Tampere University

Roosa Kuusivaara & Väinö-Waltteri Granat

Digital Image Formation And Processing - Report

Faculty of Information Technology and Communication Sciences (ITC)
Laboratory Report

Abstract

Roosa Kuusivaara & Väinö-Waltteri Granat: Digital Image Formation And Processing - Report
Laboratory Report
Tampere University
Master's Degree Programme in Signal Processing
December 2023

This report documents the work done in the Digital Image Formation And Processing assignment as a part of the Advanced signal processing laboratory course. In this assignment we familiarized ourselves with basics of digital image formation from raw sensor data as well as image processing.

Keywords: Image processing, Image formation.

Contents

1 Introduction

In this report we describe our work done in the 'Digital Image Formation and Processing' laboratory assignment for the Advanced Signal Processing Laboratory course. In this assignment we implemented a basic raw image formation and processing pipeline in Matlab to construct RGB images from raw sensor data.

2 Methodology

In this section we will present our implementation of the image processing pipeline, by going over the main parts of the Matlab code as well as explaining the the decisions we took when requirements we ambigious.

2.1 Overview of the pipeline

Figure 2.1 shows the complete image processing and formation pipeline that we are going to introduce in this report. The following sections will go over the different parts of the pipeline. The pipeline takes in raw sensor data of Bayer arrays in RGGB format and produces RGB images. In addition to image formation the pipeline also performs focusing, white balancing and contrast and saturation correction.

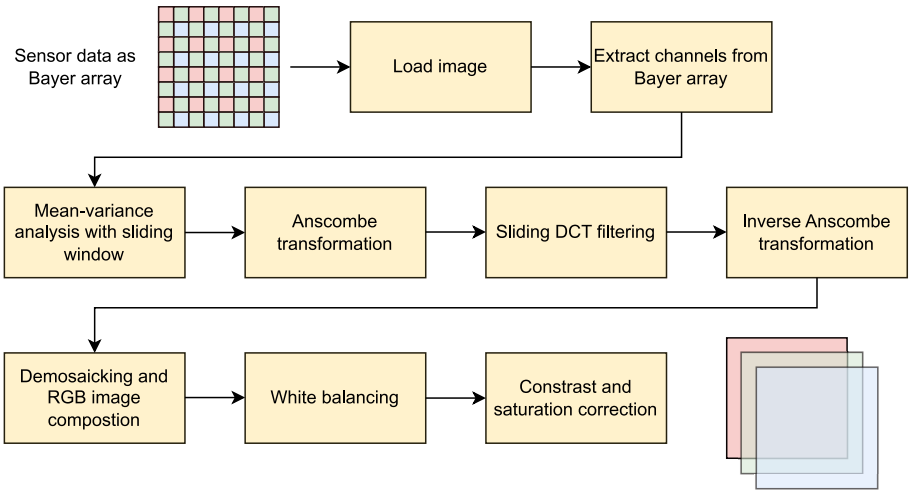


Figure 2.1 *Our image processing and formation pipeline*

2.2 Reading images and converting to doubles

The first part of the pipeline is loading the images (outoffocus.tiff and natural.tiff) using Matlab's *imread* function, and then converting the pixel values to double precision by using *im2double* function. The convertion ensure that pixel values are represented as floating-point numbers between 0 and 1. This is necessary to minimize rounding errors during processing as opposed to using integers.

```
% 1. Load image and convert to double
focus = imread('images/outoffocus.tiff');
focus = im2double(focus);

natural = imread('images/natural.tiff');
```

```
6  natural = im2double(natural);
```

2.3 Image visualization and Bayer mosaic

The second part starts with creating a figure with two subplots. The *imshow* function is used with [] to scale the pixel values automatically for better visualization. For illustrating the Bayer mosaic arrays the image is separated into its color channels (red, green 1, green 2 and blue). After separating the color channels, the next step is to create figures to illustrate the Bayer mosaic arrays for each color channel separately. This produces a kind of a intensity map for each color channel, which might be useful for visually analyzing how each color should appear in the final image.

```
1  % 2. Visualize Images, Bayer mosaic array
2  % Define the block size
3  figure;
4  subplot(1,2,1); imshow(focus, []); title('out of focus image');
5  subplot(1,2,2); imshow(natural, []); title('natural');
6
7  % 3. Separe image into subchannels
8  R_focus = focus(1:2:end, 1:2:end);
9  G1_focus = focus(1:2:end, 2:2:end);
10 G2_focus = focus(2:2:end, 1:2:end);
11 B_focus = focus(2:2:end, 2:2:end);
12
13 R_nat = natural(1:2:end, 1:2:end);
14 G1_nat = natural(1:2:end, 2:2:end);
15 G2_nat = natural(2:2:end, 1:2:end);
16 B_nat = natural(2:2:end, 2:2:end);
17
18 % Illustrare the bayer array
19
20 % Plot channels
21 figure;
22 subplot(2,2,1); imshow(R_focus, []); title('Red');
23 subplot(2,2,2); imshow(G1_focus, []); title('Green 1');
24 subplot(2,2,3); imshow(G2_focus, []); title('Green 2');
25 subplot(2,2,4); imshow(B_focus, []); title('Blue');
26 sgtitle("Out of focus image");
27
28 figure;
29 subplot(2,2,1); imshow(R_nat, []); title('Red');
```

```
subplot(2,2,2); imshow(G1_nat, []); title('Green 1');
subplot(2,2,3); imshow(G2_nat, []); title('Green 2');
subplot(2,2,4); imshow(B_nat, []); title('Blue');
sgtitle("Natural image");
```

2.4 Sliding window

In this step we analyze each subchannel of the out-of-focus and natural images using a sliding window operator. For both natural image and out-of-focus image we are calling `calculateScatterPlot` function for each color channel.

The function defines a sliding window analysis using the `blockproc` function in Matlab. The window size is set to $[2, 2]$, and $calculateMeanVar$ function is defined to calculate the mean and variance for each window. The function $calculateScatterPlot$ returns mean values and variance values that we need in the next step when calculating regression lines.

The purpose of this step is to approximate the Poisson noise present in the sensor data, which needs to be minimized before Gaussian denoising.

```
% 4. Calculate scatter plots
function [meanValues, varianceValues] = calculateScatterPlot(
    channel)
    % Define the window size for sliding window analysis
    windowSize = [2, 2];

    % Define a function to calculate mean and variance for each
        window
    calculateMeanVar = @(block_struct) [mean(block_struct.data
        (:)), var(block_struct.data(:))];

    % Apply the sliding window operator using blockproc
    meanVarResults = blockproc(channel, windowSize,
        calculateMeanVar);

    % Extract mean and variance values
    meanValues = meanVarResults(:, 1);
    varianceValues = meanVarResults(:, 2);

    % Reshape the results for scatterplot creation
    meanValues = meanValues(:);
    varianceValues = varianceValues(:);
end
```

2.5 Scatterplots and regression

In this part we calculate the regression lines for each subchannel using the `calculateRegression` function. Regression is calculated by *polyfit* function in Matlab, that returns a best-fitting polynomial to a set of data points, in this context it is used to fit a straight line to the data. The regression calculation will be done for both out-of-focus and natural subchannels.

```
% 5. Calculate regression lines
function [p] = calculateRegression(meanValues, varianceValues)
    % Fit a straight line to the data
    p = polyfit(meanValues, varianceValues, 1);
end
```

Then, `produceScatterPlot` function is responsible for generating scatter plots for each color channel of the out-of-focus image. This function takes as input the mean values and variance values obtained from the `calculateScatterPlot` function, as well as the regression lines calculated by the `calculateRegression` function. After running this function, you should have scatter plots with regression lines for each subchannel.

```
function produceScatterPlot(meanR, varR, pR, meanG1, varG1, pG1,
    meanG2, varG2, pG2, meanB, varB, pB)
    % Create a new figure
    figure;

    % Fit a straight line to the Red channel data and plot it
    subplot(2,2,1);
    scatter(meanR, varR);
    xlabel('Local Sample Mean');
    ylabel('Local Sample Variance');
    title('Mean-Variance Scatterplot for Red');
    hold on;
    fR = polyval(pR, meanR);
    plot(meanR, fR, 'r');
    hold off;

    % Fit a straight line to the Green 1 channel data and plot
       it
    subplot(2,2,2);
    scatter(meanG1, varG1);
    xlabel('Local Sample Mean');
    ylabel('Local Sample Variance');
```

```
21      title('Mean-Variance Scatterplot for Green 1');
22      hold on;
23      fG1 = polyval(pG1, meanG1);
24      plot(meanG1, fG1, 'g');
25      hold off;
26
27      % Fit a straight line to the Green 2 channel data and plot
            it
28      subplot(2,2,3);
29      scatter(meanG2, varG2);
30      xlabel('Local Sample Mean');
31      ylabel('Local Sample Variance');
32      title('Mean-Variance Scatterplot for Green 2');
33      hold on;
34      fG2 = polyval(pG2, meanG2);
35      plot(meanG2, fG2, 'g');
36      hold off;
37
38      % Fit a straight line to the Blue channel data and plot it
39      subplot(2,2,4);
40      scatter(meanB, varB);
41      xlabel('Local Sample Mean');
42      ylabel('Local Sample Variance');
43      title('Mean-Variance Scatterplot for Blue');
44      hold on;
45      fB = polyval(pB, meanB);
46      plot(meanB, fB, 'b');
47      hold off;
48  end
```

2.6 Transformation and inverse transformation

To be able to perform Gaussian denoising we first need to remove the noise caused by the sensor. This noise is inherent to each sensor and thus is different for each image [**poissonnoise2011**]. The most common approximation of this sensor noise is with Poisson noise which can be obtained from local mean-variance information which we did in the previous step. After obtaining the information about the Poisson noise, we can use the Anscombe transformation 3.1 to remove the percieved noise.

After denoising the image needs to be applied inverse transformation to reintroduce the sensor noise. Section 3.2 explores why the inverse function is really not the exact inverse function of the forward function.

2.6.1 Tranformation

Transformation is applied to each subchannel of the natural and out-of-focus images. Here is an example how the out-of-focus image's R-subchannel transformation is called in the code:

```
transformedR_focus = blockproc(R_focus, transformBlockSize, @(
    block_struct) applyTransformation(block_struct, pR_focus(1),
     pR_focus(2)));
```

The transformation uses Matlab's `blockproc` function for block processing and calls `applyTransformation` function inside the blockproc. The `applyTransformation` function calculates the transformed value for each block using the equation

$$2\sqrt{\frac{I_c}{a_c} + \frac{3}{8} + \frac{b_c}{a_c^2}} \tag{2.1}$$

where a_c and b_c are the parameters estimated in calculating regression. The function handles cases where the value inside the square root becomes negative, ensuring a non-negative result. The transformed blocks are then concatenated to produce the final transformed images back in `blockproc` function.

```
% 6. Transformation
function output = applyTransformation(block_struct, ac, bc)
    % Calculate the value inside the square root
    sqrt_val = (block_struct.data / ac) + (3/8) + (bc / (ac^2));

    % Check if the value inside the square root is negative
    sqrt_val(sqrt_val < 0) = 0;

    % Calculate the output
    output = 2 * sqrt(sqrt_val);
end
```

2.6.2 Inverse tranformation

Similar to the transformation, the inverse transformation is executed using the `blockproc` function:

```
inverseRT_nat = blockproc(filteredRT_nat, transformBlockSize, @(
    block_struct) inverseTransformation(block_struct, pR_nat(1),
    pR_nat(2)));
```

The inverse transformation is expressed by the equation:

$$a_c \left(\frac{1}{4} D^2 + \frac{1}{4} \sqrt{\frac{3}{2}} D^{-1} - \frac{11}{8} D^{-2} + \frac{5}{8} \sqrt{\frac{3}{2}} D^{-3} - \frac{1}{8} - \frac{b_c}{a_c^2} \right) \tag{2.2}$$

where D denotes the filtered transformed image. The `inverseTransformation` function calculates the inverse transformation value for each block using equation 2.2

```
%% 9. Inverse transformation
function output = inverseTransformation(block_struct, ac, bc)
    % Calculate the values inside the square roots
    sqrt_val1 = 0.25 * sqrt(3/2) * block_struct.data.^(-1);
    sqrt_val2 = (5/8) * sqrt(3/2) * block_struct.data.^(-3);

    % Check if the values inside the square roots are negative
    sqrt_val1(block_struct.data.^(-1) < 0) = 0;
    sqrt_val2(block_struct.data.^(-3) < 0) = 0;

    % Calculate the output
    output = ac * (0.25 * block_struct.data.^2 + ...
        sqrt_val1 - ...
        (11/8)*(block_struct.data.^(-2)) + ...
        sqrt_val2 - ...
        1/8 - ...
        bc/(ac^2));
end
```

2.7 DCT denoising

This part implements a denoising operation on an image using a sliding DCT (Discrete Cosine Transform) filter. The DCT image denoising applies to each subchannel of both the original and transformed versions of the natural image. The `DCTImageDenoising` function takes an image, a threshold parameter λ and the block size for the DCT operation. For the threshold parameter λ we selected 0.060 for getting the most visually convincing result. The block size used is [8 8]. The `thresholdDCT` function applies the DCT to a given block of the image, thresholds the DCT coefficients by setting values below a threshold λ to zero, and then returns the denoised block.

```
function [denoised] = DCTImageDenoising(image, lambda,
    transformBlockSize)

```

```
    % Create a custom function to be applied to each block
    fun = @(block_struct) idct2(thresholdDCT(block_struct.data,
        lambda));

    % Apply the function to each block
    denoised = blockproc(image, transformBlockSize, fun);
end

function denoised = thresholdDCT(input, lambda)

    % Apply DCT to the block
    dctBlock = dct2(input);

    % Threshold the DCT coefficients
    dctBlock(abs(dctBlock) < lambda) = 0;
    denoised = dctBlock;
end
```

2.8 Demosaicking

This section performs a simple demosaicking operation using linear interpolation. After running the function `simpleDemosaic`, it creates a full-color RGB image from the color channels by interpolating the missing color information between pixels in the Bayer mosaic pattern. For interpolation we use *interp*2 function in Matlab and nearest-neighbour method.

```
function [output] = simpleDemosaic(R, G1, G2, B)

    % Define output array;
    R_out = zeros(size(R)*2);
    G1_out = zeros(size(R)*2);
    G2_out = zeros(size(R)*2);
    B_out = zeros(size(R)*2);

    % Get the size of the images
    [M, N] = size(R_out);

    % Create a meshgrid for interpolation
    [Xout, Yout] = meshgrid(1:N, 1:M);

    % Shift the grids for each channel according to the Bayer
        pattern
```

```
    [Xr, Yr] = meshgrid(1:2:N, 1:2:M);
    [Xg1, Yg1] = meshgrid(1:2:N, 2:2:M);
    [Xg2, Yg2] = meshgrid(2:2:N, 1:2:M);
    [Xb, Yb] = meshgrid(2:2:N, 2:2:M);

    % Interpolate the images
    r_interp = interp2(Xr, Yr, R, Xout, Yout, 'nearest', 0);
    g1_interp = interp2(Xg1, Yg1, G1, Xout, Yout, 'nearest', 0);
    g2_interp = interp2(Xg2, Yg2, G2, Xout, Yout, 'nearest', 0);
    b_interp = interp2(Xb, Yb, B, Xout, Yout, 'nearest', 0);

    g_interp = (g1_interp + g2_interp)/2;

    output = cat(3, r_interp, g_interp, b_interp);
end
```

2.9 White balancing

The purpose of white balancing is to correct the color cast in an image so that the white color appears neutral. The `whiteBalance` function implements basic white balancing method by identifying the pixel with maximum intensity in the image and and then normalizes the entire image by dividing each RGB pixel by the RGB values of the maximum intensity pixel.

```
function balancedImage = whiteBalance(image)

% Check the range of the image data
minVal = min(image(:));
maxVal = max(image(:));

% If the range is not [0, 1], scale the image data
if minVal < 0 || maxVal > 1
    image = (image - minVal) / (maxVal - minVal);
    max(image, [], 'all')
end

% Convert to HSV
hsvImg = rgb2hsv(image);

% Find location of maximum intensity from the hsv representation
maxPixelValue = max(hsvImg(:,:,3), [], 'all');
[x,y] = find(hsvImg(:,:,3)==maxPixelValue, 1, 'first');
```

```
19
20 % Divide all the pixel by the found pixel in rgb space
21 balancedImage(:,:,1) = image(:,:,1) ./ image(x,y,1);
22 balancedImage(:,:,2) = image(:,:,2) ./ image(x,y,2);
23 balancedImage(:,:,3) = image(:,:,3) ./ image(x,y,3);
24
25 % Prevent values over limits
26 balancedImage = min(max(balancedImage, 0), 1);
27
28 end
```

2.10 Contrast and saturation correction

The last part of our pipeline involves contrast and saturation correction. The correction is applied using the `contrastAndSaturationCorrection` function and takes the image and a gamma value as inputs. The gamma value influences the degree of saturation correction, allowing a visual enhancement. For the correlation, the image is converted to the HSV color space, because then the function can focus on contrast adjustment in the V channel and saturation correction in the S channel.

```
1  function img_corrected = contrastAndSaturationCorrection(img,
       gamma_value)
2      % Convert the image to the HSV color space
3      img_hsv = rgb2hsv(img);
4
5      % Perform histogram equalization on the V channel for
           contrast correction
6      img_hsv(:,:,3) = histeq(img_hsv(:,:,3));
7
8      % Perform gamma correction on the S channel for saturation
           correction
9      img_hsv(:,:,2) = img_hsv(:,:,2).^gamma_value;
10
11     % Convert the image back to the RGB color space
12     img_corrected = hsv2rgb(img_hsv);
13 end
```

3 Results

In this section we will present the results from our image processing pipeline and also do some analysis on why we got the kinds of results that we did.

3.1 Bayer array

The image formation pipeline start by reading the sensor data. Since the sensor data is in interlaced RGGB format we cannot simply display it as RGB color image like we would usually do, but we can still display it as grayscale image, from which we can observe the bayer array. Figure 3.1 shows the starting images used in this assignment. Figure 3.2 illustrated the bayer array which was obtained by zooming closely in to the Natural image.

(a) Out of focues image *(b) Natural image*

Figure 3.1 *Source images probided by the assignment instructors as grayscale RGGB images*

3.2 Purpose of the Anscombe transformation

In image processing Poisson distribution is used to model the signal dependant noise specific in images, due to it better model photons hitting the sensor array then Gaussian distribution. Gaussian distribution is signal independant and thus more suitable for general purpose denoising methods, so we want to transform our image specific Poisson noise to corresponding Gaussian distribution for processing. This is accomplished with the Anscombe transformation. After denoising we can use the inverse Anscombe transformation to bring the image back to Poisson noise distribution where the image should now appear denoised.

Equation 3.1 shows the Anscombe transformation and equation 3.2 shows the function used to inverse the Anscombe transformation. Where a_c and b_c describe the

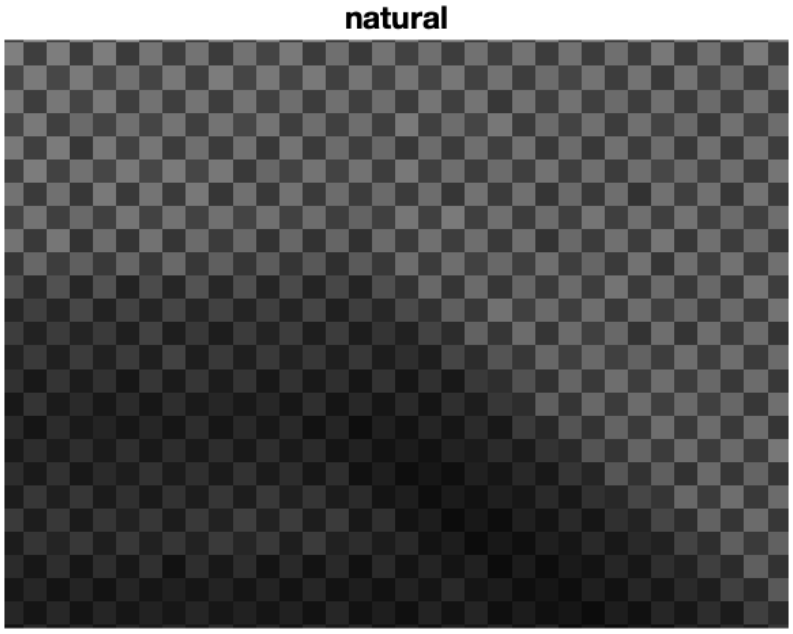


Figure 3.2 *Sample of bayer array from the Natural image*

variance of noise in the image and D denotes the denoised image. For the purposes of this exercise the inversing function is not exactly the same as inverse Anscombe tranformations. According to [**poissonnoise2011**] and [**poissonnoise2011-2**] this is because for images with low intensity (lesser frequency) the precise inverse function performs poorly due to bias caused by the non-linear forward function.

$$2\sqrt{\frac{I_c}{a_c} + \frac{3}{8} + \frac{b_c}{a_c^2}} \tag{3.1}$$

$$a_c\left(\frac{1}{4}D^2 + \frac{1}{4}\sqrt{\frac{3}{2}}D^{-1} - \frac{11}{8}D^{-2} + \frac{5}{8}\sqrt{\frac{3}{2}}D^{-3} - \frac{1}{8} - \frac{b_c}{a_c^2}\right) \tag{3.2}$$

3.3 Mean variance analysis

With a sliding window mean and variance operators we could extract local noise information from the out of focus image to visualize how the Anscombe transformation affects the image. Figure 3.3 shows the local means and variances for each channel as datapoints and the affine variance as colored lines before the Anscombe transformation. From the lines we see that they are not flat like we would expect. Figure 3.4 shows the same image after the Anscombe transformation. We can now see that the regressions lines are flat. This is because the Anscombe transformation has made the signal dependant noise (Poisson noise) a constant troughout the image.

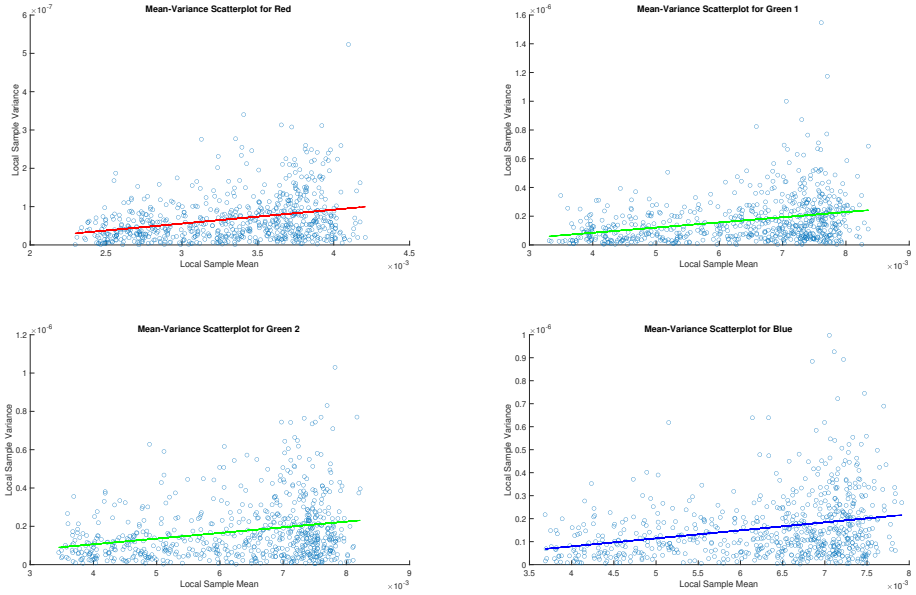


Figure 3.3 *Mean-variance for each channels before denoising*

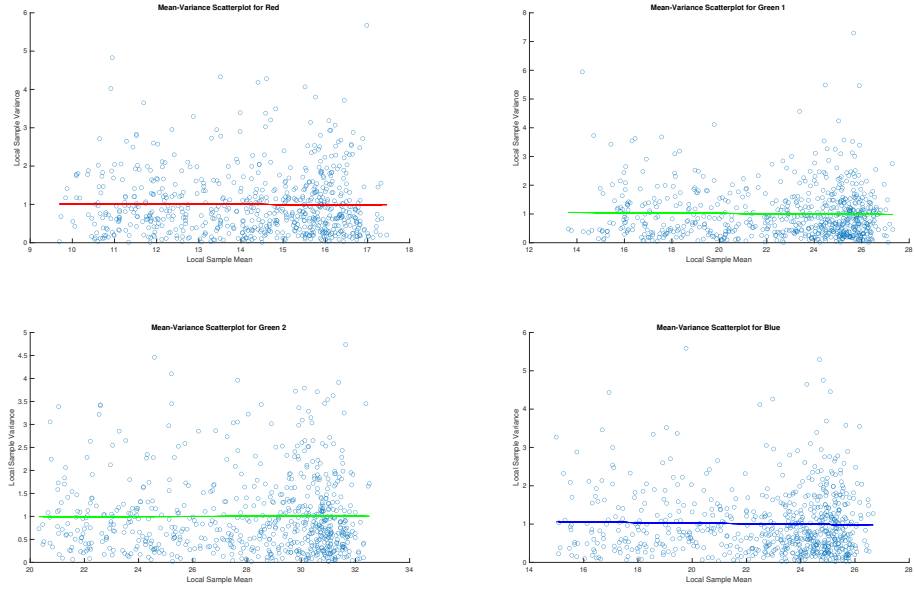
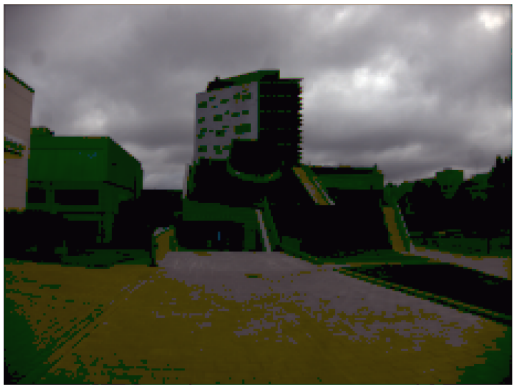


Figure 3.4 *Mean-variance for each channels after denoising*

3.4 Comparing non-transformed and transformed images

After combining the channels to a RGB image we can compare the images produced when Poisson noise was removed and when it was not. Figure 3.5 shows the images produced by the pipeline without and with the Anscombe transformation. From these images we can see that the transformed image is clearly more detailed. From this we can conclude that the transformation greatly benefits the image formation pipeline, when using Bayer array sensor data.



(a) Produced image withouth Anscombe transformation

(b) Produced image with Anscombe transformation

Figure 3.5 *Comparision of images where (a) was not applied Anscombe transformation before reconstruction, and (b) as applied Anscombe transformation.*

3.5 Color correction

After producing RGB we applied white balancing, saturation correction and contrast correction. Figure 3.6 shows the difference between the images that have not and were applied Anscombe transformation before RGB composition after white balancing, saturation and contrast corrections. The non-transformed image is a lot brighter but it is still unclear. The transformed image still looks good, as it's somewhat brighter. By eye it also looks like with the chosen lambda value the green color has been reduced somewhat, which might not be wanted.

(a) Produced image withouth Anscombe transformation after color correction

(b) Produced image with Anscombe transformation after color connection

Figure 3.6 *Comparision of images where (a) was not applied Anscombe transformation before reconstruction, and (b) as applied Anscombe transformation.*

4 Conclusions

In this laboratory work we familiarized ourselves with basics of digital image acquisition, formation and processing. The goal of this laboratory work was to analyze and process raw sensor data in Matlab, constructing an RGB image.

For the process we utilized some of Matlab build-in functions and also implemented our owns. After implementing all the steps from pipeline we were able to get realistic-looking RGB image as an output.