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.**

Part A: Bayer Image Interpolation

Introduction

The task of part A is to generate a M x N x 3 RGB image from a M x N bayer pattern image. I was given an option to adopt either bilinear method or bicubic method for interpolation, and I decided to use bilinear method. In order to complete the task, I was asked to implement the following function,

```
1 function rgb_img = bayer_to_rgb_bicubic(bayer_img)
```

where bayer_img is an M x N image, and rgb_img is an M x N x 3 image.

Process and Algorithm

First, I initialized three M x N matrices, named red_bayer, green_bayer, and blue_bayer, to store red, green, and blue intensity values, respectively, that were previously stored in bayer_img. I extracted each of the red, green, and blue values by going through all pixels and determining whether their indices are even or odd. Since the image uses RGGB Bayer pattern, if row and column indices are both odd, it was a red pixel. Similarly, if they are both even, it was a blue pixel, and for the other cases, it was a green pixel.

After I stored each of the red, green, and blue pixel values into three separate matrices, I used bilinear interpolation to fill in the empty cells. I ran through all pixels and checked what color value was occupying the cell with same indices in bayer_img. If it was a red cell, or the row and column indices were both odd, I interpolated cells in green_bayer and blue_bayer with same indices. Similarly, if the occupying cell was a blue cell, or the row

and column indices were both even, I interpolated cells in red_bayer and green_bayer. For all other cases, I interpolated cells in red_bayer and blue_bayer.

To implement actual interpolation, I used neighboring cells. Since the bayer pattern is fixed to be RGGB, all interpolation patterns could be hard coded. For example, if a red pixel is occupying (i, j) of bayer_img, the green pixel at (i, j) could be interpolated by taking average of four pixels at $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. Also, blue pixel at (i, j) could be interpolated by taking average of four pixels at $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$, and $(i+1, j+1)$. Tricky cases were corners and boundaries. I hard coded all of those cases, which can be checked in the bayer_to_rgb_bicubic.m file.

Results

Shown below is the result from main code, which runs bayer_to_rgb_bicubic on two bayer pattern images to obtain respective RGB images. Figure 1 shown below shows those two images with feature points added on them.

Figure 1: RGB images interpolated from Bayer patterns

Part B: Fundamental Matrix

Introduction

The task of this part was to find the fundamental matrix using the normalized eight-point algorithm. Please note I was unaware of normalize_points.m file that has been already provided, and I implemented normalization on my own. This function

```
function f = calculate_fundamental_matrix(pts1, pts2)
```

takes in two points, pts1 and pts2, which are 8 x 2 doubles. These two points are feature points. Given these two sets of points, this function returns f, which is a 3 x 3 fundamental matrix.

Process and Algorithm

First, I implemented normalization of points so that the average distance from origin to all points is $\sqrt{2}$. I scaled the points so that average distance from the points to their centroid is $\sqrt{2}$ and moved the points such that their centroid lies on the origin.

I calculated centroid by taking average of all points. Then, I found distance between centroid and the points to calculate the mean distance mean_dist. By scaling the distance between centroid and a specific point by $\frac{\sqrt{2}}{mean\_dist}$, I could make average distance $\sqrt{2}$. Then, to move centroid to the origin, I subtracted $\frac{\sqrt{2}}{mean\_dist} \times centroid$ from all scaled points. The code snippet below shows actual implementation.

```
l = length(pts1);
center1 = [mean(pts1(:,1)) mean(pts1(:,2))];
center2 = [mean(pts2(:,1)) mean(pts2(:,2))];
center1 = repmat(center1, 8, 1);
center2 = repmat(center2, 8, 1);
dist1 = sqrt(sum((pts1 - center1).^2, 2));
mean_dist1 = mean(dist1);
dist2 = sqrt(sum((pts2 - center2).^2, 2));
mean_dist2 = mean(dist2);
norm1 = [sqrt(2)/mean_dist1, 0,
        (-sqrt(2)/mean_dist1)*center1(1,1);...
        0, sqrt(2)/mean_dist1, (-sqrt(2)/mean_dist1)*
           center1(2,2);...
        0, 0, 1];
norm2 = [sqrt(2)/mean_dist2, 0,
        (-sqrt(2)/mean_dist2)*center2(1,1);...
        0, sqrt(2)/mean_dist2, (-sqrt(2)/mean_dist2)*
           center2(2,2);...
        0, 0, 1];
```

After finding normalization matrices norm1 and norm2, I multiplied them with feature points to find normalized points, npts1 and npts2. They represented (x, y) and (x', y'), respectively, where x, x', y, and y' are column vectors. I combined them to obtain matrix A by following given model in the instructions and calculated f, which is the eigenvector of $A^T A$ corresponding to the smallest eigenvalue.

After reshaping f to a 3 x 3 matrix, or F, I performed single value decomposition on F to find U, S, and V. Matrix S here contains singular values in its diagonal. To find the fundamental matrix, I removed the smallest singular value and calculated USV^T again, with the modified S, to get fundamental matrix F. Lastly, to transform fundamental matrix back to original units, I calculated $norm2' Fnorm1$, which is the fundamental matrix in original units. The code snippet below shows this process.

```
npts1 = [transpose(npts1(1,:)) transpose(npts1(2,:))];
npts2 = [transpose(npts2(1,:)) transpose(npts2(2,:))];
x = npts1(:,1);
y = npts1(:,2);
xp = npts2(:,1);
yp = npts2(:,2);
A = [x.*xp x.*yp x y.*xp y.*yp y xp yp ones(1, 1)];
[V,D] = eig(A'*A);
F=reshape(V(:,1), 3, 3);
[U,S,V] = svd(F);
F = U*diag([S(1,1) S(2,2) 0])*V';
F = norm2'*F*norm1;
f = F;
```

Results

To test the correctness of my implementation, I compared the results from calculate_fundamental_matrix(pts1, pts2), which is the function I implemented, with the results from estimateFundamental-Matrix(pts1, pts2, 'Method','Norm8Point'), which is the built-in function of MATLAB. Fundamental matrices resulting from each function are shown below:

$$implemented = \begin{bmatrix} 3.3023e-08 & -7.9635e-07 & -2.3878e-05 \\ 7.2653e-07 & -4.1038e-08 & -0.0046 \\ -1.6755e-04 & 0.0046 & -0.0215 \end{bmatrix}$$

$$builtin = \begin{bmatrix} -1.4734e-06 & 3.5531e-05 & 0.0011 \\ -3.2416e-05 & 1.8310e-06 & 0.2062 \\ 0.0075 & -0.2074 & 0.9572 \end{bmatrix}$$

As shown, the implemented function produces fundamental matrix that is the built-in fundamental matrix scaled by -0.02241252877. Note that since fundamental matrix is used to make correspondence between two points x and x', scale factor is not important; two fundamental matrices that differ by a scale factor behave in the same way.

Part C: Image Rectification

Introduction

The task of this part was to generate rectified image pairs given two images and their respective homography matrices that have been generated from fundamental matrices. In this part, I implemented the following function,

```
function [rectified1, rectified2] = rectify_stereo_images
    (img1, img2, h1, h2)
```

where img1 and img2 are M x N x 3 images, h1 and h2 are 3 x 3 homography matrices, and rectified1 and rectified2 are M' x N' x 3 images. The goal of this function was to produce two rectified images such that they contain respective images, which are img1 and img2, and to assign them the same size such that an anaglyph of those images can be created.

Process and Algorithm

Three important MATLAB functions were used to implement this function: 1) projective2d, 2) imwarp, and 3) transformPointsForward.

First, I called projective2d on homography matrices h1 and h2 to create hform1 and hform2, which are transformations of type "projective2d" object. This was an important step because imwarp takes in transformation as projective2d.

Then, I applied transformPointsForward on the four corners of the original image, once with hform1 and once with hform2. This was aimed at determining where the corners, which are $(1, 1)$, $(540, 1)$, $(1, 960)$, and $(540, 960)$, would map to once the transformation was applied. Rectified image would show the whole image after transformation if all of its corners after the transformation were contained within the appropriate range, and that is why I applied transformPointsForward on the corners - to determine the range and size of image.

After I found mapping of corners, I determined the range of rectified images by looking at the minimum and maximum mapped indices. Letting x_min, x_max, y_min, and y_max be the minimum and maximum of x and y mapped values, I set range to be [x_min:x_max, y_min:y_max]. Also, I set the size of rectified images to be [x_max-x_min, y_max-y_min]. Then, I performed actual transformation on the images by using imwarp function and imref2d function. As a result, the function produced two rectified, aligned images.

Shown below is the code snippet to show how the three functions have been used to created rectified images.

```
hform1 = projective2d(h1);
hform2 = projective2d(h2);
...
dim1 = transformPointsForward(hform1, corners);
dim2 = transformPointsForward(hform2, corners);
...
r1 = imwarp(img1, hform1, 'OutputView', imref2d([(x_max-
    x_min), (y_max-y_min)], [y_min y_max], [x_min x_max]))
    ;
r2 = imwarp(img2, hform2, 'OutputView', imref2d([(x_max-
    x_min), (y_max-y_min)], [y_min y_max], [x_min x_max]))
    ;
```

Results

To test if the function has been implemented correctly, the main function contains codes to draw out rectified images and their anaglyph. I compared the results with the example provided in the instructions to determine whether my results were correct or not. My results produced correct rectification and anaglyph, as shown below in figure 2.

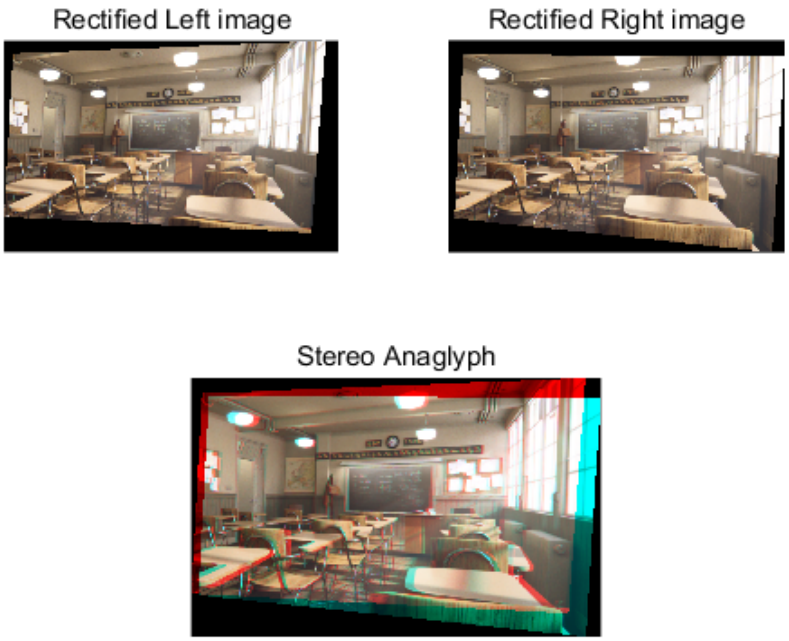


Figure 2: Rectified images and their anaglyph

Part D: NCC Disparity Map

Introduction

The task of this part was to generate a disparity map for two rectified images by using Normalized Cross Correlation(NCC). The formula for finding disparity by using NCC is given as:

$$NCC = \frac{\sum_i \sum_j A(i,j)B(i,j)}{\sqrt{\sum_i \sum_j A(i,j)^2}\sqrt{\sum_i \sum_j B(i,j)^2}}.$$

To complete the task, I implemented the following function,

```
function d = calculate_disparity_map(img_left, img_right,
    window_size, max_disparity)
```

where img_left and img_right are M x N doubles, window_size is the size of disparity window given as int, and max_disparity is maximum disparity given as int. The function returns d, which is an M x N doubles that contains maximum disparity value at each image index.

Process and Algorithm

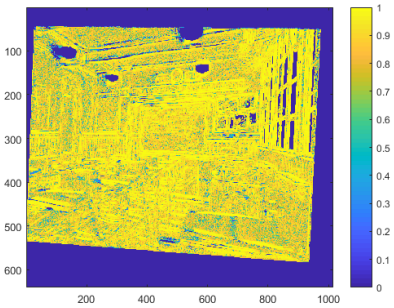
Algorithms for this task are pretty straightforward. I used a naive box filter method, where for every pixel, box filter is applied to specific disparity d. I ran a double for loop, going through all pixels. Inside that double loop, I ran through all possible values of d, from 1 to min(max_disparity, w-j-half), where w is width of image, j is current x-index being compared, and half is window_size divided by 2 - to avoid index overflow near the boundaries. Going through all pixels of the image, I created a submatrix of img_left and img_right of sizes (window_size x window_size). Submatrix of img_left was centered at (i,j), and submatrix of img_right was centered at (i,j+d). Then, the NCC value was stored at temp(i,j,d), where temp is a M x N x max_disparity matrix that stores all NCC disparity values.

After calculating NCC values of all pixels, the best disparity for each pixel was chosen. Again, I ran through all pixels using a double for loop, and for each (i,j) pixel, I checked all disparity values - from 1 to max_disparity. The highest disparity value was chosen as the winner and stored to cost_vol(i,j), where cost_vol is an M x N matrix that stores winning disparity values.

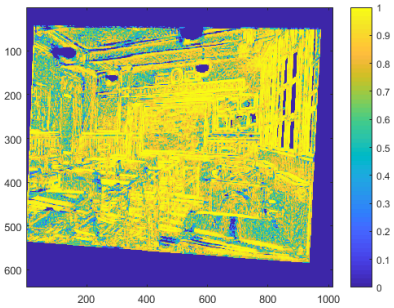
Different window sizes produce different disparity maps. Larger window sizes ensure a smoother disparity map with less noise, but it sacrifices details and produces edge bleeding. Smaller windows sizes ensure meticulous details, but it produces more noise. I tested window sizes of 3, 5, 9, and 15. Similarly, I also tested maximum disparity values of 20, 40, and 60.

Results

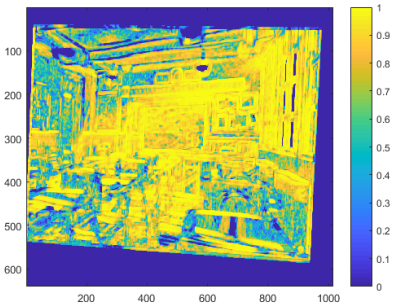
Since the disparity map differs based on window size and maximum disparity value, I tested out several possible combinations. The first set of images shows varying window sizes - 3, 5, 9, and 15 - with set maximum disparity value of 40. As clearly seen, as window size increases, the disparity map shows less noise and more smoothness but also more edge bleeding and less details.



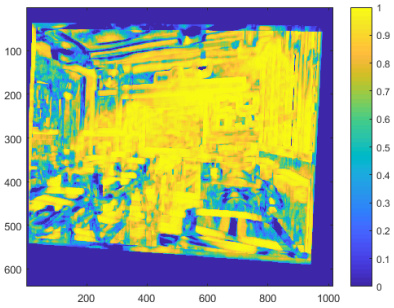
(a) Window size: 3,
Maximum disparity: 40



(b) Window size: 5,
Maximum disparity: 40

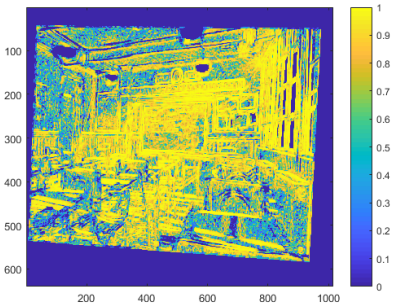


(c) Window size: 9,
Maximum disparity: 40

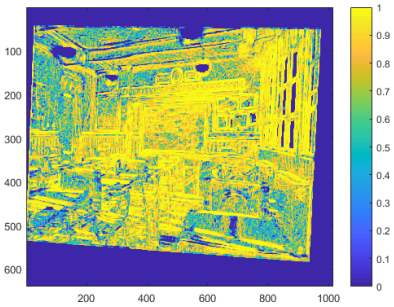


(d) Window size: 15,
Maximum disparity: 40

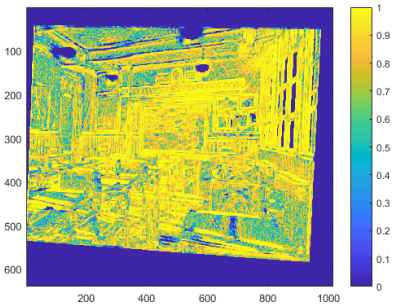
Similarly, I tested out disparity maps with varying maximum disparity - 20, 40, and 60 - with set window size of 5. In general, as the maximum disparity increases, disparity values increase to be closer to 1, thus making the disparity map less color-distinct.



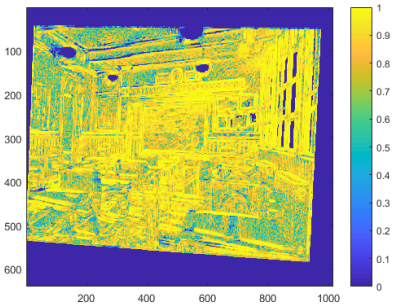
(a) Window size: 5,
Maximum disparity: 10



(b) Window size: 5,
Maximum disparity: 20



(c) Window size: 5,
Maximum disparity: 40



(d) Window size: 5,
Maximum disparity: 60