CSE 252A Computer Vision I Fall 2018 - Assignment 2

Instructor: David Kriegman

Assignment Published On: Wednesday, October 24, 2018

Due On: Wednesday, November 7, 2018 11:59 pm

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains theoretical and programming exercises. If you plan to submit hand written answers for theoretical exercises, please be sure your writing is readable and merge those in order with the final pdf you create out of this notebook. You could fill the answers within the notebook iteself by creating a markdown cell.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), but you may not use packages that directly solve the problem.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- **Late policy** - 10% per day late penalty after due date up to 3 days.

Problem 1: Steradians [2 pts]

Calculate the number of steradians contained in a spherical wedge with radius $r = 1$, defined by $\theta = \frac{\pi}{6}$, $\phi = \frac{\pi}{6}$ centered around vector $(\frac{\sqrt{2}}{4}, \frac{\sqrt{2}}{4}, \frac{\sqrt{3}}{2})$.

In [1]:

```
import numpy as np
from math import sqrt
from numpy import linalg as la
from numpy import sin, cos ,pi
from scipy.integrate import dblquad
```

In [2]:

```
a = np.array([sqrt(3)/4,1/4,sqrt(3)/2])
b = np.array([sqrt(2)/4,sqrt(2)/4,sqrt(3)/2])
Cos_alpha = np.dot(a,b)/(la.norm(a)*la.norm(b))
print(Cos_alpha)
```

```
0.991481456572267
```

In [3]:

```
def integrand(y, x):
    return Cos_alpha*sin(y)
ans, err = dblquad(integrand, 0, pi/6,
                   lambda x: 0,
                   lambda x: pi/6)
print(ans)
```

```
0.069551367794458 35
```

Problem 2: Irradiance [6 pts]

Consider a camera looking at large lambertian wall with constant albedo, illuminated by a light source at infinity such that the radiance emitted by the wall is L in all directions. The angle between optical axis and the wall's surface normal is 60 degrees. The focal length of the camera is 50mm and the pixels are 1mm by 1mm.

1. If the camera is, 1m along the line of site from the wall, what is the irradiance for that central pixel?
2. If the camera is moved back to be 2m along the line of site from the wall, what is the irradiance for that central pixel?
3. If the camera is moved back to be 4m along the line of site from the wall, what is the irradiance for that central pixel?
4. What can you learn about irradiance from this example?

In [4]:

```
def irr(z):
    f = 0.05
    d = (10**-6*z**2)/(f**2+z*f)**2
    return d
```

In [5]:

```
print(irr(1),irr(2),irr(4))
```

```
0.00036281179138321985 0.0003807257584770969 0.0003901844231062337
```

In [6]:

```
a = np.array([1,2,3])
b = np.array([1,2,3]).T
np.dot(a,b)
```

Out[6]:

```
14
```

Problem 3: Diffused Objects and Brightness [4 pts]

We see a diffuse torus centered at the origin in an orthographic camera, looking down the z-axis. The parameters of the torus are shown in the Figure "Problem3 torus" and the albedo is ρ.

Problem3 torus

This torus is illuminated by a distant point light source whose direction is $(0, 0, 1)$. There is no other illumination.

What is the brightness at a point (x, y) on the surface?

Problem 4: Occlusion, Umbra and Penumbra [2 pts]

We have a square area source and a square occluder, both parallel to a plane.

The edge lengths of the source and occluder are 2 and 4, respectively, and they are vertically above one another with their centers aligned. The distances from the occluder to the source and plane are both 3.

1. What is the area of the umbra on the plane?
2. What is the area of the penumbra on the plane?

Problem 5: Photometric Stereo, Specularity Removal [14 pts]

The goal of this problem is to implement a couple of different algorithms that reconstruct a surface using the concept of photometric stereo.

Additionally, you will implement the specular removal technique of Mallick et al., which enables photometric stereo reconstruction of certain non-Lambertian materials.

You can assume a Lambertian reflectance function once specularties are removed, but the albedo is unknown and non-constant in the images.

Your program will take in multiple images as input along with the light source direction (and color when necessary) for each image.

Data

Synthetic Images, Specular Sphere Images, Pear Images for Part 1, 2, 3: Available in `*.pickle` files (graciously provided by Satya Mallick) which contain

- `im1`, `im2`, `im3`, `im4`... images.
- `l1`, `l2`, `l3`, `l4`... light source directions.
- `c` (when required) color of light source.

Part 1: [6 pts]

Implement the photometric stereo technique described in Forsyth and Ponce 2.2.4 (*Photometric Stereo: Shape from Multiple Shaded Images*) and the lecture notes.

Your program should have two parts:

1. Read in the images and corresponding light source directions, and estimate the surface normals and albedo map.
2. Reconstruct the depth map from the surface normals. You can first try the naive scanline-based shape by integration method described in the book. If this does not work well on real images, you can use the implementation of the Horn integration technique given below in `horn_integrate` function.

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 above cases you must output:

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

An example of outputs is shown in the Figure "Problem5 example".

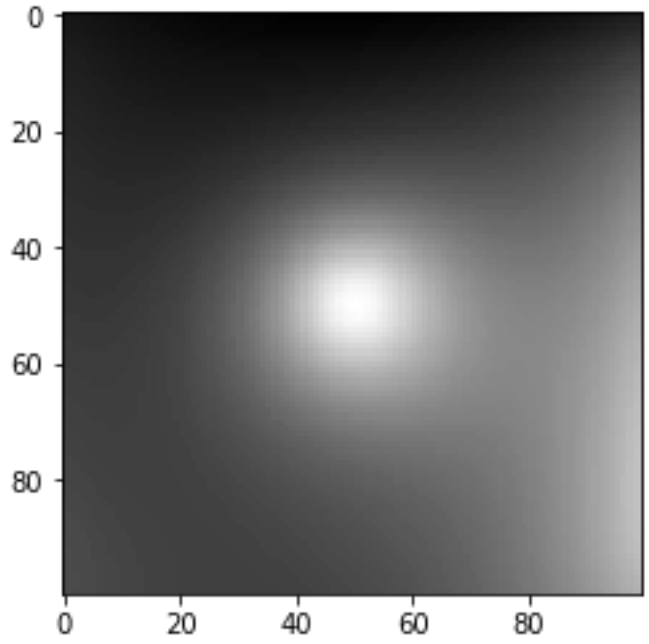
Problem5 example

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

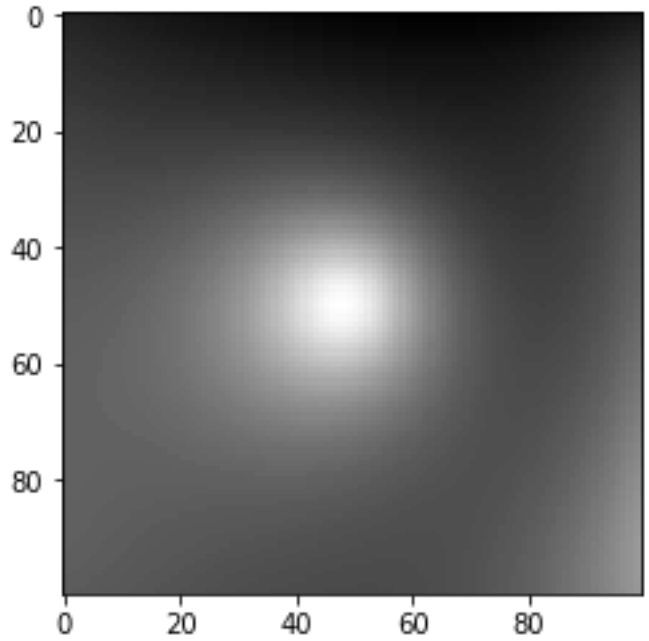
In [7]:

```
## 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)
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"]))
```

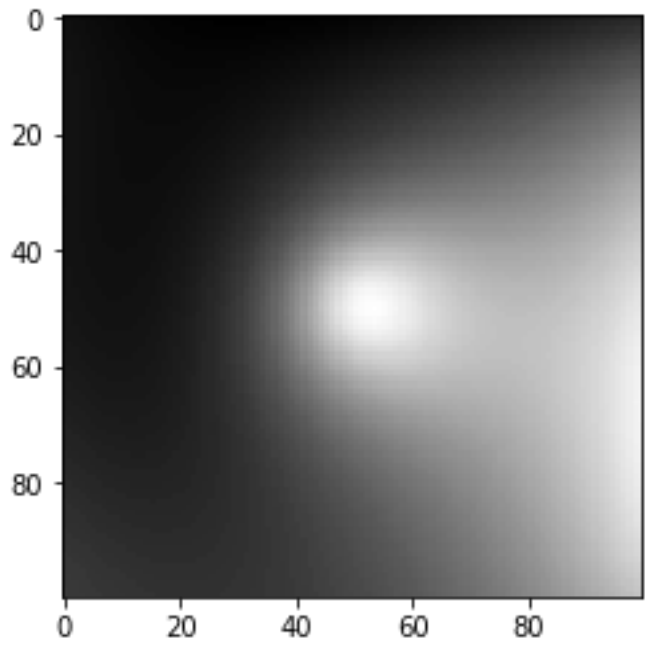
```
Keys: dict_keys(['__version__', 'l4', '__header__', 'im1', 'im3', 'i
m2', 'l2', 'im4', 'l1', '__globals__', 'l3'])
Image:
```



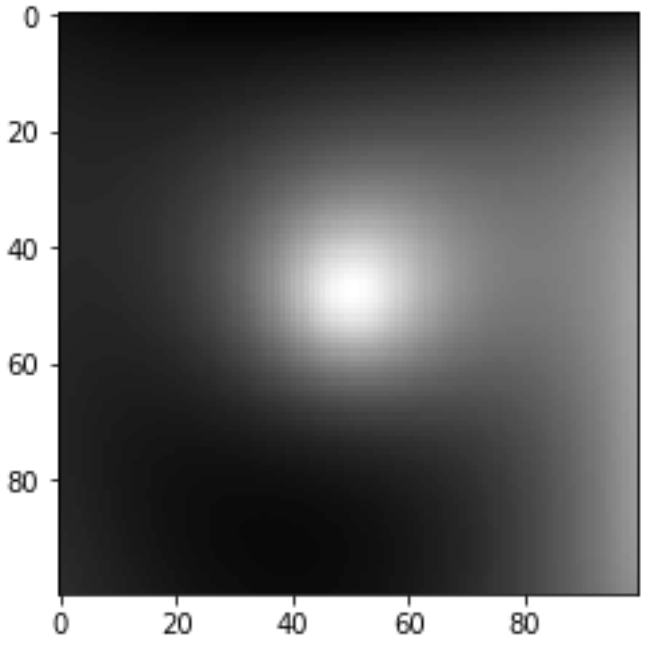

```
Light source direction: [[0 0 1]]
```




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




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




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

In [8]:

```
import numpy as np
from scipy.signal import convolve
#from numpy import linalg
from numpy import linalg as la



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
```

In [9]:

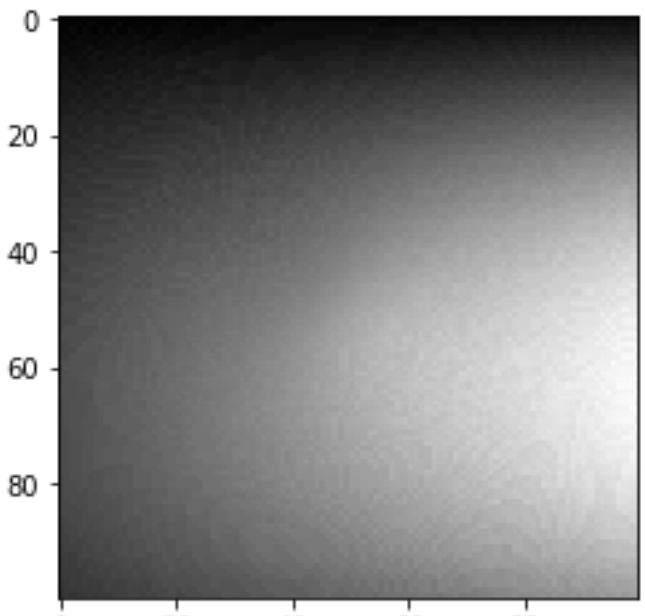
```
def photometric_stereo(images, lights, mask):
    # H -- depth
    '''
    your implementaion
    '''
    lights = np.matrix(lights)

    e = np.ones((images[0].shape[0],images[0].shape[1],lights.shape[0]))
    b = np.ones((images[0].shape[0],images[0].shape[1],3))
    p = np.ones((images[0].shape))
    q = np.ones((images[0].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)))
    H = np.ones(images[0].shape)
    H_horn = np.ones(images[0].shape)

    for i in range(images[0].shape[0]):
        for j in range(images[0].shape[1]):
            for k in range(lights.shape[0]):
                e[i,j][k] = images[k][i,j]
            b[i,j] = np.dot(la.inv(lights.T*lights)*lights.T,e[i,j])
            albedo[i,j] = la.norm(b[i,j])
            normals[i,j] = b[i,j]/albedo[i,j]
            p[i,j] = normals[i,j][0]/normals[i,j][2]
            q[i,j] = normals[i,j][1]/normals[i,j][2]
    for i in range (1,H.shape[0]):
        H[i,0] = H[i-1,0]+q[i,0]
    for i in range (1,H.shape[0]):
        for j in range (1,H.shape[1]):
            H[i,j] = H[i,j-1]+p[i,j]




    H_horn = horn_integrate(p, q, mask, 1000)



    return albedo, normals, H, H_horn
```

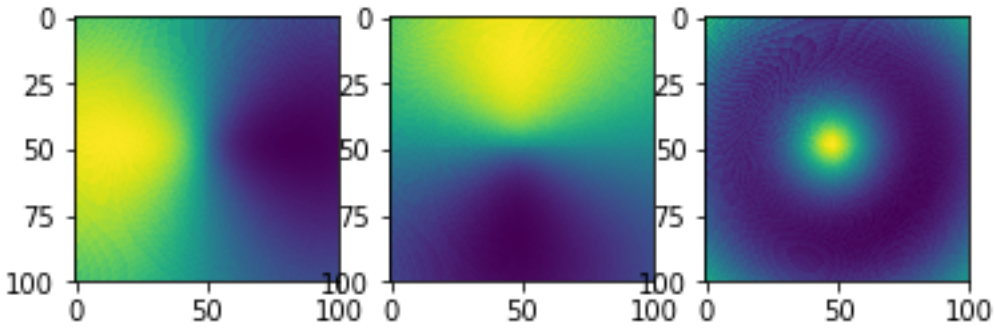
In [10]:

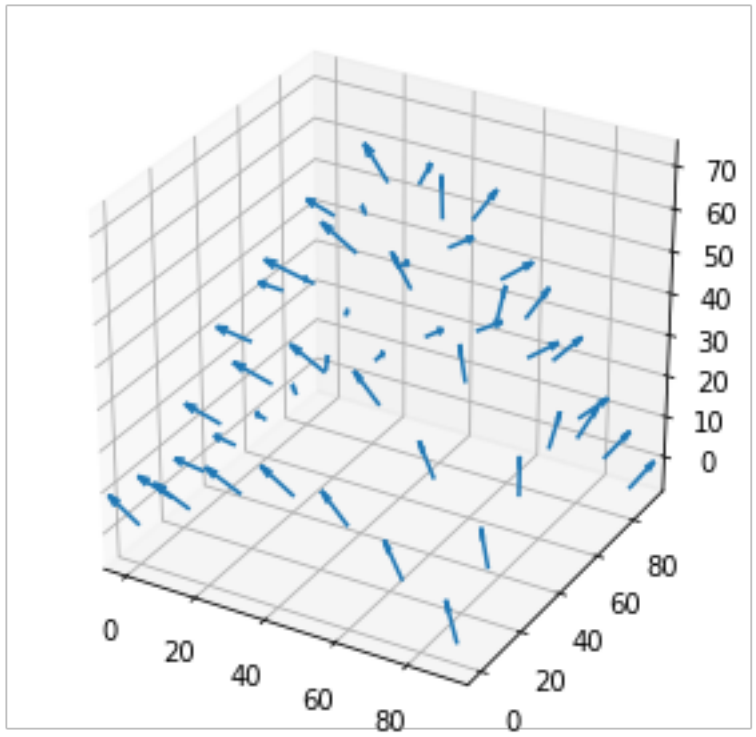
```
from mpl_toolkits.mplot3d import Axes3D

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

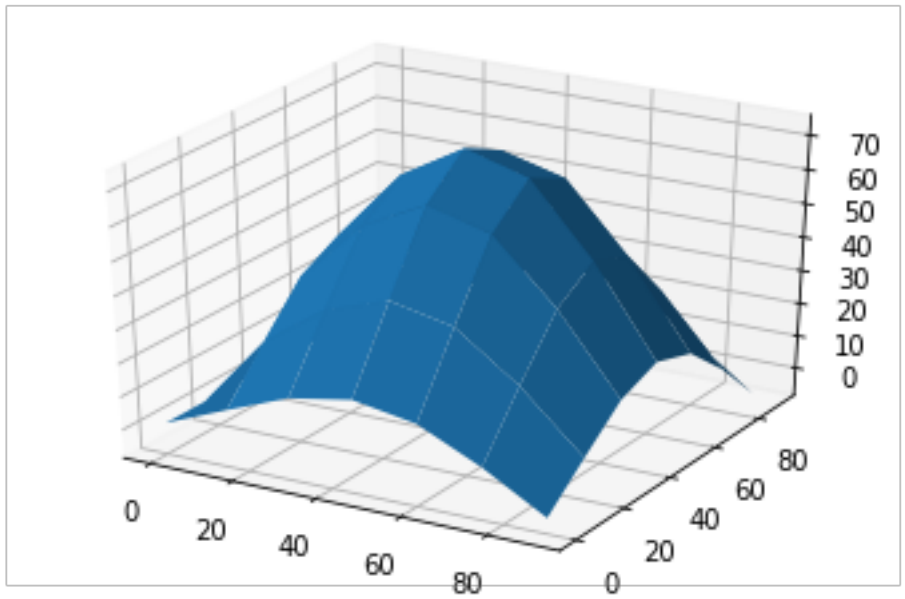
```
lights = np.vstack((data["l1"], data["l2"], data["l4"]))
# lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))

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

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

albedo, normals, depth, horn = photometric_stereo(images, lights, mask)

# ---------------------------------------------------------------------
# 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.
# ---------------------------------------------------------------------

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

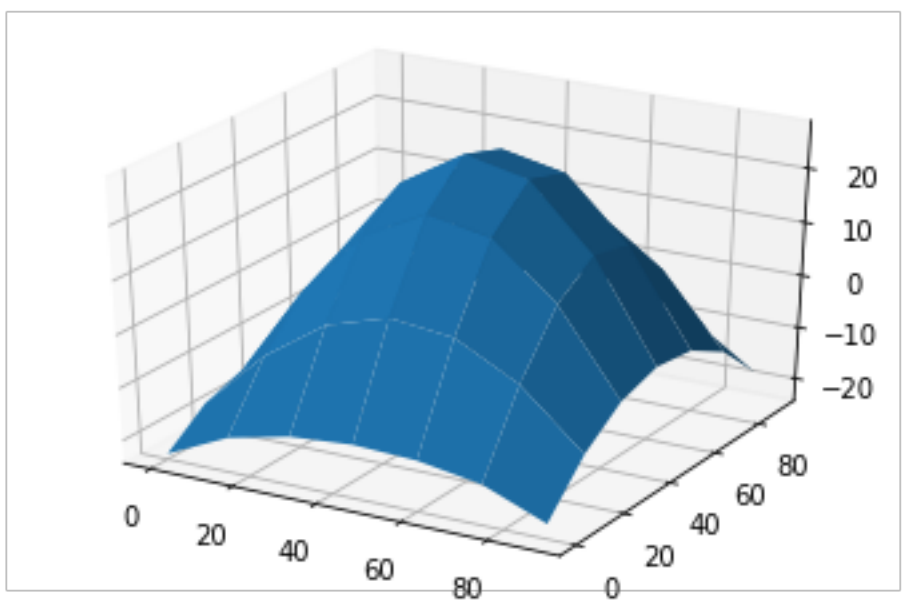
```
# plotting wireframe depth map
H = depth[::stride,::stride]
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y, H.T)
plt.show()

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






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


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


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

Part 2: [4 pts]

Implement the specularity removal technique described in *Beyond Lambert: Reconstructing Specular Surfaces Using Color* (by Mallick, Zickler, Kriegman, and Belhumeur; CVPR 2005).

Your program should input an RGB image and light source color and output the corresponding SUV image.

Try this out first with the specular sphere images and then with the pear images.

For each specular sphere and pear images, include

1. The original image (in RGB colorspace).
2. The recovered S channel of the image.
3. The recovered diffuse part of the image - Use $G = \sqrt{U^2 + V^2}$ to represent the diffuse part.

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

In [11]:

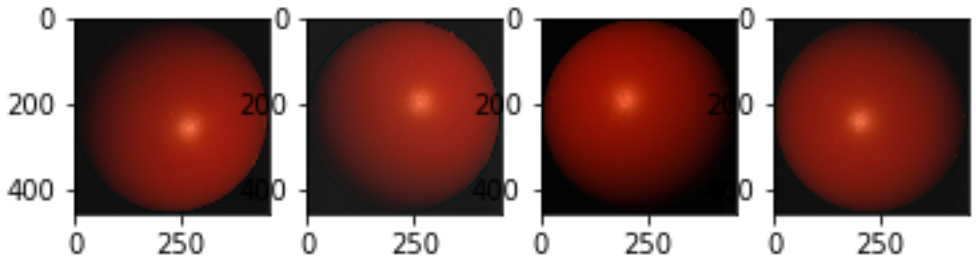
```
def get_rot_mat(rot_v, unit=None):
    '''
    Takes a vector and returns the rotation matrix required to align the
    unit vector(2nd arg) to it.
    '''
    if unit is None:
        unit = [1.0, 0.0, 0.0]

    rot_v = rot_v/np.linalg.norm(rot_v)
    uvw = np.cross(rot_v, unit) #axis of rotation

    rcos = np.dot(rot_v, unit) #cos by dot product
    rsin = np.linalg.norm(uvw) #sin by magnitude of cross product

    #normalize and unpack axis
    if not np.isclose(rsin, 0):
        uvw = uvw/rsin
    u, v, w = uvw

    # Compute rotation matrix
    R = (
        rcos * np.eye(3) +
        rsin * np.array([
            [ 0, -w,  v],
            [ w,  0, -u],
            [-v,  u,  0]
        ]) +
        (1.0 - rcos) * uvw[:,None] * uvw[None,:]
    )

    return R

def RGBToSUV(I_rgb, rot_vec):
    '''
    your implementation which takes an RGB image and a vector encoding
    the orientation of S channel wrt to RGB
    '''
    y = I_rgb.shape[0]
    x = I_rgb.shape[1]
    R = get_rot_mat(rot_vec)
    #R = np.matrix(R)
    I_suv = np.zeros((I_rgb.shape))
    S = np.ones(I_rgb.shape[:2])
    G = np.ones(I_rgb.shape[:2])
    for i in range (y):
        for j in range(x):
            I_suv[i,j] = np.dot(R,I_rgb[i,j])
            S[i,j] = I_suv[i,j][0]
            G[i,j] = la.norm(I_suv[i,j][1:])

    return S, G
```

In [12]:

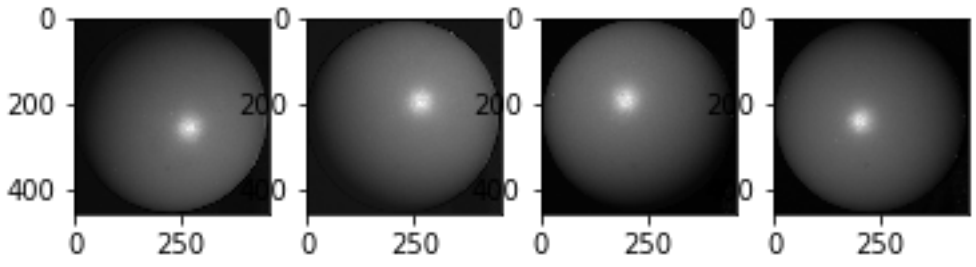
```
pickle_in = open("specular_sphere.pickle", "rb")
#data = pickle.load(pickle_in)
data = pickle.load(pickle_in, encoding="latin1")

# sample input

def rgb_range(image):
    image = (image - image.min()) / (image.max() - image.min())
    return image

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

print('The original image (in RGB colorspace).')
plt.figure(1)
for i in range (images.shape[0]):

    images[i] = rgb_range(images[i])
    plt.subplot(1, 4, i+1)
    plt.imshow(images[i], cmap = "gray")
plt.show()


S = np.zeros(images.shape[:-1])
G = np.zeros(images.shape[:-1])

print('The recovered  S channel of the image.')
plt.figure(2)
for j in range (images.shape[0]):
    S[j], G[j] = RGBToSUV(images[j], np.hstack((data["c"][0][0], data["c"][1][0]
,data["c"][2][0])))
    plt.subplot(1, 4, j+1)
    plt.imshow(S[j], cmap = "gray")
plt.show()

print('The recovered diffuse part of the image')

plt.figure(3)


for j in range (images.shape[0]):
    plt.subplot(1, 4, j+1)
    plt.imshow(G[j], cmap = "gray")
plt.show()
```

```
#image = (image - image.min()) / (image.max() - image.min())
```

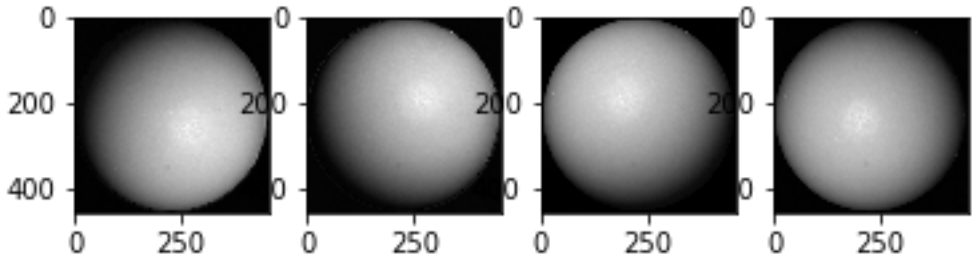
The original image (in RGB colorspace).



The recovered S channel of the image.



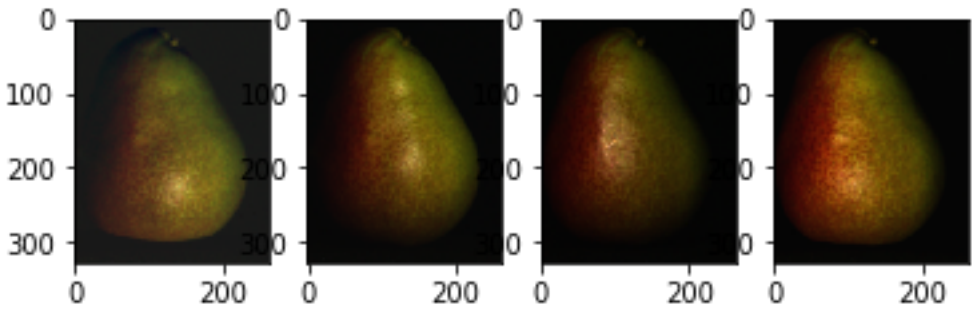
The recovered diffuse part of the image



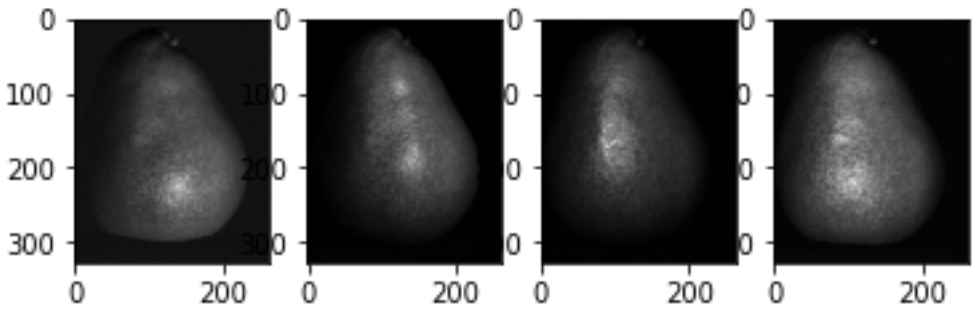
In [13]:

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

# sample input

def rgb_range(image):
    image = (image - image.min()) / (image.max() - image.min())
    return image

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

print('The original image (in RGB colorspace).')
plt.figure(1)
for i in range (images_p.shape[0]):

    images_p[i] = rgb_range(images_p[i])
    plt.subplot(1, 4, i+1)
    plt.imshow(images_p[i], cmap = "gray")
plt.show()


S_p = np.zeros(images_p.shape[:-1])
G_p = np.zeros(images_p.shape[:-1])

print('The recovered  S channel of the image.')
plt.figure(2)
for j in range (images_p.shape[0]):
    S_p[j], G_p[j] = RGBToSUV(images_p[j], np.hstack((data["c"][0][0], data["c"]
[1][0],data["c"][2][0])))
    plt.subplot(1, 4, j+1)
    plt.imshow(S_p[j], cmap = "gray")
plt.show()

print('The recovered diffuse part of the image')

plt.figure(3)


for j in range (images_p.shape[0]):
    plt.subplot(1, 4, j+1)
    plt.imshow(G_p[j], cmap = "gray")
plt.show()
```

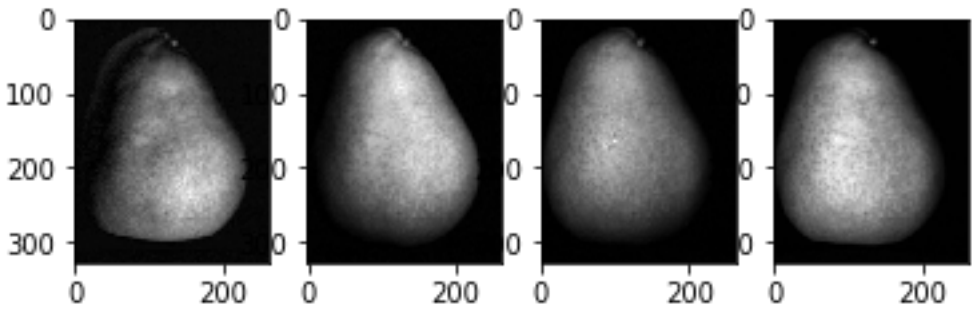
The original image (in RGB colorspace).



The recovered S channel of the image.



The recovered diffuse part of the image



Part 3: [4 pts]

Combine parts 1 and 2 by running your photometric stereo code on the diffuse components of the specular sphere and pear images.

For comparison, run your photometric stereo code on the original images (converted to grayscale) as well. You should notice erroneous "bumps" in the resulting reconstructions, as a result of violating the Lambertian assumption.

For each specular sphere and pear image sets, using all the four images, include:

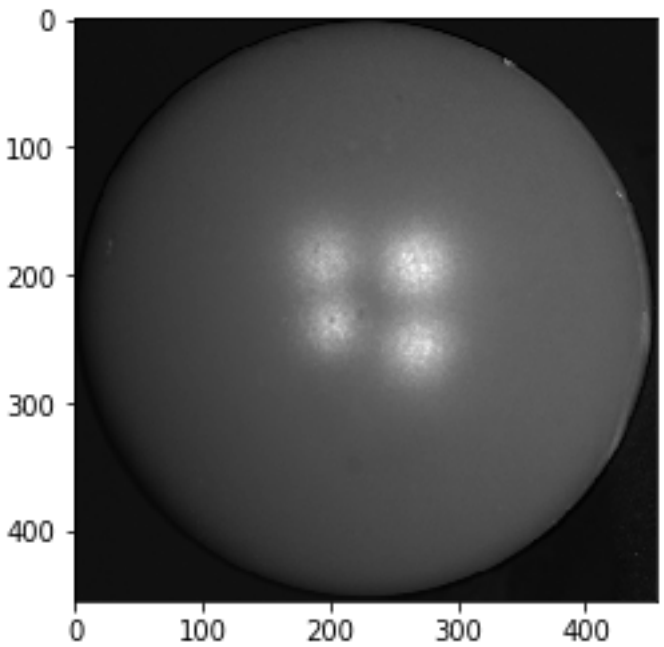
1. The estimated albedo map (original and diffuse)
2. The estimated surface normals (original and diffuse) by showing both
    A. Needle map, and
    B. Three images showing components of surface normal
3. A wireframe of depth map (original and diffuse)

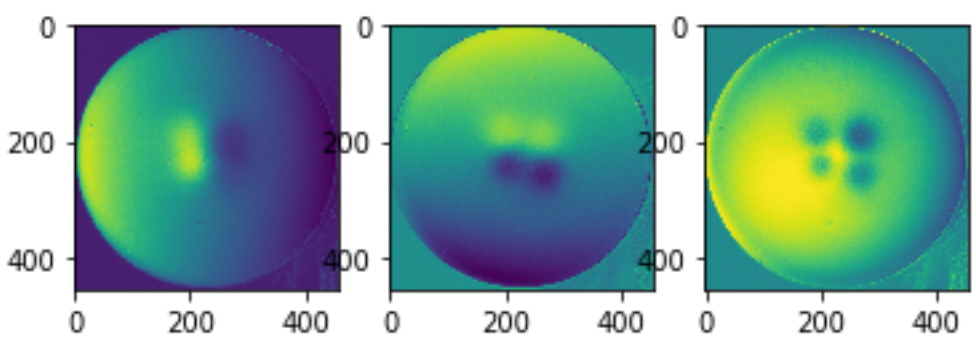
In [14]:

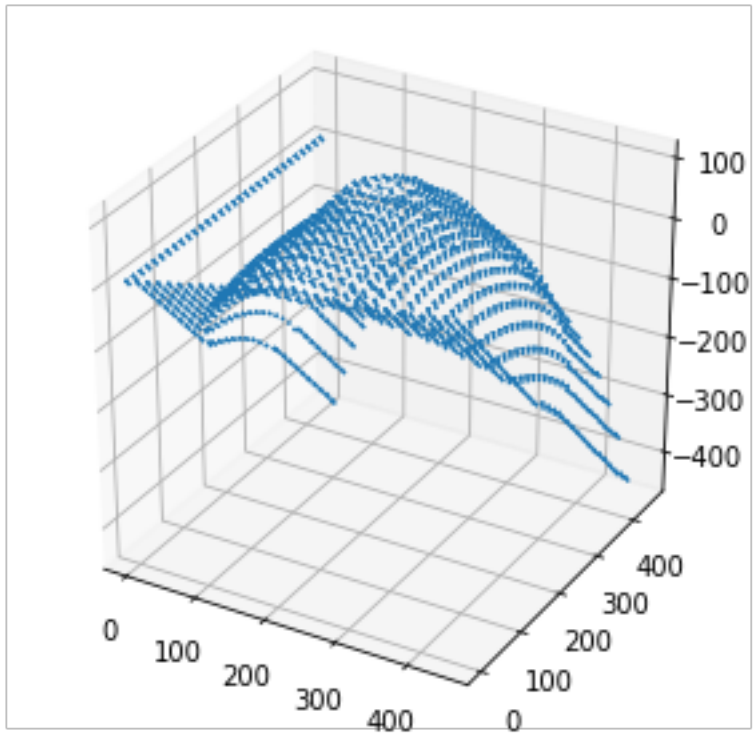
```
# ------------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
```

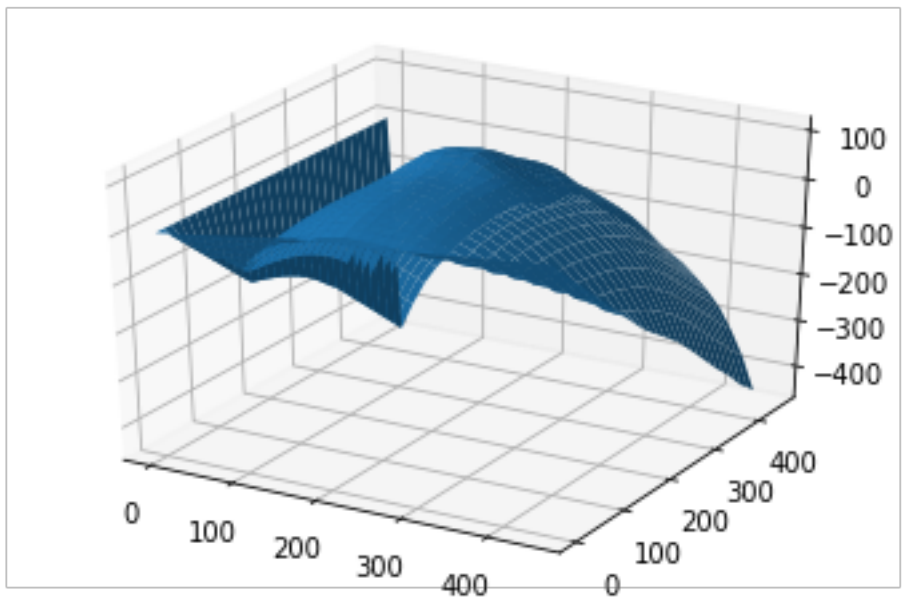
```
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals and depth maps.
# ------------------------------------------------------------------------
from mpl_toolkits.mplot3d import Axes3D

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


#mask = np.ones(data["im1"].shape)
mask = np.ones(S[0].shape)
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

images_g_s = np.zeros(images.shape[0:3])
for i in range (images.shape[0]):
    images_g_s[i] = rgb2gray(images[i])

albedo, normals_s, depth, horn = photometric_stereo(images_g_s, lights, mask)

# ------------------------------------------------------------------------
# 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.
# ------------------------------------------------------------------------

# 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_s[..., 0])
ax2 = figure.add_subplot(132)
ax2.imshow(normals_s[..., 1])
ax3 = figure.add_subplot(133)
ax3.imshow(normals_s[..., 2])
plt.show()

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals_s)[0], 15),
                      np.arange(0,np.shape(normals_s)[1], 15),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals_s[..., 0][::stride,::-stride].T
NY = normals_s[..., 1][::-stride,::stride].T
```

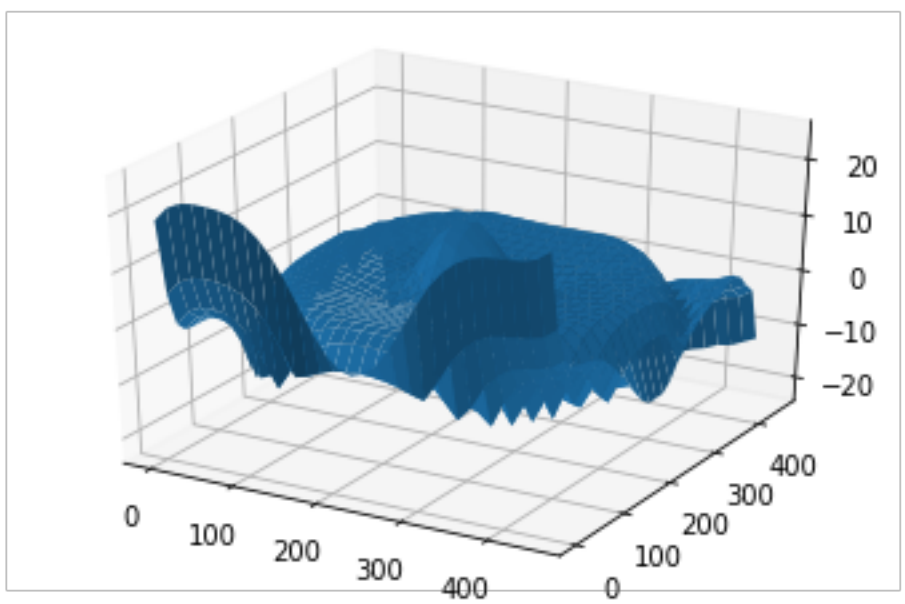
```
NZ = normals_s[..., 2][::stride,::stride].T

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

# plotting wireframe depth map
H = depth[::stride,::stride]
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y, H.T)
plt.show()

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











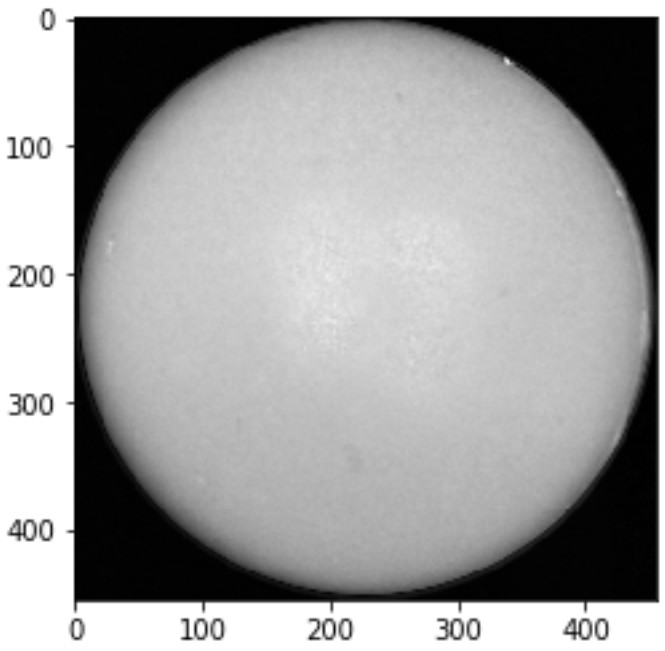
In [15]:

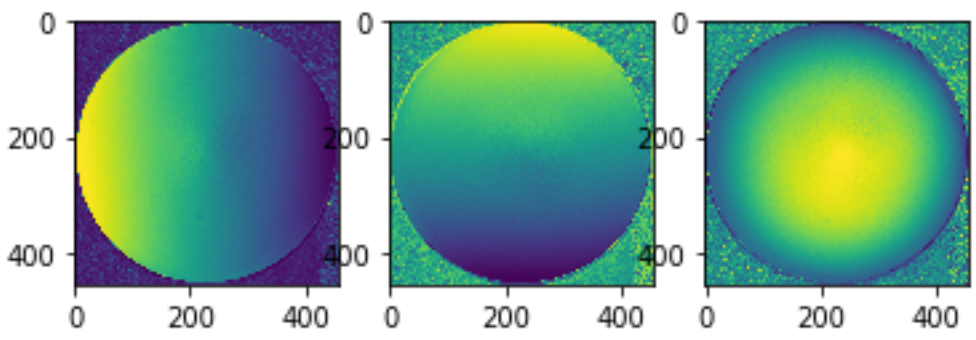
```
# ------------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals and depth maps.
```

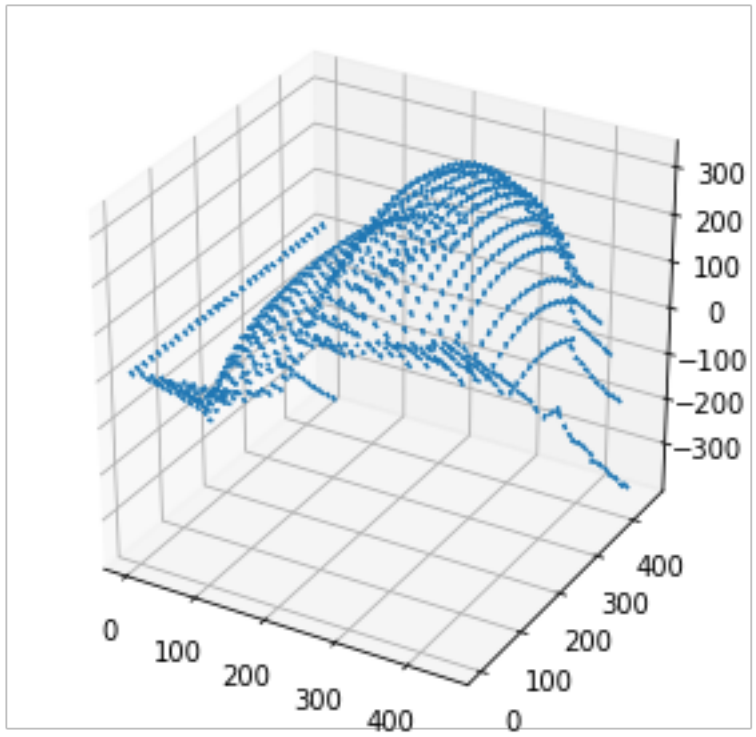
```
# ------------------------------------------------------------------------

from mpl_toolkits.mplot3d import Axes3D

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


#mask = np.ones(data["im1"].shape)
mask = np.ones(G[0].shape)

albedo, normals_s_g, depth, horn = photometric_stereo(G, lights, mask)

# ------------------------------------------------------------------------
# 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.
# ------------------------------------------------------------------------

# 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()
```

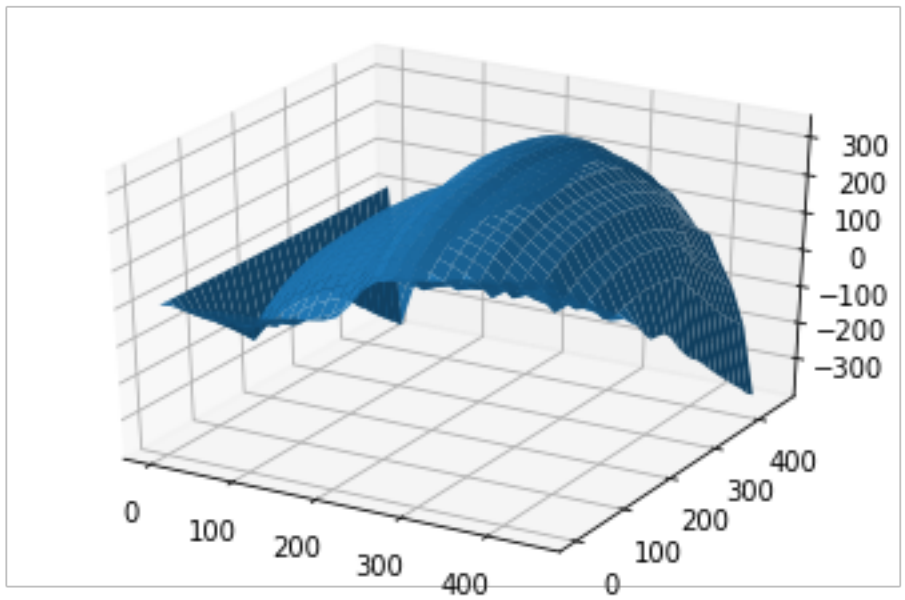
```
# showing normals as three separate channels
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.imshow(normals_s_g[..., 0])
ax2 = figure.add_subplot(132)
ax2.imshow(normals_s_g[..., 1])
ax3 = figure.add_subplot(133)
ax3.imshow(normals_s_g[..., 2])
plt.show()

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals_s_g)[0], 15),
                      np.arange(0,np.shape(normals_s_g)[1], 15),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals_s_g[..., 0][::stride,::-stride].T
NY = normals_s_g[..., 1][::-stride,::stride].T
NZ = normals_s_g[..., 2][::stride,::stride].T
fig = plt.figure(figsize=(5, 5))
ax = fig.gca(projection='3d')
ax.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()

# plotting wireframe depth map
```

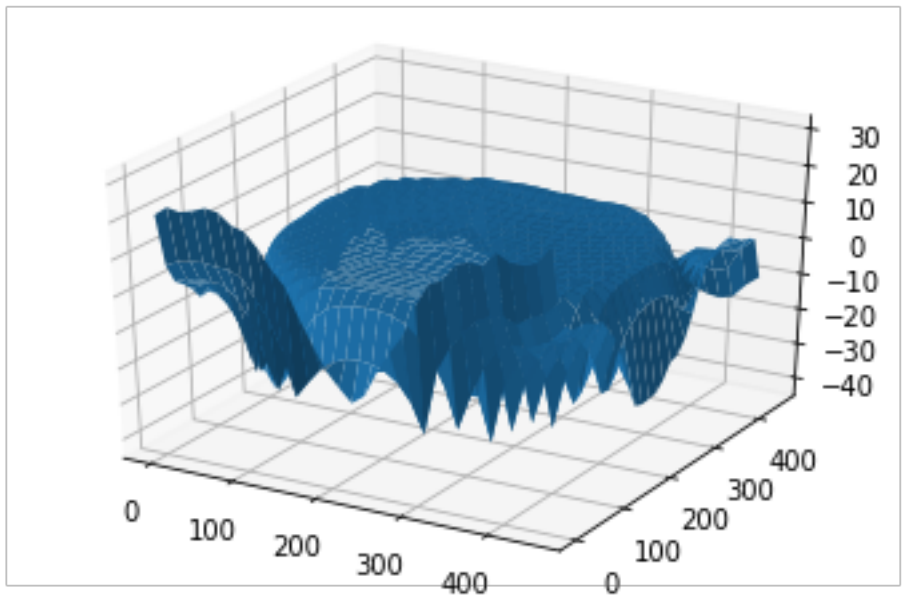
```
H = depth[::stride,::stride]

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

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






300
200
100
0
−100
−200
−300
0
100
200
300
400
0
100
200
300
400


300
200
100
0
−100
−200
−300
0
100
200
300
400
0
100
200
300
400


30
20
10
0
−10
−20
−30
−40
0
100
200
300
400
0
100
200
300
400

In [16]:

```
images.shape
```

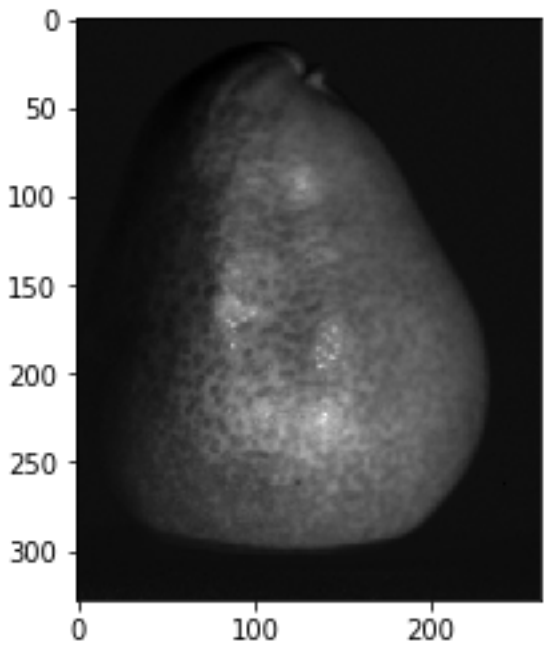
Out[16]:

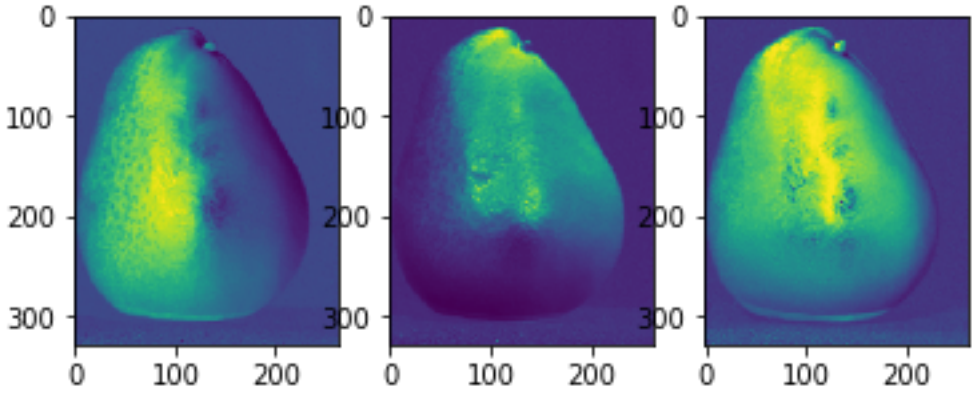
```
(4, 455, 455, 3)
```

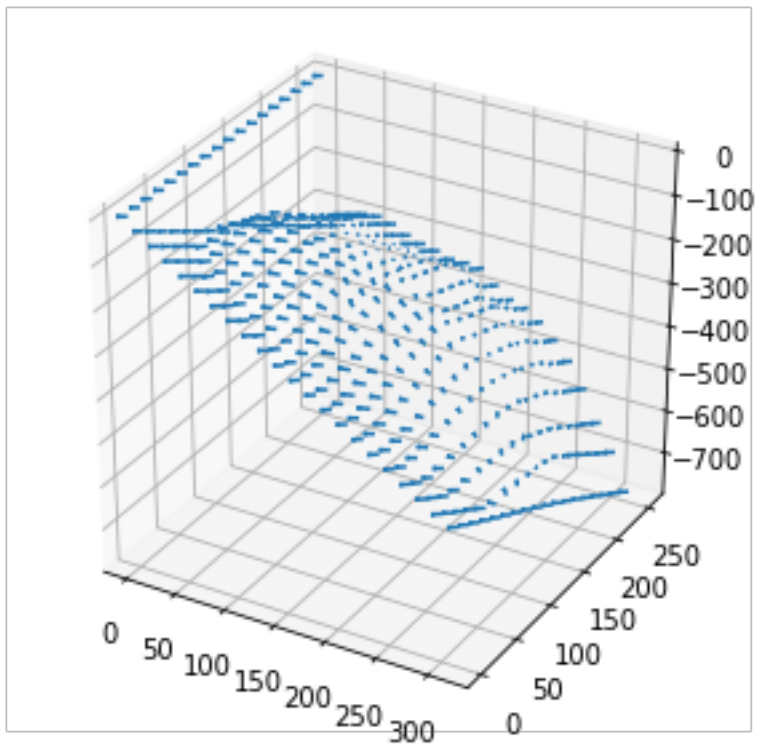
In [17]:

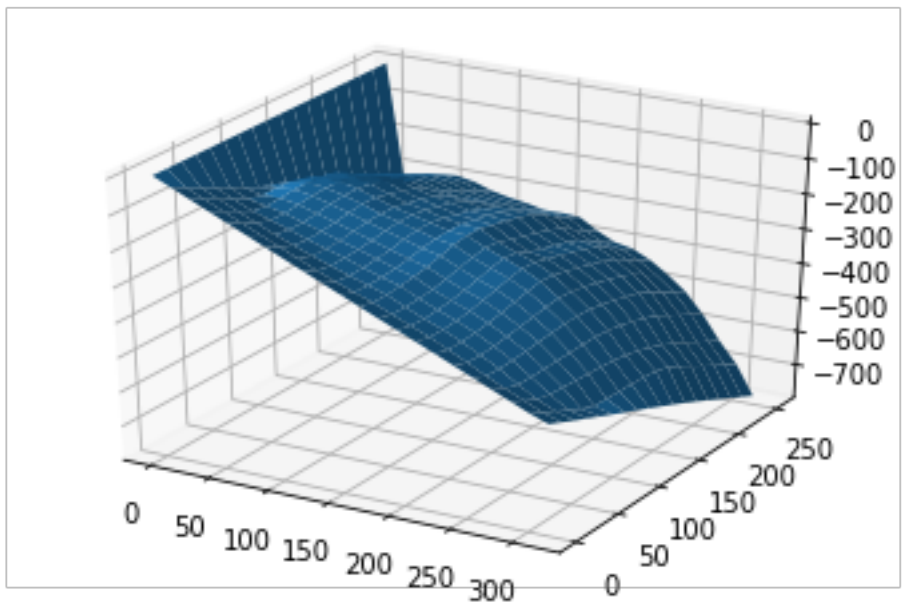
```
# ----------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals and depth maps.
# ----------------------------------------------------------------------
from mpl_toolkits.mplot3d import Axes3D

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

images_g = np.zeros(images_p.shape[0:3])

for i in range (images_p.shape[0]):
    images_g[i] = rgb2gray(images_p[i])

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



#mask = np.ones(data["im1"].shape)
mask = np.ones(S_p[0].shape)

albedo, normals_p, depth, horn = photometric_stereo(images_g, lights, mask)

# ----------------------------------------------------------------------
# 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.
# ----------------------------------------------------------------------

# 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)
```

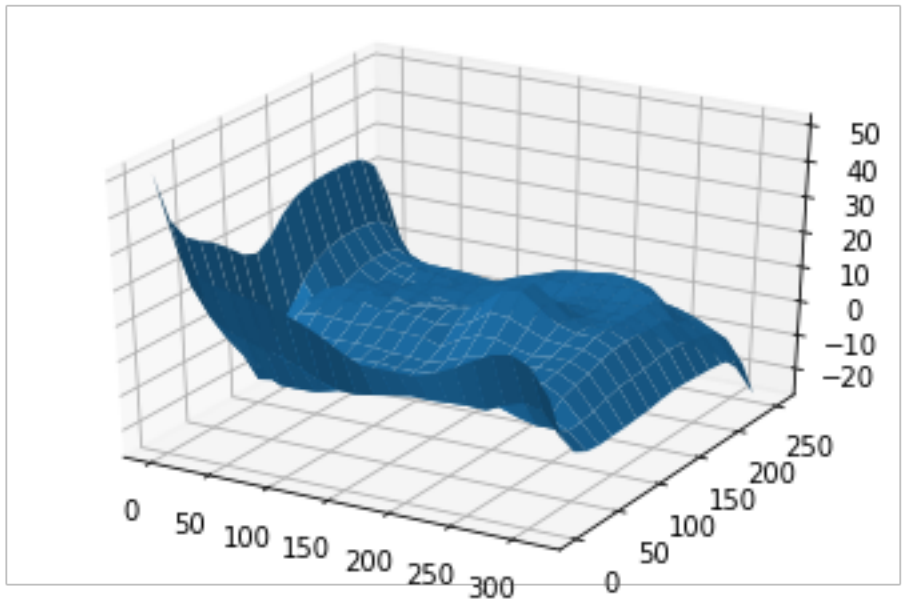
```
ax1.imshow(normals_p[..., 0])

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

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals_p)[0], 15),
                      np.arange(0,np.shape(normals_p)[1], 15),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals_p[..., 0][::stride,::-stride].T
NY = normals_p[..., 1][::-stride,::stride].T
NZ = normals_p[..., 2][::stride,::stride].T
fig = plt.figure(figsize=(5, 5))
ax = fig.gca(projection='3d')
ax.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()

# plotting wireframe depth map
H = depth[::stride,::stride]
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y, H.T)
plt.show()

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




0
100
200
300
0
100
200


0
−100
−200
−300
−400
−500
−600
−700
0
50
100
150
200
250
300
0
50
100
150
200
250

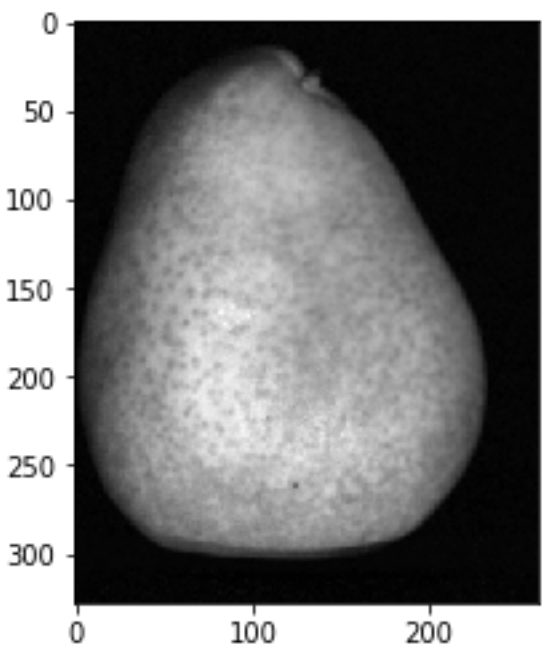

0
−100
−200
−300
−400
−500
−600
−700
0
50
100
150
200
250
300
0
50
100
150
200
250

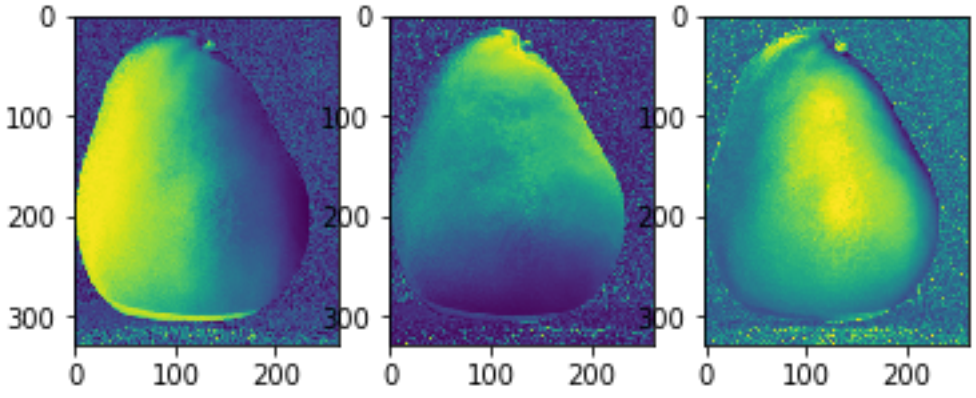


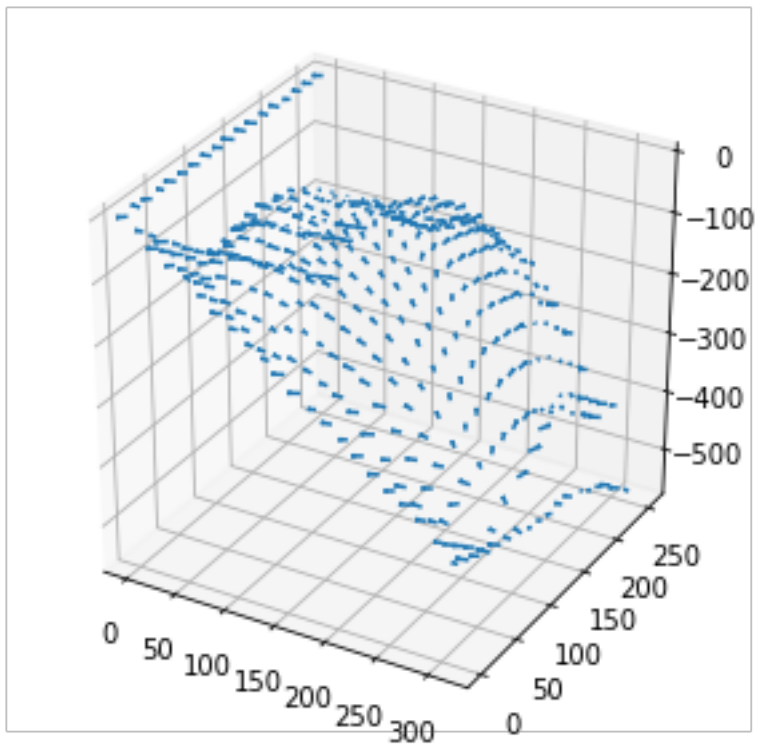
In [18]:

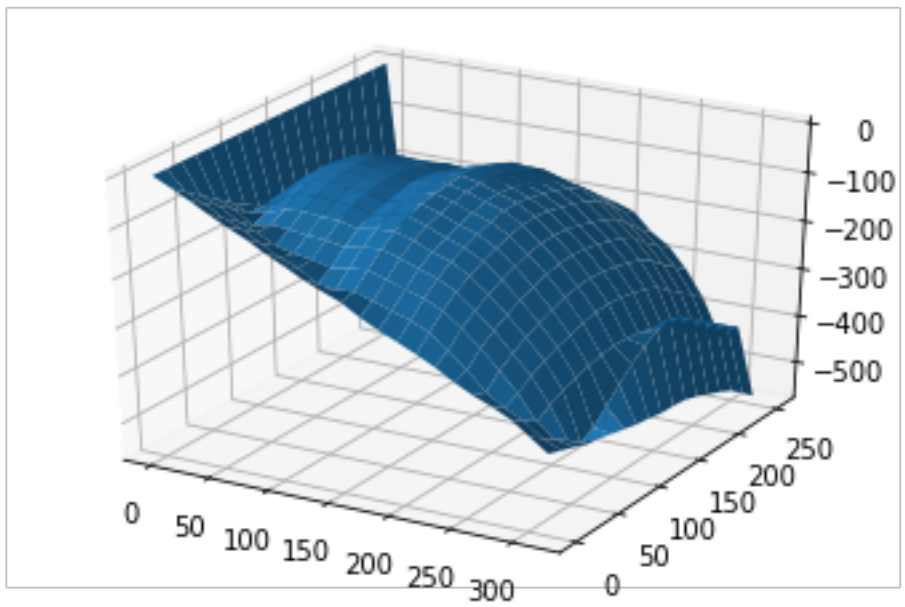
```
# ------------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals and depth maps.
# ------------------------------------------------------------------------
from mpl_toolkits.mplot3d import Axes3D

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


#mask = np.ones(data["im1"].shape)
mask = np.ones(G_p[0].shape)

albedo, normals_p_g, depth, horn = photometric_stereo(G_p, lights, mask)

# ------------------------------------------------------------------------
# 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.
# ------------------------------------------------------------------------

# 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_p_g[..., 0])
ax2 = figure.add_subplot(132)
```

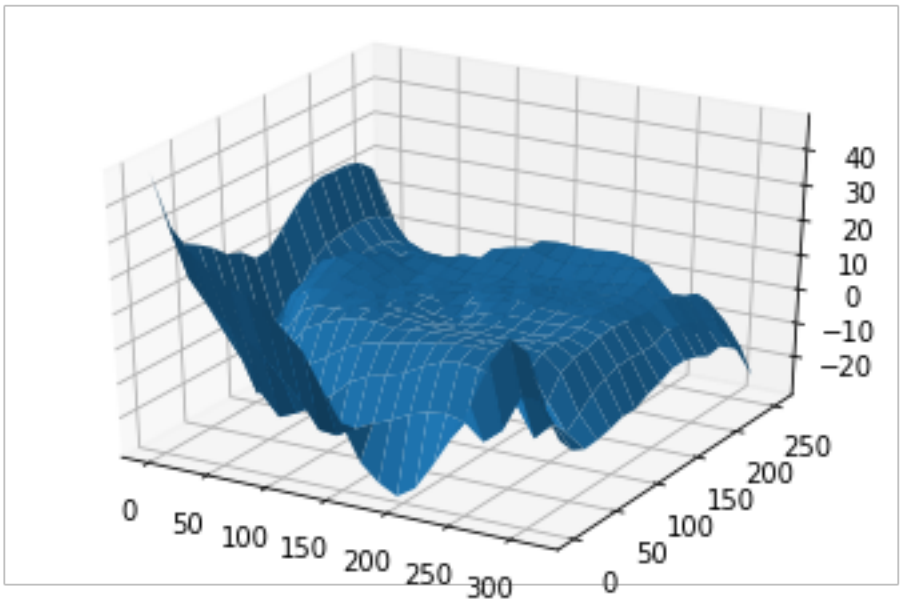
```
ax2.imshow(normals_p_g[..., 1])
ax3 = figure.add_subplot(133)
ax3.imshow(normals_p_g[..., 2])
plt.show()

# showing normals as quiver
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals_p_g)[0], 15),
                      np.arange(0,np.shape(normals_p_g)[1], 15),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals_p_g[..., 0][::stride,::-stride].T
NY = normals_p_g[..., 1][::-stride,::stride].T
NZ = normals_p_g[..., 2][::stride,::stride].T
fig = plt.figure(figsize=(5, 5))
ax = fig.gca(projection='3d')
ax.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()

# plotting wireframe depth map
H = depth[::stride,::stride]
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y, H.T)
plt.show()

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




0
100
200
300
0
100
200


0
−100
−200
−300
−400
−500
0
50
100
150
200
250
300
0
50
100
150
200
250


0
−100
−200
−300
−400
−500
0
50
100
150
200
250
300
0
50
100
150
200
250



Problem 6: Surface Rendering [10 pts]

In this portion of the assignment we will be exploring different methods of approximating local illumination of objects in a scene. As discovered in the photometeric stereo portion of this homework, we know that different light models work better with different view, illumination sources and materials. This last section of the homework will be an exercise in rendering surfaces. Here, you need use the surface normals from Part 3 of Problem 5 to calculate the image intensity of the specular sphere and pear, with various light sources, different materials, and using a number of illumination models. For the sake of simplicity, multiple reflections of light rays, and occlusion of light rays due to object/scene can be ignored.

Data

The surface normals of the specular sphere and the pear from Part 3 of Problem 5. For comparison, You should display the rendering results for both normals calculated from the original image and the diffuse components.

Assume that the albedo map is uniform.

Lambertian Illumination

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

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

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

Phong Illumination

One major drawback of Lambertian illumination is that it only considers the diffuse light in its calculation of brightness intensity. One other major component to illumination rendering is the specular component. The specular reflectance is the component of light that is reflected in a single direction, as opposed to all directions, which is the case in diffuse reflectance. One of the most used models to compute surface brightness with specular components is the Phong illumination model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Phong shading also considers the material in the scene which is characterized by four values: the ambient reflection constant (k_a), the diffuse reflection constant (k_d), the specular reflection constant (k_s) and α the Phong constant, which is the 'shininess' of an object. Furthermore, since the specular component produces 'rays', only some of which would be observed by a single observer, the observer's viewing direction ($\mathbf{V}$) must also be known. For some scene with known material parameters with M light sources the light intensity $\mathbf{I}_{phong}$ on a surface with normal vector $\mathbf{N}$ seen from viewing direction $\mathbf{V}$ can be computed by:

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

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

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

Rendering

Please complete the following:

1. Write the function `lambertian()` that calculates the Lambertian light intensity given the light direction $\mathbf{L}$ with color and intensity $\mathbf{C}$ and $I_l = 1$, and normal vector $\mathbf{N}$. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the two lighting sources found in Table 1. *Note: You do not need to worry about material coefficients in this model.*
2. Write the function `phong()` that calculates the Phong light intensity given the material constants (k_a, k_d, k_s, α), $\mathbf{V} = (0, 0, 1)^\top$, $\mathbf{N}$ and some number of M light sources. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the sets of coefficients found in Table 2 with each light source individually, and both light sources combined.

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

Table 1: Light Sources

m	**Location**	**Color (RGB)**
1	$(-\frac{1}{3}, \frac{1}{3}, \frac{1}{3})^\top$	$(1, 1, 1)$
2	$(1, 0, 0)^\top$	$(1, .5, .5)$

Table 2: Material Coefficients

Mat.	k_a	k_d	k_s	α
1	0	0.1	0.75	5
2	0	0.5	0.1	5
3	0	0.5	0.5	10

Part 1. Lambertian model [4 pts]

In [19]:

```
#Combined_Light
def lambertian(normals, lights, color, intensity, mask=None):
    '''Your implementation'''
    image = np.ones((normals.shape[0], normals.shape[1], 3))
    for i in range (normals.shape[0]):
        for j in range (normals.shape[1]):
            b_0 = np.dot(normals[i,j],color[0])
            b_1 = np.dot(normals[i,j],color[1])
            B_0 = np.dot(lights[:,0],b_0)
            B_1 = np.dot(lights[:,1],b_1)
            image[i,j] = B_0+B_1
            if image[i,j][0] < 0:
                image[i,j][0] = 0
            elif image[i,j][1] < 0:
                image[i,j][1] = 0
            elif image[i,j][2] < 0:
                image[i,j][2] = 0

    image = rgb_range(image)
    return image
```

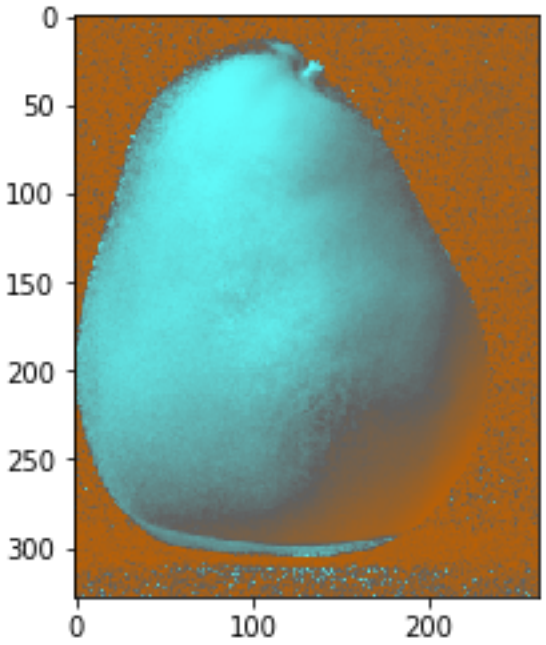
In [20]:

```
#Seperate light
def lambertian_1(normals, lights, color, intensity, mask=None):
    '''Your implementation'''
    image_1 = np.ones((normals.shape[0], normals.shape[1], 3))
    image = np.ones((normals.shape[0], normals.shape[1], 3))
    for i in range (normals.shape[0]):
        for j in range (normals.shape[1]):
            b_0 = np.dot(normals[i,j],color[0])
            b_1 = np.dot(normals[i,j],color[1])
            B_0 = np.dot(lights[:,0],b_0)
            B_1 = np.dot(lights[:,1],b_1)
            image[i,j] = B_0
            image_1[i,j] = B_1
            if image[i,j][0] < 0:
                image[i,j][0] = 0
            elif image[i,j][1] < 0:
                image[i,j][1] = 0
            elif image[i,j][2] < 0:
                image[i,j][2] = 0
    image =  rgb_range(image)
    image_1 =  rgb_range(image_1)

    return image, image_1
```

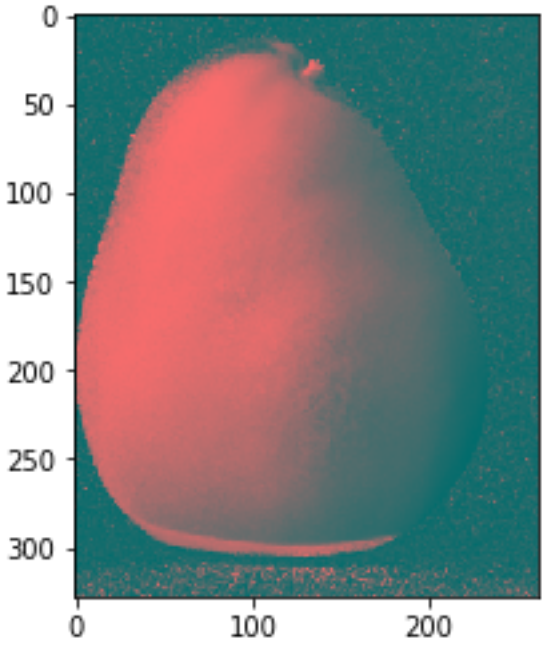
In [21]:

```
#Seperate light
lights = np.array([[-1/3,1/3,1/3],[1,0,0]]).T
color = np.array([[1,1,1],[1,0.5,0.5]])
intensity = 1
mask = np.ones(normals.shape[:-1])
pear_1,pear_2 = lambertian_1(normals_p_g, lights, color, intensity,mask)
print('Light_1')
plt.imshow(pear_1, cmap = "gray")
plt.show()
print('Light_2')
plt.imshow(pear_2, cmap = "gray")
plt.show()
```

Light_1



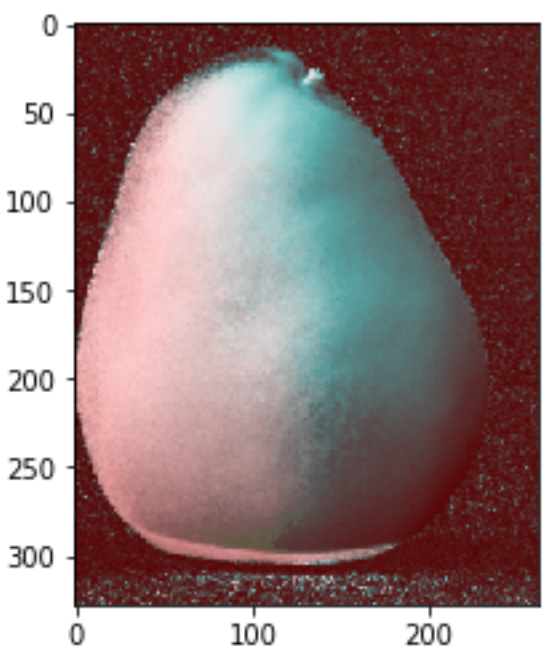
Light_2



In [22]:

```
# Combined_Lights
lights = np.array([[-1/3,1/3,1/3],[1,0,0]]).T
color = np.array([[1,1,1],[1,0.5,0.5]])
intensity = 1
mask = np.ones(normals.shape[:-1])
pear = lambertian(normals_p_g, lights, color, intensity,mask)
pear = rgb_range(pear)
print('Combined_Lights')
plt.imshow(pear, cmap = "gray")
plt.show()
```

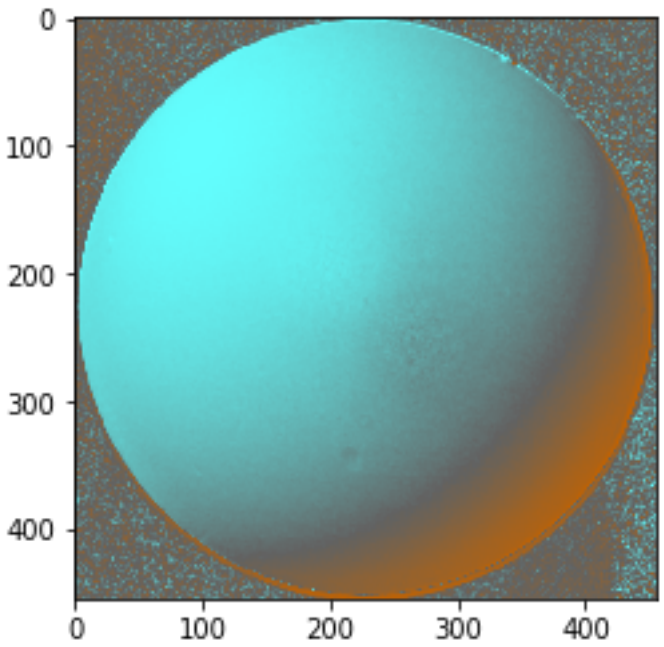
```
Combined_Lights
```



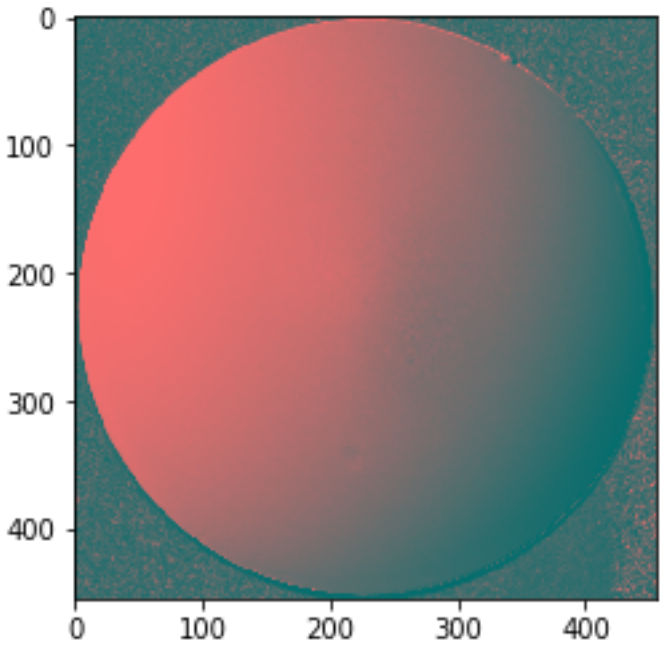
In [23]:

```
#Seperate_Light
lights = np.array([[-1/3,1/3,1/3],[1,0,0]]).T
color = np.array([[1,1,1],[1,0.5,0.5]])
mask = np.ones(normals_s_g.shape[:-1])
sphere_1,sphere_2 = lambertian_1(normals_s_g, lights, color, intensity,mask)
print('light_1')
plt.imshow(sphere_1, cmap = "gray")
plt.show()
print('light_2')
plt.imshow(sphere_2, cmap = "gray")
plt.show()
```

light_1



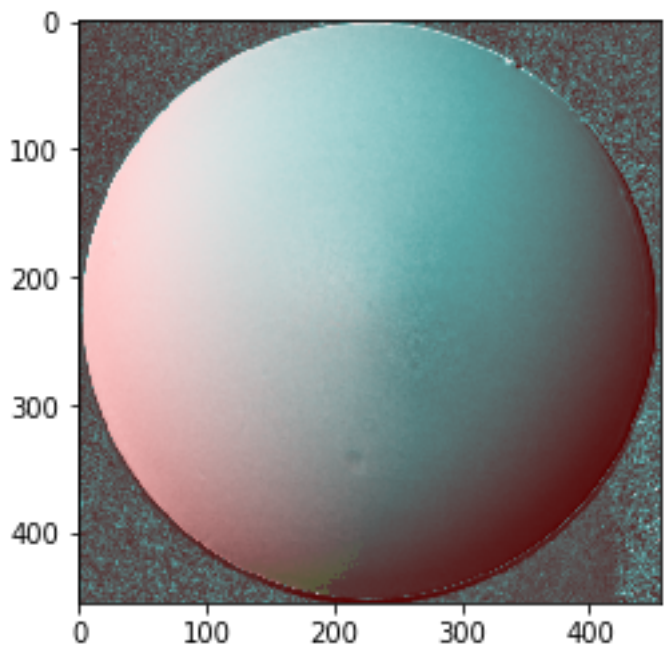
light_2



In [24]:

```
#Combined_Light
lights = np.array([[-1/3,1/3,1/3],[1,0,0]]).T
color = np.array([[1,1,1],[1,0.5,0.5]])
mask = np.ones(normals_s_g.shape[:-1])
sphere = lambertian(normals_s_g, lights, color, intensity,mask)
#sphere = rgb_range(sphere)
print('Combined_Light')
plt.imshow(sphere, cmap = "gray")
plt.show()
```

```
Combined_Light
```



Part 2. Phong model [6 pts]

In [25]:

```
normals_p.shape
```

Out[25]:

```
(328, 262, 3)
```

In [55]:

```
#Combined_light
def phong(normals, lights, color, material, view, mask):
    '''Your implementation'''
    image = np.zeros((normals.shape[0], normals.shape[1], 3))
    Rm = np.zeros((lights.shape[1],normals.shape[0],normals.shape[1],normals.sha
pe[2]))

    for k in range(Rm.shape[0]):
        for i in range (Rm.shape[1]):
            for j in range(Rm.shape[2]):
                Rm[k][i,j] = 2*np.dot(normals[i,j],np.dot(lights[:,k],normals[i,
j])) - lights[:,k]
                Im_d = material[1]*np.dot(lights[:,k],normals[i,j])*color[k]
                Im_s = material[2]*(np.dot(Rm[k][i,j],view))**material[3]*color[
k]
                #print(Rm[k][i,j])
                #print(Im_s.shape)
                #print(Im_d.shape)
                image[i,j] = image[i,j]+ Im_d + Im_s
                if image[i,j][0] < 0:
                    image[i,j][0] = 0
                elif image[i,j][1] < 0:
                    image[i,j][1] = 0
                elif image[i,j][2] < 0:
                    image[i,j][2] = 0

    image = rgb_range(image)
    return image
```

In [62]:

```
#Seperate_light
def phong_sep(normals, lights, color, material, view, mask):
    '''Your implementation'''
    image = np.zeros((lights.shape[1],normals.shape[0], normals.shape[1], 3))
    Rm = np.zeros((lights.shape[1],normals.shape[0],normals.shape[1],normals.sha
pe[2]))

    for k in range(Rm.shape[0]):
        for i in range (Rm.shape[1]):
            for j in range(Rm.shape[2]):
                Rm[k][i,j] = 2*np.dot(normals[i,j],np.dot(lights[:,k],normals[i,
j])) - lights[:,k]
                Im_d = material[1]*np.dot(lights[:,k],normals[i,j])*color[k]
                Im_s = material[2]*(np.dot(Rm[k][i,j],view))**material[3]*color[
k]
                image[k][i,j] = Im_d + Im_s
                if image[k][i,j][0] < 0:
                    image[k][i,j][0] = 0
                elif image[k][i,j][1] < 0:
                    image[k][i,j][1] = 0
                elif image[k][i,j][2] < 0:
                    image[k][i,j][2] = 0

        image[k] = rgb_range(image[k])
    return image
```

In [31]:

```
lights = np.array([[-1/3,1/3,1/3],[1,0,0]]).T
color = np.array([[1,1,1],[1,0.5,0.5]])
intensity = 1
view = np.array([[0,0,1]]).T
material = np.array([[0,0.1,0.75,5],[0,0.5,0.1,5],[0,0.5,0.5,10]])
```

In [75]:

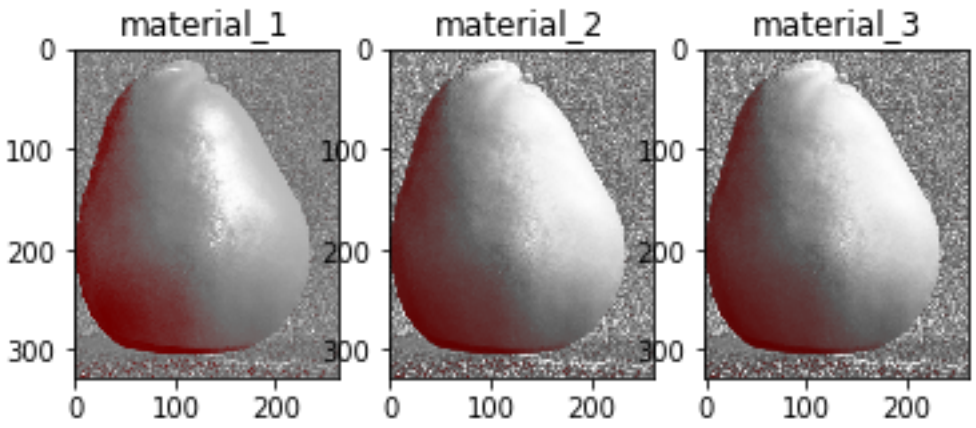
```
print('Seperate_Light_Pear')
img_phong_p_sep = np.zeros((material.shape[0],
                            lights.shape[1],
                            normals_p_g.shape[0],
                            normals_p_g.shape[1],
                            normals_p_g.shape[2]))
for i in range (material.shape[0]):
    img_phong_p_sep[i] = phong_sep(normals_p_g, lights, color, material[i], view
, mask)
```

```
Seperate_Light_Pear
```

In [67]:

```
#Light_1
print('Light_1_pear')
for i in range (material.shape[0]):
        plt.subplot(1,3,i+1)
        plt.title('material_%s'%(i+1))
        plt.imshow(img_phong_p_sep[i,0], cmap = "gray")
plt.show()
```

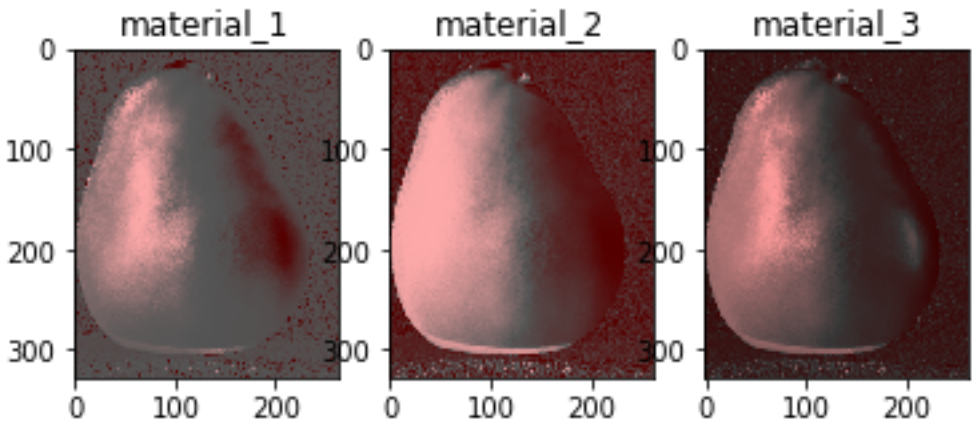
Light_1



In [68]:

```
print('Light_2_pear')
for i in range (material.shape[0]):
        plt.subplot(1,3,i+1)
        plt.title('material_%s'%(i+1))
        plt.imshow(img_phong_p_sep[i,1], cmap = "gray")
plt.show()
```

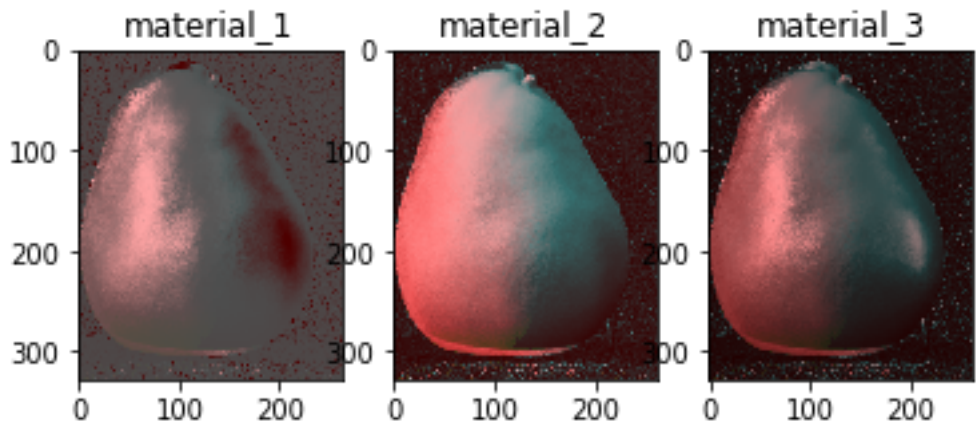
Light_2



In [50]:

```
print('Combined_Light')
img_phong_p = np.zeros((material.shape[0],
                        normals_p_g.shape[0],
                        normals_p_g.shape[1],
                        normals_p_g.shape[2]))
for i in range (material.shape[0]):
    img_phong_p[i] = phong(normals_p_g, lights, color, material[i], view, mask)
    plt.subplot(1,3,i+1)
    plt.title('material_%s'%(i+1))
    plt.imshow(img_phong_p[i], cmap = "gray")
plt.show()
```

Combined_Light



In [70]:

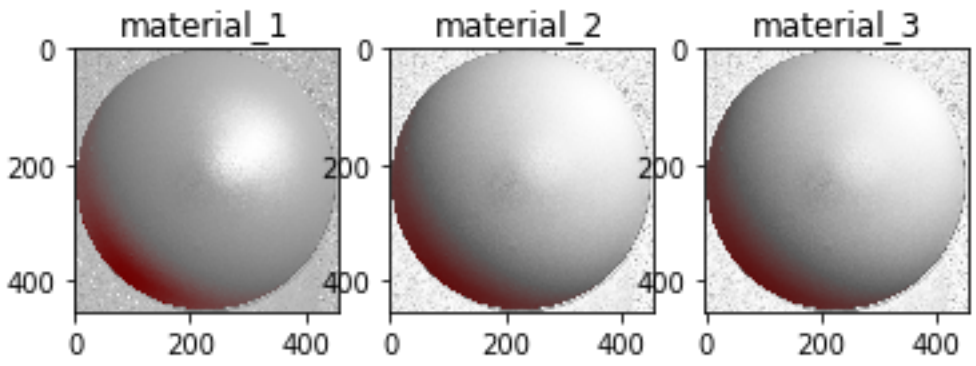
```
print('Seperate_Light_sphere')
img_phong_s_sep = np.zeros((material.shape[0],lights.shape[1],
                            normals_s_g.shape[0],
                            normals_s_g.shape[1],
                            normals_s_g.shape[2]))
for i in range (material.shape[0]):
    img_phong_s_sep[i] = phong_sep(normals_s_g, lights, color, material[i], view
, mask)
```

Seperate_Light_sphere

In [73]:

```
#Light_1
print('Light_1_Sphere')
for i in range (material.shape[0]):
        plt.subplot(1,3,i+1)
        plt.title('material_%s'%(i+1))
        plt.imshow(img_phong_s_sep[i,0], cmap = "gray")
plt.show()
```

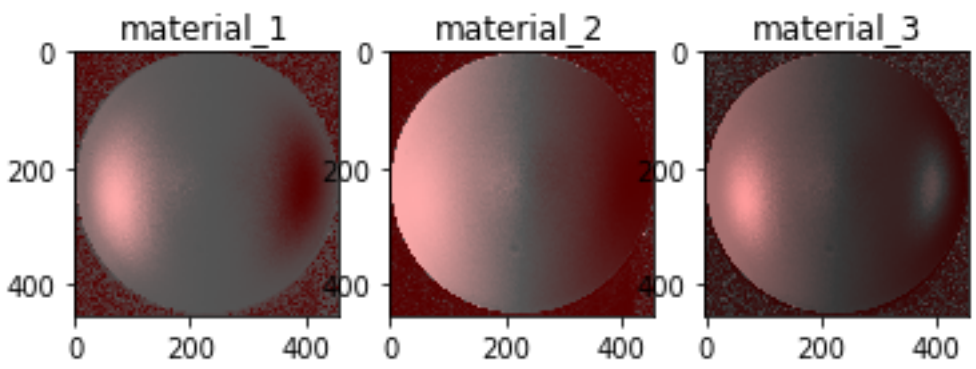
```
Light_1_Sphere
```



In [74]:

```
print('Light_2_Sphere')
for i in range (material.shape[0]):
        plt.subplot(1,3,i+1)
        plt.title('material_%s'%(i+1))
        plt.imshow(img_phong_s_sep[i,1], cmap = "gray")
plt.show()
```

```
Light_2_Sphere
```



In [47]:

```
#Combined lights_sphere
print('Combined_Light_Sphere')
img_phong_s = np.zeros((material.shape[0],normals_s_g.shape[0],normals_s_g.shape
[1],normals_s_g.shape[2]))
for i in range (material.shape[0]):
    img_phong_s[i] = phong_sep(normals_s_g, lights, color, material[i], view, ma
sk)
    plt.subplot(1,3,i+1)
    plt.title('material_%s'%(i+1))
    plt.imshow(img_phong_s[i], cmap = "gray")
plt.show()
```

```
Combined_Light_Sphere
```