Computer Vision
Mini-project 2
CS 670, Fall 2019

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

Oct 06, 2019

Contents

1 Problem 1: Color Image Demosaicing **4**
- 1.1 Bayer Pattern . . . 4
- 1.2 Nearest-Neighbor Interpolation . . . 4
  - 1.2.1 Red and Blue channels Demosaicing . . . 5
  - 1.2.2 Green channel Demosaicing . . . 8
- 1.3 Linear Interpolation . . . 10
  - 1.3.1 Red and Blue channels Demosaicing . . . 11
  - 1.3.2 Green Channel Linear Demosaicing . . . 15
- 1.4 Adaptive Gradient Interpolation . . . 17
  - 1.4.1 Green Channel Adaptive Gradient Interpolation Demosaicing . . . 17
  - 1.4.2 Improvements for Red and Blue Channels for Adaptive Gradient Interpolation for Demosaicing . . . 20
- 1.5 Output of `evalDemosaic.m` . . . 21
- 1.6 Closing Remarks . . . 26

2 Depth from Disparity **27**
- 2.1 Computing the Depth Image: Depth from Stereo . . . 27
- 2.2 Window Size and Patch Similarity Measure . . . 32
- 2.3 Stereo versus Photometric Stereo . . . 39
- 2.4 Informative Patches . . . 39
  - 2.4.1 Measuring an Information Patch . . . 40
  - 2.4.2 Visualization of R . . . 41
  - 2.4.3 What does R indicate quantitatively . . . 41

3 Extensions **44**
- 3.1 Extension 1: Transformed Color Spaces for Demosaicing 44
  - 3.1.1 Divide Red and Blue Filter Values by Interpolated Green Channel . . . 44
  - 3.1.2 Results . . . 46
  - 3.1.3 Log of Division of Red and Blue Filter Values by Interpolated Green Channel . . . 48
  - 3.1.4 Results . . . 50
- 3.2 Extension 2: Alternative Sampling Patterns . . . 52
  - 3.2.1 Simplifications . . . 52
  - 3.2.2 Brute-Force Approach . . . 52
  - 3.2.3 Good Patterns . . . 52
  - 3.2.4 Overall Errors and Concluding Remarks . . . 54

A

Matlab code for Problem 1: Color Image Demosaicing **56**

A.1 Implementation of `demosaicImage.m` 56

B

Matlab code for Problem 2: Depth from Disparity **73**

B.1 Implementation of `depthFromStereo.m` 73

B.2 Implementation of `visualizeInformation.m` . . 74

C

Matlab code for Extensions **76**

C.1 Implementation of `demosaicDivide.m` 76

C.2 Implementation of `demosaicLogDivide.m` 77

C.3 Implementation of `demosaicBruteForce.m` 79

1 Problem 1: Color Image Demosaicing

1.1 Bayer Pattern

- In digital cameras; the red, the blue and the green sensors are interlaced in the Bayer pattern
- For each channel, we need to interpret the missing values of that grid/ pattern to get all three interpolated channels, to make up a complete RGB image
- In this Bayer Pattern, red cells start from top left corner (1, 1) and skip every other pixels both row-wise and column-wise
- Blue cells start from (2, 2) and skip every other pixels both row-wise and column-wise
- Green cells fill the rest of the cells which are not occupied by red or blue; i.e., for the alternative rows, first row is where green cells start from (1, 2), skips every other column and the second row is where green cells start from (2, 1), skips every other column
- Green cells will be double the amount of red or blue cells because human eyes are more sensitive to green intensity

In the next sections, we will see few interpolation algorithms, each in detail and what kind of results they produce.

1.2 Nearest-Neighbor Interpolation

Here, we interpolate each unknown pixel's value by simply making it of the value of it's closet neighbor whose value we know from the filter pattern.

We interpolate the "missing" pixels from each color channel with the nearest pixel.

1.2.1 Red and Blue channels Demosaicing

As show in Figure 1, we can see the *red channel* mosaic has repetitive grid of 2 x 2 grids where the 2 x 2 grid is always in pattern of

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

This matrix works like a mask, where **one means the value is already available** in the *red channel* of Bayer Filer, and we need to **interpolate values for the pixels which are at 0 in the mask**.

Figure 1: Red Channel Nearest Neighbor Demosaicing [1]

Hence, we can just copy the (1, 1) pixel's value of all the 2 x 2 grids to the pixels at (1, 2) [bottom]; (2, 1) [right] and (2, 2) [bottom right].

Matlab Implementation for Nearest Neighbor Algorithm for Red Channel*

```
%% Red channel (odd rows and columns);

% Initialize the resultant red channel to all the -1
    values
redValues = ones(imageHeight, imageWidth)*-1;
% Put the already known red values to the resultant
    image corresponding pixels
redValues(1:2:imageHeight, 1:2:imageWidth) = im(1:2:
    imageHeight, 1:2:imageWidth);

% For every alternate row, starting from 1st row
```

```
for i = 1:2:imageHeight
    % For every alternate column, starting from 1st
       column
    for j = 1:2:imageWidth
        % Put the same value as that of known pixel's,
           to the pixel at right
        if( i + 1 <= imageHeight)
            redValues(i+1, j) = redValues(i, j);
        end
        % Put the same value as that of known pixel's,
           to the pixel at bottom
        if( j + 1 <= imageWidth)
            redValues(i, j+1) = redValues(i, j);
        end
        % Put the same value as that of known pixel's,
           to the pixel at bottom right
        if( j + 1 <= imageWidth && i + 1 <=
           imageHeight)
            redValues(i+1, j+1) = redValues(i, j);
        end
    end
end

%mosim(:, :, 1) = padarray(redValues, [1, 1], '
   replicate', 'post');
% Make the interpolated red value matrix, the red
   channel of the final output image
mosim(:, :, 1) = redValues;
```

*Full code in Appendix A.1

Same concept applies for Blue Channel as follows:

It follows the same basic mask concept as red channel, but a minor change is that the position of the blue cells demand us to interpolate different unknown cells than we did in red mask's case.

We can see the *blue channel* mosaic has repetitive grid of 2 x 2 grids

where the 2 x 2 grid is always in pattern of

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

because blue cells are starting at $(2, 2)$.

Hence, we can just copy the (2, 2) pixel's value of all the 2 x 2 grids to the pixels at (1, 2) [top]; (2, 1) [left] and (1, 1) [top left].

OR we can use the exact same method of interpolating bottom, right and bottom right pixels by assuming that the known pixel value is at (1, 1) of the 2 x 2 grid.

That will leave the 1^{st} row and column "un-interpolated". So, we just interpolate them in the end with the nearest values from 2^{nd} row and 2^{nd} column respectively.

Matlab Implementation for Nearest Neighbor Algorithm for Blue Channel*

```
%% Blue channel (even rows and columns);

% Initialize the resultant blue channel to all the -1
    values
blueValues = ones(imageHeight, imageWidth)*-1;

% Put the already known blue values to the resultant
    image corresponding pixels
blueValues(2:2:imageHeight, 2:2:imageWidth) = im(2:2:
    imageHeight, 2:2:imageWidth);

% For every alternate row, starting from 2nd row
for i = 2:2:imageHeight
    % For every alternate column, starting from 2nd
        column
    for j = 2:2:imageWidth
        % Put the same value as that of known pixel's,
            to the pixel at right
        if( i + 1 <= imageHeight)
            blueValues(i+1, j) = blueValues(i, j);
        end
```

```
17          % Put the same value as that of known pixel's,
                to the pixel at bottom
18          if( j + 1 <= imageWidth)
19              blueValues(i, j+1) = blueValues(i, j);
20          end
21          % Put the same value as that of known pixel's,
                to the pixel at bottom right
22          if( j + 1 <= imageWidth && i + 1 <=
               imageHeight)
23              blueValues(i+1, j+1) = blueValues(i, j);
24          end
25      end
26  end
27
28  % For the first row, copy the values from 2nd row,
       they are the "nearest"
29  blueValues(1,:) = blueValues(2,:);
30  % For the first column, copy the values from 2nd
       column, they are the "nearest"
31  blueValues(:,1) = blueValues(:,2);
32  % Make the interpolated blue value matrix, the blue
       channel of the final output image
33  mosim(:, :, 3) = blueValues;
```

*Full code in Appendix A.1

1.2.2 Green channel Demosaicing

The Green channel the algorithm is a bit different, as there are only two missing colors, instead of three. For each block of 2 x 2 pixels, it fills the first “missing” pixel with the same value as the first pixel and the second “missing” pixel with the same value as the second pixel. See Figure 2.

We can see the *green channel* mosaic has repetitive grid of 2 x 2 grids where the 2 x 2 grid is always in pattern of

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We interpolate pixel value at (1, 2) with the value of pixel at (1, 1) and the pixel value at (2, 1) with the value of pixel at (2, 2).

Figure 2: Green Channel Nearest Neighbor Demosaicing [1]

Note, that the matrix grid shown above is for the Figure 2, for the bayer pattern given in the question, it will be:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

And we will interpolate pixel value at (1, 1) with the value of pixel at (1, 2) and the pixel value at (2, 2) with the value of pixel at (2, 1).

Matlab Implementation for Nearest Neighbor Algorithm for Green Channel*

```
%% Green channel

% Initialize the resultant green channel to all the -1
    values
greenValues = ones(imageHeight, imageWidth)*-1;

% We put the already known green values to the
    resultant image corresponding pixels
% Odd rows have green pixel's known values at even
    columns
greenValues(1:2:imageHeight, 2:2:imageWidth) = im(1:2:
    imageHeight, 2:2:imageWidth);
% Even rows have green pixel's known values at odd
    columns
greenValues(2:2:imageHeight, 1:2:imageWidth) = im(2:2:
    imageHeight, 1:2:imageWidth);
```

```
11
12 % Odd rows
13 for i = 1:2:imageHeight
14     % Even columns
15     for j = 2:2:imageWidth
16         % Put the same value as that of known pixel's,
               to the pixel at left
17         if (i - 1 > 0)
18             greenValues(i - 1, j) = greenValues(i, j);
19         end
20     end
21 end
22
23 % Even rows
24 for i = 2:2:imageHeight
25     % Odd columns
26     for j = 1:2:imageWidth
27         % Put the same value as that of known pixel's,
               to the pixel at right
28         if (i + 1 <= imageHeight)
29             greenValues(i + 1, j) = greenValues(i, j);
30         end
31     end
32 end
33
34 % Make the interpolated green value matrix, the green
     channel of the final output image
35 mosim(:, :, 2) = greenValues;
```

*Full code in Appendix A.1

NOTE: All the **output images** and the **error values** are shown in the **Section 1.5** so we can see the actual comparison with the other methods, rather than just seeing the output of one method with only ground truth/ input images to compare to.

1.3 Linear Interpolation

Linear interpolation takes average value of all neighbor pixels.

1.3.1 Red and Blue channels Demosaicing

We will have to make 3 kinds of calculation for Red and Blue cells, we focus on Red cells, the logic behind blue cells is very similar:

1. Diagonal Cross Interpolation

In the case shown in Figure 3, the “missing” pixel has four neighbors cells around it. The average value of those four pixels will be the new value for the “missing” pixel.

Figure 3: Diagonal Cross Interpolation for Red and Blue Pixels where pixel values are known for a concerned pixel’s diagonal neighbors[1]

2. Horizontal Interpolation

In Figure 4, the “missing” pixel has a cell at its left and a cell at its right. Similarly to the previous case, the average value of those two pixels will be the new value for the “missing” pixel.

3. Vertical Interpolation

For the last case, the “missing” pixel has two cells, one at its top and one at its bottom. The average value of those two pixels will be the new

Figure 4: Horizontal Interpolation for Red and Blue Pixels where pixel values are known for a concerned pixel's left and right neighbors [1]

value for the "missing" pixel. See Figure 5

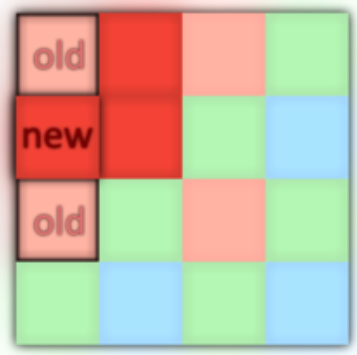


Figure 5: Vertical Interpolation for Red and Blue Pixels where pixel values are known for a concerned pixel's top and bottom neighbors [1]

Matlab Implementation for Linear Algorithm for Red Channel*

```
%% Red channel (odd rows and columns);
% Initialize the resultant red channel to all the -1
    values
redValues = ones(imageWidth, imageHeight)*-1;
% Create a mask which shows which pixels have known
    values and which are to be interpolated
```

```
redMask = zeros(imageWidth, imageHeight);
% Put the already known red values to the resultant image corresponding pixels
redValues(1:2:imageWidth, 1:2:imageHeight) = im(1:2:imageWidth, 1:2:imageHeight);
% Put mask value to 1/true where image pixels are already have known values
redMask(1:2:imageWidth, 1:2:imageHeight) = 1;

for i = 1:imageWidth
    for j = 1:imageHeight
        % If a pixel has its value yet to be interpolated
        if redValues(i, j) == -1
            count = 0;
            sum = 0;
            % Here we will just check all the 8 neighboring pixels. It will
            % take care of all the cases (diagonal, horizontal and
            % vertical) as only the redMask = 1 values will hold any
            % weight. Others will be simply 0 and won't contribute if a
            % case doesn't apply for them

            % Checking for 3 neighboring pixels at left side
            if (i - 1 > 0)
                sum = sum + redValues(i-1, j)*redMask(i-1, j);
                count = count + 1*redMask(i-1, j);
                if (j + 1 <= imageHeight)
                    sum = sum + redValues(i-1, j+1)*redMask(i-1, j+1);
                    count = count + 1*redMask(i-1, j+1);
                end
                if (j - 1 > 0)
                    sum = sum + redValues(i-1, j-1)*redMask(i-1, j-1);
```

```
33                  count = count + 1*redMask(i-1, j
                        -1);
34              end
35          end
36
37          % Checking for 3 neighboring pixels at
                right side
38          if (i + 1 <= imageWidth)
39              sum = sum + redValues(i+1, j)*redMask(
                    i+1, j);
40              count = count + 1*redMask(i+1, j);
41              if (j + 1 <= imageHeight)
42                  sum = sum + redValues(i+1, j+1)*
                        redMask(i+1, j+1);
43                  count = count + 1*redMask(i+1, j
                        +1);
44              end
45              if (j - 1 > 0)
46                  sum = sum + redValues(i+1, j-1)*
                        redMask(i+1, j-1);
47                  count = count + 1*redMask(i+1, j
                        -1);
48              end
49          end
50
51          % Checking for neighboring pixel at bottom
52          if (j + 1 <= imageHeight)
53              sum = sum + redValues(i, j+1)*redMask(
                    i, j+1);
54              count = count + 1*redMask(i, j+1);
55          end
56
57          % Checking for neighboring pixel at top
58          if (j - 1 > 0)
59              sum = sum + redValues(i, j-1)*redMask(
                    i, j-1);
60              count = count + 1*redMask(i, j-1);
61          end
62
63          % Take the average of the sum of the
                values of neighboring pixels which have
```

```
                    known
64                % values
65                redValues(i, j) = sum/count;
66            end
67        end
68    end
69    % Make the interpolated red value matrix, the red
          channel of the final output image
70    mosim(:, :, 1) = redValues;
```

*Full code in Appendix A.1

Matlab Implementation for Linear Algorithm for Blue Channel is exactly as the code for Red Channel with only locations differing

See Appendix A.1 for it

1.3.2 Green Channel Linear Demosaicing

CROSS INTERPOLATION: In this case, the "missing" pixel has four neighbors cells around it. The average value of those four pixels will be the new value for the "missing" pixel. See Figure 6

Figure 6: Linear Interpolation for Green Pixels where pixel values are known for a concerned pixel's all 4 neighbors, top, bottom, right and left [1]

Matlab Implementation for Linear Algorithm for Green Channel*

```
%% Green channel
% Initialize the resultant green channel to all the -1 values
greenValues = ones(imageWidth, imageHeight)*-1;
% Create a mask which shows which pixels have known values and which are to be interpolated
greenMask = zeros(imageWidth, imageHeight);
greenValues(1:2:imageWidth, 2:2:imageHeight) = im(1:2:imageWidth, 2:2:imageHeight);
greenMask(1:2:imageWidth, 2:2:imageHeight) = 1;
greenValues(2:2:imageWidth, 1:2:imageHeight) = im(2:2:imageWidth, 1:2:imageHeight);
greenMask(2:2:imageWidth, 1:2:imageHeight) = 1;

for i = 1:imageWidth
    for j = 1:imageHeight
        % If a pixel has its value yet to be interpolated
        if greenValues(i, j) == -1
            count = 0;
            sum = 0;
            % Here we will just check only 4 neighboring pixels [top,
            % bottom, left and right]. Becasuse at any unknown valued pixel
            % location, there will be the same pattern observed, which look
            % like as follows in the green mask:
            % 0 1 0
            % 1 0 1
            % 0 1 0

            % Averaging all 4 neighboring pixels
            if (i - 1 > 0)
                sum = sum + greenValues(i-1, j)*greenMask(i-1, j);
                count = count + 1*greenMask(i-1, j);
            end
            if (i + 1 <= imageWidth)
                sum = sum + greenValues(i+1, j)*greenMask(i+1, j);
```

```
32                  count = count + 1*greenMask(i+1, j);
33              end
34              if (j + 1 <= imageHeight)
35                  sum = sum + greenValues(i, j+1)*
                        greenMask(i, j+1);
36                  count = count + 1*greenMask(i, j+1);
37              end
38              if (j - 1 > 0)
39                  sum = sum + greenValues(i, j-1)*
                        greenMask(i, j-1);
40                  count = count + 1*greenMask(i, j-1);
41              end
42
43              % Take the average of the sum of the
                    values of neighboring pixels which have
                     known values
44              greenValues(i, j) = sum/count;
45          end
46      end
47  end
48  % Make the interpolated green value matrix, the green
       channel of the final output image
49  mosim(:, :, 2) = greenValues;
```

*Full code in Appendix A.1

1.4 Adaptive Gradient Interpolation

First, we will use Adaptive Gradient for Green Channel and use Linear Interpolation for Red and Blue Channels. Later, we will try to improve the Red and Blue Interpolation techniques.

1.4.1 Green Channel Adaptive Gradient Interpolation Demosaicing

This takes average of the pixels based on the structure of neighbors.

If **|top-pixel - botton-pixel| > |left-pixel - right-pixel|**
Then **interpolated-pixel = (left-pixel + right-pixel)/2**
Else **interpolated-pixel = (top-pixel + bottom-pixel)/2**

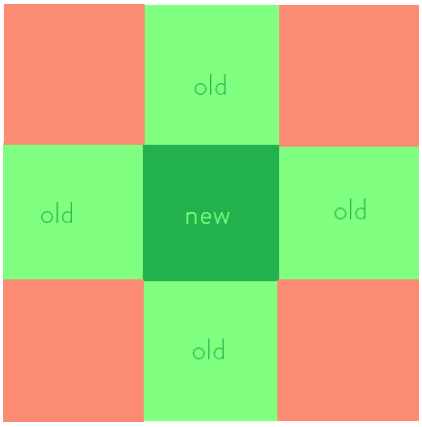


Figure 7: Adaptive Gradient Interpolation for Green Pixels where pixel values are known for a concerned pixel's all 4 neighbors, top, bottom, right and left [1]

Matlab Implementation for Adaptive Gradient Algorithm for Green Channel*

```
%% Green channel
% Initialize the resultant green channel to all the -1 values
greenValues = ones(imageWidth, imageHeight)*-1;
% Create a mask which shows which pixels have known values and which are to be interpolated
greenMask = zeros(imageWidth, imageHeight);
greenValues(1:2:imageWidth, 2:2:imageHeight) = im(1:2:imageWidth, 2:2:imageHeight);
greenMask(1:2:imageWidth, 2:2:imageHeight) = 1;
greenValues(2:2:imageWidth, 1:2:imageHeight) = im(2:2:imageWidth, 1:2:imageHeight);
greenMask(2:2:imageWidth, 1:2:imageHeight) = 1;

for i = 1:imageWidth
    for j = 1:imageHeight
        if greenValues(i, j) == -1
            sum_h = 0; abs_h = 0; sum_v = 0; abs_v = 0;

            % Finding left and right pixel values difference and sum for core, edge and
```

```
            corner cases
17          if (i - 1 > 0 && i + 1 <= imageWidth)
18              abs_h = abs(greenValues(i-1, j)*
                    greenMask(i-1, j) - greenValues(i
                    +1, j)*greenMask(i+1, j));
19              sum_h = greenValues(i-1, j)*greenMask(
                    i-1, j) + greenValues(i+1, j)*
                    greenMask(i+1, j);
20              sum_h = sum_h/2;
21          elseif i - 1 > 0
22              abs_h = greenValues(i-1, j)*greenMask(
                    i-1, j);
23              sum_h = greenValues(i-1, j)*greenMask(
                    i-1, j);
24          elseif i + 1 <= imageWidth
25              abs_h = greenValues(i+1, j)*greenMask(
                    i+1, j);
26              sum_h = greenValues(i+1, j)*greenMask(
                    i+1, j);
27          end
28
29          % Finding top and bottom pixel values
                difference and sum for core, edge and
                corner cases
30          if (j - 1 > 0 && j + 1 <= imageHeight)
31              abs_v = abs(greenValues(i, j+1)*
                    greenMask(i, j+1) - greenValues(i,
                    j-1)*greenMask(i, j-1));
32              sum_v = greenValues(i, j+1)*greenMask(
                    i, j+1) + greenValues(i, j-1)*
                    greenMask(i, j-1);
33              sum_v = sum_v/2;
34          elseif j - 1 > 0
35              abs_v = greenValues(i, j-1)*greenMask(
                    i, j-1);
36              sum_v = greenValues(i, j-1)*greenMask(
                    i, j-1);
37          elseif j + 1 <= imageHeight
38              abs_v = greenValues(i, j+1)*greenMask(
                    i, j+1);
```

```
39                  sum_v = greenValues(i, j+1)*greenMask(
                        i, j+1);
40              end
41
42              % Choose which value to adapt
43              if abs_v > abs_h
44                  greenValues(i, j) = sum_h;
45              else
46                  greenValues(i, j) = sum_v;
47              end
48          end
49      end
50  end
51  % Make the interpolated green value matrix, the green
        channel of the final output image
52  mosim(:, :, 2) = greenValues;
```

*Full code in Appendix A.1

1.4.2 Improvements for Red and Blue Channels for Adaptive Gradient Interpolation for Demosaicing

We tried to use Matlab's `imfilter` to interpolate the two channels, with a 3 x 3 grid at a time. We experimented with different weights, one of them is shown as below:

Matlab Implementation for Adaptive Gradient Algorithm for Red and Blue Channels*

```
1  % Interpolation of Red pixels
2  mosim(:,:,1) = imfilter(redValues,[1,2,1;
       2,4,2;1,2,1]/4);
3
4  % Interpolation of Blue pixels
5  mosim(:,:,3) = imfilter(blueValues,[1,2,1;
       2,4,2;1,2,1]/4);
```

*Full code in Appendix A.1

#	image	baseline	nn	linear	adagrad - 1	adagrad - 2
1	balloon.jpeg	0.179239	0.019551	0.012879	0.012002	0.012750
2	cat.jpg	0.099966	0.024257	0.013407	0.013399	0.013995
3	ip.jpg	0.231587	0.026676	0.014693	0.012823	0.016431
4	puppy.jpg	0.094093	0.014464	0.006126	0.005965	0.006618
5	squirrel.jpg	0.121964	0.039970	0.023109	0.023552	0.024288
6	pencils.jpg	0.181449	0.028519	0.017416	0.016823	0.012070
7	house.png	0.117667	0.033315	0.017240	0.015620	0.015936
8	light.png	0.097868	0.026459	0.017305	0.016551	0.017059
9	sails.png	0.074946	0.022190	0.013640	0.012881	0.013243
10	tree.jpeg	0.167812	0.028353	0.015211	0.014296	0.014963
	average	0.136659	0.026375	0.015103	0.014391	0.014735

Table 1: Output comparison for all 4 methods of interpolation: baseline, nearest neighbor, linear and adaptive gradient(v1: linear red and blue; v2: filtered red and blue)

Here, we gave more weight to direct neighbors rather than diagonal neighbors.

1.5 Output of `evalDemosaic.m`

Table 1 shows output of baseline, nearest neighbor, linear and adaptive gradient algorithms for each image.

We can see that linear and adaptive algorithms give a very near match to the original images.

adagrad - 1 is when we used linear interpolation for red and blue channels and adaptive interpolation for the green channel.

adagrad - 2 is when we used custom filter for red and blue channels and adaptive interpolation for the green channel.

The difference in accuracy is really minor but best result is achieved when we used the linear + adaptive interpolation technique.

Figures 8, 9, 10, 11, 12, 13, 14, 15, 16 and 17 are visual representation of the interpolation we get.

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

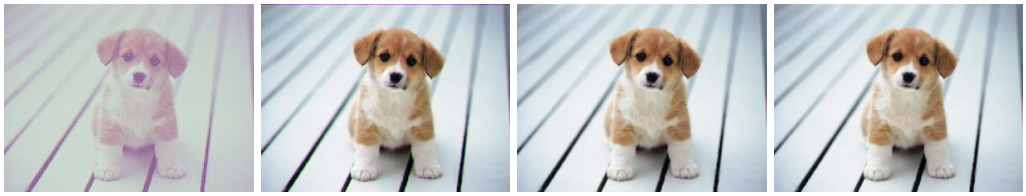
Figure 8: Different Demosaicing Interpolation Techniques for balloon.jpeg

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 9: Different Demosaicing Interpolation Techniques for cat.jpg

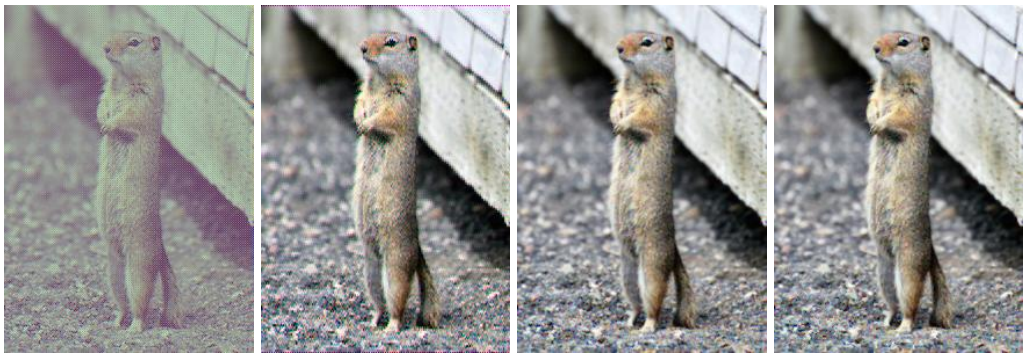
(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 10: Different Demosaicing Interpolation Techniques for ip.jpg



(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 11: Different Demosaicing Interpolation Techniques for puppy.jpg



(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 12: Different Demosaicing Interpolation Techniques for squirrel.jpg

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 13: Different Demosaicing Interpolation Techniques for pencils.jpg

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

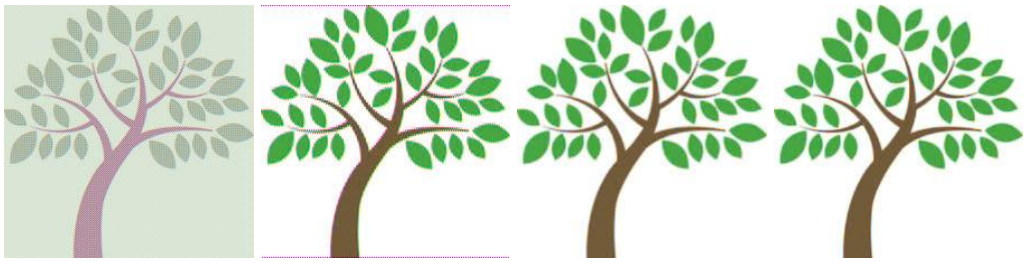
Figure 14: Different Demosaicing Interpolation Techniques for house.png

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 15: Different Demosaicing Interpolation Techniques for light.png

(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 16: Different Demosaicing Interpolation Techniques for sails.png



(a) Baseline Demosaicing | (b) Nearest Neighbor Demosaicing | (c) Linear Demosaicing | (d) Adaptive Gradient Demosaicing

Figure 17: Different Demosaicing Interpolation Techniques for tree.jpeg

1.6 Closing Remarks

The best algorithm is the **adaptive algorithm** where red and blue channels were just linearly interpolated.

That achieved an average error of as low as **0.014391**.

The close second algorithm performance wise will the adaptive interpolation we used for green channel where we compared top and bottom neighboring pixel value difference with left and right pixel differences and then decided whether to take horizontal average or the vertical average. Along with that, we used a custom filter of the form

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

for 3 x 3 grids of red and blue channels.

It achieved an average error as low as **0.014735**.

Linear interpolation also performed good as achieved error of only **0.015103**.

Nearest Neighbor didn't perform as good (although much better than baseline) with the error if **0.026375**. Because it's just a copy of its one neighbor, which might not be accurate for every case.

2 Depth from Disparity

First, we look at some definitions which will help us understand the question problem:

- EPIPOLAR GEOMETRY Epipolar geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. [3]

- STEREO is the extraction of 3D information from digital images. By comparing information about a scene from two vantage points, 3D information can be extracted by examining the relative positions of objects in the two panels. [4]

- DISPARITY is the distance between two corresponding points in the former and latter images of a stereo pair.

- DEPTH FROM STEREO If we were to perform the matching process for every pixel in the former image, finding its match in the latter frame and computing the distance between them, we would end up with an image where every pixel contained the distance/disparity value for that pixel in the former image.

We see a detailed explanation of above definitions in the next subsection:

2.1 Computing the Depth Image: Depth from Stereo

Suppose there's a camera who is capturing a world object from a certain fixed position. The image plane will have characteristics as shown in Figure 18. Now we add another camera, which is slightly away from the first one, but it is also fixed at a certain position, also capturing the same world object. The setup will look like shown in Figure 19 What happens in Epipolar geometry is that, [2]

- Image planes of cameras are PARALLEL to each other and to the baseline

- Camera centers are at SAME HEIGHT

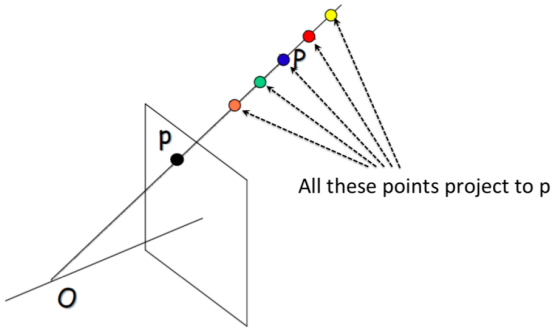


Figure 18: One camera setup: all points on projective line to P map to p

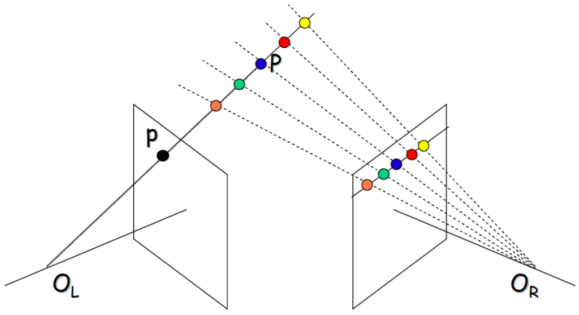


Figure 19: Two camera setup: All points on projective line to P in left camera map to a **line** in the image plane of the right camera

- FOCAL LENGTHS are the same
- Then, Epipolar lines fall along the HORIZONTAL SCAN LINES of the images

See Figure 20

If we see the top-down projection of this setup (Figure 21), we can apply similar triangle principle:

$$\frac{x - x'}{O - O'} = \frac{f}{z}$$

where $O - O'$ is the length of the BASELINE or the distance between the optical centers of the two camera. And $x - x'$ is the DISPARITY.

$$\therefore disparity = x - x' = \frac{(O - O') \cdot f}{z} = \frac{B \cdot f}{z}$$

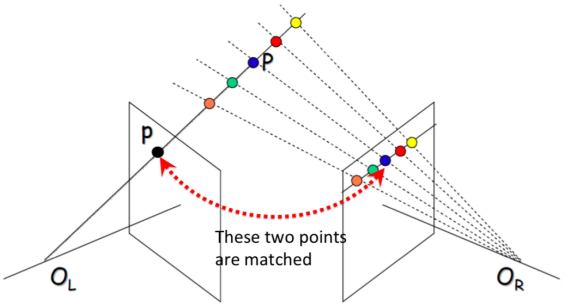


Figure 20: We need to find the corresponding epipolar line in the one image with respect to the other image, examine all pixels on the epipolar line and pick the best match

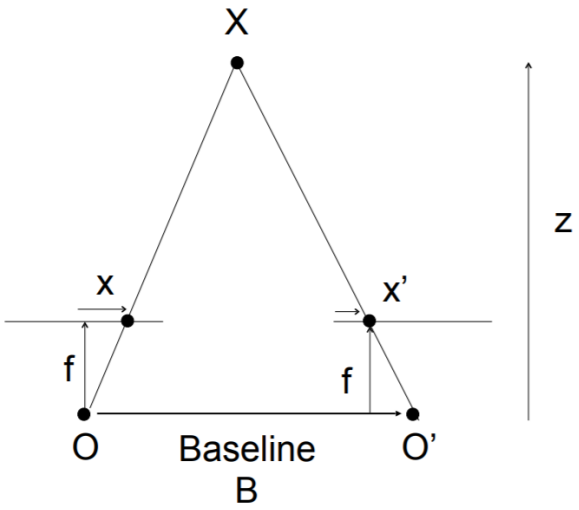


Figure 21: Disparity is inversely proportional to depth [2]

For this problem, we can assume that the images planes are parallel to each other and have the same focal length. Once we compute the disparity image by finding patch correspondences along the same epipolar line, we can then compute the depth through the following equation:

$$\therefore disparity = x - x' = \frac{B \cdot f}{depth}$$

where B is the baseline and f is the focal length. We don't have access

to B and f values, but those are constant throughout both images. This means that we will not be able to compute the true depth image, just the relative depth. $\therefore B \cdot f = 1$.

The algorithm is explained with the comments of the code as we go step by step:

Matlab Implementation for `depthFromStereo.m`*

```
function depth = depthFromStereo(img1, img2, ws)
% Implement this

% We take mean of all three channels of the first and
    second images to get a grayscale approximation of
    the images
img1_m = mean(img1, 3);
img2_m = mean(img2, 3);
% Initialize depth mean to all zeros
depth = zeros(size(img1_m, 1), size(img1_m, 2), 'uint8
    ');

% Keep the search within this range away from boundary
range = 50;
% Patch Box size
patch_size = 5;

[row, col] = size(img1_m);

% Traverse through all Epipolar Lines
for i = 1:row

    % Boundary and Index range checks
    row_min = max(1, i - patch_size);
    row_max = min(row, i + patch_size);

    % For each pixel on epipolar lines
    for j = 1:col

        % Set the patch sizes according to boudary
```

```
            ranges
28          col_min = max(1, j - patch_size);
29          col_max = min(col, j + patch_size);
30
31          % Patch box dimensions/ locations
32          pix_min = max(-range, 1 - col_min);
33          pix_max = min(range, col - col_max);
34
35          % Temporary Patch Box
36          t = img2_m(row_min:row_max, col_min:col_max);
37
38          % An array storing the SSD (Sum of Squared
               Difference)
39          count = pix_max - pix_min + 1;
40          diff = zeros(count, 1);
41
42          % Match the image patches on an epipolar line
43          for k = pix_min : pix_max
44              block = img1_m(row_min:row_max, (col_min+k
                   ):(col_max+k));
45              index = k - pix_min + 1;
46              diff(index, 1) = sumsqr(t - block);
47          end
48
49          % Get the pixel location where the patch
               disparity is the smallest
50          [x, y] = sort(diff);
51          m_index = y(1, 1);
52          disparity = m_index + pix_min - 1;
53          % Calculate the depth for that specific pixel
54          depth(i, j) = 1/disparity;
55          depth = imadjust(depth);
56      end
57  end
```

*Full code in Appendix B.1

The results are shown in the next subsection.

2.2 Window Size and Patch Similarity Measure

Here, we will try different matching costs, patch sizes and ranges to show the impact of all these on resultant depth images.

We used the following implementation of **Normalized Correlation** and **SSD** matching costs:

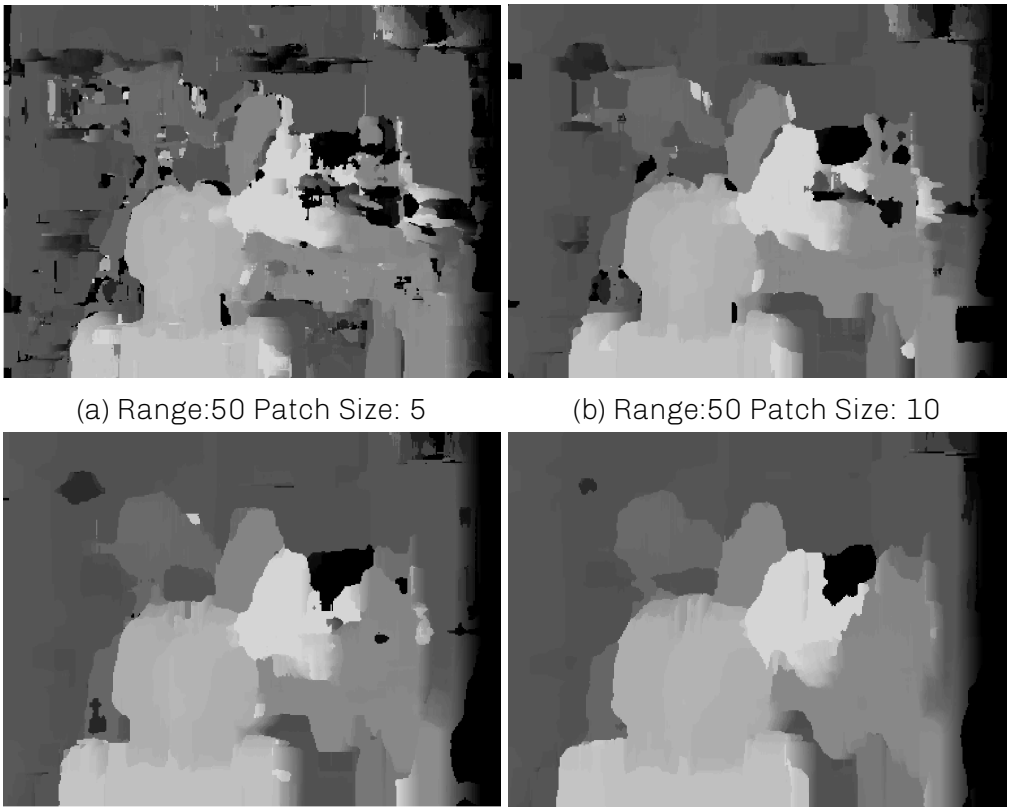
Matlab Implementation for two kinds of matching costs*

```
%% Normalized Correlation

% t = patch of the first image whose counterpart we
    want to find in second image, on the same epipolar
    line
% width = width of the image/ length of the epipolar
    line
% block = patch of the second image, it will be each
    patch on the epipolar line of the second image

% We are dividing the sum of the difference between
    two patches by the
% square roots of the variances of each patch
norm_corr(index, 1) = (1 / width) * (sum(t - mean(
    block))/sqrt(var(t)*var(block)));

%% Sum of Squarred Differences
diff(index, 1) = sumsqr(t - block);
```

*Full code in Appendix B.1

Figures 22 and 23 shows results with SSD and Normalized Correlation respectively.

- As the patch size increases, the result loses it's "sharpness" or the clarity round the edges. Ideal patch size should be somewhere from 10 to 15.
- But with small patch sizes, there's lot of noise in the depth map.
- Normalized Correlation gives better result than SSD.



(a) Range:50 Patch Size: 5 (b) Range:50 Patch Size: 10

(c) Range:50 Patch Size: 15 (d) Range:50 Patch Size: 20

Figure 22: Depth Map with SSD matching cost function for tsukuba.jpeg

In Figure 24, we manually tried to correct the errors with basic brushes and fill color tools, to check how close we are to ground truth if we can remove all the noise.

The same observations apply to `poster` images, Results are shown in Figures 25, 26 and 27.

Where our approach worked the best

As discussed earlier, our approach/ algorithm works best when:

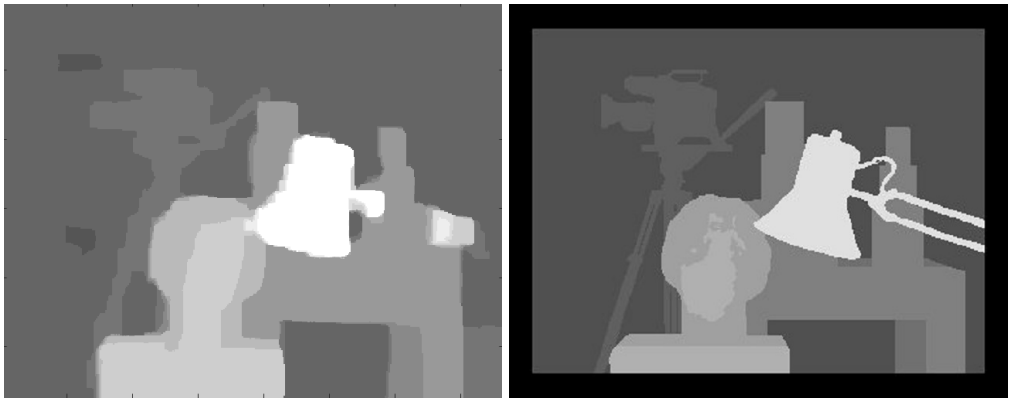
- Patch size is small e.g., 10 to 15 pixels, this will lead to more computation as the resolution goes higher; so this is an accuracy-time and space trade-off.

(a) Range:50 Patch Size: 5 (b) Range:50 Patch Size: 10

(c) Range:50 Patch Size: 15 (d) Range:50 Patch Size: 20

Figure 23: Depth Map with Normalized Correlation matching cost function for tsukuba.jpeg

- Also, with bigger patches, the artifact effect is more apparent on the boundaries.
- Therefore, this approach doesn't work as good on the boundaries with bigger patches as for the core part of the images.
- Different Matching Cost functions certainly makes a difference, SSD is a go-to method for any kind of error measurement, but application specific cost measurements work better. For patch similarity, we can also use DICE-SORENSON COEFFICIENT [5]
- We can use `imadjust` or other contrasting technique to make the difference between two nearby values of grayscale more apparent, to show the depth difference clearly.

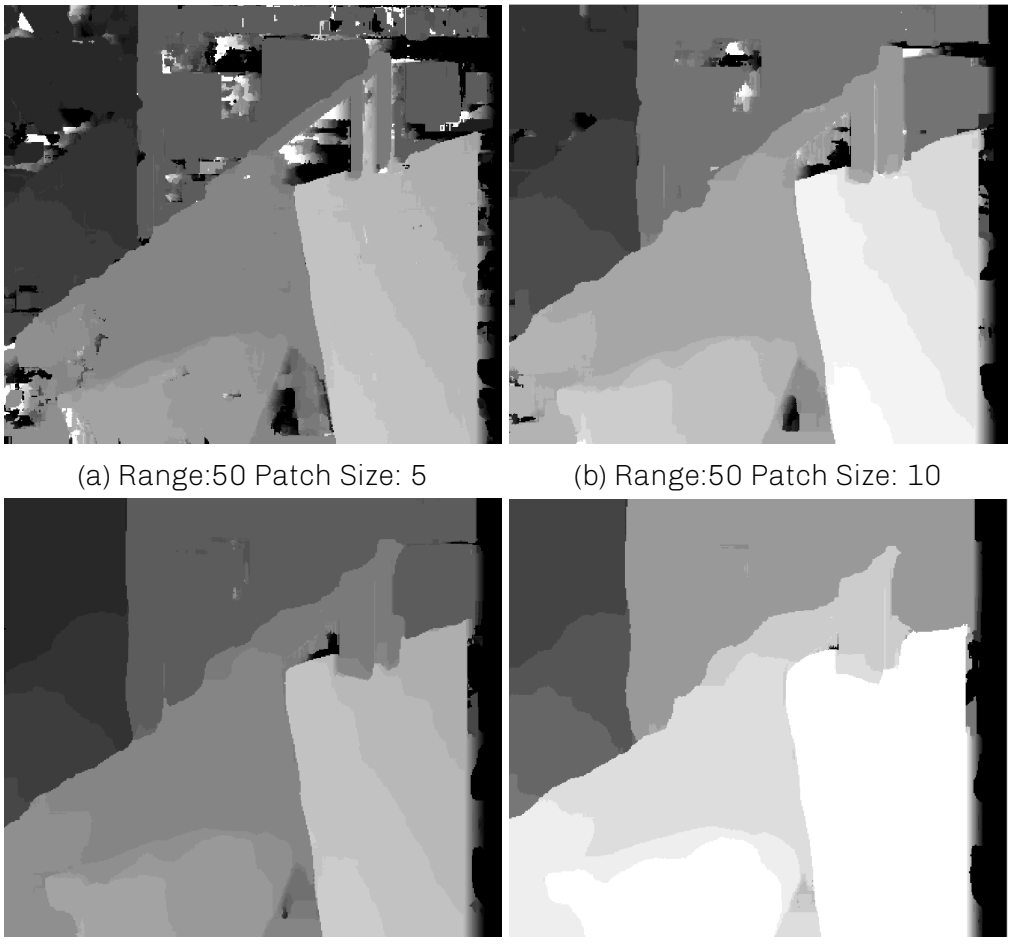


(a) Manually corrected the error spot of Figure 23 (b)

(b) Ground Truth

Figure 24: Manual Noise/ Error Spot Removal to see how close our result (a) can get to the ground truth (b)

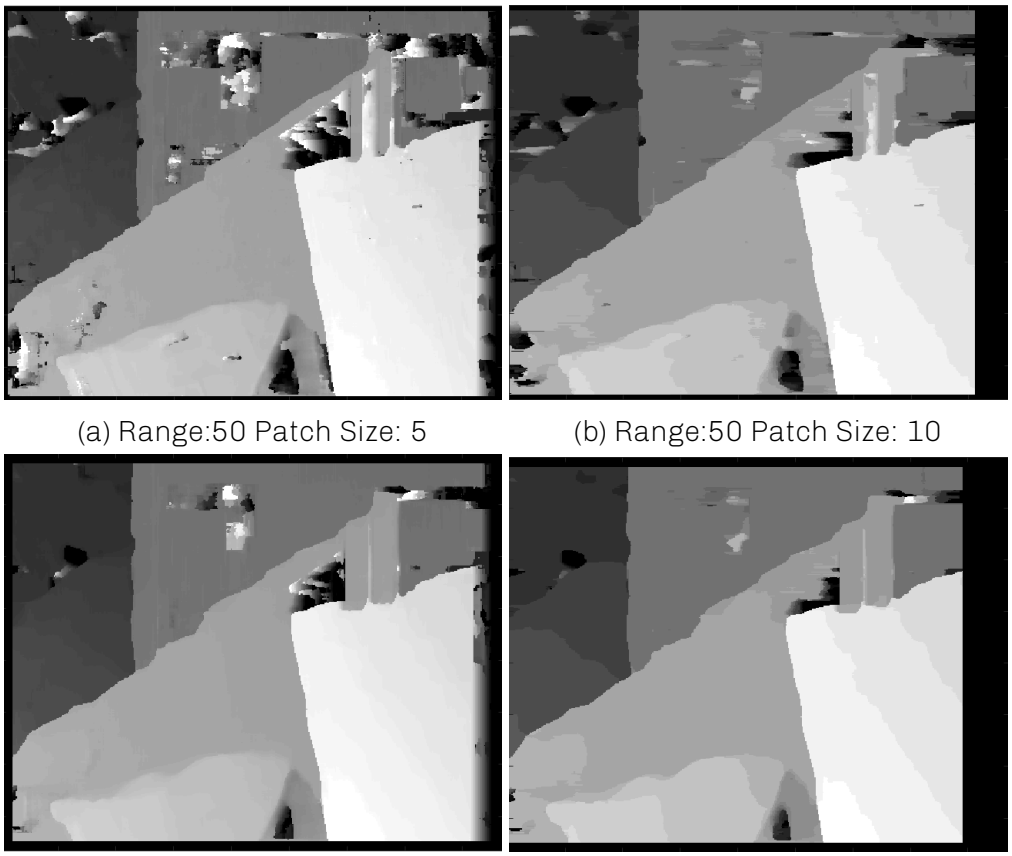
- Too small of a patch size will lead to matching a patch with totally unrelated, different part of the image because as the patch size goes smaller, the chances are higher of two pixel patterns looking very similar.
- The boundary artifact grows bigger as the distance between the two cameras increase.



(a) Range:50 Patch Size: 5 (b) Range:50 Patch Size: 10

(c) Range:50 Patch Size: 15 (d) Range:50 Patch Size: 20

Figure 25: Depth Map with SSD matching cost function for posters.jpeg



(a) Range:50 Patch Size: 5

(b) Range:50 Patch Size: 10

(c) Range:50 Patch Size: 15

(d) Range:50 Patch Size: 20

Figure 26: Depth Map with Normalized Correlation matching cost function for posters.jpeg

Figure 27: Manual Noise/ Error Spot Removal to see how it would look if no noise was present

2.3 Stereo versus Photometric Stereo

Below are listed the differences between Stereo and Photometric Stereo:

- Photometric stereo needs light falling on the object from many different Azimuthal and Elevation angles in order to guess depth with passable accuracy
- Stereo technique does not require change in lighting condition at all
- Comparatively, Stereo can work well with small number of sample images, while photometric stereo needs relatively higher number of images to guess depth
- Only small movements in camera position is acceptable for stereo, bigger/ wider distance between two cameras might result in totally wrong and illegible output
- Changing the lighting in stereo for one camera can affect the change of color of the same object, which will be an obstacle in trying to recognize the patch with the least disparity.
- Stereo is strictly based on epipolar geometry, if the cameras don't have the same height base, the whole concept of stereo which we applied won't work
- Stereo doesn't depict continuous depth like photometric stereo does, stereo just separate objects by attributing them to different "planes", but the actual smoothness of depth is not there

2.4 Informative Patches

An Informative Patch is a representation of a part or section of an image which gives us information about the shape or geometry of objects in the image.

E.g., it indicates where we are encountering flat surfaces, or edges or corners in an image.

It gives "Information" about image objects.

2.4.1 Measuring an Information Patch

One way to measure an informative patch is to computer the second order matrix:

$$M = \begin{bmatrix} \sum_{x,y} I_x I_x & \sum_{x,y} I_x I_y \\ \sum_{x,y} I_y I_x & \sum_{x,y} I_y I_y \end{bmatrix}$$

where gradients $I_x(u,v) = I(u+1,v) - I(u,v)$ and $I_y(u,v) = I(u,v+1) - I(u,v)$
and (x,y)s are all the neighbor pixels in the (u,v) pixel's window.

Before, we calculate the second-moment matrix for all pixels, we must convert an RGB image into grayscale because it really doesn't matter which color channel is intense at which part, edges will remain edges no matter in what color. So, converting them into grayscale simplifies the mathematics behind it.

Then, we simply calculate the information factor R out of eigenvalues of second-moment matrix M, which are:

$$\begin{bmatrix} \sum_{x,y} I_x I_x & \sum_{x,y} I_x I_y \\ \sum_{x,y} I_y I_x & \sum_{x,y} I_y I_y \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

$$\therefore Information factor R = \lambda_1 \lambda_2 - 0.03(\lambda_1 + \lambda_2)^2$$

Matlab Implementation for calculation of information patch and information factor*

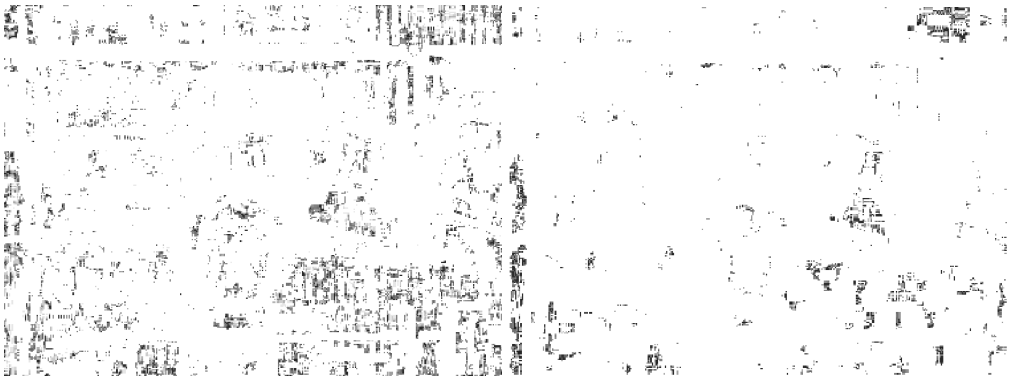
```
% For each pixel
for u = 1: width-r-1
    for v = 1:height-r-1
        % Computer Ixx, Ixy and Iyy for the given
           neighbors in the window
        Ixx = 0; Ixy = 0; Iyy = 0;
        for i = 1:r
            for j = 1:r
                Ix = im(u + i + 1, v) - im(u + i, v);
                Iy = im(u, v + i + 1) - im(u, v + i);
                Ixx = Ixx + Ix*Ix;
                Ixy = Ixy + Ix*Iy;
```

```
12                  Iyy = Iyy + Iy*Iy;
13              end
14          end
15          % Computer R
16          R(u, v) = Ixx*Iyy - 0.03*(Ixx + Iyy)*(Ixx +
                Iyy);
17      end
18  end
```

*Full code in Appendix B.2

2.4.2 Visualization of R

We produce information patch R for window radius $r = 5$ and $r = 10$ as shown Figure 28 and 29.



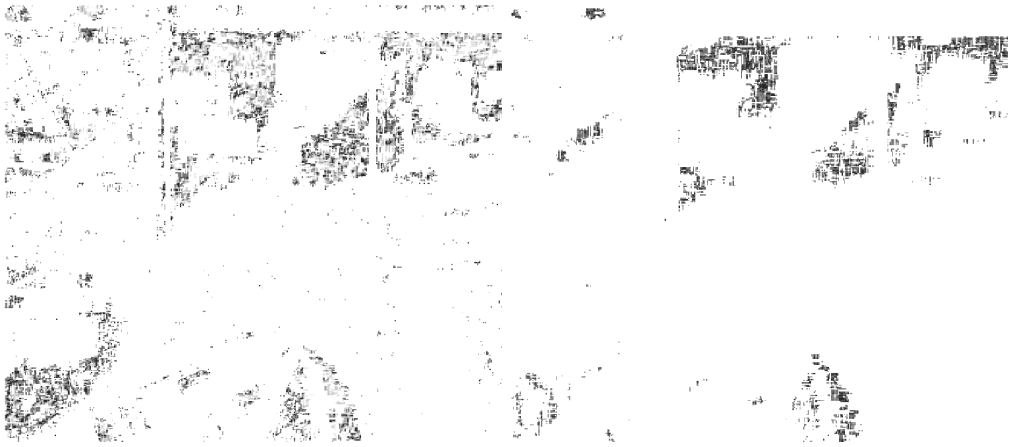
(a) window radius: 5 (b) window radius: 10

Figure 28: Information Patches for tsukuba.jpeg in grayscale

2.4.3 What does R indicate quantitatively

Referring to Figures 28 and 29, we make some observations and assertions:

- QUANTITATIVELY, when R is large; e.g., $R(u, v)$ for pixel at (u, v) is large, it would be because both λ_1 and λ_2 are large, making the first term of equation of R large, because second term would be multiplied by 0.03, so the subtraction will still keep R relatively very large because both eigenvalues are large, then we can say that it INDICATES A CORNER.



(a) window radius: 5 (b) window radius: 10

Figure 29: Information Patches for posters.jpeg in grayscale

- when R is somewhere near the mean or median of all the R values; e.g., $R(u, v)$ for pixel at (u, v) is moderate, it would be because one of λ_1 and λ_2 is large, while the other one is small, making the subtraction of two terms of equation of R depend on how large one λ is relative to the other one, the more the difference between the two eigenvalues, the sharper/ more accurate the edge depiction, then we can say that it INDICATES AN EDGE.
- If $\lambda_1 >> \lambda_2$ then the edge will be VERTICAL, else the edge will be HORIZONTAL.
- If R is small, then both λ_1 and λ_2 would be small, we can say that it INDICATES A FLAT SURFACE.
- So the value of R indicates the geometry of that particular patch window.
- In Figures 28 and 29, window radius going from 5 to 10 shows that the information indicator doesn't capture much details about the object because as the neighboring area increases, the pixel values make eigenvectors negate the total sum. Even if there's a large λ, the gradient and sum total will hide it's impact, resulting in edges fading into a flat region.
- In tsukuba image, for radius 5; we can see the outline of the sculpture, the lamp edges and bookshelf & whiteboard edges in the background.

- While for radius of 10, in tsukuba image, we can only decipher a very little part of sculture head, the lamp edges and the whiteboard edges.
- Also, the tsukuba original image has many tiny artifacts in it, making it very congested and noisy when we try to produce an information patch.
- In poster image, for radius 5; we can see 3 poster outlines, and as poster are supposed to be flat regions, most of the part is empty white, depicting very small R values.
- For radius 10 in poster image; fine edges fade into flat region. It only shows the upper corner of one poster which is highly contrasting from it's background poster.

3 Extensions

3.1 Extension 1: Transformed Color Spaces for Demosaicing

3.1.1 Divide Red and Blue Filter Values by Interpolated Green Channel

According to this algorithm, we first get interpolation of the green channel, we used adaptive gradient for that. This process is same as shown in the section of problem 1:

Matlab Implementation for getting interpolated green channel*

```
% We will use Adaptive Gradient for Green Channel
    Interpolation
mosim = demosaicAdagrad(im);
greenChannel = mosim(:,:,2);
```

*Full code in Appendix C.1

Then, we will get the redValues and blueValues acquired from the filter and divide them both with the interpolated green channel [$R \leftarrow R/G$ and $B \leftarrow B/G$]:

Matlab Implementation for dividing red and blue filter values with interpolated green channel*

```
% Get Red Filter Values
redValues = zeros(imageWidth, imageHeight);
redValues(1:2:imageWidth, 1:2:imageHeight) = im(1:2:
    imageWidth, 1:2:imageHeight);
%Divide Red Filter values with Interpolated Green
    Channel
redValues = redValues ./ greenChannel;

%Get Blue Filter Values
blueValues = zeros(imageWidth, imageHeight);
blueValues(2:2:imageWidth, 2:2:imageHeight) = im(2:2:
    imageWidth, 2:2:imageHeight);
```

```
%Divide Blue Filter values with Interpolated Green
    Channel
blueValues = blueValues ./ greenChannel;
```

*Full code in Appendix C.1

We merge new red and blue filter values with the same green *filter values* [note: it's not interpolated values] to get a whole new RGB filter array.

We then use adaptive interpolation technique on RGB filter array and extract red and blue interpolated channels.

Then we can multiply the interpolated red and blue channels with interpolated green channel to negate the influence of green channel division.

The final red, blue and green channels will give us the interpolated RGB image.

Matlab Implementation for interpolating updated red and blue filter values and then negating the green channel division to get the output*

```
% Create the new filter with updated red and blue
    values
im_new = redValues + greenValues + blueValues;

% Get the interpolated RGB Channels
mosim = demosaicAdagrad(im_new);

% Multiply Red Filter values with Interpolated Green
    Channel
mosim(:,:,1) = mosim(:,:,1) .* greenChannel;
% Multiply Blue Filter values with Interpolated Green
    Channel
mosim(:,:,3) = mosim(:,:,3) .* greenChannel;
```

*Full code in Appendix C.1

#	image	divide_green
1	balloon.jpeg	0.007239
2	cat.jpg	0.008356
3	ip.jpg	0.010864
4	puppy.jpg	0.003411
5	squirrel.jpg	0.007415
6	pencils.jpg	0.010486
7	house.png	0.008702
8	light.png	0.011325
9	sails.png	0.008145
10	tree.jpeg	0.009961
	average	0.008590

Table 2: Output image error for “divide red and blue filtered values by interpolated green channel” interpolation

3.1.2 Results

See Table 2 and Figures 30
This shows significant improvements over previous methods [baseline: 0.136659, nearest neighbors: 0.026375, linear: 0.015103, adaptive - 1: 0.014391, adaptive - 2: 0.014735].

Figure 30: All image outputs for the division by green algorithm

3.1.3 Log of Division of Red and Blue Filter Values by Interpolated Green Channel

This method works exact same way as the only division method, with only few changes:
We will take log of the ratio, e.g., $R \leftarrow log(R/G)$ and $B \leftarrow log(B/G)$.

But this causes an issue, $log(0) = notdefined$ and $3/4^{th}$ of the values in both red and blue filters will be 0.0

To avoid this, we add one to the ratio before taking log.

$$\therefore R \leftarrow log(R/G + 1) and B \leftarrow log(B/G + 1)$$

$log(x)$ where x < 1.0 will be in negative. But we need our values from 0.0 to 1.0. Thus, we use `rescale` function

Matlab Implementation for log-dividing red and blue filter values with interpolated green channel*

```
% Get Red Filter Values
redValues = zeros(imageWidth, imageHeight);
redValues(1:2:imageWidth, 1:2:imageHeight) = im(1:2:
    imageWidth, 1:2:imageHeight);
% Divide Red Filter values with Interpolated Green
    Channel; add 1 and
% take log of it so log(0) situation doesn't occur
redValues = log(redValues ./ greenChannel + 1);
redValues = rescale(redValues);

% Get Blue Filter Values
blueValues = zeros(imageWidth, imageHeight);
blueValues(2:2:imageWidth, 2:2:imageHeight) = im(2:2:
    imageWidth, 2:2:imageHeight);
% Divide Blue Filter values with Interpolated Green
    Channel; add 1 and
% take log of it so log(0) situation doesn't occur
blueValues = log(blueValues ./ greenChannel + 1);
blueValues = rescale(blueValues);
```

*Full code in Appendix C.2

For the negation of log and division to get the final output, we take exponential of the interpolated channel which are multiplied with interpolated green channel values and subtract 1 from it to negate the +1:

$$exp(log(x+1)) - 1 = x + 1 - 1 = x$$

Matlab Implementation for interpolating updated red and blue filter values and then negating the green channel log-division to get the output*

```
% Create the new filter with updated red and blue
    values
im_new = redValues + greenValues + blueValues;

% Get the interpolated RGB Channels
mosim = demosaicAdagrad(im_new);

% Multiply Red Filter values with Interpolated Green
    Channel; take
% exponential to even out the log effect and add -1
    because we had
% added +1 before taking log of the ratio

% exp(log(x + 1)) - 1 = x + 1 - 1 = x
mosim(:,:,1) = exp(mosim(:,:,1) .* greenChannel ) -1;
% Multiply Blue Filter values with Interpolated Green
    Channel
mosim(:,:,3) = exp(mosim(:,:,3) .* greenChannel ) -1;
```

*Full code in Appendix C.2

#	image	log_divide_green
1	balloon.jpeg	0.024953
2	cat.jpg	0.071324
3	ip.jpg	0.051392
4	puppy.jpg	0.045464
5	squirrel.jpg	0.025644
6	pencils.jpg	0.068421
7	house.png	0.050330
8	light.png	0.050980
9	sails.png	0.052192
10	tree.jpeg	0.015367
	average	0.045606

Table 3: Output image error for "log of division of red and blue filtered values by interpolated green channel" interpolation

3.1.4 Results

See Table 3 and Figures 31

- This method only performed better than baseline method, which is not much of an achievement.
- Taking $log(x)$ where $x < 1.0$ results in negative values and to make them be in $[0.0, 1.0]$, we are normalize the whole channel again.
- That can lead to misrepresentation of the relative value differences between neighbors.
- Which might lead to drastic change in interpolated values from the original ones.
- ONLY DIVISION METHOD performed the best because we were dividing all by a constant, it was still LINEAR.
- But with LOG DIVISION METHOD, the linearity is lost and thus, the approximation falls far off.

Figure 31: All image outputs for the log of division by green algorithm

3.2 Extension 2: Alternative Sampling Patterns

3.2.1 Simplifications

There are some assumptions we make to simplify the problem of finding an alternative for filter patterns:

- We permute three colors Red Green and Blue in 2 x 2 grid, which will be replicated for the whole filter image.
- We use linear interpolation, which is just average of neighbors, no special logic needed, we can use masks and swipe arounf all 8 neighbors to find out which pixels are really the neighbors in any kind of pattern.
- No pan-chromatic/ white cells. Which will deal away with keeping a cell for more than one color filter in consideration. This way, patterns are fixed and do not change because of a cell having potential of holding multiple color values.

3.2.2 Brute-Force Approach

In this approach, we just take 4 nested loops, and go through all permutations of R G and B in 4 cells of 2 x 2 grid. '1' is for Red Color, '2' is for Green Color and '3' is for Blue Color.

For each permutation, we replicate that 2 x 2 grid to the test image dimension, then we need to create the masks according to the pattern for each image and the rest is same as Linear Interpolation we discussed in Problem 1.

See full code at Appendix C.3.

3.2.3 Good Patterns

We display top 8 least error and bottom 4 most error-prone patterns in Table 4

We see from Table 4 that the best patterns are where green cells are double the amount of both red and blue cells individually.

And obviously, the worst patterns are where inly one color filter is

Rank	Pattern	Error
1	[G, G; R, B]	0.372335
2	[G, G; B, R]	0.372418
3	[G, B; G, R]	0.372491
4	[G, R; G, B]	0.372631
5	[R, B; G, G]	0.372695
6	[B, R; G, G]	0.372820
7	[B, G; R, G]	0.372959
8	[R, G; B, G]	0.373056
78	[R, B; R, B]	0.875837
79	[G, G; G, G]	0.947724
80	[R, R; R, R]	0.985257
81	[B, B; B, B]	1.022524

Table 4: Top 8 and Bottom 4 results for 81 brute forced filter patterns with linear interpolation

used, even there, green only filter is better than Red or Blue only.

A filter without green channel doesn't give good results at all.

3.2.4 Overall Errors and Concluding Remarks

Even though the error rate for best pattern according to our brute force search is much higher, the conclusions are pretty sturdy:
Green contributes double the amount of red or blue, so it is advised to have green cells equal to the sum of red and blue cells.

Red:Blue:Green in 1:1:2 ratio works the best, with all their different permutations.

Red:Blue:Green in -:1:- ratio works the worst.

References

[1] apertus° wiki *OpenCine.Nearest Neighbor and Bilinear Interpolation* https://wiki.apertus.org/index.php/OpenCine.Nearest_Neighbor_and_Bilinear_Interpolation

[2] Subhransu Maji, *Optical Flow* https://www.dropbox.com/s/9y5gkgycho3iqd7/lec07_optical_flow.pdf?dl=0

[3] Richard Szeliski, *Computer Vision: Algorithms and Applications* https://www.dropbox.com/s/9y5gkgycho3iqd7/lec07_optical_flow.pdf?dl=0 ISBN: 978-1-848-82934-3

[4] A. Ortis, F. Rundo, G. Di Giore, S. Battiato; *Adaptive Compression of Stereoscopic Images* https://link.springer.com/chapter/10.1007/978-3-642-41181-6_40 ISBN: 978-3-642-41181-6

[5] A. Ortis, F. Rundo, G. Di Giore, S. Battiato; *Sørensen–Dice coefficient* https://en.wikipedia.org/wiki/SÃÿrensenâĂŞDice_coefficient

A

Matlab code for Problem 1: Color Image Demosaicing

A.1 Implementation of `demosaicImage.m`

```
function output = demosaicImage(im, method)
% DEMOSAICIMAGE computes the color image from mosaiced input
%   OUTPUT = DEMOSAICIMAGE(IM, METHOD) computes a demosaiced OUTPUT from
%   the input IM. The choice of the interpolation METHOD can be
%   'baseline', 'nn', 'linear', 'adagrad'.
%
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%

switch lower(method)
    case 'baseline'
        output = demosaicBaseline(im);
    case 'nn'
        output = demosaicNN(im);          % Implement this
    case 'linear'
        output = demosaicLinear(im);      % Implement this
    case 'adagrad'
        output = demosaicAdagrad(im);     % Implement this
    case 'divide'
        output = demosaicDivide(im);
    case 'logdivide'
        output = demosaicLogDivide(im);
end
```

```
28 end
29
30 %--------------------------------------------------------------
31 % Baseline demosacing algorithm.
32 % The algorithm replaces missing values with the
33 %  mean of each color channel.
34 %--------------------------------------------------------------
35 function mosim = demosaicBaseline(im)
36 mosim = repmat(im, [1 1 3]); % Create an image by
      stacking the input
37 [imageHeight, imageWidth] = size(im);
38
39 % Red channel (odd rows and columns);
40 redValues = im(1:2:imageHeight, 1:2:imageWidth);
41 meanValue = mean(mean(redValues));
42 mosim(:,:,1) = meanValue;
43 mosim(1:2:imageHeight, 1:2:imageWidth,1) = im(1:2:
      imageHeight, 1:2:imageWidth);
44
45 % Blue channel (even rows and colums);
46 blueValues = im(2:2:imageHeight, 2:2:imageWidth);
47 meanValue = mean(mean(blueValues));
48 mosim(:,:,3) = meanValue;
49 mosim(2:2:imageHeight, 2:2:imageWidth,3) = im(2:2:
      imageHeight, 2:2:imageWidth);
50
51 % Green channel (remaining places)
52 % We will first create a mask for the green pixels (+1
       green, -1 not green)
53 mask = ones(imageHeight, imageWidth);
54 mask(1:2:imageHeight, 1:2:imageWidth) = -1;
55 mask(2:2:imageHeight, 2:2:imageWidth) = -1;
56 greenValues = mosim(mask > 0);
57 meanValue = mean(greenValues);
58 % For the green pixels we copy the value
59 greenChannel = im;
60 greenChannel(mask < 0) = meanValue;
61 mosim(:,:,2) = greenChannel;
62 end
63
64 %--------------------------------------------------------------
```

```
65 % Nearest neighbour algorithm
66 %-------------------------------------------------------------
67 function mosim = demosaicNN(im)
68 %
69 % Implement this
70 %
71 mosim = repmat(im, [1 1 3]); % Create an image by
       stacking the input
72 [imageHeight, imageWidth] = size(im); % Got size of
       the image
73
74 %% Red channel (odd rows and columns);
75
76 % Initialize the resultant red channel to all the -1
       values
77 redValues = ones(imageHeight, imageWidth)*-1;
78 % Put the already known red values to the resultant
       image corresponding pixels
79 redValues(1:2:imageHeight, 1:2:imageWidth) = im(1:2:
       imageHeight, 1:2:imageWidth);
80
81 % For every alternate row, starting from 1st row
82 for i = 1:2:imageHeight
83     % For every alternate column, starting from 1st
          column
84     for j = 1:2:imageWidth
85         % Put the same value as that of known pixel's,
              to the pixel at right
86         if( i + 1 <= imageHeight)
87             redValues(i+1, j) = redValues(i, j);
88         end
89         % Put the same value as that of known pixel's,
              to the pixel at bottom
90         if( j + 1 <= imageWidth)
91             redValues(i, j+1) = redValues(i, j);
92         end
93         % Put the same value as that of known pixel's,
              to the pixel at bottom right
94         if( j + 1 <= imageWidth && i + 1 <=
              imageHeight)
95             redValues(i+1, j+1) = redValues(i, j);
```

```
96          end
97      end
98  end
99
100 %mosim(:, :, 1) = padarray(redValues, [1, 1], '
    replicate', 'post');
101 % Make the interpolated red value matrix, the red
    channel of the final output image
102 mosim(:, :, 1) = redValues;
103
104 %% Blue channel (even rows and columns);
105
106 % Initialize the resultant blue channel to all the -1
    values
107 blueValues = ones(imageHeight, imageWidth)*-1;
108
109 % Put the already known blue values to the resultant
    image corresponding pixels
110 blueValues(2:2:imageHeight, 2:2:imageWidth) = im(2:2:
    imageHeight, 2:2:imageWidth);
111
112 % For every alternate row, starting from 2nd row
113 for i = 2:2:imageHeight
114     % For every alternate column, starting from 2nd
        column
115     for j = 2:2:imageWidth
116         % Put the same value as that of known pixel's,
              to the pixel at right
117         if( i + 1 <= imageHeight)
118             blueValues(i+1, j) = blueValues(i, j);
119         end
120         % Put the same value as that of known pixel's,
              to the pixel at bottom
121         if( j + 1 <= imageWidth)
122             blueValues(i, j+1) = blueValues(i, j);
123         end
124         % Put the same value as that of known pixel's,
              to the pixel at bottom right
125         if( j + 1 <= imageWidth && i + 1 <=
            imageHeight)
126             blueValues(i+1, j+1) = blueValues(i, j);
```

```
127             end
128         end
129     end
130
131     % For the first row, copy the values from 2nd row,
            they are the "nearest"
132     blueValues(1,:) = blueValues(2,:);
133     % For the first column, copy the values from 2nd
            column, they are the "nearest"
134     blueValues(:,1) = blueValues(:,2);
135     % Make the interpolated blue value matrix, the blue
            channel of the final output image
136     mosim(:, :, 3) = blueValues;
137
138     %% Green channel
139
140     % Initialize the resultant green channel to all the -1
            values
141     greenValues = ones(imageHeight, imageWidth)*-1;
142
143     % We put the already known green values to the
            resultant image corresponding pixels
144     % Odd rows have green pixel's known values at even
            columns
145     greenValues(1:2:imageHeight, 2:2:imageWidth) = im(1:2:
            imageHeight, 2:2:imageWidth);
146     % Even rows have green pixel's known values at odd
            columns
147     greenValues(2:2:imageHeight, 1:2:imageWidth) = im(2:2:
            imageHeight, 1:2:imageWidth);
148
149     % Odd rows
150     for i = 1:2:imageHeight
151         % Even columns
152         for j = 2:2:imageWidth
153             % Put the same value as that of known pixel's,
                    to the pixel at left
154             if (i - 1 > 0)
155                 greenValues(i - 1, j) = greenValues(i, j);
156             end
157         end
```

```
158 end
159
160 % Even rows
161 for i = 2:2:imageHeight
162     % Odd columns
163     for j = 1:2:imageWidth
164         % Put the same value as that of known pixel's,
              to the pixel at right
165         if (i + 1 <= imageHeight)
166             greenValues(i + 1, j) = greenValues(i, j);
167         end
168     end
169 end
170
171 % Make the interpolated green value matrix, the green
    channel of the final output image
172 mosim(:, :, 2) = greenValues;
173
174 end
175
176 %--------------------------------------------------------
177 % Linear interpolation
178 %--------------------------------------------------------
179 function mosim = demosaicLinear(im)
180 %
181 % Implement this
182 %
183 mosim = repmat(im, [1 1 3]); % Create an image by
    stacking the input
184 [imageWidth, imageHeight] = size(im);
185
186 %% Red channel (odd rows and columns);
187 % Initialize the resultant red channel to all the -1
    values
188 redValues = ones(imageWidth, imageHeight)*-1;
189 % Create a mask which shows which pixels have known
    values and which are to be interpolated
190 redMask = zeros(imageWidth, imageHeight);
191 % Put the already known red values to the resultant
    image corresponding pixels
192 redValues(1:2:imageWidth, 1:2:imageHeight) = im(1:2:
```

```
    imageWidth, 1:2:imageHeight);
% Put mask value to 1/true where image pixels are already have known values
redMask(1:2:imageWidth, 1:2:imageHeight) = 1;

for i = 1:imageWidth
    for j = 1:imageHeight
        % If a pixel has its value yet to be interpolated
        if redValues(i, j) == -1
            count = 0;
            sum = 0;
            % Here we will just check all the 8 neighboring pixels. It will
            % take care of all the cases (diagonal, horizontal and
            % vertical) as only the redMask = 1 values will hold any
            % weight. Others will be simply 0 and won't contribute if a
            % case doesn't apply for them

            % Checking for 3 neighboring pixels at left side
            if (i - 1 > 0)
                sum = sum + redValues(i-1, j)*redMask(i-1, j);
                count = count + 1*redMask(i-1, j);
                if (j + 1 <= imageHeight)
                    sum = sum + redValues(i-1, j+1)*redMask(i-1, j+1);
                    count = count + 1*redMask(i-1, j+1);
                end
                if (j - 1 > 0)
                    sum = sum + redValues(i-1, j-1)*redMask(i-1, j-1);
                    count = count + 1*redMask(i-1, j-1);
                end
            end
```

```
221
222             % Checking for 3 neighboring pixels at
                    right side
223             if (i + 1 <= imageWidth)
224                 sum = sum + redValues(i+1, j)*redMask(
                        i+1, j);
225                 count = count + 1*redMask(i+1, j);
226                 if (j + 1 <= imageHeight)
227                     sum = sum + redValues(i+1, j+1)*
                            redMask(i+1, j+1);
228                     count = count + 1*redMask(i+1, j
                            +1);
229                 end
230                 if (j - 1 > 0)
231                     sum = sum + redValues(i+1, j-1)*
                            redMask(i+1, j-1);
232                     count = count + 1*redMask(i+1, j
                            -1);
233                 end
234             end
235
236             % Checking for neighboring pixel at bottom
237             if (j + 1 <= imageHeight)
238                 sum = sum + redValues(i, j+1)*redMask(
                        i, j+1);
239                 count = count + 1*redMask(i, j+1);
240             end
241
242             % Checking for neighboring pixel at top
243             if (j - 1 > 0)
244                 sum = sum + redValues(i, j-1)*redMask(
                        i, j-1);
245                 count = count + 1*redMask(i, j-1);
246             end
247
248             % Take the average of the sum of the
                    values of neighboring pixels which have
                    known
249             % values
250             redValues(i, j) = sum/count;
251         end
```

```
252     end
253 end
254 % Make the interpolated red value matrix, the red
    channel of the final output image
255 mosim(:, :, 1) = redValues;
256
257 %% Blue channel (even rows and columns);
258 % Initialize the resultant blue channel to all the -1
    values
259 blueValues = ones(imageWidth, imageHeight)*-1;
260 % Create a mask which shows which pixels have known
    values and which are to be interpolated
261 blueMask = zeros(imageWidth, imageHeight);
262 % Put the already known blue values to the resultant
    image corresponding pixels
263 blueValues(2:2:imageWidth, 2:2:imageHeight) = im(2:2:
    imageWidth, 2:2:imageHeight);
264 % Put mask value to 1/true where image pixels are
    already have known values
265 blueMask(2:2:imageWidth, 2:2:imageHeight) = 1;
266
267 for i = 1:imageWidth
268     for j = 1:imageHeight
269         % If a pixel has its value yet to be
            interpolated
270         if blueValues(i, j) == -1
271             count = 0;
272             sum = 0;
273             % Here we will just check all the 8
                neighboring pixels. It will
274             % take care of all the cases (diagonal,
                horizontal and
275             % vertical) as only the redMask = 1 values
                 will hold any
276             % weight. Others will be simply 0 and won'
                t contribute if a
277             % case doesn't apply for them
278
279             % Checking for 3 neighboring pixels at
                left side
280             if (i - 1 > 0)
```

```
281                 sum = sum + blueValues(i-1, j)*
                        blueMask(i-1, j);
282                 count = count + 1*blueMask(i-1, j);
283                 if (j + 1 <= imageHeight)
284                     sum = sum + blueValues(i-1, j+1)*
                            blueMask(i-1, j+1);
285                     count = count + 1*blueMask(i-1, j
                            +1);
286                 end
287                 if (j - 1 > 0)
288                     sum = sum + blueValues(i-1, j-1)*
                            blueMask(i-1, j-1);
289                     count = count + 1*blueMask(i-1, j
                            -1);
290                 end
291             end
292
293             % Checking for 3 neighboring pixels at
                    right side
294             if (i + 1 <= imageWidth)
295                 sum = sum + blueValues(i+1, j)*
                        blueMask(i+1, j);
296                 count = count + 1*blueMask(i+1, j);
297                 if (j + 1 <= imageHeight)
298                     sum = sum + blueValues(i+1, j+1)*
                            blueMask(i+1, j+1);
299                     count = count + 1*blueMask(i+1, j
                            +1);
300                 end
301                 if (j - 1 > 0)
302                     sum = sum + blueValues(i+1, j-1)*
                            blueMask(i+1, j-1);
303                     count = count + 1*blueMask(i+1, j
                            -1);
304                 end
305             end
306
307             % Checking for neighboring pixel at bottom
308             if (j + 1 <= imageHeight)
309                 sum = sum + blueValues(i, j+1)*
                        blueMask(i, j+1);
```

```
310                     count = count + 1*blueMask(i, j+1);
311                 end
312
313                 % Checking for neighboring pixel at top
314                 if (j - 1 > 0)
315                     sum = sum + blueValues(i, j-1)*
                           blueMask(i, j-1);
316                     count = count + 1*blueMask(i, j-1);
317                 end
318
319                 % Take the average of the sum of the
                       values of neighboring pixels which have
                        known values
320                 blueValues(i, j) = sum/count;
321             end
322         end
323     end
324     % Make the interpolated blue value matrix, the blue
          channel of the final output image
325     mosim(:, :, 3) = blueValues;
326
327     %% Green channel
328     % Initialize the resultant green channel to all the -1
          values
329     greenValues = ones(imageWidth, imageHeight)*-1;
330     % Create a mask which shows which pixels have known
          values and which are to be interpolated
331     greenMask = zeros(imageWidth, imageHeight);
332     greenValues(1:2:imageWidth, 2:2:imageHeight) = im(1:2:
          imageWidth, 2:2:imageHeight);
333     greenMask(1:2:imageWidth, 2:2:imageHeight) = 1;
334     greenValues(2:2:imageWidth, 1:2:imageHeight) = im(2:2:
          imageWidth, 1:2:imageHeight);
335     greenMask(2:2:imageWidth, 1:2:imageHeight) = 1;
336
337     for i = 1:imageWidth
338         for j = 1:imageHeight
339             % If a pixel has its value yet to be
                  interpolated
340             if greenValues(i, j) == -1
341                 count = 0;
```

```
342                 sum = 0;
343                 % Here we will just check only 4
                        neighboring pixels [top,
344                 % bottom, left and right]. Becasuse at any
                        unknown valued pixel
345                 % location, there will be the same pattern
                        observed, which look
346                 % like as follows in the green mask:
347                 % 0 1 0
348                 % 1 0 1
349                 % 0 1 0
350
351                 % Averaging all 4 neighboring pixels
352                 if (i - 1 > 0)
353                     sum = sum + greenValues(i-1, j)*
                            greenMask(i-1, j);
354                     count = count + 1*greenMask(i-1, j);
355                 end
356                 if (i + 1 <= imageWidth)
357                     sum = sum + greenValues(i+1, j)*
                            greenMask(i+1, j);
358                     count = count + 1*greenMask(i+1, j);
359                 end
360                 if (j + 1 <= imageHeight)
361                     sum = sum + greenValues(i, j+1)*
                            greenMask(i, j+1);
362                     count = count + 1*greenMask(i, j+1);
363                 end
364                 if (j - 1 > 0)
365                     sum = sum + greenValues(i, j-1)*
                            greenMask(i, j-1);
366                     count = count + 1*greenMask(i, j-1);
367                 end
368
369                 % Take the average of the sum of the
                        values of neighboring pixels which have
                        known values
370                 greenValues(i, j) = sum/count;
371             end
372         end
373     end
```

```
374 % Make the interpolated green value matrix, the green
      channel of the final output image
375 mosim(:, :, 2) = greenValues;
376
377 end
378
379 %--------------------------------------------------------
380 % Adaptive gradient
381 %--------------------------------------------------------
382 function mosim = demosaicAdagrad(im)
383 %
384 % Implement this
385 %
386 [imageWidth, imageHeight] = size(im);
387 % Linear Interpolation of Red and Blue Channels
388 mosim = demosaicLinear(im);
389
390 %% Green channel
391 % Initialize the resultant green channel to all the -1
      values
392 greenValues = ones(imageWidth, imageHeight)*-1;
393 % Create a mask which shows which pixels have known
      values and which are to be interpolated
394 greenMask = zeros(imageWidth, imageHeight);
395 greenValues(1:2:imageWidth, 2:2:imageHeight) = im(1:2:
      imageWidth, 2:2:imageHeight);
396 greenMask(1:2:imageWidth, 2:2:imageHeight) = 1;
397 greenValues(2:2:imageWidth, 1:2:imageHeight) = im(2:2:
      imageWidth, 1:2:imageHeight);
398 greenMask(2:2:imageWidth, 1:2:imageHeight) = 1;
399
400 for i = 1:imageWidth
401     for j = 1:imageHeight
402         if greenValues(i, j) == -1
403             sum_h = 0; abs_h = 0; sum_v = 0; abs_v =
                  0;
404
405             % Finding left and right pixel values
                  difference and sum for core, edge and
                  corner cases
406             if (i - 1 > 0 && i + 1 <= imageWidth)
```

```
407             abs_h = abs(greenValues(i-1, j)*
                    greenMask(i-1, j) - greenValues(i
                    +1, j)*greenMask(i+1, j));
408             sum_h = greenValues(i-1, j)*greenMask(
                    i-1, j) + greenValues(i+1, j)*
                    greenMask(i+1, j);
409             sum_h = sum_h/2;
410         elseif i - 1 > 0
411             abs_h = greenValues(i-1, j)*greenMask(
                    i-1, j);
412             sum_h = greenValues(i-1, j)*greenMask(
                    i-1, j);
413         elseif i + 1 <= imageWidth
414             abs_h = greenValues(i+1, j)*greenMask(
                    i+1, j);
415             sum_h = greenValues(i+1, j)*greenMask(
                    i+1, j);
416         end
417
418         % Finding top and bottom pixel values
                difference and sum for core, edge and
                corner cases
419         if (j - 1 > 0 && j + 1 <= imageHeight)
420             abs_v = abs(greenValues(i, j+1)*
                    greenMask(i, j+1) - greenValues(i,
                    j-1)*greenMask(i, j-1));
421             sum_v = greenValues(i, j+1)*greenMask(
                    i, j+1) + greenValues(i, j-1)*
                    greenMask(i, j-1);
422             sum_v = sum_v/2;
423         elseif j - 1 > 0
424             abs_v = greenValues(i, j-1)*greenMask(
                    i, j-1);
425             sum_v = greenValues(i, j-1)*greenMask(
                    i, j-1);
426         elseif j + 1 <= imageHeight
427             abs_v = greenValues(i, j+1)*greenMask(
                    i, j+1);
428             sum_v = greenValues(i, j+1)*greenMask(
                    i, j+1);
429         end
```

```
430
431                 % Choose which value to adapt
432                 if abs_v > abs_h
433                     greenValues(i, j) = sum_h;
434                 else
435                     greenValues(i, j) = sum_v;
436                 end
437             end
438         end
439     end
440     % Make the interpolated green value matrix, the green
          channel of the final output image
441     mosim(:, :, 2) = greenValues;
442     end
443
444     %-----------------------------------------------------------
445     % Tranformed Color Space Divide
446     %-----------------------------------------------------------
447     function mosim = demosaicDivide(im)
448         % Get dimensions of the filter values
449         [imageWidth, imageHeight] = size(im);
450
451         % We will use Adaptive Gradient for Green Channel
              Interpolation
452         mosim = demosaicAdagrad(im);
453         greenChannel = mosim(:,:,2);
454
455         % Get Red Filter Values
456         redValues = zeros(imageWidth, imageHeight);
457         redValues(1:2:imageWidth, 1:2:imageHeight) = im
              (1:2:imageWidth, 1:2:imageHeight);
458         % Divide Red Filter values with Interpolated Green
               Channel
459         redValues = redValues ./ greenChannel;
460
461         % Get Blue Filter Values
462         blueValues = zeros(imageWidth, imageHeight);
463         blueValues(2:2:imageWidth, 2:2:imageHeight) = im
              (2:2:imageWidth, 2:2:imageHeight);
464         % Divide Blue Filter values with Interpolated
              Green Channel
```

```
465         blueValues = blueValues ./ greenChannel;
466
467         % We will get Green Filter only values from the
                filter again, to make a new filter
468         greenValues = zeros(imageWidth, imageHeight);
469         greenValues(1:2:imageWidth, 2:2:imageHeight) = im
                (1:2:imageWidth, 2:2:imageHeight);
470         greenValues(2:2:imageWidth, 1:2:imageHeight) = im
                (2:2:imageWidth, 1:2:imageHeight);
471
472         % Create the new filter with updated red and blue
                values
473         im_new = redValues + greenValues + blueValues;
474
475         % Get the interpolated RGB Channels
476         mosim = demosaicAdagrad(im_new);
477
478         % Multiply Red Filter values with Interpolated
                Green Channel
479         mosim(:,:,1) = mosim(:,:,1) .* greenChannel;
480         % Multiply Blue Filter values with Interpolated
                Green Channel
481         mosim(:,:,3) = mosim(:,:,3) .* greenChannel;
482
483     end
484
485     %-------------------------------------------------------
486     % Tranformed Color Space Divide and Log
487     %-------------------------------------------------------
488
489     function mosim = demosaicLogDivide(im)
490         % Get dimensions of the filter values
491         [imageWidth, imageHeight] = size(im);
492
493         % We will use Adaptive Gradient for Green Channel
                Interpolation
494         mosim = demosaicAdagrad(im);
495         greenChannel = mosim(:,:,2);
496
497         % Get Red Filter Values
498         redValues = zeros(imageWidth, imageHeight);
```

```
499     redValues(1:2:imageWidth, 1:2:imageHeight) = im
            (1:2:imageWidth, 1:2:imageHeight);
500     % Divide Red Filter values with Interpolated Green
            Channel; add 1 and
501     % take log of it so log(0) situation doesn't occur
502     redValues = log(redValues ./ greenChannel + 1);
503
504     % Get Blue Filter Values
505     blueValues = zeros(imageWidth, imageHeight);
506     blueValues(2:2:imageWidth, 2:2:imageHeight) = im
            (2:2:imageWidth, 2:2:imageHeight);
507     % Divide Blue Filter values with Interpolated
            Green Channel; add 1 and
508     % take log of it so log(0) situation doesn't occur
509     blueValues = log(blueValues ./ greenChannel + 1);
510
511     % We will get Green Filter only values from the
            filter again, to make a new filter
512     greenValues = zeros(imageWidth, imageHeight);
513     greenValues(1:2:imageWidth, 2:2:imageHeight) = im
            (1:2:imageWidth, 2:2:imageHeight);
514     greenValues(2:2:imageWidth, 1:2:imageHeight) = im
            (2:2:imageWidth, 1:2:imageHeight);
515
516     % Create the new filter with updated red and blue
            values
517     im_new = redValues + greenValues + blueValues;
518
519     % Get the interpolated RGB Channels
520     mosim = demosaicAdagrad(im_new);
521
522     % Multiply Red Filter values with Interpolated
            Green Channel; take
523     % exponential to even out the log effect and add
            -1 because we had
524     % added +1 before taking log of the ratio
525
526     % exp(log(x + 1)) - 1 = x + 1 - 1 = x
527     mosim(:,:,1) = exp(mosim(:,:,1) .* greenChannel )
            -1;
528     % Multiply Blue Filter values with Interpolated
```

```
        Green Channel
529     mosim(:,:,3) = exp(mosim(:,:,3) .* greenChannel )
        -1;
530
531 end
```

B Matlab code for Problem 2: Depth from Disparity

B.1 Implementation of `depthFromStereo.m`

```
1  function depth = depthFromStereo(img1, img2, ws)
2  % Implement this
3
4  % We take mean of all three channels of the first and
       second images to get a grayscale approximation of
       the images
5  img1_m = mean(img1, 3);
6  img2_m = mean(img2, 3);
7  % Initialize depth mean to all zeros
8  depth = zeros(size(img1_m, 1), size(img1_m, 2), 'uint8
       ');
9
10 % Keep the search within this range away from boundary
11 range = 50;
12 % Patch Box size
13 patch_size = 5;
14
15 [row, col] = size(img1_m);
16
17 % Traverse through all Epipolar Lines
18 for i = 1:row
19
20     % Boundary and Index range checks
21     row_min = max(1, i - patch_size);
22     row_max = min(row, i + patch_size);
23
24     % For each pixel on epipolar lines
25     for j = 1:col
```

```
26
27          % Set the patch sizes according to boudary
               ranges
28          col_min = max(1, j - patch_size);
29          col_max = min(col, j + patch_size);
30
31          % Patch box dimensions/ locations
32          pix_min = max(-range, 1 - col_min);
33          pix_max = min(range, col - col_max);
34
35          % Temporary Patch Box
36          t = img2_m(row_min:row_max, col_min:col_max);
37
38          % An array storing the SSD (Sum of Squared
               Difference)
39          count = pix_max - pix_min + 1;
40          diff = zeros(count, 1);
41
42          % Match the image patches on an epipolar line
43          for k = pix_min : pix_max
44              block = img1_m(row_min:row_max, (col_min+k
                   ):(col_max+k));
45              index = k - pix_min + 1;
46              diff(index, 1) = sumsqr(t - block);
47          end
48
49          % Get the pixel location where the patch
               disparity is the smallest
50          [x, y] = sort(diff);
51          m_index = y(1, 1);
52          disparity = m_index + pix_min - 1;
53          % Calculate the depth for that specific pixel
54          depth(i, j) = 1/disparity;
55          depth = imadjust(depth);
56      end
57  end
```

B.2 Implementation of `visualizeInformation.m`

```
1  function R = visualizeInformation(im)
2      %im = imread('../data/disparity/tsukuba_im1.jpg');
```

```
3       im = imread('../data/disparity/poster_im2.jpg');
4       [width, height, c] = size(im);
5       % Converting image to grayscale because color
            channels don't matter for second-moment matrix,
            just the corresponding relative pixel values.
6       im = rgb2gray(im);
7       % Radius
8       r = 5;
9       % Informative Patch
10      R = zeros(width, height, 'uint8');
11
12      % For each pixel
13      for u = 1: width-r-1
14          for v = 1:height-r-1
15              % Computer Ixx, Ixy and Iyy for the given
                    neighbors in the window
16              Ixx = 0; Ixy = 0; Iyy = 0;
17              for i = 1:r
18                  for j = 1:r
19                      Ix = im(u + i + 1, v) - im(u + i, v
                            );
20                      Iy = im(u, v + i + 1) - im(u, v + i
                            );
21                      Ixx = Ixx + Ix*Ix;
22                      Ixy = Ixy + Ix*Iy;
23                      Iyy = Iyy + Iy*Iy;
24                  end
25              end
26              % Computer R
27              R(u, v) = Ixx*Iyy - 0.03*(Ixx + Iyy)*(Ixx
                    + Iyy);
28          end
29      end
30      figure;
31      subplot(121); imshow(im);
32      subplot(122); imshow(imcomplement(R)); colorbar;
33      figure;
34      imshow(imcomplement(R));
35  end
```

C
Matlab code for Extensions

C.1 Implementation of `demosaicDivide.m`

```
%-------------------------------------------------------------
% Tranformed Color Space Divide
%-------------------------------------------------------------
function mosim = demosaicDivide(im)
    % Get dimensions of the filter values
    [imageWidth, imageHeight] = size(im);

    % We will use Adaptive Gradient for Green Channel
       Interpolation
    mosim = demosaicAdagrad(im);
    greenChannel = mosim(:,:,2);

    % Get Red Filter Values
    redValues = zeros(imageWidth, imageHeight);
    redValues(1:2:imageWidth, 1:2:imageHeight) = im
       (1:2:imageWidth, 1:2:imageHeight);
    % Divide Red Filter values with Interpolated Green
       Channel
    redValues = redValues ./ greenChannel;

    % Get Blue Filter Values
    blueValues = zeros(imageWidth, imageHeight);
    blueValues(2:2:imageWidth, 2:2:imageHeight) = im
       (2:2:imageWidth, 2:2:imageHeight);
    % Divide Blue Filter values with Interpolated
       Green Channel
    blueValues = blueValues ./ greenChannel;

    % We will get Green Filter only values from the
       filter again, to make a new filter
    greenValues = zeros(imageWidth, imageHeight);
    greenValues(1:2:imageWidth, 2:2:imageHeight) = im
       (1:2:imageWidth, 2:2:imageHeight);
    greenValues(2:2:imageWidth, 1:2:imageHeight) = im
       (2:2:imageWidth, 1:2:imageHeight);
```

```
28
29      % Create the new filter with updated red and blue
            values
30      im_new = redValues + greenValues + blueValues;
31
32      % Get the interpolated RGB Channels
33      mosim = demosaicAdagrad(im_new);
34
35      % Multiply Red Filter values with Interpolated
            Green Channel
36      mosim(:,:,1) = mosim(:,:,1) .* greenChannel;
37      % Multiply Blue Filter values with Interpolated
            Green Channel
38      mosim(:,:,3) = mosim(:,:,3) .* greenChannel;
39
40  end
```

C.2 Implementation of `demosaicLogDivide.m`

```
1   %-------------------------------------------------------
2   % Tranformed Color Space Divide and Log
3   %-------------------------------------------------------
4
5   function mosim = demosaicLogDivide(im)
6       % Get dimensions of the filter values
7       [imageWidth, imageHeight] = size(im);
8
9       % We will use Adaptive Gradient for Green Channel
            Interpolation
10      mosim = demosaicAdagrad(im);
11      greenChannel = mosim(:,:,2);
12
13      % Get Red Filter Values
14      redValues = zeros(imageWidth, imageHeight);
15      redValues(1:2:imageWidth, 1:2:imageHeight) = im
            (1:2:imageWidth, 1:2:imageHeight);
16      % Divide Red Filter values with Interpolated Green
            Channel; add 1 and
17      % take log of it so log(0) situation doesn't occur
18      redValues = log(redValues ./ greenChannel + 1);
19
```

```
20      % Get Blue Filter Values
21      blueValues = zeros(imageWidth, imageHeight);
22      blueValues(2:2:imageWidth, 2:2:imageHeight) = im
           (2:2:imageWidth, 2:2:imageHeight);
23      % Divide Blue Filter values with Interpolated
           Green Channel; add 1 and
24      % take log of it so log(0) situation doesn't occur
25      blueValues = log(blueValues ./ greenChannel + 1);
26
27      % We will get Green Filter only values from the
           filter again, to make a new filter
28      greenValues = zeros(imageWidth, imageHeight);
29      greenValues(1:2:imageWidth, 2:2:imageHeight) = im
           (1:2:imageWidth, 2:2:imageHeight);
30      greenValues(2:2:imageWidth, 1:2:imageHeight) = im
           (2:2:imageWidth, 1:2:imageHeight);
31
32      % Create the new filter with updated red and blue
           values
33      im_new = redValues + greenValues + blueValues;
34
35      % Get the interpolated RGB Channels
36      mosim = demosaicAdagrad(im_new);
37
38      % Multiply Red Filter values with Interpolated
           Green Channel; take
39      % exponential to even out the log effect and add
           -1 because we had
40      % added +1 before taking log of the ratio
41
42      % exp(log(x + 1)) - 1 = x + 1 - 1 = x
43      mosim(:,:,1) = exp(mosim(:,:,1) .* greenChannel )
           -1;
44      % Multiply Blue Filter values with Interpolated
           Green Channel
45      mosim(:,:,3) = exp(mosim(:,:,3) .* greenChannel )
           -1;
46
47  end
```

C.3 Implementation of `demosaicBruteForce.m`

```
1  function error = demosaicBruteForce(im)
2
3  [imageWidth, imageHeight, channels] = size(im);
4  im = double(im);
5  c = 1;
6  error = zeros(81, 1);
7  % Try every permutation
8  for p = 1:3
9      for q = 1:3
10         for r = 1:3
11             for s = 1:3
12                 % 2 x 2 Grid
13                 temp = [p, q, r, s];
14                 % Create Masks, 1 = R, 2 = G, 3 = B
15                 imageMask = repmat(reshape(temp, [2,
                       2]), round(imageWidth/2), round(
                       imageHeight/2));
16                 redMask = imageMask==1;
17                 redMask = redMask>0;
18
19                 greenMask = imageMask==2;
20                 greenMask = greenMask>0;
21
22                 blueMask = imageMask==3;
23                 blueMask = blueMask>0;
24
25                 redValues = im(:,:,1);
26                 greenValues = im(:,:,2);
27                 blueValues = im(:,:,3);
28
29                 % Linear Interpolation
30                 for i = 1:imageWidth
31                     for j = 1:imageHeight
32                         if redMask(i, j) == 0
33                             count = 0;
34                             sum = 0;
35                             if (i - 1 > 0)
36                                 sum = sum + redValues(
                                       i-1, j)*uint8(
```

```
                redMask(i-1, j));
37              count = count + 1*
                    uint8(redMask(i-1,
                    j));
38              if (j + 1 <=
                    imageHeight)
39                  sum = sum +
                        redValues(i-1,
                        j+1)*uint8(
                        redMask(i-1, j
                        +1));
40                  count = count + 1*
                        uint8(redMask(i
                        -1, j+1));
41              end
42              if (j - 1 > 0)
43                  sum = sum +
                        redValues(i-1,
                        j-1)*uint8(
                        redMask(i-1, j
                        -1));
44                  count = count + 1*
                        uint8(redMask(i
                        -1, j-1));
45              end
46          end
47          if (i + 1 <= imageWidth)
48              sum = sum + redValues(
                    i+1, j)*uint8(
                    redMask(i+1, j));
49              count = count + 1*
                    uint8(redMask(i+1,
                    j));
50              if (j + 1 <=
                    imageHeight)
51                  sum = sum +
                        redValues(i+1,
                        j+1)*uint8(
                        redMask(i+1, j
                        +1));
52                  count = count + 1*
```

```
                                    uint8(redMask(i
                                    +1, j+1));
53                              end
54                              if (j - 1 > 0)
55                                  sum = sum +
                                    redValues(i+1,
                                    j-1)*uint8(
                                    redMask(i+1, j
                                    -1));
56                                  count = count + 1*
                                    uint8(redMask(i
                                    +1, j-1));
57                              end
58                          end
59                          if (j + 1 <= imageHeight)
60                              sum = sum + redValues(
                                i, j+1)*uint8(
                                redMask(i, j+1));
61                              count = count + 1*
                                uint8(redMask(i, j
                                +1));
62                          end
63                          if (j - 1 > 0)
64                              sum = sum + redValues(
                                i, j-1)*uint8(
                                redMask(i, j-1));
65                              count = count + 1*
                                uint8(redMask(i, j
                                -1));
66                          end
67
68                          redValues(i, j) = sum/
                            count;
69                  end
70              end
71          end
72          mosim(:, :, 1) = redValues;
73
74          for i = 1:imageWidth
75              for j = 1:imageHeight
76                  if blueMask(i, j) == 0
```

```
77                  count = 0;
78                  sum = 0;
79                  if (i - 1 > 0)
80                      sum = sum + blueValues
                            (i-1, j)*uint8(
                            blueMask(i-1, j));
81                      count = count + 1*
                            uint8(blueMask(i-1,
                             j));
82                      if (j + 1 <=
                            imageHeight)
83                          sum = sum +
                                blueValues(i-1,
                                 j+1)*uint8(
                                blueMask(i-1, j
                                +1));
84                          count = count + 1*
                                uint8(blueMask(
                                i-1, j+1));
85                      end
86                      if (j - 1 > 0)
87                          sum = sum +
                                blueValues(i-1,
                                 j-1)*uint8(
                                blueMask(i-1, j
                                -1));
88                          count = count + 1*
                                uint8(blueMask(
                                i-1, j-1));
89                      end
90                  end
91                  if (i + 1 <= imageWidth)
92                      %blueValues(i+1, j)
93                      %uint8(blueMask(i+1, j
                            ))
94                      %i + 1
95                      sum = sum + blueValues
                            (i+1, j)*uint8(
                            blueMask(i+1, j));
96                      count = count + 1*
                            uint8(blueMask(i+1,
```

```
                j));
97              if (j + 1 <=
                    imageHeight)
98                  sum = sum +
                        blueValues(i+1,
                         j+1)*uint8(
                        blueMask(i+1, j
                        +1));
99                  count = count + 1*
                        uint8(blueMask(
                        i+1, j+1));
100             end
101             if (j - 1 > 0)
102                 sum = sum +
                        blueValues(i+1,
                         j-1)*uint8(
                        blueMask(i+1, j
                        -1));
103                 count = count + 1*
                        uint8(blueMask(
                        i+1, j-1));
104             end
105         end
106         if (j + 1 <= imageHeight)
107             sum = sum + blueValues
                    (i, j+1)*uint8(
                    blueMask(i, j+1));
108             count = count + 1*
                    uint8(blueMask(i, j
                    +1));
109         end
110         if (j - 1 > 0)
111             sum = sum + blueValues
                    (i, j-1)*uint8(
                    blueMask(i, j-1));
112             count = count + 1*
                    uint8(blueMask(i, j
                    -1));
113         end
114
115         blueValues(i, j) = sum/
```

```
                    count;
116                 end
117             end
118         end
119         mosim(:, :, 3) = blueValues;
120
121         for i = 1:imageWidth
122             for j = 1:imageHeight
123                 if greenMask(i, j) == 0
124                     count = 0;
125                     sum = 0;
126                     if (i - 1 > 0)
127                         sum = sum +
                                greenValues(i-1, j)
                                *uint8(greenMask(i
                                -1, j));
128                         count = count + 1*
                                uint8(greenMask(i
                                -1, j));
129                     end
130                     if (i + 1 <= imageWidth)
131                         sum = sum +
                                greenValues(i+1, j)
                                *uint8(greenMask(i
                                +1, j));
132                         count = count + 1*
                                uint8(greenMask(i
                                +1, j));
133                     end
134                     if (j + 1 <= imageHeight)
135                         sum = sum +
                                greenValues(i, j+1)
                                *uint8(greenMask(i,
                                 j+1));
136                         count = count + 1*
                                uint8(greenMask(i,
                                j+1));
137                     end
138                     if (j - 1 > 0)
139                         sum = sum +
                                greenValues(i, j-1)
```

```
                                        *uint8(greenMask(i,
                                         j-1));
140                                   count = count + 1*
                                        uint8(greenMask(i,
                                        j-1));
141                                end
142
143                                greenValues(i, j) = sum/
                                      count;
144                            end
145                        end
146                    end
147                    mosim(:, :, 2) = greenValues;
148                    gt = im2double(im);
149
150                    pixelError = abs(gt - mosim);
151                    error(c) = mean(mean(mean(double(
                          pixelError))));
152
153                    c = c + 1;
154
155                end
156            end
157        end
158    end
```