Homework 3 Writeup

Instructions

- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- Use as many pages as you need, but err on the short side If you feel you only need to write a short amount to meet the brief, th
- **Please make this document anonymous.**

Algorithm Implementation

The implementation of the stereo imaging is divided into four parts.

0.1 Bayer Image Interpolation

To convert the Bayer image into an RGB version, bicubic interpolation was used. Shortened code is shown below, please refer to the *bayer_to_rgb_bicubic.m* file for the full version. In lines up to 8, the Bayer image is separated into four channels of R,G1,G2 and B. The G channel is divided into G1 and G2 because bicubic interpolation uses 16 nearest known colors, but it was difficult to choose 16 nearest points from the layout of the G channel. So G1 and G2 were considered as two separate channels. They were later combined to form a single channel G in lines 10 to 17. In the function *bicubic*, bicubic interpolation of the input channel is implemented as described in InterpolationBicubic.pdf. The channel is first padded appropriately by repeating the border pixels. Then I found matrices *X*, *Y*, *Z*, *Px* and *Py* one by one. *X* and *Y* are offset arrays in x and y directions, respectively. *Z* is the array of values of nearest 16 pixels. *Px* and *Py* are the arrays of normalized distance to the nearest pixels, raised to up to the third power. For each *(i,j)* of channel, the distance to the nearest upper left pixel was used to find *Z*, *Px* and *Py*. The result of interpolation can be expressed as $[Py][Y^{-1}][Z][X^{-1}][Px]$. An interesting point is that if the commented matrix in line 33 for the offset matrix *X* is used instead of the original one, the resulting image looks more natural. The comparison is discussed in the results section (Figure 2).

```
1 function rgb_img = bayer_to_rgb_bicubic(bayer_img)
2     ...
3     window=["r", "g1"; "g2", "b"];
4     repWindow=repmat(window, img_size/2);
5     r_channel(repWindow=="r")=bayer_img(repWindow=="r");
```

```
    g1_channel(repWindow=="g1")=bayer_img(repWindow=="g1");
    g2_channel(repWindow=="g2")=bayer_img(repWindow=="g2");
    b_channel(repWindow=="b")=bayer_img(repWindow=="b");
    r=bicubic(r_channel);
    g1=rot90(bicubic(rot90(g1_channel)),3);
    g2=rot90(bicubic(rot90(g2_channel,3)));
    g=zeros(size(g1));
    avgg=(g1+g2)/2;
    g((repWindow=='b'))=avgg(repWindow=='b');
    g((repWindow=='r'))=avgg(repWindow=='r');
    g((repWindow=='g1'))=g1_channel(repWindow=='g1');
    g((repWindow=='g2'))=g2_channel(repWindow=='g2');
    b=rot90(bicubic(rot90(b_channel,2)),2);
    rgb_img=uint8(cat(3,r,g,b));
    ...
end

function result = bicubic(channel)
    padded=cat(1, channel(1:2,:), channel);
    padded=cat(1,padded, padded(size(padded,1)-1:size(padded,1),:));
    padded=cat(1,padded, padded(size(padded,1)-1,:));
    padded=cat(2,padded(:,1:2), padded);
    padded=cat(2, padded, padded(:,size(padded,2)-1:size(padded,2)));
    padded=cat(2, padded, padded(:,size(padded,2)-1));
    paddedCpy=padded;

    X=[-1 0 1 8; 1 0 1 4; -1 0 1 2; 1 1 1 1];
%    X=[-8 0 8 64; 4 0 4 16; -2 0 2 4; 1 1 1 1];
    Y=X';
    for i=3:size(padded,1)-3
        for j=3:size(padded,2)-3
            if padded(i,j)==0
                d=normalizedDistance(i,j);
                centerP=[i j]-d;
                Z=reshape(nonzeros(padded(centerP(1)-2:centerP(1)+4,centerP(2)-2:centerP(2)+4)),4,4);
                px=[d(1)^3; d(1)^2; d(1); 1];
                py=[d(2)^3 d(2)^2 d(2) 1];
                p=py*inv(Y)*Z*inv(X)*px;
                paddedCpy(i,j)=p;
            end
```

```
46          end
47      end
48      result=paddedCpy(3:size(padded,1)-3,3:size(padded,2)
            -3);
49  end
50
51  function n=normalizedDistance(i,j)
52      n=[0 0];
53      if mod(i,2)==0
54          n(1)=1;
55      else
56          n(1)=0;
57      end
58      if mod(j,2)==0
59          n(2)=1;
60      else
61          n(2)=0;
62      end
63  end
```

0.2 Fundamental Matrix Calculation

The fundamental matrix is calculated using the eight-point algorithm. Full code shown below. First, the matching points are normalized in lines 3 to 4. Then, the A matrix of the eight point algorithm formula $Af = 0$ is constructed in lines 10 to 12. The fundamental matrix F is initially made by reshaping the eigenvector corresponding to the smallest eigenvalue of $A^T A$. This is done in line 13. This is part of the least squares approach. Then, F is updated by changing the smallest eigenvalue to 0. This is done in line 14. Because this is a normalized eight point algorithm, the fundamental matrix should be turned back into original units. This is done in line 16.

```
1   function f = calculate_fundamental_matrix(pts1, pts2)
2       clc;
3       [pts1, T1]=normalize_points(pts1', 2);
4       [pts2, T2]=normalize_points(pts2', 2);
5       pts1=pts1';
6       pts2=pts2';
7       x=padarray(pts1',1,1,'post');
8       xp=padarray(pts2,[0 1],1,'post');
9       A=[];
10      for i=1:size(x,2)
11          A=cat(1,A,reshape((x(:,i).*xp(i,:))',[1 9]));
12      end
13      F=reshape(smallest_eigenvector(A'*A),[3 3])
14      F=update_sv(F)
```

```
15
16      f = T2'*F*T1
17  end
18
19  function v = smallest_eigenvector(A)
20      [V,D] = eig(A);
21      [d,ind] = sort(diag(D));
22      Ds = D(ind,ind);
23      Vs = V(:,ind);
24      v=Vs(:,1);
25  end
26
27  function Fp = update_sv(F)
28      [U,S,V]=svd(F);
29      [~,idx]=sort(diag(S));
30      S=S(idx, idx);
31      V=V(:,idx);
32      U=U(:,idx);
33      S(1,1)=0;
34      Fp=U*S*V';
35  end
```

0.3 Image Rectification

In this section, the left and right images were rectified using their respective homography matrices. Full code shown below. First, the two homograpy matrices were converted into transformations in lines 3 and 4. Then the four corners of each image was transformed into the space of the final image in lines 9 and 10. But because these points contain negative coordinates, they should be shifted so that they exist in the positive coordinate space. Such shift amount was found in line 13 by finding the minimum values in the corner coordinates. The corners were then shifted in line 14. Then, the images were warped using the transformations in lines 16 and 17. Now the images are transformed into the final image space, but they are not aligned. The amount of offset for alignment of each image can be found from the transformed corners as in lines 21 and 22. They are the minimum values in the respective corner coordinates. The image are translated by the found amount in lines 24 and 25. Then, the size of the aligned image is determined in line 27, which is the maximum of the dimensions of the separate images. Lines 29 and 30 are executed so that the aligned images have the same size (the final image size) by adding appropriate amount of padding.

```
1  function [rectified1, rectified2] = rectify_stereo_images
       (img1, img2, h1, h2)
2
3      tform1 = projective2d(h1);
4      tform2 = projective2d(h2);
```

```
5
6       corners1 = [1 1; size(img1,2) 1; 1 size(img1,1); size
            (img1,2) size(img1,1)]
7       corners2 = [1 1; size(img2,2) 1; 1 size(img2,1); size
            (img2,2) size(img2,1)]
8
9       tcorners1 = transformPointsForward(tform1,corners1);
10      tcorners2 = transformPointsForward(tform2,corners2);
11      tcorners = [tcorners1; tcorners2]
12
13      shift = [-min(tcorners(:,1))+1, -min(tcorners(:,2))
            +1]
14      tcorners = tcorners+shift
15
16      img1 = imwarp(img1, tform1);
17      img2 = imwarp(img2, tform2);
18
19      tcorners1 = tcorners1+shift;
20      tcorners2 = tcorners2+shift;
21      offset1 = [min(tcorners1(:,1))-1 min(tcorners1(:,2))
            -1]
22      offset2 = [min(tcorners2(:,1))-1 min(tcorners2(:,2))
            -1]
23
24      img1 = imtranslate(img1,offset1,'OutputView','full');
25      img2 = imtranslate(img2,offset2,'OutputView','full');
26
27      max_size = round([max(size(img1,1), size(img2,1)) max
            (size(img1,2), size(img2,2))])
28
29      rectified1 = padarray(img1,[max_size(1)-size(img1,1)
            max_size(2)-size(img1,2)], 0,'post');
30      rectified2 = padarray(img2,[max_size(1)-size(img2,1)
            max_size(2)-size(img2,2)], 0,'post');
31  end
```

0.4 Disparity Map Calculation

The disparity map was created from the aligned images. Different parameters such as the window size, maximum disparity, and filter amount were tested to get the best results. The full code of disparity map calculation is show below. First, an empty cost volume is created in line 3 with size according to the image size and the maximum disparity. Then, each pixel of the left image *(i,j)* is iterated, storing the normalized cross correlation value with the pixel of the right image separated by k pixels in *(i,j,k)*. Because we can assume that a corresponding window of the right window exists on the right of it, in other words,

the left image exists on the right of the right image, we can improve efficiency by only looking at the windows on the right of the left image located at *(j-k)*, as shown in line 11. Then, for each plane in the cost volume, cost was aggregated using a Gaussian filter and was experimented with varying standard deviation values. The normalized cross correlation was implemented in a separate function. The input matrices A and B were subtracted the mean of each. Then, the cross correlation was calculated according to the formula $\frac{\sum\sum A(i,j)B(i,j)}{\sqrt{\sum\sum A(i,j)^2}\sqrt{\sum\sum B(i,j)^2}}$. Because a NaN error occurs if either A or B contains only 0 elements, such cases were avoided by immediately returning 0, thus not altering the cost volume. The comparison of various parameters are discussed in the results section.

```
1  function d = calculate_disparity_map(img_left, img_right,
       window_size, max_disparity)
2
3      cost_vol = zeros(size(img_left,1), size(img_left,2),
           max_disparity);
4      size(cost_vol)
5
6      for i = ceil(window_size/2):size(img_left,1)-floor(
           window_size/2)
7          for j = ceil(window_size/2):size(img_left,2)-
               floor(window_size/2)
8              l = img_left(i-floor(window_size/2):i+floor(
                   window_size/2),j-floor(window_size/2):j+
                   floor(window_size/2));
9              for k = 1:max_disparity
10                 if (j-k)-floor(window_size/2)>0
11                     r = img_right(i-floor(window_size/2):
                           i+floor(window_size/2),(j-k)-floor
                           (window_size/2):(j-k)+floor(
                           window_size/2));
12                     cost_vol(i,j,k) = ncc(l,r);
13                 end
14             end
15         end
16     end
17
18     cost_vol = smooth_planes(cost_vol);
19
20
21     % winner takes all
22     [min_val, d] = max(cost_vol,[],3);
23
24 end
25
26 function c = ncc(A, B)
```

```
    A = A-mean(A(:));
    B = B-mean(B(:));
    if nnz(A) == 0 || nnz(B) == 0
        c = 0;
    else
        rA = sqrt(sum(sum(A.^2,1)));
        rB = sqrt(sum(sum(B.^2,1)));
        ab = sum(sum(A.*B,1));
        c = ab/(rA*rB);
    end
end

function vol = smooth_planes(vol)
    for d = 1:size(vol,3)
        vol(:,:,d) = imgaussfilt(vol(:,:,d),1);
    end
end
```

Results

0.5 Bayer Image Interpolation

The figure below shows the matching points of left and right interpolated images.

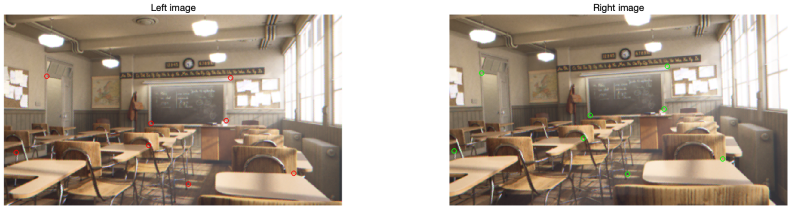


Figure 1: Matching points.

Figure 2 compares the different values for X. The first image is made by using the correct offset matrix

$$\begin{bmatrix} -1 & 0 & 1 & 8 \\ 1 & 0 & 1 & 4 \\ -1 & 0 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

while the second one used

$$\begin{bmatrix} -8 & 0 & 8 & 64 \\ 4 & 0 & 4 & 16 \\ -2 & 0 & 2 & 4 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

. Using the second option means we use the second-closest 16 points not the closest 16 points. That's why the offsets are mutiples of 2 not 1. Although the second option is not the true bicubic interpolation, it provided somewhat clearer and more natural looking image. As shown in Figure 2, the first one has visible red, green and blue blocks at edges in the image, while the second one does not. The first option was used through later parts but using the second option created no visible differences.

Figure 2: *Left:* First X matrix. *Right:* Second X matrix.

0.6 Fundamental Matrix Calculation

The fundamental matrix is calculated as below from the algorithm explained in the implementation section.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0.0046 \\ 0.0002 & -0.0046 & 0.0215 \end{bmatrix}$$

0.7 Image Rectification

Figure 3 shows the rectified images before paddings were applied for alignment. After the calculation of the offset, the images were translated by the amount calculated in the algorithm to be aligned. Figure 4 shows the rectified images and their alignment.

0.8 Disparity Map Calculation

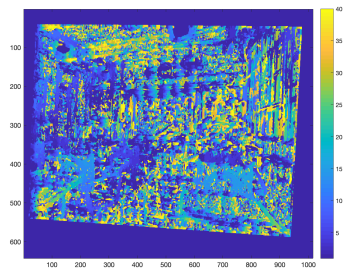
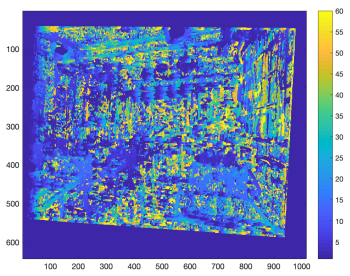
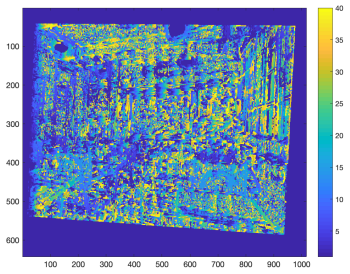
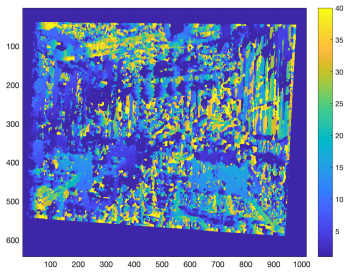
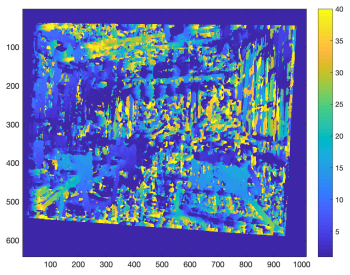
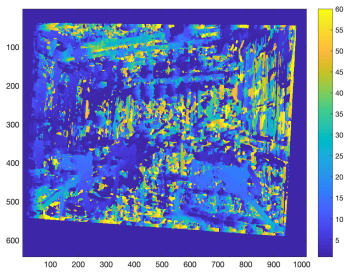
Table 1 shows the comparison of disparity maps created with different parameters. In my opinion, the map created with window size of 5 pixel, standard deviation of 2, and

Figure 3: Before alignment.

Figure 4: Rectified images and alignment.

maximum disparity of 40 looks the best. It is shown in Figure 5. Smaller window size tend to make the map look noisy and larger sigma made the regions rounder in the map. The difference between the maximum disparity of 40 and 60 was not great. Therefore, we can conclude that maximum disparity of 40 was enough to capture all the disparities.

Table 1: Comparison of different parameters of disparity map.

	
window size:5px, sigma:1, max_disparity:40	window size:5px, sigma:1, max_disparity:60
	
window size:3px, sigma:1, max_disparity:40	window size:5px, sigma:2, max_disparity:40
	
window size:9px, sigma:1, max_disparity:40	window size:9px, sigma:1, max_disparity:60

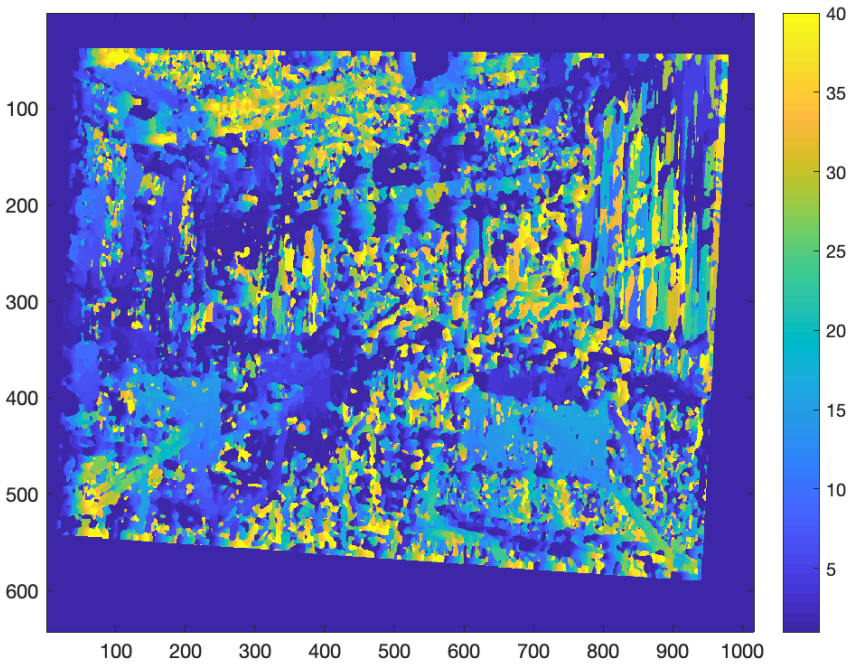


Figure 5: Depth map from 5px window, sigma 2, max disparity 40.