COMPSCI 670: Computer Vision Fall 22

Homework 1

Prachi Jain

1 Short answers

1.1

Implementation of *question1.py*

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.lib import scimath as SM
np.seterr(divide='ignore', invalid='ignore')
import os
import warnings

def lambertian_sphere(radius, albedo):
        xx = np.linspace(-1, 1, 1501)
        yy = np.linspace(-1, 1, 1501)
        x, y = np.meshgrid(xx, yy)

        #p = -x/sqrt(r^2 - (x^2 + y^2)) where r is radius - 1 meter
        px = np.power(x,2)
        py = np.power(y,2)
        pz = SM.sqrt(radius ** 2 - (px + py))


        p = -x / pz
        q = -y / pz
        mask = np.copy(1 ** 2 - (px + py))

        mask[mask >= 0] = 1
        mask[mask < 0] = 0

        I = np.array([1,1,1])
        I = np.transpose(I)

        N_dot_S = (-I[0] * p - I[1] * q + I[2]) / (SM.sqrt(1 + np.power(p,2) + np.power(q,2)))
        #For a perfect Lambertian surface : Radiosity = Albedo * Unit Normal * Source vector
        R = (albedo * N_dot_S )

        mask = np.reshape(mask, (1501,1501))
        R = np.multiply (R, mask)
        R*=-1
```

```python

        E = np.copy(R)
        E[E < 0] = 0
        where_are_NaNs = np.isnan(E)
        E[where_are_NaNs] = 0
        E = E / np.amax(E)
        plt.imshow(np.real(E), 'gray')
        plt.title('1.a A perfect Lambertian surface.')
        plt.show()

def spec_sphere(r=1, z=2, specular=True, f=128, h=256, w=384, cx=192, cy=128):
  #Refrences and articles used for understanding:
  #https://www.robots.ox.ac.uk/~att/index.html
  #https://github.com/amanrajdce/ComputerVision-CSE-252A/blob/master/HW2/HW2.ipynb
  # Assume that camera is at origin looking in the +Z direction. X axis to the right, Y axis downwards.
  # Also sphere is at (0, 0, z) and of radius r.
  x, y = np.meshgrid(np.arange(w), np.arange(h))
  x = x - cx
  y = y - cy

  phi = np.arccos(f / np.sqrt(x**2 + y**2 + f**2))
  # equation is rho*2 (1+tan^2(phi)) - 2 rho * z + z*z - r*r = 0
  a = np.tan(phi)**2 + 1
  b = -2 * z
  c = z**2 - r**2

  rho = -b - np.sqrt(b**2 - 4*a*c)
  rho = rho / 2 / a
  depth = rho

  # Calculate the point on the sphere surface
  X = x * depth / f
  Y = y * depth / f
  Z = depth

  N = np.array([X, Y, Z])
  N = np.transpose(N, [1,2,0])
  N = N - np.array([[[0, 0, z]]])
  N = N / np.linalg.norm(N, axis=2, keepdims=True)
  Z[np.isnan(Z)] = np.inf

  #compute k_d, k_a
  A = (np.sign(X*Y) > 0) * 0.5 + 0.1

  #set specularity
  if specular:
    S = np.invert(np.isinf(depth)) + 0
  else:
    S = A-A
  return Z, N, A, S

ke = 50

def render(Z, N, A, S, point_light_location, point_light_strength, directional_light, directional_light_strength, ambient_light, ke):
  #Assume (cx, cy) denote the center of the image, f is the focal length.
```

```python
h, w = A.shape
cx, cy = w / 2, h /2
f = 128.

#Get the camera coordinates of all the points in the pixel coordinates
coordinates = np.zeros((h, w, 3))
for y in range(h):
    for x in range(w):
        depth = Z[y, x]
        coord = [(x - cx) * depth/f, (y - cy) * depth/f, depth]
        coordinates[y, x, :] = np.array(coord)

#Normalize the directional light
directional_light = np.array(directional_light)/np.linalg.norm(np.array(directional_light))

#Ambient Term
I = A * ambient_light

#Diffuse Term
point_light_direction = np.array(point_light_location) - coordinates
point_light_direction = (point_light_direction.T/np.linalg.norm(point_light_direction, axis=2).T).T

#Dot product of direction vectors with the surface normal vectors
point_diffuse = np.sum(point_light_direction * N, axis=2)
directional_diffuse = np.sum(directional_light * N, axis=2)

#Filter out the values that are negative in the diffuse matrix
filter = point_diffuse < 0
point_diffuse[filter] = 0
filter = directional_diffuse < 0
directional_diffuse[filter] = 0

#Specular Term
I += A*(point_diffuse * np.array(point_light_strength) + directional_diffuse * np.array(directional_light_strength))

#Get the viewing directions and normalize
vr = np.zeros(3) - coordinates
vr = (vr.T/np.linalg.norm(vr, axis=2).T).T

#Reflection is the reverse of the incident light
point_reflection = (2*np.sum(point_light_direction * N, axis=2).T*N.T).T - point_light_direction
directional_reflection = (2*np.sum(directional_light * N, axis=2).T*N.T).T - directional_light

#Get the specular matrices
point_specular = np.sum(point_reflection *vr, axis=2)
directional_specular = np.sum(directional_reflection * vr, axis=2)

#Filter out the negative values in the specular matrices
filter = point_specular < 0
point_specular[filter] = 0
filter = directional_specular < 0
directional_specular[filter] = 0

```

```
  I += S*(np.array(point_light_strength) * (point_specular)**ke + np.array(directional_light_strength) * (directional_specular)** ke)

  I = np.minimum(I, 1)*255
  I = I.astype(np.uint8)
  I = np.repeat(I[:,:,np.newaxis], 3, axis=2)
  return I

def main():
  lambertian_sphere(1, 1)
  for specular in [True]:
    # spec_sphere function returns:
    # - Z (depth image: distance to scene point from camera center, along the Z-axis)
    # - N is the per pixel surface normals
    # - A is the per pixel ambient and diffuse reflection coefficient per pixel
    # - S is the per pixel specular reflection coefficient
    Z, N, A, S = spec_sphere(specular=True)

    # Strength of the ambient light.
    ambient_light = 0.5

    #No directional light, only point light source that moves around the object.
    #point_light_strength = [1.5]
    directional_light_dirn = [[1, 0, 0]]
    directional_light_strength = [0.0]
    exponents = [0.0, 0.25, 0.5 , 0.75, 1, 1.25, 1.5, 1.75, 2.00, 2.5, 5, 10, 15, 20, 25, 30]

    fig, axes = plt.subplots(4, 4, figsize=(15,10))
    axes = axes.ravel()[::-1].tolist()
    theta =  5.86430629
    for point_light_strength in exponents:
      point_light_loc = [[10*np.cos(theta), 10*np.sin(theta), -3]]
      I = render(Z, N, A, S, point_light_loc, point_light_strength, directional_light_dirn, directional_light_strength, ambient_light, ke)
      ax = axes.pop()
      ax.imshow(I)
      ax.set_axis_off()

    #Specular surface rendered using the Phong reflection model with varying exponents
    plt.show()
    plt.close()

if __name__ == '__main__':
    main()
```

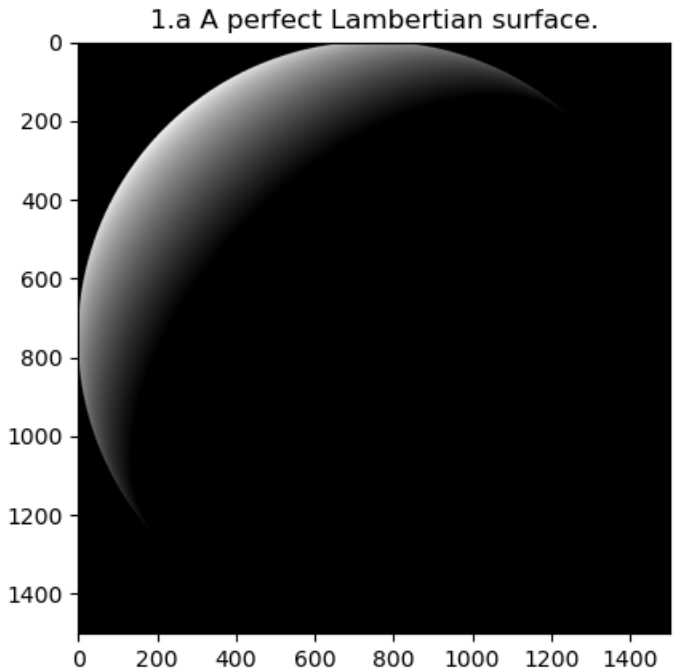


Figure 1: A perfect Lambertian surface.

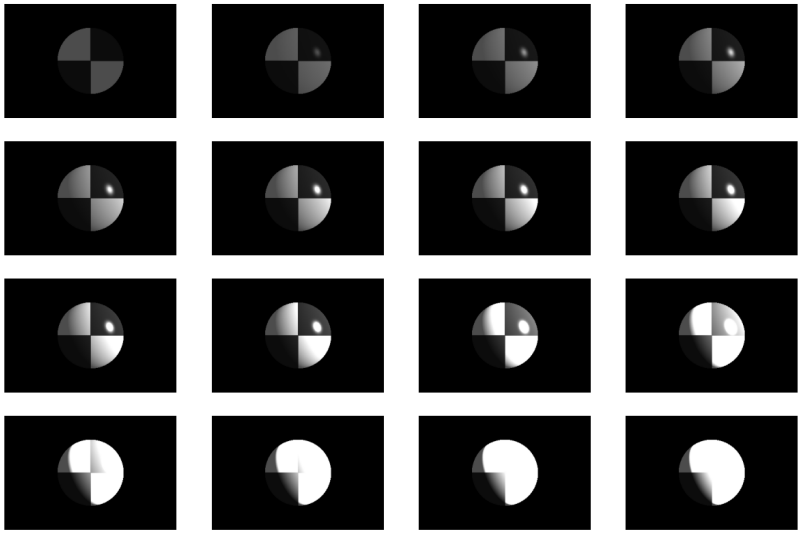


Figure 2: Specular surface rendered using the Phong reflection model with varying exponents)

1.2

The color we perceive results from the degree to which the light cones in our eyes are excited/react to stimuli of primary colors/cones which only humans can perceive. For example, red, green, and blue light can collectively stimulate all three light cones and thus give rise to a white light (appearance).

Paint (pigment) absorbs some of the light, reflecting less. Mixing paints absorb more and reflect less. The color we see is what is reflected, so mixing paint is subtractive; The more we add paints of the same colors, the more the accumulation of similar wavelengths, and the closer it produces to black/darker shades of color.

1.3

Metamerism is the condition when two colors that are not actually the same (they reflect different wavelengths of light) appear the same under certain lighting conditions. Colors that match under some lights but not others are called metamers. These subjective color matches are most common in near-neutral or dark colors and are problematic when purchasing clothing or painting a room.

People see colors when specific wavelengths of light stimulate elements of the eye. Cones, vital for human vision, act as filters on the spectrum across such multiple wavelengths. The pigmentation of objects reflects these wavelengths. When light hits an object, specific wavelengths responsible for specific colors are absorbed while others are reflected. The reflected wavelengths are what we see and understand as a color (illusion). This entire process is based on three different things: reflectivity of a substance, light source, and which colors can be perceived by your eyes.

For example, a light source inside a retail store emits only blue and red wavelengths, and natural light outside emits red, blue, and green wavelengths; how would this affect what a green shirt looks like? In the store, the shirt absorbs blue and red wavelengths, and no light is reflected, so the shirt appears black. When you take the shirt back outside, it is exposed to all three wavelengths. Red and blue are absorbed under the natural light, but the additional green wavelength is reflected and picked up by your sight, so outside, the shirt is green.

Humans only have 3 (Red, Blue, and Green)cones for vision which takes the light source vector containing all those three frequencies and outputs a single value. Each type of the three cones translates to 3 numbers. However, humans cannot represent the entire spectrum using these three numbers. Thus, although we get spectra of different wavelengths, we may not be able to distinguish - metamers (them).

1.4

It is often difficult to read under low light because when we read the text in a book, we use cones to see the nuances of the text (to view/read from the center position). This makes cones ineffective in the dark light/night time. Rods help us see from the edge positions (not center positions) in the dark light. Hence when we can use peripheral vision (viewing from edges/sides) as objects are easier to view/read. However, this is not our conventional body posture, so we need to learn it to adapt our vision in dark light/nighttime.

1.5

Choose a pair of parallel lines from plane $N^T X = d$ where N is the normal (perpendicular direction) to the plane and d is the distance between those lines.

$$Line\ A\ :\ Y = mX,\ Z = d - N_x X - N_y Y$$
$$Line\ B\ :\ Y = mX + b,\ Z = d - N_x X - N_y Y$$

Using Geometry, *the projection of the lines in a 2D plane are also lines.*
Choose $(0,\ 0,\ d), (s,\ ms,\ z_a)$ from *Line A* and $(0,\ b,\ z_0), (s,\ ms{+}b,\ z_b)$ from *Line B* where

$$z_0 = Z(0,\ b)$$
$$z_a = Z(s,\ ms)$$
$$z_b = Z(s,\ ms + b)$$

to compute the equations of the projected lines in 2D plane.

Converting camera coordinates to image coordinates, using points $(0,\ 0)$ and $(\frac{fs}{z_a},\ \frac{fms}{z_a})$ from *Line A* and, $(0,\ \frac{fb}{z_0})$ and $(\frac{fs}{z_b},\ \frac{f(m+b)}{z_b})$ from *Line B*:

$$Line\ A\ :\ y = mx$$
$$Line\ B\ :\ y = (m + \frac{b(z_0 - z_b)}{z_0 s})x + \frac{fb}{z_0}$$

A *vanishing point* of any given line in space is located at the point in an image where a parallel line through the center of projection intersects the image plane.
A *vanishing line* is formed by such vanishing points of all the lines that lie on the same plane.

Solving for intersecting point (vanishing point) of *Line A* and *Line B* in an image plane:

$$x^{'} = \frac{fs}{z_b - z_0} = \frac{fs}{\frac{d - N_x s - N_y(ms+b) - d + N_y b}{N_z}} = -\frac{fN_z}{N_x + N_y m}$$
$$y^{'} = mx^{'} = -\frac{fN_z m}{N_x + N_y m}$$

Replacing m with its values $\frac{y}{x}$:

$$x^{'} = -fN_z \frac{x}{N_x x + N_y y}$$
$$y^{'} = -fN_z \frac{y}{N_x x + N_y y}$$

Rearranging the terms in the previous equation:

$$N_x x^{'} + N_y y^{'} = -fN_z$$

This deduces that the intersecting point of *Line A* and *Line B* in an image plane also known as the vanishing point always lies on the line $N_x x^{'} + N_y y^{'} + fN_z = 0$ i.e. the vanishing line of the plane.

2 Photometric stereo

2.1 Prepare the data

Implementation of *prepareData.py*

```
import numpy as np

def prepareData(imArray, ambientImage):

    h, w, c = imArray.shape
    data = np.zeros((h, w, c))

    #removing the affect of ambient lighting by linearity of light
    for i in range(c):
        data[:,:,i] = imArray[:,:,i] - ambientImage
        #renormalize images
        data[data < 0] = 0
    data = data / np.max(imArray)

    return data
```

2.2 Estimate the albedo and normals

Implementation of *photometricStereo.py*

```
import numpy as np

# importing library to obtain the least-squares solution of a linear system
from scipy.linalg import lstsq

def photometricStereo(imarray, lightdirs):
    h, w, c = imarray.shape
    reshape_array = np.zeros((c, h * w))
    for i in range(c):
        reshape_array[i,:] = np.reshape(imarray[:,:,i],[1,h*w])

    #Setting up least-squares solution of a linear system with the assumption of using a perfectly Lambertian surface
    gxy = lstsq(lightdirs, reshape_array)

    gxy = np.array(gxy[0])

    albedo1 = np.zeros((1,h * w))
    normal1 = np.zeros((3,h * w))

    #computing albedo and normals
    for i in range(h * w):
        albedo1[0, i] = np.linalg.norm(gxy[:,i])
        normal1[:, i] = gxy[:,i]/np.linalg.norm(gxy[:,i])

    #resizing albedo and normals
    albedo2 = np.reshape(albedo1,(h, w))
    normal2 = np.dstack([np.reshape(normal1[0,:], (h, w))[...,np.newaxis], np.reshape(normal1[1,:],(h, w))[...,np.newaxis], np.reshape(normal1[2,:],(h, w))[...,np.newaxis]]
                        )
    return albedo2, normal2
```

2.3 Estimate the surface height map

Implementation of *getSurface.py*

Row-column Integration:
Integrating first rows, then columns i.e. path first goes along the same row as the pixel along the top, and then goes vertically down to the pixel.

Column-row Integration:
Integrating first along the columns, then the rows.

Average Integration:
Integration is performed over both row-column and column-row paths and then averaged.

Random Integration:
Flip coins to generate 50 random paths. Only allowed to move right and down as follows:
Coin > Direction > Height

0 > Right > + partial derivative X

1 > Down > + partial derivative Y

Algorithm:
1. For each pixel(x, y) except start pixel(0, 0):
Assuming Height Map of (0, 0) is always 0 therefore it doesn't require any path.
2. For each path traversed:
3. A coin is flipped of length X + Y
4. Move according to coin(s) until reach target pixel(x, y)
5. Add the cumsum along the path to Height Map
6. Take average over all paths

```
import numpy as np

#library for performing random integration
import random

def getSurface(surfaceNormals, method):

    #partial derivatives computation wrt z
    partial_x = surfaceNormals[:, :, 0] / surfaceNormals[:, :, 2]
    partial_y = surfaceNormals[:, :, 1] / surfaceNormals[:, :, 2]

    row_sum = np.cumsum(partial_x, axis=1)
    col_sum = np.cumsum(partial_y, axis=0)

    if method == 'row-column':
        '''
        Integrating first the rows, then the columns. That is, your path first goes along the same row as
        the pixel along the top, and then goes vertically down to the pixel. Call this option  row -column
```

```
        '''
        return col_sum[:, 0][:, np.newaxis] + row_sum

    if method == 'column-row':
        '''
        Integrating first along the columns, then the rows. Call this option column-row.
        '''
        return row_sum[0] + col_sum

    if method == 'average':
        '''
        Average of the first two options.
        '''
        row_column_ = getSurface(surfaceNormals, method="row-column")
        column_row_ = getSurface(surfaceNormals, method="column-row")
        heightMap = (row_column_ + column_row_)//2
        return heightMap

    if method == 'random':
        '''
        Average of multiple random paths. There are several ways to sample random paths,
        and we leave it up to you to figure out what works best.Determine the number of required paths experimentally
        by visualizing the results for different number of paths. You might find switching back to the toy example and
        running the algorithm on smaller images useful to debugging as this approach is relatively slower.
        '''
        #flipping coins to 50 generate random paths
        h = surfaceNormals.shape[0]
        w = surfaceNormals.shape[1]
        heightMap = np.zeros((h, w))

        #configure number of random paths
        num_paths = 50

        #loop through each pixel.
        for y in range(h):#Row
            for x in range(w):#Column
                #exclude the start pixel(0, 0)
                if x != 0 or y != 0:
                    for path in range(num_paths):
                        #guarantees #0 = x, #1 = y in coins
                        zeros = [0] * x
                        ones = [1] * y
                        coin = np.array(zeros + ones)
                        #randomly shuffle coins to create sudo-random path
                        np.random.shuffle(coin)

                        #start pixel
                        xCurrent = 0
                        yCurrent = 0
                        step = 0
                        cumsum = 0

                        #move along path until reach target pixel
```

```
                        while xCurrent < x or yCurrent < y:
                            #right movement
                            if coin[step] == 0:
                                cumsum += partial_x[yCurrent, xCurrent]
                                xCurrent += 1
                            else:
                                cumsum += partial_y[yCurrent, xCurrent]
                                yCurrent += 1

                            step += 1

                        #add cumsum along path to heightMap
                        heightMap[y, x] += cumsum

                    #compute average over paths
                    heightMap[y, x] = heightMap[y, x]/num_paths
        return heightMap
```

2.4 Display the results using displayOutput and displaySurfaceNormals functions

The following outputs- **Estimated Albedo, Estimated Normals and Integration Height Maps** of four subjects namely, *YaleB01, YaleB02, YaleB05 and YaleB07* were generated with respect to *random* (best) integration method and 10 view points for each subject was visualized.

Figure 3: YaleB01 Estimated Albedo

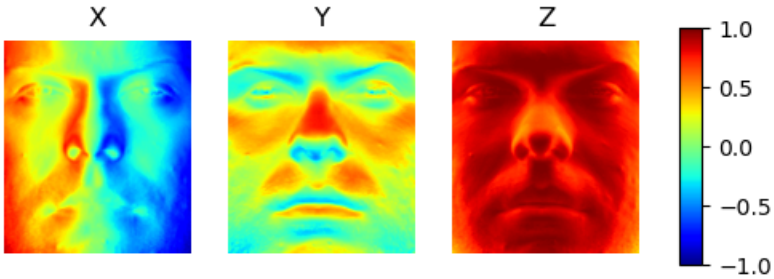


Figure 4: YaleB01 Estimated Normals

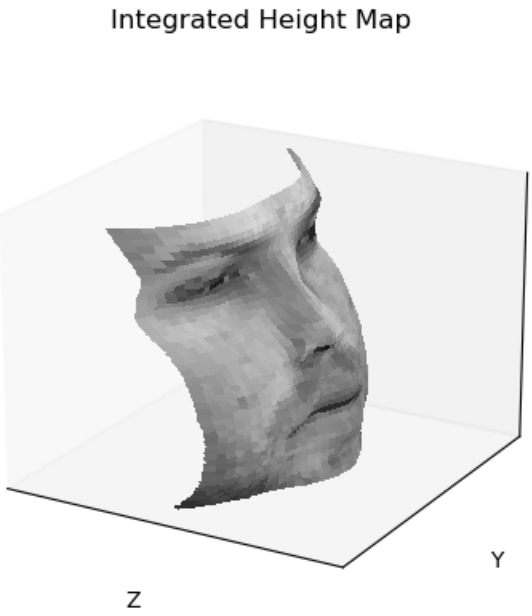


Figure 5: YaleB01 with view: (-60,20)

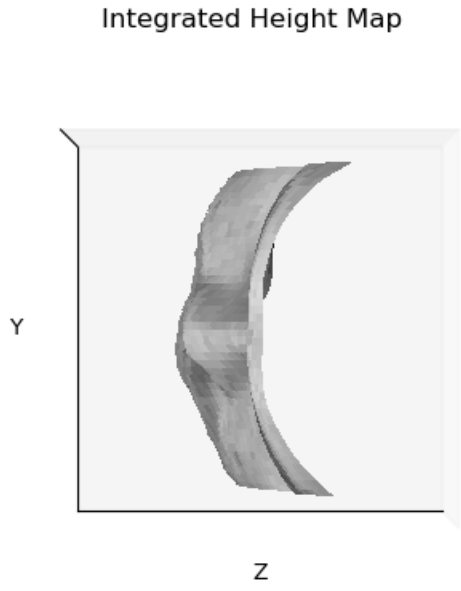


Figure 6: YaleB01 with view: (90,90)

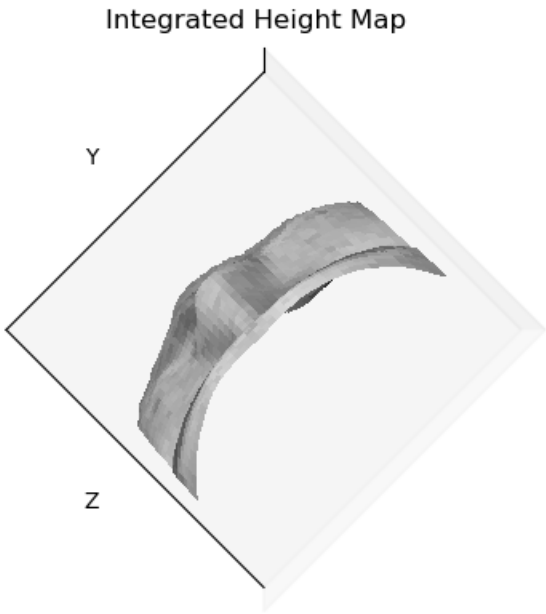


Figure 7: YaleB01 with view: (135,90)

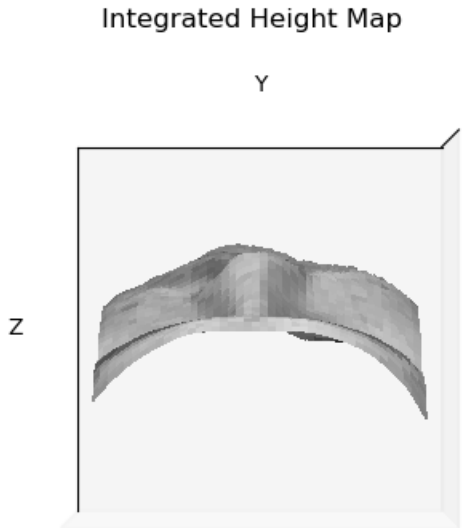


Figure 8: YaleB01 with view: (180,90)

Figure 9: YaleB01 with view: (90,55)

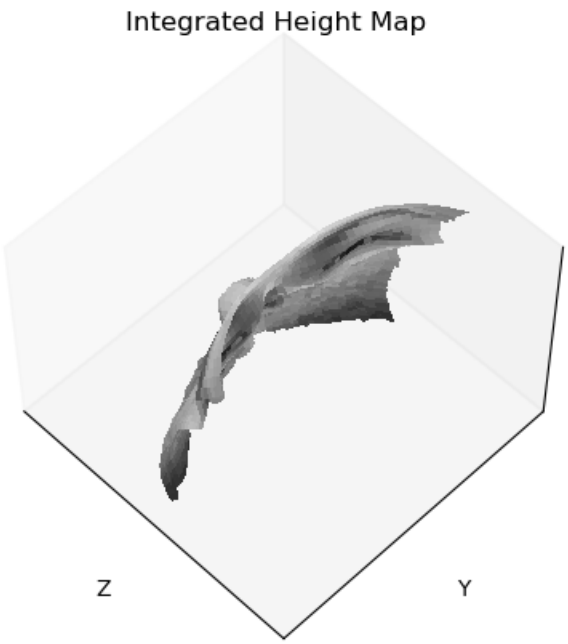


Figure 10: YaleB01 with view: (135,55)

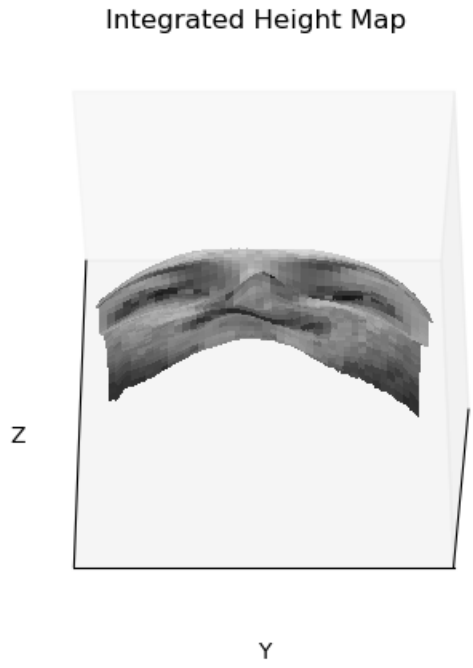


Figure 11: YaleB01 with view: (180,55)

Figure 12: YaleB01 with view: (90,20)

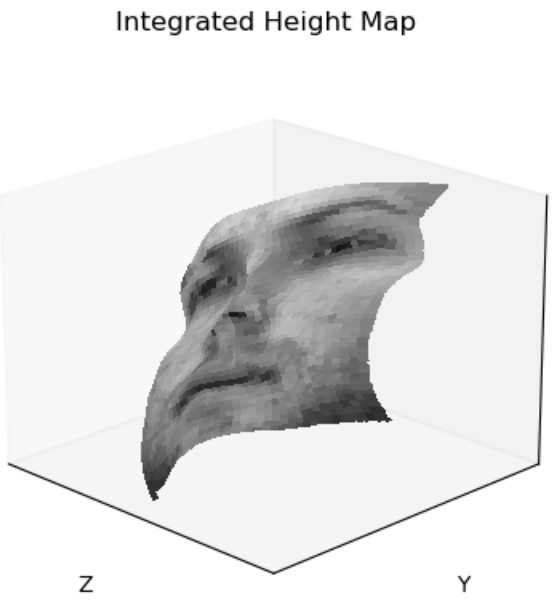


Figure 13: YaleB01 with view: (135,20)

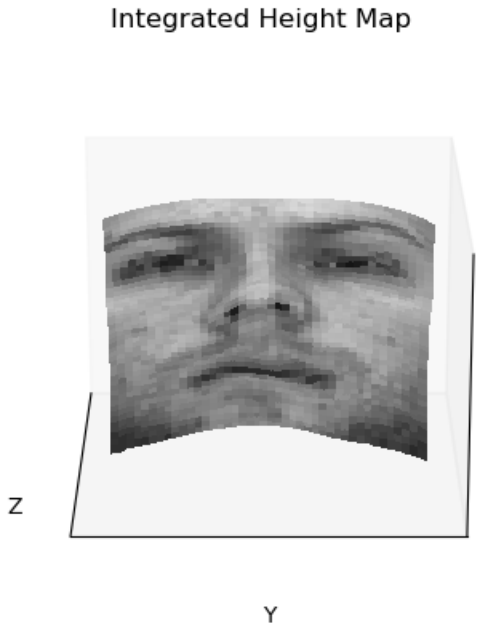


Figure 14: YaleB01 with view: (180,20)

Figure 15: YaleB02 Estimated Albedo

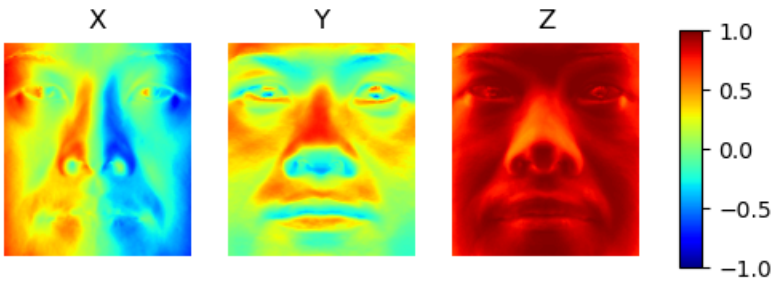


Figure 16: YaleB02 Estimated Normals

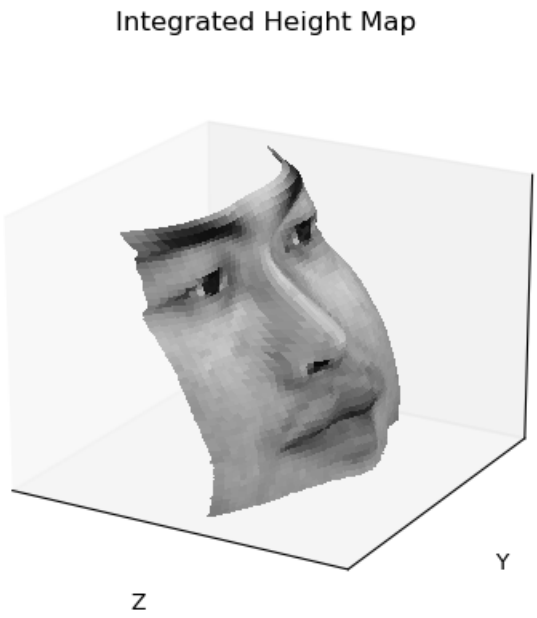


Figure 17: YaleB02 with view: (-60,20)

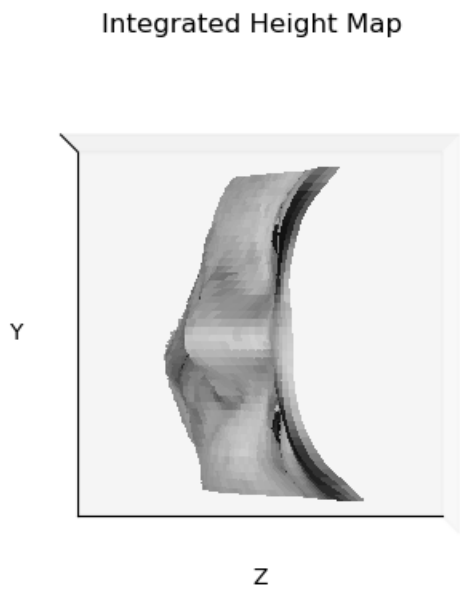


Figure 18: YaleB02 with view: (90,90)

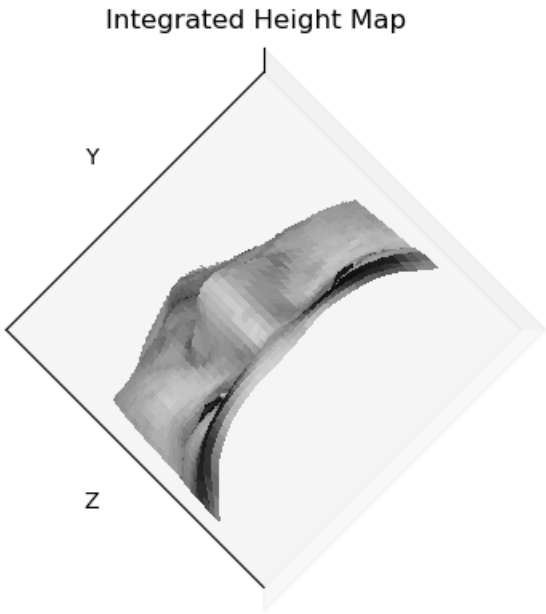


Figure 19: YaleB02 with view: (135,90)

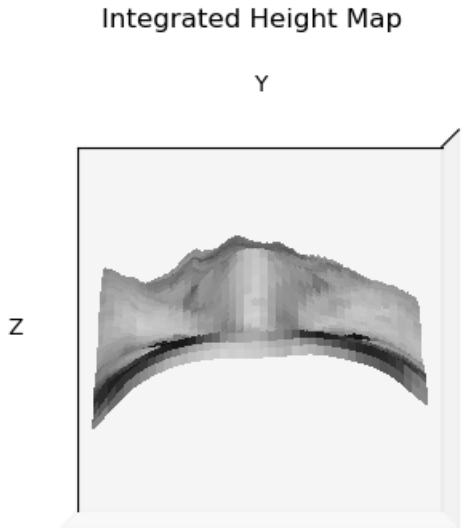


Figure 20: YaleB02 with view: (180,90)

Figure 21: YaleB02 with view: (90,55)

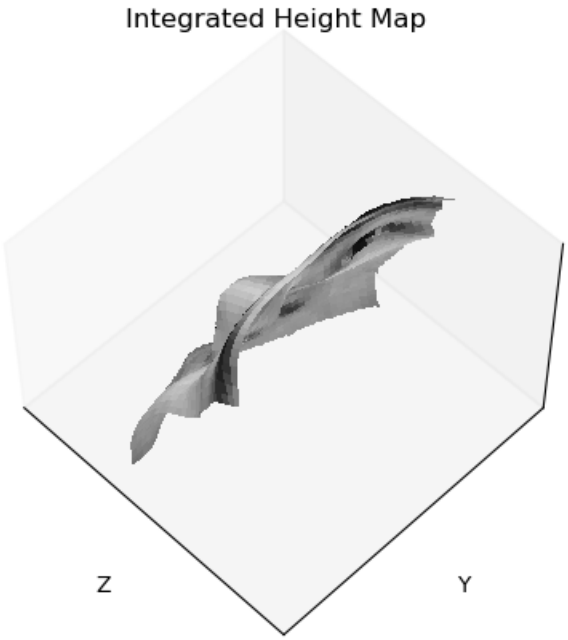


Figure 22: YaleB02 with view: (135,55)

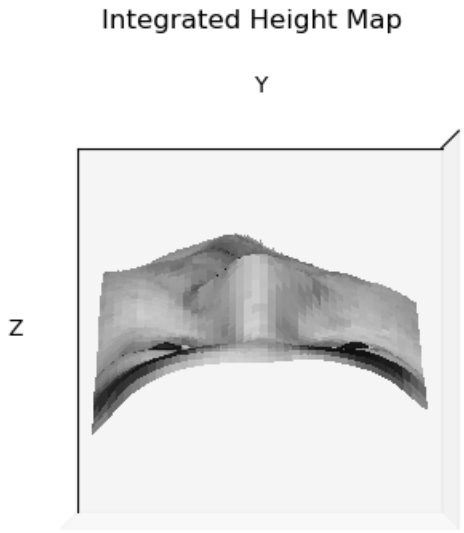


Figure 23: YaleB02 with view: (180,55)

Figure 24: YaleB02 with view: (90,20)

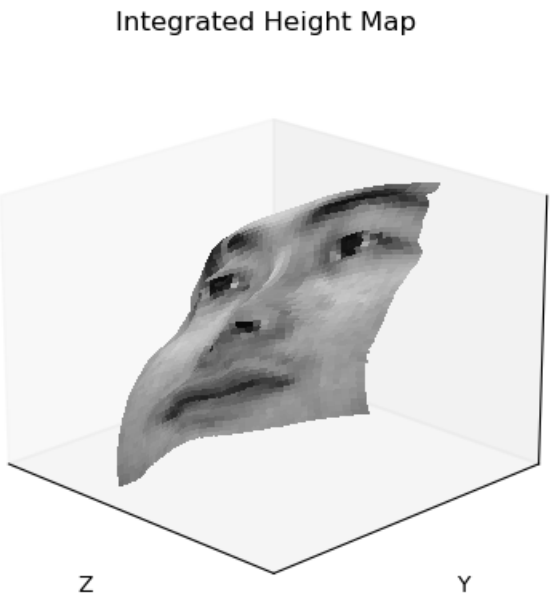


Figure 25: YaleB02 with view: (135,20)

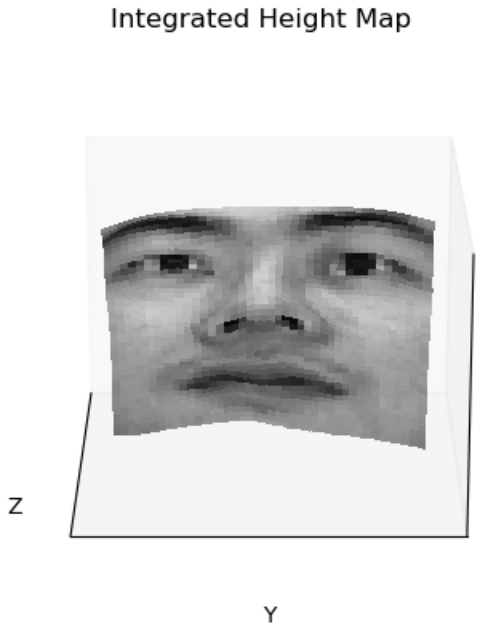


Figure 26: YaleB02 with view: (180,20)

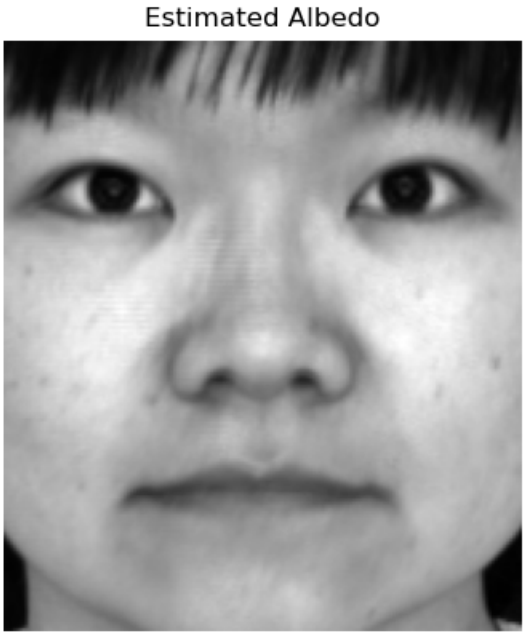


Figure 27: YaleB05 Estimated Albedo

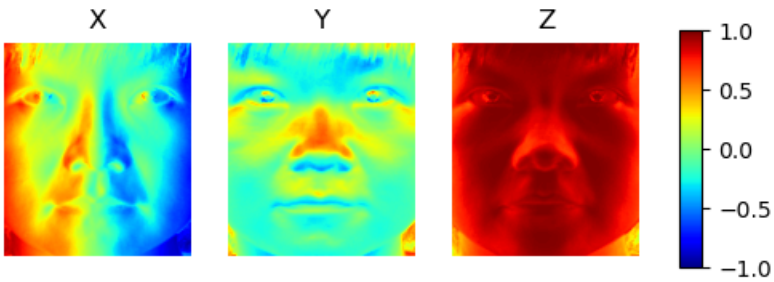


Figure 28: YaleB05 Estimated Normals

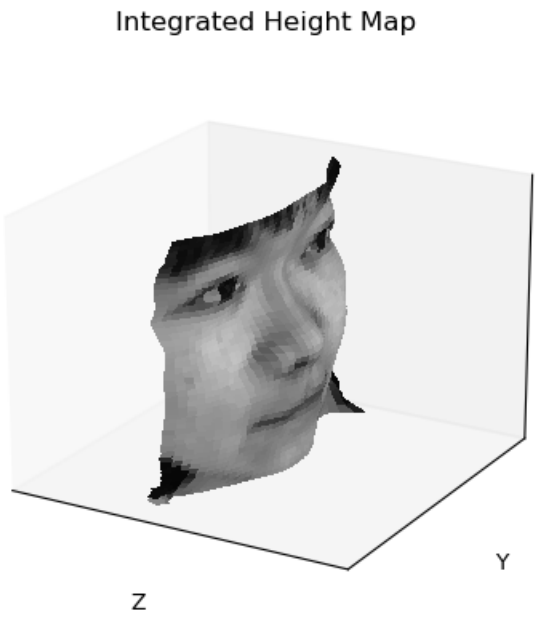


Figure 29: YaleB05 with view: (-60,20)

Figure 30: YaleB05 with view: (90,90)

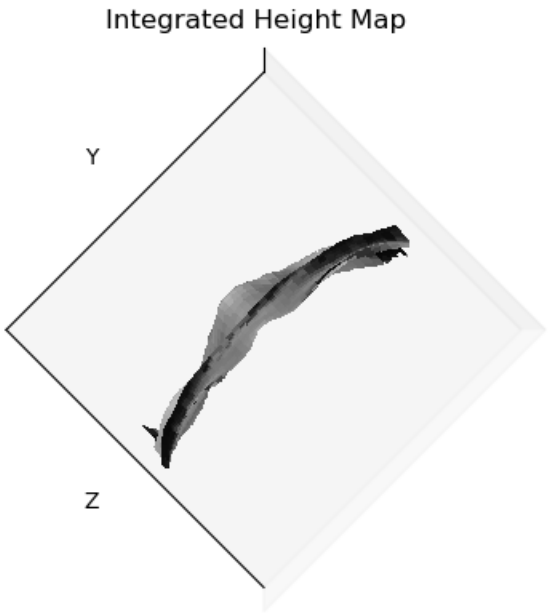


Figure 31: YaleB05 with view: (135,90)

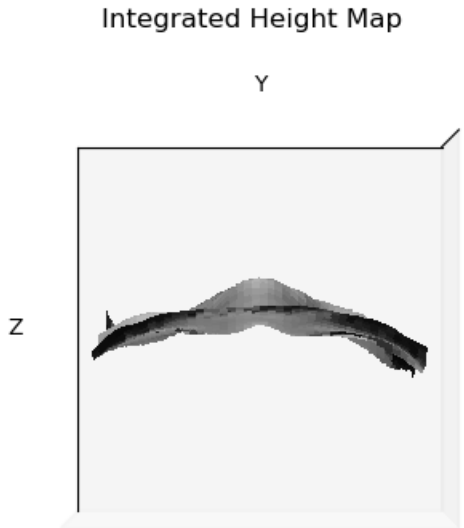


Figure 32: YaleB05 with view: (180,90)

Figure 33: YaleB05 with view: (90,55)

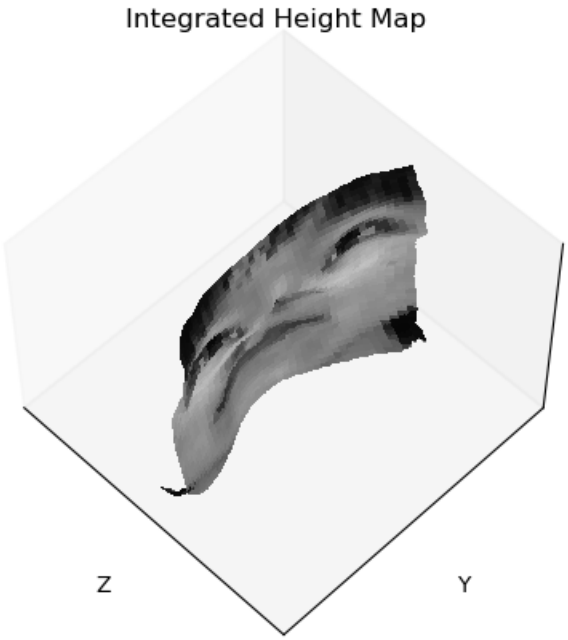


Figure 34: YaleB05 with view: (135,55)

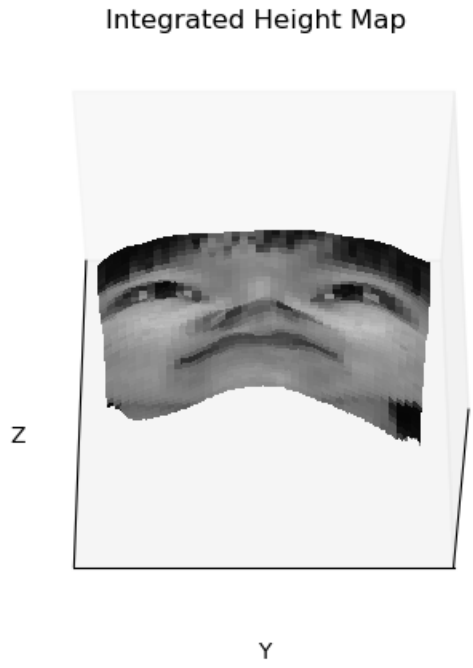


Figure 35: YaleB05 with view: (180,55)

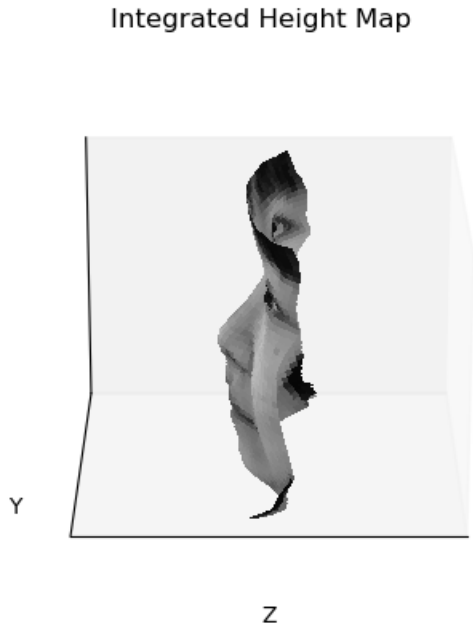


Figure 36: YaleB05 with view: (90,20)

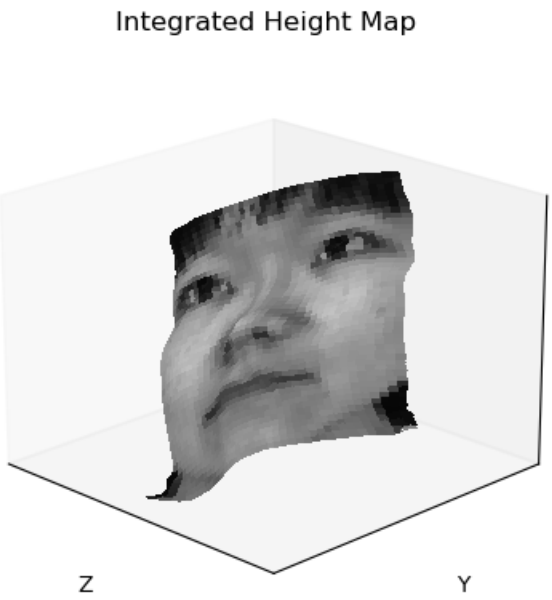


Figure 37: YaleB05 with view: (135,20)

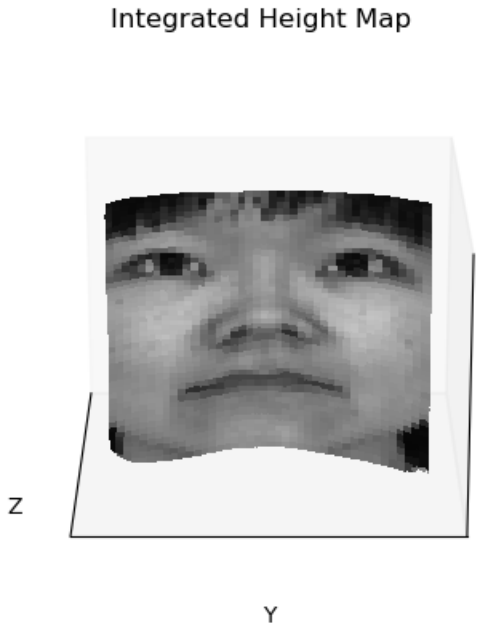


Figure 38: YaleB05 with view: (180,20)

Figure 39: YaleB07 Estimated Albedo

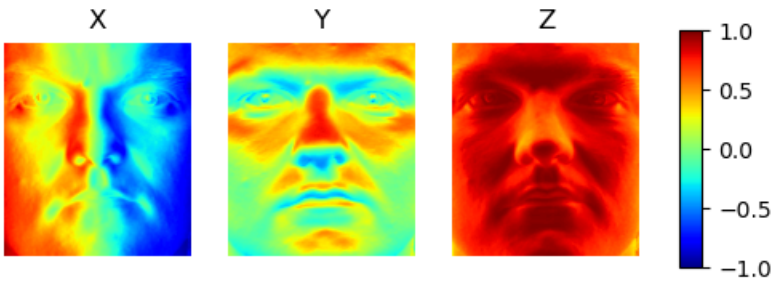


Figure 40: YaleB07 Estimated Normals

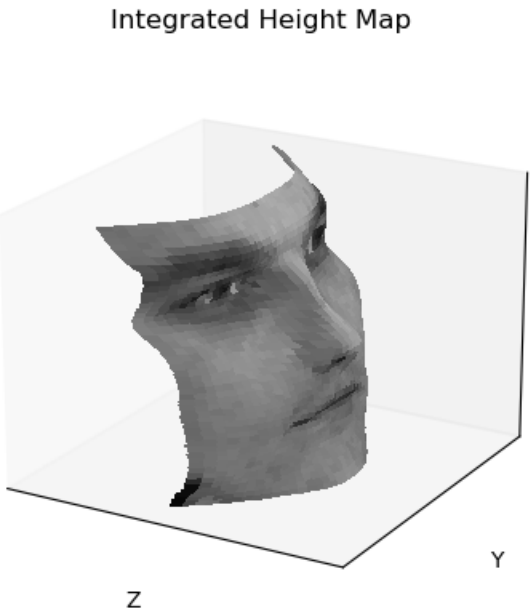


Figure 41: YaleB07 with view: (-60,20)

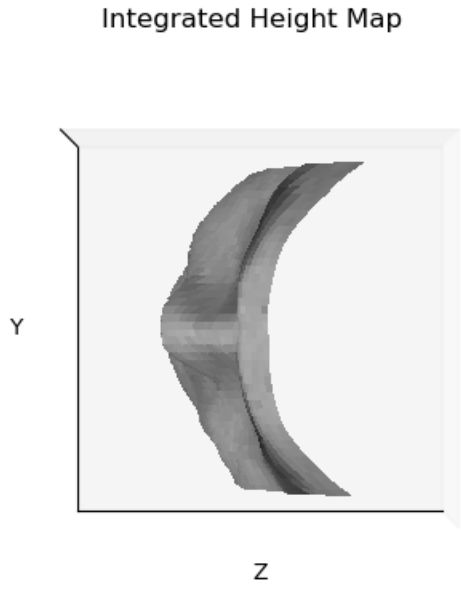


Figure 42: YaleB07 with view: (90,90)

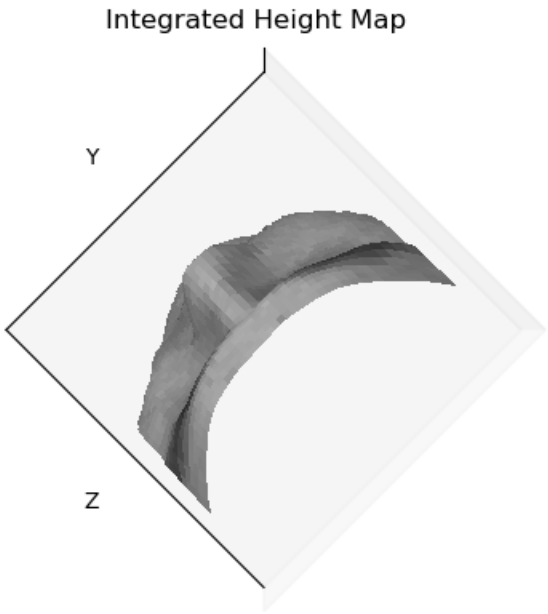


Figure 43: YaleB07 with view: (135,90)

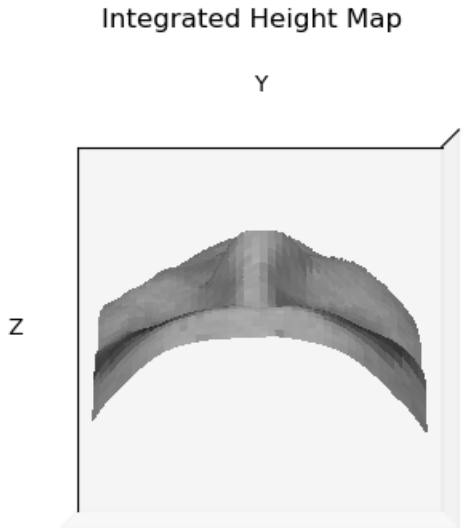


Figure 44: YaleB07 with view: (180,90)

Figure 45: YaleB07 with view: (90,55)

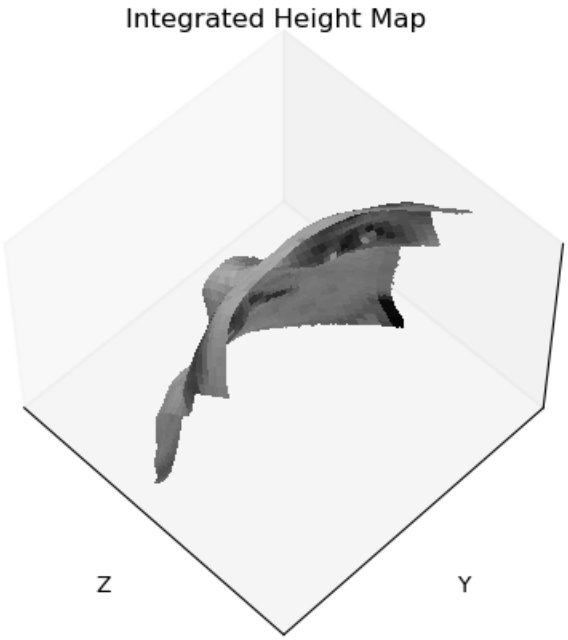


Figure 46: YaleB07 with view: (135,55)

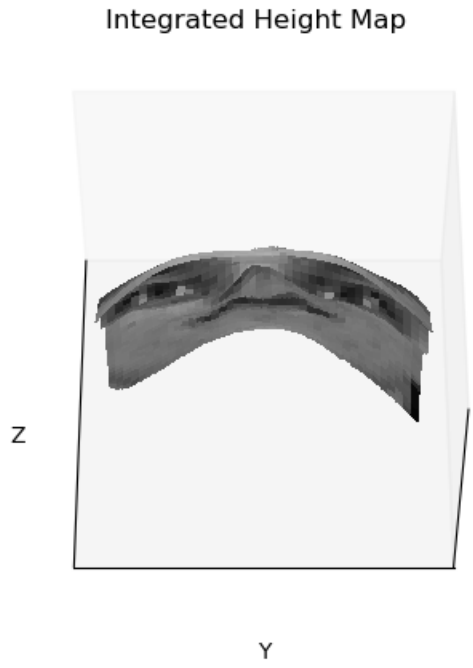


Figure 47: YaleB07 with view: (180,55)

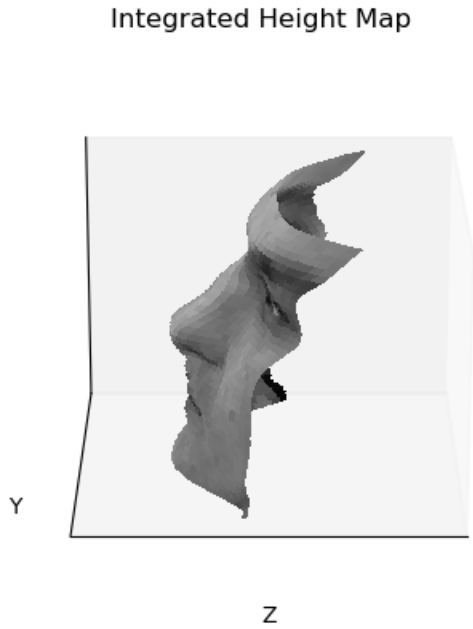


Figure 48: YaleB07 with view: (90,20)

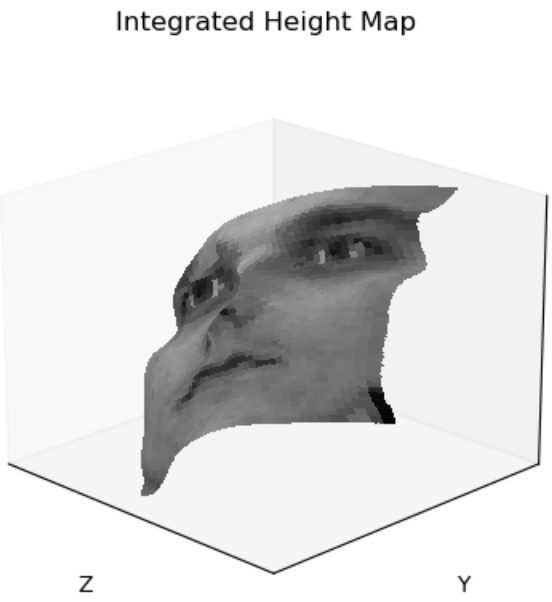


Figure 49: YaleB07 with view: (135,20)

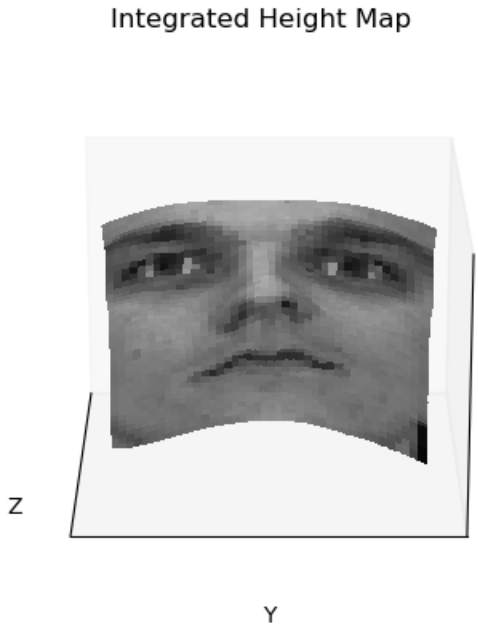


Figure 50: YaleB07 with view: (180,20)

2.5 Discuss the differences between the different integration methods you have implemented.Also, discuss how the Yale Face data violate the assumptions of the shape-from shading method covered in the slides

Although 'row-column' and 'column-row' integration methods are the most straightforward, they may introduce ridges in the reconstruction.

For the YaleB02 subject, the 'row-column' method starts from the center and integrates along the x-axis (a partial derivative of x constraints the horizontal centerline) to produce a smooth horizontal centerline. Then, the height map at other locations is determined by integrating the partial derivative of y vertically (all vertical lines are smooth). In this process, 'row-column' integration loses its smoothness along the x-axis and has vertical ridges.

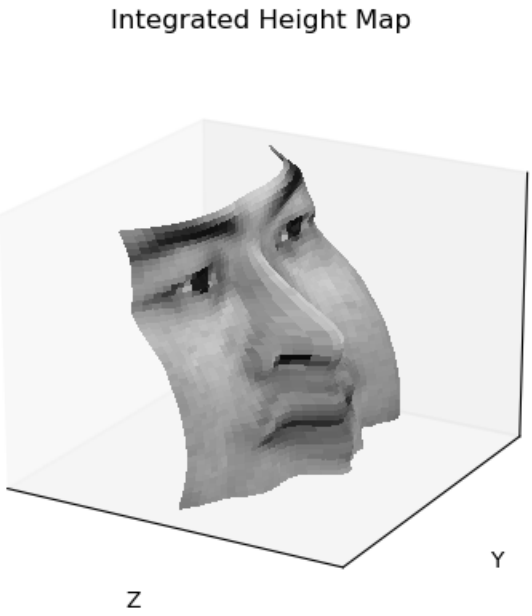


Figure 51: YaleB02 Reconstruction using row-column integration

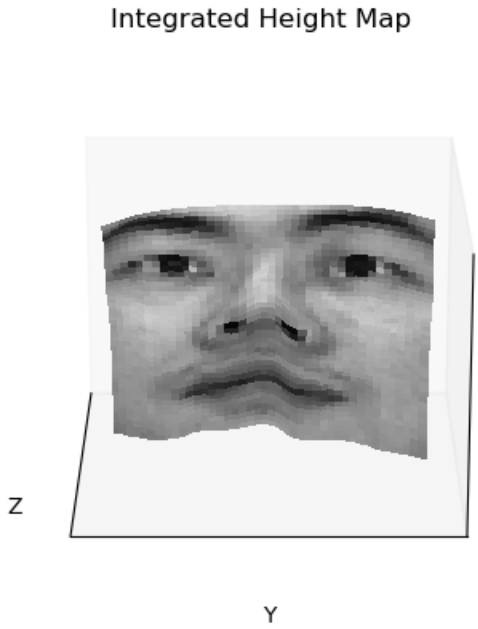


Figure 52: YaleB02 Reconstruction using row-column integration

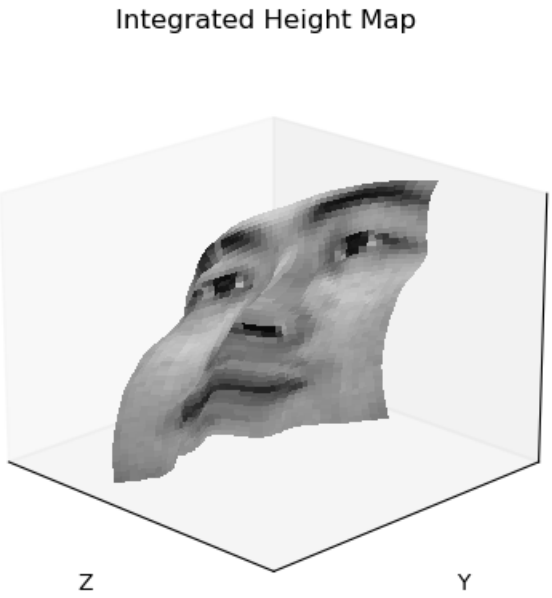


Figure 53: YaleB02 Reconstruction using row-column integration

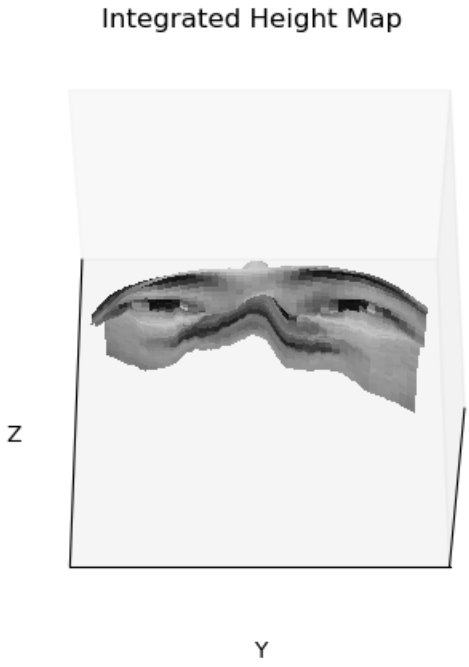


Figure 54: YaleB02 Reconstruction using row-column integration

On similar lines, the 'column-row' method starts from the center and integrates along the y-axis, to produce a smooth vertical centerline. The height map at other locations is determined by the partial derivative of x horizontally (all horizontal lines are smooth). In this process,

’column-row’ integration loses its smoothness along the y-axis and has horizontal ridges.

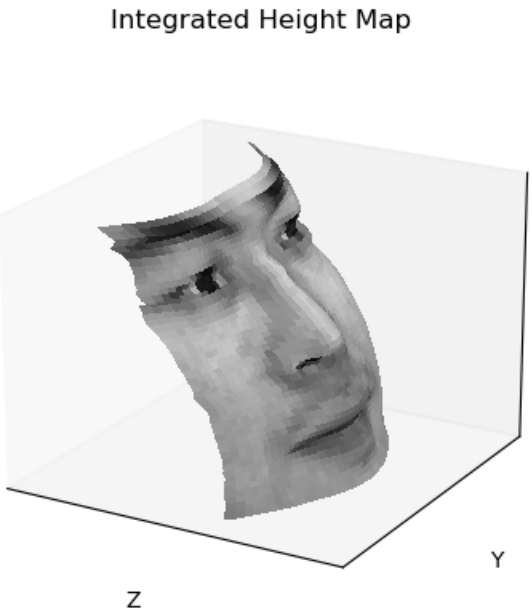


Figure 55: YaleB02 Reconstruction using column-row integration

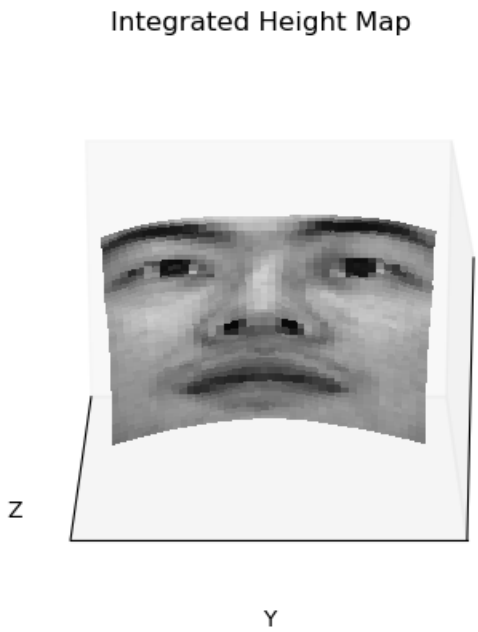


Figure 56: YaleB02 Reconstruction using column-row integration

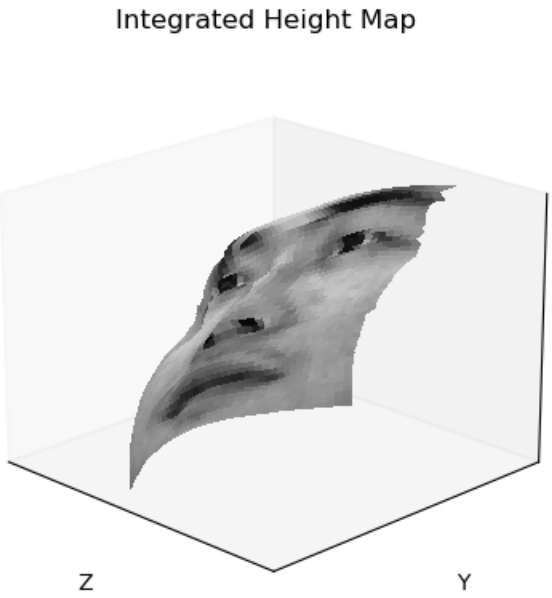


Figure 57: YaleB02 Reconstruction using column-row integration

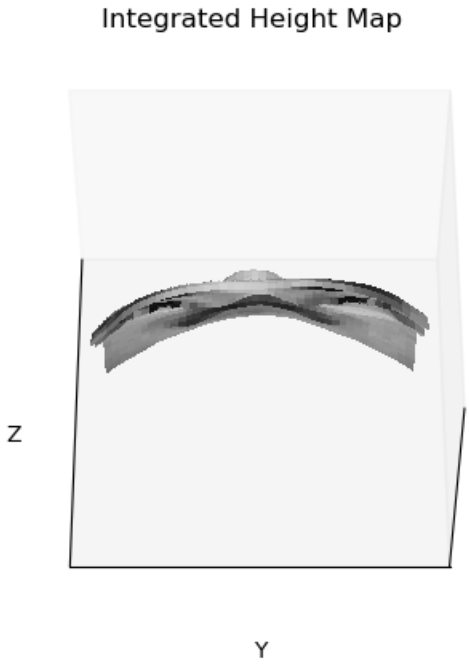


Figure 58: YaleB02 Reconstruction using column-row integration

The average method combines the 'row-column' and 'column-row' methods to return the 2D constraints partial derivative of x and y for all surface locations. Ridges were less relatively but the did not disapper completely. More prominent ridges were also observed. The following

four figures from average show a better reconstruction for the YaleB02 subject.

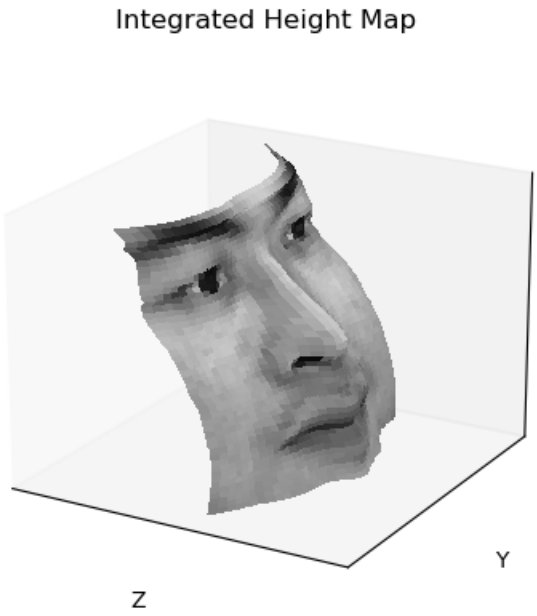


Figure 59: YaleB02 Reconstruction using average integration

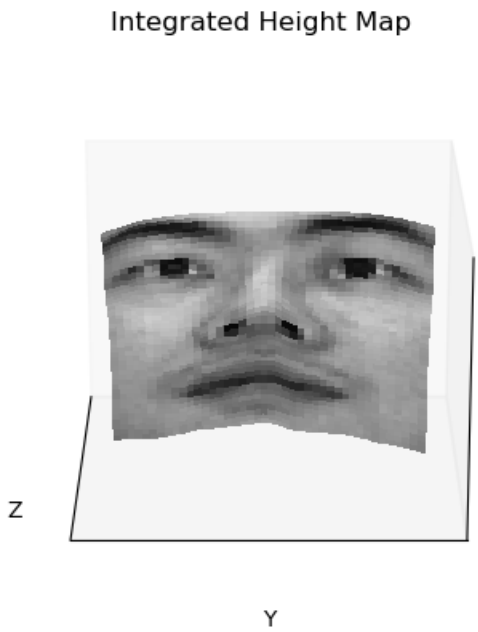


Figure 60: YaleB02 Reconstruction using average integration

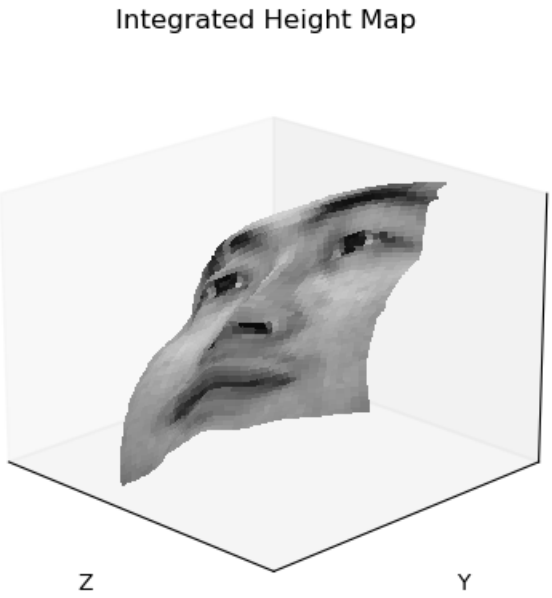


Figure 61: YaleB02 Reconstruction using average integration

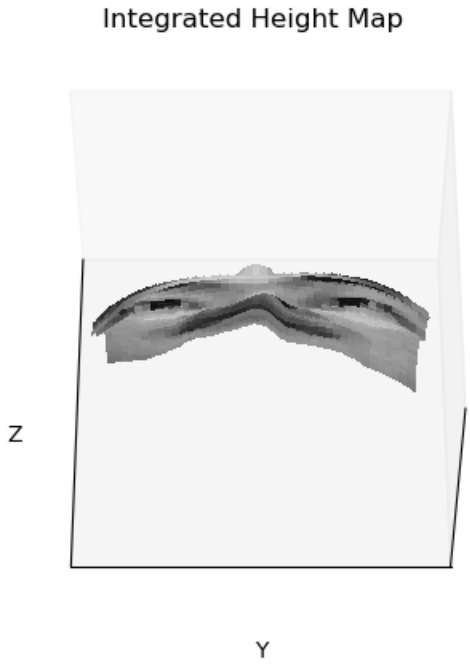


Figure 62: YaleB02 Reconstruction using average integration

Using this random method, coins were flipped to generate random paths (random starting points) to reach the destination coordinates Horizontal ridges near the cheekbone and vertical ridges near the lip are successful smoothed out in the sample shown below. Hence this random

implementation outperforms the other four for face cases as quality of surface increases.The reasoning being it is not specific to one particular gradient.Following four figures for the YaleB02 subject show random integration.

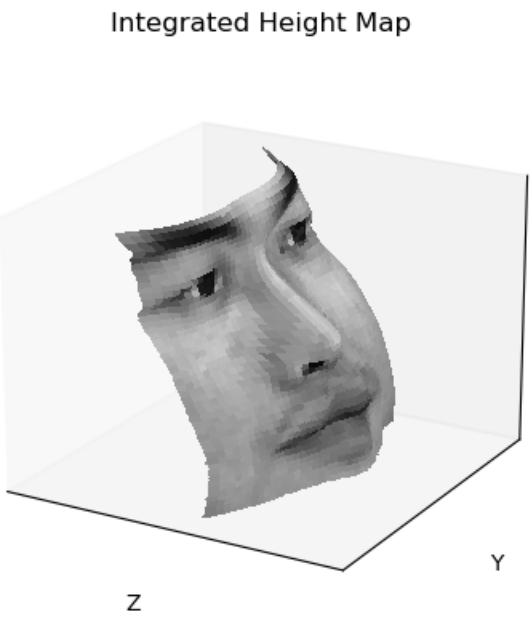


Figure 63: YaleB02 Reconstruction using Random integration

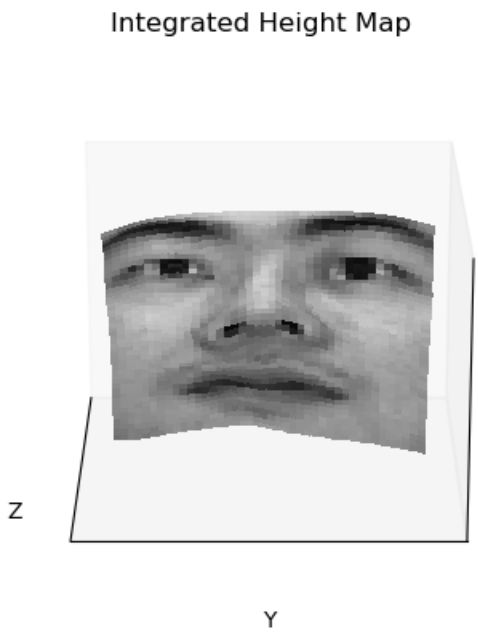


Figure 64: YaleB02 Reconstruction using Random integration

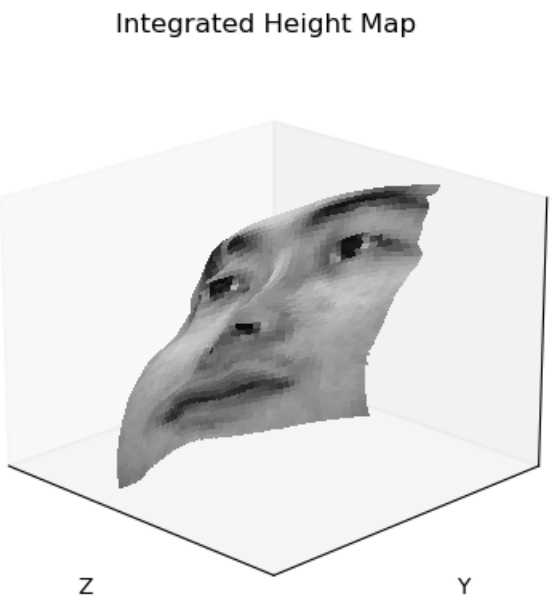


Figure 65: YaleB02 Reconstruction using Random integration

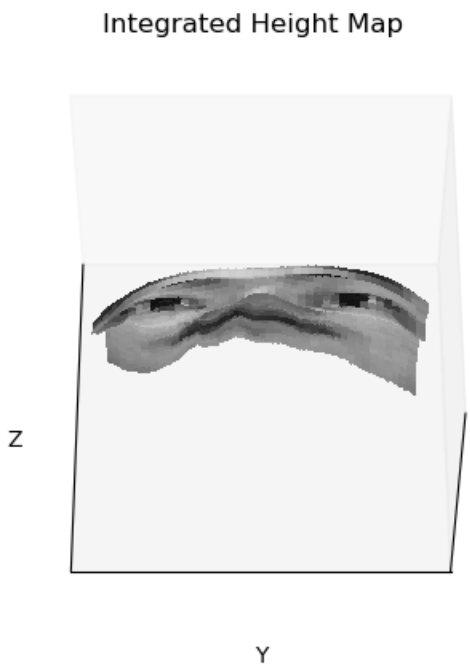


Figure 66: YaleB02 Reconstruction using Random integration

The Yale Face data violate the assumptions of the shape-from-shading method in the followings ways:

1. Since there is an assumption of Lambert's law is assumed in the reconstruction, the shiny/-

glowy/oily parts like eyeballs, skin with makeup, etc. do not satisfy the equal diffuse reflection of light in all directions. Therefore it is strongly recommended to have closing eyes while taking pictures to produce a better reconstruction.

2. In this dataset, a local shading model is applied, implying that each point on the surface receives light only from visible sources at that particular point. (Does not receive light from other point's reflection)

3. This dataset is not accurately reconstructed in the shadow areas (chin, neck, face edges, hair, etc.

4. Moreover, integration of gradient field requires the surface's curvature to be not large and not exceed the pixel's resolution.

3 Aligning Prokudin-Gorskii images

3.1 Implementation of alignChannels.py

The code first centrally crops images by ignoring borders and then performs green shift followed by red shift which creates color image in RGB format.The final result is cropped to remove boundary artifacts.

Shifting is performed by computing score which can be measured by the metrics **SSD** or **CS** referring to Sum of Squared Differences and Cosine Similarity respectively.

There is also an option of using sobel filter while using edges in score computation.

```
import numpy as np

# importing libraries for saving images in lighter memory format and edge
    detection
import skimage as sk
from scipy.ndimage import sobel

def alignChannels(img, max_shift):
    #convert to double to save memory
    img = sk.img_as_float(img)

    #separate color channels
    #Blue channel
    channel1 = img[:, :, 2]
    #Green channel
    channel2 = img[:, :, 1]
    #Red channel
    channel3 = img[:, :, 0]

    #aligning a central crop of the images to ignore borders
    img_c1 = channel1[int(0.15 * len(channel1)):-int(0.15 * len(channel1)), int
    (0.15 * len(channel1[0])):-int(0.15 * len(channel1[0]))]
    img_c2 = channel2[int(0.15 * len(channel2)):-int(0.15 * len(channel2)), int
    (0.15 * len(channel2[0])):-int(0.15 * len(channel2[0]))]
    img_c3 = channel3[int(0.15 * len(channel3)):-int(0.15 * len(channel3)), int
    (0.15 * len(channel3[0])):-int(0.15 * len(channel3[0]))]


    best_score1 = -float('inf')
```

```
    best_score2 = -float('inf')
    best_shift1 = [0, 0]
    best_shift2 = [0, 0]

    #using cosine similarity and ignoring edge
    method = 'CS'
    use_edge = False

    #perfroming green shift by looping over all different displacement permutations wrt to blue channel
    for i in range(-max_shift[0], max_shift[0] + 1):
        for j in range(-max_shift[1], max_shift[1] + 1):
            current_score1 = compute_score(np.roll(img_c2, (i, j), (0, 1)), img_c1, method, use_edge)
            if current_score1 > best_score1:
                best_score1 = current_score1
                best_shift1 = [i, j]

    shifted_img1 = np.roll(channel2, best_shift1, (0, 1))

    #perfroming red shift by looping over all the different displacement permutations wrt to blue channel
    for i in range(-max_shift[0], max_shift[1] + 1):
        for j in range(-max_shift[0], max_shift[1] + 1):
            current_score2 = compute_score(np.roll(img_c3, (i, j), (0, 1)), img_c1, method, use_edge)
            if current_score2 > best_score2:
                best_score2 = current_score2
                best_shift2 = [i, j]

    shifted_img2 = np.roll(channel3, best_shift2, (0, 1))

    #creates color image in RGB format - red shift, green shift & blue channel
    new_shifted_img = np.dstack([shifted_img2,shifted_img1,channel1])

    #crops final result to remove boundary artifacts
    colour_img = new_shifted_img[int(0.1 * len(new_shifted_img)):-int(0.1 * len(new_shifted_img)), int(0.1 * len(new_shifted_img[0])):-int(0.1 * len(new_shifted_img[0]))]

    #returns best displaced image along with the displacement vector
    return colour_img, np.vstack((best_shift1,best_shift2))

def compute_score(image1, image2, method, use_edge):
    #returns similarity score of image1 and image2
    if use_edge:
    #computes sobel filter by considering boundaries
        image1 = np.abs(sobel(image1))
        image2 = np.abs(sobel(image2))

    if method == 'SSD':
    #computes Sum of Squared Differences(SSD) between two images
        return np.sum((np.array(image1, dtype=np.float32) - np.array(image2, dtype=np.float32))**2)

    elif method == 'CS':
    #computes Cosine Similarity(CS) between two images
        image1 = np.ndarray.flatten(image1)
```

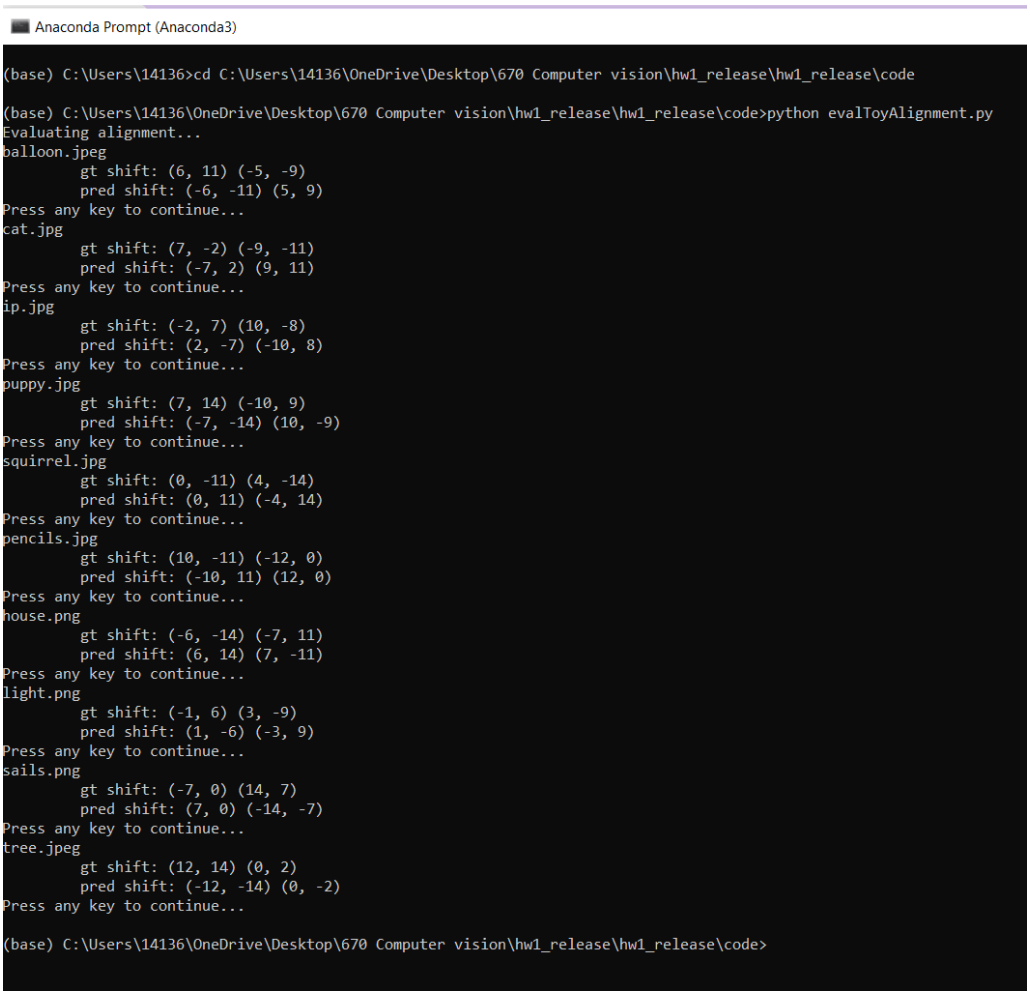
```
        image2 = np.ndarray.flatten(image2)
        return np.dot(image1 / np.linalg.norm(image1), image2 / np.linalg.norm(
    image2))
```

3.2 Verification of evalToyAlignment.py for toy example

As seen from the output figure [], evalToyAlignment.py correctly recovers the color image and shifts for the toy example . This is implied by the predicted shifts which are equal to the negation of the gt shifts.

It is observed that output images are visually constructed much better in alignments and more robust when using Cosine Similarity rather than Sum of Squared Differences as seen in case of balloons.jpeg in figure [] and [] respectively.

The channels are cropped 10 percent during image matching while looping over all the different displacement permutations using Cosine Similarity. The final colored image is cropped by 10 percent from the edges to remove boundary artifacts.



```
Anaconda Prompt (Anaconda3)

(base) C:\Users\14136>cd C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw1_release\hw1_release\code

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw1_release\hw1_release\code>python evalToyAlignment.py
Evaluating alignment...
balloon.jpeg
        gt shift: (6, 11) (-5, -9)
        pred shift: (-6, -11) (5, 9)
Press any key to continue...
cat.jpg
        gt shift: (7, -2) (-9, -11)
        pred shift: (-7, 2) (9, 11)
Press any key to continue...
ip.jpg
        gt shift: (-2, 7) (10, -8)
        pred shift: (2, -7) (-10, 8)
Press any key to continue...
puppy.jpg
        gt shift: (7, 14) (-10, 9)
        pred shift: (-7, -14) (10, -9)
Press any key to continue...
squirrel.jpg
        gt shift: (0, -11) (4, -14)
        pred shift: (0, 11) (-4, 14)
Press any key to continue...
pencils.jpg
        gt shift: (10, -11) (-12, 0)
        pred shift: (-10, 11) (12, 0)
Press any key to continue...
nouse.png
        gt shift: (-6, -14) (-7, 11)
        pred shift: (6, 14) (7, -11)
Press any key to continue...
light.png
        gt shift: (-1, 6) (3, -9)
        pred shift: (1, -6) (-3, 9)
Press any key to continue...
sails.png
        gt shift: (-7, 0) (14, 7)
        pred shift: (7, 0) (-14, -7)
Press any key to continue...
tree.jpeg
        gt shift: (12, 14) (0, 2)
        pred shift: (-12, -14) (0, -2)
Press any key to continue...

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw1_release\hw1_release\code>
```

Figure 67: Output of evalToyAlignment.py

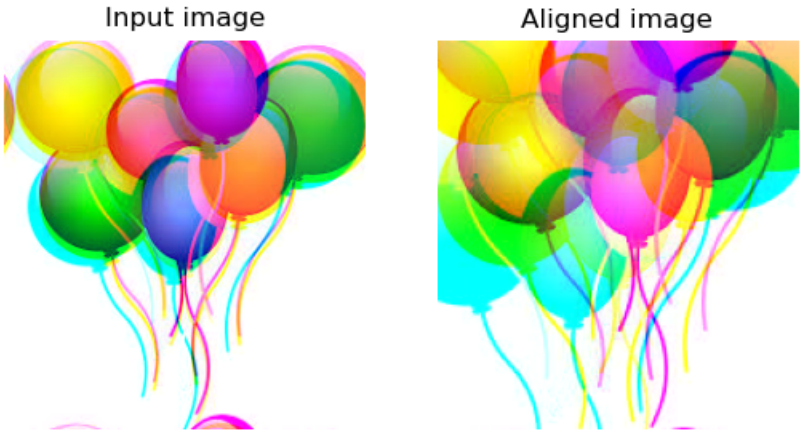


Figure 68: Output image: Balloon.jpeg using SSD

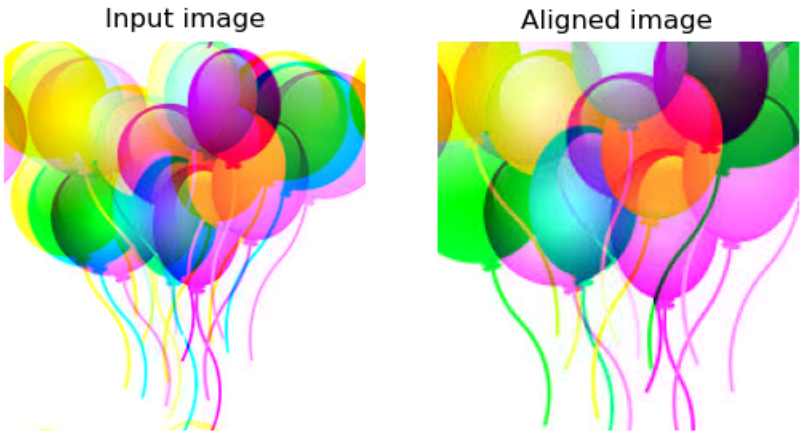


Figure 69: Output image: Balloon.jpeg using Cosine Similarity

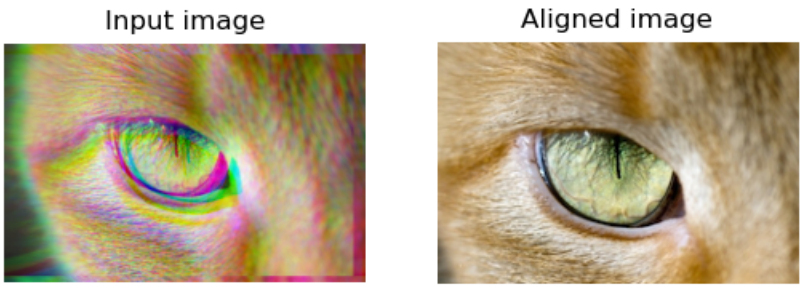


Figure 70: Output image: cat.jpg using Cosine Similarity

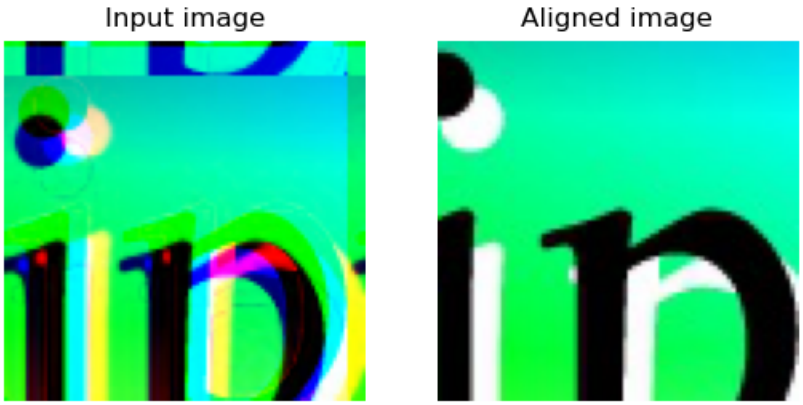


Figure 71: Output image: ip.jpg using Cosine Similarity

Figure 72: Output image: puppy.jpg using Cosine Similarity

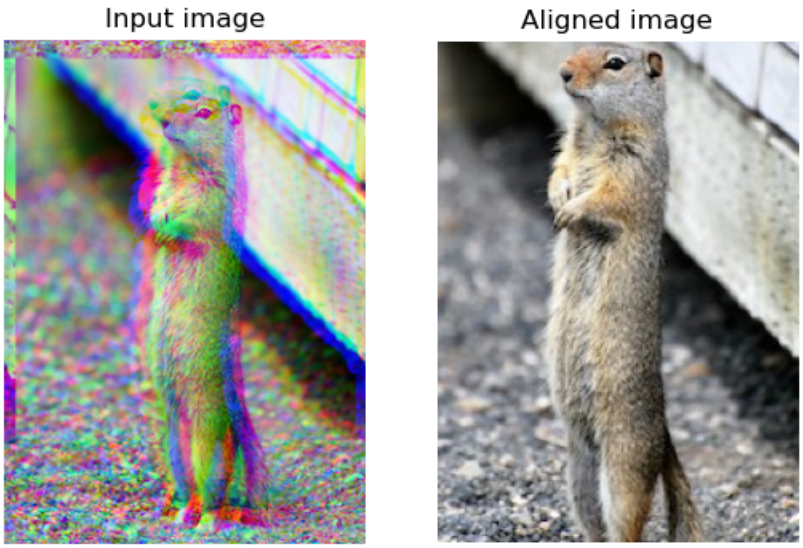


Figure 73: Output image: squirrel.jpg using Cosine Similarity

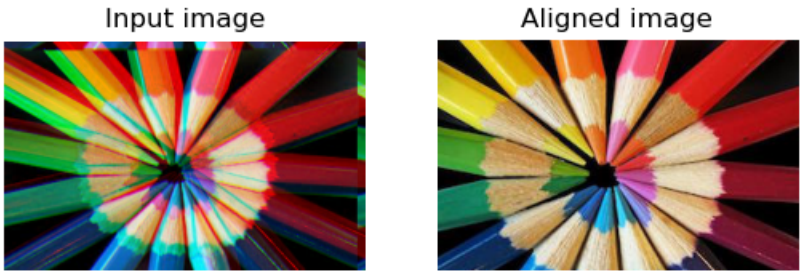


Figure 74: Output image: pencils.jpg using Cosine Similarity

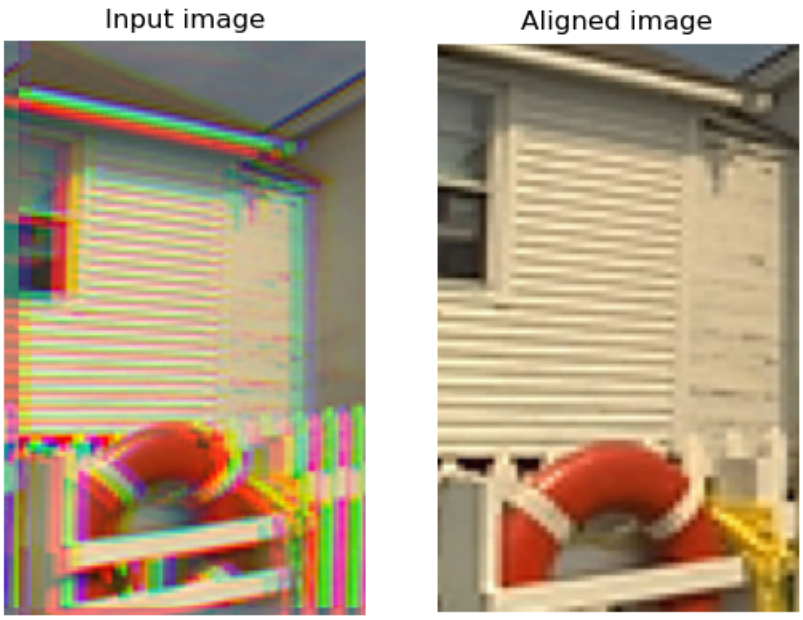


Figure 75: Output image: house.png using Cosine Similarity

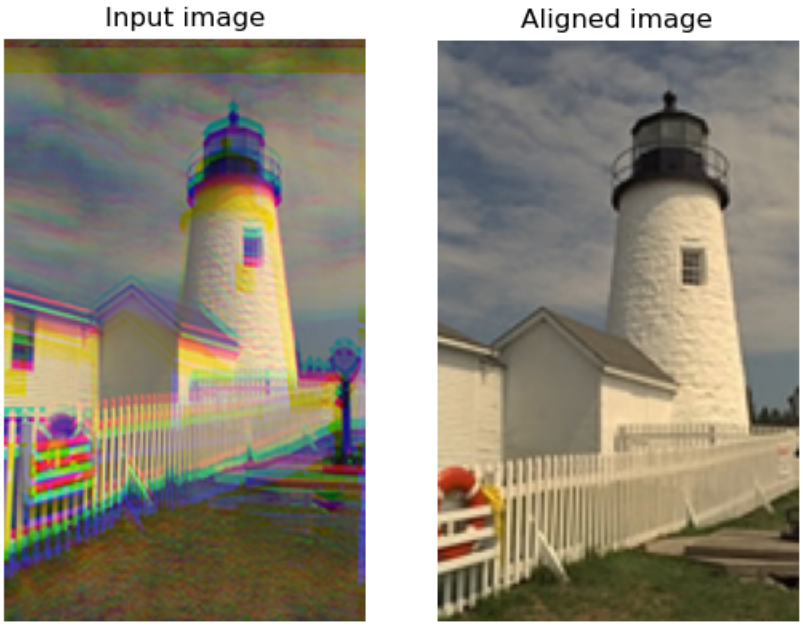


Figure 76: Output image: light.png using Cosine Similarity

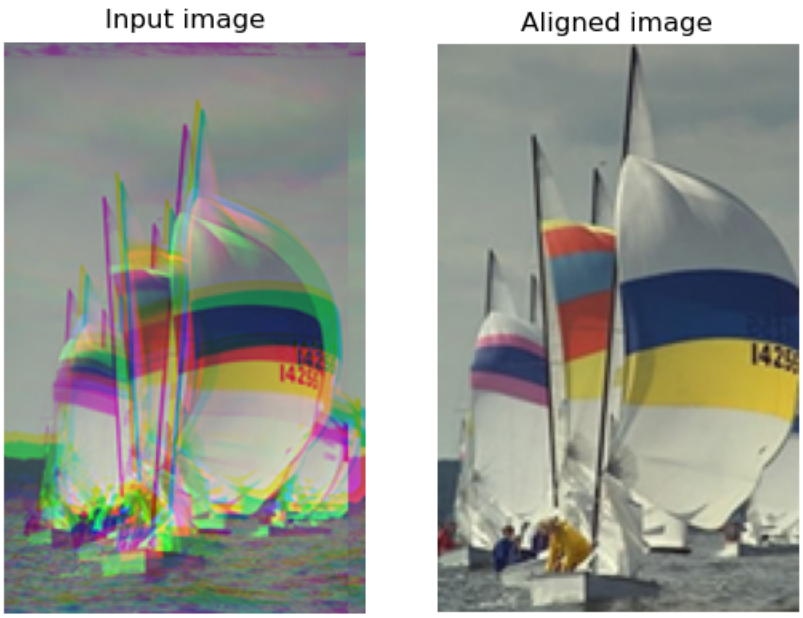


Figure 77: Output image: snails.png using Cosine Similarity

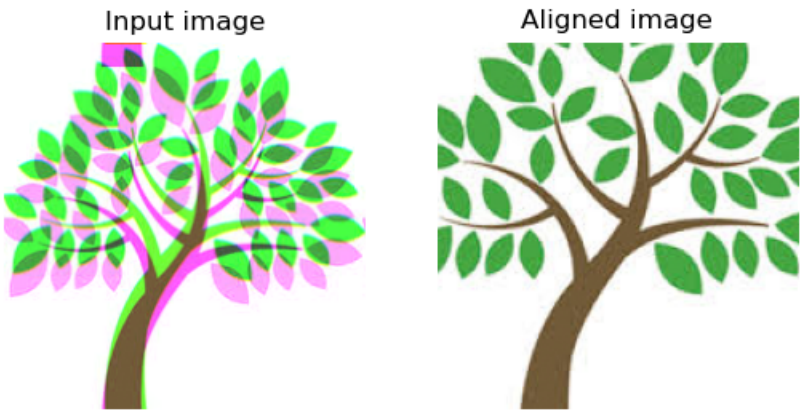


Figure 78: Output image: tree.jpeg using Cosine Similarity

3.3 Results of evalProkudinAlignment.py

It is observed that the color images produced from **Prokudin-Gorskii Collection** are in good alignment when using Cosine Similarity metric for scoring computation for possible displacements.

The channels are cropped 15 percent during image matching while looping over all the different displacement permutations. The final colored image is cropped by 10 percent from the edges to remove boundary artifacts.

The computed shifted vectors can be found in the *figure[] : Output of evalProkudinAlignment.py*

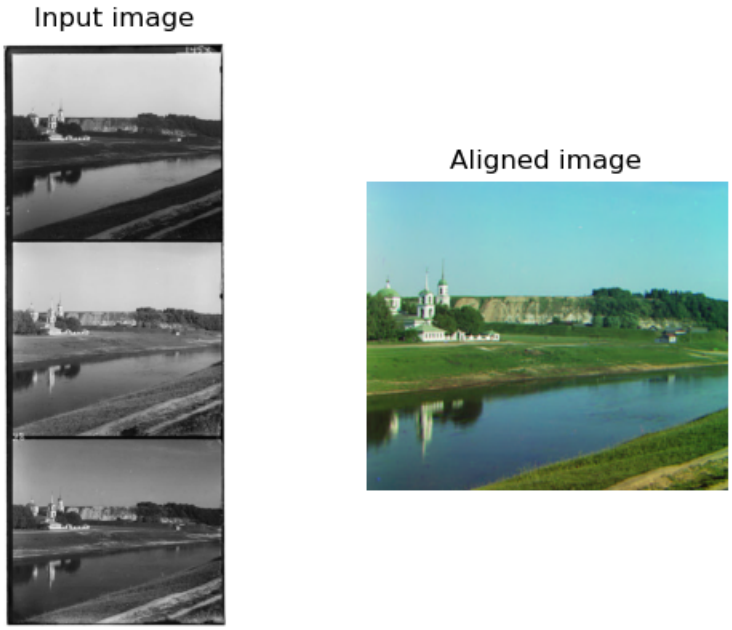


Figure 79: Output Image 1 using Cosine Similarity - Shift: G(5, 2) R(9, 1)

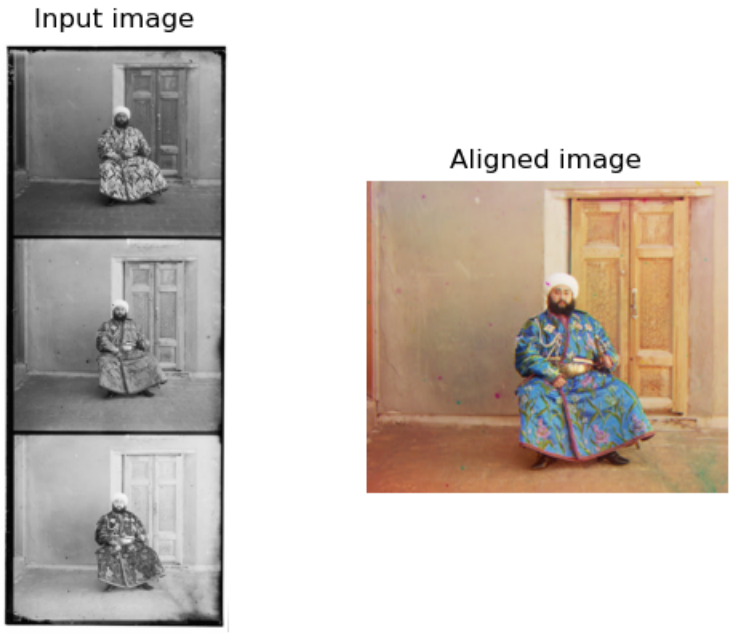


Figure 80: Output Image 2 using Cosine Similarity - Shift: G(7, 3) R(14, 5)

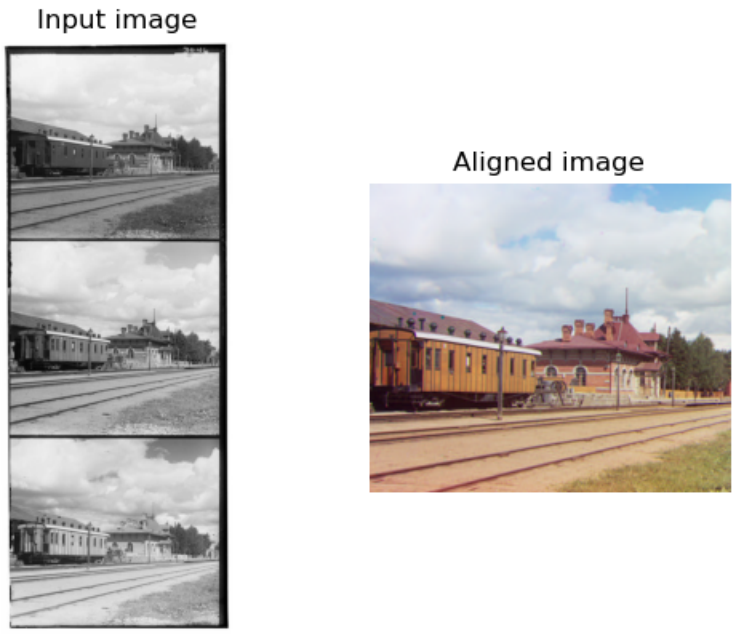


Figure 81: Output Image 3 using Cosine Similarity - Shift: G(5, 3) R(11, 4)

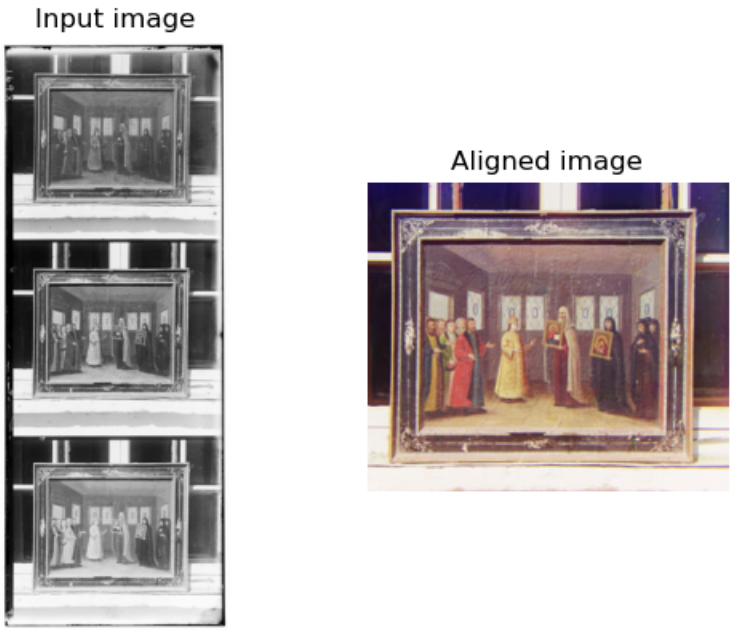


Figure 82: Output Image 4 using Cosine Similarity - Shift: G(4, 2) R(9, 2)

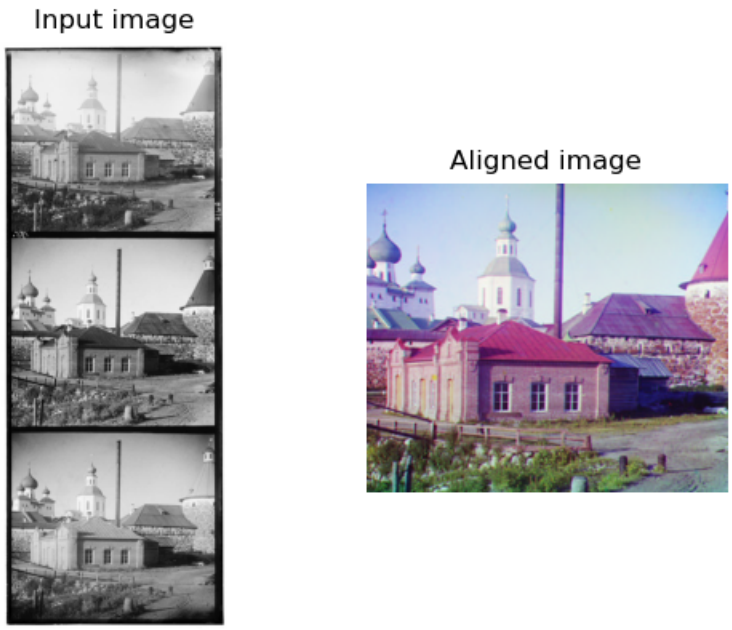


Figure 83: Output Image 5 using Cosine Similarity - Shift: G(4, 1) R(13, 1)

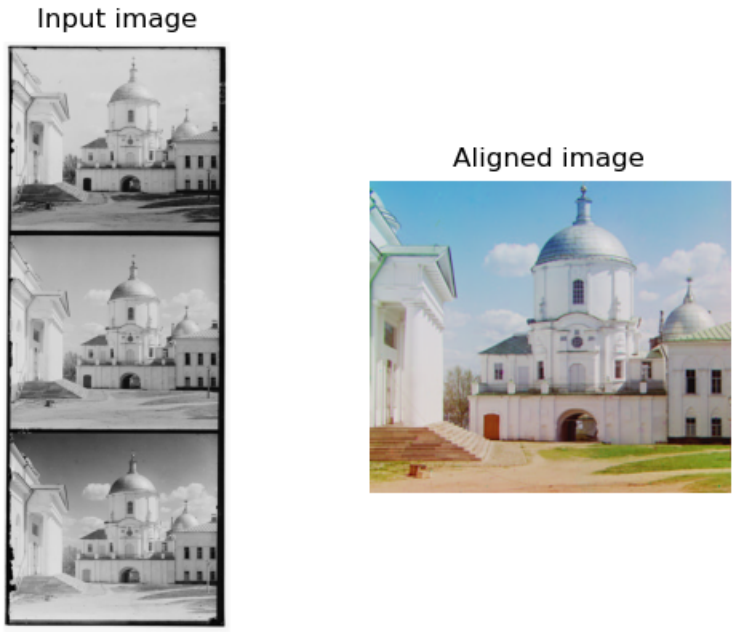
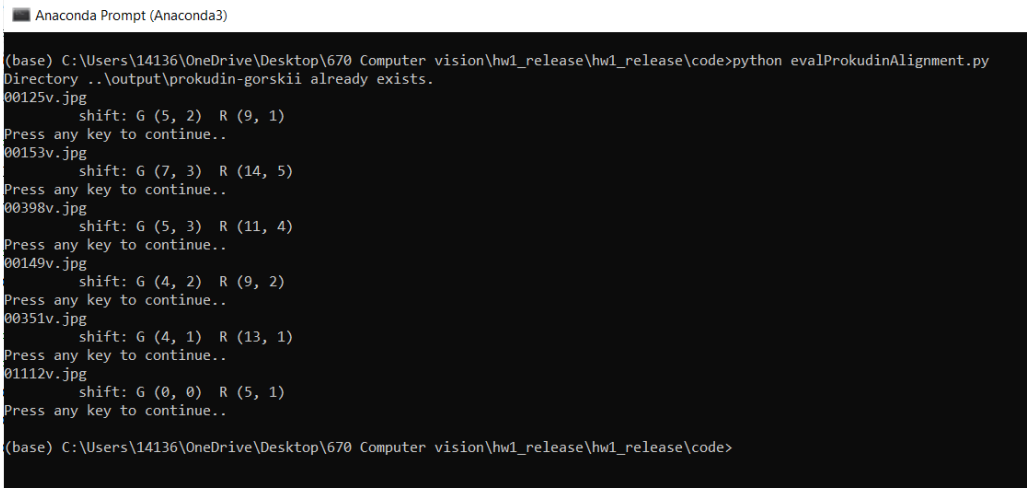


Figure 84: Output Image 6 using Cosine Similarity - Shift: G(0, 0) R(5, 1)



```
Anaconda Prompt (Anaconda3)

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw1_release\hw1_release\code>python evalProkudinAlignment.py
Directory ..\output\prokudin-gorskii already exists.
00125v.jpg
        shift: G (5, 2)  R (9, 1)
Press any key to continue..
00153v.jpg
        shift: G (7, 3)  R (14, 5)
Press any key to continue..
00398v.jpg
        shift: G (5, 3)  R (11, 4)
Press any key to continue..
00149v.jpg
        shift: G (4, 2)  R (9, 2)
Press any key to continue..
00351v.jpg
        shift: G (4, 1)  R (13, 1)
Press any key to continue..
01112v.jpg
        shift: G (0, 0)  R (5, 1)
Press any key to continue..

(base) C:\Users\14136\OneDrive\Desktop\670 Computer vision\hw1_release\hw1_release\code>
```

Figure 85: Output of evalProkudinAlignment.py