

CSE 152A Fall 2023 – Assignment 1

- Assignment Published On: **Tue, Oct 10, 2023**
- Due On: **Sat, Oct 21, 2023 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 posted on lecture slides.
- 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 open source codes 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.
- Make sure that you read hints for questions (wherever given).

Late Policy: Assignments submitted late will receive a 25% grade reduction for each 12 hours late (that is, 50% per day).

Problem 1: Photometric Stereo [20 pts]

The goal of this problem is to implement Lambertian photometric stereo.

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.

Data

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

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

You will find all the data for this part in `synthetic_data.pickle`.

Note: Make sure to use `%matplotlib inline` to enable plotting by matplotlib in jupyter notebook/lab IDE

```
In [ ]: ## Example: How to read and access data from a pickle
```

```
import pickle
import matplotlib.pyplot as plt
%matplotlib inline
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: " + str(data.keys()))

# To access the value of an entity, refer it by its key.
print("Image:")
plt.imshow(data["im1"], cmap = "gray")
plt.show()

print("Light source direction: " + str(data["l1"]))

plt.imshow(data["im2"], cmap = "gray")
plt.show()

print("Light source direction: " + str(data["l2"]))

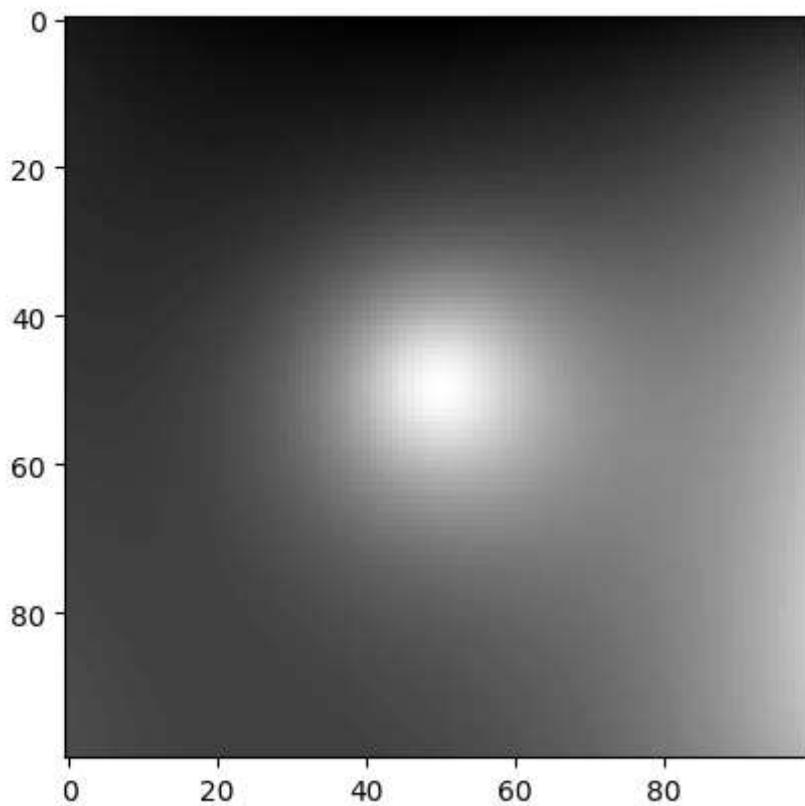
plt.imshow(data["im3"], cmap = "gray")
plt.show()

print("Light source direction: " + str(data["l3"]))

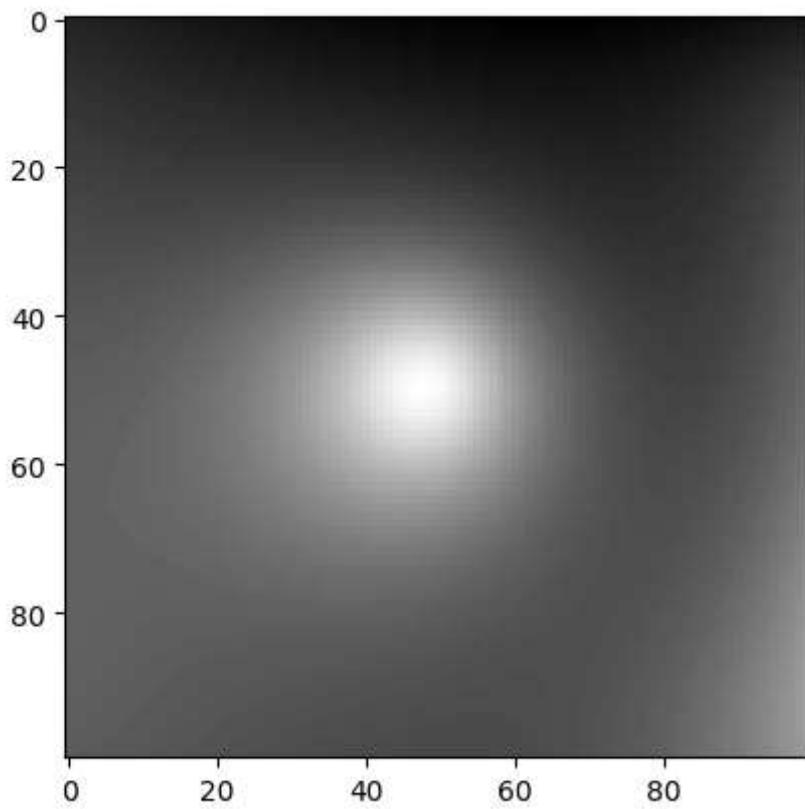
plt.imshow(data["im4"], cmap = "gray")
plt.show()

print("Light source direction: " + str(data["l4"]))
```

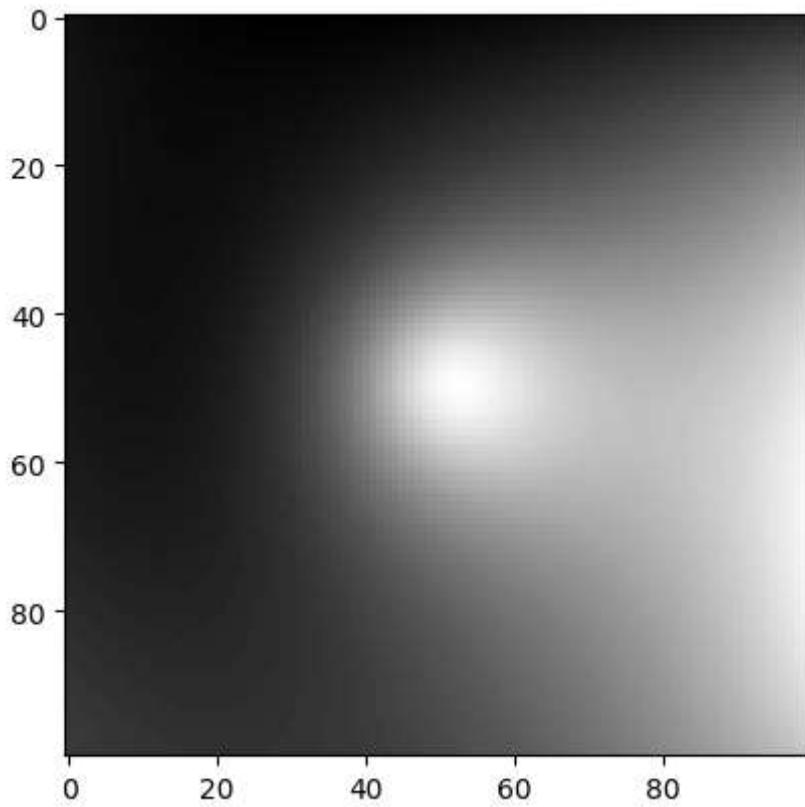
Keys: dict_keys(['__version__', 'l4', '__header__', 'im1', 'im3', 'im2', 'l2', 'im4', 'l1', '__globals__', 'l3'])
Image:



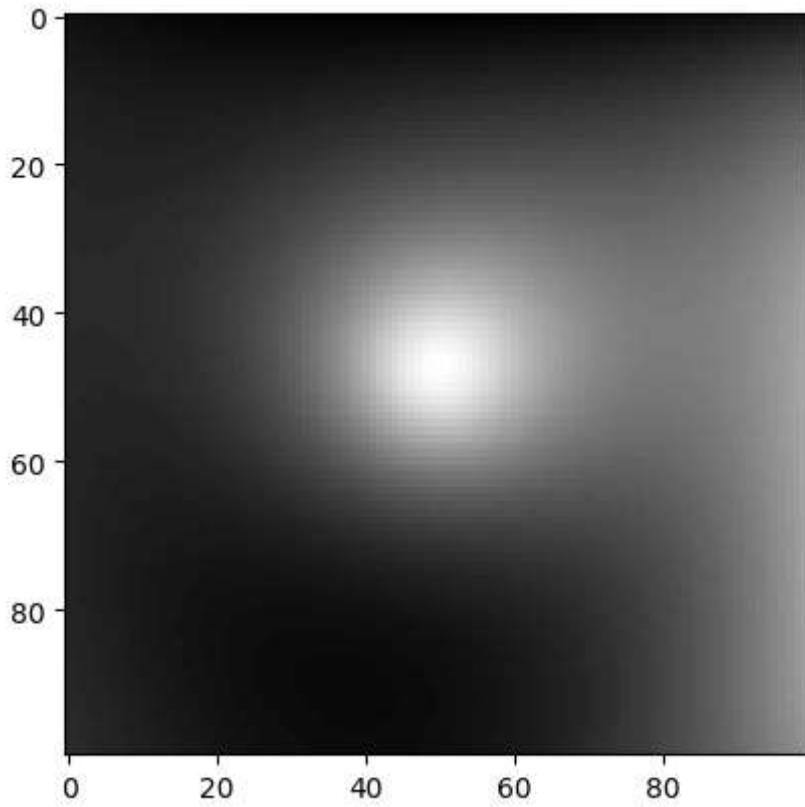
Light source direction: [[0 0 1]]



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



Light source direction: $[-0.2 \quad 0. \quad 1. \quad]$



Light source direction: $[0. \quad 0.2 \quad 1. \quad]$

1(a) Photometric Stereo [8 pts]

Implement the photometric stereo technique described in the lecture. Your program should have two parts:

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

- Reconstruct the depth map from the surface with the implementation of the Horn integration technique given below in `horn_integrate` function. Note that you will typically want to run the `horn_integrate` function with 10000 - 100000 iterations, meaning it will take a while.

Note: The following `horn_integrate()` function is not to be edited for the purpose of solving this homework. Consider this to be a blackbox!

In []:

```

import numpy as np
from scipy.signal import convolve
from numpy import linalg

# This function should be used as is for this homework.
def horn_integrate(gx, gy, mask, niter):
    """
    horn_integrate recovers the function g from its partial
    derivatives gx and gy.
    mask is a binary image which tells which pixels are
    involved in integration.
    niter is the number of iterations.
    typically 100,000 or 200,000,
    although the trend can be seen even after 1000 iterations.
    """
    g = np.ones(np.shape(gx))

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

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

    d_mask = A + B + C + D

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

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

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

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

    return g

```

!! Hint !!

For computing partial derivatives, please note that this specific question assumes the following convention compared to the one presented in the class (that contains a negative sign).

$$\frac{\partial z}{\partial x} = \frac{\hat{n}_1}{\hat{n}_3} \quad (1)$$

$$\frac{\partial z}{\partial y} = \frac{\hat{n}_2}{\hat{n}_3} \quad (2)$$

Make sure to use the above convention to avoid painful debugging later on! Also, double check that the normals being used for plotting are unit normals.

```
In [ ]: def photometric_stereo(images, lights, mask, horn_niter=25000):
    """mask is an optional parameter which you are encouraged to use.
    It can be used e.g. to ignore the background when integrating the normals.
    It should be created by converting the images to grayscale and thresholding
    (only using locations for which the pixel value is above some threshold).

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

    # 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_horn = np.ones(images[0].shape)

    h, w = images[0].shape
    S_inv = np.linalg.pinv(lights)
    normals = np.zeros((h, w, 3))
    gx = np.zeros((h, w))
    gy = np.zeros((h, w))

    for i in range(h):
        for j in range(w):
            I = images[:, i, j]
            g = S_inv @ I
            norm = np.linalg.norm(g)
            normal = g / norm
            normals[i, j] = normal
            albedo[i, j] = norm

            gx = normals[:, :, 0] / normals[:, :, 2]
            gy = normals[:, :, 1] / normals[:, :, 2]
            H_horn = horn_integrate(gx, gy, mask, horn_niter)

    return albedo, normals, H_horn
```

1(b) Display outputs using `im1`, `im2` and `im4` [6 Points]

- The estimated albedo map.
- The estimated surface normals by showing both
 - Needle map, and
 - Three images showing the components of the surface normal.
- A wireframe of the depth map.

An example of the outputs are shown below:



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

# -----
# 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, horn_depth):
    # Stride in the plot, you may want to adjust it to different images
    stride = 15

    # 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], 15),
        np.arange(0, np.shape(normals)[1], 15),
        np.arange(1),
    )
    X = X[:, :, 0]
    Y = Y[:, :, 0]
    Z = horn_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.add_subplot(projection="3d")
```

```
plt.quiver(Y, X, Z, NX, NY, NZ, length=10)
plt.show()

# plotting wireframe depth map

H = horn_depth[::stride, ::stride]
fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.plot_surface(X, Y, H.T)
plt.show()
```

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

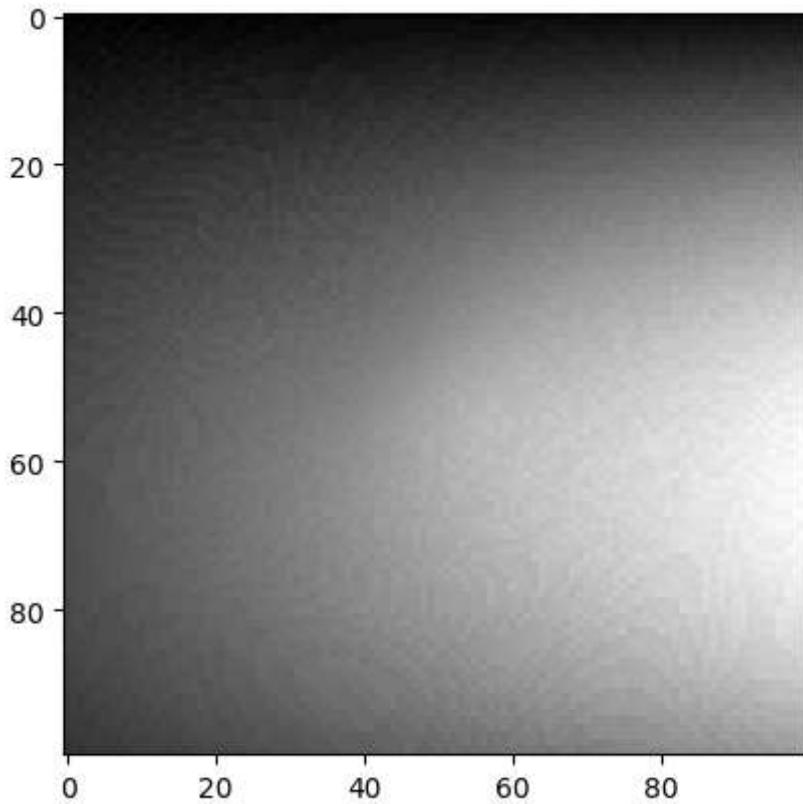
lights = np.vstack((data["l1"], data["l2"], data["l4"]))

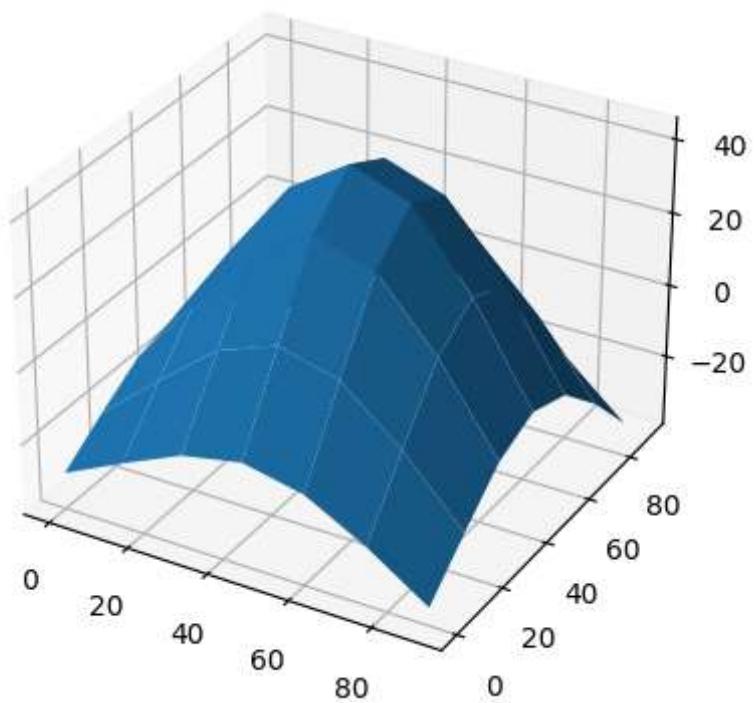
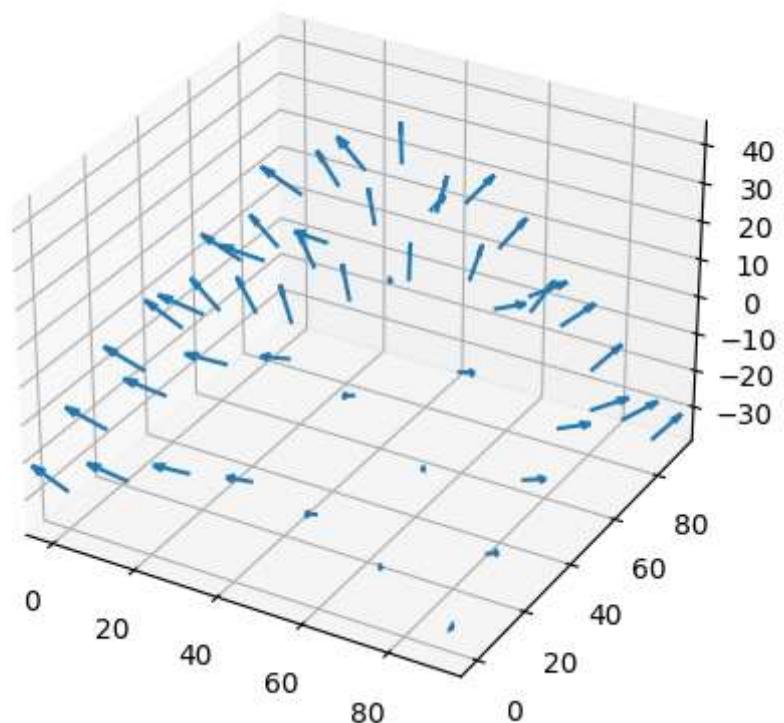
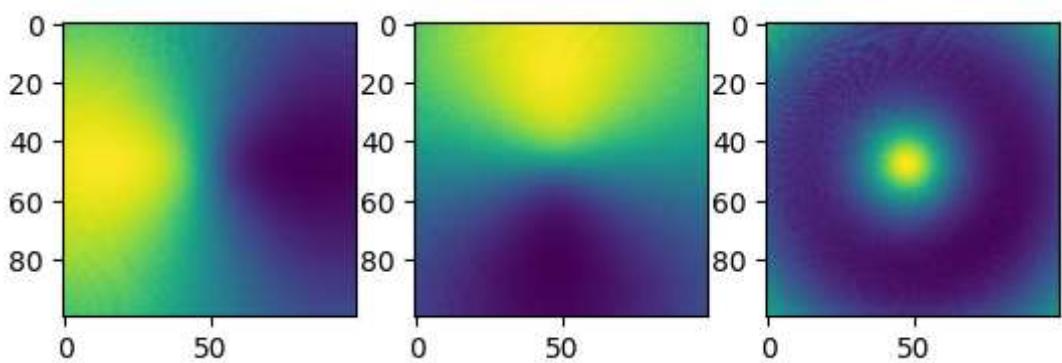
images = []
images.append(data["im1"])
images.append(data["im2"])
images.append(data["im4"])
images = np.array(images)

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

albedo, normals, horn_depth = photometric_stereo(images, lights, mask)

visualize(albedo, normals, horn_depth)
```





1(c) Display outputs using all four images (most accurate result) [6 points]

- The estimated albedo map.
- The estimated surface normals by showing both
 - Needle map, and
 - Three images showing components of surface normal.
- A wireframe of depth map.

Note: Feel free to use the same mask as used in the previous part.

!! Hint !!

You can refer to the code used in the cell above and adapt it to display the outputs for all the four images i.e im1, im2, im3 and im4 instead of just the 3 images i.e im1, im2 and im4 used before.

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

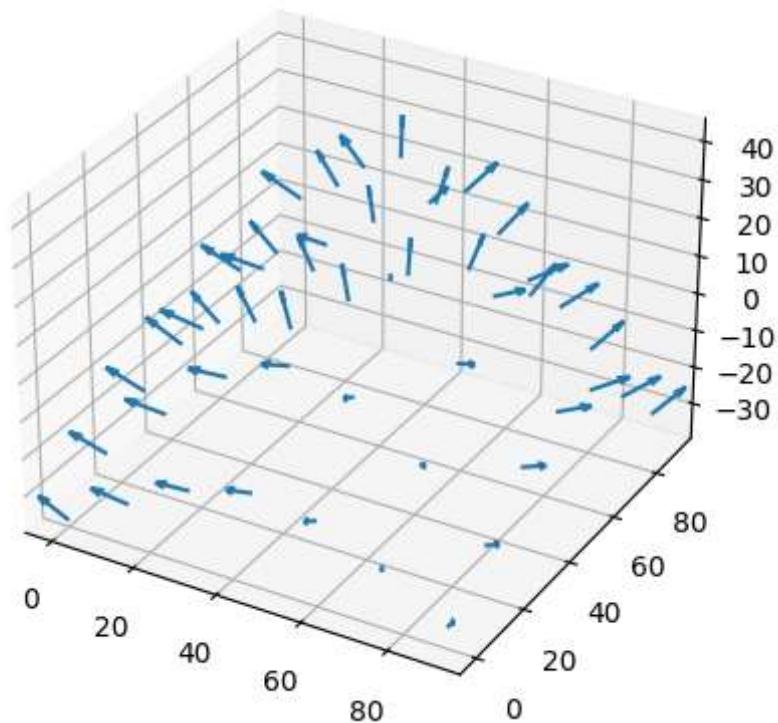
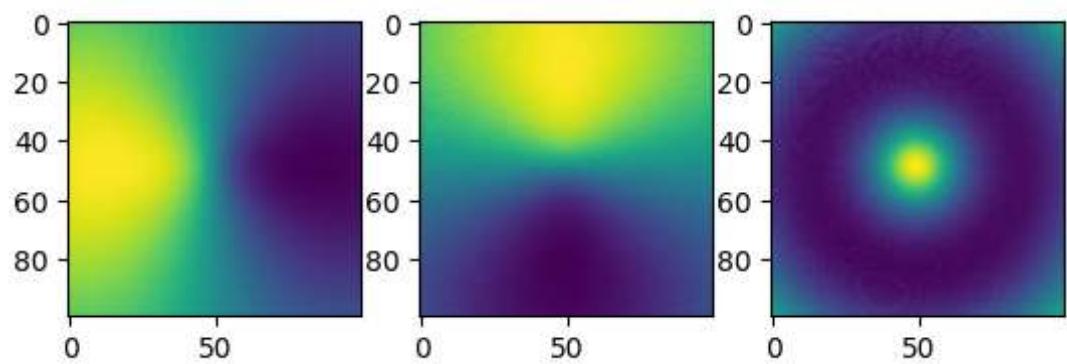
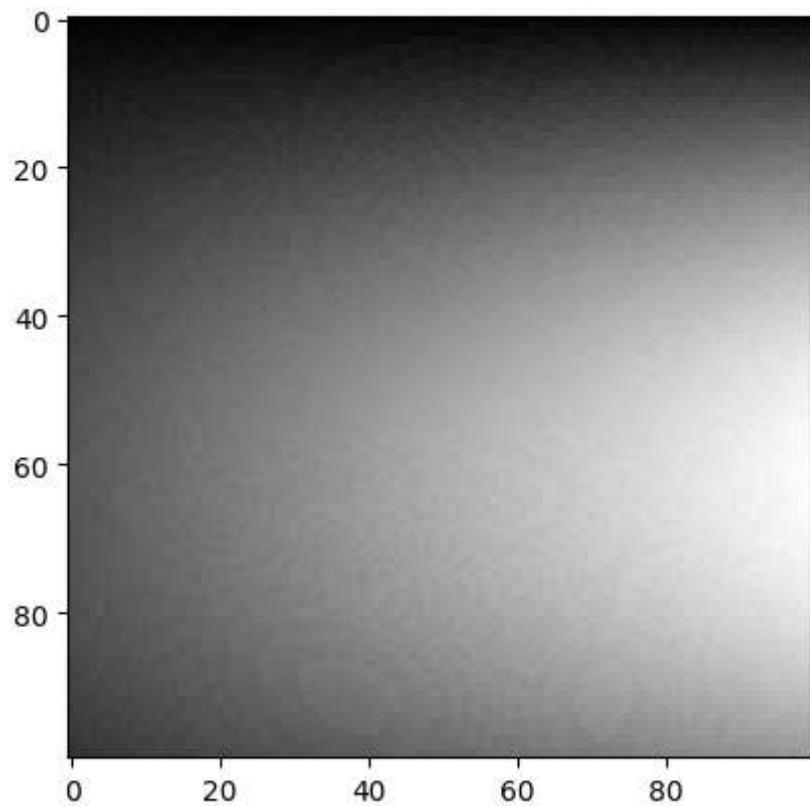
lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))

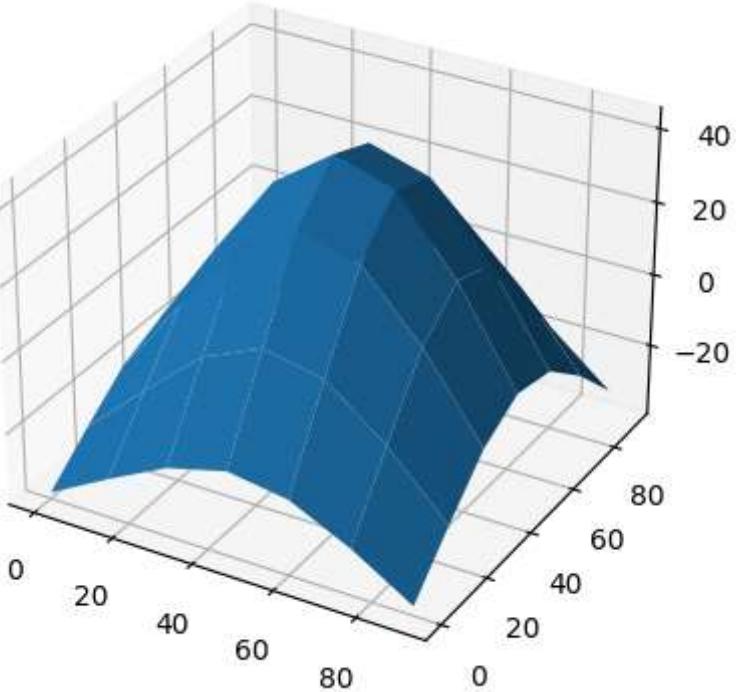
images = []
images.append(data["im1"])
images.append(data["im2"])
images.append(data["im3"])
images.append(data["im4"])
images = np.array(images)

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

albedo, normals, horn_depth = photometric_stereo(images, lights, mask)

visualize(albedo, normals, horn_depth)
```





Problem 2: Image Rendering [20 points]

In this exercise, we will render the image of a face with two different point light sources using a Lambertian reflectance model. We will use two albedo maps, one uniform and one that is more realistic. The face heightmap, the light sources, and the two albedo are given in `facedata.npy` for Python (each row of the `lightsource` variable encode a light location). The data from `facedata.npy` is already provided to you.

Note: Please make good use out of subplot to display related image next to eachother.

2(a) Plot the face in 2-D [2 pts]

Plot both albedo maps using `imshow`. Explain what you observe in brief.

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

# Load facedata.npy as ndarray
face_data = np.load("facedata.npy", encoding="latin1", allow_pickle=True)

# Load albedo matrix
albedo = face_data.item().get("albedo")

# Load uniform albedo matrix
uniform_albedo = face_data.item().get("uniform_albedo")

# Load heightmap
heightmap = face_data.item().get("heightmap")
```

```
# Load Light source
light_source = face_data.item().get("lightsource")
```

!! Hint !!

Use the following link to learn about subplots:

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

Make sure to use `cmap=gray` while showing the images in grayscale format otherwise you can expect to see green-yellow colors.

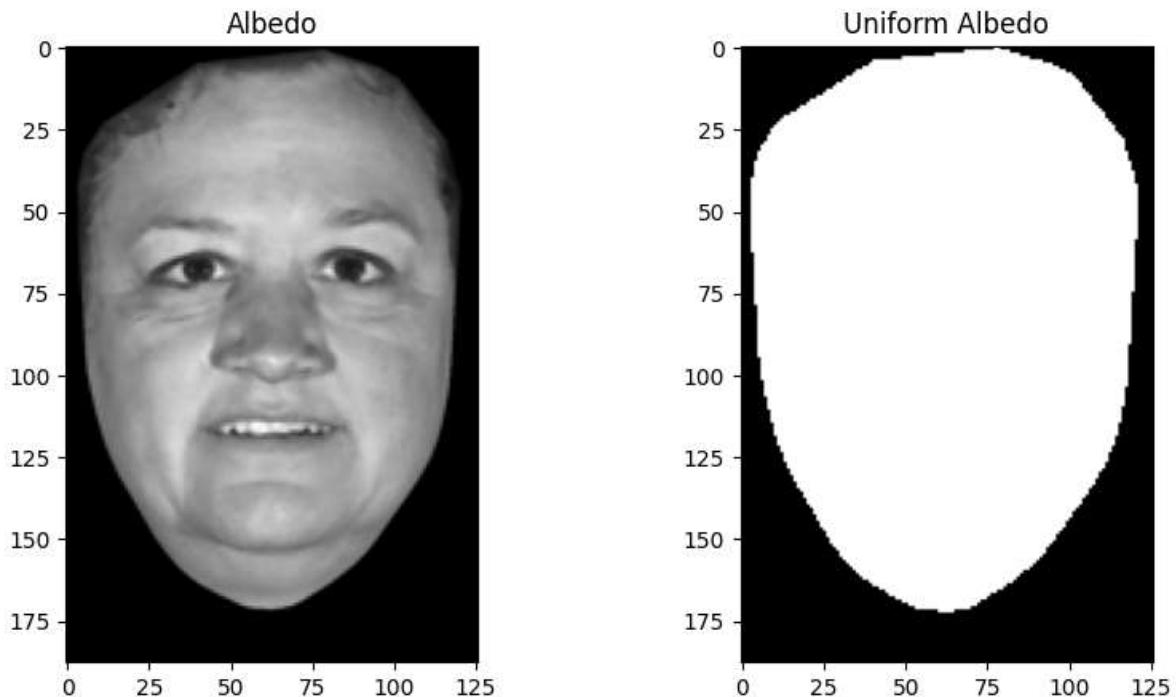
```
In [ ]: # Plot the face in 2-D (plot both albedo maps using imshow)
```

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].imshow(albedo, cmap="gray")
axes[0].set_title("Albedo")

axes[1].imshow(uniform_albedo, cmap="gray")
axes[1].set_title("Uniform Albedo")

plt.show()
```



Observations :

We can observe that Albedo 1 represents the entire face, whereas Albedo 2 (uniform albedo) does not capture the facial details. It only outlines the contours, resulting in a much more pixelated image. The facial contours are not as sharp and distinct

2(b) Plot the face in 3-D [2 pts]

Using both the heightmap and the albedo, plot the face using `plot_surface`. Do this for both albedos. Explain what you see.

Learn about creating mesh grids here:

<https://numpy.org/doc/stable/reference/generated/numpy.mgrid.html> alternatively use
<https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html>

Learn about plotting surfaces here:

[https://matplotlib.org/2.2.5/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.html?
highlight=plot_surface#mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface](https://matplotlib.org/2.2.5/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.html?highlight=plot_surface#mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface)

This example may help you to put things together:

https://matplotlib.org/2.0.2/examples/mplot3d/surface3d_demo.html

!! Hint !!

Make sure to set strides to one, cmap should be gray, linewidth is zero and antialiased is set to false. Figure out how to use albedo via the parameter facecolors -- you may want to look up `plt.cm.gray()` !!

```
In [ ]: # Plot the face in 3-D
# (Using the heatmap & albedo plot the faces using plot_surface)

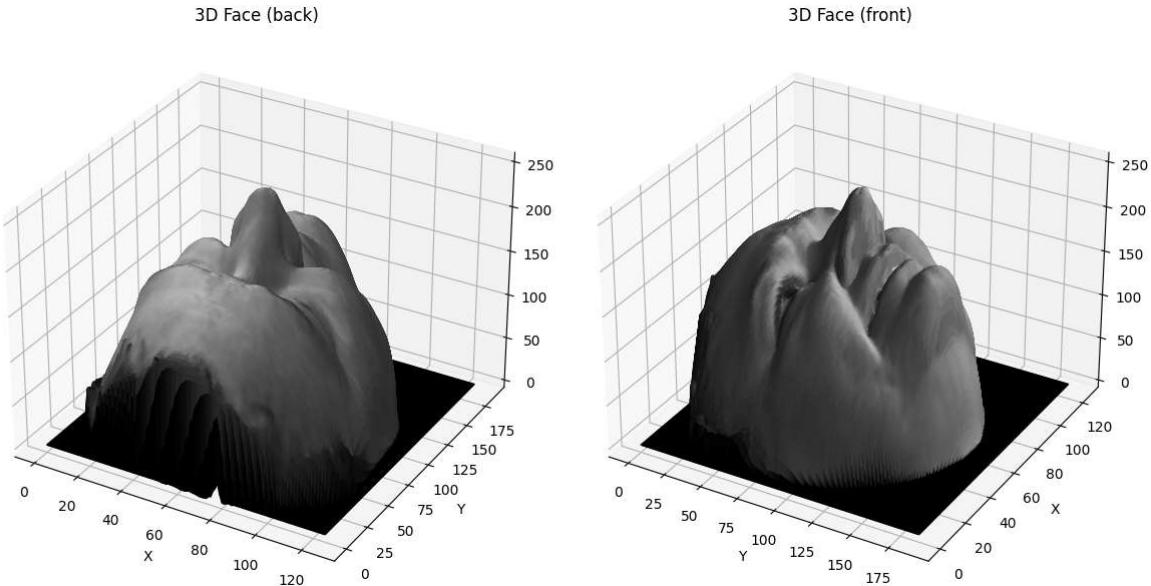
x = np.arange(heatmap.shape[1])
y = np.arange(heatmap.shape[0])
x, y = np.meshgrid(x, y)

fig = plt.figure(figsize=(12, 6))

ax1 = fig.add_subplot(121, projection="3d")
ax1.plot_surface(
    x,
    y,
    heatmap,
    facecolors=plt.cm.gray(albedo),
    rstride=1,
    cstride=1,
    antialiased=True,
)
ax1.set_title("3D Face (back)")
ax1.set_xlabel("X")
ax1.set_ylabel("Y")

ax2 = fig.add_subplot(122, projection="3d")
ax2.plot_surface(
    y,
    x,
    heatmap,
    facecolors=plt.cm.gray(albedo),
    rstride=1,
    cstride=1,
    antialiased=True,
)
ax2.set_title("3D Face (front)")
ax2.set_xlabel("Y")
ax2.set_ylabel("X")
```

```
plt.tight_layout()
plt.show()
```



2(c) Surface normals [8 pts]

Calculate the surface normals and display them as a quiver plot using quiver in matplotlib.pyplot in Python. Recall that the surface normals are given by

$$[-\frac{\delta f}{\delta x}, -\frac{\delta f}{\delta y}, 1]. \quad (3)$$

Also, recall, that each normal vector should be normalized to unit length.

!! Hint !!

You can use differencing to compute spatial derivatives (think about how to handle the boundaries?). You are also free to use convolutions (covered later in the class) which happens to be a much easier way of computing these derivatives. Learn about them here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html> Which mode should you use?

For the quiver plots, read up <https://matplotlib.org/stable/gallery/mplot3d/quiver3d.html> and make sure to use `length=10`.

```
In [ ]: # Compute and plot the surface normals
# (make sure that the normal vector is normalized to unit length)

normals = np.stack(
    (np.zeros(heightmap.shape), np.zeros(heightmap.shape), np.ones(heightmap.shape))
    axis=-1,
)

# This part of the code should compute the normals i.e fill in the above normals va
# execute the next cell to plot the surface normals

from scipy.signal import convolve2d

kernel_x = np.array([[0, 0, 0], [1, 0, -1], [0, 0, 0]]) * 0.5
```

```

kernel_y = np.array([[0, 1, 0], [0, 0, 0], [0, -1, 0]]) * 0.5

dx = convolve2d(heightmap, kernel_x, mode="same", boundary="symm")
dy = convolve2d(heightmap, kernel_y, mode="same", boundary="symm")

normals[:, :, 0] = -dx
normals[:, :, 1] = -dy

norms = np.linalg.norm(normals, axis=2)
normals[:, :, 0] /= norms
normals[:, :, 1] /= norms
normals[:, :, 2] /= norms

```

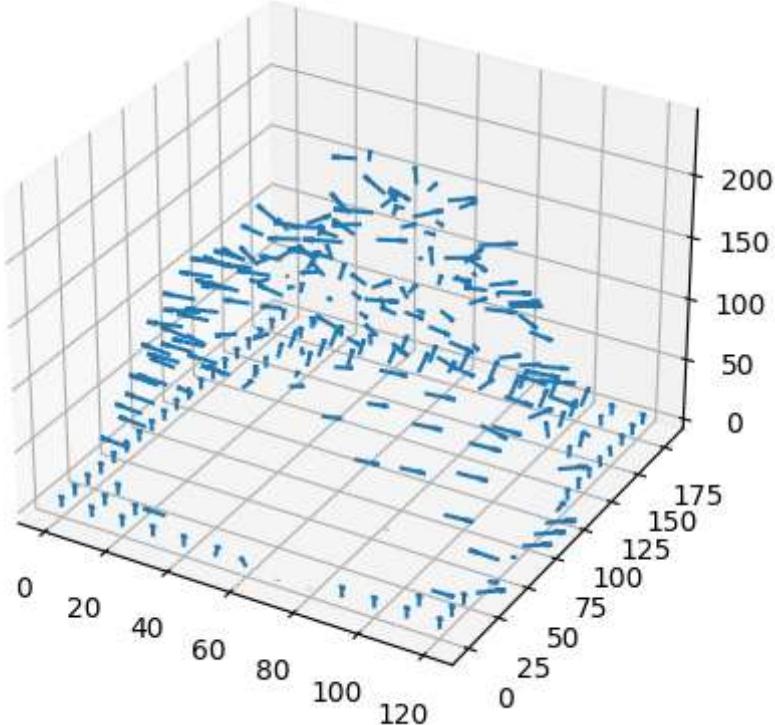
```

In [ ]: # This code is used to visualise the surface normals plot.
# Feel free to change the stride variable below. However DO NOT change the other pa

stride = 10
X, Y = np.meshgrid(
    np.arange(0, np.shape(normals)[0], stride),
    np.arange(0, np.shape(normals)[1], stride),
)

Z = heightmap[::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") doesn't work : TypeError: FigureBase.gca() got an u
ax = fig.add_subplot(projection="3d")
plt.quiver(Y, X, Z, NX, NY, NZ, length=10)
plt.show()

```



2(d) Render images [8 pts]

For each of the two albedos, render three images. One for each of the two light sources, and one for both light-sources combined. Display these in a 2×3 subplot figure with titles.

Recall that the general image formation equation is given by

$$I = a(x, y)\hat{\mathbf{n}}(x, y)^\top \hat{\mathbf{s}}(x, y)s_0 \quad (4)$$

where $a(x, y)$ is the albedo for pixel (x, y) , $\hat{\mathbf{n}}(x, y)$ is the corresponding surface normal, $\hat{\mathbf{s}}(x, y)$ the light source direction, s_0 the light source intensity. Let the light source intensity be 1 and make the 'distant light source assumption'. Use imshow with appropriate keyword arguments .

!! Hint !!

Make sure that the images computed by `lambertian()` are non-negative. Otherwise you will see gray image background (which should be black).

```
In [ ]: def lambertian(normals, light, albedo, intensity, mask):
    dot_product = np.sum(normals * light, axis=-1)
    dot_product = np.maximum(0, dot_product)

    reflected_intensity = albedo * dot_product * intensity * mask

    return reflected_intensity
```

!! Hint !!

When combining the two light sources, notice the linearity of the image formation equation and make sure that image values lie between 0 and 1.

```
In [ ]: # For each of the two albedos, render three images.
# One for each of the two light sources, and one for both light-sources combined.

light_source1 = light_source[0]
light_source2 = light_source[1]

intensity = 1

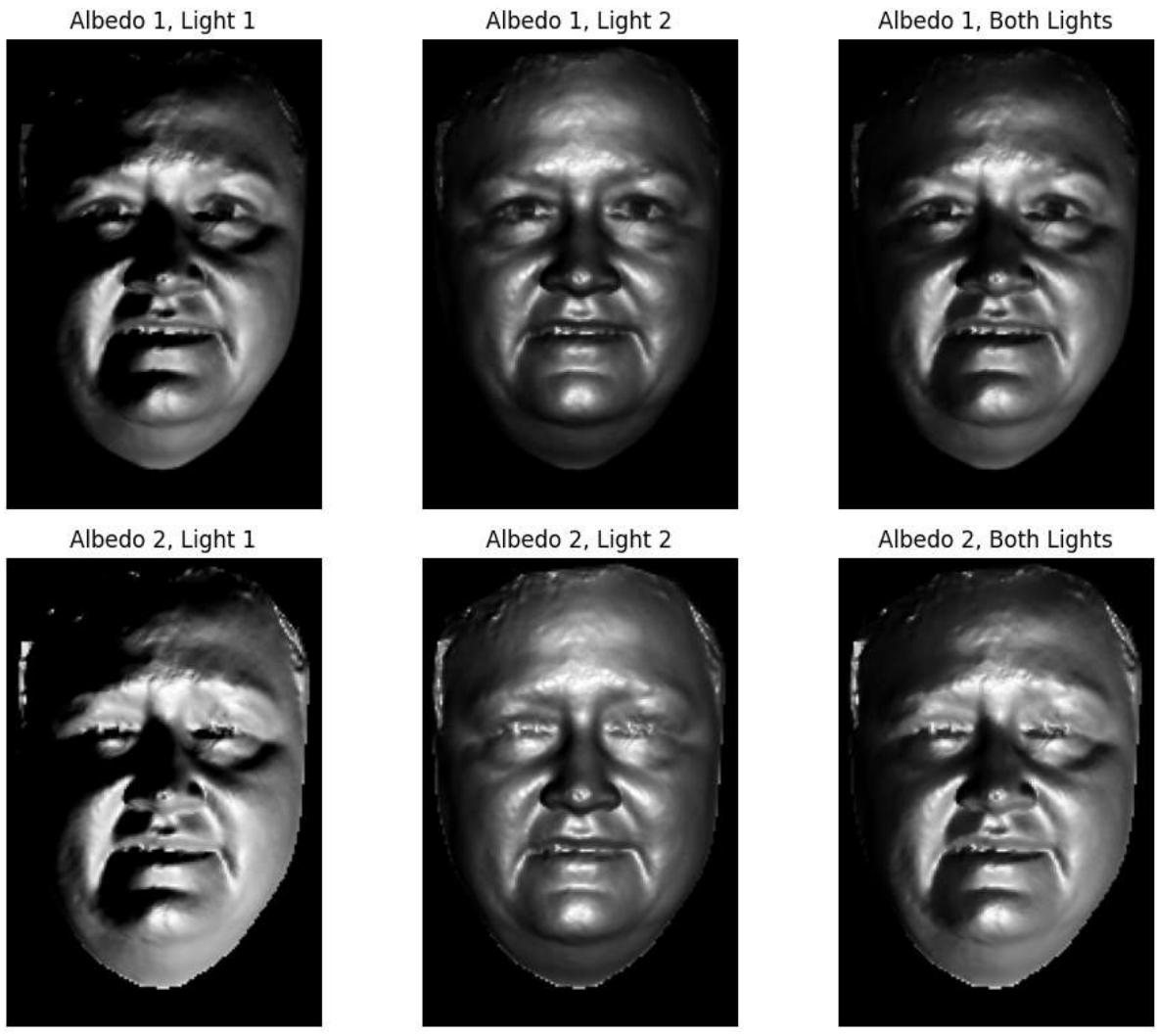
mask = np.ones(albedo.shape)

rendered_images_albedo1_light1 = lambertian(
    normals, light_source1, albedo, intensity, mask
)

rendered_images_albedo1_light2 = lambertian(
    normals, light_source2, albedo, intensity, mask
)
rendered_images_albedo1_both_lights = lambertian(
    normals, light_source1, albedo, intensity, mask
) + lambertian(normals, light_source2, albedo, intensity, mask)

rendered_images_albedo2_light1 = lambertian(
    normals, light_source1, uniform_albedo, intensity, mask
)
rendered_images_albedo2_light2 = lambertian(
    normals, light_source2, uniform_albedo, intensity, mask
```

```
)  
rendered_images_albedo2_both_lights = lambertian(  
    normals, light_source1, uniform_albedo, intensity, mask  
) + lambertian(normals, light_source2, uniform_albedo, intensity, mask)  
  
images = [  
    rendered_images_albedo1_light1,  
    rendered_images_albedo1_light2,  
    rendered_images_albedo1_both_lights,  
    rendered_images_albedo2_light1,  
    rendered_images_albedo2_light2,  
    rendered_images_albedo2_both_lights,  
]  
  
titles = [  
    "Albedo 1, Light 1",  
    "Albedo 1, Light 2",  
    "Albedo 1, Both Lights",  
    "Albedo 2, Light 1",  
    "Albedo 2, Light 2",  
    "Albedo 2, Both Lights",  
]  
  
plt.figure(figsize=(10, 8))  
  
for i in range(len(images)):  
    plt.subplot(2, 3, i + 1)  
    plt.imshow(images[i], cmap="gray")  
    plt.title(titles[i])  
    plt.axis("off")  
  
plt.tight_layout()  
plt.show()
```



Problem 3: Homogeneous Coordinates and Vanishing Points [20 points]

In class, we discussed the concept of homogeneous coordinates. In this example, we will confine ourselves to the real 2D plane. A point $(x, y)^\top$ on the real 2D plane can be represented in homogeneous coordinates by a 3-vector $(wx, wy, w)^\top$, where $w \neq 0$ is any real number. All values of $w \neq 0$ represent the same 2D point. Dividing out the third coordinate of a homogeneous point (x, y, z) converts it back to its 2D equivalent:

$$\left(\frac{x}{z}, \frac{y}{z} \right)^\top.$$

Consider a line in the 2D plane, whose equation is given by $ax + by + c = 0$. This can equivalently be written as $\mathbf{l}^\top \mathbf{x} = 0$, where $\mathbf{l} = (a, b, c)^\top$ and $\mathbf{x} = (x, y, 1)^\top$. Noticing that \mathbf{x} is a homogeneous representation of $(x, y)^\top$, we define \mathbf{l} as a homogeneous representation of the line $ax + by + c = 0$. Note that the line $(ka)x + (kb)y + (kc) = 0$ for $k \neq 0$ is the same as the line $ax + by + c = 0$, so the homogeneous representation of the line $ax + by + c = 0$ can be equivalently given by $(a, b, c)^\top$ or $(ka, kb, kc)^\top$ for any $k \neq 0$.

All points (x, y) that lie on the line $ax + by + c = 0$ satisfy the equation $\mathbf{l}^\top \mathbf{x} = 0$, thus, we can say that a condition for a homogeneous point \mathbf{x} to lie on the homogeneous line \mathbf{l} is that their dot product is zero, that is, $\mathbf{l}^\top \mathbf{x} = 0$. We note this down as a fact:

Fact 1: A point \mathbf{x} in homogeneous coordinates lies on the homogeneous line \mathbf{l} if and only if

$$\mathbf{x}^\top \mathbf{l} = \mathbf{l}^\top \mathbf{x} = 0$$

Now let us solve a few simple examples:

- **3(a)** Give at least two homogeneous representations for the point $(4, 5)^\top$ on the 2D plane, one with $w > 0$ and one with $w < 0$.

[1 Point]

Response 3(a)

the homogeneous representation for the point $(x, y)^\top$ is $(wx, wy, w)^\top$ with w not null

the homogeneous representation for the point $(4, 5)^\top$ is :

- $w > 0$: if $w = 1$: $(4, 5, 1)^\top$
- $w < 0$: if $w = -1$: $(-4, -5, -1)^\top$

- **3(b)** What is the equation of the line passing through the points $(1, 1)^\top$ and $(-1, 2)^\top$ [in the usual Cartesian coordinates]? Now write down a 3-vector that is a homogeneous representation for this line.

[2 Points]

Response 3(b)

equation : $x + 2y - 3 = 0$

the homogeneous representation of the line $ax + by + c = 0$ can be equivalently given by $(a, b, c)^\top$ or $(ka, kb, kc)^\top$ for any $k \neq 0$.

homogeneous representation : $(1, 2, -3)^\top$ or $(-1, -2, 3)^\top$

We will now move on to consider the intersection of two lines. We make the claim that: "The (homogeneous) point of intersection, \mathbf{x} , of two homogeneous lines \mathbf{l}_1 and \mathbf{l}_2 is $\mathbf{x} = \mathbf{l}_1 \times \mathbf{l}_2$, where \times stands for the vector (or cross) product".

- **3(c)** In plain English, how will you express the condition a point must satisfy to lie at the intersection of two lines? Armed with this simple condition, and using **Fact 1**, can you briefly explain why $\mathbf{l}_1 \times \mathbf{l}_2$ must lie at the intersection of lines \mathbf{l}_1 and \mathbf{l}_2 ?

[5 Points]

Response 3(c)

All points (x, y) that lie on the line $ax + by + c = 0$ satisfy the equation $\mathbf{l}^\top \mathbf{x} = 0$, thus, we can say that a condition for a homogeneous point \mathbf{x} to lie on the homogeneous line \mathbf{l} is that their dot product is zero, that is, $\mathbf{l}^\top \mathbf{x} = 0$.

To lie at the intersection of two lines a point must belong to both lines. That's why it should satisfy this two equations : $\mathbf{l}_1^\top \mathbf{x} = 0$ and $\mathbf{l}_2^\top \mathbf{x} = 0$

This means that the dot products of the row vectors \mathbf{l}_1^\top and \mathbf{l}_2^\top with \mathbf{x} are both zero. It means that \mathbf{x} is orthogonal to both. It means that \mathbf{x} in homogeneous coordinates lies in the plane defined by \mathbf{l}_1^\top and \mathbf{l}_2^\top . In other words, \mathbf{x} is the normal to the plane defined by \mathbf{l}_1^\top and \mathbf{l}_2^\top . That is why $\mathbf{x} = \mathbf{l}_1 \times \mathbf{l}_2$

In the following, we will use the above stated claim for the intersection of two lines.

- **3(d)** Consider the two lines $x + y + 5 = 0$ and $3x - 5y + 7 = 0$. Use the claim in question **3(c)** to find their intersection in homogeneous coordinates. Next, convert this homogeneous point back to standard Cartesian coordinates and report the 2D point of intersection.

[3 Points]

Response 3(d)

According to 3(c) $\mathbf{l}_1 \times \mathbf{l}_2$ must lie at the intersection of lines \mathbf{l}_1 and \mathbf{l}_2

Here : $\mathbf{l}_1 = (1, 1, 5)^\top$ and $\mathbf{l}_2 = (3, -5, 7)^\top$

$$\mathbf{l}_1 \times \mathbf{l}_2 = (32, 8, -8)^\top$$

The homogeneous representation of the intersection can be $(32, 8, -8)^\top$ or $(4, 1, -1)^\top$

Back to standard Cartesian coordinates the 2D point of intersection is : $(-4, -1)^\top$

- **3(e)** Consider the two lines $x + 2y + 1 = 0$ and $2x + 4y - 2 = 0$. What is the special relationship between these two lines in the Euclidean plane? What is their intersection in standard Cartesian coordinates?

[2 Points]

Response 3(e)

These two lines are **parallel** in the Euclidean plane, so they do not intersect in standard Cartesian coordinates.

```
In [ ]: line1 = (1, 2, 1)
line2 = (2, 4, -2)

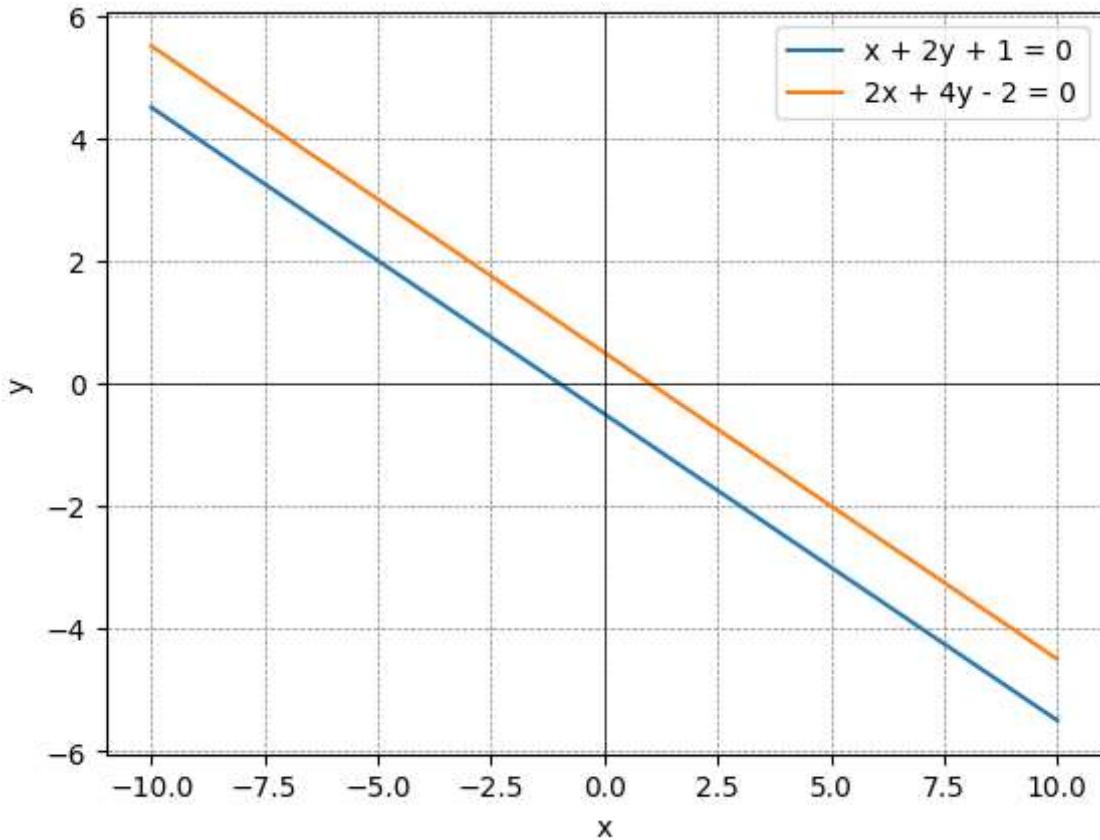
x = np.linspace(-10, 10, 400)

y1 = (-line1[0] * x - line1[2]) / line1[1]
y2 = (-line2[0] * x - line2[2]) / line2[1]

plt.plot(x, y1, label="x + 2y + 1 = 0")
plt.plot(x, y2, label="2x + 4y - 2 = 0")

plt.xlabel("x")
plt.ylabel("y")
plt.axhline(0, color="black", linewidth=0.5)
plt.axvline(0, color="black", linewidth=0.5)
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.legend()
```

```
plt.show()
```



- **3(f)** Write the homogeneous representations of the above two lines from part **3(e)** and compute their point of intersection in homogeneous coordinates. What is this point of intersection called in computer vision parlance?

[3 Points]

Response 3(f)

Here : $\mathbf{l}_1 = (1, 2, 1)^\top$ and $\mathbf{l}_2 = (2, 4, -2)^\top$

$$\mathbf{l}_1 \times \mathbf{l}_2 = (-8, 4, 0)^\top$$

The homogeneous representation of the intersection can be : $(-8, 4, 0)^\top$ or $(-2, 1, 0)^\top$

This point of intersection is a point at "infinity" called an "**ideal point**"

- **3(g)** Do questions **3(e)** and **3(f)** justify the claim in class, that homogeneous coordinates provide a uniform treatment of line intersection, regardless of parallelism? Briefly explain.

[2 Points]

Response 3(g)

The ideal point provides a unified representation of the intersection, even though the lines are parallel in standard Cartesian coordinates because homogeneous coordinates offer a

means of representing both points and lines using an extended coordinate system. They allow for an intuitive treatment of points at infinity. When $z=0$, it signifies that a point is located at infinity along a particular direction. Consequently, angles are no longer preserved, distant objects appear smaller, and parallel lines converge. This means that all lines can be treated uniformly, regardless of whether they are parallel or not.

One key advantage of homogeneous coordinates is that they handle both points at infinity and finite points in a consistent manner. This is crucial when dealing with parallel lines. In Euclidean geometry, parallel lines do not intersect, which can lead to complications when working with them. However, in projective geometry, parallel lines can be thought of as intersecting at a point at infinity.

- **3(h)** Give (with justification) an expression for the homogeneous representation of the line passing through two homogeneous points \mathbf{x}_1 and \mathbf{x}_2 . [Hint: Construct an argument analogous to the one for the intersection of two lines in part **3(c)**.]

[2 Points]

Response 3(h)

The homogeneous representation of the line passing through two homogeneous points \mathbf{x}_1 and \mathbf{x}_2 imply that the two point must both lie to the line \mathbf{l} . That's why it should satisfy this two equations : $\mathbf{l}^\top \mathbf{x}_1 = 0$ and $\mathbf{l}^\top \mathbf{x}_2 = 0$

This means that the dot products of the row vectors \mathbf{x}_1^\top and \mathbf{x}_2^\top with \mathbf{l} are both zero. It means that \mathbf{l} is orthogonal to both. It means that \mathbf{l} in homogeneous coordinates lies in the plane defined by \mathbf{x}_1^\top and \mathbf{x}_2^\top . In other words, \mathbf{l} is the normal to the plane defined by \mathbf{x}_1^\top and \mathbf{x}_2^\top . That is why $\mathbf{l} = \mathbf{x}_1 \times \mathbf{x}_2$

Problem 4: Camera Matrices and Rigid-Body Transformations [20 points]

Consider a world coordinate system W , centered at the origin $(0, 0, 0)$, with axes given by unit vectors $\hat{\mathbf{i}} = (1, 0, 0)^\top$, $\hat{\mathbf{j}} = (0, 1, 0)^\top$ and $\hat{\mathbf{k}} = (0, 0, 1)^\top$. We use a notation where boldfaces stand for a vector and a hat above a boldface letter stands for a unit vector.

- **4(a)** Consider another coordinate system, with unit vectors along two of the orthogonal axes as:
 $\hat{\mathbf{i}}' = (0.688, -0.688, 0.229)^\top$ and $\hat{\mathbf{j}}' = (0.267, 0.534, 0.801)^\top$. Find the unit vector, $\hat{\mathbf{k}}'$, along the third axis orthogonal to both $\hat{\mathbf{i}}'$ and $\hat{\mathbf{j}}'$. Is there a unique such unit vector? If not, choose the one that makes an acute angle with $\hat{\mathbf{k}}$.

[2 Points]

```
In [ ]: i = np.array([0.688, -0.688, 0.229])
j = np.array([0.267, 0.534, 0.801])

k = np.cross(i, j)
```

```
print(np.round(k, 3))
```

```
[-0.673 -0.490  0.551]
```

Response 4(a)

To find the unit vector $\hat{\mathbf{k}}'$ we have to compute the cross product between $\hat{\mathbf{i}}'$ and $\hat{\mathbf{j}}' : \hat{\mathbf{i}}' \times \hat{\mathbf{j}}'$

The result is approximatively : $\hat{\mathbf{k}}' = (-0.673, -0.490, 0.551)^\top$

There is **not** a unique such unit vector : $\hat{\mathbf{k}} = (0.673, 0.490, -0.551)^\top$ is also a possibility

But the one that makes an acute angle with $\hat{\mathbf{k}}$ is : $\hat{\mathbf{k}}' = (-0.673, -0.490, 0.551)^\top$

- **4(b)** Find the rotation matrix that rotates any vector in the $(\hat{\mathbf{i}}, \hat{\mathbf{j}}, \hat{\mathbf{k}})$ coordinate system to the $(\hat{\mathbf{i}}', \hat{\mathbf{j}}', \hat{\mathbf{k}}')$ coordinate system.

[2 Points]

!! Hint !!

The rotation matrix is given by $R =$

$$\begin{bmatrix} u^T \\ v^T \\ w^T \end{bmatrix}$$

Response 4(b)

The rotation matrix is:

$$\begin{bmatrix} \hat{\mathbf{i}}'^T \\ \hat{\mathbf{j}}'^T \\ \hat{\mathbf{k}}'^T \end{bmatrix} = \begin{bmatrix} 0.688 & -0.688 & 0.229 \\ 0.267 & 0.534 & 0.801 \\ -0.673 & -0.490 & 0.551 \end{bmatrix}$$

- **4(c)** What is the *extrinsic parameter matrix* for a camera at a displacement $(-1, -2, -5)^\top$ from the origin of W and oriented such that its principal axis coincides with $\hat{\mathbf{k}}'$, the x-axis of its image plane coincides with $\hat{\mathbf{i}}'$ and the y-axis of the image plane coincides with $\hat{\mathbf{j}}'$?

[3 Points]

!! Hint !!

The extrinsic parameter matrix is given by:

$$\begin{pmatrix} R & -RC \\ 0^T & 1 \end{pmatrix}$$

where R is the rotation matrix and C is the camera center.

```
In [ ]: C = np.array([-1, -2, -5])
R = np.array([i, j, k])

RC = R @ C

extrinsic_matrix = np.column_stack((R, -RC))

extrinsic_matrix = np.vstack((extrinsic_matrix, [0, 0, 0, 1]))

print(np.round(extrinsic_matrix, 3))

[[ 0.688 -0.688  0.229  0.457]
 [ 0.267  0.534  0.801  5.34 ]
 [-0.673 -0.490  0.551  1.102]
 [ 0.       0.       0.       1.     ]]
```

Response 4(c)

The extrinsic parameter matrix is:

$$\begin{pmatrix} R & -RC \\ 0^T & 1 \end{pmatrix} = \begin{bmatrix} 0.688 & -0.688 & 0.229 & 0.457 \\ 0.267 & 0.534 & 0.801 & 5.340 \\ -0.673 & -0.490 & 0.551 & 1.102 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **4(d)** What is the *intrinsic parameter matrix* for this camera, if its focal length in the x-direction is 1024 pixels, aspect ratio is 1, pixels deviate from rectangular by 0 degrees and principal point is offset from the center $(0,0)^\top$ of the image plane to the location $(10, -5)^\top$?

[3 Points]

Reponse 4(d)

- Focal length in the x-direction (dx) = 1024 pixels

Peut etre négatif

- Aspect ratio (α) = 1
- $dy = dx / \alpha \Rightarrow dy = 1024$ pixels
- Pixels deviate from rectangular by 0 degrees ($\theta = 0$)
- Principal point offset (cx, cy) = $(10, -5)$

The *intrinsic parameter matrix* for this camera is :

$$K = \begin{bmatrix} -1024 & 0 & 10 \\ 0 & -1024 & -5 \\ 0 & 0 & 1 \end{bmatrix}$$

- **4(e)** Write down the projection matrix for the camera described by the configuration in parts **4(c)** and **4(d)**.

[3 Points]

!! Hint !!

The projection matrix is given by $P = K [R \mid -RC]$ where K is the intrinsic matrix, R is the rotation matrix and C is the camera center.

```
In [ ]: K = np.array([[-1024, 0, 10], [0, -1024, -5], [0, 0, 1]])

concat = np.column_stack((R, -RC))

P = K @ concat

for k in range(3):
    P[k] = np.round(P[k], 3)
    print(P[k])

[-711.246  699.613 -228.985 -456.946]
[-270.041 -544.366 -822.979 -5473.671]
[-0.673 -0.49   0.551   1.102]
```

Response 4(e)

The *projection matrix* for this camera is :

$$P = \begin{bmatrix} -711.246 & 699.613 & 228.985 & -456.946 \\ -270.041 & -544.366 & -822.979 & -5473.671 \\ -0.673 & -0.49 & 0.551 & 1.102 \end{bmatrix}$$

- **4(f)** Consider a plane, orthogonal to $\hat{\mathbf{k}}$, at a displacement of 2 units from the origin of W along the $\hat{\mathbf{k}}$ direction. Consider a circle with radius 1, centered at $(0, 0, 2)^\top$ in the coordinate system W . We wish to find the image of this circle, as seen by the camera we constructed in part **4(e)**. The following questions need programming (use Python) and the code for each part should be turned in along with any figures and answers to specific questions. Explain your variable names (with comments). Feel free to supply any additional description or explanation to go with your code.

1. Compute 10000 well-distributed points on the surface of unit circle. One way to do this is to sample the angular range 0 to 360 degrees into 10000 equal parts and convert the resulting points from polar coordinates (radius is 1) to Cartesian coordinates. Display the circle, make sure that the axes of the display figure are equal.

[2 Points]

!! Hint !!

The following code snippet generates one such random point:

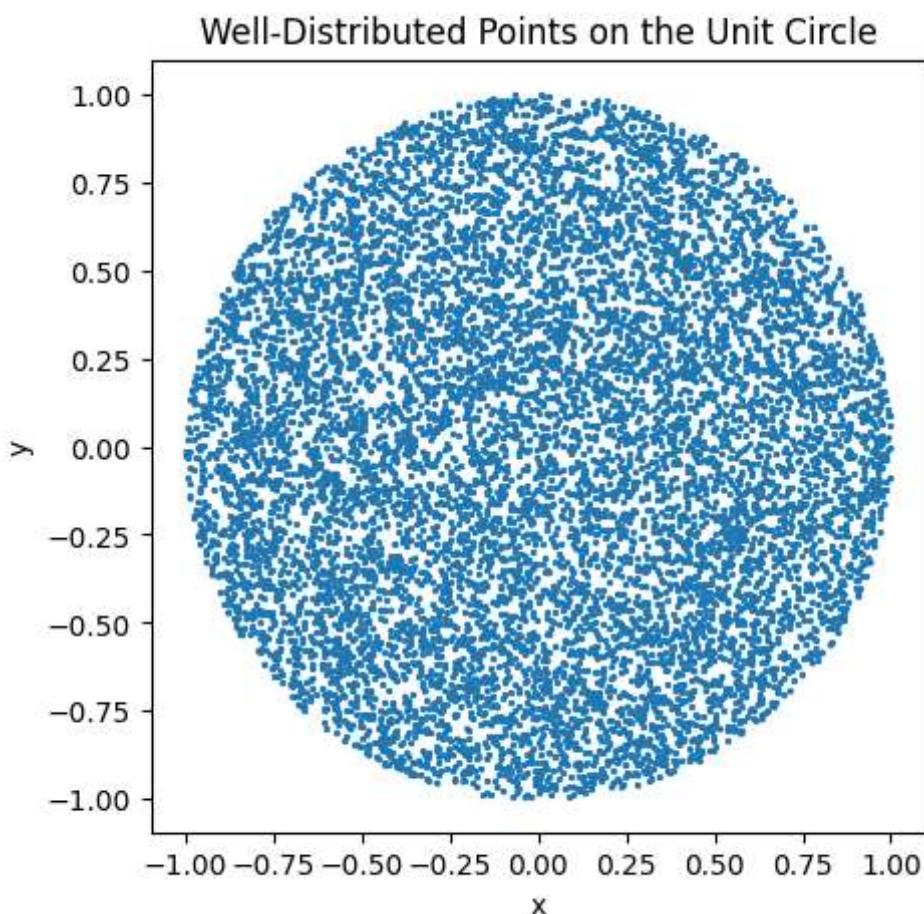
```
r = R * sqrt(random())
theta = random() * 2 * PI`
```

```
In [ ]: num_points = 10000

theta = np.linspace(0, 2 * np.pi, num_points)
r = np.sqrt(np.random.uniform(0, 1, num_points))

x_circle = r * np.cos(theta)
y_circle = r * np.sin(theta)

plt.figure(figsize=(5, 5))
plt.scatter(x_circle, y_circle, s=2)
plt.title("Well-Distributed Points on the Unit Circle")
plt.xlabel("x")
plt.ylabel("y")
plt.gca().set_aspect("equal", adjustable="box")
plt.show()
```



2. Add the z coordinate to these points, which is 2 for all of them. Make all the points homogeneous by adding a fourth coordinate equal to 1.

[1 Point]

```
In [ ]: z_circle = np.full(num_points, 2)
homogeneous_coordinate = np.full(num_points, 1)

points_3d = np.column_stack((x_circle, y_circle, z_circle, homogeneous_coordinate))

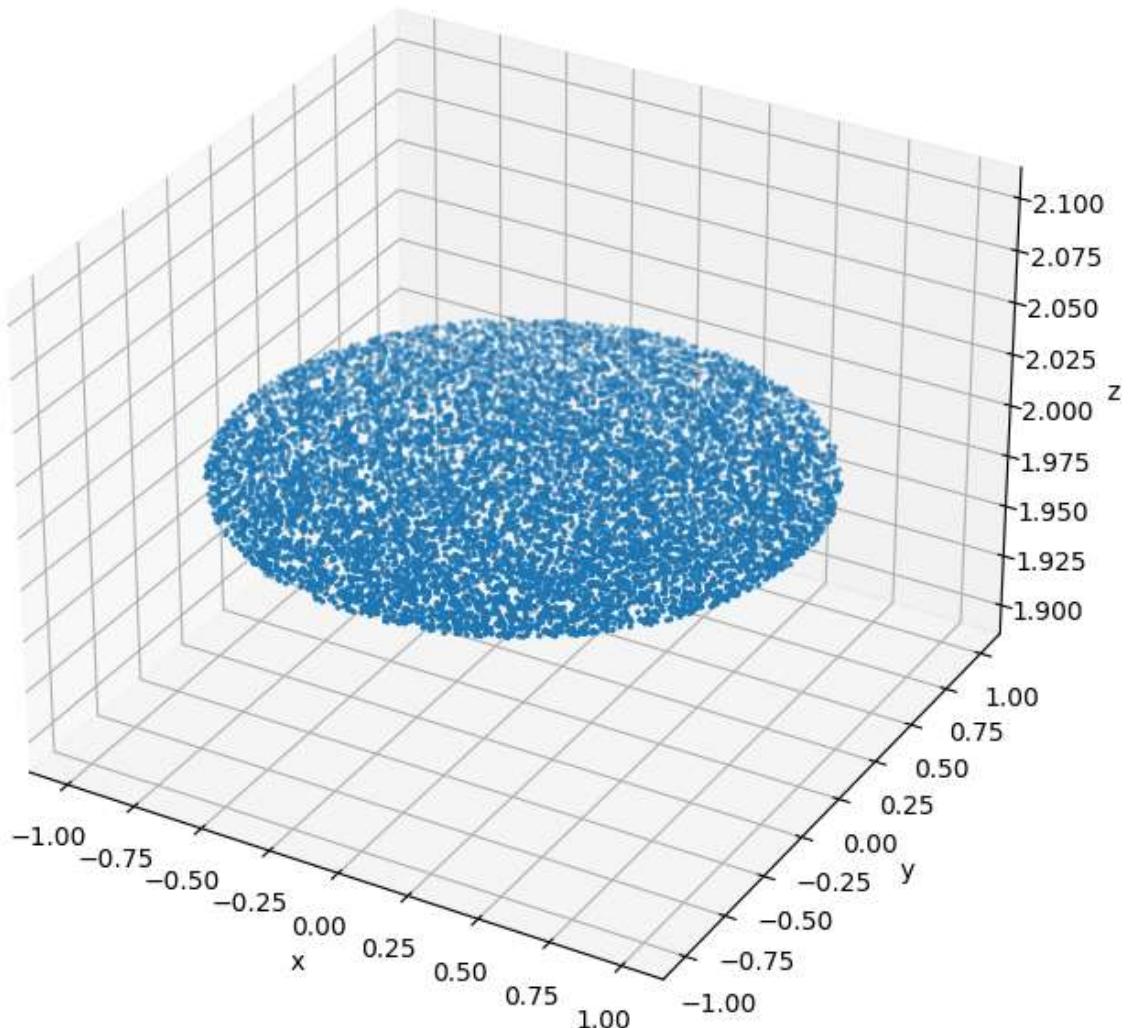
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
```

```

ax.scatter(points_3d[:, 0], points_3d[:, 1], points_3d[:, 2], s=2)
ax.set_title("Well-Distributed Points in 3D")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")
plt.show()

```

Well-Distributed Points in 3D



3. Compute the projection of these homogeneous points using the camera matrix from part **4(e)**. Convert the homogeneous projected points to 2D Cartesian points by dividing out (and subsequently discarding) the third coordinate of each point.

[2 Points]

```

In [ ]: projected_points_homogeneous = np.dot(P, points_3d.T).T

projected_points_2d = (
    projected_points_homogeneous[:, :2] / projected_points_homogeneous[:, 2:]
)

```

4. Plot the projected 2D points, again ensure that the axes of your plot are equal.

What is the shape of the image of a circle?

[2 Points]

Response

The shape of the image of a circle is an ellipse.

```
In [ ]: plt.figure(figsize=(5, 5))
plt.scatter(projected_points_2d[:, 0], projected_points_2d[:, 1], s=2)
plt.title("Projected 2D Points")
plt.xlabel("x")
plt.ylabel("y")
plt.gca().set_aspect("equal", adjustable="box")
plt.show()
```

