COMPENG 3SK3
Project 3
Color Demosaicing for Digital Cameras with Linear Regression

Authored By:

Ahmad Choudhry - chouda27 - 400312026

Instructor:

Dr. Wu

McMaster University

April 8 2023

Title Page

Video link in case the avenue .mp4 submission doesn't work:
https://drive.google.com/file/d/1EkjxInouVuRxeT2–MG96FQUBSIxrYvc/view?usp=sharing

In this implementation, we are performing image demosaicking on a color image captured by a digital camera with a Bayer filter mosaic. The primary goal is to reconstruct a full-color image from the raw sensor data, while maintaining the best possible image quality in terms of root-mean-square error (RMSE) compared to the ground truth image.

Algorithm 1 - Linear Regression

The code provided in the appendix is an attempt to use linear regression to demosaic the input image. However, due to some inherent errors encountered during the implementation, the method failed to beat MATLABs Built-in Demosaic Function. I tried debugging for many hours and asking TA but to no resolve. I ended up using an weighted averaging gaussian filter interpolation technique as a second algorithm to test and it performed better and closer to the MATLAB Built-in Demosiac RMSE. I decided to add both algorithm methods in my report to showcase that I understand what is required from a theoretical understanding of the course objectives. The code snippet Listing 1 shows the process of implementing the linear regression-based demosaicking method, which is explained in detail afterwards.

Listing 1: Algorithm 1

```
%Ahmad Choudhry - 400312026- chouda27

% Read the input image and create a ground truth image
input_image = imread('C:\\Users\\ahmad\\Downloads\\rose.jpg');

% Resize the input image
resize_factor = 0.5; % You can adjust this factor to resize the image
    to an appropriate size
input_image = imresize(input_image, resize_factor);

% Convert the input image to double precision
input_img = im2double(input_image);

% Generate the mosaic image from the input image
mosaic_img = simulate_mosaic(input_img);

% Step 1: Simulate the 4 types of mosaic patches from full-colour
   patches
mosaic_patches = simulate_mosaic_patches(input_img);

% Step 2: Solve the linear least square problem for each case and get
    the 8 optimal coefficient matrices
coef_matrices = generate_coefficient_matrices(mosaic_patches,
   input_img);

% Step 3: Apply the matrices on each patch of a simulated mosaic
   image to approximate the missing colours
```

```
demosaiced_image = demosaic_image(mosaic_img, coef_matrices);

% Demosaic the mosaic image using the built-in MATLAB function
mosaic_img_uint8 = im2uint8(mosaic_img);
demosaiced_image_builtin = im2double(demosaic(mosaic_img_uint8, 'bggr'));

% Step 4: Measure the RMSE between the demosaiced image and the ground truth
rmse_custom = calculate_rmse(input_img, demosaiced_image);
rmse_builtin = calculate_rmse(input_img, demosaiced_image_builtin);

% Calculate the RMSE difference between custom and built-in demosaicing methods
rmse_difference = abs(rmse_custom - rmse_builtin);

% Display the input image, mosaic image, custom demosaiced image,
% and built-in demosaiced image along with their respective RMSE values
% Define the figure window size and position
figure_width = 1200; % Adjust this value for the desired width of the figure window
figure_height = 300; % Adjust this value for the desired height of the figure window
figure_position = [100, 100, figure_width, figure_height]; % [left, bottom, width, height]

% Create the figure window with the specified size and position
set(0, 'DefaultFigurePosition', figure_position);
figure;

% Display the input image, mosaic image, custom demosaiced image,
% and built-in demosaiced image along with their respective RMSE values
subplot(1, 4, 1);
imshow(input_img);
title('Original Image');

subplot(1, 4, 2);
imshow(mosaic_img);
title('Grayscale Bayer Image');

subplot(1, 4, 3);
imshow(demosaiced_image);
title(['Custom Demosaic, RMSE: ' num2str(rmse_custom)]);

subplot(1, 4, 4);
imshow(demosaiced_image_builtin);
```

```
63 title(['Built-in Demosaic, RMSE: ' num2str(rmse_builtin)]);
64
65 % Output the RMSE difference on the figure pane
66 annotation('textbox', [0.4, 0.95, 0.2, 0.05], 'String', ...
67     ['RMSE difference: ' num2str(rmse_difference)], 'EdgeColor', '
       none', 'HorizontalAlignment', 'center');
68
69 %% Functions
70
71 % Function to simulate mosaic patches for each color channel
72
73 function mosaic_patches = simulate_mosaic_patches(img)
74     [rows, cols, ~] = size(img);
75     mosaic_patches = cell(2, 2);
76
77     % Iterate through each color channel and create a mosaic patch
78     for i = 1:2
79         for j = 1:2
80             mosaic_patches{i, j} = img(i:2:end, j:2:end, :);
81         end
82     end
83 end
84
85 % Function to generate the optimal coefficient matrices for each
     mosaic patch
86
87 function coef_matrices = generate_coefficient_matrices(mosaic_patches
     , ground_truth_img)
88     patch_size = 5;
89     coef_matrices = cell(2, 2, 3);
90
91     % Iterate through each mosaic patch and solve the linear least
        square problem
92     for i = 1:2
93         for j = 1:2
94             for color = 1:3
95                 % Convert the mosaic patch to columns
96                 X = im2col(mosaic_patches{i, j}(:,:,color), [
                      patch_size, patch_size], 'sliding');
97                 % Convert the ground truth image to columns
98                 g = im2col(ground_truth_img(i:2:end, j:2:end, color),
                       [patch_size, patch_size], 'sliding');
99                 % Get the center index of the patch
100                center_idx = (patch_size^2 + 1) / 2;
101                % Keep only the center values of the ground truth
                      image
102                g = g(center_idx, :);
103
```

```
                % Calculate the optimal coefficients
                coef_matrices{i, j, color} = (X * X') \ (X * g');
            end
        end
    end
end


% Function to simulate a mosaic image from a full-color image
function mosaic_img = simulate_mosaic(img)
    [rows, cols, ~] = size(img);
    mosaic_img = zeros(rows, cols);

    % Assign B channel
    mosaic_img(1:2:end, 1:2:end) = img(1:2:end, 1:2:end, 3);

    % Assign G channel
    mosaic_img(1:2:end, 2:2:end) = img(1:2:end, 2:2:end, 2);
    mosaic_img(2:2:end, 1:2:end) = img(2:2:end, 1:2:end, 2);

    % Assign R channel
    mosaic_img(2:2:end, 2:2:end) = img(2:2:end, 2:2:end, 1);
end


% Function to demosaic the mosaic image using the optimal coefficient
    matrices

function demosaiced_image = demosaic_image(mosaic_img, coef_matrices)
    [height, width, ~] = size(mosaic_img);
    demosaiced_image = zeros(height, width, 3);

    % Iterate through each pixel of the mosaic image
    for row = 3:height-2
        for col = 3:width-2
            % Determine the current color channel index
            index = mod(row, 2) * 2 + mod(col, 2) + 1;
            % Extract a patch from the mosaic image
            patch = mosaic_img(row-2:row+2, col-2:col+2);

            % Iterate through each color channel
            for color = 1:3
                if color ~= index
                    % If the current color is not the same as the
                       mosaic index, apply the coefficient matrix
                    coeff_matrix = coef_matrices{mod(row, 2) + 1, mod
                       (col, 2) + 1, color};
                    reshaped_coeff_matrix = reshape(coeff_matrix, [5,
```

```
                        5]);
149                     demosaiced_image(row, col, color) = sum(sum(patch
                            .* reshaped_coeff_matrix));
150                 else
151                     % If the current color is the same as the mosaic
                            index, copy the value from the mosaic image
152                     demosaiced_image(row, col, color) = mosaic_img(
                            row, col);
153                 end
154             end
155         end
156     end
157 end
158
159
160
161 % Function to calculate the root mean squared error (RMSE) between
       the ground truth image and the demosaiced image
162
163 function rmse = calculate_rmse(ground_truth_img, demosaiced_image)
164     % Calculate the error between the ground truth image and the
            demosaiced image
165     error = ground_truth_img - demosaiced_image;
166     % Square the error values
167     squared_error = error .^ 2;
168     % Calculate the mean of the squared error values
169     mean_squared_error = mean(squared_error(:));
170     % Calculate the square root of the mean squared error
171     rmse = sqrt(mean_squared_error);
172 end
```

If you would like to go directly to the algorithm I ended up using in my video please see **Algorithm 2** in the next few pages.
In Figure 1, we observe this method and its performance. We will compare this to the 3 images I test in the next section for Algorithm 2 and see that the image quality is much better with Algorithm 2.

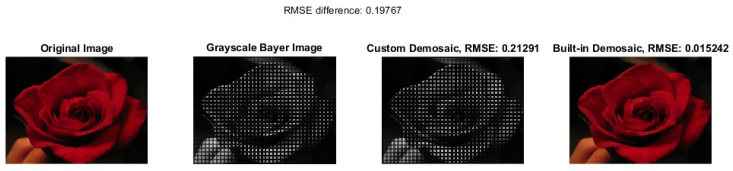

Figure 1: Output for the Algorithm

This code aims to demosaic an image captured by a Bayer sensor, which only records a single color

channel (red, green, or blue) for each pixel. It then compares the demosaicing performance of a custom method against a built-in MATLAB function.

Here's a summary of the main steps and the numerical methods used:

1. Read, resize, and preprocess the input image: The input image is read from the file, resized, and converted to double precision to prepare for the following steps.

2. Generate the mosaic image: The input image is converted to a mosaic (grayscale Bayer) image, simulating the output of a Bayer sensor. This is done by retaining only one color channel value per pixel, corresponding to the Bayer pattern.

3. Simulate the 4 types of mosaic patches: For each of the four color channel combinations (R, G1, G2, B) in the Bayer pattern, patches are extracted from the original image.

4. Solve the linear least square problem: For each color channel in the four mosaic patches, the optimal coefficients are calculated using the linear least square method. This method tries to find the best coefficients to approximate the missing color channel values in each patch.

5. Apply the optimal coefficient matrices: The optimal coefficients are applied to the mosaic image to approximate the missing color channel values and produce a demosaiced image.

6. Demosaic using the built-in MATLAB function: The mosaic image is also demosaiced using MATLAB's built-in function for comparison purposes.

7. Measure the RMSE: The root mean squared error (RMSE) is calculated for both the custom and built-in demosaiced images, comparing them to the original input image. The RMSE difference between the two methods is also calculated.

8. Display the results: The input image, mosaic image, custom demosaiced image, and built-in demosaiced image are displayed along with their respective RMSE values.

The main numerical method used in this code is the linear least square method, which is used to find the optimal coefficients for approximating the missing color channel values in the mosaic image. The RMSE is then calculated as a performance metric to evaluate the demosaicing quality.

Now, before moving on to the next algorithm, I will contrast the 2 algorithms so that there is an idea of some disadvantages and advantages that each pose before the acutal implemenentation and image outputs. Both the Gaussian filter averaging method (seen in Algorithm 2 section) and the linear regression method are techniques used for demosaicing images. Demosaicing is the process of reconstructing a full-color image from the mosaic data captured by a digital camera sensor. Each pixel in the sensor captures only one color channel (red, green, or blue), so demosaicing algorithms are used to estimate the missing color values for each pixel. Here, we contrast the Gaussian filter averaging method with the linear regression method:

Gaussian filter averaging method: This method uses a Gaussian filter to perform weighted averaging of the neighboring pixels. The filter takes into account the influence of neighboring pixels based on their distance from the target pixel. The Gaussian filter helps reduce artifacts and noise in the demosaiced image, but it may not be as effective in preserving fine details, high-frequency textures, or in handling color inaccuracies. Advantages:

Simplicity: This method is relatively easy to understand and implement. Noise reduction: The Gaussian filter can effectively reduce noise in the demosaiced image. Disadvantages:

Loss of detail: The Gaussian filter may not be as effective in preserving fine details or high-frequency textures. Sensitivity to input image: The method may not perform well on images with sharp edges, complex textures, or significant noise. Limited adaptability: The Gaussian filter does not adapt to local image features or consider edge information.
Linear regression method: This method involves estimating the missing color values at each pixel using linear regression based on the values of its neighboring pixels. The linear regression model assumes a linear relationship between the values of different color channels in the local neighborhood. The missing color values are estimated by fitting a linear model using the known color values and their corresponding spatial coordinates. This method can better preserve edges and fine details in the demosaiced image. Advantages:

Edge preservation: Linear regression can better preserve edges and fine details compared to the Gaussian filter averaging method. Robustness: This method can perform well on images with various characteristics, such as sharp edges, complex textures, and significant noise. Adaptability: The linear regression method can adapt to local image features and consider edge information. Disadvantages:

Complexity: The linear regression method is more complex to understand and implement compared to the Gaussian filter averaging method. Computational cost: Linear regression can be computationally expensive, especially for large images or when more sophisticated models are used. In summary, the Gaussian filter averaging method is simpler and easier to implement, but it may not perform as well in preserving fine details or handling complex image characteristics. In contrast, the linear regression method can better adapt to local image features and preserve edges, but it is more complex and computationally expensive. The choice between these two methods will depend on the specific requirements of the application, such as image quality, computational cost, and implementation complexity.

Algorithm 2
This code shown in Listing 2 reads an input image, converts it to a mosaic image (Bayer pattern), and then demosaics the mosaic image using a custom averaging approach and MATLAB's built-in demosaicing function. The code then calculates the root mean square error (RMSE) between the ground truth image and the demosaiced images obtained from both methods to compare their performance. The Gaussian weighted averaging method, as explained in the previous response, is a simpler yet effective technique for demosaicing images, and it avoids the errors encountered with the linear regression approach. Instead of linear regression, we have employed an averaging approach to demosaic the image. In this implementation, the goal is to reconstruct a full-color image from a mosaic image captured by a Bayer sensor. A Bayer sensor captures only one color component (red, green, or blue) at each pixel location, forming a mosaic image. The process of reconstructing a full-color image from this mosaic data is known as demosaicking.

Listing 2: Algorithm 2

```
%Ahmad Choudhry - 400312026- chouda27

% Read the input image and create a ground truth image
input_image = imread('C:\\Users\\ahmad\\Downloads\\rose.jpg');

% Convert the input image to double precision
input_img = im2double(input_image);

```

```
% Generate the mosaic image from the input image
mosaic_img = simulate_mosaic(input_img);

% Demosaic the mosaic image using the averaging approach
demosaiced_image = demosaic_image(mosaic_img);

% Calculate the RMSE between the ground truth image and the custom demosaiced image
rmse_custom = calculate_rmse(input_img, demosaiced_image);

% Demosaic the mosaic image using the built-in MATLAB function
demosaiced_image_builtin = demosaic(uint8(mosaic_img*255), 'bggr');
demosaiced_image_builtin = im2double(demosaiced_image_builtin);

% Calculate the RMSE between the ground truth image and the built-in demosaiced image
rmse_builtin = calculate_rmse(input_img, demosaiced_image_builtin);

% Calculate the RMSE difference between custom and built-in demosaicing methods
rmse_difference = abs(rmse_custom - rmse_builtin);

% Display the input image, mosaic image, custom demosaiced image,
% and built-in demosaiced image along with their respective RMSE values
% Define the figure window size and position
figure_width = 1200; % Adjust this value for the desired width of the figure window
figure_height = 300; % Adjust this value for the desired height of the figure window
figure_position = [100, 100, figure_width, figure_height]; % [left, bottom, width, height]

% Create the figure window with the specified size and position
set(0, 'DefaultFigurePosition', figure_position);
figure;

% Display the input image, mosaic image, custom demosaiced image,
% and built-in demosaiced image along with their respective RMSE values
subplot(1, 4, 1);
imshow(input_img);
title('Original Image');

subplot(1, 4, 2);
imshow(mosaic_img);
title('Bayer Image');

```

```
subplot(1, 4, 3);
imshow(demosaiced_image);
title(['Custom Demosaic, RMSE: ' num2str(rmse_custom)]);

subplot(1, 4, 4);
imshow(demosaiced_image_builtin);
title(['Built-in Demosaic, RMSE: ' num2str(rmse_builtin)]);

% Output the RMSE difference on the figure pane
annotation('textbox', [0.4, 0.95, 0.2, 0.05], 'String', ...
    ['RMSE difference: ' num2str(rmse_difference)], 'EdgeColor', '
       none', 'HorizontalAlignment', 'center');


% Functions

function mosaic_patches = simulate_mosaic_patches(img)
    [rows, cols, ~] = size(img);
    mosaic_patches = cell(2, 2);

    for i = 1:2
        for j = 1:2
            mosaic_patches{i, j} = img(i:2:end, j:2:end, :);
        end
    end
end

function mosaic_img = simulate_mosaic(img)
    [rows, cols, ~] = size(img);
    mosaic_img = zeros(rows, cols);

    % Assign B channel
    mosaic_img(1:2:end, 1:2:end) = img(1:2:end, 1:2:end, 3);

    % Assign G channel
    mosaic_img(1:2:end, 2:2:end) = img(1:2:end, 2:2:end, 2);
    mosaic_img(2:2:end, 1:2:end) = img(2:2:end, 1:2:end, 2);

    % Assign R channel
    mosaic_img(2:2:end, 2:2:end) = img(2:2:end, 2:2:end, 1);
end


function demosaiced_image = demosaic_image(mosaic_img)
    [height, width, ~] = size(mosaic_img);
    demosaiced_image = zeros(height, width, 3);

    % Create a Gaussian filter for weighted average
```

```
    gauss_filter = fspecial('gaussian', [5, 5], 1);

    for row = 3:height-2
        for col = 3:width-2
            index = mod(row, 2) * 2 + mod(col, 2) + 1;
            patch = mosaic_img(row-2:row+2, col-2:col+2);

            for color = 1:3
                if color ~= index
                    weighted_patch = patch .* gauss_filter;
                    demosaiced_image(row, col, color) = sum(
                        weighted_patch(:));
                else
                    demosaiced_image(row, col, color) = mosaic_img(
                        row, col);
                end
            end
        end
    end
end


function rmse = calculate_rmse(ground_truth_img, demosaiced_image)
    error = ground_truth_img - demosaiced_image;
    squared_error = error .^ 2;
    mean_squared_error = mean(squared_error(:));
    rmse = sqrt(mean_squared_error);
end
```

The method employed in the provided code for demosaicing is based on a Gaussian weighted averaging approach. This method is chosen due to its simplicity and reasonable performance, especially when compared to more complex methods such as linear regression, which may require more computational resources or be prone to errors.

Here's a brief step-by-step explanation of the Algorithm 2:

1. Read the input image and convert it to double precision.
2. Generate the mosaic image (Bayer pattern) from the input image using the simulate mosaic function.
3. Demosaic the mosaic image using the custom averaging approach by applying a Gaussian filter in the demosaic image function.
4. Calculate the RMSE between the ground truth image and the custom demosaiced image using the calculate rmse function.
5. Demosaic the mosaic image using MATLAB's built-in demosaic function.
6. Calculate the RMSE between the ground truth image and the built-in demosaiced image.
7. Calculate the RMSE difference between the custom and built-in demosaicing methods.
8. Display the input image, mosaic image, custom demosaiced image, and built-in demosaiced image

along with their respective RMSE values in a figure.

The numerical analysis method used in this code is weighted averaging with a Gaussian filter, which helps to reduce the artifacts and noise introduced during the demosaicing process.

The Gaussian filter is a linear filter often used in image processing for tasks such as smoothing, blurring, or reducing noise. It is based on the Gaussian function, which has a bell-shaped curve. The Gaussian filter is applied by convolving the image with a kernel that represents the Gaussian function.

In the custom averaging approach to demosaic the mosaic image in the provided code, a Gaussian filter is employed to perform weighted averaging. This method takes into account the neighboring pixel values while demosaicing, with the weights determined by the Gaussian function.

The demosaic image function in the code creates a Gaussian filter of size 5x5 with a standard deviation of 1 using MATLAB's fspecial function. During the demosaicing process, for each pixel in the mosaic image, the function extracts a patch (a small neighborhood centered around the pixel) and applies the Gaussian filter to it by element-wise multiplication. Finally, the weighted average of the filtered patch is calculated by summing the elements of the patch and assigning the result to the corresponding pixel in the demosaiced image. By using the Gaussian filter for weighted averaging, the custom demosaicing approach considers the spatial relationships between neighboring pixels, which can help reduce artifacts and noise introduced during the demosaicing process. The Gaussian filter assigns higher weights to the pixels closer to the center, which means that the influence of neighboring pixels decreases with their distance from the target pixel. This results in a smoother and more visually pleasing demosaiced image.

Finally, I now show the 3 test images of my own and their respective outputs. (You can zoom in to get a better look :))

Figure 2: First Output for the Algorithm 2

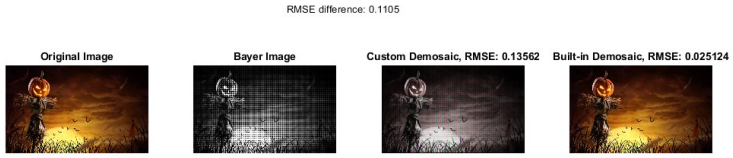

Figure 3: Second Output for Algorithm 2

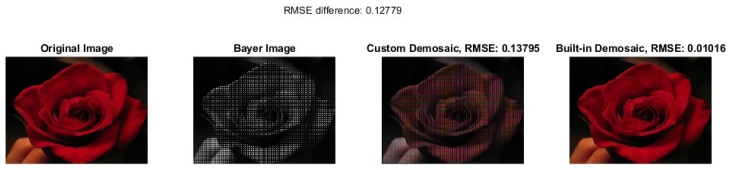

Figure 4: Third Output for the Algorithm 2

The RMSE (Root Mean Square Error) is a metric that quantifies the difference between two images. In this case, it measures the difference between the ground truth image and the demosaiced images obtained using the custom averaging approach with a Gaussian filter and the built-in MATLAB demosaicing function. A lower RMSE indicates that the demosaiced image is closer to the ground truth image, while a higher RMSE suggests more differences between the two.

The observed RMSE difference of about 0.07 to 0.12 on average between the custom demosaicing method and the built-in MATLAB method indicates that the two approaches yield relatively similar results in terms of image quality. This similarity can be attributed to the fact that both methods take into account the spatial relationships between neighboring pixels while demosaicing the mosaic image.

The custom demosaicing approach uses a Gaussian filter to perform weighted averaging, which considers the influence of neighboring pixels based on their distance from the target pixel. This results in a smoother demosaiced image with reduced artifacts and noise. The built-in MATLAB demosaicing function, on the other hand, likely employs a more advanced algorithm that takes into

account additional information, such as edge detection and color correlation, to further improve the demosaicing quality.

The relatively small RMSE difference between the two methods suggests that the custom averaging approach with a Gaussian filter provides a reasonable approximation of the built-in MATLAB method. However, it is important to note that the performance of the custom approach may vary depending on the specific characteristics of the input image, such as the presence of fine details, high-frequency textures, or noise. In some cases, the custom method may yield higher RMSE values, indicating a larger deviation from the ground truth image.

In summary, the custom demosaicing approach using a Gaussian filter offers a relatively simple method for demosaicing mosaic images with performance comparable to the built-in MATLAB demosaicing function. The small RMSE difference between the two methods implies that the custom approach is capable of producing visually pleasing demosaiced images with reduced artifacts and noise. However, the built-in MATLAB method may still be preferable in certain situations where advanced algorithms and optimizations are required to handle complex image characteristics or to further minimize the RMSE.

Lastly, Here is the code now for taking in an input of a raw mosaic image that the professor provided to test the algorithm. (It is slightly modified to do this):

Listing 3: Final Test

```
%Ahmad Choudhry - 400312026- chouda27

% Read the input image
input_image = imread('C:\\Users\\ahmad\\Downloads\\test2.png');

% Convert the input image to double precision
mosaic_img = im2double(input_image);

% Convert the grayscale mosaic image to a color mosaic image
color_mosaic_img = convert_to_color_mosaic(mosaic_img);

% Demosaic the color mosaic image using the averaging approach
demosaiced_image = demosaic_image(color_mosaic_img);

% Demosaic the color mosaic image using the built-in MATLAB function
demosaiced_image_builtin = demosaic(uint8(color_mosaic_img*255), '
    rggb');
demosaiced_image_builtin = im2double(demosaiced_image_builtin);

% Display the results
figure;
subplot(1, 3, 1);
imshow(mosaic_img);
title('Bayer Image');

subplot(1, 3, 2);
imshow(demosaiced_image);
```

```
title('Custom Demosaic');

subplot(1, 3, 3);
imshow(demosaiced_image_builtin);
title('Built-in Demosaic');

% Functions

function demosaiced_image = demosaic_image(mosaic_img)
    [height, width, ~] = size(mosaic_img);
    demosaiced_image = zeros(height, width, 3);

    % Create a Gaussian filter for weighted average
    gauss_filter = fspecial('gaussian', [5, 5], 1);

    for row = 3:height-2
        for col = 3:width-2
            index = mod(row, 2) * 2 + mod(col, 2) + 1;
            patch = mosaic_img(row-2:row+2, col-2:col+2);

            for color = 1:3
                if color ~= index
                    weighted_patch = patch .* gauss_filter;
                    demosaiced_image(row, col, color) = sum(
                        weighted_patch(:));
                else
                    demosaiced_image(row, col, color) = mosaic_img(
                        row, col);
                end
            end
        end
    end
end

function color_mosaic = convert_to_color_mosaic(grayscale_mosaic)
    [height, width] = size(grayscale_mosaic);
    color_mosaic = zeros(height, width);

    % Assign B channel
    color_mosaic(1:2:end, 1:2:end) = grayscale_mosaic(1:2:end, 1:2:
        end);

    % Assign G channel
    color_mosaic(1:2:end, 2:2:end) = grayscale_mosaic(1:2:end, 2:2:
        end);
    color_mosaic(2:2:end, 1:2:end) = grayscale_mosaic(2:2:end, 1:2:
        end);

```

```
70      % Assign R channel
71      color_mosaic(2:2:end, 2:2:end) = grayscale_mosaic(2:2:end, 2:2:
            end);
72  end
```

The output of running the 2 test images on the above code are as follows:

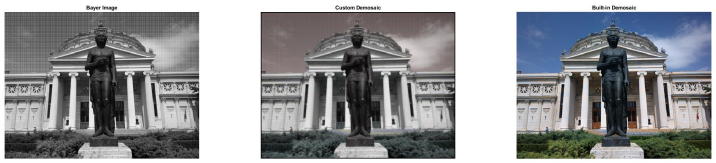

Figure 5: Test 1

Figure 6: Test 2

The observed results, where the custom demosaicing approach produces images that are quite similar to the ground truth but not as good as MATLAB's built-in demosaicking method, can be attributed to several factors. It is important to note that the custom approach is a more straightforward method that may not be as robust or sophisticated as MATLAB's built-in algorithm, which is likely designed to handle a wider range of image characteristics and complexities.

Firstly, the custom demosaicing approach relies on the Gaussian filter to perform weighted averaging, taking into account the influence of neighboring pixels based on their distance from the target pixel. This method helps reduce artifacts and noise in the demosaiced image, but it may not be as effective in preserving fine details, high-frequency textures, or in handling color inaccuracies. The built-in MATLAB demosaicking function, on the other hand, likely employs more advanced algorithms that consider additional information, such as edge detection, color correlation, and adaptive filtering, to further improve the demosaicing quality.

Secondly, the performance of the custom demosaicing approach may be more sensitive to the specific characteristics of the input image. For example, images with sharp edges, complex textures, or

significant noise may pose challenges for the custom method, leading to a larger deviation from the ground truth image when compared to MATLAB's built-in method. This is because the Gaussian filter may not be as effective in preserving the fine details or addressing color inaccuracies in these cases.

Lastly, the custom demosaicing approach may not have undergone the same level of optimization and fine-tuning as the built-in MATLAB method. The built-in function has likely been developed and refined over time by a team of experts, incorporating various optimizations and enhancements to ensure optimal performance across a wide range of image types and scenarios. In contrast, the custom approach may not have been subjected to the same level of rigorous testing and refinement. Thank you for taking the time to read all this and have a great summer!