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 construcut 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.

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.

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

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

```
% 2. Visualize Images, Bayer mosaic array
% Define the block size
figure;
subplot(1,2,1); imshow(focus, []); title('out of focus image');
```

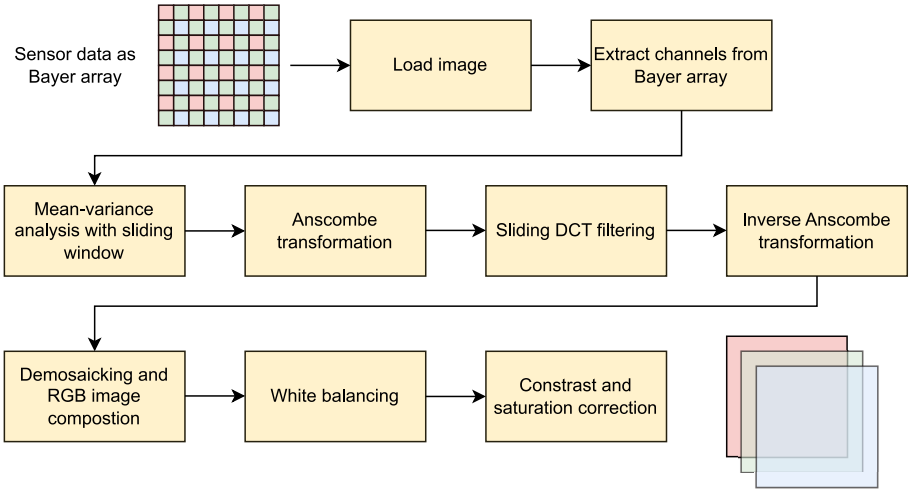

Figure 2.1 *Our image processing and formation pipeline*

```
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');
30 subplot(2,2,2); imshow(G1_nat, []); title('Green 1');
31 subplot(2,2,3); imshow(G2_nat, []); title('Green 2');
32 subplot(2,2,4); imshow(B_nat, []); title('Blue');
```

```
33 sgtitle("Natural image");
```

2.4 Sliding window

In this section we are analyzing 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.

```
1
2 % 4. Calculate scatter plots
3 function [meanValues, varianceValues] = calculateScatterPlot(
    channel)
4     % Define the window size for sliding window analysis
5     windowSize = [2, 2];
6
7     % Define a function to calculate mean and variance for each
          window
8     calculateMeanVar = @(block_struct) [mean(block_struct.data
          (:)), var(block_struct.data(:))];
9
10    % Apply the sliding window operator using blockproc
11    meanVarResults = blockproc(channel, windowSize,
          calculateMeanVar);
12
13    % Extract mean and variance values
14    meanValues = meanVarResults(:, 1);
15    varianceValues = meanVarResults(:, 2);
16
17    % Reshape the results for scatterplot creation
18    meanValues = meanValues(:);
19    varianceValues = varianceValues(:);
20 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');
    title('Mean-Variance Scatterplot for Green 1');
    hold on;
    fG1 = polyval(pG1, meanG1);
    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

[kesken]

Transformation is done by using this equation

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

, where a_c and b_c are the parameters estimated from the out-of-focus image.

```
1  % 6. Transformation
2  function output = applyTransformation(block_struct, ac, bc)
3      % Calculate the value inside the square root
4      sqrt_val = (block_struct.data / ac) + (3/8) + (bc / (ac^2));
5
6      % Check if the value inside the square root is negative
7      sqrt_val(sqrt_val < 0) = 1;
```

```
8
9       % Calculate the output
10      output = 2 * sqrt(sqrt_val);
11  end
```

The inverse transformation is done by using the equation

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

```
1  %% 9. Inverse transformation
2  function output = inverseTransformation(block_struct, ac, bc)
3      % Calculate the values inside the square roots
4      sqrt_val1 = 0.25 * sqrt(3/2) * block_struct.data.^(-1);
5      sqrt_val2 = (5/8) * sqrt(3/2) * block_struct.data.^(-3);
6
7      % Check if the values inside the square roots are negative
8      sqrt_val1(block_struct.data.^(-1) < 0) = 0;
9      sqrt_val2(block_struct.data.^(-3) < 0) = 0;
10
11     % Calculate the output
12     output = ac * (0.25 * block_struct.data.^2 + ...
13         sqrt_val1 - ...
14         (11/8)*(block_struct.data.^(-2)) + ...
15         sqrt_val2 - ...
16         1/8 - ...
17         bc/(ac^2));
18 end
```

2.7 DCT denoising

[kesken] This part implements a denoising operation on an image using a sliding DCT (Discrete Cosine Transform) filter.

```
1  function [denoised] = DCTImageDenoising(image, lambda,
       transformBlockSize)
2
3      % Create a custom function to be applied to each block
4      fun = @(block_struct) idct2(thresholdDCT(block_struct.data,
           lambda));
5
6      % Apply the function to each block
7      denoised = blockproc(image, transformBlockSize, fun);
```

```
8  end
9
10 function denoised = thresholdDCT(input, lambda)
11
12     % Apply DCT to the block
13     dctBlock = dct2(input);
14
15     % Threshold the DCT coefficients
16     dctBlock(abs(dctBlock) < lambda) = 0;
17     denoised = dctBlock;
18 end
```

2.8 Demosaicking

2.9 White balancing

2.10 Contrast and saturation correction

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 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. For the purposes of this exercise the inversing function is not exactly the same as inverse Anscombe tranformations. This is because TODO: miksi näin on?

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

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

4 Conclusions