Comp Eng 3SK3

Project 3 – Color Demosaicing for Digital Cameras with Linear Regression

Abd Elelah Arafah - arafaha - 400197623

Course Code: CoE 3SK3

Term: Winter 2023

Date: April 7th, 2023

Introduction:

Below I have included and discussed every result that was outputted from the code that I have developed for this experiment. Along with this write-up and the source code were attached to the submission package for reference.

To follow how each graph is plotted please refer to the code provided for instructive comments for every major line of code. The code is easy to follow as it is written in modules for repeated calculations and concise variable names for each statement as needed.

Part 1: Pseudo Code:

Below I have included the pseudo code which specifically describes the details on pseudocode for the linear regression and demosaicing parts of the code:

```
Pseudocode for the linear regression and demosaicing parts of the code:
Linear Regression:
1. Initialize indices for different patch types
2. Define input matrices for each tile type
3. Preallocate output matrix
4. Calculate input matrices for each tile type by scanning each pixel of learning image
5. Compute linear regression coefficients for each color channel (8 optimal coefficient matrices)
    a. Select the appropriate input matrix
    b. Calculate the linear regression coefficients
Demosaicing:
1. Format the image
2. Expand borders for calculation purposes
3. Compute demosaic image by updating it based on the Bayer filter pattern
    a. Extract 7x7 submatrix
    b. Update demosaiced image based on the Bayer filter pattern for Red, Green, and Blue tiles
```

Part 2: MATLAB Code & Implementation:

Since the source code for this lab is around 250 lines of code, it will not be included in this report. Please refer to the package submission to find the developed code for this project. The code is written in a modular and readable manner, and both the variable name selection and comments should allow for easier interpretation and understanding of it.

The following is the order of steps which were involved in implementing the algorithm:

1. Reading and processing the sample images:
    a. Read test and ground truth image (if available).
    b. Read 3 learning images and resize them into the same size, then concatenate them into one image. I found that combining a few images into one provides better results than training with one image.

```
% input the raw image 'test_img.png'
input_image = imread('sample_image_2.png');


%%%%% comment this out if ground truth image does not exist %%%%%%
% Ground truth demosiaced image for comparison
gt_image = imread('gt_image_2.png');

% Learning from image
% combined_learning_img = imread('test.png');

% Learning from 3 different HR images from the DIV2K database
learn_img_1 = imresize(imread('0804.png'), [512,512]);
learn_img_2 = imresize(imread('0805.png'), [512,512]);
learn_img_3 = imresize(imread('0806.png'), [512,512]);

% Combine learn images into one learn images for more accurate learning.
combined_learning_img = cat(2, learn_img_1, learn_img_2, learn_img_3);
```

2. Extract the Bayer filters of the raw image:
    a. We call the 'image_bayer' function to extract the Bayer filter for the input sample images and the learning images to prepare them for learning.
    b. 'image_bayer' is a function which generates a Bayer mosaic pattern and applies it to the input to out a Bayer image.

```
% Extract bayer filter
image_bayer_var = image_bayer(input_image);
learn_bayer_img = image_bayer(combined_learning_img);
```

```
% Convert input image to a Bayer image
function outputImage = image_bayer(inputImage)
    % Initialize Bayer filter to the size of the input image
    [numRows, numCols, ~] = size(inputImage);
    bayerFilterRGB = inputImage * 0; % Preallocate filter

    % Generate Bayer mosaic patterns in RGB format
    % The same uint8 format as the input image
    for row = 1:numRows
        for col = 1:numCols
            % Red
            if mod(row, 2) == 1 && mod(col, 2) == 1
                bayerFilterRGB(row, col, 1) = 255;
            % Blue
            elseif mod(row, 2) == 0 && mod(col, 2) == 0
                bayerFilterRGB(row, col, 3) = 255;
            % Green
            else
                bayerFilterRGB(row, col, 2) = 255;
            end
        end
    end
    outputImage = inputImage .* (bayerFilterRGB / 255);
end
```

3. Expand image borders:
    a. We call the 'extendImageBorders' function to expand the borders of the learning image.
    b. 'extendImageBorders' is a function which generates a Bayer mosaic pattern and applies it to the input to out a Bayer image. This function is implemented so that it mirrors neighboring pixels to expand the image borders by 3 pixels on each side to add a padding so that the image content is not affected. This allows for the calculations which will be used in the next step for the submatrices of the learning image.

```
% expand borders with custom function
learn_img_exp_borders = extendImageBorders(learn_layer_img);
```

```
% Extend image borders by mirroring neighboring pixels
% Extends by 3 pixels on each side of the original image
function extendedImage = extendImageBorders(inputImage)
    [numRows, numCols, ~] = size(inputImage);

    % Preallocate image matrix to extended size and copy original with offset
    extendedImage = zeros(numRows + 6, numCols + 6);
    extendedImage(4:numRows + 3, 4:numCols + 3) = inputImage;

    % Add mirrored elements
    % Left and right mirror components
    for col = 1:3
        extendedImage(4:numRows + 3, col:col) = inputImage(1:numRows, 5 - col);
        extendedImage(4:numRows + 3, col + numCols + 3:col + numCols + 3) = inputImage(1:numRows, numCols - col);
    end

    % Top and bottom mirror components
    for row = 1:3
        extendedImage(row:row, 1:numCols + 6) = extendedImage(8 - row:8 - row, 1:numCols + 6);
        extendedImage(row + numRows + 3:row + numRows + 3, 1:numCols + 6) = extendedImage(3 + numRows - row:3 + numRows - row, 1:numCols + 6);
    end
end
```

4. Linear regression training of the learning images:
    a. Define indices for each of the 4 patch types and also defines the input matrices for each tile type (Red, Green and Blue).
    b. Prelocate the output matrix.
    c. Compute the input matrices for each type by scanning each pixel of the learning image.
    d. Based on the Bayer pattern which was acquired from the last step.
    e. Finally, compute the linear regression coefficients (8 sets) for the color channel using the calculated input and output matrices.

Note: for better results 7x7 patches were implemented.

```
% Training with machine learning-based linear regression for demosaicing
% Initialize indices for different patch types
redIdx = 0;
green1Idx = 0;
green2Idx = 0;
blueIdx = 0;

% Define input matrices for each tile type
tileCount = (row_count_learn*col_count_learn/4);
inputMatrix = zeros(tileCount,49,4);

%preallocate y-matrix (true values)
outputMatrix = zeros(tileCount,8);
```

```
% Calculate input matrices for each tile type
% Scan each pixel of learning image (offset by 3 pixels to get center pixel of expanded image)
for row = 4:(row_count_learn + 3)
    for col = 4:(col_count_learn + 3)
        % 7x7 sub matrcies
        subMatrix = learn_img_exp_borders(row-3:row+3,col-3:col+3);
        % Convert submatrix to a row vector
        subMatrix = reshape(subMatrix.',1,[]);

        % Update input and output matrices based on the Bayer filter pattern
        if (mod(row,2)==0 && mod(col,2)==0)
            % Red tile (even row, even column)
            redIdx=redIdx+1;
            inputMatrix( redIdx, :, 1 ) = subMatrix;
            outputMatrix(redIdx,1)=train_image_double(row-3,col-3,2);%green1
            outputMatrix(redIdx,2)=train_image_double(row-3,col-3,3);%blue1
        elseif mod(row,2)==0 && mod(col,2)==1
            % Green tile - first row (even row, odd column)
            green1Idx=green1Idx+1;
            inputMatrix( green1Idx, :, 2 ) = subMatrix;
            outputMatrix(green1Idx,3)=train_image_double(row-3,col-3,1);%red1
            outputMatrix(green1Idx,4)=train_image_double(row-3,col-3,3);%blue2
        elseif mod(row,2)==1 && mod(col,2)==0
            % Green tile - second row (odd row, even column)
            green2Idx=green2Idx+1;
            inputMatrix( green2Idx, :, 3 ) = subMatrix;
            outputMatrix(green2Idx,5)=train_image_double(row-3,col-3,1);%red2
            outputMatrix(green2Idx,6)=train_image_double(row-3,col-3,3);%blue3
        else
            % Blue tile (odd row, odd column)
            blueIdx=blueIdx+1;
            inputMatrix( blueIdx, :, 4 ) = subMatrix;
            outputMatrix(blueIdx,7)=train_image_double(row-3,col-3,1);%red3
            outputMatrix(blueIdx,8)=train_image_double(row-3,col-3,2);%green2
        end
    end
end
```

```
% Compute linear regression coefficients for each color channel (8 optimal coefficient matrices)
regressionCoefficients = zeros(49,8);
for idx = 1:8
    % Select the appropriate input matrix
    inputMatrixIdx = round(idx/2); % Adjusts index range to 1-4
    currentInputMatrix = inputMatrix(:,:,inputMatrixIdx);
    transposedInputMatrix = transpose(currentInputMatrix);

    % Calculate the linear regression coefficients
    regressionCoefficients(:,idx) = inv(transposedInputMatrix*currentInputMatrix)*transposedInputMatrix*outputMatrix(:,idx);
end
```

5. Demosaicing using linear regression model:
    a. Format the image and process it using ‘extendImageBorders’ function.
    b. Compute the demosaiced image by updating it using the Bayer filter pattern. For each pixel, we have a 7x7 submatrix and the demosaiced image variable is updated. This is done based on the red, green and blue tiles using the linear regression coefficients which were computed in the last step.

```
% Expand borders for calculation purposes
expandedBayerImage = extendImageBorders(bayerImageSingleLayer);

% Compute demosaic image
offset=-3;
for row = 4:(row_count_input + 3)
    for col = 4:(col_count_input + 3)
        % Extract 7x7 submatrix
        subMatrix = expandedBayerImage(row-3:row+3,col-3:col+3);
        subMatrix=reshape(subMatrix.',1,[]);

        % Update demosaicedImage based on the Bayer filter pattern
        if (mod(row,2)==0 && mod(col,2)==0)
            % Red tile (even row, even column)
            demosaicedImage(row+offset,col+offset,2)=subMatrix*regressionCoefficients(:,1);
            demosaicedImage(row+offset,col+offset,3)=subMatrix*regressionCoefficients(:,2);

        elseif mod(row,2)==0 && mod(col,2)==1
            % Green tile - first row (even row, odd column)
            demosaicedImage(row+offset,col+offset,1)=subMatrix*regressionCoefficients(:,3);
            demosaicedImage(row+offset,col+offset,3)=subMatrix*regressionCoefficients(:,4);
        elseif mod(row,2)==1 && mod(col,2)==0
            % Green tile - second row (odd row, even column)
            demosaicedImage(row+offset,col+offset,1)=subMatrix*regressionCoefficients(:,5);
            demosaicedImage(row+offset,col+offset,3)=subMatrix*regressionCoefficients(:,6);
        else
            % Blue tile (odd row, odd column)
            demosaicedImage(row+offset,col+offset,1)=subMatrix*regressionCoefficients(:,7);
            demosaicedImage(row+offset,col+offset,2)=subMatrix*regressionCoefficients(:,8);
        end
    end
end
```

6. Error computation:
    a. This part is only implemented for testing an image based on the raw image vs it's ground truth rgb format.
    b. We call the built-in immse function to calculate the RMSE between the ground truth image and the outputted images from the implemented algorithm vs the built-in Matlab demosaic function.

```
% Matlab builtin image demosiacing
% specify the Bayer pattern of the raw image (RGGB, GRBG, etc.)
bayer_patterns_mosiac = {'rggb', 'grbg', 'gbrg', 'bggr'};
%input_image = rgb2gray(input_image);
% demosaic the raw image using the built-in demosaic function
rgb_image = demosaic(input_image, bayer_patterns_mosiac{1});

% display the demosaicked RGB image from the builtin function
figure;
imshow(rgb_image);
title('Matlab Image');

%%%%% comment this out if ground truth image does not exist %%%%%%
% RMSE compute
RMSE_error_of_output = immse(demosaicedImage,gt_image)
RMSE_error_matlab_builtin = immse(rgb_image,gt_image)
```

Please note the code takes around 15 seconds to display the output.

Part 3: Results and Discussions:

For testing we use the following sample raw mosaic images:

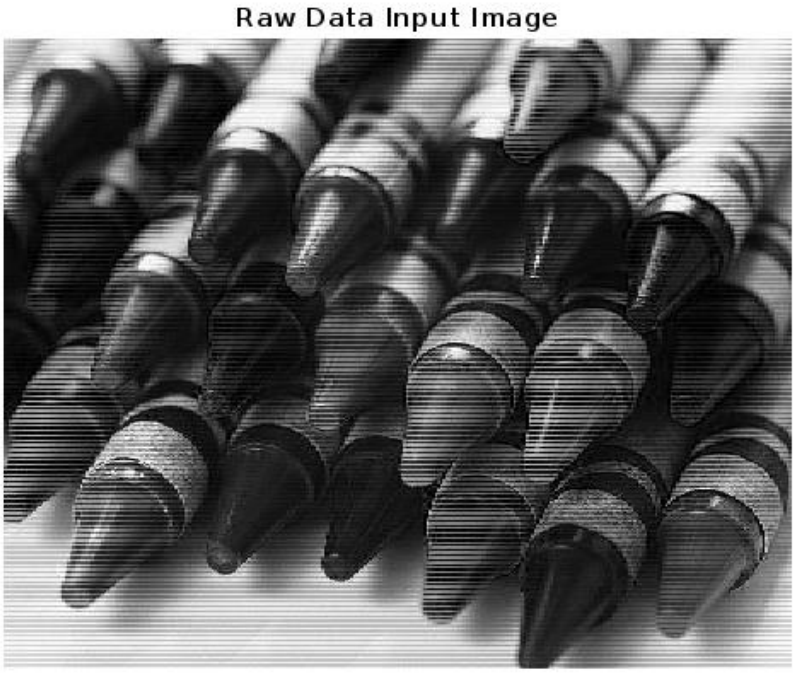

Figure 1. raw input image 1

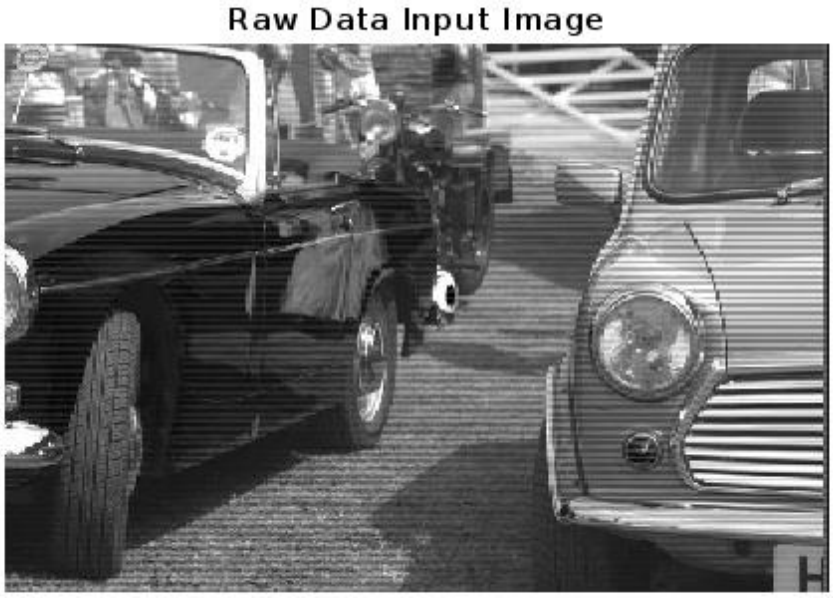

Figure 2. raw input image 2

Figure 3. raw input image 3

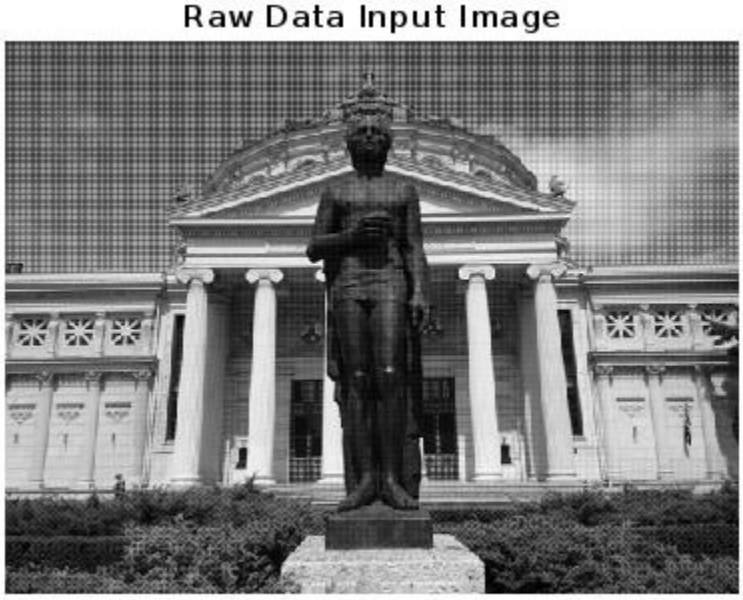

Figure 4. raw input image 4

Figure 5. raw input image 5

The Ground truth images for comparison purposes:

Figure 6. ground truth image 1

Figure 7. ground truth image 2

Figure 8. ground truth image 3

Note that the ground truth image is not provided for the fourth and fifth test mosaic images.

Bayer pattern of the images:

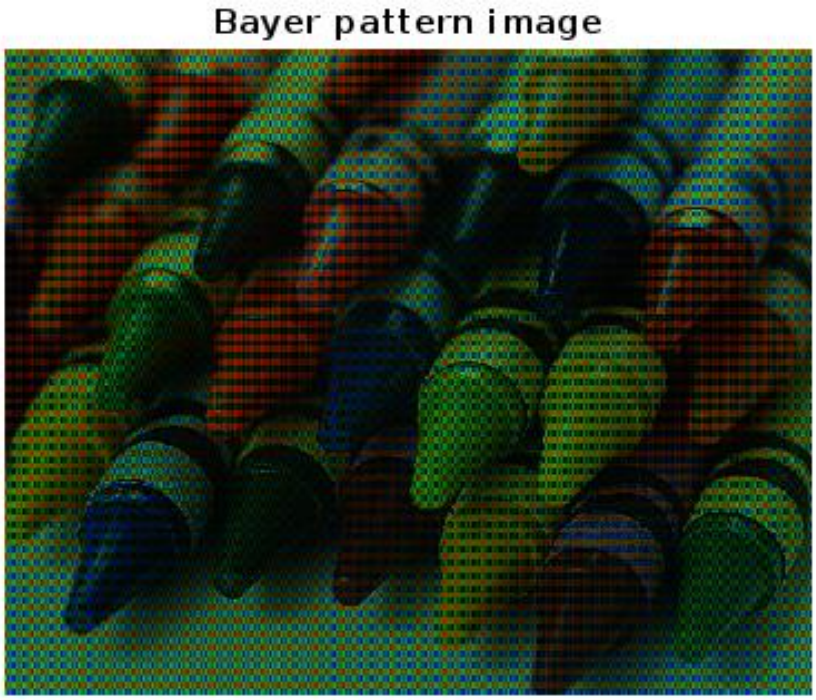

Figure 9. Extracted Bayer pattern image 1

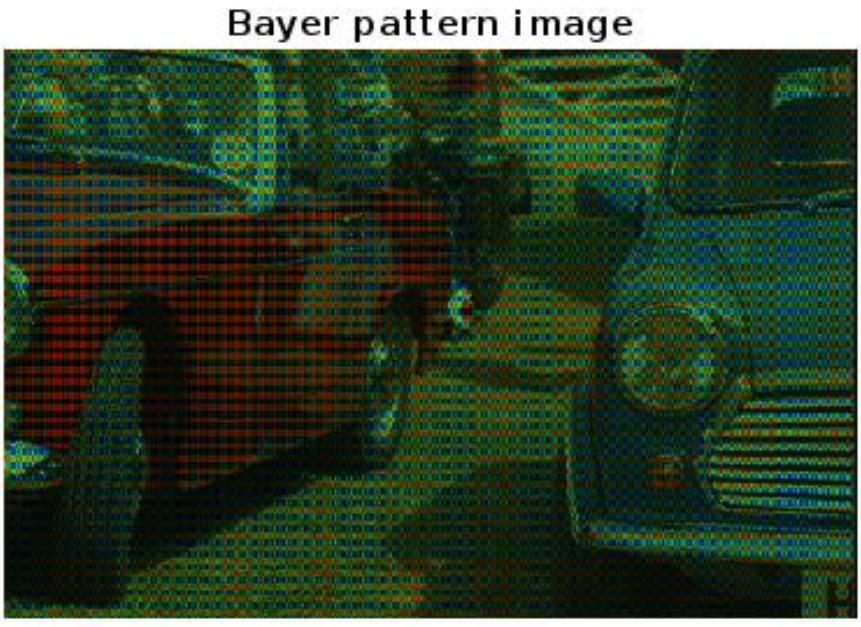

Figure 10. Extracted Bayer pattern image 2

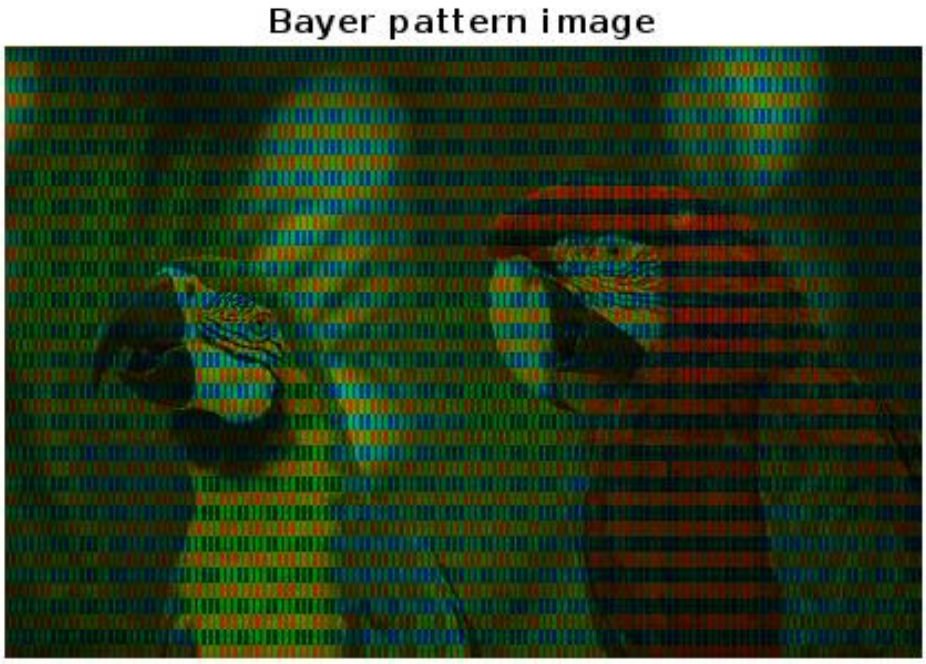

Figure 11. Extracted Bayer pattern image 3

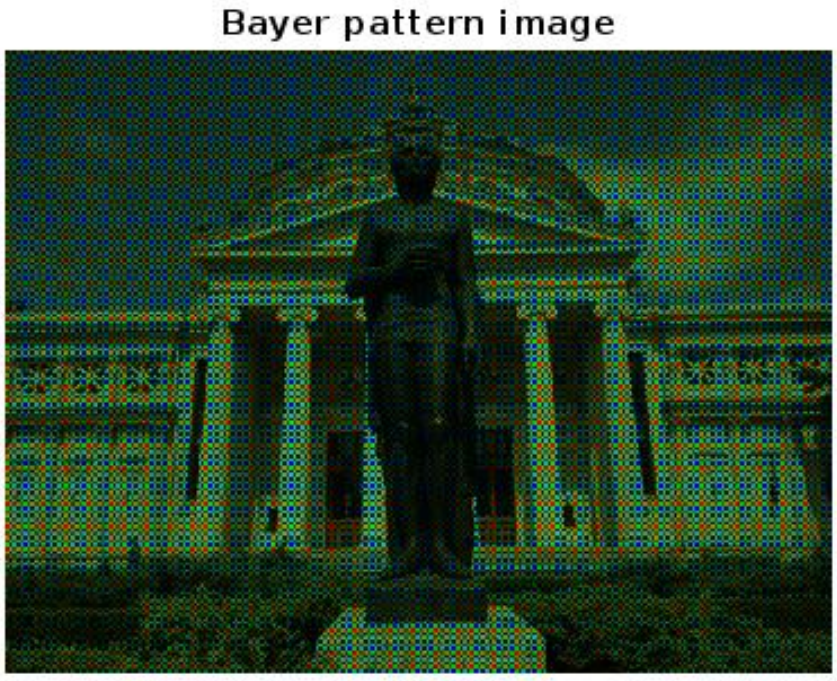

Figure 12. Extracted Bayer pattern image 4

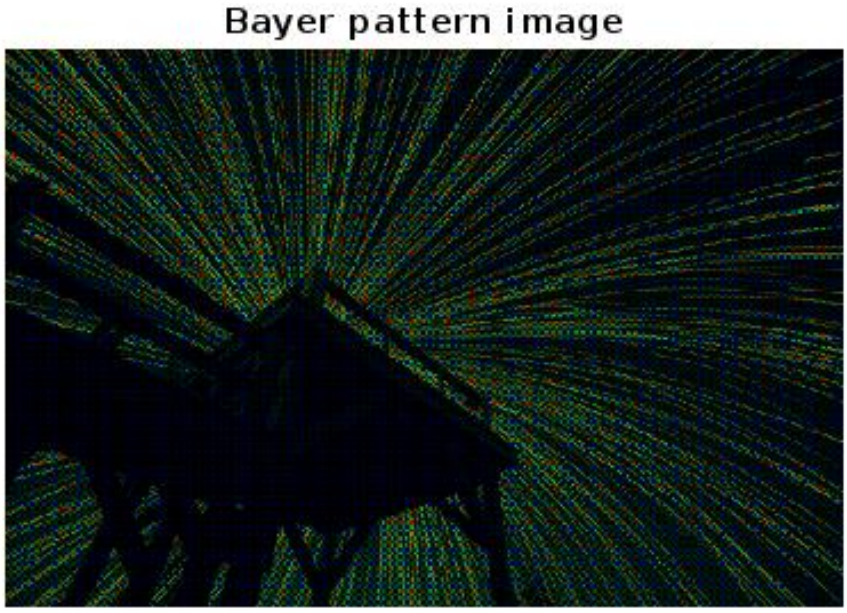

Figure 13. Extracted Bayer pattern image 5

The output of the demosaic() function vs the linear regression model:

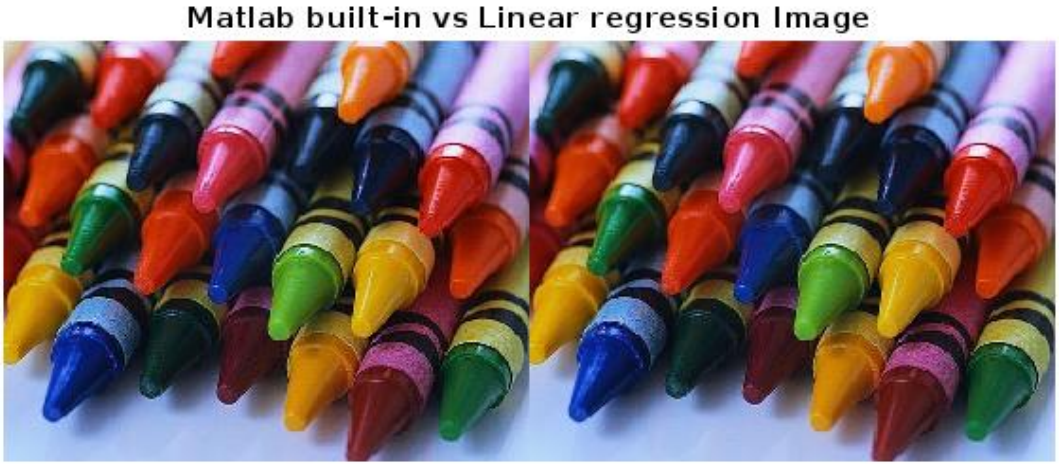

Figure 14. demosaic() function vs linear regression output image 1

Figure 15. demosaic() function vs linear regression output image 2

Figure 16. demosaic() function vs linear regression output image 3

Figure 17. demosaic() function vs linear regression output image 4

Figure 18. demosaic() function vs linear regression output image 5

RMSE Errors:

The RMSE error for sample image 1 when comparing it to the ground truth image (figures. 1, 9, 14):

```
Linear Regression model completed!
Now preforming Demosaicing...

RMSE_error_of_output =

  14.453906249999999


RMSE_error_matlab_builtin =

  15.665890046296296
```

The RMSE error for sample image 2 when comparing it to the ground truth image (figures. 2, 10, 15):

```
Linear Regression model completed!
Now preforming Demosaicing...

RMSE_error_of_output =

  38.163863763260757


RMSE_error_matlab_builtin =

  44.384765494137348
```

The RMSE error for sample image 3 when comparing it to the ground truth image (figures. 3, 11, 16):

```
Linear Regression model completed!
Now preforming Demosaicing...

RMSE_error_of_output =

     2.180420778062609e+02


RMSE_error_matlab_builtin =

     2.243297263251410e+02
```

The results presented above indicate that the root mean square error (RMSE) for all 3 test images are better when compared to that of the built-in demosaic() function. In the case of the first sample image, the relative error is approximately 7.7% when comparing the errors. For the second image, the relative error amounts to 14%. Additionally, the third sample image yields a

relative error of 2.8%. However, this error cannot be observed in the visual output image as they look virtually identical.

As mentioned in the implementation section, I used 7x7 submatrices to simulate the 4 types of patches. This decision was made as when simulating patches with 5x5 submatrices, I got a relatively higher RMSE value. Additionally, the 5x5 patches algorithm produced a slightly higher RMSE value than the built-in Matlab demosaic() function. This is expected as the 7x7 submatrices captures a larger context and it allows for the linear regression model to consider a larger range of the neighboring pixels, which enhances the accuracy of the predicted outputs.

For example, when comparing the first sample image, for the 7x7 patch algorithm, we got an RMSE error of 14.454, while for the 5x5 patch we got an RMSE value of 17.051 as seen below:

```
Linear Regression model completed!
Now preforming Demosaicing...

RMSE_error_of_output =

  17.050510205430001


RMSE_error_matlab_builtin =

  15.665890046296296
```

We see that implementing the 7x7 patches has produced a clear improvement in the accuracy of the demosaiced image when compared to the original image even though it requires a slightly higher computational complexity.

In conclusion, from this experiment, we can see how implementing a custom linear regression model for demosaicing a raw image could be beneficial as we see a more precise result as compared to the matlab demosaic() function. This can be incorporated into applications where the precision of the demosaic is optimized for accuracy and performance.