Learning objectives

- Learn how to generate RGB image from raw image data.
- Learn how different interpolation methods can be implemented.
- Learn how to analyze the outcome of different interpolation methods.
- Improve your skills in writing technical reports.

Organization and overview of tasks

The aim of this work is to implement three interpolation methods to create a full RGB image from raw Bayer data. Data are provided in the course Moodle page. Group size for this assignment is two students.

Basic knowledge of Python and programming is mandatory to succeed in this work. However, guidance for the work will be provided during the project according to requests. The instructor for this work is Jani Björklund. If you have questions, you are welcome to send an email to `jani.bjorklund@tuni.fi`.

Use Python version 3.x and common packages (e.g. numpy, scikit-image, scikit-learn, matplotlib, scipy). In addition, you will need PIL and time packages.

Overall the work can be divided in three parts:

I. Implementation of the interpolation methods (60%)

II. Testing the outcome of the methods (20%)

III. Documentation (20%)

Raw data processing

To use the raw image data provided in Moodle and to proceed with the rest of the assignment, you need to implement the following steps as a function:

1. Open the file with `rawData = open(filename,'rb').read()`.
2. Read the data from bytes with PIL.Image.frombytes() function and transform to numpy array:
```
from PIL import Image
import numpy as np
rawImage = Image.frombytes('F',imSize,rawData,'raw',imType)
rawImage = np.asarray(rawImage)
```
The image size for raw_image2 and raw_image5 is [1008 1018] and data type to read is int16 (parameter `imType='F;16'`), but the values are 10bit little endian. The image size for testikuva.raw is [512 512] and the data type is uint8 (parameter `imType='F;8'`) with 8 bit little endian values.
3. The values of raw_image2 and raw_image5 must be divided by $2^{10}-1$ and those of testikuva.raw by $2^8 - 1$ to normalize them. After this, the values of `rawImage` should be double precision in the range [0, 1].

4. Note that the image is transposed in the reading process; thus, the size of `rawImage` will become [1018, 1008]. After you have read the image as numpy array, you should always use `rawImage.shape` if you need to access image size (not the imSize parameter).

5. Pick the pixel values to the R, G and B layer matrices from the read data. The original data in `rawImage` are grayscale values, some representing red, some green and some blue pixels, organized in the order visualized in Figures 1 and 4. Final R, G and B matrices should contain a grid of known intensity values picked from `rawImage` and missing values in between containing zeros (Figure 2, note that R, G and B are shown here in color but they are actually grayscale). *Tip: initiate R, G and B as matrices of zeros with correct dimensions (same as `rawImage`), then go through the entire raw image and pick the red pixels from `rawImage` to R, green pixels to G and blue pixels to B using either for-loops or clever indexing.*

6. The function call should be: `[R,G,B] = readimagefile(filename,imSize,imType)`.

Theory

Most digital cameras save the data in RGB format, where all pixel values are formed with a color vector of red (R), green (G) and blue (B) components. These components are measured with the camera sensor by separated color cells. A single cell (pixel) of a sensor can only measure one intensity value at a time, without any color information.

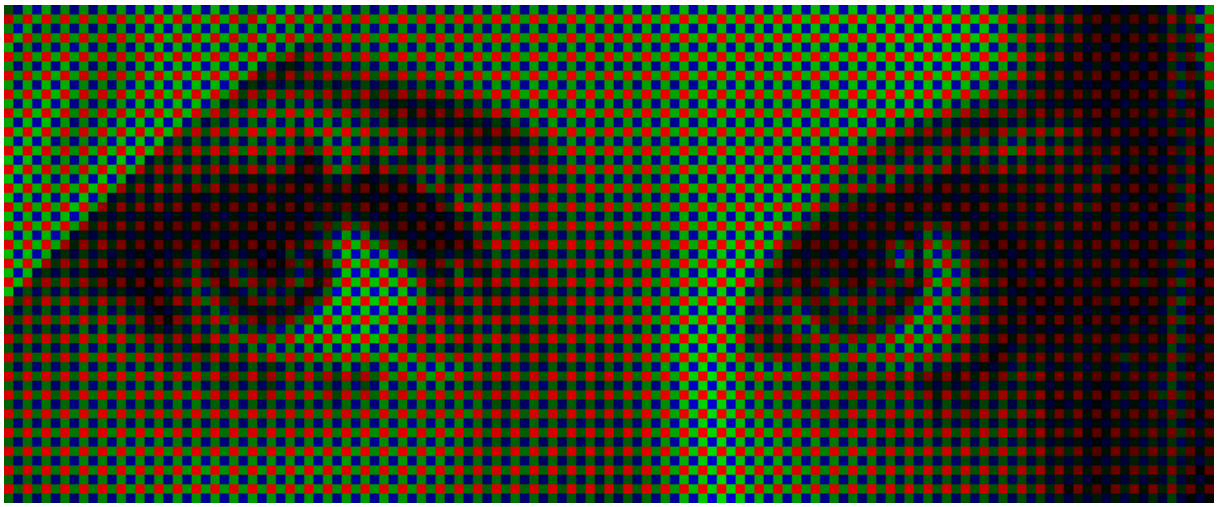

Figure 1: Raw data from Bayer CFA visualized without interpolation

To measure the three main colors, separated color filter arrays (CFA) are used with the sensor. On top of each cell (pixel), there will be a single color filter allowing either red, green or blue light to pass to the sensor. The most common CFA is the Bayer filter, where red, green and blue filters are in group of four cells with two green and single red and blue cells (see raw data at Figures 1 and 2). The green color is measured with two cells instead of one, as the human eye is more sensitive to the green color. The amount of cells in the sensor defines the amount of pixels in the output image. Thus in the raw data there is only one color value for each pixel. For example, in a 16 Mpix camera sensor the number of pixels can be 4928×3264.

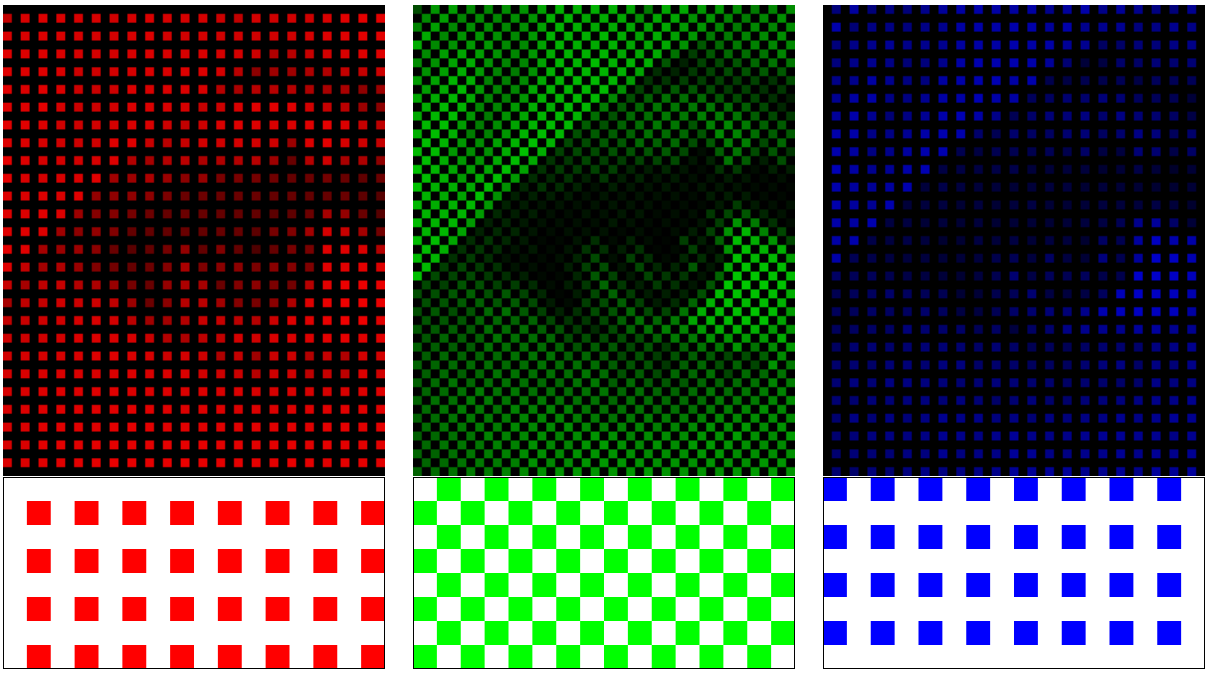

Figure 2: Separated colors with the corresponding CFA

To be able to create a full RGB image from the sensor data, the two missing color values of each pixel must be interpolated from surrounding pixels. Commonly the interpolation is done in a 3×3 window over the pixel. Multiple interpolation methods exist for this task and the interpolation step has a significant effect on the output quality. For example, Figure 3 presents the results of nearest neighbor and bilinear interpolation methods. In common compact cameras the interpolation is often implemented in the hardware and the camera saves only the RGB-images in some common compressed image format like jpeg. In industry, the interpolation procedures can be implemented with external software for the raw data depending on the final usage of the image. DSLR cameras and more advanced compact cameras often allow selecting the raw data format, which enables external raw image processing.

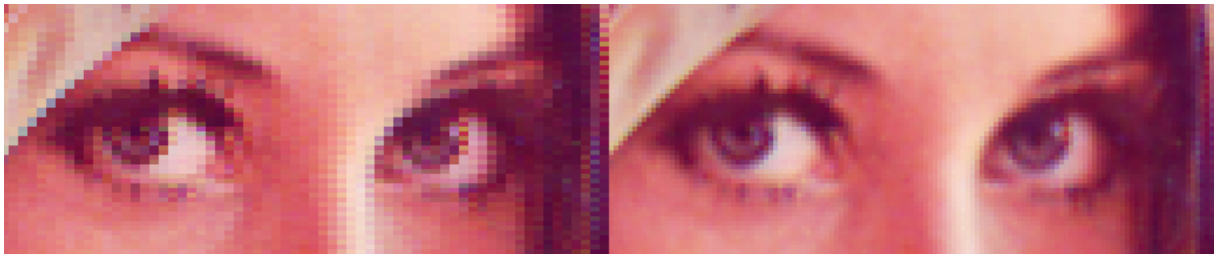

Figure 3: Nearest neighbor (left) and bilinear interpolation (right) results from the same data

1 Nearest neighbor interpolation

B1	G2	B3	G4
G5	R6	G7	R8
B9	G10	B11	G12
G13	R14	G15	R16

Figure 4: 4×4 Bayer filter matrix for RGB filter array

The nearest neighbor (NN) interpolation is the simplest interpolation method. It has two advantages over the other methods: it is simple to implement and it is computationally fast. As a negative side, the quality of the result is lesser than with more advanced methods.

Figure 4 presents the Bayer matrix of the used data. The pixels in Figure 4 are numbered from 1-16 and named according to the Bayer filter. At the starting point only one value (red, green or blue) is known for each pixel. After interpolation each pixel should have all three RGB-values. In the following the pixels are named according to number and corresponding color, for example B1 is the blue value of pixel in the location 1.

In principle NN-interpolation is performed in 2×2 windows: The blue values in the corresponding locations of G2, G5 and R6 are defined by copying them from the pixel B1. The red values for locations B1, G2 and G5 are formed similarly by copying them from R6. The green pixel in B1 is copied from G2 and for pixel R6 the green value is copied from G5. The process is repeated in all 2×2 non-overlapping windows over the image. After the whole image is processed, there are three color layers which all have the same size as the original raw image. By combining these layers to one $N \times M \times 3$ RGB matrix we have a full RGB color image. The final uint8 image should have values in the range [0, 255].

2 Bilinear interpolation

Bilinear interpolation is one step more complex than NN-interpolation. In bilinear interpolation, pixel values are computed via linear interpolation from surrounding pixels, rather than direct copying. This prevents the pixel artifacts, which are clearly visible in the NN-interpolation results. In bilinear interpolation the border pixels do not have all necessary pixels, which must be dealt with somehow. One option is to forget the border pixels, which creates smaller images to output, the second option is to add zeros to surroundings of the image (zero padding) and so fill the missing pixels, third option is to mirror them from the border pixels and then use these values for interpolation. Here you can select the method freely by your own choice.

The bilinear interpolation procedure is as follows: in the case of green pixels the value is always the average from surrounding four green pixels. For example G6 = (G2 + G5 + G7 + G10) / 4. The blue and red pixels are interpolated in similar manner as an average of the neighbouring two or four pixels, depending on the location within the Bayer matrix. For example, B2 = (B1+B3)/2 and B5 = (B1 + B9)/2 are computed as the average of two pixels. The average of four pixels is used to compute, for example, B6 = (B1 + B3 + B9 + B11) / 4. This procedure is then repeated over the whole image and results are combined to an RGB image as in the NN-interpolation. The final uint8 image should have values in the range [0, 255].

3 Patterned pixel grouping interpolation

Patterned pixel grouping (PPG) interpolation [1] is an advanced interpolation method which is designed to maintain high image quality and minimize interpolation artifacts in the output. It is used for example in the raw image processing software Ufraw [2]. Implement the PPG interpolation according to the instructions of [1]. As in the case of bilinear interpolation, you have to handle the border pixels using zero padding or some other approach to avoid indexing outside of the image array.

When implementing the algorithm, you should note that instructions of [1] assume that the pixel in top left corner is **red**, not **blue** as we have here in our data, so you have to deviate from the instructions accordingly.

Also note, that the part "Computing Red and Blue Values at Green Pixels" only explicitly handles green pixels on even rows where green pixels are flanked by blue pixels on the left and right hand sides and by red pixels on the top and bottom (i.e., odd rows in our case). When applying this step on odd rows (i.e. even rows in our case), where green pixels are flanked by red pixels on the left and right hand sides and blue pixels on the top and bottom, you have to switch the blue and red arrays in the two function calls to *hue_transit* presented in the instructions to get the coordinates right.

Again, the final uint8 image should have values in the range [0, 255].

Result quality measurement

The first quality measurement to use here is the Mean Squared Error (MSE), which is defined in Equation 1:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2, \tag{1}$$

where the $\hat{Y}_i$ is the i^{th} value of the ground truth and Y_i is the i^{th} value of the result image of the current method.

The second measurement is the Mean Absolute Error (MAE), which is defined in Equation 2:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i|, \tag{2}$$

where the f_i is the i^{th} value of the ground truth and y_i is the i^{th} value of the result image from the current method. In both methods, it is critical that all images have the **same size**, are in the **same scale [0-255]** and the computations are done in **double precision**. Compute the MSE and MAE measures over all pixels, rather than separately for different color channels.

Required tasks

1. Implement the *readimagfile* function to read raw images into correct format.
2. Implement all three interpolation methods and process the images raw_image2 and raw_image5 with each method. Visualize a magnified area from each of the final RGB images processed using different methods to allow visual comparison.
3. Compute MSE and MAE measures for all interpolation methods using testikuva.raw as the image to process and testikuva.tiff as the ground truth.
4. Test also the computing times of all three of your implementations with raw_image2. In Python, you can get computation times with time.time() function:
```
import time
a = time.time()
#computations....
print(time.time() - a,'seconds have passed.')
```
5. Present the results of the three interpolation methods either as a table or using three bar charts: one for MSE, one for MAE and one for the computation times. Discuss the results of the quality measurements and computing times in your report. Report also the amount of time you used to do the whole image processing laboratory work.

Tip: use the `sharex` *and* `sharey` *parameters to sync multiple subplots side by side, as in: plt.figure(); ax1 = plt.subplot(121); plt.imshow(Im); ax2 = plt.subplot(122,sharex=ax1,sharey=ax1); imshow(Im2);.*

Returning your submission

The implementation must have **a single main.py script** which can be simply run without providing any user input. The script should take care of calling the functions where the interpolations and tasks are implemented and of computing and visualizing the required results. Each of the different interpolation methods should be implemented in **a separate Python function**. If the implementation of some method needs more functions than one, the other functions should be implented as subfunctions inside the corresponding main function. Don't use absolute paths: assume that images are in the same folder as your main script and functions.

The program code should be easy to read, fast to run and also intuitive to analyze and debug, if the results are not what was expected. This means that the code should be well commented and also use descriptive variable names. Please ensure that your solution works in Python 3.x,

with Anaconda environment where only packages mentioned in these instructions are installed. Take care that no errors are shown in any situation of use.

Please write the report in **PDF format** using the course template such that it is printer friendly and user-friendly. Avoid adding too many images. The number of words is not important, but the content is. Keep in mind the questions:

- Does this report explain and visualize my project achievements in such a way that I would like to read it through later and would also understand it then again?
- Could I repeat the work without any planning just with the report and instructions?

Return your report and code in a single ZIP-file via Moodle. **Strict deadline for the exercise is 7th of April, 2019 at 23:55.**

References

[1] `https://sites.google.com/site/chklin/demosaic/`

[2] `http://ufraw.sourceforge.net/`

Extra material

[3] `http://en.wikipedia.org/wiki/Color_filter_array`

[4] `http://en.wikipedia.org/wiki/Mean_squared_error`

[5] `http://en.wikipedia.org/wiki/Mean_absolute_error`