Computer Vision
Mini-project 1:
Aligning Prokudin-Gorskii Images & Photometric Stereo
CMP SCI 670, Fall 2019

Name: Kunjal Panchal
Student ID: 32126469
Email: kpanchal@umass.edu

September 23, 2019

Contents

1 Aligning Prokudin-Gorskii images

1.1 Overview

In order to align the RGB channels of the Prokudin filter images, we need to keep few things in mind; these points also depicts the step by step algorithm:

- INPUTS photographs of each plate
- OUTPUTS an aligned color image
- ONE MORE THING TO TAKE CARE OF when channels are shifted, the boundaries won't be aligned and will produce some unseemly artifacts, we try to remove that

The objectives are roughly stated, now we can go into more details.

- First, we focus on removing the bias we might be introducing through also comparing the PIXELS AT IMAGE BOUNDARY; as stated, those artifacts might be quite different and have potential to skew the results greatly. Hence, we CROP the core part out of the boundary artifacts.
- Second, we extract the RGB CHANNELS of the Prokudin images; keeping in mind that the original images were in BGR ORDER.
- Third, we decide a MINIMUM VALUE which is big enough that the dissimilarity will always be less than that. This is to get an initial value which might be consecutively compared with the dissimilarity variable.
- Fourth thing we take care of is which dissimilarity function we are using: simplest one is just the L2 norm also known as the Sum of Squared Differences (SSD).
- Then, we compare the RED CHANNEL AND BLUE CHANNEL of cropped image WITH THE GREEN CHANNEL of cropped image [we are fixing the green channel here]. The distance is matched with the current smallest distance.

- If the new distance we found is the smallest one, then we take not of the shift value and keep looping through the rest of the possible shifts to ascertain which ones are the shift values for both horizontal edges for both channels.
- That we store in `predicted shift` 2 x 2 matrix. Now we know the shift values.
- Now, with the predicted shift values, we use `circushift()` Matlab function to shift the image matrix in left or right direction.
- There we will get the new Red and Blue channels, from which we can get AN ALIGNED IMAGE.
- Again, we remove the boundary artifacts by finding how many edge pixels have a very different mean value than the pixels at relatively core part of the image, we CROP OUT THOSE EDGE PIXELS.
- And that's how we will implement the `alignChannels()` function.

1.2 Align the Channels

We approximately decide the boudary of the image which will be under the influence of halo effect or some other kinds of artifact. Following code snippet depicts how we achieved that:

```
1 %% Crop the image so we don't consider boundary
     differences, since they will be under artifact
     effect
2 padding = floor([.2*size(im, 2), .2*size(im, 1), .6*
     size(im, 2), .6*size(im, 1)]);
3 im_crop = imcrop(im, padding);
```

* complete code in Appendix A.1

The assumption is that the 20% [even more than that will work since alignment of core part will make sure that the whole image will be aligned, but for the sake of robustness, 20% works good] of every edge will be under influence of artifacts, so we get the prediction about shift values without those boundaries and then align the image with that boundary.

We decide a big value for minimum then go about finding the shift which minimizes the difference between two channels. Here we show an example of a loop comparing Red and Green channels:

```
%%Match Red and Green Channels
for i = -maxShift(1):maxShift(1)
    for j = -maxShift(1):maxShift(1)
        temp = circshift(redChannel, [i, j]);
        error = sum(sum((greenChannel-temp).^2));
        if min > error
            min = error;
            predShift(1,1) = -i;
            predShift(1,2) = -j;
            correctedGreenChannel = circshift(im
                (:,:,1), [i, j]);
        end
    end
end
```

* complete code in Appendix A.1

We will get $predictedshift(1,:)$, which are 2, horizontal shift values for the red channels. $predictedshift(2,:)$, which are 2, horizontal shift values for the blue channels. We reconstruct the image:

```
%% Image we get after aligning three channels
imShift = cat(3, correctedGreenChannel, im(:,:,2),
    correctedBlueChannel);
```

* complete code in Appendix A.1

1.3 About Boundary Artifacts

Now to remove the boundary artifact, we detect what kind of pixel intensities those boundary pixels will have, then we create a mask which blocks all those edge:

```
h = size(imShift,1);     %height
w = size(imShift,2);     %width
c = size(imShift,3);     %channels
c_img = imShift(1:floor(h/10), 1:floor(w/10), :);
```

```
5
6  vert = 0;
7  horz = 0;
8
9  % For all 3 channels
10 for i = 1:c
11     % Find edges
12     temp = edge(c_img(:,:,i), 'canny', 0.1);    %
           Detect edges
13
14     % Find mean value and mask the values.
15     vert_avg = mean(temp, 1);      %find the average
           of vertical boundaries
16     horz_avg = mean(temp, 2);      %find the average
           of horizontal boundaries
17     threshold_1 = 3*mean(horz_avg); % got horizontal
           threshold
18     threshold_2 = 3*mean(vert_avg); % got vertical
           threshold
19     % only consider the pixels which are higher valued
           than vertical threshold
20     vert_mask = vert_avg > threshold_2;
21     % only consider the pixels which are higher valued
           than horizontal threshold
22     horz_mask = horz_avg > threshold_1;
23
24     % Find last values
25     last_vert = find(vert_mask, 1, 'last');
26     last_horz = find(horz_mask, 1, 'last');
27
28     if last_vert > vert
29         vert = last_vert;
30     end
31     if last_horz > horz
32         horz = last_horz;
33     end
34 end
```

* complete code in Appendix A.1

Cropping the boundary artifacts:

```
%% Final result, cropped the boundaries calculates by
    taking mean and creating a mask
imShift = imcrop(imShift, [horz vert (w-2*horz) (h-2*
    vert)]);
```

* complete code in Appendix A.1

1.4 Results

For Toy Images Evaluating alignment .. See Table 1

Table 1: Toy Image Results

1	balloon.jpeg	gt shift: (-2, 8) (-4, 9)	pred shift: (2, -8) (4, -9)
2	cat.jpg	gt shift: (-10,-2) (0, 5)	pred shift: (10,2) (0, -5)
3	ip.jpg	gt shift: (6,-7) (8, 6)	pred shift: (-15,3) (-8, -6)
4	puppy.jpg	gt shift: (5,-12) (-10, 0)	pred shift: (-5,12) (10, 0)
5	squirrel.jpg	gt shift: (14, 3) (-5,-9)	pred shift: (-14, -3) (5,9)
6	pencils.jpg	gt shift: (8, 0) (-8, 6)	pred shift: (-8, 0) (8, -6)
7	house.png	gt shift: (12, 1) (14,-11)	pred shift: (-12, -1) (-14,11)
8	light.png	gt shift: (-11,11) (-8,-8)	pred shift: (11,-11) (8,8)
9	sails.png	gt shift: (10,13) (-8,-5)	pred shift: (-10,-13) (8,5)
10	tree.jpeg	gt shift: (-9, 4) (-8,-1)	pred shift: (9, -4) (8,1)

For Prokudin Images See Table 2 The images are shown in Figure 1

Table 2: tab:Prokudin Image Results

1(a)	00125v.jpg	R (-4, 1)	B (5, 2)
1(b)	00153v.jpg	R (-7, -2)	B (7, 3)
1(c)	00398v.jpg	R (-6, -1)	B (5, 3)
1(d)	00149v.jpg	R (-5, 0)	B (4, 2)
1(e)	00351v.jpg	R (-9, -1)	B (4, 1)
1(f)	01112v.jpg	R (-5, -1)	B (0, 0)

(a) 00125v.jpg

(b) 00153v.jpg

(c) 00398v.jpg

(d) 00149v.jpg

(e) 00351v.jpg

(f) 01112v.jpg

Figure 1: Aligned Prokudin-Gorskii Images

1.5 Conclusions and Bottom Line

The images are almost accurately aligned. As already in definition of the problem, note that in the case of the Emir of Bukhara (Figure 1(c)), the images to be matched do not actually have the same brightness values (they are different color channels), so the distance between them will not be zero even after alignment.

This shows how recording three exposures of every scene onto a glass plate using a red, a green, and a blue filter was a robust concept and the fact that we can recover color images from that era is a proof to it.

There are still some ugly boundary artifacts visible which leaves space for improvement, but too much overfitting will not work in general cases. Our current boundary artifact removal algorithm works in 4 out of 6 cases which is pretty satisfactory for our problem.

2 Photometric Stereo

Before we explore the photometric stereo and an example of it in detail, we explain some of the common terms we deal with in this application of computer vision:

Photometric Stereo: is a technique in computer vision for estimating the surface normals of objects by observing that object under different lighting conditions.
It is based on the fact that the amount of light reflected by a surface is dependent on the orientation of the surface in relation to the light source and the observer. [1]

Radiometry/ Radiosity: is a set of techniques for measuring electromagnetic radiation, including visible light.
Radiometric techniques in optics characterize the distribution of the radiation's power in space, as opposed to photometric techniques, which characterize the light's interaction with the human eye. [2]

Monge Patch: is a representation from which we can determine a unique point on the surface by giving the image coordinates.

To measure the shape of the surface we need to obtain the depth to the surface. This suggests representing the surface as $(x, y, f(x, y))$.

Notice that to obtain a measurement of a solid object, we would need to reconstruct more than one patch, because we need to observe the back of the object. See Figure 2 [3]

PGM[Portable Gray Map] File Format: is a lowest common denominator grayscale file format. It is designed to be extremely easy to learn and write programs for. [6]

Ambient Component: accounts for the small amount of light that is scattered about the entire scene. [4]

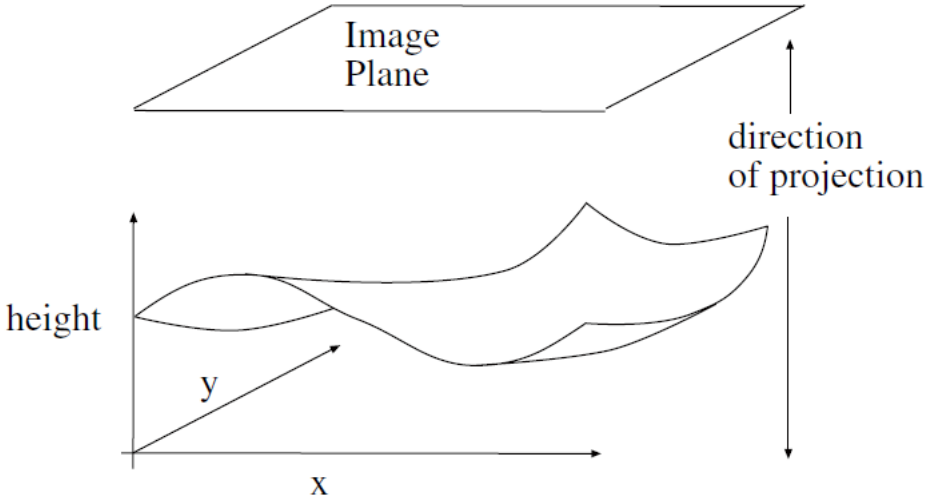


Figure 2: Monge patch is a representation of a piece of surface as a height function. For the photometric stereo example, we assume that an orthographic camera — one that maps (x, y, z) in space to (x, y) in the camera — is viewing a Monge patch. This means that the shape of the surface can be represented as a function of position in the image. [3]

Albedo: is the measure of the diffuse reflection of solar radiation out of the total solar radiation received by an astronomical body. It is dimensionless and measured on a scale from 0 to 1. Surface albedo is defined as the ratio of radiosity to the irradiance received by a surface. [5]

2.1 Overview

Here, we will disuss the system configuration, the setup and the constraints (if any) that might affect our algorithm:

For this shape from shading algorithm,

- INPUTS Set of photographs taken with known lighting directions [around

64 images]; plus one image taken without any light source: Ambient Image
courtesy: Yale Face Database

- OUTPUTS Albedo/ paint, normal direction of the surface, and the height map
- LIGHT SOURCE DIRECTIONS Encoded in file names
- READING the input The images are read by `loadFaceImages()`. Returns the images for the 64 light source directions, an ambient image (i.e., image taken with all the light sources turned off), and the light directions for each image
- INTEGRATION METHOD Row-wise/ Column-wise/ Average/ Random

2.2 Preprocess the Data

We need the images under the illumination of only the point light source. Fortunately, due to linearity of light we can do this by subtracting the ambient image from all the images.

In this section, we discuss `prepareData()` function:

The Matlab code* is as follows:

```
%% Subtracting ambience
output = imArray - ambientImage;
```

* complete code in Appendix B.1

As defined earlier, the ambience contains the lighting information about the entire scene, which is under no influence of particular light sources.

So naturally, they don't contribute in what geometry is generted by some particular light source(s). It is safe to remove that information from our face images so we can be sure that the information we are working on is strictly due to effects of light source(s).

This intermediate result will consist of a pre-processed image which

is free of any ambient and/or environmental bias or noise.

There might be some pixel values that brighter in ambient image than under the light source(s); that case will result in our intermediary result having thos pixel values in negative.

That doesn't make sense in case of double type images (not even in uint8 type images) as the RGB = [0, 0, 0] is the darkest (black) and RGB = [1, 1, 1] is the lightest (white) color one can achieve.

So to remove this anomaly and have all pixel values in acceptable RGB value range, we turn any negative value to 0.

The Matlab code* is as follows:

```
1 %% Negative values in the output matrix will be
     converted to zero
2 output(output<0) = 0;
```

* complete code in Appendix B.1

If there were any negative pixel value, converting them to 0 changes the normalization of the pixel intensities i.e. if for some pixel, intensity was 0.54 with respect to some other pixel being at the intensity -0.23; that relation will no longer hold as -0.23 is now 0.

Thus, all values after enforcing the [0, 1] range must adjust.

That's where we normalize the values.

The Matlab code* is as follows:

```
1 %% To work with double datatypes [0, 1]; we must make
     sure that all the values are normalized with that
     range. Because after changing some negative values
     to 0, the pixel values we get will not be
     normalized as the range is getting changed with the
      lower limit getting converted to 0
2 output = rescale(output);
```

* complete code in Appendix B.1

2.3 Estimate the Albedo and Normals

`photometricStereo()` function takes as input the stack of images and the light source directions, and returns an albedo image and normal directions. The normal directions can be encoded as a three dimensional array of size $h * w * 3$ where the third dimension corresponds to the x, y, and z components of the normal of each pixel.

To calculate the albedo (and then normals), we need to set a linear system of equations which will compute the albedo for each pixel, independently of other pixels.

For that, let's explore lambert's law eq:1 [7]:

Lambert's Law

- KNOWN S_j is Source Vector; $I_j(x, y)$ is Pixel Values of an Image I_j [(x, y) denotes a particular pixel]
- UNKNOWN $N(x, y)$ is Surface Normal; $\rho_(x, y)$ is the albedo
- LAMBERT'S LAW $I_j(x, y)$ = k $\rho_(x, y)$ ($N(x, y)$ ·S_j)
  = ($\rho_(x, y)$ $N(x, y)$)·(kS_j)
  = $g(x, y)$ ·V_j

Thus, the linear equation system looks as follows:

$$\begin{bmatrix} I_1(x,y) \\ I_2(x,y) \\ \dots \\ I_n(x,y) \end{bmatrix} = \begin{bmatrix} V_1^T \\ V_2^T \\ \dots \\ V_n^T \end{bmatrix} g(x,y)$$

(2)

Now we see how this system is implement in Matlab*

```
% Loop through all pixels of the image
for i = 1:h
    for j = 1:w
        % Got a particular pixel at (i, j) from all
           the images
        I = imArray(i, j, 1:n);
        % Converted into a n x 1 column vector
        I = I(:);
        % Calculated g of I(column matrix of pixel
           values) = V(light directions) * g
        g(i,j,:) = I\lightDirs;
        % Calculated the magnitude of g to get the
           albedo pixel values p(x, y) = |g(x, y)|
        p(i,j) = sqrt(g(i,j,1)^2 + g(i,j,2)^2 + g(i,j
           ,3)^2);

        % To calculate surface normals
        fx = g(i, j, 1)/g(i, j, 3); % Partial
           derivative of the surface with respect to x
        fy = g(i, j, 2)/g(i, j, 3); % Partial
           derivative of the surface with respect to y
        temp = sqrt(fx^2 + fy^2 + 1); % Magnitude
        % N(x, y) = (fx, fy, 1)/sqrt(fx^2 + fy^2 + 1)
        surfaceNormals(i, j, :) = [fx/temp, fy/temp,
           1/temp];
    end
end
% Now we got albedo and surface normal for each pixel
   in image
```

* complete code in Appendix B.2

Explanation:

- We loop through all pixels, we get a particular (x, y) pixel from all images and store it in a vector or a column matrix I
- Then we calculate $g(x, y)$ by using the backslash operator \to solve $I(x, y)/V$
- We get $g(x, y)$ which is the product of multiplication of Albedo and Normal Vector

Compute Albedo:

Albedo will be the magnitude of $g(x, y)$

We recover it in Matlab as follows: *

```
% Calculated the magnitude of g to get the albedo
    pixel values p(x, y) = |g(x, y)|
p(i,j) = sqrt(g(i,j,1)^2 + g(i,j,2)^2 + g(i,j,3)^2);
```

* complete code in Appendix B.2

Results are shown in 3

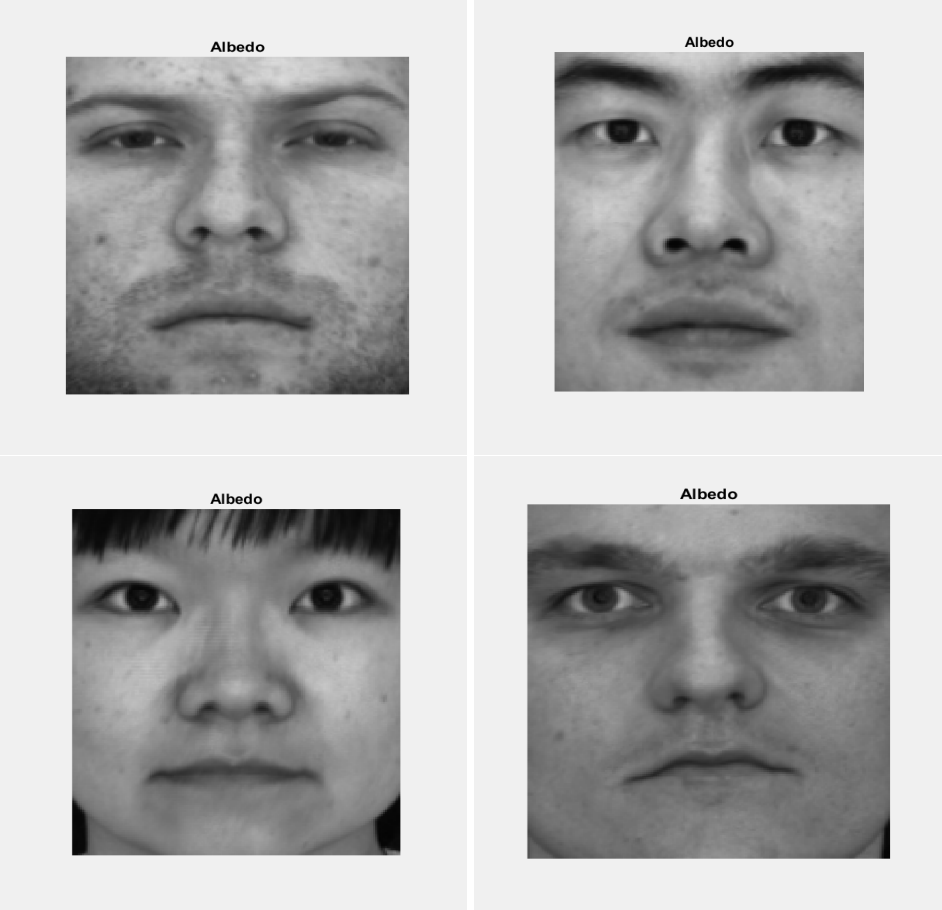


Figure 3: Albedo images for all four test subject faces

Compute Normals:

$N(x, y)$ is the unit normal given by:

$$N(x, y) = g(x, y)/\rho(x, y) \quad (3)$$

The surface is written as $(x, y, f(x, y))$,
If we rewrite the estimated vector g as $g(x, y) = (g_1(x, y), g_2(x, y), g_3(x, y))$
Then we obtain values for the partial derivatives of the surface:

$$f_x(x,y) = g_1(x,y) \,/\, g_3(x,y) \tag{4}$$

$$f_y(x,y) = g_2(x,y) \,/\, g_3(x,y) \tag{5}$$

This means the normal has the form: [7]

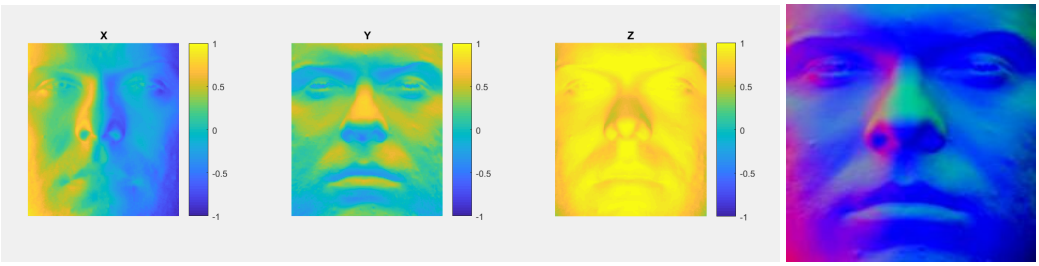
$$N(x, y) = \frac{(f_x, f_y, 1)}{\sqrt{f_x^2 + f_y^2 + 1}} \tag{6}$$

We get normals in Matlab as follows: *

```
% To calculate surface normals
fx = g(i, j, 1)/g(i, j, 3); % Partial derivative of
    the surface with respect to x
fy = g(i, j, 2)/g(i, j, 3); % Partial derivative of
    the surface with respect to y
temp = sqrt(fx^2 + fy^2 + 1); % Magnitude
% N(x, y) = (fx, fy, 1)/sqrt(fx^2 + fy^2 + 1)
surfaceNormals(i, j, :) = [fx/temp, fy/temp, 1/temp];
```

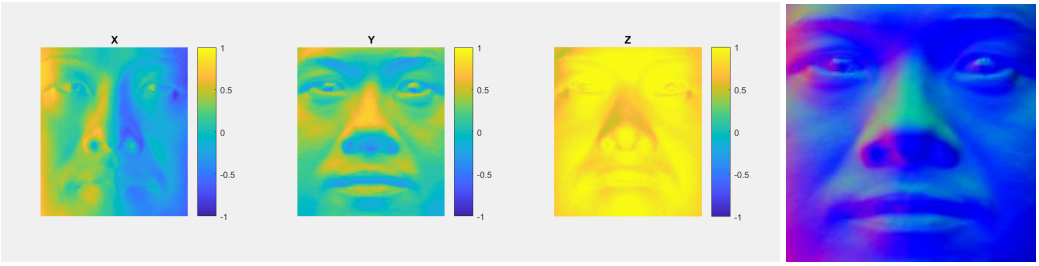
* complete code in Appendix B.2

Results are shown in Figures 2.3, 2.3, 2.3 and 2.3



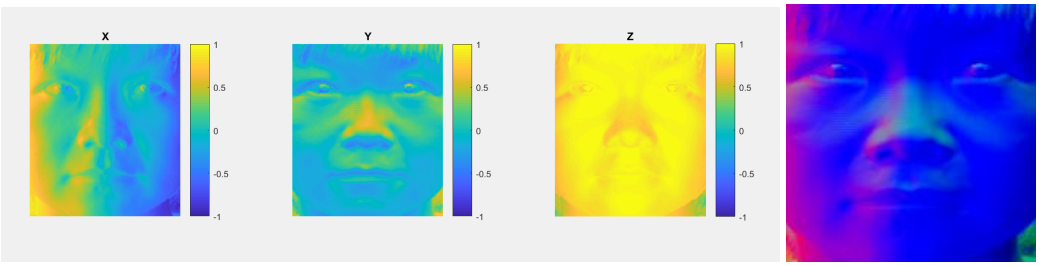
(a) (b)

Figure 4: YaleB01 (a) x, y, z components/ channels of the surface normal depicted in the second image (b) Calculated surface normal from the $g(x, y)$



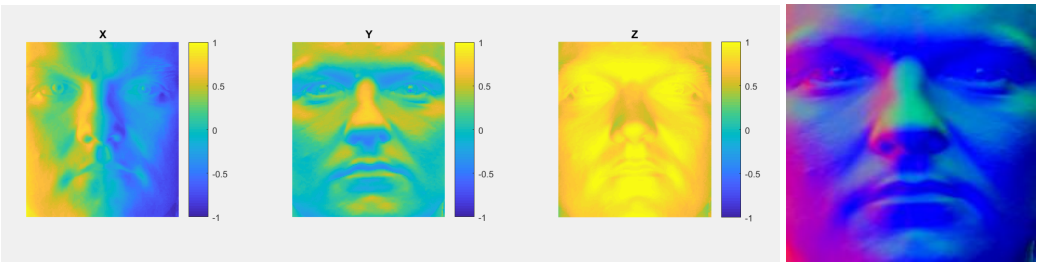
(a) (b)

Figure 5: YaleB02 (a) x, y, z components/ channels of the surface normal depicted in the second image (b) Calculated surface normal from the $g(x, y)$



(a) (b)

Figure 6: YaleB05 (a) x, y, z components/ channels of the surface normal depicted in the second image (b) Calculated surface normal from the $g(x, y)$



(a) (b)

Figure 7: YaleB07 (a) x, y, z components/ channels of the surface normal depicted in the second image (b) Calculated surface normal from the $g(x, y)$

2.4 Compute the Surface Height Map by Integration

We can now recover the surface height at any point by integration along some path:

$$f(x, y) = \int_0^x f_x(s, y)ds + \int_0^y f_y(x, t)dt + C \quad (7)$$

For robustness, we will take integrals over many different paths and average the results:

- Integrating first the ROWS, then the columns. That is, the path first goes along the same row as the pixel along the top, and then goes vertically down to the pixel. It is possible to implement this without nested loops using the `cumsum()` function.
- Integrating first along the COLUMNS, then the rows.
- AVERAGE of the first two options.
- Average of multiple RANDOM paths. We will determine the number of paths experimentally

Instead of continuous integration of the partial derivatives over a path, we will be be summing their discrete values:

2.4.1 Cumulative sum over column

```
case 'column'
        % cumulative sum of fx over columns (the
           second argument depicts dimension #2, which
            is columns)
        temp(1,2:w) = cumsum(fx(1,2:w),2);
        % we just copy the whole fy from 2nd row
           afterwards
        temp(2:h,:) = fy(2:h,:);
        % the depth map will be the cumulative sum
           over rows of the both previous results
           shown in matrix form
        heightMap = cumsum(temp);
```

* complete code in Appendix B.3

Explantion:

First we produced cumulative sum over the columns of first row of f_x.

We copy all other rows of f_y as it is to the output matrix.

Then we produce cumulative sum over all rows of the output matrix.

2.4.2 Cumulative sum over row

```
case 'row'
        %heightMap = cumsum(surfaceNormals(:,:,1), 1)
           + cumsum(surfaceNormals(:,:,2), 2);
        % cumulative sum of fy over rows (the second
           argument depicts dimension #11, which is
           rows)
        temp(2:h,1) = cumsum(fy(2:h,1));
        % we just copy the whole fx from 2nd column
           afterwards
        temp(:,2:w) = fx(:,2:w);
        % the depth map will be the cumulative sum
           over columns of the both previous results
           in matrix form
        heightMap = cumsum(temp,2);
```

* complete code in Appendix B.3

Explantion:

First we produced cumulative sum over the rows of first column of f_y.

We copy all other columns of f_x as it is to the output matrix.

Then we produce cumulative sum over all columns of the output matrix.

2.4.3 Average of cumulative sum over column and cumulative sum over row

```
case 'average'
        temp(2:h,1) = cumsum(fy(2:h,1));
        temp(:,2:w) = fx(:,2:w);

        temp1(1,2:w) = cumsum(fx(1,2:w));
        temp1(2:h,:) = fy(2:h,:);
```

```
7
8           heightMap = (cumsum(temp1)+cumsum(temp,2))./2;
```

* complete code in Appendix B.3

Explantion:

We averaged out the cumulative sum over all columns and cumulative sum over all rows with the ./ elementwise operator for division by 2.

2.4.4 Random Paths

```
1  case 'random'
2           heightMap(2:h,1) = ysum(2:h,1);
3           heightMap(1,2:w) = xsum(1,2:w);
4
5           for i = 2:h
6               for j = 1:w
7                   t = 0;
8                   s = 0;
9
10                  for k = 1:i-1
11                      if j-k >= 1
12                          t = t+ysum(1+k)+xsum(1+k,j-k)+
                               ysum(i,j-k)-ysum(1+k,j-k)+
                               xsum(i,j)-xsum(i,j-k);
13                          s= s+1;
14                      end
15                  end
16
17                  if i==2 || i== h || k == i
18                      t = t+ysum(i,1)+xsum(i,j);
19                      s = s+1;
20                  end
21
22                  heightMap(i,j) = t/s;
23              end
24          end
```

* complete code in Appendix B.3

Explantion:

We will understand this method better by seeing a graphics on it, taken from [8], shown in Figure 8.

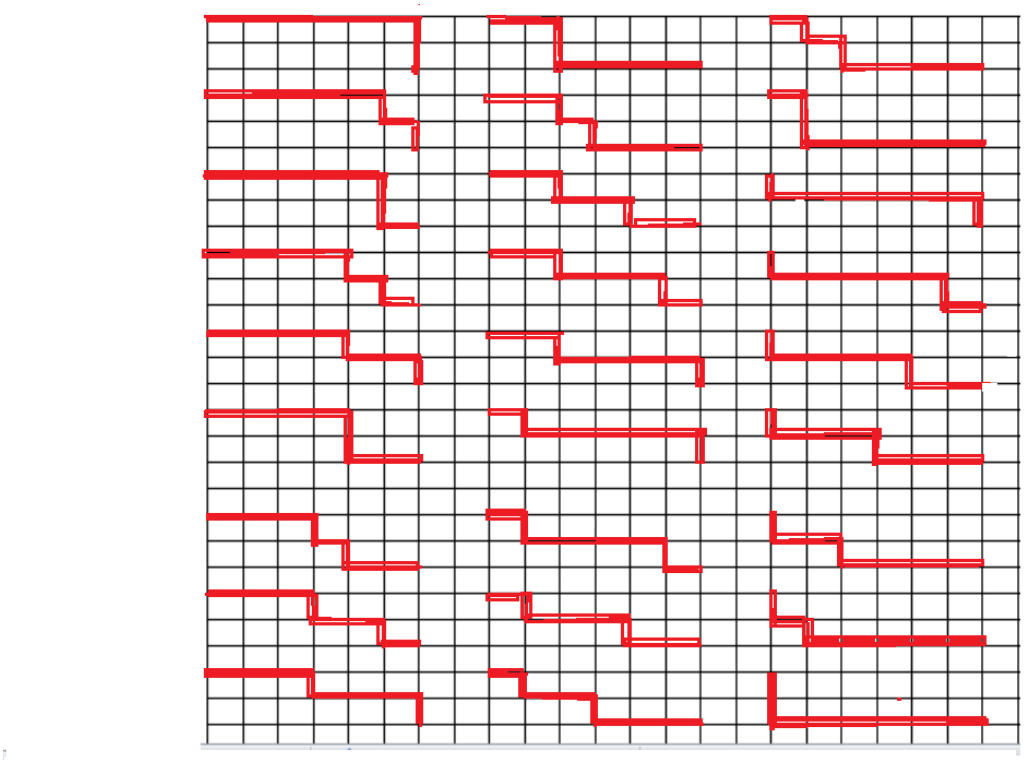


Figure 8: Matrix grid navigation for one point to another. The figure depicts all the random paths possible for a hypothetical 2 x 6 grid. The same concept is used to generate random paths for our integration to the surface.[8]

2.5 About the Integration Methods

Let's first observe the depth map for all four different integration methods first, see Figure 14

- COLUMN WISE INTEGRATION The depth map looks like it's stretched vertically; it's right column-wise but doesn't give good result for the rows. For each row, the column integration generates lot of vertical waviness. We see this effect the most where columns

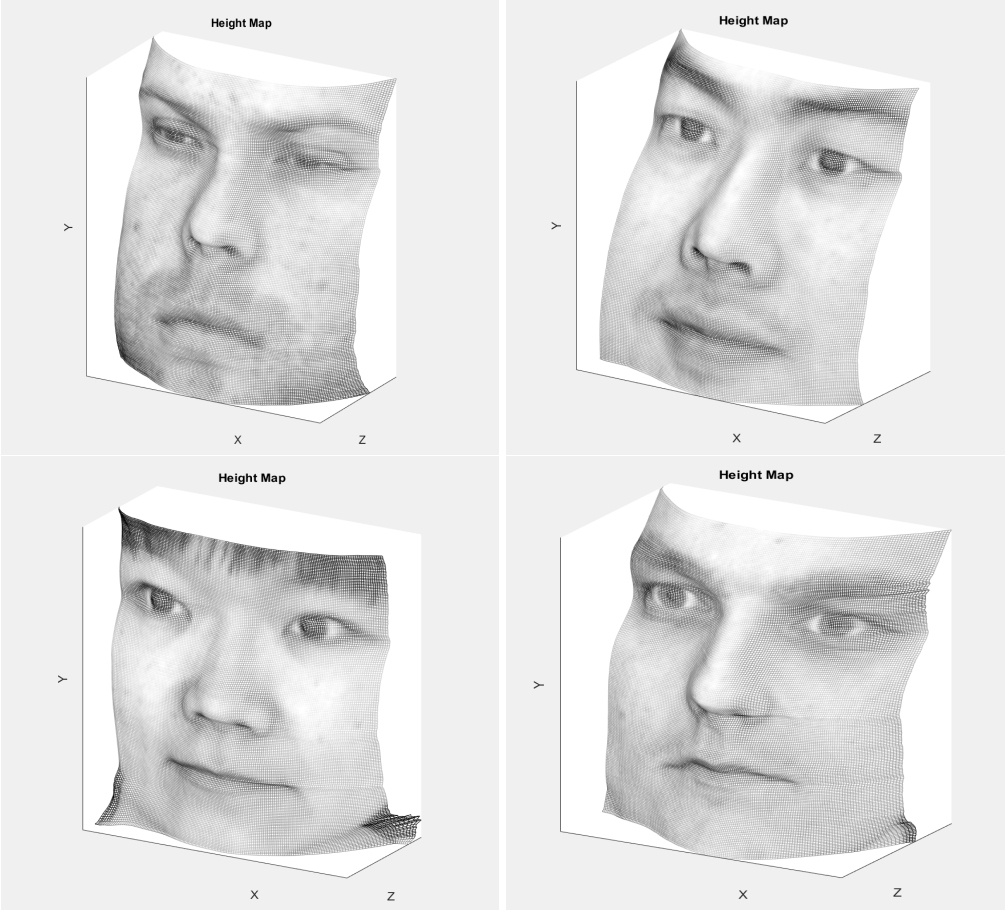


Figure 9: All 4 subject depth approximation shown from angle 1; integration mode = random

have very different, varied values for each row, e.g. mouth and lips are higher up than cheeks.

- ROW WISE INTEGRATION The depth map looks like it's stretched horizontally; it's right row-wise but doesn't give good result for the columns. For each column, the row integration generates lot of horizontal waviness. We see this effect the most where rows have very different, varied values for each column, e.g. eyes are deeper as compared to nose.

- AVERAGE INTEGRATION This just adds both of the above shortcomings but in a milder way because of the few anomalies getting averaged out.

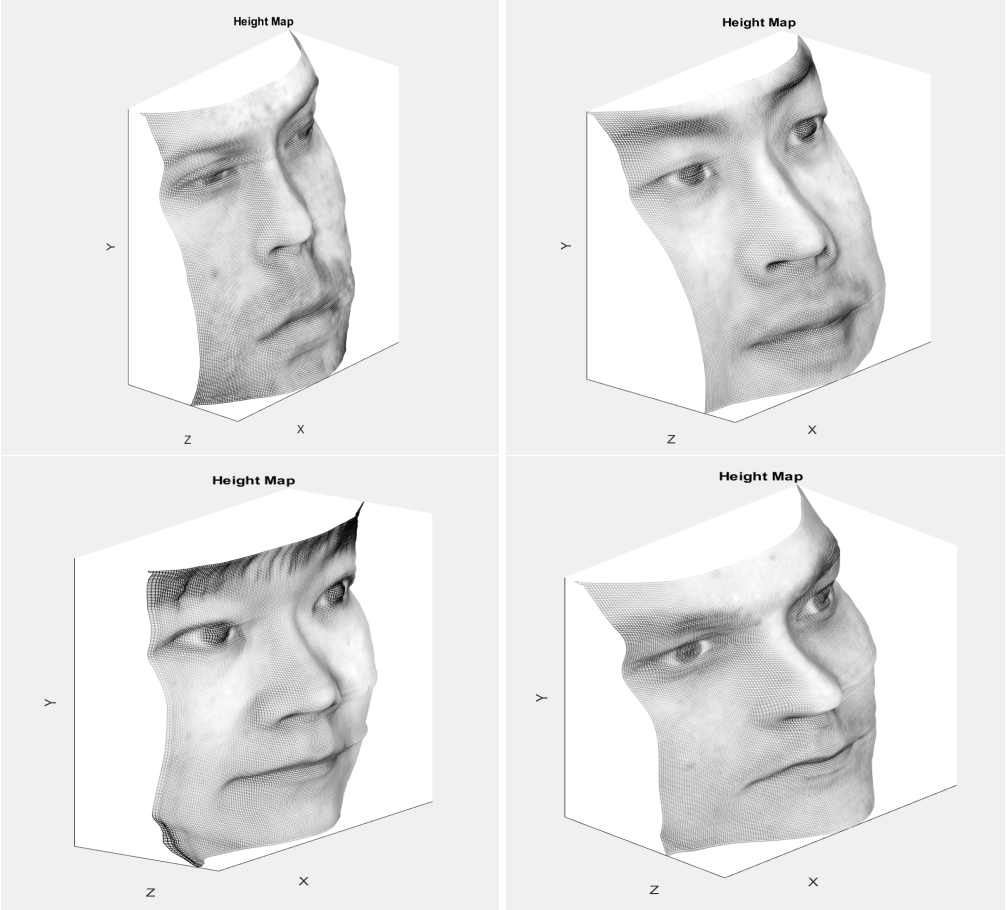


Figure 10: All 4 subject depth approximation shown from angle 2; integration mode = random

- RANDOM PATH INTEGRATION This doesn't produce the errors we can see for the previous three results as random path technically chooses each row and column combination equally, which results in getting less wavelike and smoother results. The only column or only row integration path will be evened out by the paths which mix taking cumulative sums over rows and columns in every combination.

2.6 About the Yale Face Data

Despite getting good and acceptable results with random path integration method, there are some shortcomings and assumptions in Yale Face Database which must be pointed out which might be violating shape

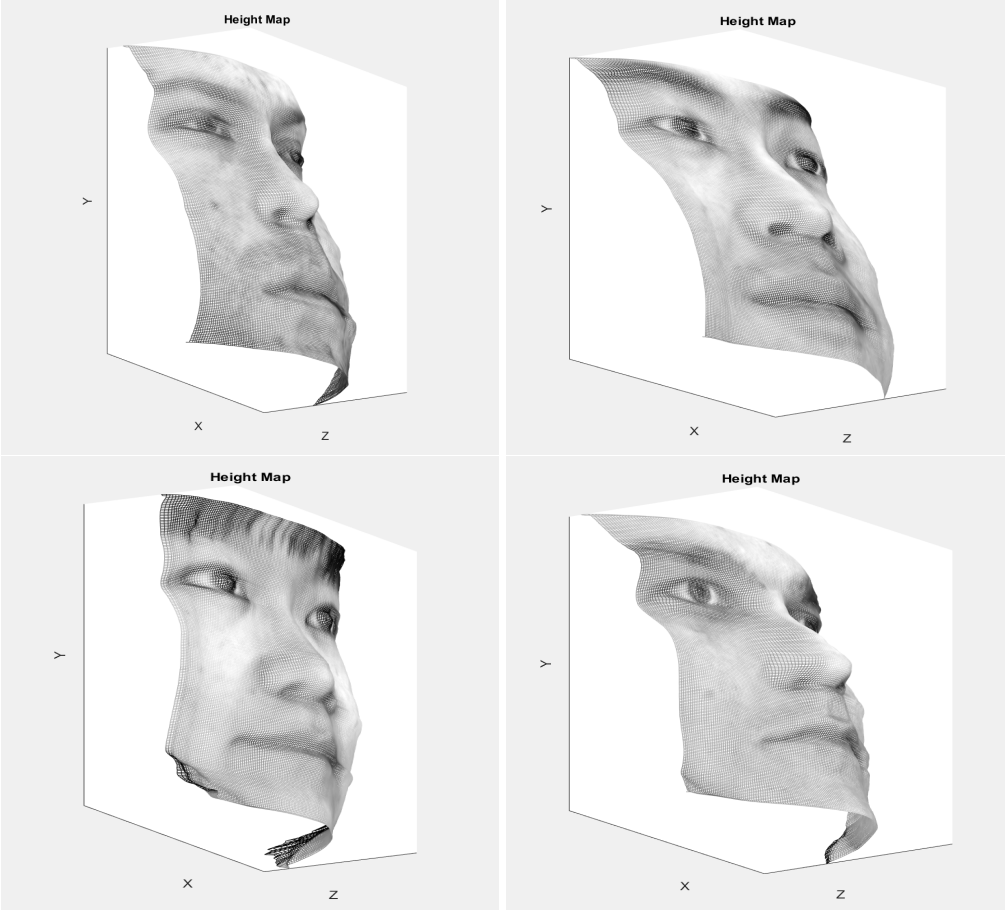


Figure 11: All 4 subject depth approximation shown from angle 3; integration mode = random

from shading concepts:

- ORTHOGRAPHIC CAMERA MODEL We cannot really get accurate depth maps from orthographic model, it has a fixed depth, there's no distance from the camera.
- SIMPLISTIC REFLECTANCE AND LIGHTING MODEL We didn't take into account the intrinsics of reflectance physics and how different lighting model will take different types of reflectances into account e.f. diffuse, specular reflectances.
- NO SHADOWS If some part of image is always in shadow, we wouldn't be able to approximate its depth, as this whole idea depends on radiance and irradience

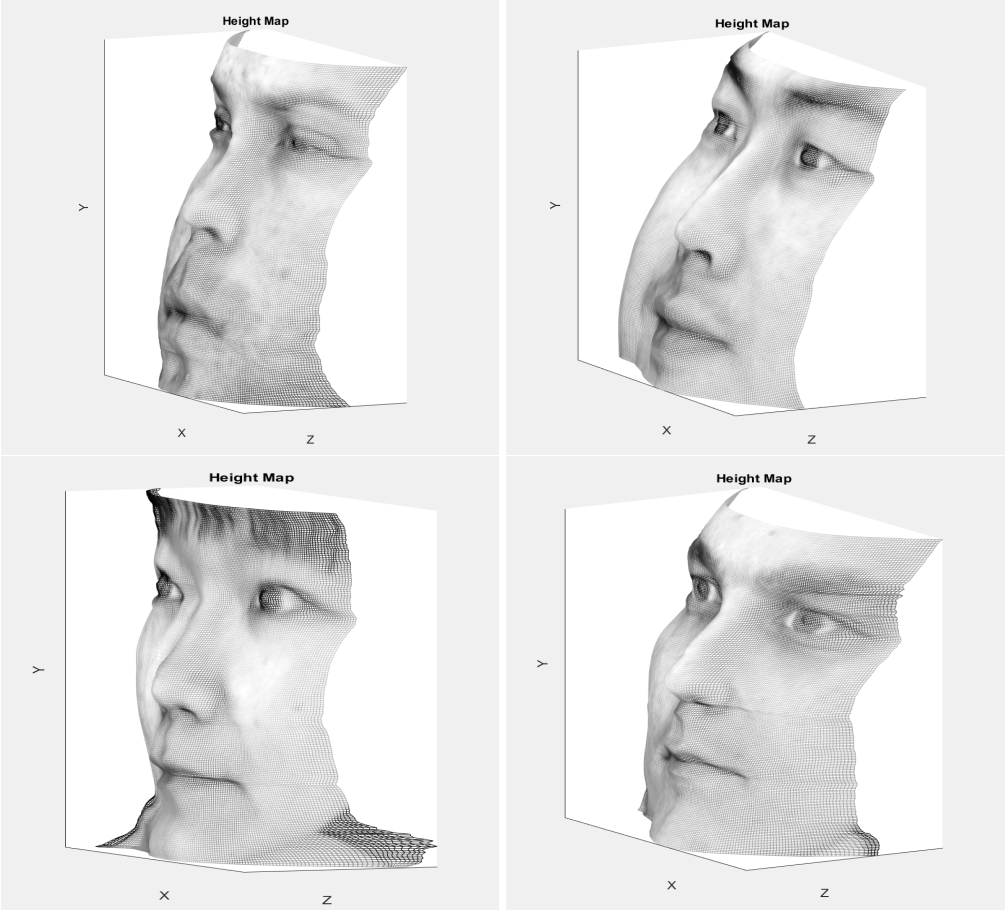


Figure 12: All 4 subject depth approximation shown from angle 4; integration mode = random

- NO INTER-REFLECTANCE Same as the second point
- NO MISSING DATA Same as the third point; also, we need plenty of data
- INTEGRATION IS TRICKY We saw in previous subsection, getting integration right and the cost of computation can get pretty high pretty quick, as the image resolution increases. We need to take multiple path to even out directional errors, which is costly in computation.

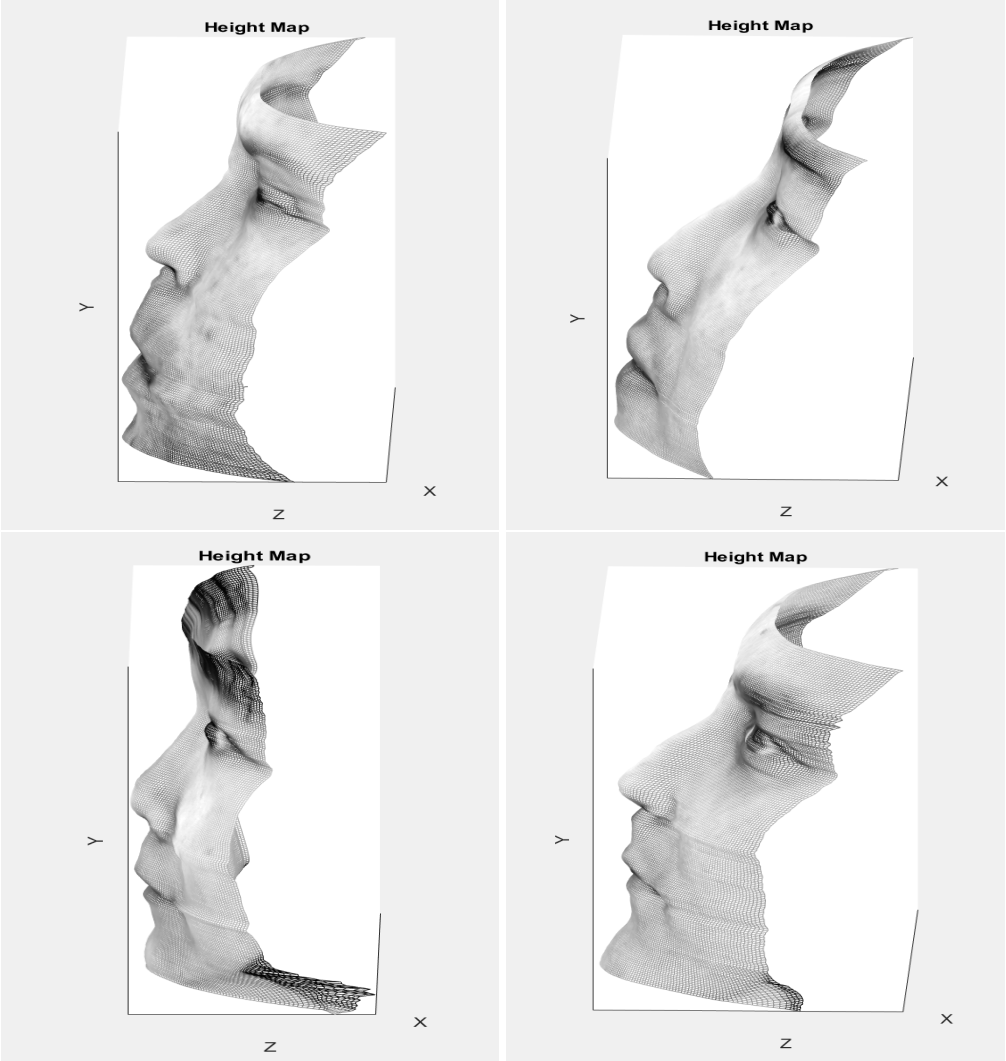
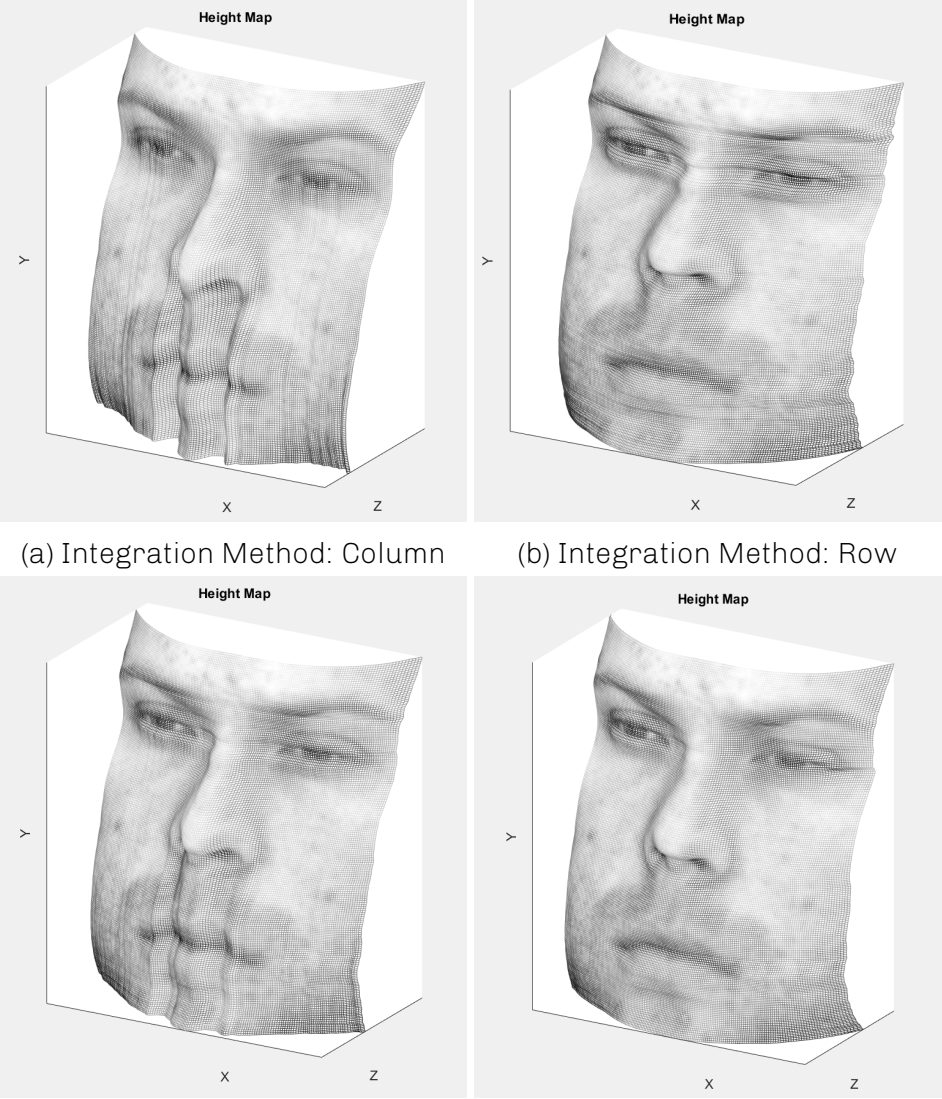


Figure 13: All 4 subject depth approximation shown from angle 5; integration mode = random

2.7 Conclusions and Bottom Line

Clearly, random path integration gives much better results. We can conclude that from Figure 14. Using both direction and permutating and combining them in different combinations will make sure we are free from directional bias or waviness.



(a) Integration Method: Column (b) Integration Method: Row

(c) Integration Method: Average (d) Int Method: Random Path

Figure 14: YaleB01 depth maps with all four integration methods

3 Extension for Extra Credits

One way to get extra credit was to **generate synthetic input data using a 3D model** and a graphics renderer and run the above stated methods on the data. I tried to accomplish this in the following manner:

3.1 Generating the Data

To generate data about 2D images from a fixed camera position, we used a pre-built model from Unity3D Assets Store [9] library. Unity3D allows us to set the object/ model, camera and light positions.

Object/ model and world Characteristics

- WORLD POSITION The model is at (0, 0, 0)
- MODEL UNDER AMBIENCE See Figure 15

Figure 15: A 3D tree model of tree viewed from a fixed camera postion, under no effects of environmental or any other kinds of lights

- MODEL UNDER LIGHT SOURCE AT DIFFERENT POSITIONS we have followed the Yale Dataset Standards for Azimuth and Elevation angles, we took 64 images of the model from both A and E angles

which are same as angles chosen in the dataset given to us. They provided enough variety. See Figure 16

(a) Azimuth=+15 and Elevation = +20 | (b) Azimuth=+110 and Elevation = +40 | (c) Azimuth=-35 and Elevation = +40 | (d) Azimuth=-110 and Elevation = -20

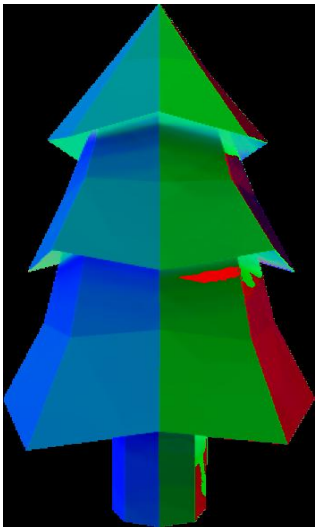
Figure 16: Few (out of 64) examples of tree dataset: 2D rendering of the tree model with light source at azimuth and elevation angles specified

- LIGHT SOURCE We used a directional light source so the different angles have some significance.
- LIGHT SOURCE POSITION is at (5, 0, 0) and rotating on X or Y axis for Elevation or Azimuth angle respectively.
- SKYBOX We had to change the default skybox to pure black so the world intricacies or differences won't matter for this test.
- CAMERA is fixed at (5, 0, 0)

3.2 Estimate the Albedo and Normals

Applying the same algorithm as for the Yale Faces, we get the following results for albedo and normals:
See Figures 17 & 18

(a) Recovered Albedo .



(b) Recovered Normal Map(all 3 channels, colored)

Figure 17: Albedo and Normal Map for the Tree Model

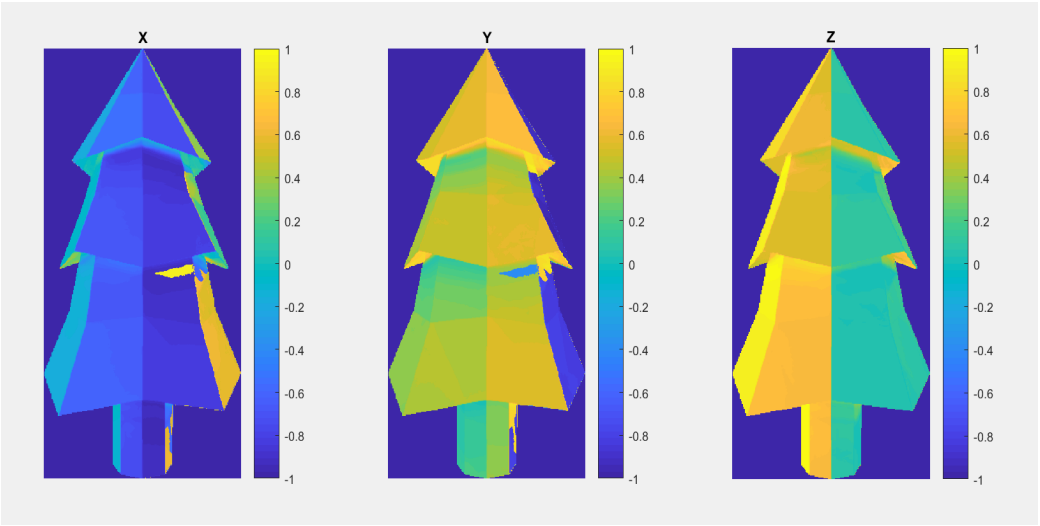


Figure 18: Recovered Surface Normals

3.3 Compute the Surface Height Map by Integration and Improvements

Unfortunately, this step is not getting computed because of an unresolved error.

The potential reasons might that the somewhere, an image matrix is being "rank deficient", but given that the previous Yale Faces are getting their height maps just fine, I cannot figure out why this error is just in this dataset.

There might be that the light information isn't enough to provide some knowledge about the height or depth, but again, I have used the same angles used in capturing the face dataset.

I conclude that the problem might be in how I screen-captured the rendered projections. Maybe light directions aren't making sense there.

I would like to conclude this section with an appeal to consider grade the extra credit section according to whatever parts I was able to accomplish, and hopefully during the course of this semester, I would figure out the source of this error.

References

[1] Ying Wu, *Radiometry, BRDF and Photometric Stereo - Radiometry, BRDF and Photometric Stereo* http://users.eecs.northwestern.edu/~yingwu/teaching/EECS432/Notes/lighting.pdf

[2] Leslie D. Stroebel, Richard D. Zakia, *Focal Encyclopedia of Photography* ISBN 0-240-51417-3

[3] David A. Forsyth, Jean Ponce, *Computer Vision: A Modern Approach (2nd Edition)* ISBN 978-0-13608-592-8

[4] https://en.wikipedia.org/wiki/Phong_reflection_model

[5] Pharr & Humphreys *"Physically Based Rendering": Fundametals of Rendering - Radiometry / Photometry*

[6] Netpbm Documentation http://netpbm.sourceforge.net/doc/pgm.html

[7] Subhransu Maji *Computer Vision Lecture 02: Radiometry* https://www.dropbox.com/s/0rtuxli868hcwq9/lec02_radiometry.pdf?dl=0

[8] Calculating the number of possible paths through some squares https://math.stackexchange.com/questions/636128/calculating-the-number-of-possible-paths-through-some-squares

[9] Unity3D Assets Store https://assetstore.unity.com

A

Matlab code for Problem 1: Aligning Prokudin-Gorskii images

A.1 Implementing `alignChannels.m`

```
function [imShift, predShift] = alignChannels(im, maxShift)
% ALIGNCHANNELS align channels in an image.
%   [IMSHIFT, PREDSHIFT] = ALIGNCHANNELS(IM, MAXSHIFT) aligns the channels in an
%   NxMx3 image IM. The first channel is fixed and the remaining channels
%   are aligned to it within the maximum displacement range of MAXSHIFT (in
%   both directions). The code returns the aligned image IMSHIFT after
%   performing this alignment. The optimal shifts are returned as in
%   PREDSHIFT a 2x2 array. PREDSHIFT(1,:) is the shifts  in I (the first)
%   and J (the second) dimension of the second channel, and PREDSHIFT(2,:)
%   are the same for the third channel.
%
% This code is part of:
%
%   CMPSCI 670: Computer Vision, Fall 2016
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%
%   Homework 1: Color images
%   Author: Subhransu Maji


% Sanity check
assert(size(im,3) == 3);
assert(all(maxShift > 0));

```

```
26 % Dummy implementation (replace this with your own)
27 predShift = zeros(2, 2);
28
29 padding = floor([.2*size(im, 2), .2*size(im, 1), .6*
       size(im, 2), .6*size(im, 1)]);
30 im_crop = imcrop(im, padding);
31
32 min = 999999;
33 redChannel = im_crop(:,:,1);
34 greenChannel = im_crop(:,:,2);  %fixed
35 blueChannel = im_crop(:,:,3);
36
37 %%Match Red and Green Channels
38 for i = -maxShift(1):maxShift(1)
39     for j = -maxShift(1):maxShift(1)
40         temp = circshift(redChannel, [i, j]);
41         error = sum(sum((greenChannel-temp).^2));
42         if min > error
43             min = error;
44             predShift(1,1) = -i;
45             predShift(1,2) = -j;
46             correctedGreenChannel = circshift(im
                   (:,:,1), [i, j]);
47         end
48     end
49 end
50
51 min = 999999;
52 %%Match Green and Blue Channels
53 for i = -maxShift(2):maxShift(2)
54     for j = -maxShift(2):maxShift(2)
55         temp = circshift(blueChannel, [i, j]);
56         error = sum(sum((greenChannel-temp).^2));
57         if min > error
58             min = error;
59             predShift(2,1) = -i;
60             predShift(2,2) = -j;
61             correctedBlueChannel = circshift(im(:,:,3)
                   , [i, j]);
62         end
63     end
```

```
64 end
65
66 imShift = cat(3, correctedGreenChannel, im(:,:,2),
       correctedBlueChannel);
67
68 h = size(imShift,1);
69 w = size(imShift,2);
70 c = size(imShift,3);
71 c_img = imShift(1:floor(h/10), 1:floor(w/10), :);
72
73 vert = 0;
74 horz = 0;
75
76 for i = 1:c
77     % Find edges
78     temp = edge(c_img(:,:,i), 'canny', 0.1);
79
80     % Find mean value and mask the values.
81     vert_avg = mean(temp, 1);
82     horz_avg = mean(temp, 2);
83     threshold_1 = 3*mean(horz_avg);
84     threshold_2 = 3*mean(vert_avg);
85     vert_mask = vert_avg > threshold_2;
86     horz_mask = horz_avg > threshold_1;
87
88     % Find last values
89     last_vert = find(vert_mask, 1, 'last');
90     last_horz = find(horz_mask, 1, 'last');
91
92     if last_vert > vert
93         vert = last_vert;
94     end
95     if last_horz > horz
96         horz = last_horz;
97     end
98 end
99
100 imShift = imcrop(imShift, [horz vert (w-2*horz) (h-2*
       vert)]);
```

B
Matlab code for Problem 2: Photometric Stereo

B.1 Implementing `prepareData.m`

```
function output = prepareData(imArray, ambientImage)
% PREPAREDATA prepares the images for photometric stereo
%   OUTPUT = PREPAREDATA(IMARRAY, AMBIENTIMAGE)
%
%   Input:
%       IMARRAY - [h w n] image array
%       AMBIENTIMAGE - [h w] image
%
%   Output:
%       OUTPUT - [h w n] image, suitably processed
%
% Author: Subhransu Maji
%

% Implement this %
% Step 1. Subtract the ambientImage from each image in imArray
% Step 2. Make sure no pixel is less than zero
% Step 3. Rescale the values in imarray to be between 0 and 1

%% Subtracting ambience
output = imArray - ambientImage;

%% Negative values in the output matrix will be converted to zero
output(output<0) = 0;

%% To work with double datatypes [0, 1]; we must make sure that all the values are normalized with that range. Because after changing some negative values to 0, the pixel values we get will not be normalized as the range is getting changed with the lower limit getting converted to 0
```

```
27 output = rescale(output);
```

B.2 Implementing photometricStereo.m

```
1  function [albedoImage, surfaceNormals] =
       photometricStereo(imArray, lightDirs)
2  % PHOTOMETRICSTEREO compute intrinsic image
       decomposition from images
3  %   [ALBEDOIMAGE, SURFACENORMALS] = PHOTOMETRICSTEREO(
       IMARRAY, LIGHTDIRS)
4  %   comptutes the ALBEDOIMAGE and SURFACENORMALS from
       an array of images
5  %   with their lighting directions. The surface is
       assumed to be perfectly
6  %   lambertian so that the measured intensity is
       proportional to the albedo
7  %   times the dot product between the surface normal
       and lighting
8  %   direction. The lights are assumed to be of unit
       intensity.
9  %
10 %   Input:
11 %       IMARRAY - [h w n] array of images, i.e., n
       images of size [h w]
12 %       LIGHTDIRS - [n 3] array of unit normals for
       the light directions
13 %
14 %   Output:
15 %        ALBEDOIMAGE - [h w] image specifying albedos
16 %        SURFACENORMALS - [h w 3] array of unit
       normals for each pixel
17 %
18 % Author: Subhransu Maji
19 %
20 % Acknowledgement: Based on a similar homework by Lana
        Lazebnik
21
22 [h, w, n] = size(imArray);
23 % g(x, y) = rho(x, y)*N(x, y) or p(x, y)*N(x, y). The
       unknown
24 g = zeros(h,w,3);
```

```
25 p = zeros(h,w); % rho. Albedo.
26 surfaceNormals = zeros(h,w,3); % Surface Normal
27
28 % Loop through all pixels of the image
29 for i = 1:h
30     for j = 1:w
31         % Got a particular pixel at (i, j) from all
              the images
32         I = imArray(i, j, 1:n);
33         % Converted into a n x 1 column vector
34         I = I(:);
35         % Calculated g of I(column matrix of pixel
              values) = V(light directions) * g
36         g(i,j,:) = I\lightDirs;
37         % Calculated the magnitude of g to get the
              albedo pixel values p(x, y) = |g(x, y)|
38         p(i,j) = sqrt(g(i,j,1)^2 + g(i,j,2)^2 + g(i,j
              ,3)^2);
39
40         % To calculate surface normals
41         fx = g(i, j, 1)/g(i, j, 3); % Partial
              derivative of the surface with respect to x
42         fy = g(i, j, 2)/g(i, j, 3); % Partial
              derivative of the surface with respect to y
43         temp = sqrt(fx^2 + fy^2 + 1); % Magnitude
44         % N(x, y) = (fx, fy, 1)/sqrt(fx^2 + fy^2 + 1)
45         surfaceNormals(i, j, :) = [fx/temp, fy/temp,
              1/temp];
46     end
47 end
48 % Now we got albedo and surface normal for each pixel
     in image
49
50 %% Albedo Image
51 % Some pixel values will be out of [0, 1] because of
     matrix divisions. Rescale them to [0, 1]. The
     result will be inverted, complement the values.
52 albedoImage = imcomplement(rescale(p));
```

B.3 Implementing `getSurface.m`

```
function  heightMap = getSurface(surfaceNormals, method)
% GETSURFACE computes the surface depth from normals
%   HEIGHTMAP = GETSURFACE(SURFACENORMALS, IMAGESIZE, METHOD) computes
%   HEIGHTMAP from the SURFACENORMALS using various METHODs.
%
% Input:
%   SURFACENORMALS: height x width x 3 array of unit surface normals
%   METHOD: the intergration method to be used
%
% Output:
%   HEIGHTMAP: height map of object

[h, w, channel] = size(surfaceNormals);
temp = zeros(h, w);
temp1 = zeros(h, w);
fx = surfaceNormals(:,:,1)./surfaceNormals(:,:,3);
fy = surfaceNormals(:,:,2)./surfaceNormals(:,:,3);
xsum = cumsum(fx,2);
ysum = cumsum(fy);

switch method
    case 'column'
        % cumulative sum of fx over columns (the second argument depicts dimension #2, which is columns)
        temp(1,2:w) = cumsum(fx(1,2:w),2);
        % we just copy the whole fy from 2nd row afterwards
        temp(2:h,:) = fy(2:h,:);
        % the depth map will be the cumulative sum over rows of the both previous results shown in matrix form
        heightMap = cumsum(temp);

    case 'row'
        %heightMap = cumsum(surfaceNormals(:,:,1), 1) + cumsum(surfaceNormals(:,:,2), 2);
```

```
32          % cumulative sum of fy over rows (the second
               argument depicts dimension #11, which is
               rows)
33          temp(2:h,1) = cumsum(fy(2:h,1));
34          % we just copy the whole fx from 2nd column
               afterwards
35          temp(:,2:w) = fx(:,2:w);
36          % the depth map will be the cumulative sum
               over columns of the both previous results
               in matrix form
37          heightMap = cumsum(temp,2);
38
39      case 'average'
40          temp(2:h,1) = cumsum(fy(2:h,1));
41          temp(:,2:w) = fx(:,2:w);
42
43          temp1(1,2:w) = cumsum(fx(1,2:w));
44          temp1(2:h,:) = fy(2:h,:);
45
46          heightMap = (cumsum(temp1)+cumsum(temp,2))./2;
47
48      case 'random'
49          heightMap(2:h,1) = ysum(2:h,1);
50          heightMap(1,2:w) = xsum(1,2:w);
51
52          for i = 2:h
53              for j = 1:w
54                  t = 0;
55                  s = 0;
56
57                  for k = 1:i-1
58                      if j-k >= 1
59                          t = t+ysum(1+k)+xsum(1+k,j-k)+
                               ysum(i,j-k)-ysum(1+k,j-k)+
                               xsum(i,j)-xsum(i,j-k);
60                          s= s+1;
61                      end
62                  end
63
64                  if i==2 || i== h || k == i
65                      t = t+ysum(i,1)+xsum(i,j);
```

```
                    s = s+1;
                end

                heightMap(i,j) = t/s;
            end
        end
end
```

C Matlab code for Extra Credit Problem:

C.1 Cropping the Region of Interest `getSurface.m`

```

images = dir('A*.png');
for i = 1:64

    filePath = 'C:\Users\Panchal''s\Pictures\
       Screenshots\Tree';

    image = imread(fullfile(filePath, images(i).name))
       ;
    %figure;
    imwrite(imcrop(image, [355, 218, 348, 685]),
       sprintf('Tree%s',images(i).name));

end

image = imread(fullfile(filePath, '_Ambient.png'));
imwrite(imcrop(image, [355, 218, 348, 685]), sprintf('
   Tree%s','_Ambient.png'));
```