not cut-out or partially cropped in your final PDF file.

- While submitting on gradescope, please make sure to assign the relevant pages in your PDF submission for each problem.

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.

1.2 Problem 1: Geometry (15 points)

Note: The solution **must be** typed using Markdown or Latex, handwritten answers will not be accepted.

Consider a line in the 2D plane, whose equation is given by $a\tilde{x} + b\tilde{y} + c = 0$, where $l = (a, b, c)^\top$ and $x = (\tilde{x}, \tilde{y}, 1)^\top$. Noticing that x is a homogeneous representation of $\tilde{x} = (\tilde{x}, \tilde{y})^\top$, we can view 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 $(3, 4)$ and $(7, -6)$.

3. [2 points] Find a vector that is a homogeneous representation of the line that passes through the points $(-2, 6)$ and $(9, 1)$.

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 $\times$ stands for the vector (or cross) product.

1.3 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 $\tilde{X}_1 = [2 \ 4 \ \text{-}5]^T$, $\tilde{X}_2 = [4 \ 2 \ \text{-}5]^T$, $\tilde{X}_3 = [\text{-}2 \ \text{-}4 \ \text{-}5]^T$, $\tilde{X}_4 = [\text{-}4 \ \text{-}2 \ \text{-}5]^T$ in the world coordinate frame. First, recall the following formula for rigid body transformation

$$\tilde{X}_{cam} = R\tilde{X} + t$$

Where $\tilde{X}_{cam}$ is the point coordinate in the camera coordinate system. $\tilde{X}$ is a point in the world coordinate frame, and R and t are the rotation and translation that transform points from the world coordinate frame to the camera coordinate frame. Together, R and 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 [I|0]. Given K, R, and t, the image of a point $\tilde{X}$ is $x = K[I|0]X_{Cam} = K[R|t]X$, where the homogeneous points $X_{Cam} = (\tilde{X}_{Cam}^\top, 1)^\top$ and $X = (\tilde{X}^\top, 1)^\top$. 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 example output in figure below).

1. [No rigid body transformation]. Focal length = 2. The optical axis of the camera is aligned with the z-axis.

2. [Translation]. Focal length $= 2$. $t = [0\ 0\ 3]^T$. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length $= 2$. R encodes a 45 degrees around the z-axis and then 30 degrees around the y-axis. $t = [0\ 0\ 3]^T$.
4. [Translation and Rotation, long distance]. Focal length $= 4$. R encodes a 45 degrees around the z-axis and then 30 degrees around the y-axis. $t = [0\ 0\ 8]^T$.

We will not use a full intrinsic camera matrix (e.g., that maps centimeters to pixels, and specifies the coordinates of the principal point), 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.

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


def to_homog(points):
    # convert points from inhomogeneous to homogeneous
    #
    # inputs:
    # points is a dxn matrix where n is the number of d dimensional␣
    ↪inhomogeneous points
    # (e.g., d = 3 for 3D inhomogeneous points)
    #
    # outputs:
    # homo_points is a (d+1)xn matrix of n d-dimensional homogeneous points

    # write your code here

    return homo_points


def from_homog(points_homog):
    # convert points from homogeneous to inhomogeneous
    #
    # inputs:
    # points_homog is a (d+1)xn matrix of n d-dimensional homogeous points (e.g.
    ↪, d = 2
    # for 2D homogeneous points)
    #
    # outputs:
    # inhomo_points is a dxn matrix of n d-dimensional inhomogeous points
```

```python
    # write your code here

    return inhomog_points


def project_points(P_int, P_ext, pts):
    # project 3D inhomogeneous points to 2D inhomogeneous points
    #
    # inputs:
    #    P_int - 3x3 intrinsic camera matrix
    #    P_ext - 3x4 extrinsic camera matrix
    #    pts - 3xn inhomogeneous points
    #
    # outputs:
    #    pts_2d - 2xn inhomogeneous points

    # write your code here

    return pts_2d


# 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␣
 ↪formula
# (if one exists) being used to fill in the matrices. Feel free to document with
# comments any thing you feel the need to explain.


def intrinsic_cam_mat(f):
    """
    K = [f 0 0
         0 f 0
         0 0 1]
    """
    # given the focal length, compute the intrinsic camera matrix

    # write your code here
    return int_cam_mat

def extrinsic_cam_mat(angles, t):
    """
        ext_cam_mat = [R|t]
    """
    # Compute the extrinsic camera matrix
    #
    # inputs:
```

```python
    #     angles - a tuple of angles (alpha, beta, gamma), representing the
 →rotation
    #     angles around x-axis, y-axis, and z-axis repectively in degrees.
    #
    # outputs:
    #     ext_cam_mat - 3x4 extrinsic camera matrix


    # write your code here
    return ext_cam_mat

def camera1():
    """
    replace with your code
    """
    # write your code here
    return P_int_proj, P_ext


def camera2():
    """
    replace with your code
    """
    # write your code here
    return P_int_proj, P_ext


def camera3():
    """
    replace with your code
    """
    # write your code here
    return P_int_proj, P_ext


def camera4():
    """
    replace with your code
    """
    # write your code here
    return P_int_proj, P_ext



#####################################################
# Do not modify
#####################################################

def plot_points(ax, points, title='', style='.-b', axis=[]):
    inds = list(range(points.shape[1]))+[0]
    ax.plot(points[0,inds], points[1,inds],style)
```

```
    if title:
        ax.set_title(title)
    if axis:
        ax.axis('scaled')

def main():
    point1 = np.array([[2,4,-5]]).T
    point2 = np.array([[4,2,-5]]).T
    point3 = np.array([[-2,-4,-5]]).T
    point4 = np.array([[-4,-2,-5]]).T
    points = np.hstack((point1,point2,point3,point4))

    fig = plt.figure(figsize=(10,7))
    for i, camera in enumerate([camera1, camera2, camera3, camera4]):
        P_int_proj, P_ext = camera()
        ax = fig.add_subplot(2,2,i+1)
        plot_points(ax, project_points(P_int_proj, P_ext, points),␣
↪title='Camera %d Projective'%(i+1), axis=[-1,1,-1,1])
    plt.tight_layout()
    plt.show()

main()
```

1.4 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.

1.4.1 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.

Masks serve the purpose as the binary_mask given to you last assignment0. Mask is a HxW matrix with each element being either 1 or 0, indicating if a pixel at given location is sphere/pear or not. It is provided for you to remove the background of the rendered sphere/pear.

Assume that the albedo map is uniform.

1.4.2 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 $\hat{\mathbf{s}}$ and the normal vector on the surface of the object $\hat{\mathbf{n}}$. The brightness intensity at a given point on an object's surface, e, with a single light source is found using the following relationship:

$$e = aC \max(0, \hat{\mathbf{n}}^\top \mathbf{s})$$

$$\mathbf{s} = s_0 \hat{\mathbf{s}}$$

where, a is the albedo of the surface facet imaged by pixel, C is the light color, s_0 is the intensity of the light source and you can assume that the intensity is 1 here, and $\hat{\mathbf{s}}$ is the unit direction to the light source from the surface facet imaged by pixel.

1.4.3 Blinn-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 Blinn-Phong reflectance model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Blinn-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 Blinn-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 $\hat{\mathbf{v}}$ must also be known. For some scene with known material parameters with M light sources the light intensity, e, on a surface with normal vector $\hat{\mathbf{n}}$ seen from viewing direction $\hat{\mathbf{v}}$ can be computed by:

$$e = \sum_{m \in M} \left\{ s_{m,a} k_a + s_{m,d} k_d f_d + s_{m,s} k_s f_s \right\},$$

$$f_d = \max(0, \hat{\mathbf{n}}^\top \hat{\mathbf{s}}), \quad f_s = \max(0, \hat{\mathbf{n}}^\top \hat{\mathbf{h}})^\alpha$$

$$\hat{\mathbf{h}} = \frac{\mathbf{h}}{||\mathbf{h}||}, \quad \mathbf{h} = \hat{\mathbf{s}} + \hat{\mathbf{v}}$$

where $s_{m,a}$, is the ambient light intensity, $s_{m,d}$ and $s_{m,s}$ are the intensity of the the diffuse and specular light respectively for the mth light source.

1.4.4 Rendering

Please complete the following:

1. Write the function `lambertian()` that calculates the Lambertian light intensity given the light direction $\hat{\mathbf{s}}$ with intensity s_0, and normal vector $\hat{\mathbf{n}}$. 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.*

7

2. Write the function `blinn_phong()` that calculates the Blinn-Phong light intensity given the material constants (k_a, k_d, k_s, α), $\hat{\mathbf{v}} = (0, 0, 1)^\top$, $\hat{\mathbf{n}}$ 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.

Table 1: Light Sources

m	Location	Color (RGB)
1	$(-\frac{1}{2}, \frac{1}{2}, \frac{1}{2})^\top$	$(1, 1, 1)$
2	$(1, 0, 0)^\top$	$(1, .45, 1)$

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

1.4.5 Part 1. Plot 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 plot the normals using the function plot_normals which is provided.

```python
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()
```

```
[ ]: #Plot the normals for the sphere and pear for both the normal and diffuse␣
      ↪components.
     #1 : Load the different normals
     import pickle

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

     #2 : Plot the normals using plot_normals
     # What do you observe? What are the differences between the diffuse component␣
      ↪and the original images shown?
     # (Just something to think about, no need to provide an answer, but feel free␣
      ↪to add markdown cells to explain your thoughts)

     #PLOT HERE
```

1.4.6 Part 2. Lambertian model [6 pts]

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

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

```
[ ]: def lambertian(normals, lights, color, intensity, mask):
         image = np.zeros((normals.shape[0], normals.shape[1], 3))

         '''Your implementation'''



         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-

9

1)Mask for the sphere
2)Mask for the pear

Plot the normalized image using the function normalize which is provided.

With 2 light directions and 2 light colors, we expect 4 images for Pear and 4 images for Sphere.

```python
# Load the masks for the sphere and pear
with open('masks.pkl', 'rb') as h:
    data1 = pickle.load(h)

# Output the rendering results for Pear
# Read light direction and color from the table
dirn1 = np.zeros([3,1])
color1 = np.zeros([1,3])
dirn2 = np.zeros([3,1])
color2 = np.zeros([1,3])

#Display the rendering results for pear for both diffuse and for both the light
 sources
```

```python
# Output the rendering results for Sphere
dirn1 = np.zeros([3,1])
color1 = np.zeros([1,3])
dirn2 = np.zeros([3,1])
color2 = np.zeros([1,3])
#Display the rendering results for sphere for both diffuse and for both the
 light sources
```

1.4.7 Part 3. Blinn-Phong model [8 pts]

Please fill in your implementation for the Blinn-Phong model below.

```python
def blinn_phong(normals, lights, color, material, view, mask):
    '''Your implementation'''
    return (image)
```

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

With 2 light sources and 3 materials, we expect 9 images each for diffuse and original components. In total, 3*3*2 = 18 images for sphere and 18 images for pear.

```python
# 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([[0,1],[1,1],[0,0.5]])
```

```
#Display rendered results for sphere for all materials and light sources and␣
 ↪combination of light sources
```

```
[ ]: # 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([[0,1],[1,1],[0,0.5]])
     #Display rendered results for pear for all materials and light sources and␣
      ↪combination of light sources
```

1.5 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.

1.5.1 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.

1.5.2 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 10000 (ten 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).

Note: DO NOT normalize the images prior to use in the photemetric stero algorithm. the images must be used **as-is**.

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.)

```python
# 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.
fig = plt.figure(figsize=(7,7))
for i in range(1, 5):
    sub = fig.add_subplot(int('22'+str(i)))
    sub.set_title('Image ' + str(i))
    sub.set_xlabel("Light source direction: " + str(data["l%d" % i]))
    sub.imshow(data["im%d" % i], cmap="gray")
plt.tight_layout()
plt.show()
```

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.

```python
import numpy as np
from scipy.signal import convolve

def horn_integrate(gx, gy, mask, niter):
    """
    Inputs:
    - 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 between 10,000 - 100,000,
    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
```

```python
    return g
```

```python
'''Function to compute the albego, normals,
    and height map using the photometric stereo
    method & horn integration'''
def photometric_stereo(images, lights, mask, horn_niter=25000):

    """
    *** Input images should not be normalized to [0, 1] range, use as-is.

    *** mask is an optional parameter used to ignore the background when
    integrating the normals. 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, just pass a
    mask of all ones),
    but you SHOULD use it to filter out the background for the specular data
    (1c).
    """


    """ ==========
    YOUR CODE HERE
    ========== """

    # 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)
    return albedo, normals, H, H_horn
```

```python
# ----------------------------------------------------------------------
# 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, imtitle='', stride = 15):
    # showing albedo map
```

```python
fig = plt.figure(figsize=(12,10))
albedo_max = albedo.max()
albedo = albedo / albedo_max
ax0 = fig.add_subplot(231)
ax0.set_title("Albedo")
ax0.imshow(albedo, cmap="gray")

# showing normals as three separate channels
ax1 = fig.add_subplot(253)
fig.colorbar(ax1.imshow(normals[..., 0]), ax=ax1, orientation='horizontal')
ax2 = fig.add_subplot(254)
ax2.set_title("Normals as 3 separate channels.")
fig.colorbar(ax2.imshow(normals[..., 1]), ax=ax2, orientation='horizontal')
ax3 = fig.add_subplot(255)
fig.colorbar(ax3.imshow(normals[..., 2]), ax=ax3, orientation='horizontal')

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], stride),
                      np.arange(0,np.shape(normals)[1], stride),
                      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
ax4 = fig.add_subplot(234, projection='3d')
ax4.set_title("Needle map")
ax4.quiver(X,Y,Z,NX,NY,NZ, length=10, color='g')

# plotting wireframe depth map
H = depth[::stride,::stride]
ax5 = fig.add_subplot(235, projection='3d')
ax5.set_title("Wireframe - PS")
ax5.plot_surface(X,Y, H.T, color='g')

H = horn[::stride,::stride]
ax6 = fig.add_subplot(236,projection='3d')
ax6.set_title("Wireframe - HORN")
ax6.plot_surface(X,Y, H.T, color='g')
fig.suptitle(imtitle)
plt.show()
```

```python
# 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}.)
# Some code is given to you below to help you get started.
```

```
from mpl_toolkits.mplot3d import Axes3D

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

lights =

images = []

mask = np.ones(data["im1"].shape) # these images don't have a background, so␣
 ↪we'll just use a mask of all ones

albedo, normals, depth, horn = photometric_stereo(images, lights, mask)
visualize(albedo, normals, depth, horn, "Photometric Sterio - Synthetic data (3␣
 ↪images)", 5)
```

1.5.3 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 $D = \sqrt{U^2 + V^2}$ 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`.

```
[ ]: 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
```

```python
        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.
    '''


    """ ==========
    YOUR CODE HERE
    ========== """

    S = np.ones(I_rgb.shape[:2])
    G = np.ones(I_rgb.shape[:2])
    return S, G

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

# sample input
S, G = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                        data["c"][1][0],
                                        data["c"][2][0])))
```

1.5.4   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 **Part 1** on the original images (converted to grayscale). 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 **Part 1**.

Next, combine parts 1 and 2 by removing the specularities (using your code from **Part 2**) 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.

Note: While creating the masks, please use 0.1 as your threshold. **DO NOT** "normalize" or ortherwise modify the images prior to use in the photometric stereo algorithm. The images must be used **as-is**.

For the specular sphere image set in `specular_sphere.pickle` and specular pear images set in `specular_pear.pickle`, using all of four images in each, 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.)

```
# -------------------------------------------------------------------------
# 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.
# -------------------------------------------------------------------------
```