HW1

January 27, 2021

1 CSE 152A Winter 2021 – Assignment 1

1.1 Instructor: Ben Ochoa

- Assignment Published On: **Wed, Jan 13, 2021**
- Due On: **Wed, Jan 27, 2021 11:59 PM (Pacific Time)**

1.2 Instructions

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

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

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

1.3 Problem 1: Geometry [10 points]

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

represents the same line. All points $(\tilde{x}, \tilde{y})$ that lie on the line $a\tilde{x} + b\tilde{y} + c = 0$ satisfy the equation $\boldsymbol{l}^\top \boldsymbol{x} = 0$.

Statement 1: A point $\boldsymbol{x}$ lies on the line $\boldsymbol{l}$ if and only if $\boldsymbol{l}^\top \boldsymbol{x} = \boldsymbol{x}^\top \boldsymbol{l} = 0$

1. [6 points] Prove the following two statements that follow from **Statement 1**.

   a). The cross product between two points gives us the line connecting the two points
   b). The cross product between two lines gives us their point of intersection

2. [4 points] What is the line, in homogenous coordinates, joining the inhomogeneous points $(2, 9)$ and $(-3, 4)$.

1a) Suppose we have two arbitrary points $\boldsymbol{x_1} = (\tilde{x_1}, \tilde{y_1}, 1)^\top$ and $\boldsymbol{x_2} = (\tilde{x_2}, \tilde{y_2}, 1)^\top$.

$\boldsymbol{l} = \boldsymbol{x_1} \times \boldsymbol{x_2} = (\tilde{y_1} - \tilde{y_2}, \tilde{x_2} - \tilde{x_1}, \tilde{x_1}\tilde{y_2} - \tilde{x_2}\tilde{y_1})^\top$.

$\boldsymbol{l}^\top \boldsymbol{x_1} = (\tilde{y_1} - \tilde{y_2})\tilde{x_1} + (\tilde{x_2} - \tilde{x_1})\tilde{y_1} + \tilde{x_1}\tilde{y_2} - \tilde{y_1}\tilde{x_2} = \tilde{x_1}\tilde{y_1} - \tilde{x_1}\tilde{y_2} + \tilde{x_2}\tilde{y_1} - \tilde{x_1}\tilde{y_1} + \tilde{x_1}\tilde{y_2} - \tilde{x_2}\tilde{y_1} = 0.$

$\boldsymbol{l}^\top \boldsymbol{x_2} = (\tilde{y_1} - \tilde{y_2})\tilde{x_2} + (\tilde{x_2} - \tilde{x_1})\tilde{y_2} + \tilde{x_1}\tilde{y_2} - \tilde{y_1}\tilde{x_2} = \tilde{x_2}\tilde{y_1} - \tilde{x_2}\tilde{y_2} + \tilde{x_2}\tilde{y_2} - \tilde{x_1}\tilde{y_2} + \tilde{x_1}\tilde{y_2} - \tilde{x_2}\tilde{y_1} = 0.$

Since $\boldsymbol{l}^\top \boldsymbol{x_1} = 0$ and $\boldsymbol{l}^\top \boldsymbol{x_2} = 0$, $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ both lie on the line $\boldsymbol{l}$ by **Statement 1**. So the line $\boldsymbol{l}$, which is the cross product of $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, is the line connecting $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$.

1b) Suppose we have two arbitrary lines $\boldsymbol{l_1} = (a_1, b_1, c_1)^\top$ and $\boldsymbol{l_2} = (a_2, b_2, c_2)^\top$.

$\boldsymbol{x} = \boldsymbol{l_1} \times \boldsymbol{l_2} = (b_1c_2 - b_2c_1, a_2c_1 - a_1c_2, a_1b_2 - a_2b_1)^\top$.

$\boldsymbol{l_1}^\top \boldsymbol{x} = a_1(b_1c_2 - b_2c_1) + b_1(a_2c_1 - a_1c_2) + c_1(a_1b_2 - a_2b_1) = a_1b_1c_2 - a_1b_2c_1 + a_2b_1c_1 - a_1b_1c_2 + a_1b_2c_1 - a_2b_1c_1 = 0.$

$\boldsymbol{l_2}^\top \boldsymbol{x} = a_2(b_1c_2 - b_2c_1) + b_2(a_2c_1 - a_1c_2) + c_2(a_1b_2 - a_2b_1) = a_2b_1c_2 - a_2b_2c_1 + a_2b_2c_1 - a_1b_2c_2 + a_1b_2c_2 - a_2b_1c_2 = 0.$

Since $\boldsymbol{l_1}^\top \boldsymbol{x} = 0$ and $\boldsymbol{l_2}^\top \boldsymbol{x} = 0$, the point $\boldsymbol{x}$ lies on the lines $\boldsymbol{l_1}$ and $\boldsymbol{l_2}$, respectively, by **Statement 1**. This means the point $\boldsymbol{x}$, which is the cross product of $\boldsymbol{l_1}$ and $\boldsymbol{l_2}$, is the point of intersection of $\boldsymbol{l_1}$ and $\boldsymbol{l_2}$.

2) First, convert the inhomogeneous points to homogeneous points. We get $\boldsymbol{x_1} = (2, 9, 1)^\top$ and $\boldsymbol{x_2} = (-3, 4, 1)^\top$.

Then, we can use statement 1a to compute the line joining the two points: $\boldsymbol{l} = \boldsymbol{x_1} \times \boldsymbol{x_2} = (9 - 4, -3 - 2, 2 \cdot 4 - (-3) \cdot 9)^\top = (5, -5, 35)^\top$.

1.4 Problem 2: Image Formation and Rigid Body Transformations [20 points]

In this problem we will practice rigid body transformations and image formations through the projective camera model. The goal will be to photograph the following four points $\widetilde{\boldsymbol{X}}_1 = [-2\ -4\ 3]^T$, $\widetilde{\boldsymbol{X}}_2 = [2\ -4\ 3]^T$, $\widetilde{\boldsymbol{X}}_3 = [2\ 4\ 3]^T$, $\widetilde{\boldsymbol{X}}_4 = [-2\ 4\ 3]^T$ in the world coordinate frame. First, recall the following formula for rigid body transformation

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

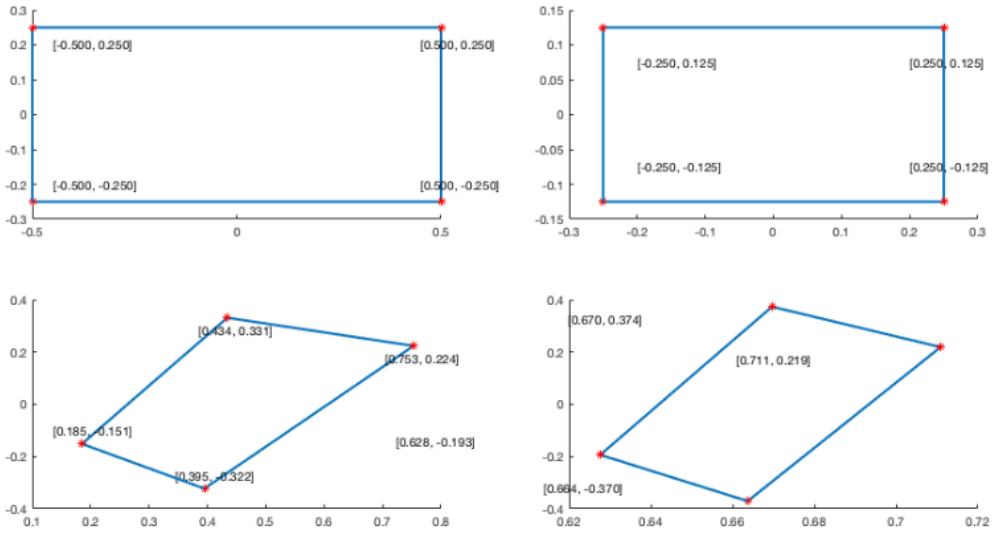
Where $\widetilde{\boldsymbol{X}}_{cam}$ is the point coordinate in the camera coordinate system. $\widetilde{\boldsymbol{X}}$ is a point in the world coordinate frame, and R and $\boldsymbol{t}$ are the rotation and translation that transform points from the

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

a). The extrinsic transformation matrix,

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

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



1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
2. [Translation]. $\boldsymbol{t} = [0\ 0\ 3]^T$. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. R encodes a 45 degrees around the z-axis and then 45 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ 3]^T$.
4. [Translation and Rotation, long distance]. Focal length = 7. R encodes a 60 degrees around the z-axis and then 30 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ -10]^T$.

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

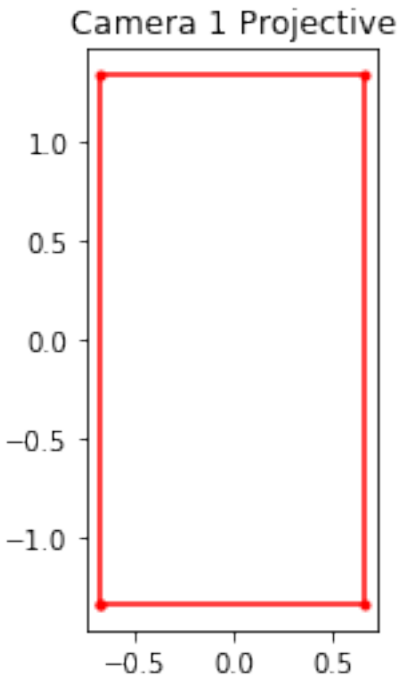
For all the four cases, include a image like above. Note that the axis are the same for each row, to facilitate comparison between the two camera models. Note: the angles and offsets used to generate these plots may be different from those in the problem statement, it's just to illustrate how to report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

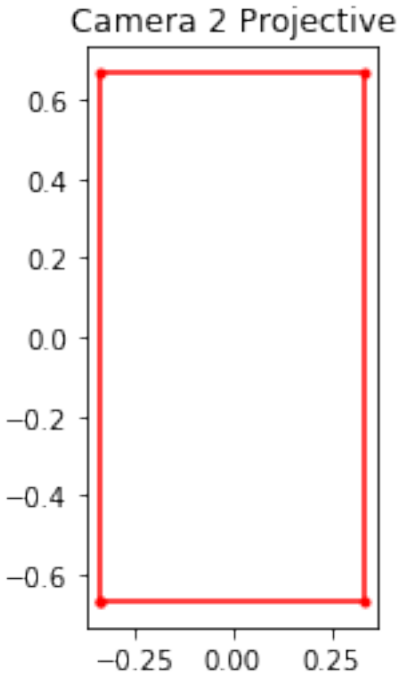
Also, Explain why you observe any distortions in the projection, if any, under this model.

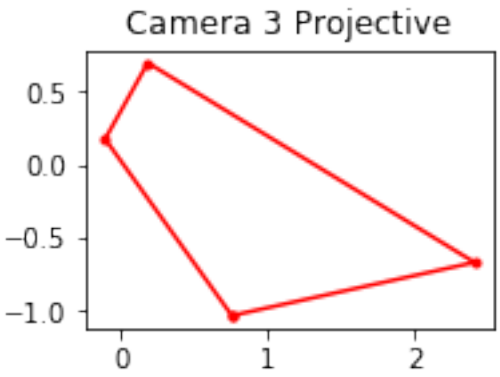
[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

# convert points from euclidian to homogeneous
def to_homog(points):
    """
    your code here
    """
    num_pts = points.shape[1]
    for i in range(num_pts):
        point = points[:,i:(i+1)]
        point_homog = np.concatenate((point, np.array([[1]])))
        if i == 0:
            points_homog = point_homog
        else:
            points_homog = np.hstack((points_homog,point_homog))
    return points_homog

# convert points from homogeneous to euclidian
def from_homog(points_homog):
    """
    your code here
    """
    num_pts = points_homog.shape[1]
    for i in range(num_pts):
        point_homog = points_homog[:-1,i:(i+1)]
        point = point_homog / points_homog[-1,i]
        if i == 0:
            points = point
        else:
            points = np.hstack((points,point))
    return points

# project 3D euclidian points to 2D euclidian
def project_points(P_int, P_ext, pts):
    """
    your code here
    """
    pts_homog = to_homog(pts)
    pts_2d_homog = P_int.dot(P_ext).dot(pts_homog)
    pts_2d = from_homog(pts_2d_homog)
    return pts_2d

def camera1():
```

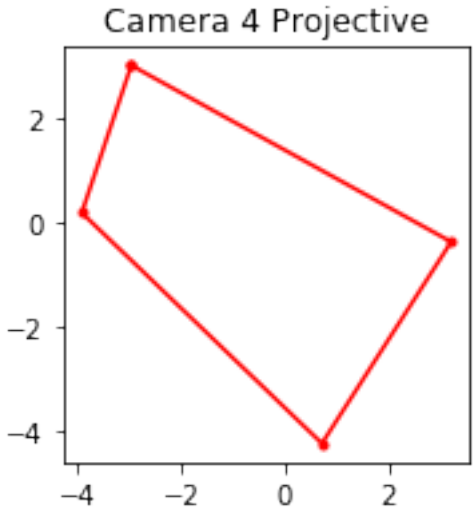
```
    """
    replace with your code
    """
    P_int_proj = np.eye(3,4)
    P_ext = np.eye(4,4)
    return P_int_proj, P_ext

def camera2():
    """
    replace with your code
    """
    P_int_proj = np.eye(3,4)
    P_ext = np.eye(4,4)
    P_ext[2,3] = 3
    return P_int_proj, P_ext

def camera3():
    """
    replace with your code
    """
    P_int_proj = np.eye(3,4)
    P_ext = np.eye(4,4)
    P_ext[2,3] = 3
    R_z = np.eye(3,3)
    R_z[0,0] = math.cos(math.pi/4)
    R_z[0,1] = -math.sin(math.pi/4)
    R_z[1,0] = math.sin(math.pi/4)
    R_z[1,1] = math.cos(math.pi/4)
    R_y = np.eye(3,3)
    R_y[0,0] = math.cos(math.pi/4)
    R_y[0,2] = math.sin(math.pi/4)
    R_y[2,0] = -math.sin(math.pi/4)
    R_y[2,2] = math.cos(math.pi/4)
    R = R_y.dot(R_z)
    P_ext[:3,:3] = R
    return P_int_proj, P_ext

def camera4():
    """
    replace with your code
    """
    P_int_proj = np.eye(3,4)
    P_int_proj[0,0] = 7
    P_int_proj[1,1] = 7
    P_ext = np.eye(4,4)
    P_ext[2,3] = -10
    R_z = np.eye(3,3)
```

```
    R_z[0,0] = math.cos(math.pi/3)
    R_z[0,1] = -math.sin(math.pi/3)
    R_z[1,0] = math.sin(math.pi/3)
    R_z[1,1] = math.cos(math.pi/3)
    R_y = np.eye(3,3)
    R_y[0,0] = math.cos(math.pi/6)
    R_y[0,2] = math.sin(math.pi/6)
    R_y[2,0] = -math.sin(math.pi/6)
    R_y[2,2] = math.cos(math.pi/6)
    R = R_y.dot(R_z)
    P_ext[:3,:3] = R
    return P_int_proj, P_ext


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

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

def main():
    point1 = np.array([[-2,-4,3]]).T
    point2 = np.array([[2,-4,3]]).T
    point3 = np.array([[2,4,3]]).T
    point4 = np.array([[-2,4,3]]).T
    points = np.hstack((point1,point2,point3,point4))

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

main()
```


Camera 1 Projective
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5


Camera 2 Projective
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.25
0.00
0.25





For the output images, image 1 is larger in size than image 2 because image 1 is closer to camera. We don't see distortion in shape in image 1 and image 2 because they are not rotated. We can see distortion in shape in image 3 and image 4 since they are rotated, which makes the points at different distances from the camera (z-values of the points become different). So the projected locations of the points on the image plane are not consistent with the original shape.

1.5 Problem 3: 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.

1.5.1 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.

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 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.

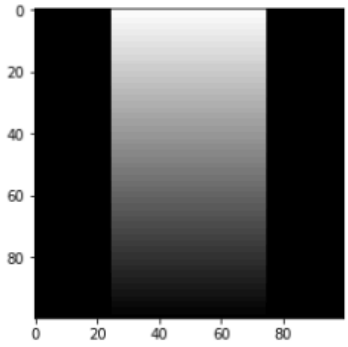
You will find all the data for this part in `synthetic_data.pickle`. Try using only `im1`, `im2` and `im4` first. Display your outputs as mentioned below.

Then use all four images (most accurate).

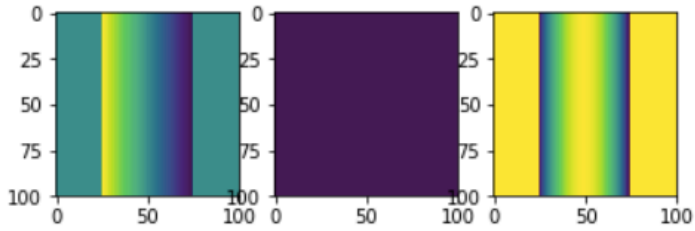
For **each** of the **two above cases** you must output:

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

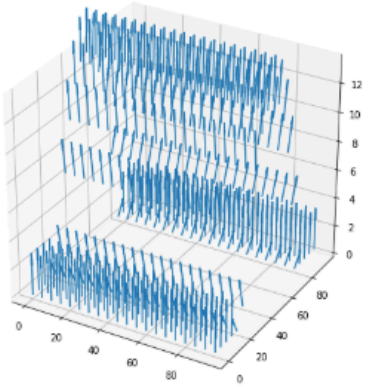
An example of outputs is shown in the figure below.



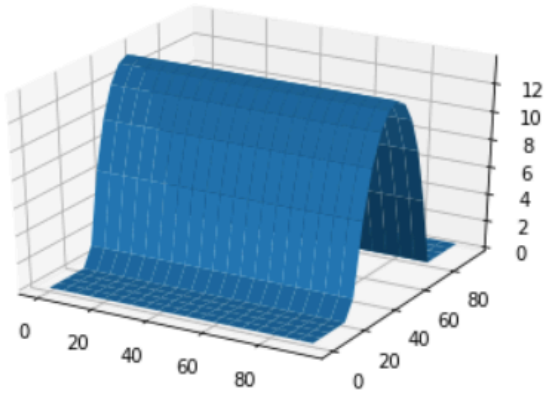
Albedo map



Normals as three separate channels



Needle map

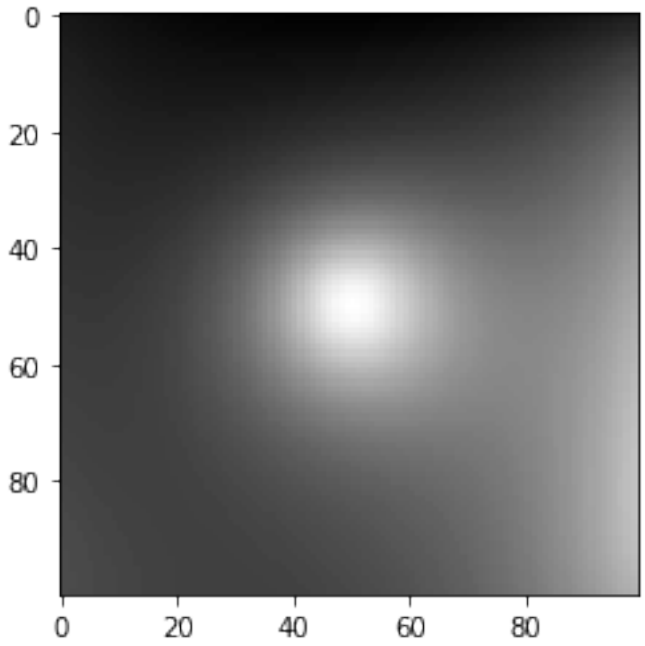


Wireframe of depth map

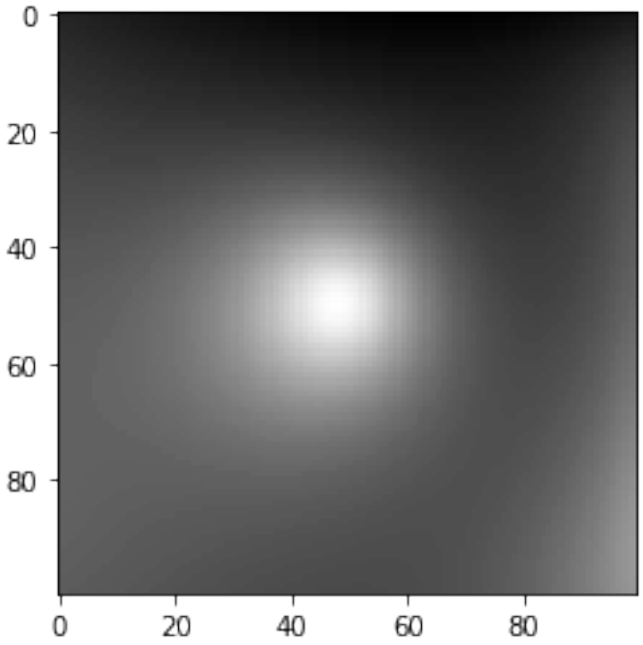
[3]:
```
## Example: How to read and access data from a pickle
import pickle
import matplotlib.pyplot as plt
%matplotlib inline
pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# 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"]))
```

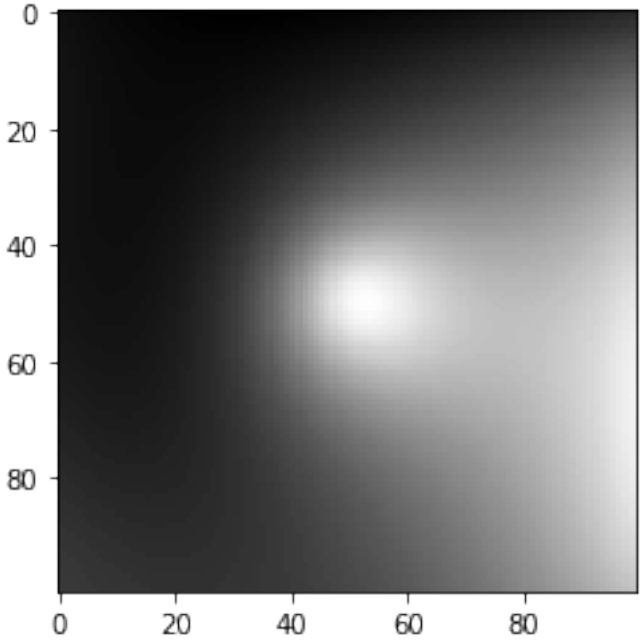
```
Keys: dict_keys(['__version__', 'l4', '__header__', 'im1', 'im3', 'im2', 'l2',
'im4', 'l1', '__globals__', 'l3'])
Image:
```



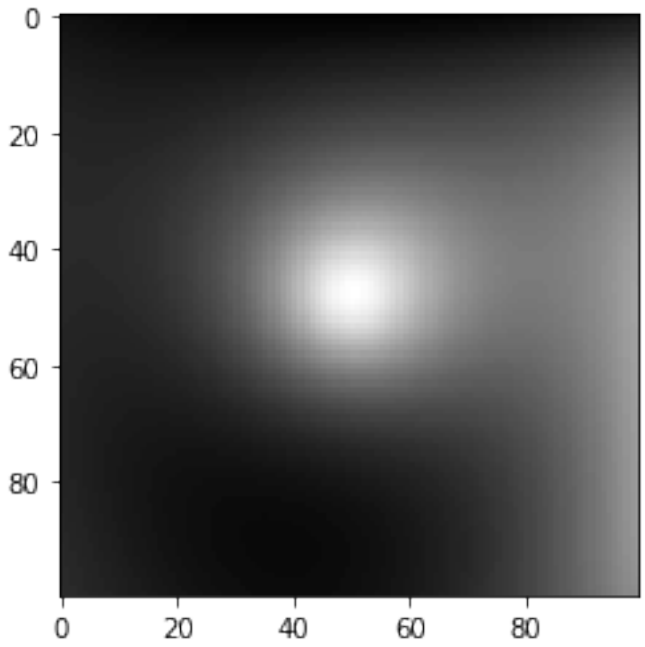

```
Light source direction: [[0 0 1]]
```




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




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




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

[4]:
```
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
```

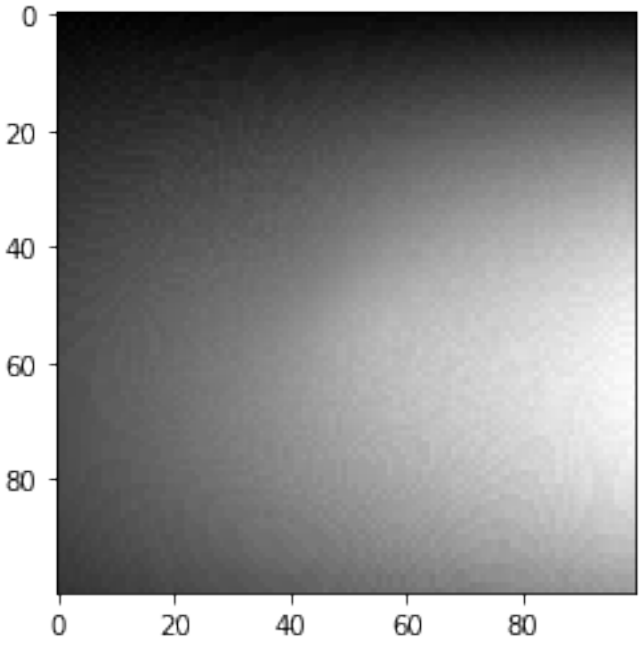
[5]:
```
def photometric_stereo(images, lights, mask, horn_niter=25000):

    """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.
    """

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

    # note:
    # images : (n_ims, h, w)
    # lights : (n_ims, 3)
    # mask   : (h, w)

    lights_pinv = np.linalg.pinv(lights)
    n_ims, h, w = images.shape
    albedo = np.ones(images[0].shape)
    normals = np.dstack((np.zeros(images[0].shape),
```

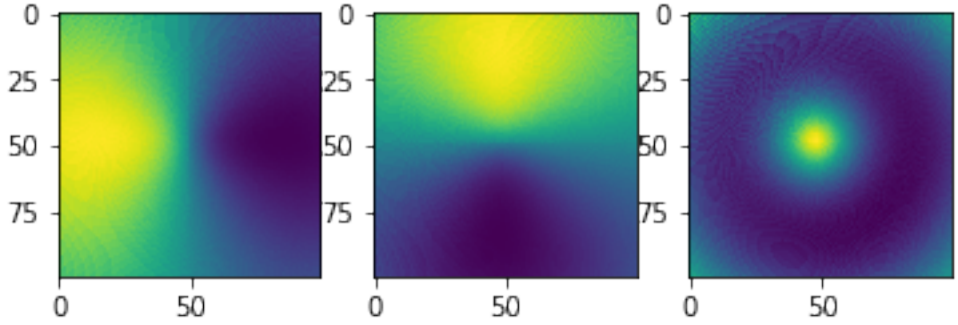
```
                        np.zeros(images[0].shape),
                        np.ones(images[0].shape)))

    for i in range(h):
        for j in range(w):
            intensity = np.ones((n_ims,1))
            for n in range(n_ims):
                intensity[n,0] = images[n,i,j]
            normal = lights_pinv.dot(intensity)
            norm = np.linalg.norm(normal)
            albedo[i,j] = norm
            normals[i,j] = normal.reshape(3) / norm

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

    for i in range(h):
        for j in range(w):
            normal = normals[i,j]
            gx[i,j] = normal[0] / normal[2]
            gy[i,j] = normal[1] / normal[2]

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

    return albedo, normals, H_horn
```

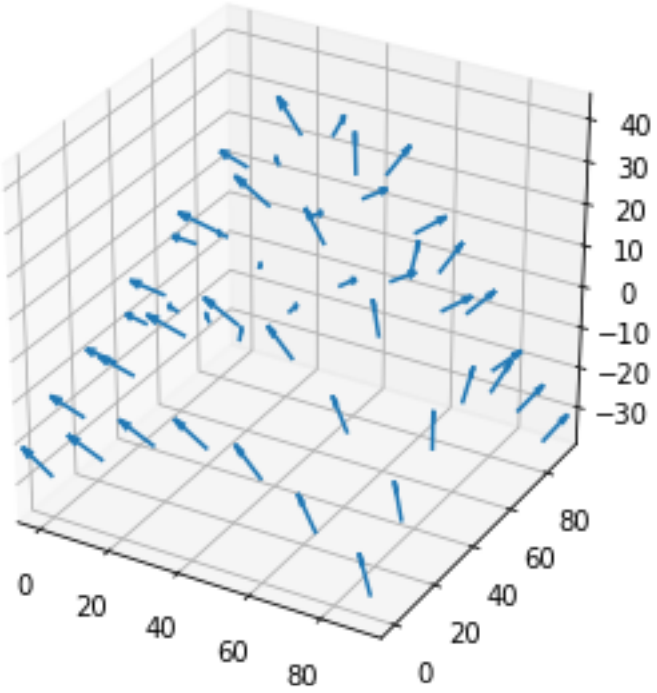
[6]:
```
from mpl_toolkits.mplot3d import Axes3D

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

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

## Set-1: USING ONLY im1, im2, im4

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)

# ------------------------------------------------------------------------
# The following code is just a working example so you don't get stuck with any
```

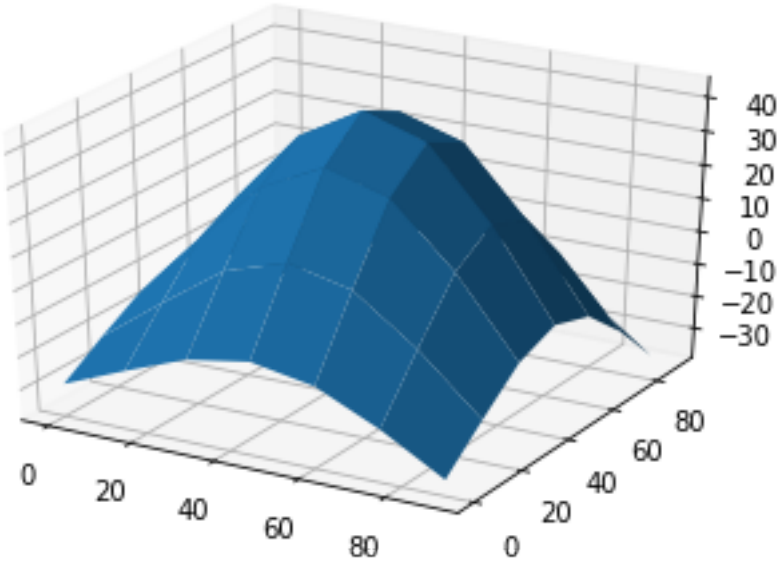
```
# of the graphs required. You may want to write your own code to align the
# results in a better layout. You are also free to change the function
# however you wish; just make sure you get all of the required outputs.
# ----------------------------------------------------------------------

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()

visualize(albedo, normals, horn_depth)
```



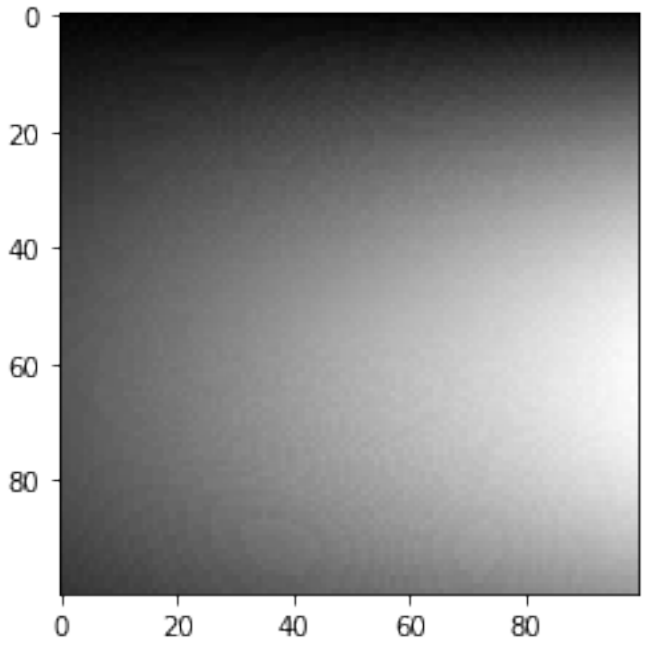


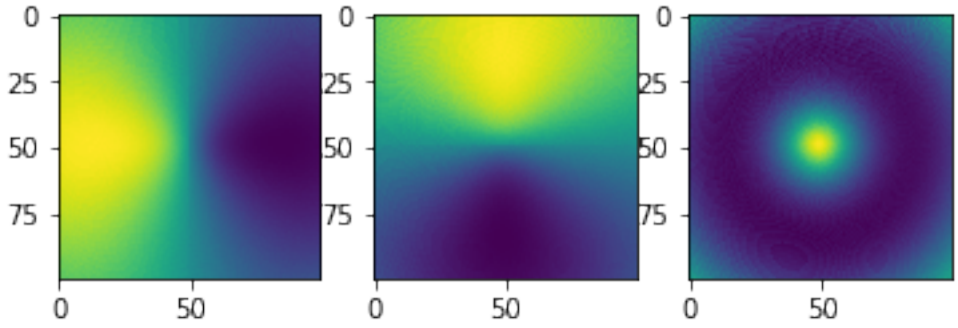


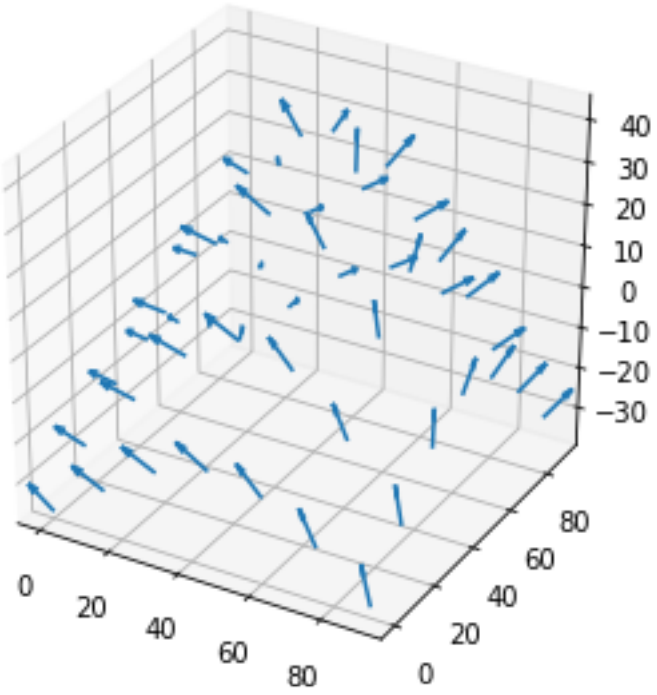


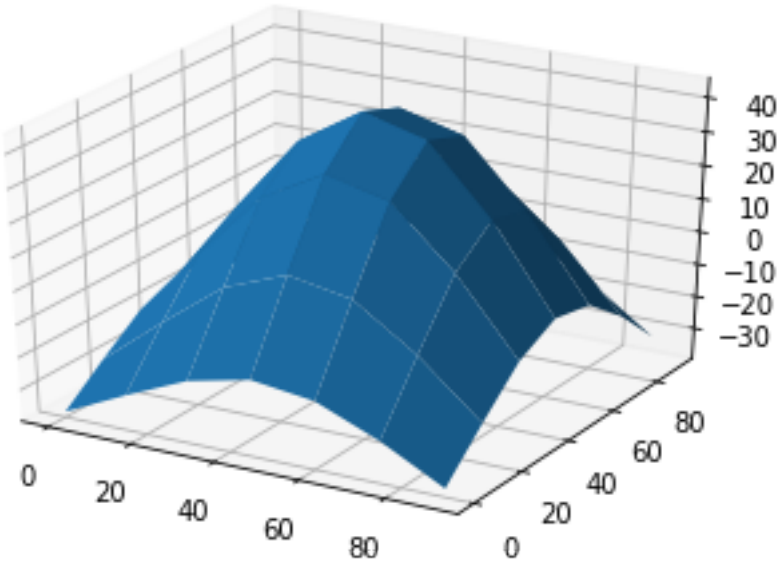
[7]:
```
# Don't forget to run your photometric stereo code on TWO sets of images!
# (One being {im1, im2, im4}, and the other being {im1, im2, im3, im4}.)
""" ==========
```

```
YOUR CODE HERE
========== """
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)
```




0
25
50
75
0
50
0
25
50
75
0
50
0
25
50
75
0
50


40
30
20
10
0
−10
−20
−30
0
20
40
60
80
0
20
40
60
80



1.6 Problem 4: 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.

Plot the face in 2-D [2 pts]

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

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.

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.

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

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

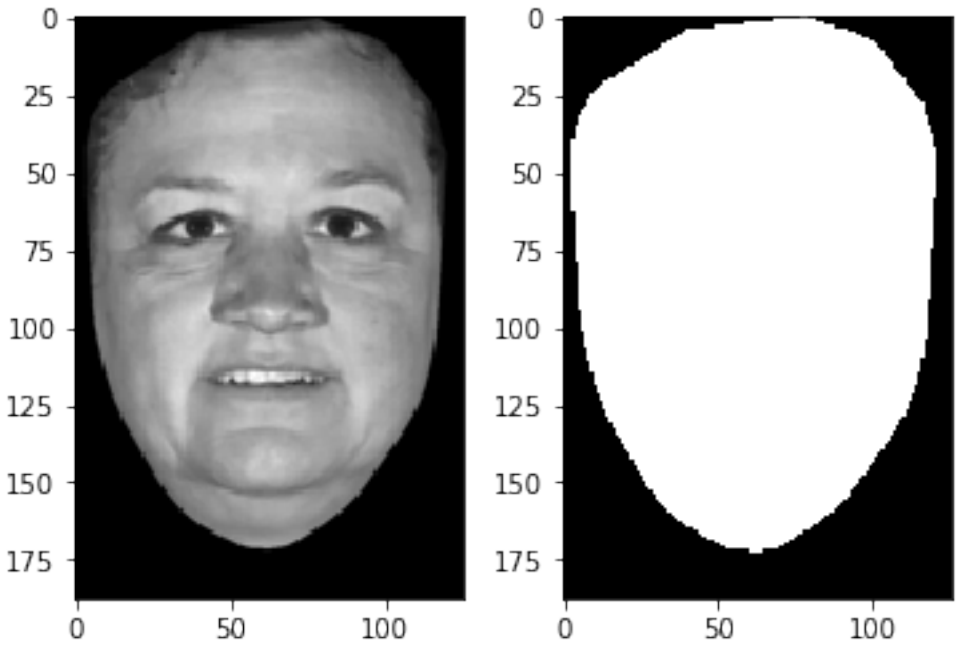
where $a(x, y)$ is the albedo for pixel (x, y), $\hat{n}(x, y)$ is the corresponding surface normal, $\hat{s}(x, y)$ the light source direction, s_0 the light source intensity. Let the light source intensity be 1 and make the 'distant light source assumption'. Use imshow with appropriate keyword arguments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

[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')
```

Plot the face in 2-D

We can see the texture of the face. Image on the left is realistic albedo, so we can see the realistic face. And image on the right is uniform albedo, so we can just see the face with uniform color, which is white in this case.

[9]:
```
plt.subplot(1, 2, 1)
plt.imshow(albedo, cmap="gray")

plt.subplot(1, 2, 2)
plt.imshow(uniform_albedo, cmap="gray")

plt.show()
```

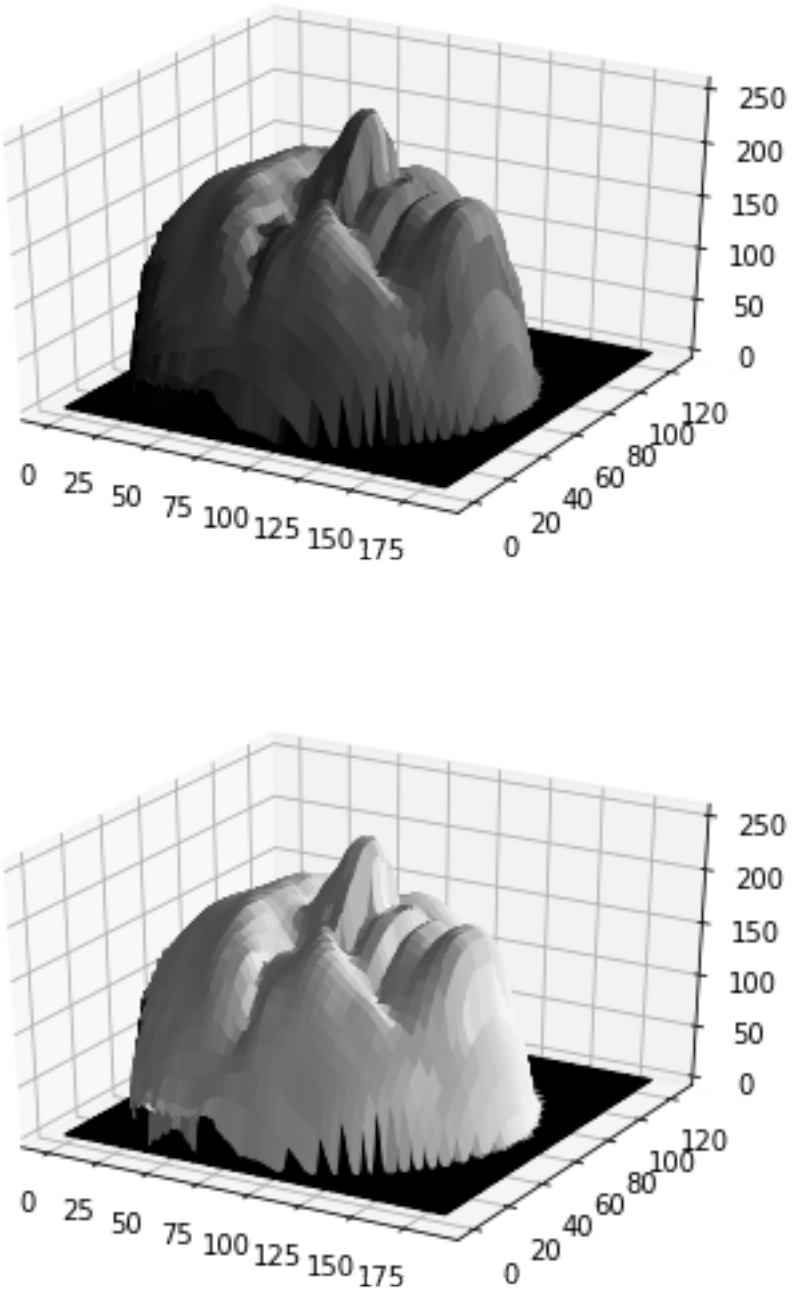


Plot the face in 3-D

We can see the contour or the surface of the face in 3-D. The two output images are the same in the contour of the face, but different in color. The color of each surface is the same as the corresponding color in its 2-D albedo.

[10]:
```
shape = albedo.shape
X = np.ones(shape)
for i in range(shape[0]):
    X[i,:] = X[i,:] * i
Y = np.ones(shape)
for i in range(shape[1]):
    Y[:,i] = Y[:,i] * i

albedo_RGBA = plt.cm.gray(albedo)
uniform_albedo = 1.0 * uniform_albedo
u_albedo_RGBA = plt.cm.gray(uniform_albedo)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, heightmap, facecolors=albedo_RGBA)
plt.show()

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, heightmap, facecolors=u_albedo_RGBA)
```

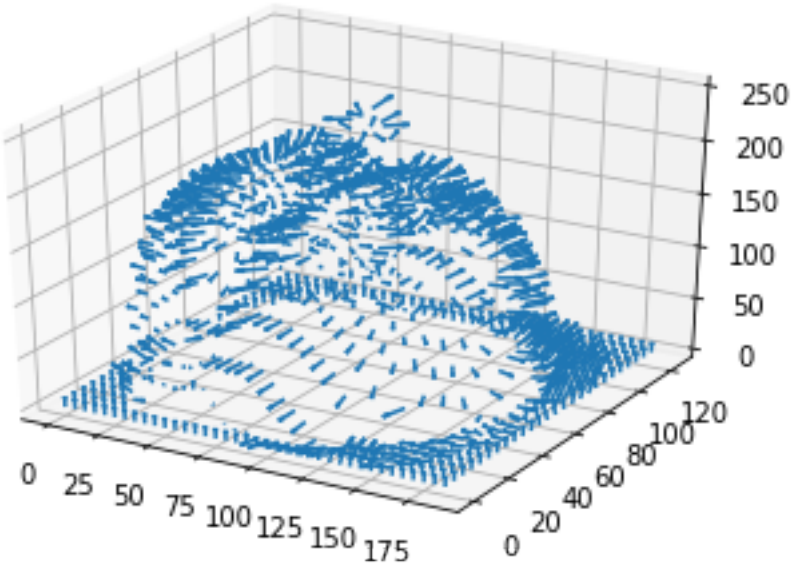
```
plt.show()
```



Surface normals

[11]:
```
gx = np.gradient(heightmap, axis=0)
gy = np.gradient(heightmap, axis=1)
normals = np.dstack((-gx, -gy, np.ones(heightmap.shape)))
```

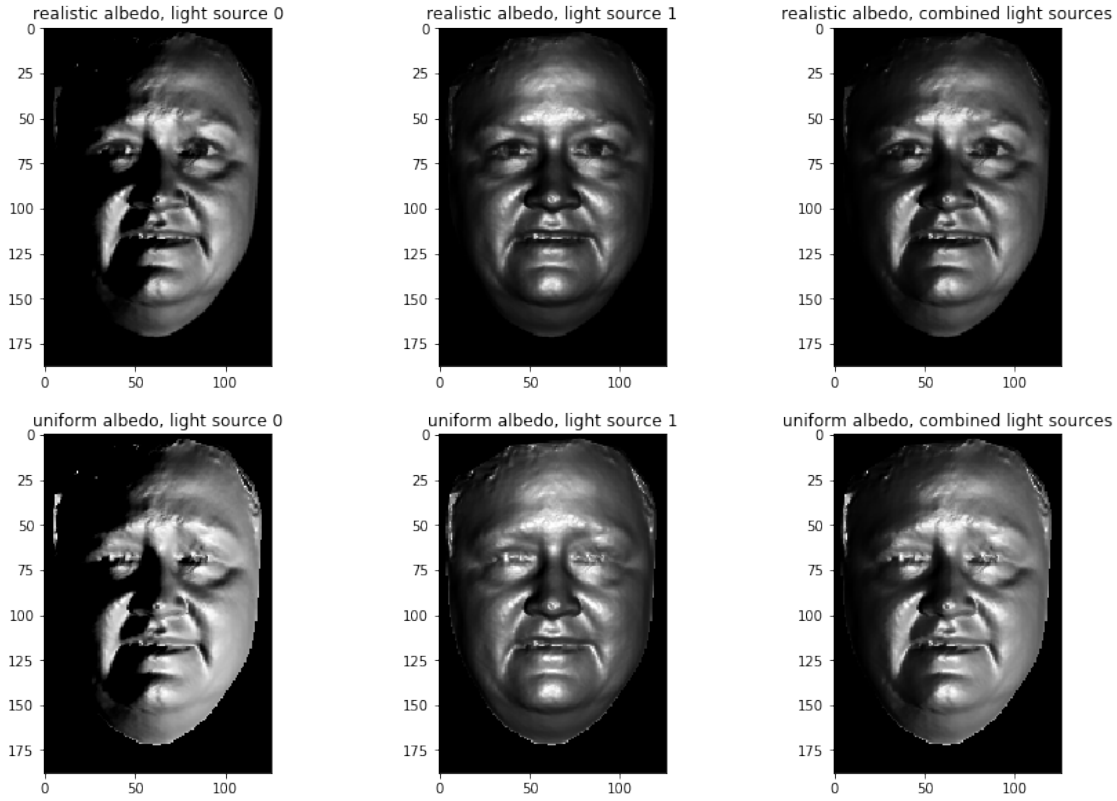
```
for i in range(normals.shape[0]):
    for j in range(normals.shape[1]):
        normals[i,j] = normals[i,j] / linalg.norm(normals[i,j])

stride = 5
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 = 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()
ax = fig.gca(projection='3d')
plt.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()
```



Render images

[12]:
```
def form_image(albedo, normals, light_source, light_intensity):
    shape = albedo.shape
    image = np.zeros(shape)
    for i in range(shape[0]):
        for j in range(shape[1]):
            normal = normals[i,j].reshape(3,1)
```

```
            light_direction = light_source.reshape(3,1)
            image[i,j] = max(0, albedo[i,j] * normal.T.dot(light_direction) *␣
↪light_intensity)

    return image

image1 = form_image(albedo, normals, light_source[0], 1)
plt.subplot(2, 3, 1)
plt.gca().set_title('realistic albedo, light source 0')
plt.imshow(image1, cmap="gray")

image2 = form_image(albedo, normals, light_source[1], 1)
plt.subplot(2, 3, 2)
plt.gca().set_title('realistic albedo, light source 1')
plt.imshow(image2, cmap="gray")

image3 = image1 + image2
plt.subplot(2, 3, 3)
plt.gca().set_title('realistic albedo, combined light sources')
plt.imshow(image3, cmap="gray")

image4 = form_image(uniform_albedo, normals, light_source[0], 1)
plt.subplot(2, 3, 4)
plt.gca().set_title('uniform albedo, light source 0')
plt.imshow(image4, cmap="gray")

image5 = form_image(uniform_albedo, normals, light_source[1], 1)
plt.subplot(2, 3, 5)
plt.gca().set_title('uniform albedo, light source 1')
plt.imshow(image5, cmap="gray")

image6 = image4 + image5
plt.subplot(2, 3, 6)
plt.gca().set_title('uniform albedo, combined light sources')
plt.imshow(image6, cmap="gray")

plt.subplots_adjust(right=2, top=2)

plt.show()
```


realistic albedo, light source 0
realistic albedo, light source 1
realistic albedo, combined light sources
uniform albedo, light source 0
uniform albedo, light source 1
uniform albedo, combined light sources

[ ]: