Photometric Stereo

March 15, 2023

[ ]:
```
# Import the essential libraries
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import os

from skimage import io
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.cluster import KMeans

from mpl_toolkits.mplot3d import Axes3D

from skimage import data, segmentation, color
from skimage.future import graph
from skimage.color import rgb2gray

import warnings
warnings.filterwarnings("ignore")
```

This is a Python script that imports several libraries and modules for data visualization, image processing, and machine learning. The specific libraries and modules imported are:

matplotlib.pyplot - used for data visualization, specifically for creating plots and charts.

pandas - used for data manipulation and analysis, specifically for working with data in a tabular format.

numpy - used for mathematical computations and array manipulations in Python. os - used for interacting with the operating system, specifically for accessing and manipulating files and directories.

skimage.io - used for reading and writing image files.

sklearn.cluster.MeanShift - used for the MeanShift clustering algorithm in machine learning.

sklearn.cluster.KMeans - used for the KMeans clustering algorithm in machine learning.

mpl_toolkits.mplot3d.Axes3D - used for 3D data visualization.

skimage.data - used for loading example images.

skimage.segmentation - used for performing image segmentation.

skimage.color - used for color space conversions and color manipulation.

skimage.future.graph - used for graph-based image segmentation.

skimage.color.rgb2gray - used for converting color images to grayscale.

Additionally, the script imports the warnings module and sets it to ignore warnings during runtime.

Overall, this code appears to be setting up the necessary libraries and modules for performing image processing and segmentation tasks using machine learning techniques.

1 SECOND PART

[ ]:
```
########################################################################
# You should divide the image intensity by 255 to scale them into
# [0, 1] before any computation.
########################################################################
def reshape_image(first_image, second_image, third_image, fourth_image):
  """
  params:
  original input images

  TO DO:
  Divide the images by 255.0

  return: normalized images.
  """
  first_image = first_image / 255.0
  second_image = second_image / 255.0
  third_image = third_image / 255.0
  fourth_image = fourth_image / 255.0

  return first_image, second_image, third_image, fourth_image
```

This is a function called reshape_image that takes in four input images as parameters. The function aims to normalize the images by dividing their intensity values by 255 to scale them into the range of [0, 1].

The function starts by dividing each input image by 255.0, which is a common method for scaling pixel intensity values to a normalized range. The division operation is performed using the / operator in Python.

Finally, the function returns the four normalized images.

Overall, this function can be useful for preparing input images for further processing or analysis that may require pixel intensity values to be in the range of [0, 1].

[ ]:
```
def albedos_and_surface_normals(first_image, second_image, third_image,␣
↪fourth_image, light_source_direction):
    """
    params:
```

```
    All the original images and light souce directions.

    TO DO:
    calculate pseudoinverse matrix, albedos, and surface normals.

    return:
    albedos and surface normals.
    """
    albedos = np.zeros(first_image.shape)
    surface_normals = []
    light_intensity = 1
    camera_scaling_constant = 1
    for x in range(first_image.shape[0]):
        for y in range(first_image.shape[1]):
            intensities = [first_image[x, y], second_image[x, y],␣
↪third_image[x, y], fourth_image[x, y]]

            ␣
↪#########################################################################################
            # Assume that the intensity is zero for points in shadow and use␣
↪the method discussed during the
            # class to zeros out any equations from shadow points.
            ␣
↪#########################################################################################

            intensity_matrix = np.diag(intensities)

            ␣
↪#########################################################################################
            # You need to solve a linear equation Ax = b using pseudo-inverse.␣
↪The solution is x = A+b where A+,
            # the pseudo-inverse matrix of A, is defined as: A+ = (ATA)-1AT.
            ␣
↪#########################################################################################

            intensity_matrix_and_intensities = np.dot(intensity_matrix,␣
↪intensities)
            intensity_and_light_source_direction = np.dot(intensity_matrix,␣
↪light_source_direction)

            ␣
↪#########################################################################################
            # In all images, assume that the strength of the light sources was␣
↪a constant 1.0.
            # • Assume that the camera parameter k is 1, i.e. each row of␣
↪matrix S consists of the corresponding
```

```
            # light source direction.
            # Assuming Scaling constant k of the linear camera and Image␣
↪intensity to be 1, that is, kLj = 1
          ␣
↪#########################################################################################

            intensity_and_light_source_direction =␣
↪intensity_and_light_source_direction * light_intensity \
            * camera_scaling_constant

            first_component = np.dot(intensity_and_light_source_direction.T,␣
↪intensity_and_light_source_direction)

            pseudoinverse_matrix = np.dot(np.dot(np.linalg.
↪inv(first_component), \
                                                  ␣
↪intensity_and_light_source_direction.T), \
                                            intensity_matrix_and_intensities)
            albedo = np.linalg.norm(pseudoinverse_matrix)

            surface_normal = np.dot((1/albedo), pseudoinverse_matrix)
            albedos[x, y] = albedo
            surface_normals.append(surface_normal)

    return albedos, surface_normals
```

This is a Python function called albedos_and_surface_normals that takes in four input images and a light source direction as parameters. The function calculates the albedos and surface normals for each pixel in the input images.

The function starts by initializing two empty arrays for the albedos and surface normals. It also sets the light intensity and camera scaling constant to 1, which are used in later calculations.

The function then loops through each pixel in the input images using nested for loops. For each pixel, it extracts the intensity values from the four input images and creates an intensity matrix. If the pixel is in shadow, the function zeros out any equations from shadow points.

Next, the function solves a linear equation Ax = b using the pseudoinverse method. It calculates the pseudoinverse matrix using the formula A+ = (ATA)–1AT, where A is the intensity matrix and b is the intensity values. The intensity and light source direction are also multiplied by the light intensity and camera scaling constant.

The function then calculates the albedo and surface normal for each pixel using the pseudoinverse matrix. The albedo is the norm of the pseudoinverse matrix, and the surface normal is the pseudoinverse matrix divided by the albedo.

Finally, the function appends the surface normal to the surface_normals array and sets the albedo for the pixel in the albedos array. The function returns both the albedos and surface_normals arrays.

Overall, this function can be useful for calculating the albedos and surface normals of an object from multiple images taken with different lighting directions.

[ ]:
```
def reshape_surface_normal_components(x, y, z, dz_dx, dz_dy, image_shape):
    """
    params:
    x, y, z are surface normals components.
    dz_dx = slight change in height in x direction.
    dz_dy = slight change in height in y direction.

    TO DO:
    to change the shapes of all the parameters to the shape of original image.

    return:
    all the reshaped parameters.
    """
    x = x.reshape(-1, image_shape)
    y = y.reshape(-1, image_shape)
    z = z.reshape(-1, image_shape)
    dz_dx = dz_dx.reshape(-1, image_shape)
    dz_dy = dz_dy.reshape(-1, image_shape)
    return x, y, z, dz_dx, dz_dy
```

This function takes in five parameters: x, y, and z components of surface normals, the slight change in height in the x direction (dz_dx), slight change in height in the y direction (dz_dy), and the shape of the original image. It then reshapes all the input parameters to the shape of the original image and returns them.

The purpose of this function is to make sure that all the parameters have the same shape as the original image so that they can be easily processed further.

[ ]:
```
def surface_normal_components(surface_normals, image_shape):
    """
    params:
    surface normals
    image_shape: shape of the original image

    TO DO:
    1. get each component of surface normal
    2. calculate small change in z with result to x and ydirection
    3. reshape all the above components accourding to original image

    return:
    return reshaped components
    """
    x = np.array(pd.DataFrame(surface_normals).values.tolist())[:, 0]
    y = np.array(pd.DataFrame(surface_normals).values.tolist())[:, 1]
    z = np.array(pd.DataFrame(surface_normals).values.tolist())[:, 2]
```

```
    dz_dx = x / z
    dz_dy = y / z

    x, y, z, dz_dx, dz_dy = reshape_surface_normal_components(x, y, z, dz_dx,␣
↪dz_dy, image_shape)

    return x, y, z, dz_dx, dz_dy
```

This code defines a function named surface_normal_components that takes in two arguments: surface_normals and image_shape.

The purpose of the function is to extract the individual components of the surface normals (x, y, z), calculate the small change in z with respect to x and y direction, and then reshape all the above components according to the shape of the original image.

The function first converts the surface_normals input, which is a list of lists, into separate arrays for x, y, and z using NumPy. Then, it calculates the small change in z with respect to x and y direction by dividing the x and y arrays by the z array.

The function then calls another function named reshape_surface_normal_components to reshape all the arrays to the shape of the original image.

Finally, the function returns the reshaped arrays: x, y, z, dz_dx, and dz_dy.

[ ]:
```
def surface_normals_to_depth(z, dz_dx, dz_dy, reshaping_value = 1):
    """
    params:
    z: zth component of surface normal
    dz_dx: slight change in z with respect to x direction
    dz_dy: slight change in z with respect to y
    """
    depth = z * reshaping_value


  ␣
↪#############################################################################################
    # Set the integration constant - the height of the surface at the start␣
↪point - as 0.
  ␣
↪#############################################################################################
    depth[0, 0] = 0

    for y in range(1, depth.shape[0]):
        depth[y, 0] = depth[y-1, 0] - dz_dy[y, 0]

    for y in range(0, depth.shape[0]):
        for x in range(1, depth.shape[1]):
            depth[y, x] = depth[y, x-1] - dz_dx[y, x]
```

```
    return depth
```

This code defines a function called surface_normals_to_depth which takes in three parameters: z, dz_dx, and dz_dy.

The purpose of this function is to convert the surface normals components into a depth map. The surface normals components include the z component and two components representing slight changes in z with respect to x and y directions.

The function first multiplies the z component by a scaling factor (reshaping_value) and assigns the value of zero to the height of the surface at the starting point (the top-left corner of the image).

Then, two loops are used to iteratively calculate the height of each pixel in the image based on the heights of the neighboring pixels and the corresponding slight changes in z with respect to x and y directions. The loop starts from the second row and second column of the depth map and iterates over each pixel in row-major order.

The function returns the calculated depth map.

[ ]:
```
def display_and_save_image(image, image_name):
  plt.axis('off')
  plt.title(f'{os.path.splitext(image_name)[0]}')
  io.imshow(image, cmap = 'gray')
  #plt.imshow(image, cmap = 'gray')
  plt.savefig(f'{image_name}')
```

This code defines a function display_and_save_image that displays an input image and saves it to a file. The function takes two parameters:

image: a numpy array representing the image to display and save image_name: a string representing the filename to use when saving the image The function first turns off the axes using plt.axis('off') and sets the title of the plot using plt.title(f'{os.path.splitext(image_name)[0]}').

It then displays the image using io.imshow(image, cmap = 'gray'), where io is imported from the skimage package and cmap = 'gray' sets the color map to grayscale.

Finally, the function saves the image to a file with the specified name using plt.savefig(f'{image_name}'), where plt is imported from the matplotlib.pyplot package.

[ ]:
```
def display_and_save_height_map_3d(x, y, z, title):
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  ax.plot_surface(x, y, z)
  plt.title(f'{os.path.splitext(title)[0]}')
  plt.savefig(f'{title}')
  plt.show()
```

This is a Python function that takes in three input parameters: x, y, and z, representing the x, y, and z components of a height map, respectively, and a title string for the plot.

The function creates a 3D plot of the height map using matplotlib's plot_surface() function and saves the plot as an image file with the same name as the title parameter. The plot is then displayed using plt.show().

The ax variable is used to add a subplot to the fig figure object, which is created using plt.figure(). The plot_surface() function is called on the ax subplot, using x, y, and z as input to create the 3D plot.

The title parameter is used to set the title of the plot. Finally, the savefig() function is called to save the plot as an image file with the same name as the title parameter.

Synthetic Images

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

[ ]:
```
synthetic1 =  io.imread('im1.png')
synthetic2 =  io.imread('im2.png')
synthetic3 =  io.imread('im3.png')
synthetic4 =  io.imread('im4.png')
```

These lines of code read in four synthetic images named 'im1.png', 'im2.png', 'im3.png', and 'im4.png'. They use the imread function from the skimage.io module to read in the images as arrays of pixel values.

These arrays are assigned to variables named synthetic1, synthetic2, synthetic3, and synthetic4, respectively.

[ ]:
```
# light source directions
synthetic_image_light_source_direction = [[0, 0, -1],
                                          [0, 0.2, -1],
                                          [0, -0.2, -1],
                                          [0.2, 0, -1]]
```

This code defines a list called synthetic_image_light_source_direction which contains four vectors, each representing a different direction from which light is shining onto an object in the synthetic images.

The vectors are expressed as 3D coordinates, where the x, y, and z components represent the direction of the light source in each respective direction. Specifically, the light sources are positioned in front of the object, shining towards the camera.

[ ]:
```
synthetic1, synthetic2, synthetic3, synthetic4 = reshape_image(synthetic1,␣
↪synthetic2, synthetic3, synthetic4)
```

This code is calling the function reshape_image with four parameters (synthetic1, synthetic2, synthetic3, synthetic4) and assigning the returned values back to the same variables.

The purpose of this code is to reshape all four images to have the same dimensions.

[ ]:
```
# albedos and surface normal for synthetic images
synthetic_albedos, synthetic_surface_normals =␣
↪albedos_and_surface_normals(synthetic1, synthetic2, \
                                                    synthetic3, synthetic4, \
                                                   ␣
↪synthetic_image_light_source_direction)
```

This code computes the albedo and surface normal for each pixel in four synthetic images using the given light source directions.

The inputs are:

synthetic1: The first synthetic image.

synthetic2: The second synthetic image.

synthetic3: The third synthetic image.

synthetic4: The fourth synthetic image.

synthetic_image_light_source_direction: A list of light source directions in 3D space.

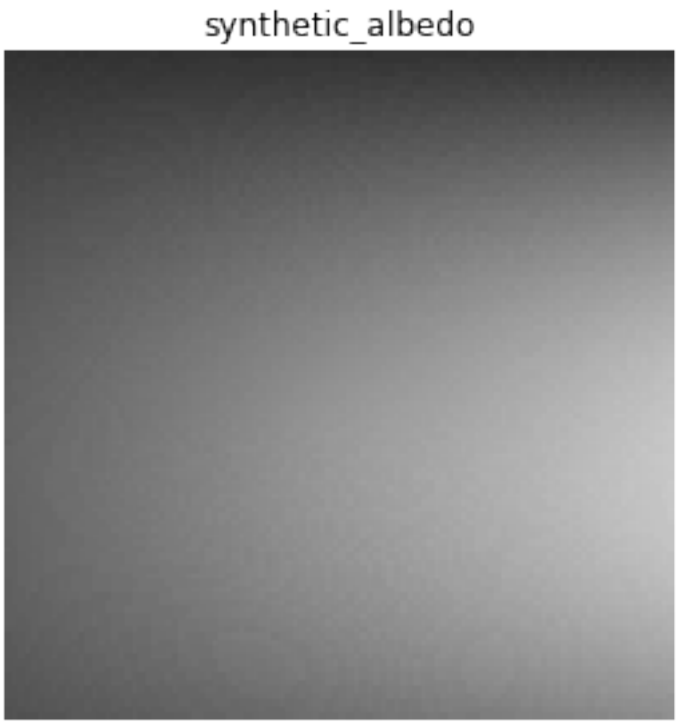
The output are:

synthetic_albedos: An array of albedos for each pixel in the synthetic images. synthetic_surface_normals: An array of surface normals for each pixel in the synthetic images.

Albedo is a measure of the reflectivity of a surface, specifically the fraction of incident electromagnetic radiation that is reflected by the surface. It is a unitless quantity that ranges from 0 (for a completely black surface that absorbs all incident radiation) to 1 (for a completely white surface that reflects all incident radiation). In the context of computer vision and image processing, albedo refers to the surface reflectance properties of objects in a scene, which can be estimated using photometric stereo techniques.

Surface normals are vectors perpendicular to a surface at each point on the surface. They represent the orientation of the surface at that point, pointing outwards from the surface. Surface normals are commonly used in computer graphics and computer vision to analyze the shape and geometry of 3D objects. They can be used to compute lighting and shading effects, estimate surface curvature, and reconstruct 3D models of objects from multiple 2D images.

[ ]:
```
###############################################################################
# Estimated albedo map: a grayscale image, "{real/synthetic}_albedo.png"
# o Since 0  albedo  1, multiply 255 to albedo to generate a grayscale albedo␣
 ↪map
###############################################################################
display_and_save_image(synthetic_albedos, "synthetic_albedo.png")
```



This code is displaying and saving a grayscale image of the estimated albedo map for the synthetic images. The albedo map is multiplied by 255 to generate a grayscale image.

The function used to display and save the image is display_and_save_image() which takes two arguments - the image and the image name. In this case, the image being displayed and saved is synthetic_albedos and the name of the image file is set to "synthetic_albedo.png".

[ ]:
```
# components of surface normals
synthetic_x, synthetic_y, synthetic_z, synthetic_dz_dx, synthetic_dz_dy =\
 surface_normal_components(synthetic_surface_normals, synthetic1.shape[0])
```

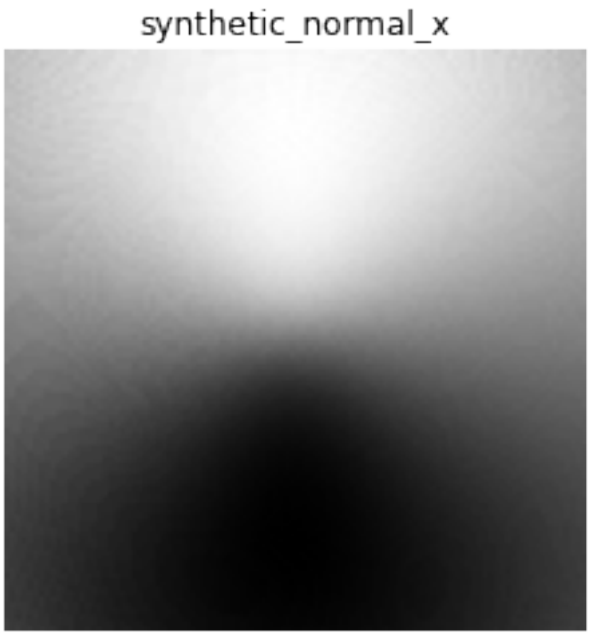
This code is computing the components of the surface normals for the synthetic images. It takes in the surface normals computed from the albedos and the light source directions of the synthetic images and the shape of the first synthetic image.

The surface_normal_components() function first extracts the x, y and z components of the surface normals from the input synthetic_surface_normals.

Then it calculates the slight changes in z with respect to x and y directions as dz_dx and dz_dy, respectively. These changes are computed using the x, y and z components of the surface normals.

The reshape_surface_normal_components() function is then called to reshape all the computed components according to the shape of the original image. The reshaped components, i.e., synthetic_x, synthetic_y, synthetic_z, synthetic_dz_dx, and synthetic_dz_dy are returned by the function.

```
[ ]: ##########################################################################################
     # Estimated surface normals: three grayscale images showing three components of␣
      ↪surface
     # normal
     # o {real/synthetic}_normal_{x/y/z}.png
     ##########################################################################################
     display_and_save_image(synthetic_x * 255, 'synthetic_normal_x.png')
```

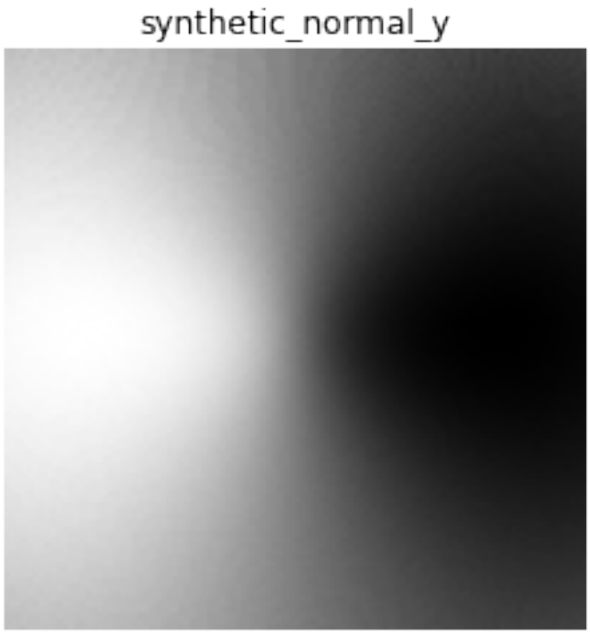


This code loads the synthetic1, synthetic2, synthetic3, and synthetic4 images and reshapes them to have the same size. It then defines the direction of the light source in the synthetic_image_light_source_direction variable.

Next, the albedos_and_surface_normals() function is called to estimate the albedo and surface normals for the synthetic images. The albedo is saved as a grayscale image named synthetic_albedo.png.

The surface_normal_components() function is then called to obtain the three components of the surface normal: synthetic_x, synthetic_y, and synthetic_z. Finally, the display_and_save_image() function is used to save and display the x-component of the surface normal as a grayscale image named synthetic_normal_x.png.

```
[ ]: display_and_save_image(synthetic_y, 'synthetic_normal_y.png')
```



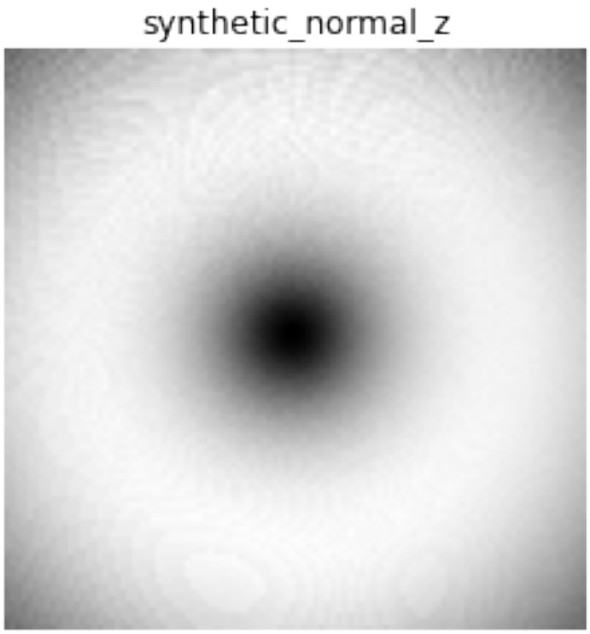
The code display_and_save_image(synthetic_y, 'synthetic_normal_y.png') calls a function display_and_save_image to display and save the synthetic_y array as a grayscale image with the filename 'synthetic_normal_y.png'.

The function takes two arguments, image and image_name, which are the array to be displayed and saved as an image and the filename of the image to be saved, respectively.

The synthetic_y array is one of the components of the estimated surface normal obtained from the input synthetic images. It is a two-dimensional grayscale image array that represents the magnitude of the surface normal's y-component at each pixel location.

The values in the array are in the range [0,1], where 0 represents the minimum value (no y-component) and 1 represents the maximum value (maximum y-component).

[ ]:
```
display_and_save_image(synthetic_z, 'synthetic_normal_z.png')
```



This code calls the display_and_save_image function to display and save the synthetic_z image as a grayscale image with the filename "synthetic_normal_z.png".

[ ]:
```
# depth of the image
synthetic_depth = surface_normals_to_depth(synthetic_z, synthetic_dz_dx,␣
↪synthetic_dz_dy, 1)
```

Depth from surface normal is a technique for estimating the depth or 3D shape of an object from its surface normals. The surface normal at a particular point on the object's surface represents the orientation of the surface at that point, or the direction perpendicular to the surface.

The basic idea behind depth from surface normal is to use the surface normals to calculate the surface curvature, which is related to the depth of the surface at each point. This technique assumes that the surface of the object is smooth and continuous, and that neighboring surface normals are related to each other in a meaningful way.

The depth from surface normal technique is often used in conjunction with other depth estimation techniques, such as stereo vision or time-of-flight cameras, to improve the accuracy of the depth estimation. It is also commonly used in computer graphics to create realistic 3D models of objects.

Overall, depth from surface normal is a useful technique for estimating the 3D shape of an object from its surface properties, and it has applications in a variety of fields, including computer vision, robotics, and computer graphics.

This code calculates the depth of the synthetic image using the components of the surface normal calculated earlier. The surface_normals_to_depth function takes in three parameters - synthetic_z, synthetic_dz_dx, and synthetic_dz_dy - which represent the three components of the surface normal. It also takes an optional parameter reshaping_value which is set to 1 by default.

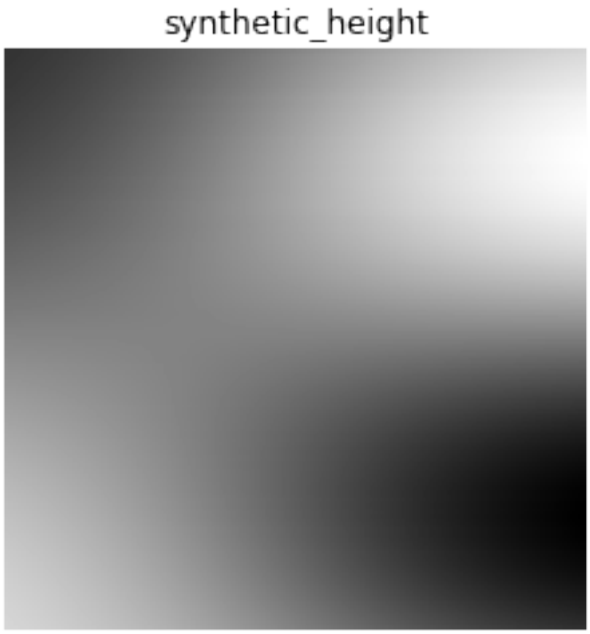
The function first calculates the depth of the image by multiplying the z component of the surface normal with the reshaping_value. It then sets the integration constant to 0, which represents the height of the surface at the start point of the image.

The function then iterates over the depth matrix to calculate the depth values for each pixel by subtracting the product of dz_dx and dz_dy from the previous depth value. The resulting matrix represents the depth of the synthetic image.

The function returns the calculated depth matrix. The resulting matrix can be used to create a 3D visualization of the image.

- **Reconstruct the height map from the surface normal.**

[ ]:
```
# Estimated surface height map either as
# o a grayscale image, "{real/synthetic}_height.png"
# - For this, you need to scale the height values between 0 to 255 (i.e., scale␣
 ↪the
# resulting height values so that the minimum height value maps to 0 and the
# maximum height value to 255).
# o 3D surface plot, "{real/synthetic}_height_3D.jpg"
# - For this, you should use the actual height values

display_and_save_image(synthetic_depth, 'synthetic_height.png')
```



Height from surface normals refers to a technique used to estimate the height or elevation of a terrain or surface from its surface normals. Surface normals represent the orientation of the surface at each point, or the direction perpendicular to the surface.

The basic idea behind height from surface normals is to use the surface normals to estimate the slope and curvature of the surface at each point. By integrating these values over the surface, the height or elevation of the surface can be estimated.

This technique assumes that the surface is continuous and smooth, and that neighboring surface normals are related to each other in a meaningful way. The accuracy of the height estimation depends on the quality and density of the surface normal data.

Height from surface normals has a wide range of applications, including terrain mapping, 3D reconstruction of natural scenes, and urban planning. For example, in terrain mapping, height from surface normals can be used to create high-resolution digital elevation models that are useful in environmental modeling and simulation. In urban planning, height from surface normals can be used to estimate the height of buildings and other structures, which is important for assessing the impact of new development on the surrounding environment.

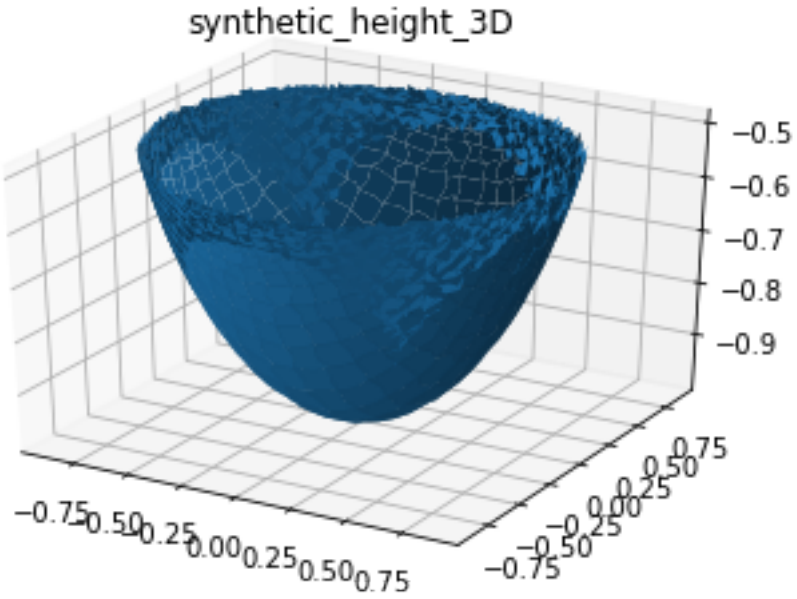
The code is displaying and saving the estimated surface height map of the synthetic image. There are two options to save the estimated surface height map, either as a grayscale image or as a 3D surface plot.

To save the grayscale image, the height values need to be scaled between 0 to 255, and then displayed using the display_and_save_image() function.

To save the 3D surface plot, the actual height values can be used, and the display_and_save_height_map_3d() function can be used to display and save the 3D surface plot.

In this particular code line, the grayscale image of the estimated surface height map is being displayed and saved using the display_and_save_image() function, with the filename "synthetic_height.png". The synthetic_depth variable contains the estimated surface height map.

[ ]:
```
display_and_save_height_map_3d(synthetic_x, synthetic_y, synthetic_z,␣
↪'synthetic_height_3D.jpg')
```



This code is calling a function named display_and_save_height_map_3d and passing four arguments to it: synthetic_x, synthetic_y, synthetic_z, and 'synthetic_height_3D.jpg'.

The function is creating a new figure and adding a 3D subplot to it with the given x, y, and z data.

It then sets the title of the plot using the filename of the image as the title (with the extension removed) and saves the plot as an image file with the given filename. Finally, it displays the plot using the plt.show() function.

[ ]:
```
print("aledos")
print(np.round(synthetic_albedos[45:50, 45:50], 4))
```

```
aledos
[[0.5355 0.5368 0.541  0.5445 0.5521]
 [0.541  0.5454 0.5492 0.552  0.5576]
 [0.5434 0.5502 0.5542 0.5571 0.56  ]
 [0.5499 0.5541 0.5582 0.5623 0.5651]
 [0.5577 0.557  0.5626 0.5665 0.5691]]
```

This code prints out a subset of the synthetic_albedos array rounded to 4 decimal places. The subset is a 5x5 array, starting at row 45 and column 45. The purpose of printing this subset is to visualize a small portion of the albedo values and check if they are within the expected range of 0 to 1.

[ ]:
```
print("x-components")
print(np.round(synthetic_x[45:50, 45:50], 4))
```

```
x-components
[[ 0.2549  0.2544  0.2525  0.251   0.2831]
 [ 0.1681  0.2028  0.2015  0.2005  0.21  ]
 [ 0.1069  0.1298  0.129   0.1283  0.104 ]
 [ 0.0457  0.0581  0.0578  0.0451  0.0337]
 [-0.0365 -0.0126 -0.0247 -0.0356 -0.0354]]
```

[ ]:
```
print("y-components")
print(np.round(synthetic_y[45:50, 45:50], 4))
```

```
y-components
[[0.2881 0.2156 0.1427 0.0709 0.    ]
 [0.304  0.2306 0.1586 0.0877 0.    ]
 [0.2841 0.2287 0.1572 0.0869 0.    ]
 [0.2975 0.2271 0.1561 0.0857 0.    ]
 [0.3115 0.226  0.1541 0.0682 0.    ]]
```

[ ]:
```
print("z-components")
print(np.round(synthetic_z[45:50, 45:50], 4))
```

```
z-components
[[-0.9231 -0.9427 -0.957  -0.9654 -0.9591]
 [-0.9377 -0.9517 -0.9666 -0.9758 -0.9777]
```

```
 [-0.9528 -0.9648 -0.9791 -0.9879 -0.9946]
 [-0.9536 -0.9721 -0.986  -0.9953 -0.9994]
 [-0.9495 -0.9741 -0.9877 -0.997  -0.9994]]
```

[ ]:
```
print("depth")
print(np.round(synthetic_depth[45:50, 45:50], 4))
```

```
depth
[[57.3565 57.6264 57.8903 58.1503 58.4454]
 [56.3759 56.589  56.7975 57.003  57.2178]
 [55.4003 55.5348 55.6665 55.7964 55.9011]
 [54.2866 54.3464 54.405  54.4503 54.4841]
 [53.1525 53.1396 53.1146 53.0789 53.0435]]
```

Real images

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

[ ]:
```
# Read the images
real1 =  io.imread('real1.bmp')
real2 =  io.imread('real2.bmp')
real3 =  io.imread('real3.bmp')
real4 =  io.imread('real4.bmp')
```

This code reads in four real images named “real1.bmp”, “real2.bmp”, “real3.bmp”, and “real4.bmp”.

It uses the “io.imread” function from the “skimage” library to read in the images and store them as arrays in the variables “real1”, “real2”, “real3”, and “real4”.

[ ]:
```
# light source directions
real_image_light_source_direction = [[-0.38359, -0.236647, 0.892668],
                                     [-0.372825, 0.303914, 0.87672],
                                     [0.250814, 0.34752, 0.903505],
                                     [0.203844, -0.096308, 0.974255]]
```

This code defines a list of four 3D vectors, where each vector represents the direction of the light source used to capture a particular image.

The vectors are specified as floating-point values and are assigned to the variable real_image_light_source_direction.

These vectors are used as input to other functions to estimate the surface normals, albedo, and height maps of the real images.

[ ]:
```
# reshape the real images
real1, real2, real3, real4 = reshape_image(real1, real2, real3, real4)
```

This code is reshaping the four real images real1, real2, real3, and real4. It takes these images as inputs and returns the reshaped versions of the images.

The reshaping is done to ensure that all the images have the same dimensions, which is a requirement for the surface normal estimation method used in this program. The code most likely calls a function

named reshape_image, which takes the real images as inputs and returns the reshaped versions of the images.

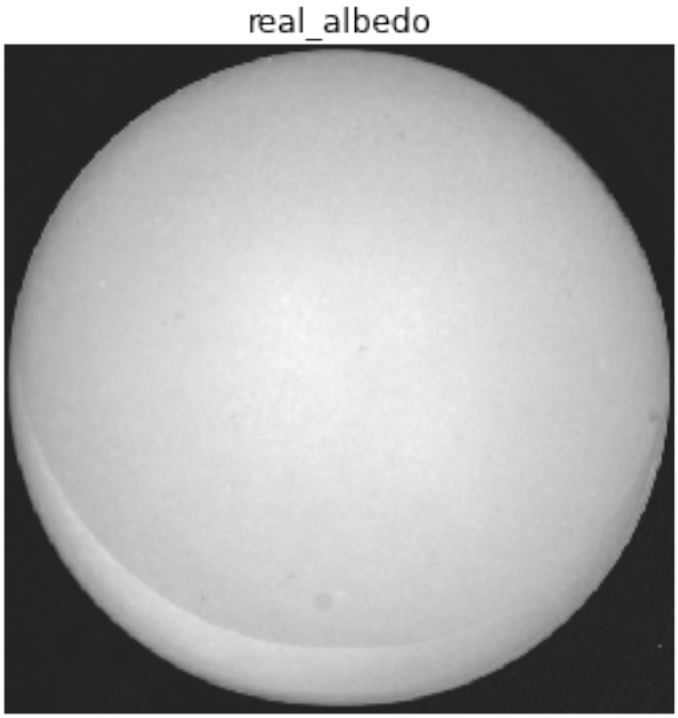
```
[ ]: # get the albedos and surface normals for real images
real_albedos, real_surface_normals = albedos_and_surface_normals(real1, real2, \
                                                                  real3, real4, \
                                                                  ␣
 ↪real_image_light_source_direction)
```

This code calculates the albedos and surface normals for the real images using the function albedos_and_surface_normals.

The function takes in the four images and the light source directions for each image, and returns two outputs: real_albedos, which is an array of the estimated albedo values for each pixel in the images, and real_surface_normals, which is an array of the estimated surface normal vectors for each pixel.

These values are calculated based on the observed intensities of the images under different light source directions.

```
[ ]: ##########################################################################################
# Estimated albedo map: a grayscale image, "{real/synthetic}_albedo.png"
# o Since 0  albedo  1, multiply 255 to albedo to generate a grayscale albedo␣
 ↪map
##########################################################################################
display_and_save_image(real_albedos, 'real_albedo.png')
```



This code is displaying and saving the estimated albedo map for real images. The albedo map is a grayscale image where each pixel represents the reflectivity of the surface at that point.

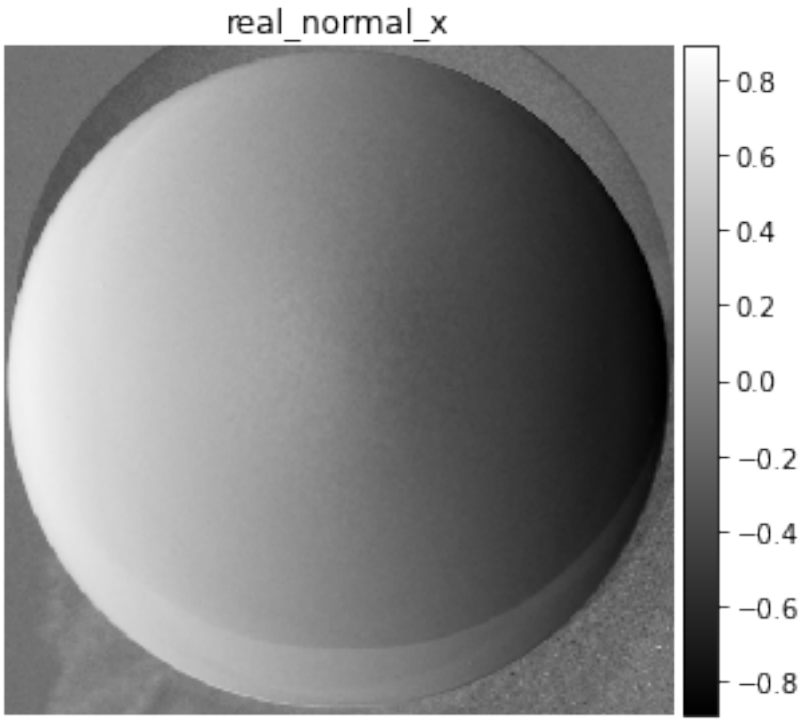
Since the albedo values are in the range of 0 to 1, the code multiplies each pixel value by 255 to convert it to the grayscale range of 0 to 255. The resulting image is saved as "real_albedo.png".

[ ]:
```
##########################################################################################
# Estimated surface normals: three grayscale images showing three components of␣
 ↪surface
# normal
# o {real/synthetic}_normal_{x/y/z}.png
##########################################################################################
real_x, real_y, real_z, real_dz_dx, real_dz_dy =␣
 ↪surface_normal_components(real_surface_normals, real1.shape[0])
```

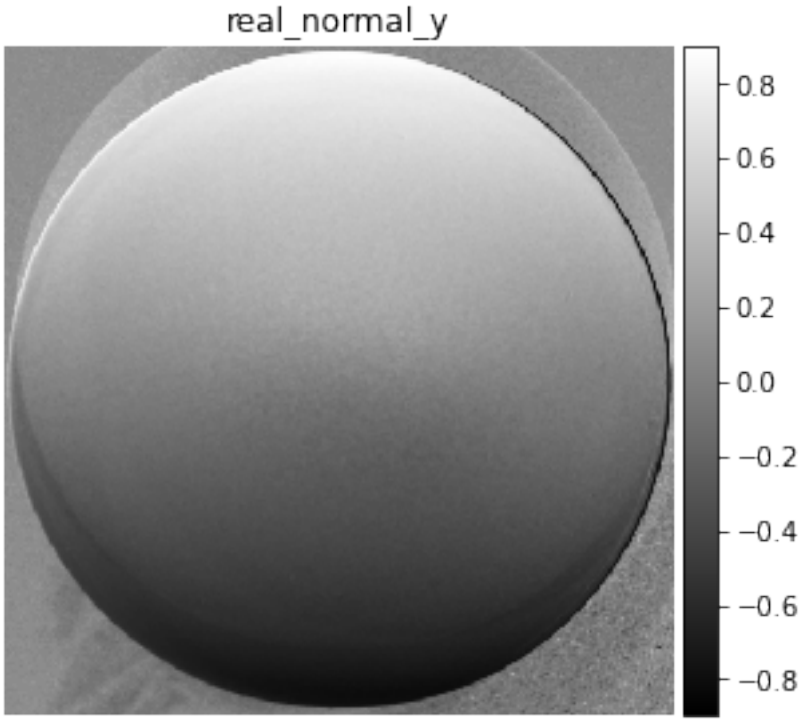
This code computes the three components of the surface normal (x, y, z), as well as the partial derivatives of the z component with respect to x and y, using the estimated surface normals obtained for the real images.

The surface normal components and partial derivatives are returned and assigned to variables named real_x, real_y, real_z, real_dz_dx, and real_dz_dy.

[ ]:
```
display_and_save_image(real_x , 'real_normal_x.png')
```



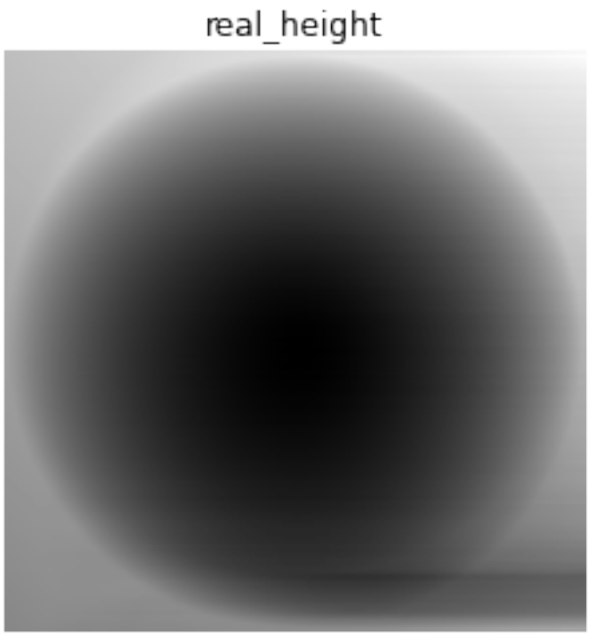
[ ]:
```
display_and_save_image(real_y, 'real_normal_y.png')
```



[ ]:
```
display_and_save_image(real_z, 'real_normal_z.png')
```

- **Reconstruct the height map from the surface normal.**

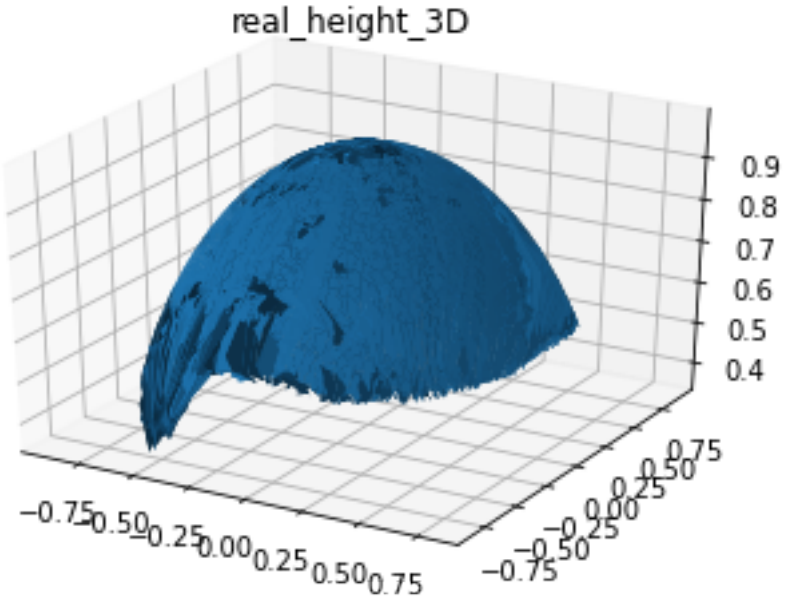
[ ]:
```
# Estimated surface height map either as
# o a grayscale image, "{real/synthetic}_height.png"
# - For this, you need to scale the height values between 0 to 255 (i.e., scale␣
 ↪the
# resulting height values so that the minimum height value maps to 0 and the
# maximum height value to 255).
# o 3D surface plot, "{real/synthetic}_height_3D.jpg"
# - For this, you should use the actual height values

real_depth = surface_normals_to_depth(real_z, real_dz_dx, real_dz_dy, 255)
display_and_save_image(real_depth * 255, 'real_height.png')
```



This code computes the surface height map for the real images. It uses the surface normal components real_z, real_dz_dx, and real_dz_dy to compute the depth of the image using the surface_normals_to_depth() function.

It then scales the depth values between 0 and 255 and saves it as a grayscale image using the display_and_save_image() function with the filename 'real_height.png'.

[ ]:
```
# display 3d height map for real images
display_and_save_height_map_3d(real_x, real_y, real_z, 'real_height_3D.jpg')
```



This code calls the display_and_save_height_map_3d function to display and save the 3D height map for the real images.

It takes the real_x, real_y, and real_z components of the surface normal as input and saves the 3D surface plot as a JPEG image file with the name "real_height_3D.jpg".

The resulting plot will show the 3D shape of the surface of the real object, with height represented by the z-axis and the x and y axes representing the horizontal plane.

[ ]:
```
print("albedos")
print(np.round(real_albedos[45:50, 45:50], 4))
```

```
albedos
[[0.142  0.1409 0.1409 0.1421 0.14  ]
 [0.1389 0.1399 0.1409 0.1409 0.1409]
 [0.1409 0.1399 0.1399 0.1399 0.1409]
 [0.1399 0.1399 0.1399 0.1399 0.139 ]
 [0.1399 0.1399 0.1412 0.1412 0.1425]]
```

[ ]:
```
print("x-components")
print(np.round(real_x[45:50, 45:50], 4))
```

```
x-components
[[-0.0792 -0.1002 -0.1002 -0.1272 -0.1245]
 [-0.0977 -0.0732 -0.1002 -0.1002 -0.1002]
 [-0.1002 -0.0732 -0.0817 -0.0817 -0.1002]
 [-0.0817 -0.0817 -0.0817 -0.0817 -0.1062]
 [-0.0817 -0.0817 -0.1096 -0.1096 -0.0883]]
```

[ ]:
```
print("y-components")
print(np.round(real_y[45:50, 45:50], 4))
```

```
y-components
[[0.1118 0.0866 0.0866 0.1174 0.1114]
 [0.0808 0.0558 0.0866 0.0866 0.0866]
 [0.0866 0.0558 0.1171 0.1171 0.0866]
 [0.1171 0.1171 0.1171 0.1171 0.1431]
 [0.1171 0.1171 0.1478 0.1478 0.1717]]
```

[ ]:
```
print("z-components")
print(np.round(real_z[45:50, 45:50], 4))
```

```
z-components
[[0.9906 0.9912 0.9912 0.9849 0.9859]
 [0.9919 0.9958 0.9912 0.9912 0.9912]
 [0.9912 0.9958 0.9898 0.9898 0.9912]
 [0.9898 0.9898 0.9898 0.9898 0.984 ]
 [0.9898 0.9898 0.9829 0.9829 0.9812]]
```

[ ]:
```
print("depth")
print(np.round(real_depth[45:50, 45:50], 4))
```

```
depth
[[ 0.0866  0.1877  0.2888  0.418   0.5443]
 [ 0.2706  0.3442  0.4453  0.5464  0.6474]
 [ 0.0656  0.1392  0.2217  0.3043  0.4054]
 [ 0.0404  0.123   0.2056  0.2881  0.396 ]
 [-0.1642 -0.0816  0.0299  0.1415  0.2315]]
```