Mini-Project 1

1. Aligning Prokudin-Gorskii Images

Aim:

To reconstruct a color photograph by aligning the blue, green and red channels of the image.

Implementation of alignChannels.m:

Steps –

1. Selected the “Blue channel” as the reference channel.
2. Circularly shifted the “Red channel” and “Green channel” exhaustively in the range of -15 to 15 pixels, in both horizontal and vertical directions.
3. Calculated the sum of squared distances between the circularly shifted channels and the reference channels, where, sum of squared distances is:
$$SSD = sum(sum(image1 - image2)^2))$$
where, the sum is taken over the pixel values.
4. The values of the shift, for which the SSD was minimum for a particular image, was stored in the variable *predShift.*
5. The image channels were shifted by the values stored in *predShift*, in order to align them.

Code –

```
function [imShift, predShift] = alignChannels(im, maxShift)
assert(size(im,3) == 3);
assert(all(maxShift > 0));


shifti=zeros(1,2);
shiftj=zeros(1,2);


imShift(:,:,1)=im(:,:,1);

bestg=intmax('uint64');
bestr=intmax('uint64');
for n=-maxShift(1):maxShift(2)
    for m=-maxShift(1):maxShift(2)
        disp1=sum(sum((circshift(im(:,:,2),[n,m])-im(:,:,1)).^2));
        disp=sum(sum((circshift(im(:,:,3),[n,m])-im(:,:,1)).^2));
```

```
          if(disp<bestg)
            bestg=disp;
            shifti(1)=n;
            shiftj(1)=m;
            imShift(:,:,2)=circshift(im(:,:,2),[n,m]);

        end

          if(disp<bestr)
            bestr=disp;
            shifti(2)=n;
            shiftj(2)=m;
            imShift(:,:,3)=circshift(im(:,:,3),[n,m]);

        end



    end
end



predShift = [shifti',shiftj'];
```

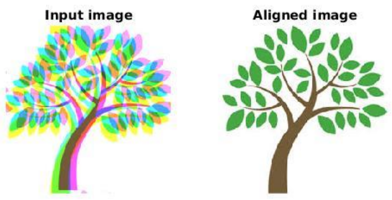
Verifying the code – evalAlignment.m

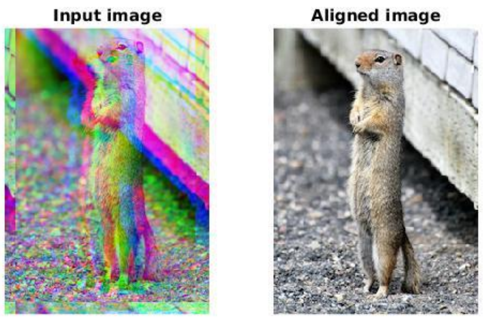
For the sake of verification, random shifts were introduced in two channels of sample images (*gtShift* in *randomlyShiftChannels.m*). The code *alignChannels.m* should be able to identify the random shifts introduced and revert them to produce align images. Thus, the values stored in the array *predShift* should be the negative of the values stored in the array *gtShift.*

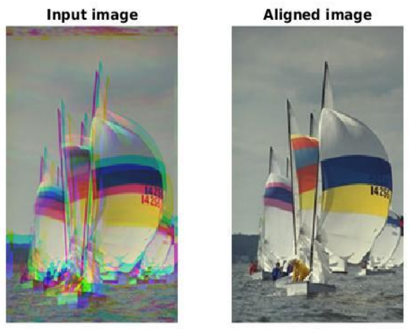
OUTPUT:

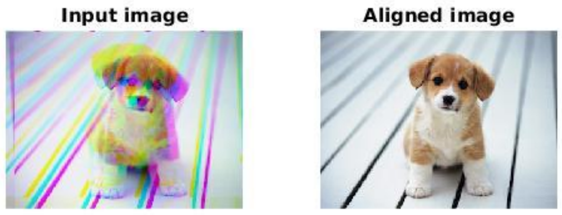
1.balloon.jpeg	gt shift: (-15, 4) (-11,-2)	pred shift: (15,-4) (11, 2)
2. cat.jpg	gt shift: (14, 6) (-15, 5)	pred shift: (-14,-6) (15,-5)
3 ip.jpg	gt shift: (-2,-12) (8,-3)	pred shift: (2,12) (-8, 3)
4 puppy.jpg	gt shift: (2,-13) (-12,-11)	pred shift: (-2,13) (12,11)
5 squirrel.jpg	gt shift: (4,-10) (-1,-8)	pred shift: (-4,10) (1, 8)

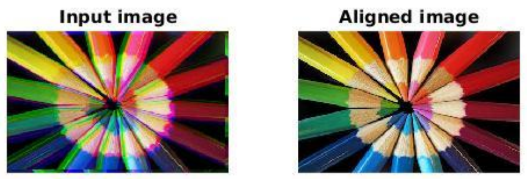
6 pencils.jpg	gt shift: (-5, 6) (-11,-13)	pred shift: (5,-6) (11,13)
7 house.png	gt shift: (7,-12) (-6,11)	pred shift: (-7,12) (6,-11)
8 light.png	gt shift: (5, 1) (-5,-15)	pred shift: (-5,-1) (5,15)
9 sails.png	gt shift: (7,-3) (11, 6)	pred shift: (-7, 3) (-11,-6)

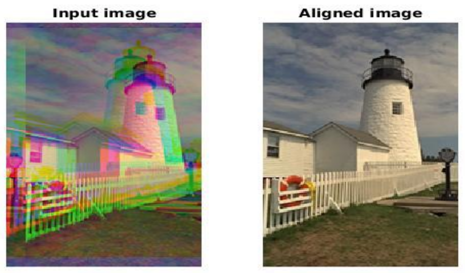


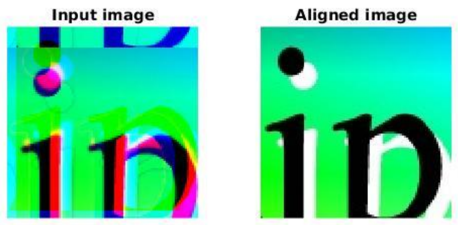


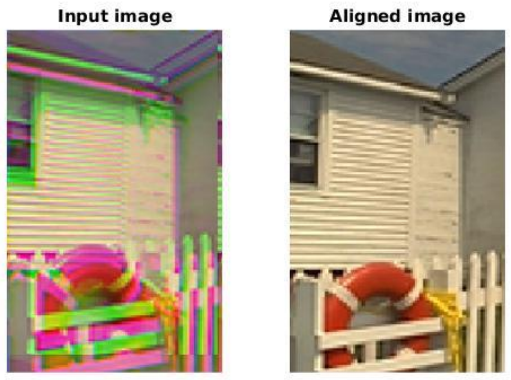


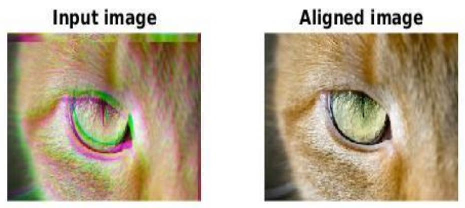


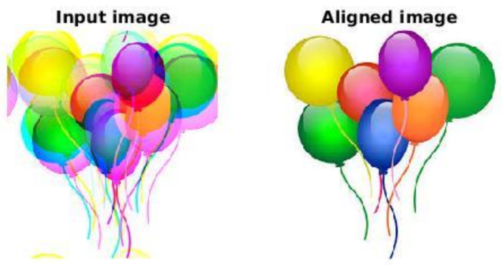

Input image
Aligned image


Input image
Aligned image


Input image
Aligned image


Input image
Aligned image


Input image
Aligned image


Input image
Aligned image

Aligned Color Images from evalProkudinAlignment.m

The following are the shift vectors and the aligned images from the Prokudin-Gorskii Image Collection.

Table 1: Shift vectors for the red and green channels obtained using alignChannels.m for the Prokudin-Gorskii Images

Image	**Shift Vectors**
00125-aligned.jpeg	(-4,1), (-10,2)
00149-aligned.jpeg	(-13,-2), (-11,-3)
00153-aligned.jpeg	(0,1), (-8,2)
00351-aligned.jpeg	(-5,0), (-9,-1)
00398-aligned.jpeg	(-9,0), (-13,1)
01112-aligned.jpeg	(-8,-1), (-8,-3)

Photometric Stereo

Aim: In this experiment, our aim is to generate the 3-D shape from the 2-D images taken under different lighting conditions be generating the height map from the recovered albedo and the surface normals

<u>Implementation of prepareData.m:</u>

Steps –

1. Found the number of images, n from the image array
2. Subtracted all the pixels of the ambient image from all the pixels of each of the image array
   where, the sum is taken over the pixel values.
3. After performing the above operation, if any of the pixel in the image array is negative then I make it 0.
4. Then I normalize the image array by dividing all the pixels of each of the
5. The image channels were shifted by the values stored in *predShift*, in order to align them.

CODE:

```
function output = prepareData(imArray, ambientImage)

n=size(imArray,3);

for i=1:n
     imArray(:,:,i)=imArray(:,:,i)-ambientImage(:,:);
    %step 1: subtract ambient image


    imArray(imArray(:,:)<0)=0;
   %step 2: make all negative pixels 0

imArray(:,:,i)= imArray(:,:,i)./(max(max(imArray(:,:,i))));
   %step 3: Normalize
 end

end
output=imArray;
```

Implementation of Photometric stereo.m:

1. In this function, we had to compute the albedo and the surface normal for each pixel given the image array and the lighting direction for each of the 2D image
2. The approach I used was as follows
3. First I found out the height ,width of each image and the number of images from the image array
4. Then I reshaped the image array into a 2-Dimensional array i.e. changes it from a [h,w,n] matrix to a [p,n] matrix and renamed it to pixel
5. The reason for doing this was to easily solve the system of linear equations
6. Then I took the transpose of the matrix pixel, effectively making it a [n,p] matrix
   Reason:
   We had to solve:
   Ax=b
   - ➔ $I(x,y) = \rho * N(x,y).S(x,,y)$ [This is the equation giving radiosity at each point-]
   - ➔ $I(x,y) = g(x,y) * S(x,y)$
     So in our case:
     g(x,y)=N(x,y)*ρ
   - ➔ Pixel=lightdir*g
   - ➔ A-> lightdirs array, b->pixel array, x->g
     Where,

$$\left[\qquad\right] = \left[\qquad\right] * \left[\quad\right]$$

Pixel-n*p **lightdirs(n*3)** **g(3*p)**

7. Thus we obtained g as a through solving the system of linear equations using least squares and it will be of shape 3*p
   $$g = lightdirs\backslash pixel$$
   Which is effectively saying, $x = A\backslash b$

8. Then I took transpose of g, and reshaped it into a [h,w,3] matrix
9. Then for each pixel (x,y), I found the albedo $\rho(x,y)$= $||g||_2$
10. Thus obtained the surfaceNormals $N(x,y)$ by dividing g by the albedo

CODE:

```
function [albedoImage, surfaceNormals] = photometricStereo(imArray,
lightDirs)
h=size(imArray,1);
w=size(imArray,2);
n=size(imArray,3);
%pixel=[h*w;n];
pixel=reshape(imArray,h*w,n);

pixel=pixel';%pixel=n*p

g=lightDirs\pixel;%g=3*p

g=reshape(g,h,w,3);

for i=1:h
    for j=1:w
        sv=g(i,j,:).*g(i,j,:);

       dp=sum(sv(1,1,:));
       albedoImage(i,j)=sqrt(dp);
      surfaceNormals(i,j,:)=g(i,j,:)./albedoImagetemp(i,j);

    end
end
```

`Implementation of surfaceNormals.m`:

1. In this we had to find the heightmap given the surfacenormals and the method of integration
2. So first I found the Fx and Fy which is the change in $z = f(x, y)$ w.r.t x and y respectively
3. Fx=g_1/g_3 and similarly Fy=g_2/g_3
4. Then I found out the cumsum of Fx matrix row-wise and the cumsum of the Fy matrix columnwise and stored it in an array

ROW method:

```
CODE:
case 'row'
        for i=1:h
             for j=1:w

               depth(i,j)=(cs1(1,j))+cs2(i,j);
            end

        end
```

- ➔ In this method, for every pixel, I found the heighmap byjust indexing the cumsum array which was calculated earlier
- ➔ Thus cs1(1,j) will give the sum of all Fx values till row 1 and column j.

Column Method:

CODE:
```
case 'column'
           for i=1:h
               for j=1:w
                depth(i,j)=cs2(i,1)+cs1(i,j);
               end
           end
```

1. For the column method, followed the same strategy as the row method, just the difference being the for column, first I found the cumsum till ith row in first column cs2(i,1) in the Fy matrix and then did the same with the Fx matrix

```
Average Method:


          for i=1:h
             for j=1:w
               %depth1(i,j)= sum(p(1,1:j))+sum(q(1:i, j));

               depth1(i,j)=cs1(1,j)+cs2(i,j);
            end

        end

        %method='column';
          for i=1:h
               for j=1:w
               %depth2(i,j)=sum(q(1:i, 1)) + sum(p(i,1:j));
               depth2(i,j)=cs2(i,1)+cs1(i,j);
               end
           end

        depth=(depth1+depth2)/2;
```

In the average method, I just averaged the heighmap values found by the row and the column method

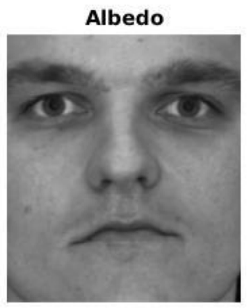
Random Method:
- ➔ In this method, I found out experimentally that number of paths=7, then the height map comes out to be the best
- ➔ Hence for each pixel, I find 7 random paths
- ➔ Suppose we have to find the depth of pixel (x,y)

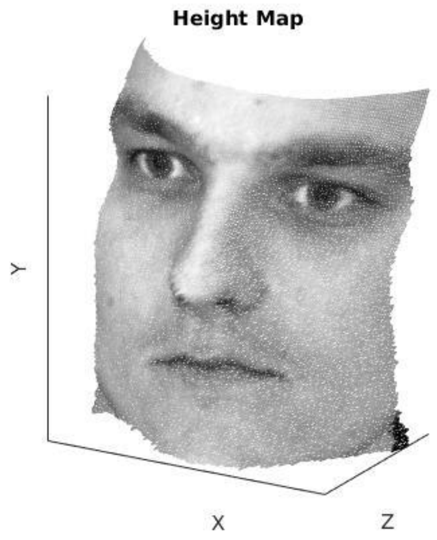
- ➔ We find a pixel(i,j)<pixel(x,y) a
- ➔ We choose a method randomly(either row method or the column method)
- ➔ Then find the heightmap(i,j) using that method chosen above
- ➔ Repeat the entire above procedure for all the till we reach x and y

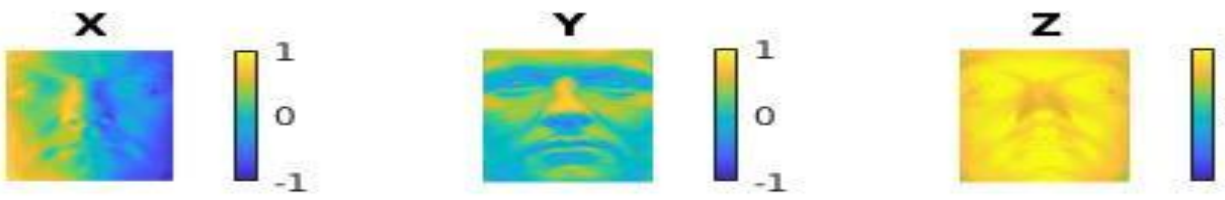
CODE:- (Random method)

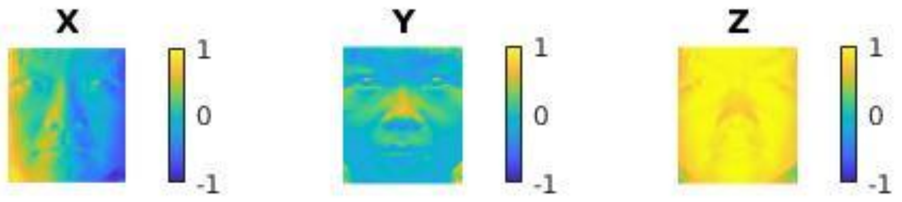
```
   num_paths=7;

        for i=1:h %for each pixel
            for j=1:w
                  total_height=0;
                for k=1:num_paths %choose no of paths from 1

                    path_height=0;
                    prev_i=1;prev_j=1;% start with 1,1 for every path
                    rand_i=1;rand_j=1;

                      while(prev_i<i || prev_j<j)
  % loop till you reach i,j for each path

                          if(prev_i~=i)
                          rand_i=randi([prev_i+1,i]);
 % for every iteration,choose a pixel between prev_i and i

                          end


                      if(prev_j~=j)
                   rand_j=randi([prev_j+1,j]);
                   end

                 rand_num=randi(3);
             switch rand_num
                 case 1
                       %rand_method='row';
    temp=sum(p(prev_i,prev_j:rand_j))+sum(q(prev_i:rand_i,rand_j));


                 case 2
    temp=sum(q(prev_i:rand_i,prev_j))+sum(p(rand_i,prev_j:rand_j));

                 case 3
                       %rand_method='average';
      temp1=sum(p(prev_i,prev_j:rand_j))+sum(q(prev_i:rand_i,rand_j));
   temp2=sum(q(prev_i:rand_i,prev_j))+sum(p(rand_i,prev_j:rand_j));


                       temp=(temp1+temp2)./2;
                  end

                       prev_i=rand_i;%set the prev to curr i and j
                       prev_j=rand_j;

                       path_height=path_height+temp;

              end

                  total_height=total_height+path_height;



             end


                       depth(i,j)=total_height/num_paths;


              end
          end

     end
```

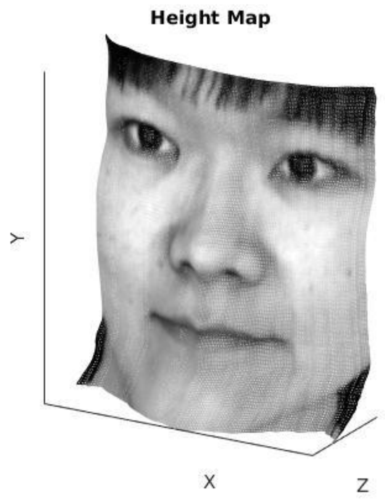
```
Output:
Debug: Method:row
yaleB01: Method:column
yaleB02: Method:average
yaleB05: Method:average
yaleB07: Method:random
```

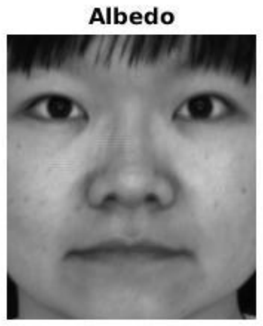

Albedo

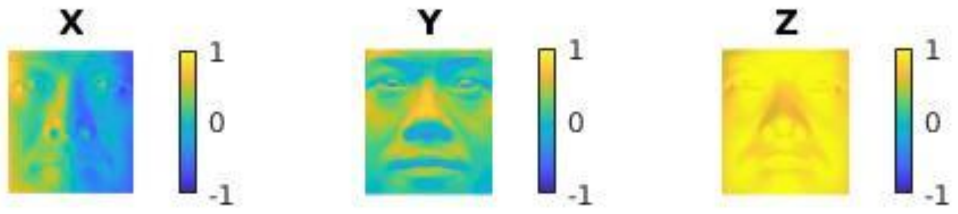

Height Map
Y
X
Z

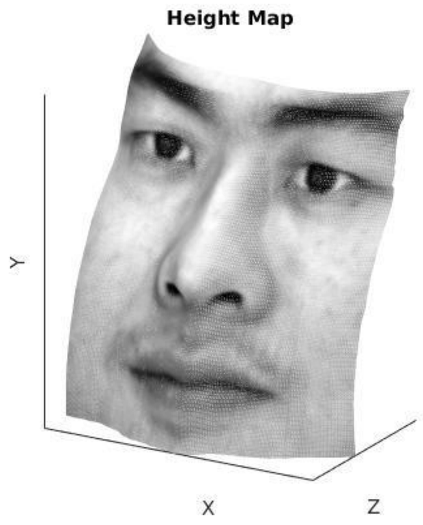

X
1
0
-1
Y
1
0
-1
Z

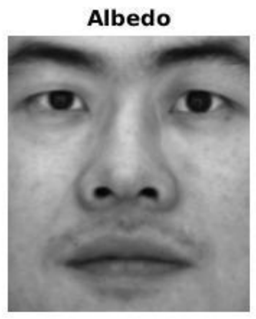

X
1
0
-1
Y
1
0
-1
Z
1
0
-1

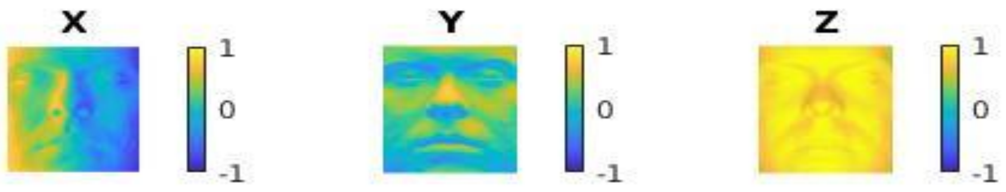

Height Map
Y
X
Z

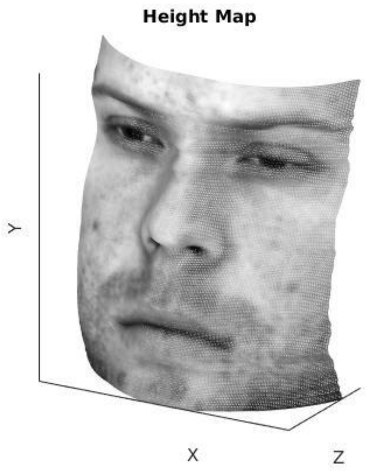

Albedo


X
1
0
-1
Y
1
0
-1
Z
1
0
-1

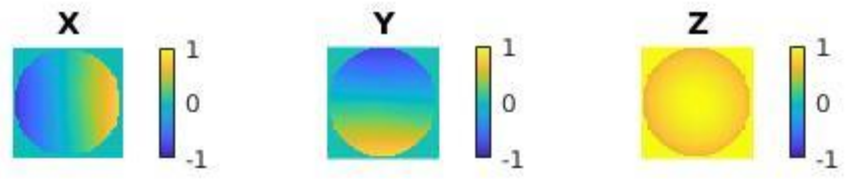

Height Map
Y
X
Z

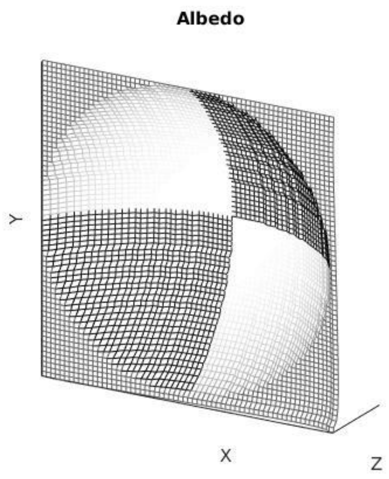

Albedo


X
1
0
-1
Y
1
0
-1
Z
1
0
-1


Height Map
Y
X
Z

Albedo

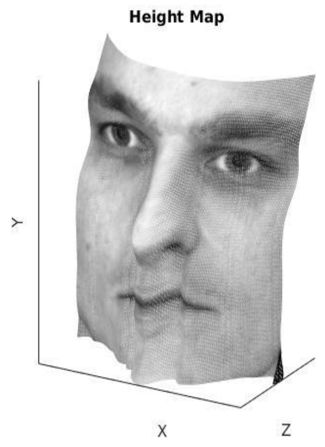

X
1
0
-1
Y
1
0
-1
Z
1
0
-1


Albedo
Y
X
Z

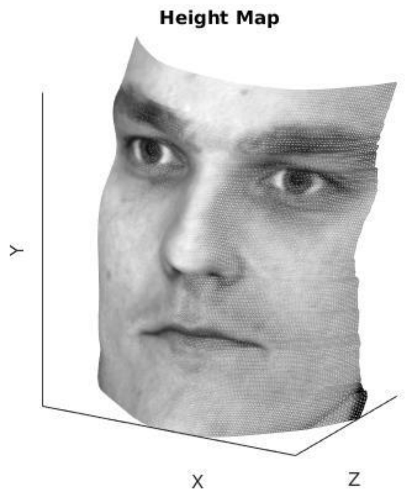
Albedo

Analysis:

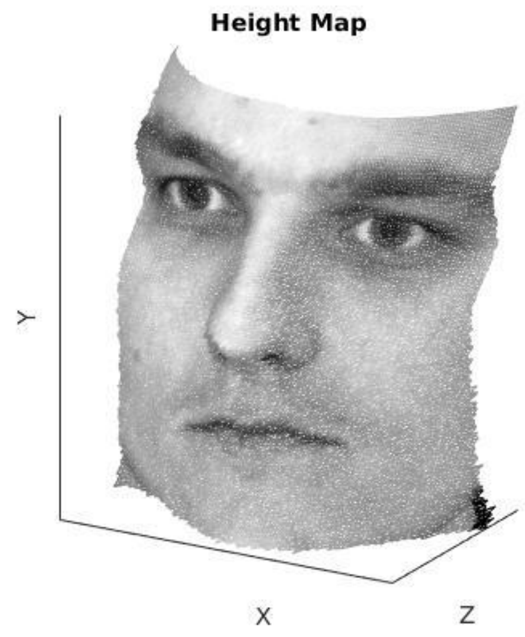
1. The different integration methods implemented are the row, column, average and the random method.
2. Lets us take the subject YaleB07 to compare these methods.
3. The row and column integration methods are essentially the same, just operating on different order of axis. There might be error along one of the dimensions so one might be a little better than the other
4. Averaging the two reduces the noise from the inferior axis of each of them ,thus mostly yield a superior result
5. Random method works well but if there is noise present in the lower order pixels, it will accumulate over multiple runs and might degrade the result as every integration path starts from top left pixel
6. This can be improved by starting the path from different pixel
7. In my case, in the yaleB07 image, the random path gives much better results than the other methods. There was a ripple in average integration method, which the random integration method could overcome



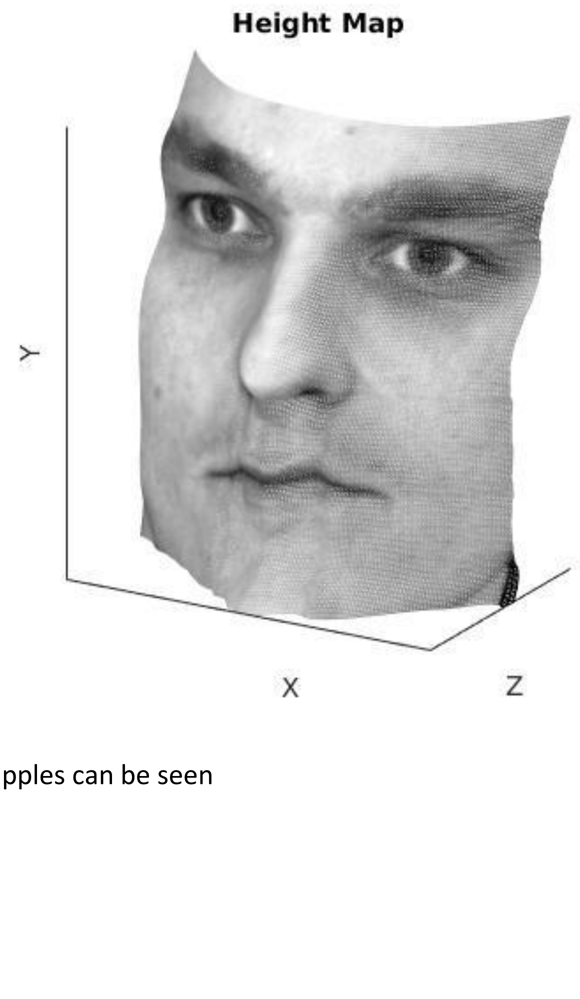
Row – lots of ripples along the chin



Column-ripples along cheeks

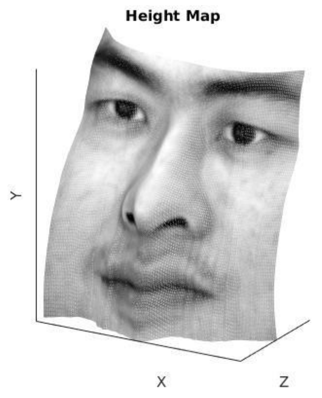


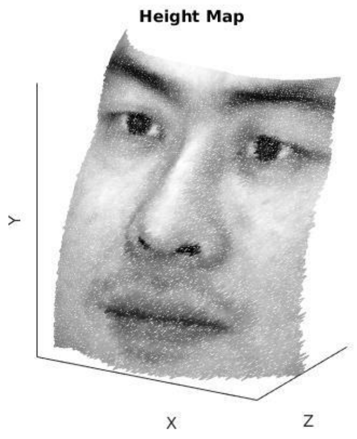
Random-best method of all 4



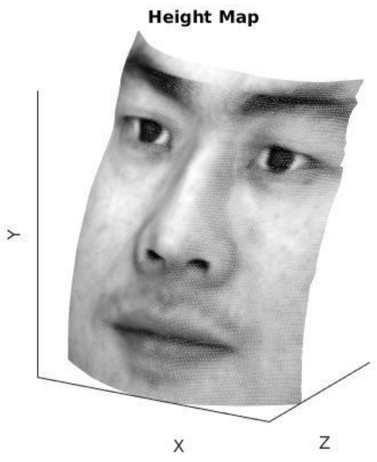
Average-some ripples can be seen

But as mentioned earlier, in some cases, due to noise in lower order pixels, the random methods does perform well. This happened in the case of yaleB02 as we can see the average method yields a better result than the other 3

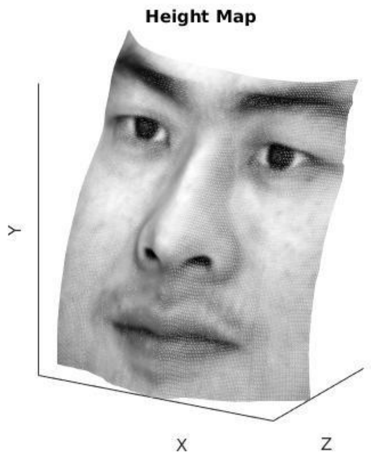




row random



column



average

Similary in yaleB01, the column method yielded the best results because the noise along he row dimensions degraded the performance of the average method.

Assumptions violated by the Yale Face data:

We made the following assumptions for our data:

1. A lambertian object
2. A local shading model (no global illumination)
3. A set of known light source directions
4. A set of pictures of an object, obtained in exactly the same camera/object configuration but using different light sources
5. Orthographic projection

1. Assumption of Lambertian surface
    - The shape from shading method assumes that the surface is a Lambertian object and thus has diffuse reflections
    - But this might not be true as we see that when the light falls on certain regions of the face such as tip of our nose or even somewhere on the forehead, it gives specular reflections
    - Also, the Human skin is far from lambertian. Light can be sub-scattered beneath the skin. Additionally, oil on the skin often makes for an extremely specular surface. This is especially true on the nose, cheeks and chin. Eyes are also extremely specular.
    - Thus, the face data has diffuse reflections with certain specular highlights violating the assumption of the face being a Lambertian object giving only diffuse reflections
2. Assumption of a local shading model:
    - The shape from shading method used assumes that the shading model is local wherein each point on the surface receives light only from sources visible at that point
    - The local shading model captures :
      Direct illumination from light sources
      Diffuse and *Specular* components
      (Very) Approximate effects of global lighting
    - But it does not capture: Shadows, Mirrors, Refraction, etc, which might be present in our Yale face data, hence violating the assumption of a local shading model