HW1_complete

October 18, 2022

1 CSE 152A Fall 2022 – Assignment 1

- Assignment Published On: **Wed, Oct 05, 2022**
- Due On: **Fri, Oct 14, 2022 11:59 PM (Pacific Time)**

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 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.

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (that is, 30% per day).

1.2 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.

1.2.1 Data

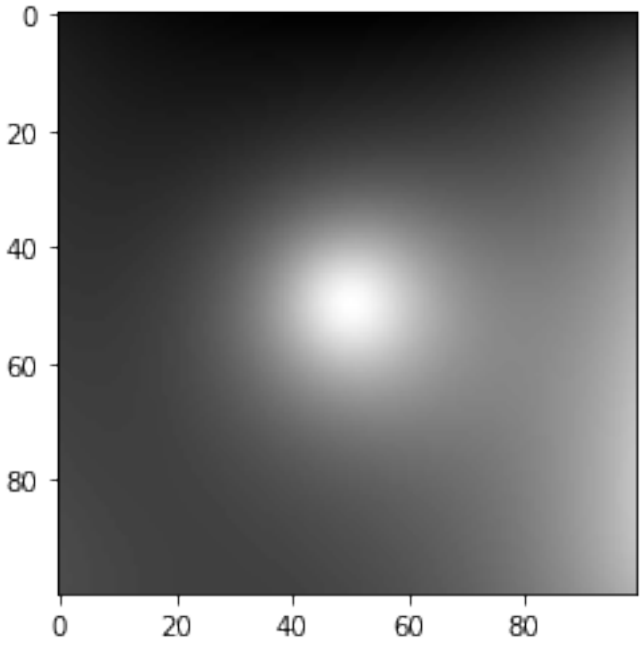
You will use synthetic images as data. These images are stored in `.pickle` files which were graciously provided by Satya Mallick. Each `.pickle` file contains

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

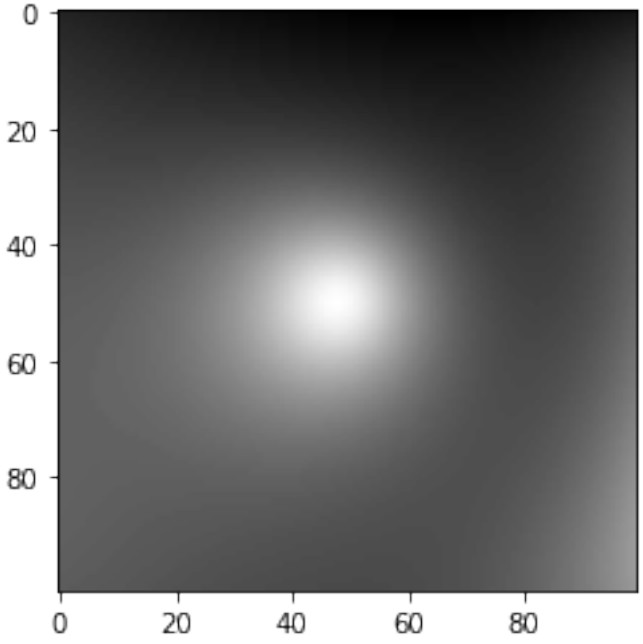
You will find all the data for this part in `synthetic_data.pickle`.

[2]:
```
## Example: How to read and access data from a pickle

import pickle
import matplotlib.pyplot as plt

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"]))
```

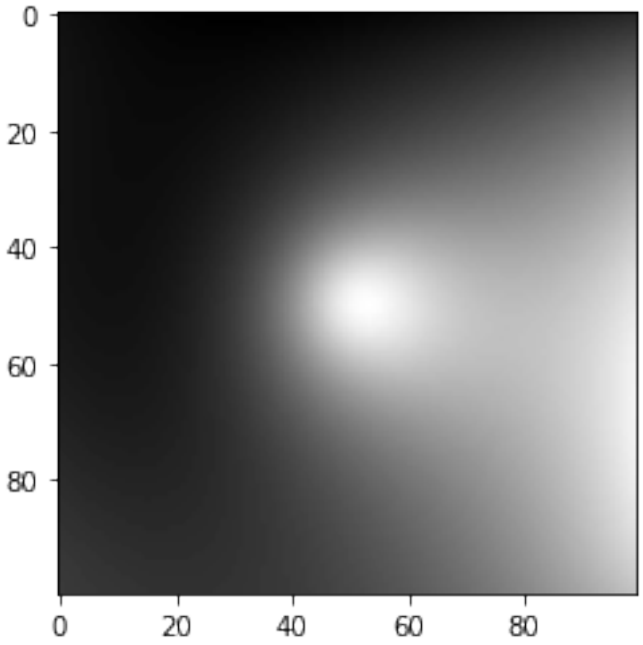
```
Keys: dict_keys(['__version__', 'l4', '__header__', 'im1', 'im3', 'im2', 'l2',
'im4', 'l1', '__globals__', 'l3'])
Image:
```



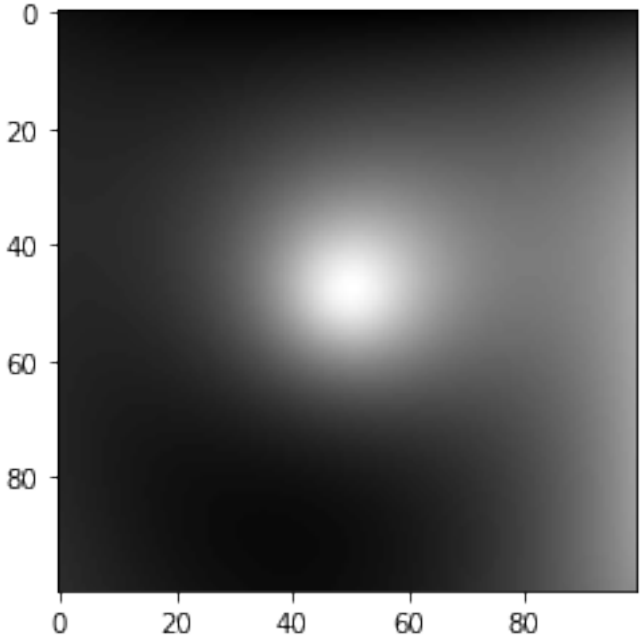

```
Light source direction: [[0 0 1]]
```




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




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




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

1.2.2 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.

[3]:
```
import numpy as np
from scipy.signal import convolve
from numpy import linalg

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
```

[4]:
```
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)

    gx = np.zeros(images[0].shape)
    gy = np.zeros(images[0].shape)

    #initializes surface normals:
    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)

    # lights_t = lights.T
    # square = np.matmul(lights_t,lights)
    # sqr_inv = linalg.inv(square)
    # pseudo_semi = np.matmul(sqr_inv,lights_t)

    pseudo_semi = np.linalg.pinv(lights)

    albedo_xy = 0
    surfn_xy = 0
```

```
    #print(images.shape) #3x100x100
    #print(normals.shape) #100x100x3

    #for each pixel:
    for x in range(0,len(images[0])):
        for y in range(0, len(images[0][0])):
            e_xy = []
            #stack up 3 observations for a pixel:
            for i in range(0,len(images)):
                e_xy.append(images[i][x][y])
            #solve linear system for the unknowns(of that pixel):
            b_xy = np.matmul(pseudo_semi,e_xy)

            albedo_xy = linalg.norm(b_xy, ord=2)
            #albedo_xy = np.absolute(b_xy)
            surfn_xy = b_xy/albedo_xy

            #assign final stuff:
            albedo[x][y] = albedo_xy
            normals[x][y][0] = surfn_xy[0]
            normals[x][y][1] = surfn_xy[1]
            normals[x][y][2] = surfn_xy[2]

            #gx = -n_hat1/n_hat3, gy = -n_hat2/n_hat3
            gx[x][y] = (surfn_xy[0]/surfn_xy[2])
            gy[x][y] = (surfn_xy[1]/surfn_xy[2])


    H_horn = horn_integrate(gx, gy, mask, horn_niter)

    return albedo, normals, H_horn
```

1.2.3 1(b) Display outputs using `im1`, `im2` and `im4` [6 Points]

The estimated albedo map.

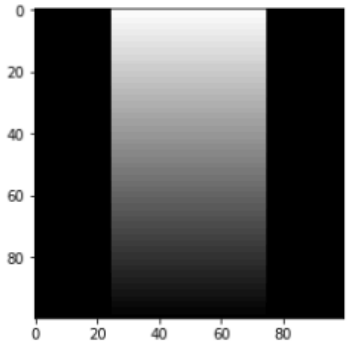
The estimated surface normals by showing both

Needle map, and

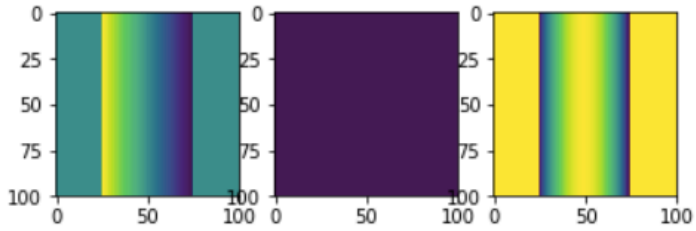
Three images showing components of surface normal.

A wireframe of depth map.

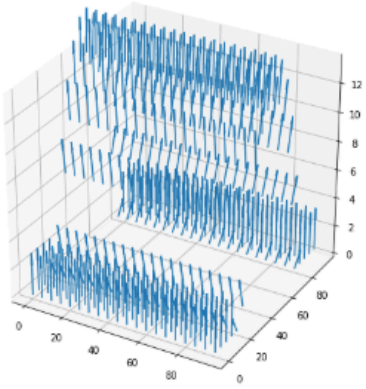
An example of outputs is shown in the figure below.



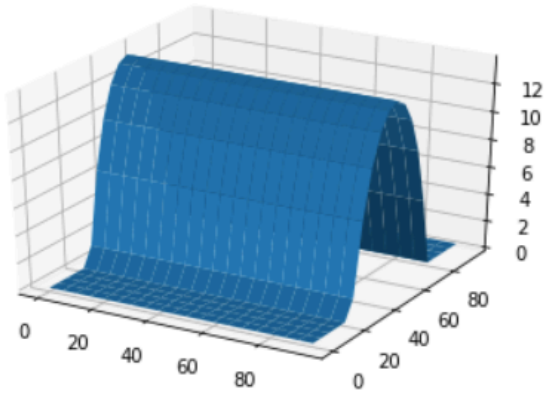
Albedo map



Normals as three separate channels



Needle map



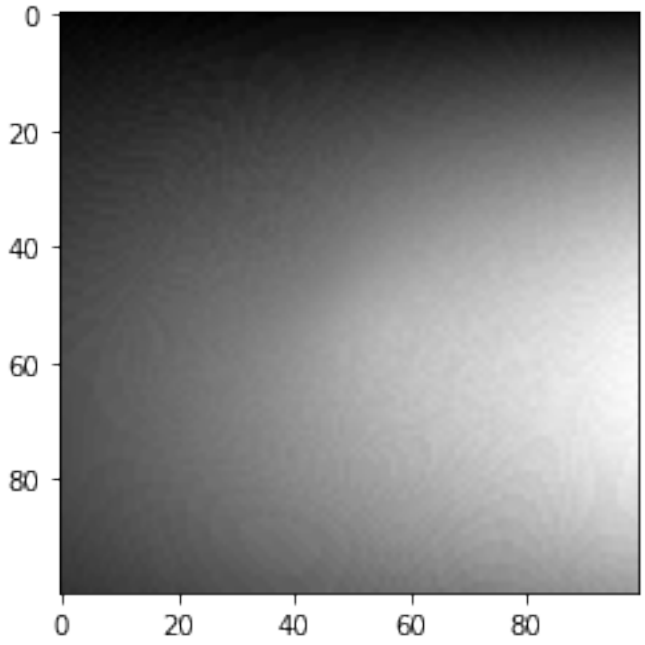
Wireframe of depth map

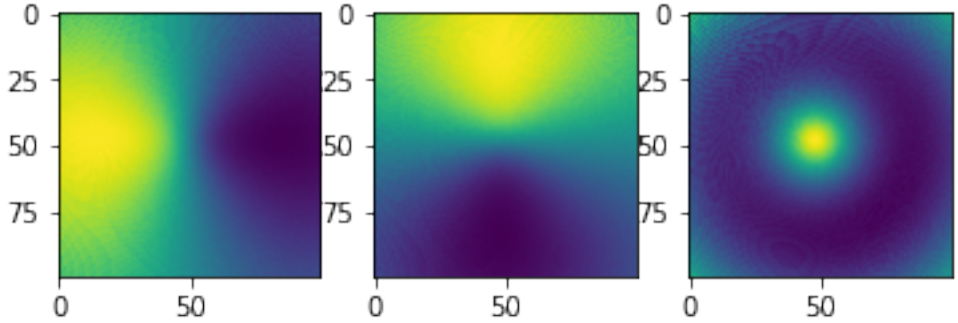
[5]:
```
from mpl_toolkits.mplot3d import Axes3D

# ------------------------------------------------------------------------
# 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.gca(projection='3d')
    plt.quiver(X,Y,Z,NX,NY,NZ, length=10)
    plt.show()

    # plotting wireframe depth map

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

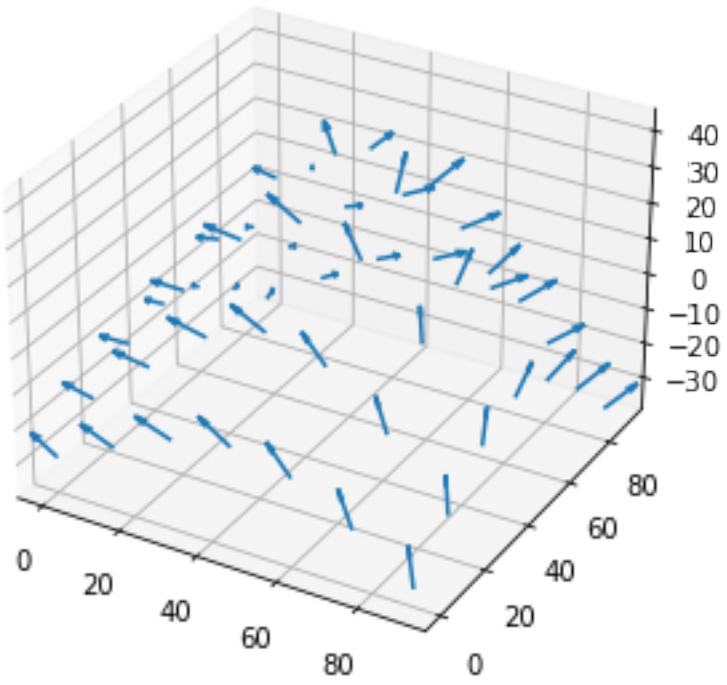
[6]:
```
pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

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)
```

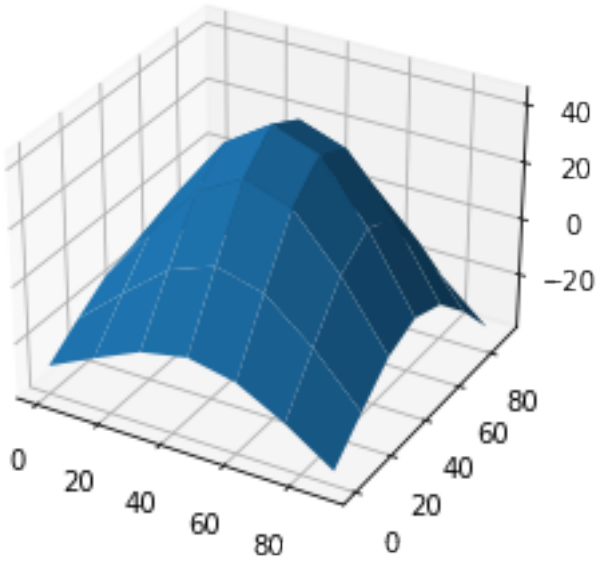





```
/tmp/ipykernel_162/3265356501.py:42: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```




```
/tmp/ipykernel_162/3265356501.py:50: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



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

The estimated albedo map.

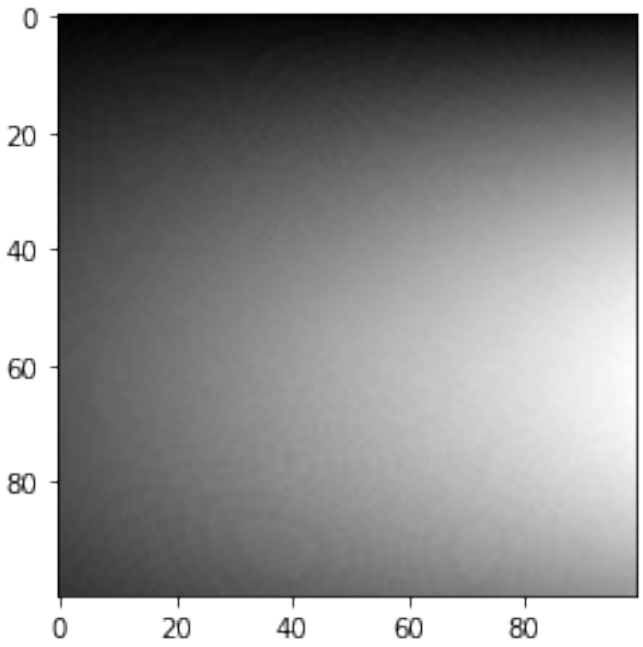
The estimated surface normals by showing both

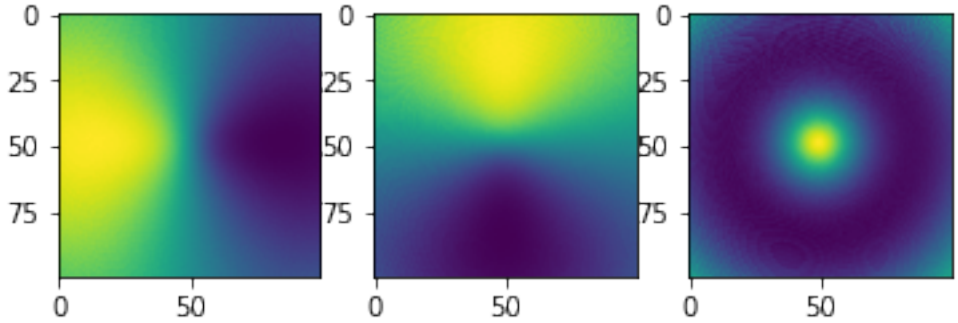
Needle map, and

Three images showing components of surface normal.

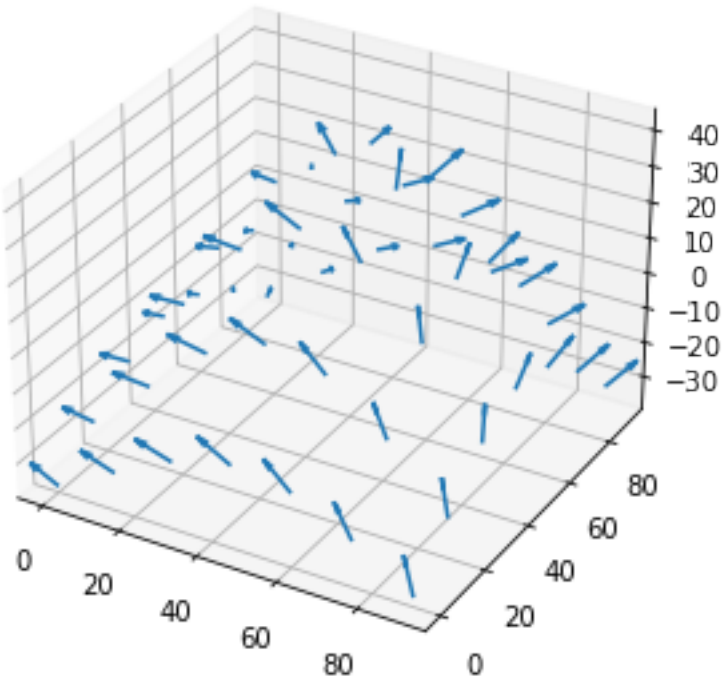
A wireframe of depth map.

[7]:
```
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)
```

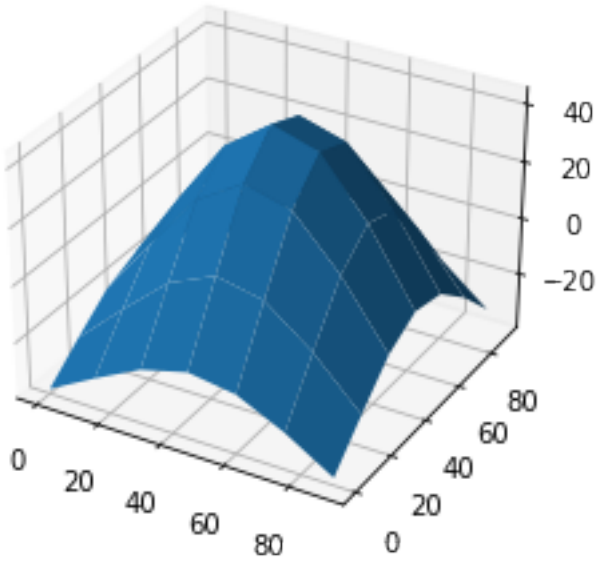





```
/tmp/ipykernel_162/3265356501.py:42: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```




```
/tmp/ipykernel_162/3265356501.py:50: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



1.3 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.

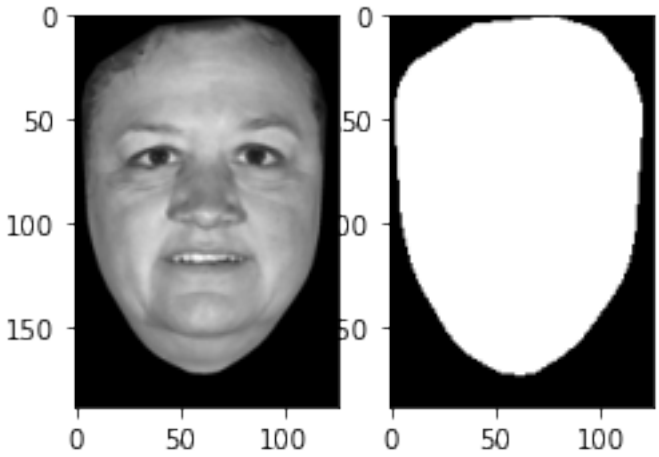
2(a) Plot the face in 2-D [2 pts]

Plot both albedo maps using imshow. Explain what you see.

[8]:
```
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')
```

[9]:
```
# Plot the face in 2-D (plot both albedo maps using imshow)
fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.imshow(albedo, cmap="gray")
ax2 = fig.add_subplot(132)
ax2.imshow(uniform_albedo, cmap="gray")
plt.show()
```



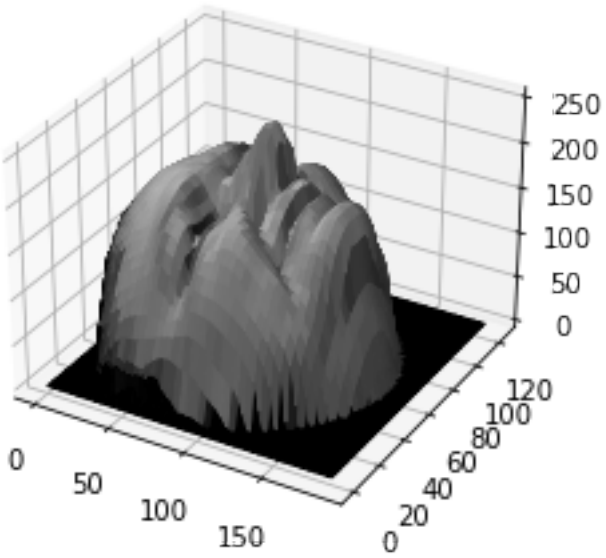
2(b) Plot the face in 3-D [2 pts]

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

[10]:
```
# Plot the face in 3-D
# (Using the heightmap & albedo plot the faces using plot_surface)

figure = plt.figure()

stride = 10

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

X = np.zeros((heightmap.shape[0],heightmap.shape[1]))
```

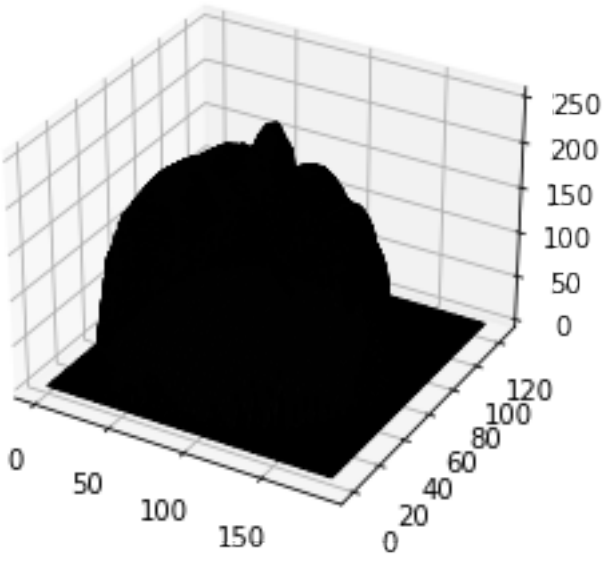
```
Y = np.zeros((heightmap.shape[0],heightmap.shape[1]))
Z = heightmap#[::stride,::stride].T
ax = figure.add_subplot(projection='3d')

for x in range(0,heightmap.shape[0]):
    for y in range(0, heightmap.shape[1]):
        X[x][y]=x
        Y[x][y]=y


#print(X.shape)
#print(Y.shape)
#print(albedo.T.shape)
#print(heightmap.T.shape)

#print(heightmap.shape)

ax.plot_surface(X.T,Y.T,Z.T,facecolors=plt.cm.gray(albedo.T))
```

[10]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f61ceceaa30>



[11]:
```
figure = plt.figure()
ax = figure.add_subplot(projection='3d')

ax.plot_surface(X.T,Y.T,Z.T,facecolors=plt.cm.gray(uniform_albedo.T))
```

[11]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f61cec85190>



What do I see: When the albedo is uniform we have a black face shape since this implies that no light is being absorbed at varying degrees and therefore every feature in the face has a uniform color. Since this is not the case with the regular albedo, we have a face shape with varying degrees of light absorption at different points which is why we can make out features in the non-uniform albedo image.

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

$$[-\frac{\delta f}{\delta x}, -\frac{\delta f}{\delta y}, 1]. \tag{1}$$

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

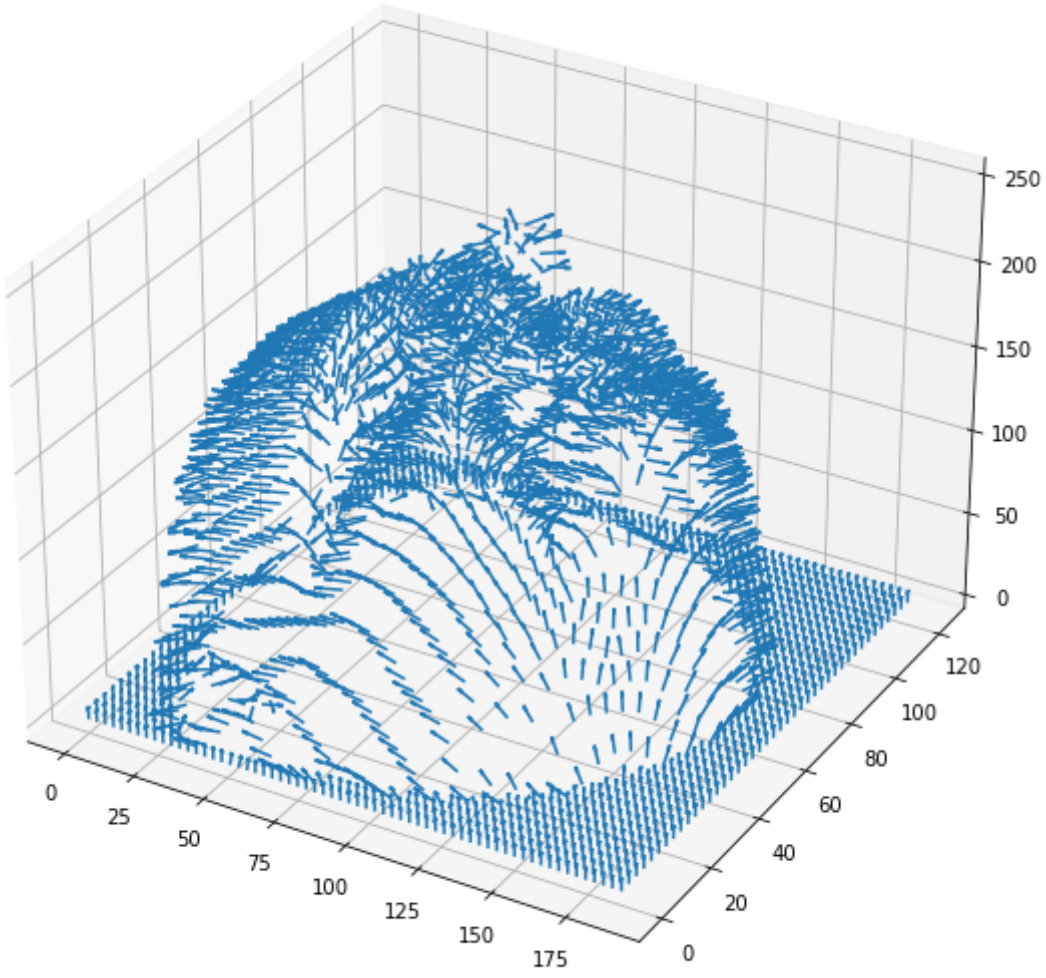
[12]:
```
# Compute and plot the surface normals
# (make sure that the normal vector is normalized to unit length)

#Step 1 - Get Normals:

f = np.array([[0,0,0], [1/2,0,-1/2], [0,0,0]])
g = np.array([[0,1/2,0], [0,0,0], [0,-1/2,0]])

dz_over_dx = convolve(heightmap, f, mode="same")
dz_over_dy = convolve(heightmap, g, mode="same")
```

```
#initializes surface normals:
normals = np.zeros((188,126,3))

#print(normals.shape)

#for each pixel:
for x in range(0,heightmap.shape[0]):
    for y in range(0, heightmap.shape[1]):


        n_xy = np.array([-dz_over_dx[x][y],-dz_over_dy[x][y],1])
        albedo_xy = linalg.norm(n_xy, ord=2)
        n_xy = n_xy/albedo_xy

        #e_xy = []
        #stack up 3 observations for a pixel:
        #for i in range(0,len(images[0][0])):

        #    e_xy.append(images[x][y][i])

        #solve linear system for the unknowns(of that pixel):
        #b_xy = np.matmul(pseudo_semi,e_xy)
        #print(b_xy.shape)

        #albedo_xy = linalg.norm(b_xy, ord=2)
        #surfn_xy = b_xy/albedo_xy

        #assign final stuff:
        normals[x][y][0] = n_xy[0]
        normals[x][y][1] = n_xy[1]
        normals[x][y][2] = n_xy[2]


#Step 2 - Plot Using the Quiver Thing:

stride = 3

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0],3),
                          np.arange(0,np.shape(normals)[1], 3),
                          np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
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=(10, 10))
ax = fig.gca(projection='3d')
print(X.shape, Y.shape, NX.shape, NY.shape, NZ.shape)
plt.quiver(X,Y,Z,NX,NY,NZ, length=8)
plt.show()
```

```
/tmp/ipykernel_162/1802199410.py:64: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

```
(42, 63) (42, 63) (42, 63) (42, 63) (42, 63)
```



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

$$I = a(x,y)\hat{\mathbf{n}}(x,y)^{\top}\hat{\mathbf{s}}(x,y)s_0 \tag{2}$$

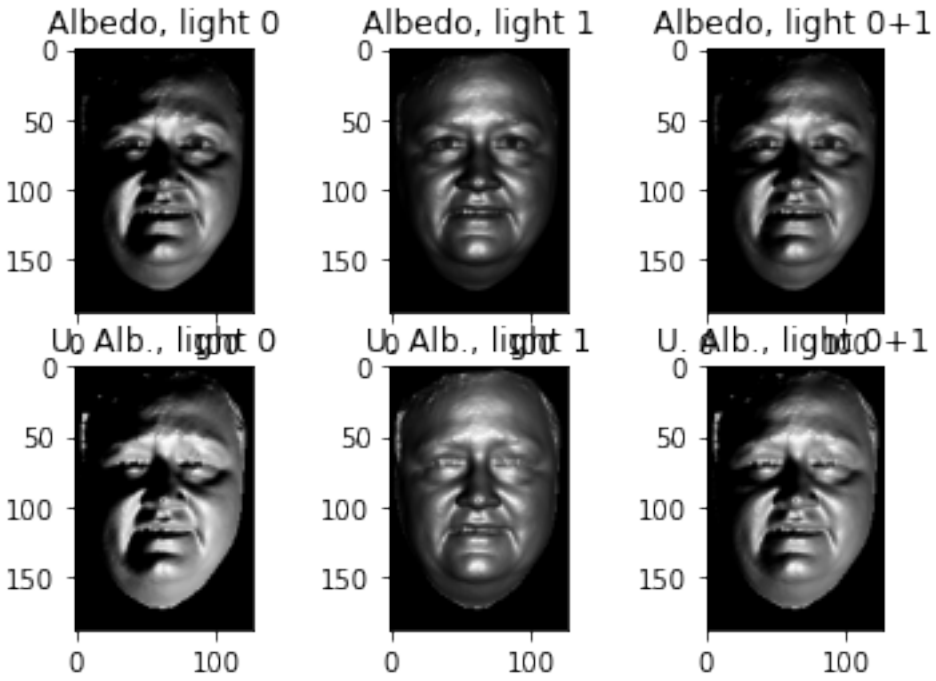
where $a(x,y)$ is the albedo for pixel (x,y), $\hat{n}(x,y)$ is the corresponding surface normal, $\hat{s}(x,y)$ the light source direction, s_0 the light source intensity. Let the light source intensity be 1 and make the 'distant light source assumption'. Use imshow with appropriate keyword arguments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

[13]:
```
#image formation:
#I  = a(x,y)((n_hat(x,y))^T)s_hat(x,y)s0
```

```
#a(x,y): albedo of pixel, have it
#n_hat(x,y): surface normal of pixel, have it, 3x1
#s_hat(x,y): light source direction, have it, each is 1x3
#so: light source intensity (1?)

#so do we have to form each image before we render them (sounds kinda trivial␣
 ↪ik)?

# Render Images

def lambertian(normals, light, albedo, intensity, mask):

    #Step 1 - forming the image:

    #albedo shape: 188 * 126
    image = np.ones(albedo.shape)

    for x in range(0,len(albedo)):
        for y in range(0, len(albedo[0])):
            a_xy = albedo[x][y] #scalar
            n_hat_xy = normals[x][y]
            s_hat_xy = light

            semi = np.matmul(n_hat_xy.T,s_hat_xy)*intensity
            image[x][y] = a_xy * semi

            if(image[x][y] < 0):
                image[x][y] = 0

    return image
```

[14]:
```
# For each of the two albedos, render three images.
# One for each of the two light sources, and one for both light-sources␣
 ↪combined.

fig = plt.figure()
ax1 = fig.add_subplot(2,3,1)
alb_1 = lambertian(normals, light_source[0], albedo, 1, 1)

ax1.imshow(alb_1, cmap="gray")
ax1.title.set_text('Albedo, light 0')

ax2 = fig.add_subplot(2,3,2)
alb_2 = lambertian(normals, light_source[1], albedo, 1, 1)
ax2.imshow(alb_2, cmap="gray")
ax2.title.set_text('Albedo, light 1')
```

```
combo_lights = np.add(light_source[0], light_source[1])

ax3 = fig.add_subplot(2,3,3)
alb_3 = lambertian(normals, combo_lights, albedo, 1, 1)
ax3.imshow(alb_3, cmap="gray")
ax3.title.set_text('Albedo, light 0+1')

ax4 = fig.add_subplot(2,3,4)
alb_4 = lambertian(normals, light_source[0], uniform_albedo, 1, 1)
ax4.imshow(alb_4, cmap="gray")
ax4.title.set_text('U. Alb., light 0')

ax5 = fig.add_subplot(2,3,5)
alb_5 = lambertian(normals, light_source[1], uniform_albedo, 1, 1)
ax5.imshow(alb_5, cmap="gray")
ax5.title.set_text('U. Alb., light 1')

ax6 = fig.add_subplot(2,3,6)
alb_6 = lambertian(normals, combo_lights, uniform_albedo, 1, 1)
ax6.imshow(alb_6, cmap="gray")
ax6.title.set_text('U. Alb., light 0+1')

plt.show()
```



1.4 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 $(3, 5)^\top$ on the 2D plane, one with $w > 0$ and one with $w < 0$.

[1 Point]

[15]:
```
#(x,y)^T -> (wx,wy,wz)^T
#w > 0: w=2, (3,5)^T -> (6,10,2)^T
#w < 0: w=-3, (3,5)^T -> (-9,-15,-3)^T

pos_w = np.array([6, 10, 2]).T

neg_w = np.array([-9, -15, -3]).T
```

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

[2 Points]

[16]:
```
# m = delta(y)/delta(x) = (3-1)/(-1-1) = 2/-2=-1
# The equation of the line is  y=-x+2 or equivalently, -x-y+2=0. This can be␣
 ↪written as the 3-vector (-1,-1,2)^T.
```

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]

[17]:
```
# We are given that lines l1 and l2 intersect. From fact 1 we know that if␣
↪point x lies on l1 if annd only if l1^Tx = 0.
# We also know that point x lies in l2 if and only if l2^Tx=0.
# Since a cross product would be a vector orthogonal to both l1 and l2, that␣
↪satisifies both l1^Tx=0 and l2^Tx=0,
# we can conclude that such a cross product x would lie on the intersection of␣
↪l1 and l2
```

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 $4x - 5y + 7 = 0$. 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]

[18]:
```
#I1=(1,1,-5), I2=(4,-5,7), I1 x I2 = (-18, -27, -9). 2D(divide by -9 and␣
↪discard z coordinate): I1 x I2 = (2,3)
```

3(e) Consider the two lines $x + 2y + 1 = 0$ and $3x + 6y - 2 = 0$. What is the special relationship between these two lines in the Euclidean plane? What is their intersection in standard Cartesian coordinates?

[2 Points]

[19]:
```
# The two lines are parallel to each other in a 2D plane, and therefore they␣
↪don't ever intersect.
```

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]

[20]:
```
# line1: (1,2,1)
# line2: (3,6,-2)
# point of intersection: line1 x line2 = i(-4-6)-j(-2-3)+k(6-6) = -10i+5j+0k =␣
↪(-10, 5, 0)
# In computer vision, this point is known as the vanishing 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]

[21]:
```
#Yes, since the homogenous coordinates of the two lines provide a point of line␣
↪intersection, even though these lines would
#never intersect at a standard cartesian plane. This is because homogeneous␣
↪coordinates will have a zero for the last coordinate when representing
#points 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]

[22]:
```
# We are given two homoegenous points x1 and x2, and that a line l passes␣
↪through these. Since this implies l^Tx1 = 0 and l^Tx2 = 0,
# x1 X x2 must be equal to l.
```

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 $\widehat{\mathbf{i}} = (1, 0, 0)^\top$, $\widehat{\mathbf{j}} = (0, 1, 0)^\top$ and $\widehat{\mathbf{k}} = (0, 0, 1)^\top$. 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: $\widehat{\mathbf{i}}' = (0.9,\ 0.4,\ 0.1\sqrt{3})^\top$ and $\widehat{\mathbf{j}}' = (-0.41833,\ 0.90427,\ 0.08539)^\top$. Find the unit vector, $\widehat{\mathbf{k}}'$, along the third axis orthogonal to both $\widehat{\mathbf{i}}'$ and $\widehat{\mathbf{j}}'$. Is there a unique such unit vector? If not, choose the one that makes an acute angle with $\widehat{\mathbf{k}}$.

[2 Points]

[23]:
```
#There is not a unique unit vector that's orthogonal to both i_hat' and j_hat',␣
↪since this vector could be pointing in
#the positive j_hat direction or in the negative j_hat direction.

import math

i_hat_prime = np.array([0.9,0.4,0.1*math.sqrt(3)])
j_hat_prime = np.array([-0.418333,0.90427,0.08539])

k_hat_prime = np.cross(i_hat_prime,j_hat_prime)
#k_hat_prime = k_hat_prime / linalg.norm(k_hat_prime, ord=2)

k_hat = np.array([0,0,1])

print("K hat prime:")
print(k_hat_prime)
```

```
dot = np.dot(k_hat_prime,k_hat)
khp_mag = linalg.norm(k_hat_prime, ord=2)
kh_mag = linalg.norm(k_hat, ord=2)

theta = np.arccos(dot/(kh_mag*khp_mag))

print("Acute angle of K hat prime: ")
print(theta)
```

```
K hat prime:
[-0.12246816 -0.1493084   0.9811762 ]
Acute angle of K hat prime:
0.19433094224726435
```

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

[2 Points]

[24]:
```
# Let v be a 3x1 vector
# Let A be a 3x3 basis matrix
# Let B be another 3x3 basis matrix
# v's coordinate in basis A: [v]A
# v's coordinate in basis B: [v]B
# Therefore the following must be true: v = A[v]A, v = B[v]B

#To change the coordinate from A to B, the rotation matrix S must satisfy:
# S[v]A = [v]B ->  S[v]A = (B^-1)A[v]A
# S = (B^-1)A, Let A = I, then S = B^-1

B = np.array([i_hat_prime,j_hat_prime,k_hat_prime]).T

#R is the rotation matrix:
R = linalg.inv(B)
print("Rotation matrix R:")
print(R)
```

```
Rotation matrix R:
[[ 0.89999928  0.40000155  0.17320523]
 [-0.41833221  0.90427233  0.08539045]
 [-0.12246838 -0.14930867  0.98117798]]
```

4(c) What is the extrinsic parameter matrix for a camera at a displacement $(-1, -2, -3)^\top$ from the origin of W and oriented such that its principal axis coincides with $\widehat{\mathbf{k}}'$, the x-axis of its image plane coincides with $\widehat{\mathbf{i}}'$ and the y-axis of the image plane coincides with $\widehat{\mathbf{j}}'$?

[3 Points]

[25]:
```
#Extrinsics matrix: |  R  -Rc |
#Extrinsics matrix: | 0 0 0 1 |
#c = (-1,-2,-3)

extr_mat_pre = np.zeros((3,4))
c_neg = np.array([1,2,3])
Rc_neg = np.matmul(R,c_neg)
#print(Rc_neg)

#extr_mat[2,3] = 1
extr_mat_pre[:,0] = R[:,0]
extr_mat_pre[:,1] = R[:,1]
extr_mat_pre[:,2] = R[:,2]
extr_mat_pre[:,3] = Rc_neg

new_row = np.array([0,0,0,1])

extr_mat = np.vstack([extr_mat_pre,new_row])

print("Extrinsic matrix: ")
print(extr_mat)
```

```
Extrinsic matrix:
[[ 0.89999928  0.40000155  0.17320523  2.21961807]
 [-0.41833221  0.90427233  0.08539045  1.6463838 ]
 [-0.12246838 -0.14930867  0.98117798  2.52244822]
 [ 0.          0.          0.          1.        ]]
```

4(d) What is the intrinsic parameter matrix for this camera, if its focal length in the x-direction is 1050 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]

[26]:
```
#dx = 1050, 1 = dx/dy, dy = 1050, (cx, cy) = (10,-5)-(0,0) = (10,-5)
#Intrinsics matrix: | -dx   s    cx|
#                   |  0   -dy   cy|
#                   |  0    0     1|

intr_mat = np.zeros((3,3))
intr_mat[0,0] = -1050
intr_mat[1,1] = -1050
intr_mat[0,2] = 10
intr_mat[1,2] = -5
intr_mat[2,2] = 1

projection_m = np.zeros((3,4))
projection_m[0,0] = 1
```

```
projection_m[1,1] = 1
projection_m[2,2] = 1

print(projection_m)

intr_f = np.matmul(intr_mat,projection_m)

print("Intrinsic matrix:")
print(intr_f)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
Intrinsic matrix:
[[-1.05e+03  0.00e+00  1.00e+01  0.00e+00]
 [ 0.00e+00 -1.05e+03 -5.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00  1.00e+00  0.00e+00]]
```

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

[3 Points]

[27]:
```
#projection matrix = intrinsic_matrix * [R | -Rc]

prj_mat = np.matmul(intr_f,extr_mat)

print("Projection matrix:")
print(prj_mat)
```

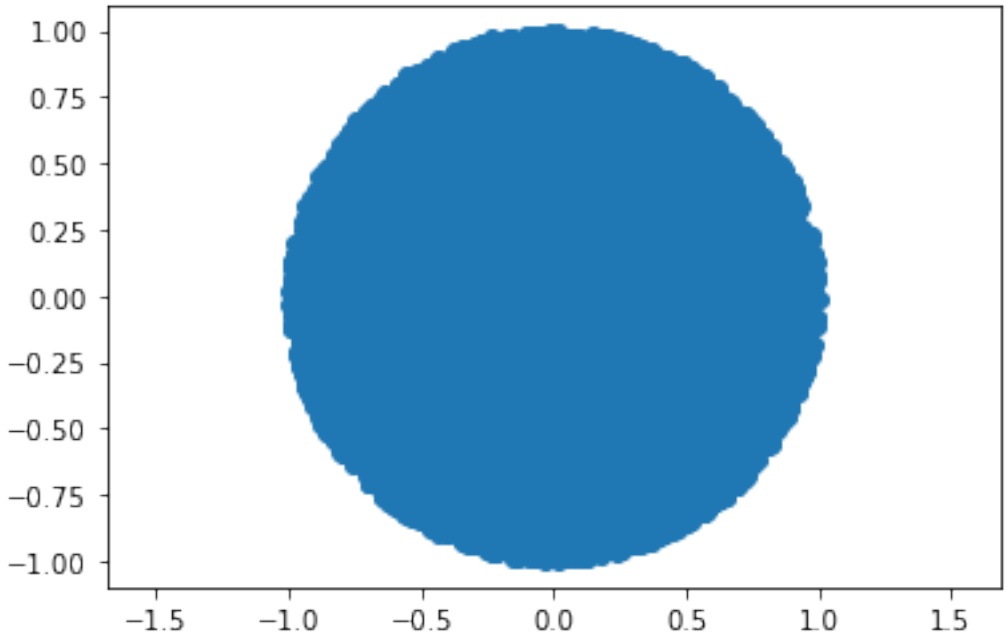
```
Projection matrix:
[[-9.46223929e+02 -4.21494718e+02 -1.72053709e+02 -2.30537449e+03]
 [ 4.39861166e+02 -9.48739403e+02 -9.45658652e+01 -1.74131523e+03]
 [-1.22468381e-01 -1.49308672e-01  9.81177982e-01  2.52244822e+00]]
```

4(f) Consider a plane, orthogonal to $\widehat{\mathbf{k}}$, at a displacement of 2 units from the origin of W along the $\widehat{\mathbf{k}}$ 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.

Compute 10000 well-distributed points on the 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]

[28]:
```
import math
import random

point_cnt = 10000 #number of points to compute
R = 1 #radius of unit circle
PI = math.pi
data = np.zeros((point_cnt,2))

for i in range(0,point_cnt):

    r = R * math.sqrt(random.random())
    theta = random.random() * 2 * PI

    x = r * math.cos(theta)
    y = r * math.sin(theta)

    #print(x)
    #print(y)

    data[i] = [x,y]
    #print(data[i])
    #print(data[i][0])
    #print(data[i][1])

x = data[:,0]
y = data[:,1]

#print(x)
#print(y)

plt.scatter(x,y)
plt.axis('equal')
plt.show()
```



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]

[29]:
```
data_4 = np.ones((point_cnt,4))

for i in range(0,point_cnt):

    r = R * math.sqrt(random.random())
    theta = random.random() * 2 * PI

    x = r * math.cos(theta)
    y = r * math.sin(theta)

    data_4[i] = [x,y,2,1]

print(data_4.shape)
```

```
(10000, 4)
```

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]

[30]:
```
homo_proj = np.matmul(prj_mat,data_4.T)

#print(homo_proj.shape)

homo_proj = homo_proj.T

#print(homo_proj)

for i in range(0,point_cnt):
    homo_proj[i][0] = homo_proj[i][0] / homo_proj[i][2]
    homo_proj[i][1] = homo_proj[i][1] / homo_proj[i][2]
    homo_proj[i][2] = homo_proj[i][2] / homo_proj[i][2]

#print(homo_proj)

proj_2d = np.zeros((point_cnt,2))
proj_2d[:,0] = homo_proj[:,0]
proj_2d[:,1] = homo_proj[:,1]

print(proj_2d)
```

```
[[-708.68285084 -466.1628257 ]
 [-546.42425596 -602.11495044]
 [-668.24959906 -654.62513402]
 …
 [-738.54314271 -603.34806949]
 [-491.00545244 -279.28756373]
 [-814.43241971 -543.67271672]]
```

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]

[31]:
```
#The shape looks a bit like a stretched out circle:

x = proj_2d[:,0]
y = proj_2d[:,1]

fig1 = plt.figure(figsize=[4, 4])
axes1 = fig1.add_axes([-1.5,-1.5,1.5,1.5])
plt.scatter(x, y)
axes = plt.gca()
axes.set_xlim([-900,-100])
axes.set_ylim([-900,-100])
axes1.set_title('The projected 2D points')
plt.show()
```

```
#plt.scatter(x,y)
#plt.axis('square')
#plt.show()
```