McMaster University

COMPENG 4TN4 - Image Processing

Liam Luimes

400322938

luimesl@mcmaster.ca

https://github.com/LuimesLiam/demosaicing

Project Phase 1
Demosaicing

Due Date: February 18^{th}, 2024

Contents

1 Introduction

Many processes go into the construction of images taken on digital cameras, one of which in the image reconstruction pipeline is demosaicing. Demosaicing, also known as colour reconstruction, is an algorithm used to reconstruct a full-colour image from a colour filter overlay such as a Bayer filter [1]. There are various different approaches to demosaicing, from simple linear interpolation to neural networks. In this paper, a linear interpolation approach is first explored that simply averages surrounding pixels in a sliding window to estimate the missing RGB values. Then a directional filtering approach is implemented to attempt to solve pixel aliasing and artifacts [2]. A more advanced linear regression approach is also used to compare and contrast. The results of these implemented approaches are compared to the Matlab demosaic() function with a bilinear method selected as default [3]. After the demosaicing step, further image reconstruction is implemented such as simple histogram equalization using a CLAHE [4] approach and white balancing using gray world approximation [5]. These results are discussed and comments are made on how to increase the quality of their results in phase 2.

Please note, the Github repository will remain private until after the completion of this class as to not allow for students in the same class to copy this work.

2 Background

2.1 Demosaicing

Demosaicing is a process used in digital image processing pipeline to reconstruct a full colour image from Bayer patterns. A digital camera will collect the data for the photo using a colour filter array (CFA). This is essentially an array of photo sensors that have red, green, and blue filters to allow for those colours to only pass through and hit the photo sensors [6].

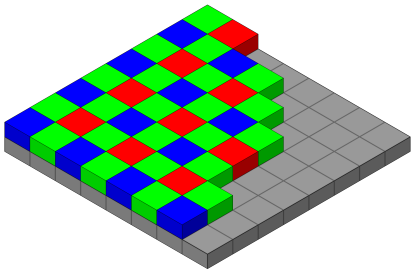


Figure 1: CFA

This raw image will have a mosaic of colours that appear discrete and uncoordinated. To get colour from this CFA image, we need to perform the demosaicing task in the pipeline. This demosaicing task can be done with various approaches, some of which are implemented in this paper. The main idea is to predict the missing R, G, or B value given the pixels in near proximity to the center missing pixel [7].

Figure 2: Raw CFA vs. Demosaiced Image

CFA typically comes in four pattern configurations- RGGB, BGGR, BGRG, and GRBG [7]. The demosaicing algorithm will have to know what pattern is being used before it can demosaic the image. The programs implemented in this paper have options to select between these four patterns.

2.2 White Balancing

White balancing ensures that colours appear natural and accurate in the image, regardless of the lighting conditions under which the photo was taken. It corrects for colour casts caused by different light sources (e.g., daylight, incandescent, fluorescent) that may introduce unwanted color tints [8]. One way to do this is to implement the "gray world approximation". The Gray World approach assumes that in an average scene, the average colour of the image should appear as a neutral gray. By adjusting the colour channels such that their average becomes gray, it aims to correct for any colour casts present in the image [5]. While it's a simple and intuitive method, it may not always produce accurate results in scenes where the assumption of an average gray world does not hold true, such as scenes dominated by a single colour or scenes with strong colour casts. Nonetheless, it serves as a basic and effective starting point for white balancing in many applications.

2.3 Histogram Equalization

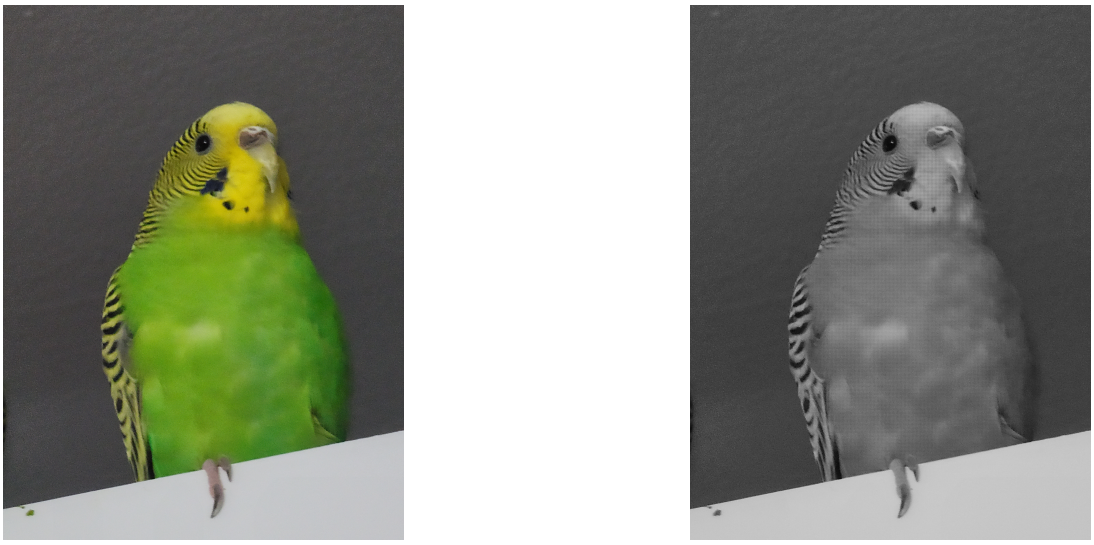
Histogram equalization is a technique used to enhance the contrast and visibility of an image by redistributing the intensity values of pixels [9]. It works by stretching the intensity range of the image histogram to cover a wider range of values, thereby increasing the dynamic range of the image and improving its overall appearance. Contrast Limited Adaptive Histogram Equalization (CLAHE) is a variant of histogram equalization that operates locally on small regions of the image. It divides the image into tiles or regions and applies histogram equalization independently to each tile. This approach prevents over-enhancement of local details by limiting the contrast enhancement within each tile, thus avoiding the introduction of artifacts while still improving the overall contrast of the image [4].

3 Building Dataset

To train and test the demosaicing algorithm, we must generate training and test data. Some options could be implementing a open-source library with mosaic images, or building a data set by simulating mosaic images. Simulated data lacks some aspects that may be helpful in training, such as noise or other undesired events that would have already been filtered out in the ground truth images. However, for this application, simulated data will suffice.

3.1 Code Description

Four different mosaic pattern modes can be selected. Each selected pattern specifies a different arrangement for sampling colour channels from the original RGB image. Each function works by iterating through the pixels of the RBG image and assigning colour values to the corresponding pixels of the mosaiced image according to the mode. Modes can be selected between CFA pattern types RGGB, BGGR, GBRG, and BGRG. See "Mosaicing Code" in the code section to see code snips 10.4, or see the mosaic.py file in the program folder.



(a) Ground Truth

(b) CFA Simulation

Figure 3: CFA simulation and ground truth

4 MatLab

This paper will compare the demosaicing methods implemented in this paper to the demosaicing function in Matlab from the computer vision toolbox [3]. Matlab uses bi-linear interpolation as its default method which will be used here [3]. This is a basic algorithm, so for more advanced cases the implementations in this paper should meet or surpass the quality output. To get the mean square error, the output Matlab demosaiced image will be saved in .png format and then compared in Python to the ground truth image using mean sqaure error code 10.1.

4.1 Bird Demosaicing

This picture 4 is the result of using the demosaic() function described above. This produces good results for a simple image like this, with a mean square error of just 2.806105415e-5.

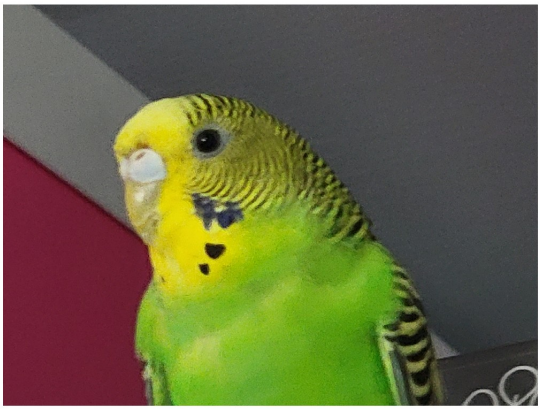


Figure 4: Matlab Bird Demosaicing

4.2 Truck

A more complicated image with thin lines and chrome may produce artifacts and less accurate results. This output can be seen to have strange colour artifacts on the chrome parts of the truck and the thin tool in the background. This demosaicing function produces a mean square error of 5.017278916e-3.

Figure 5: Matlab Truck Demosaicing

5 Linear Interpolation

A simple way to reconstruct a colour image from a raw Bayer image is to use linear interpolation. Implemented in this paper is a simple linear interpolation algorithm to test demosaicing. This algorithm estimates the colour value of the pixel by averaging two or four surrounding pixels of its colour [10]. The approach used here uses a matrix filter that is convolved over the image to interpolate the pixel values. Each colour channel is filtered individually and then stacked together at the end to produce the result. This is acceptable due to the regulation of the CFA Bayer pattern [10].

This algorithm uses G and RB matrices to interpolate the missing pixel. These matrices are simply filters used to select and then average the pixel values using an RGGB Bayer pattern.

$$G = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, RB = \frac{1}{4}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The algorithm first splits the input CFA image into the three colour channels (R, G, B) that can be interpolated. The above matrices are then applied by using the Python library skimage "convolve2" function. The green channel is simply applied directly and summed with the green channel to get the green results.

```
convg =convolve2d(g, k_g, 'same')
g = g + convg
```

The R and B channels then use the RB matrix shown above to partially interpolate their missing pixel values and store that result in a variable. Then another convolution is applied with the base channel and the stored partially interpolated value. The kernel used in this second convolution is the G matrix above and is done to further interpolate and refine the channel image. The same approach is done for the blue channel. Then the colour channels are stacked and normalized using a cv2 function.

```
convr1 =convolve2d(r, k_r_1, 'same')
convr2 =convolve2d(r+convr1, k_g, 'same')
r = r + convr1 + convr2

demosaiced_image = np.stack((r,g,b), axis=2)
demosaiced_image= cv2.normalize(demosaiced_image, None,
    alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
```

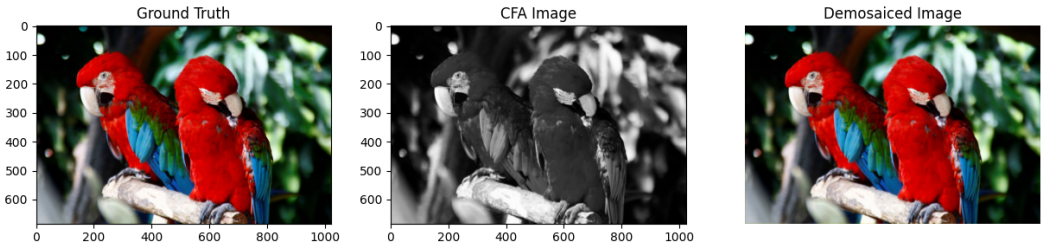
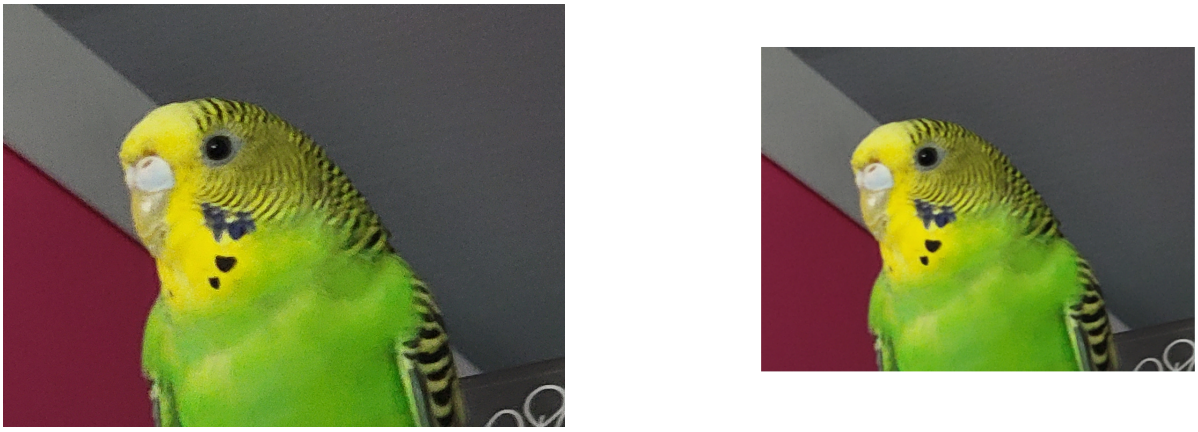


Figure 6: Linear Interpolation Results

5.1 Results

5.1.1 Bird Demosaicing

Testing this linear interpolation demosaicing method on a basic image shows acceptable results. The image in 7b shows great colours with little to no artifacts. The mean square error compared to ground truth is 1.74930e-4 which is acceptable. However, compared to the demosaic() Matlab function results for figure 4, this approach produces a result with 6 times worse mean square error. This is expected as this section only implements a simple linear interpolation.



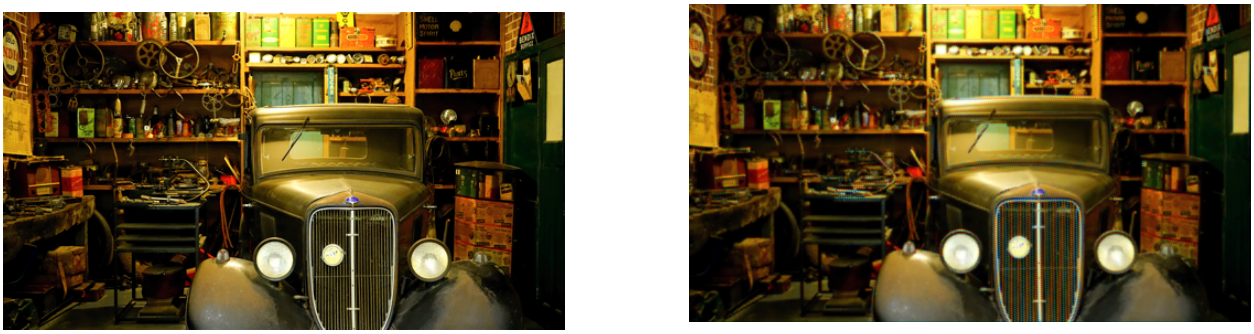
(a) Ground Truth Bird

(b) Demosaiced Bird

Figure 7: Linear Interpolation Demosaic

5.1.2 Truck Demosaicing

While this algorithm works well for simple images, there can be issues with artifacts and aliasing on more complex images. In the test CFA given, there is a chrome presence around the truck and a very strong horizontal edge along a tool in the background. This causes some very strange pixel effects shown in figure 8b. Although it is accepted that human visual systems are less sensitive to a change in colour than a change in brightness, these artifact's colours are irrelevant to the scene and stand out very clearly [2]. Furthermore, the mean square error between the ground truth and the demosaiced interpolated image is 0.009834899 as per the mean square error code 10.1. Compared to Matlab's demosaicing function in figure 5, this approach produces an error of about 1.8 times greater than Matlab. This is expected as this is a simple linear interpolation implementation.



(a) Ground Truth Truck

(b) Demosaiced Truck

Figure 8: Linear Interpolation Truck Results

6 Directional Filtering

Another approach implemented in this paper is the directional filtering by K. Hirakawa et al [2], resulting in a better demosaicing algorithm, especially when the image contains shiny chrome areas or sharp edges. As seen the in linear interpolation section, there tend to be pixel sparkling effects around the shiny chrome areas on figure 8b. This directional filter algorithm produces a better colour image than the linear interpolation above. The algorithm implemented here estimates missing pixels by interpolation in the direction with fewer colour artifacts to reduce this strange pixel artifact effect [11].

6.1 Algorithm Implementation

Here will be a walk through of this implementation done in python. For the full code, please see "direction_filtering_demosaicing.py" in the repository. Supporting functions such as "create_bayer_mask()" will not be discussed as they are not the focus of this paper.

```
red_mask, green_mask, blue_mask = create_bayer_masks(cfa_data.shape, bayer_pattern)

filter_basic = np.array([0.0, 0.5, 0.0, 0.5, 0.0])
filter_correction = np.array([-0.25, 0.0, 0.5, 0.0, -0.25])

red_channel = cfa_data * red_mask
green_channel = cfa_data * green_mask
blue_channel = cfa_data * blue_mask
```

Here create red, green, and blue mask pixel masks based on the input pattern that the user selects based on the CFA pattern. Here the filters that will be used for interpolation are defined. "filter_basic" is simply an average of the neighborhood green pixels, while "filter_correction" aims to adjust for the change in intensity of the pixels. The red, green, and blue channels are then extracted by using their respective masks.

```
green_horiz = np.where(green_mask == 0,
    horizontal_convolution(cfa_data, filter_basic)
    + horizontal_convolution(cfa_data, filter_correction), green_channel)
green_vert = np.where(green_mask == 0,
    vertical_convolution(cfa_data, filter_basic)
    + vertical_convolution(cfa_data, filter_correction), green_channel)

color_diff_horiz = np.where(red_mask == 1, red_channel - green_horiz,
    np.where(blue_mask == 1, blue_channel - green_horiz, 0))
color_diff_horiz = np.where(blue_mask == 1, red_channel - green_horiz,
    np.where(blue_mask == 1, blue_channel - green_horiz, color_diff_horiz))

color_diff_vert = np.where(red_mask == 1, red_channel - green_vert,
    np.where(blue_mask == 1, blue_channel - green_vert, 0))
color_diff_vert = np.where(blue_mask == 1, red_channel - green_vert,
    np.where(blue_mask == 1, blue_channel - green_vert, color_diff_vert))
```

For green channel interpolation, horizontal and vertical convolutions are applied to the entire image and then filtered with the green mask, where the actual green pixels are kept unchanged. The horizontal and vertical colour differences are calculated by subtracting the interpolated green channel from the red and blue channels.

```
smoothness_horiz = np.abs(color_diff_horiz -
    np.pad(color_diff_horiz, ((0, 0), (0, 2)), mode="reflect")[:, 2:])

smoothness_vert = np.abs(color_diff_vert -
    np.pad(color_diff_vert, ((0, 2), (0, 0)), mode="reflect")[2:, :])
```

```
smooth_kernel = np.array([
    [0, 0, 1, 0, 1],
    [0, 0, 0, 1, 0],
    [0, 0, 3, 0, 3],
    [0, 0, 0, 1, 0],
    [0, 0, 1, 0, 1]])

smoothness_horiz_conv = convolve(smoothness_horiz, smooth_kernel, mode="constant")

smoothness_vert_conv = convolve(smoothness_vert,
    np.transpose(smooth_kernel), mode="constant")

interp_direction_mask = smoothness_vert_conv >= smoothness_horiz_conv
```

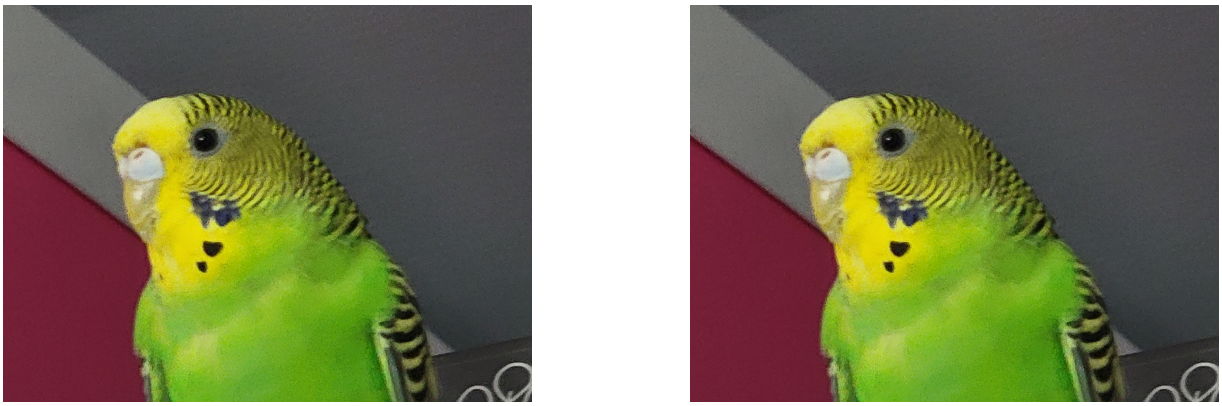
Then, the smoothness of these color differences is computed. This step is crucial for directional filtering, as it determines the direction in which interpolation will be less likely to introduce artifacts. The smoothness is a measure of how much change there is in the colour difference across the image, with less change indicating a preferred direction for interpolation. The smoothness in each direction is convolved with a smoothing kernel, which effectively compares the smoothness in the neighborhood of each pixel. The result of this convolution is used to create a mask that indicates whether vertical or horizontal interpolation is preferable at each pixel location.

The red and blue channels are then interpolated by applying directional filtering in both directions. the choice of direction for each pixel is based on the mask calculated in the code snippet above. After the R, G, and B values are calculated, they are then stacked, smoothed, and displayed.

6.2 Results

6.2.1 Bird Demosaicing

Testing on a relatively normal picture produces very accurate results. The mean square error to ground truth on figure 9 is extremely small at 2.1596496e-5. This shows an 8 times increase in accurate colour demosaicing compared to linear interpolation in section 5, which is substantial. Compared to the Matlab demosaicing function in figure 4, the mean square results are nearly the same, with only a 1.3 times increase in quality with reference to the mean square error. This is expected as this image is very simple, so Matlab should be able to handle it well.



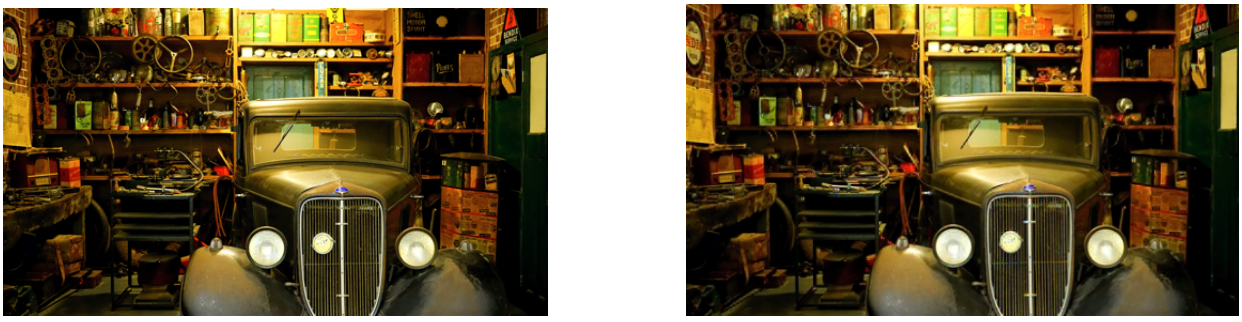
(a) Ground Truth Bird

(b) Demosaiced Bird

Figure 9: Directional Filtering Demosaic

6.2.2 Truck Demosaicing

As seen in this test image, the strange pixel artifacts are reduced and nearly gone due to the directional demosaicing. The mean square error has also dramatically reduced to 4.24733e-3 with reference to the ground truth. This is a 2.3 time increase in accuracy compared to the linear interpolation in figure 8, which is significantly better. Compared to Matlab's demosaicing function, this produces a mean square error of 1.20 times better. This may not seem like much, but comparing the visual results shows the lack of strange pixel artifacts around the horizontal tool and chrome areas.



(a) Ground Truth Truck

(b) Demosaiced Truck

Figure 10: Directional Filtering Truck Results

7 Linear Regression

As seen in the linear interpolation approach in section 5, one can convoluted a filter over the raw CFA image to interpolate pixel values. To get the best matrix for interpolating the image, linear regression can be used. In this approach, a 5x5 matrix for A and B is initialized with random numbers. This matrices serve as starting points and will be refined with linear regression. Various methods were tested when implementing linear regression, but a simple model using PyTorch regression was implemented. Earlier attempts implemented training until a threshold gradient vector length was researched, but it was found this over trained the model and was too cumbersome to train, so that method was abandoned.

7.1 Regression Model

The regression model works by splitting up the CFA image into its R, G, and B components. It then uses randomly initialized 5x5 (or any shape the user chooses) and convolves it over the single channel image. If the user has the option to input a custom matrix as a starting place. The model internalized its components to begin the regression. It also pads the image to allow for the model to convolve using the "same" settings so that the output image will be the same size as the input image. This model uses the PyTorch built in SDG optimizer. The user can choose how much of the image they want to train over. For example, to save time on training, the model is defaulted to train over the center 2/3's of the image (i.e. the focal point where the most detail should be).

```
input_height, input_width = input_images[k].shape
kernel_height, kernel_width = kernels[k].shape
pad_height = (kernel_height - 1) // 2
pad_width = (kernel_width - 1) // 2
padded_image = np.pad(input_images[k], ((pad_height, pad_height),
    (pad_width, pad_width)), mode='constant')
input_image = torch.tensor(input_images[k], dtype=torch.float32)
kernel = torch.tensor(kernels[k], dtype=torch.float32)

ground_truth = torch.tensor(ground_truths[k], dtype=torch.float32)
kernel.requires_grad = True
output = torch.zeros_like(input_image)
optimizer = optim.SGD([kernel], lr=gamma)

start_index = input_height // 3
end_index = 2 * (input_height // 3)
start_index2 = input_width // 3
end_index2 = 2 * (input_width // 3)
```

The matrix is then trained to interpolate the missing colour value and compared to a ground truth. The matrix is then updated using PyTorch gradient calculations combined with a learning rate. The gradient is determined and the regression model updates the matrix. This will continue for every window movement, and done for all three channels. In the implementation seen in "linear_regression.py", a second threaded while loop is used to take user input. The user can request to display the progress, save the data, move to the next colour channel, and various other tasks.

```
output[i, j] = torch.sum(torch.tensor(region, dtype=torch.float32) * kernel)
error = ground_truth[i, j] - output[i, j]
mse = torch.mean(error ** 2)
optimizer.zero_grad()
mse.backward(retain_graph=True)
optimizer.step()
```

The model will continue to loop through every image provided by the user and continue training the A and B matrices.

7.2 Demosaicing

This demosaicing algorithm is the simplest so far. It simply applies the learned A and B matrices to the image via a convolution to their respective colour channels, then stacked the results as a output.

```
R_m, G_m, B_m = masks_CFA_Bayer(CFA.shape, "RGGB")
R = CFA * R_m
G = CFA * G_m
B = CFA * B_m

red = convolve2d(R,RK,'same')
green = convolve2d(G, GK,'same')
blue =convolve2d(B,  RK,'same')
```

To further improve the image quality, a Gaussian filter can be applied if the user wishes.

```
smooth =  np.array([
[1,  2,  1],
[2, 16,  2],
[1,  2,  1]]
)/28

red = convolve2d(red,smooth,'same')
green= convolve2d(green,smooth,'same')
blue = convolve2d(blue,smooth,'same')
```

The simplicity of this algorithm means it's much faster than multiple convolutions as in the other algorithms, however may experience some artifacts described in section 5.

7.3 Results

This model was trained and run on Bayer CFA patterns RGGB, BGGR, GRBG, and GBRG. The output matrices are recorded as an example and shown here with an example output demosaiced image.

7.3.1 Output Matrices

7.3.2 RGGB

$$A = \begin{bmatrix} -0.00222192 & 0.00441482 & 0.00694215 & -0.05285718 & -0.00718026 \\ -0.0828956 & 0.00156459 & 0.23053804 & -0.004256 & -0.06093271 \\ 0.00176693 & 0.4623801 & 0.9856857 & 0.46688917 & 0.02074038 \\ -0.06303059 & 0.0118576 & 0.26344866 & -0.01037153 & -0.09590347 \\ -0.00933838 & -0.04106568 & 0.01058885 & -0.02172916 & -0.00552835 \end{bmatrix}$$

$$B = \begin{bmatrix} -1.6567669e-03 & 6.4156048e-02 & 1.3691127e-02 & -6.9852732e-02 & -1.7359730e-02 \\ 6.4814158e-02 & 3.5407442e-01 & 4.4436640e-01 & 1.8543483e-01 & 1.6577134e-04 \\ 1.8057724e-02 & 5.0642484e-01 & 9.5124918e-01 & 5.2959335e-01 & 3.7711043e-02 \\ -1.2587789e-02 & 2.3014025e-01 & 5.0957519e-01 & 2.8803253e-01 & 2.5533281e-02 \\ -1.7352087e-02 & 8.6155478e-03 & 3.5944175e-02 & 5.5804588e-03 & -2.0759631e-02 \end{bmatrix}$$

7.3.3 BRRG

$$A = \begin{bmatrix} 8.8047152e-03 & -1.3397861e-01 & -3.0274736e-02 & 1.6118576e-01 & -7.5177802e-03 \\ 1.0765021e-01 & 1.0780372e-01 & 2.7534381e-01 & 8.1826636e-04 & -2.1394865e-01 \\ -1.0232225e-01 & 2.0207496e-01 & 9.0032148e-01 & 4.3023455e-01 & 9.6992619e-02 \\ -1.6738902e-01 & 4.4565454e-02 & 3.3860767e-01 & -2.3990756e-02 & 4.0859189e-02 \\ 9.4118891e-03 & 9.5164590e-02 & 5.4547351e-02 & -1.2540291e-01 & -5.7237130e-02 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.01877008 & 0.03925351 & 0.0979136 & -0.01226373 & -0.06716131 \\ 0.01294238 & 0.18150336 & 0.5634988 & 0.31833664 & -0.10302378 \\ 0.01881674 & 0.36633795 & 0.8596393 & 0.5343497 & 0.08629613 \\ 0.03207455 & 0.18017216 & 0.41418955 & 0.31058827 & 0.06377131 \\ 0.02798093 & -0.01822321 & 0.01742865 & 0.14097276 & -0.01318185 \end{bmatrix}$$

7.3.4 GRBG

$$A = \begin{bmatrix} -3.2880850e-02 & -5.1339932e-02 & 3.6212612e-02 & 8.8264540e-02 & 2.3581486e-03 \\ -1.1228259e-03 & 7.7020624e-03 & 2.5487125e-01 & 1.2820997e-02 & -5.8575712e-02 \\ 2.2712197e-02 & 3.8152602e-01 & 8.8551021e-01 & 1.8844926e-01 & 2.5799152e-02 \\ -1.5419854e-01 & 3.3655711e-02 & 3.7573346e-01 & 1.3149377e-03 & -8.7610405e-04 \\ -9.1066081e-03 & 3.4600355e-02 & 3.1932961e-02 & -6.1877400e-02 & -2.0866711e-02 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.03579692 & 0.15680078 & 0.05175698 & 0.04043378 & 0.06121258 \\ -0.03162151 & 0.23945199 & 0.4323217 & 0.30452368 & 0.07642691 \\ 0.3196559 & 0.36402962 & 0.6927807 & 0.53642434 & 0.15219675 \\ 0.2374772 & 0.474933 & 0.71950555 & 0.34725773 & -0.04373903 \\ -0.08441237 & 0.21221964 & 0.22678946 & -0.02877011 & -0.10052066 \end{bmatrix}$$

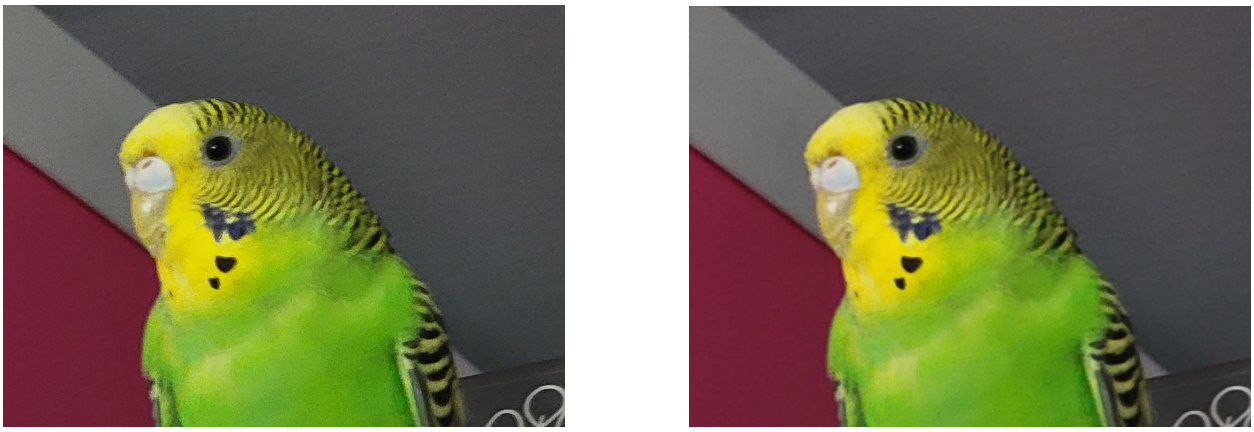
7.3.5 GBRG

$$A = \begin{bmatrix} 0.0273015 & -0.02541049 & -0.02291533 & 0.02603629 & -0.03728085 \\ -0.05174419 & 0.02760636 & 0.29843616 & 0.0682372 & -0.06839382 \\ -0.02932595 & 0.3233795 & 0.9233628 & 0.3867395 & 0.00500673 \\ 0.0118042 & 0.08482323 & 0.30386928 & 0.01560433 & -0.10365652 \\ -0.02815492 & -0.09627023 & -0.03000841 & 0.0112351 & 0.01556513 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.04845999 & 0.09117012 & 0.10936663 & -0.0077738 & 0.02255463 \\ 0.09364557 & 0.21055977 & 0.4567533 & 0.37662357 & 0.00561293 \\ 0.01926018 & 0.4343485 & 0.68812054 & 0.38444617 & 0.03213771 \\ -0.06692138 & 0.34802485 & 0.47294632 & 0.0554525 & 0.01446032 \\ 0.04451301 & 0.11907248 & 0.17561613 & 0.0377758 & -0.04305771 \end{bmatrix}$$

7.3.6 Bird Demosaicing

The result of this interpolation is visually very close to the ground truth image. The mean square error to ground truth is 2.47428365e-4, which is relatively good for such a simple demonising process described in section 7.2. Comparing these results to the other linear interpolation 5 and directional filtering 6, the output is worse by a significant degree. This is expected as the demonising process is much more simple than the other approaches. With a larger image training set and more training this error could be reduced.



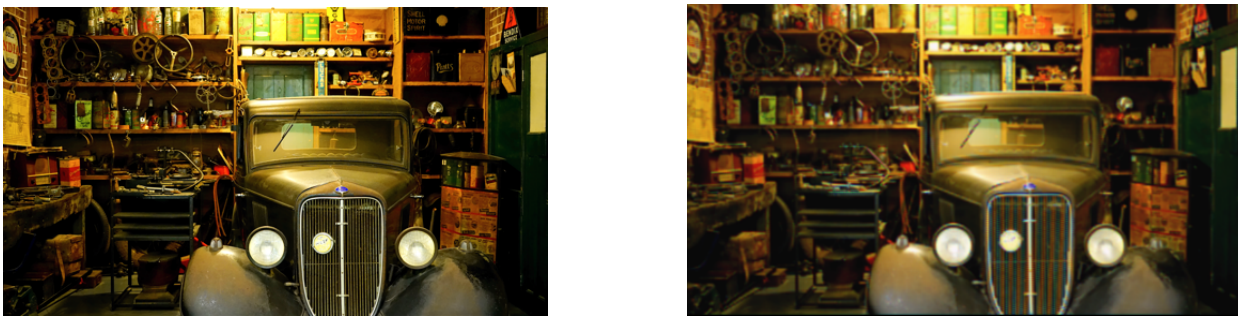
(a) Ground Truth Bird

(b) Demosaiced Bird

Figure 11: Linear Regression Demosaic

7.3.7 Truck Demosaicing

The results of this more complicated image seem relatively good. The pixel artifacts in the tool in the background and chrome areas still exists, but is seems reduced. The mean square error of this result is 1.0599969e-3, which compared to figure 10 results is 4 times better. Compared to Matlab's demosaicing method in figure 5 is 5 times better with reference to the mean square error. This is likely due the the fact the model was trained on similar images to the truck image, with dark colours and possible strange artifacts.



(a) Ground Truth Truck

(b) Demosaiced Truck

Figure 12: Linear Regression Truck Results

8 White Balance and Histogram Equalization

Further image processing is required in the image processing pipeline. The next steps are white balancing and histogram equalization and those will be implemented here.

8.1 Histogram Equalization

Implemented here is a very simple CLAHE histogram equalization. CLAHE divides up the image into sections and computes a histogram equalization over each block [4]. To reduce an extreme difference in adjacent blocks, an overlap is defined. CLAHE aims to reduce amplification contact by clipping the values based on predefined values in the histogram before computing the CDF. [4].

8.1.1 CLAHE Implementation

First, the code converts the RGB image into a LAB colour space using OpenCV's 'cv2.cvtColor' function. The result is then split into L, A, and B channels using 'cv2.split'. The tile height and width are then calculated by using the user input tile size. The overlap is then calculated using the user input as well.

```
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
l_channel, a_channel, b_channel = cv2.split(lab_image)

tile_height = l_channel.shape[0] // grid_size[0]
tile_width = l_channel.shape[1] // grid_size[1]

overlap_height = int(tile_height * overlap_ratio)
overlap_width = int(tile_width * overlap_ratio)

l_channel_clahe = np.zeros_like(l_channel)
```

The program uses nested loops to iterate over each tile grid while considering the overlap. The regions of the L channel corresponding to the current tile are calculated, then the tile is extracted from the L channel. The histogram is then computed using NumPy 'np.histogram' function.

```
y_start = i * tile_height - min(i, 1) * overlap_height
y_end = (i + 1) * tile_height + min(grid_size[0] - i - 1, 1)
    * overlap_height
x_start = j * tile_width - min(j, 1) * overlap_width
x_end = (j + 1) * tile_width + min(grid_size[1] - j - 1, 1)
    * overlap_width

tile = l_channel[y_start:y_end, x_start:x_end]

hist, _ = np.histogram(tile.flatten(), bins=256, range=[0, 256])
```

The fundamental application of CLAHE is applied by then clipping the histogram to limit the maximum pixel intensity by using the defined helper function (see section 10.3 for helper function) [4]. After this, the CDF can be computed using a helper function (see section 10.3 CDF helper function). Then the program maps the pixel values of the current tile using the computed CDF, equalizing the histogram within the tile and enhancing contrast

```
hist, _ = np.histogram(tile.flatten(), bins=256, range=[0, 256])
hist_clipped = clip_histogram(hist, clip_limit)
cdf = compute_cdf(hist_clipped)
tile_equalized = np.interp(tile.flatten(), np.arange(256), 255
    * cdf).reshape(tile.shape)
l_channel_clahe[y_start:y_end, x_start:x_end] = tile_equalized
```

The image is then stacked and converted back to RGB colour space and returned using OpenCV functions 'merge' and 'cvtColor' respectively.

8.1.2 CLAHE Results

At user input settings 8x8 tile with 6 overlap and clipped at 2.0. In figure 13, the result is a little bit blocky, but seems to be the best results for the center focal point of the picture. The constant can clearly seen to be enhanced after the CLAHE algorithm is applied.

Figure 13: Histogram Equalization

8.2 White Balancing

Gray World white balancing is a fundamental technique in digital imaging aimed at correcting colour casts and ensuring faithful colour reproduction in images [5]. Rooted in the assumption that the average color in a scene tends towards a neutral gray, this method adjusts the color channels of an image to achieve a balanced, natural appearance. By computing the average color across each channel and scaling the color values accordingly, Gray World white balancing aims to eliminate unwanted color biases caused by varying lighting conditions [5].

8.2.1 Gray World Implementation

To preforms a simple gray world white balancing approach, the program first ensure the image is in a float 32 format with a simple cast. Then the average of each channel is calculated using Numpy mean function. These are then averaged together to get the gray worlds definition of "white" [5].

```
# Convert the image to float32 to avoid overflow during calculations
img_float = image.astype(np.float32)

# Calculate the average color of the image across each color channel
avg_r = np.mean(img_float[:,:,0])
avg_g = np.mean(img_float[:,:,1])
avg_b = np.mean(img_float[:,:,2])

# Calculate the average gray value
avg_gray = (avg_r + avg_g + avg_b) / 3.0
```

Then the scaling factor is calculated for each channel. Each scale factor is computed by dividing the average gray value by the corresponding average colour value. These scale factors will be used to balance the colours. The scaling factor is then applied to balance the colours. Then the result is clipped in case if some pixels are scaled beyond 255, and then it ensures the results are in uint_8 format.

```
scale_r = avg_gray / avg_r
scale_g = avg_gray / avg_g
scale_b = avg_gray / avg_b

# Apply the scale factors to balance the colors
```

```
img_balanced = img_float * [scale_r, scale_g, scale_b]

# Clip the values to ensure they are within the valid range [0, 255]
img_balanced = np.clip(img_balanced, 0, 255)

# Convert the image back to uint8 format
img_balanced = img_balanced.astype(np.uint8)
```

8.2.2 Gray World Results

The results can vary in proper white balancing quality. When a image has little to no white in the scene, the results can have a slight hue shift. This can be seen in figure 14a as there is a very slight blue hue shift. However, figure 14b has a large white contribution, so the white balancing shift is relatively accurate. Some ways to make this white balancing better is to implement human input to declare what section should be white. This would give the program a definition on what should be white and can correct from there [12].

(a) Truck White Balanced

(b) Bird White Balanced

Figure 14: White Balance Results

9 Conclusion

Each implementation of demosaicing resulted in different strengths and weaknesses outlined in their respective sections. Each section showed two or more examples of the algorithm in action and displayed their resulting mean square error. This mean square error is useful in having a numeric reference to the accuracy of these images, but visual image quality inspection was also taken into account. For example, Matlab and Directional Filtering Demosaicing 6 displayed similar mean square error, but visually Directional Filtering produced much better results.
To give an overview of the mean square error, each of these methods was tested on 10 simulated CFA RGGB Bayer pattern images and their mean square errors were calculated and averaged. The results in table 1 show the best results are from Directional Filtering, which is expected as this is the most advanced algorithm implemented. The next best is Linear regression due the the ability of the method to learn the optimal interpolation matrix. The worst results come from the simple linear interpolation results, which is no surprise as this is the most simple method implemented.

Linear Interpolation 5	**Directional Filtering 6**	**Linear Regression 7**
4.36e-3	1.22e-3	4.09e-3

Table 1: Mean Square Average

9.1 White Balancing and Histogram Equalization

White this isn't the focal point of this paper, the results still show promise in their implementation. The white balancing algorithm, white simple, produces fairly accurate results. Tested on an artificially hued image (i.e. an induced green hue) the white balance was able to accurately return the image to a normal hue.
Also implemented in this paper, histogram equalization stands as a powerful technique for enhancing the contrast and overall visibility of images. By redistributing pixel intensities across the histogram, it effectively increases the dynamic range of the image, leading to improved visual clarity and detail. The implemented histogram equalization in this paper is a simple CLAHE method. While the results are not perfect yet, they show promise and can be refined for phase 2 of this project.
Phase 2 will introduce a variety of more advanced challenges, such as demosaicing raw images with noise non-ideal colour constants, and lighting. However, the work done in this paper lays a foundation for further refinement that will be done in phase 2 of this project. Different demosaicing methods were implemented, then compared contrast, and then simple white balancing and histogram equalization were implemented. Further work can be done to increase the quality of the linear regression demosaicing method, and make a more advanced white balancing and histogram equalization implementation.

10 Code Snippets

10.1 Mean Square Error Core

```
def mean_squared_error(imageA, imageB):
    # Ensure the images are in floating point in case they are in uint8
    imageA = np.array(imageA, dtype=np.float32)
    imageB = np.array(imageB, dtype=np.float32)

    # Compute the mean squared error between the two images
    err = np.sum((imageA - imageB) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    return err
```

10.2 Bayer Channel Split Example

10.2.1 For Linear Interpolation

```
def bayer(im):
    r = np.zeros(im.shape[:2])
    g = np.zeros(im.shape[:2])
    b = np.zeros(im.shape[:2])
    r[0::2, 0::2] += im[0::2, 0::2]
    g[0::2, 1::2] += im[0::2, 1::2]
    g[1::2, 0::2] += im[1::2, 0::2]
    b[1::2, 1::2] += im[1::2, 1::2]
    return r, g, b
```

10.2.2 For Directional Filtering and Linear Regression

Just RGGB is shown here.

```
def create_bayer_masks(shape, pattern):
R_m = np.zeros(shape)
G_m = np.zeros(shape)
B_m = np.zeros(shape)

if pattern == "RGGB":
    R_m[0::2, 0::2] = 1
    G_m[0::2, 1::2] = 1
    G_m[1::2, 0::2] = 1
    B_m[1::2, 1::2] = 1
return R_m, G_m, B_m
```

10.3 CLAHE Helper functions

10.3.1 Clip Histogram

```
def clip_histogram(hist, clip_limit):
    clipped_hist = np.minimum(hist, clip_limit)
    excess = hist - clip_limit
    clipped_hist[:-1] += excess[1:]
    clipped_hist[1:] += excess[:-1]
```

```
    return clipped_hist
```

10.3.2 Compute CDF

```
def compute_cdf(hist):
    cdf = hist.cumsum()
    return cdf / cdf[-1]
```

10.4 Mosaicing Code

One of the four functions for making a CFA mosaic pattern from a image used for simulation. See Github repo file "mosaic.py" for the rest of the functions.

```
    def bggr_mosaic(rgb_image):
    rows, columns, _ = rgb_image.shape
    mosaiced_image = np.zeros((rows, columns), dtype=np.uint8)

    for col in range(columns):
        for row in range(rows):
            if col % 2 == 0 and row % 2 == 0:
                mosaiced_image[row, col] = rgb_image[row, col, 0]  # Red
            elif col % 2 == 0 and row % 2 == 1:
                mosaiced_image[row, col] = rgb_image[row, col, 1]  # Green
            elif col % 2 == 1 and row % 2 == 0:
                mosaiced_image[row, col] = rgb_image[row, col, 1]  # Green
            elif col % 2 == 1 and row % 2 == 1:
                mosaiced_image[row, col] = rgb_image[row, col, 2]  # Blue

    return mosaiced_image
```

References

[1] L. Chang, "Hybrid color filter array demosaicking for effective artifact suppression," *Journal of Electronic Imaging*, vol. 15, no. 1, p. 013003, 2006.

[2] K. Hirakawa and T. Parks, "Adaptive homogeneity-directed demosaicing algorithm," *IEEE Transactions on Image Processing*, vol. 14, no. 3, pp. 360–369, 2005.

[3] Matlab, "Demosaic."

[4] P. Musa, F. A. Rafi, and M. Lamsani, "A review: Contrast-limited adaptive histogram equalization (clahe) methods to help the application of face recognition," in *2018 Third International Conference on Informatics and Computing (ICIC)*, pp. 1–6, 2018.

[5] E. Lam, "Combining gray world and retinex theory for automatic white balance in digital photography," in *Proceedings of the Ninth International Symposium on Consumer Electronics, 2005. (ISCE 2005).*, pp. 134–139, 2005.

[6] R. Lukac and K. Plataniotis, "Color filter arrays: Design and performance analysis," *IEEE Transactions on Consumer Electronics*, vol. 51, p. 1260–1267, Nov 2005.

[7] P. A. Cheremkhin, V. V. Lesnichii, and N. V. Petrov, "Use of spectral characteristics of dslr cameras with bayer filter sensors," *Journal of Physics: Conference Series*, vol. 536, p. 012021, Sep 2014.

[8] T. Akazawa, Y. Kinoshita, S. Shiota, and H. Kiya, "Three-color balancing for color constancy correction," *Journal of Imaging*, vol. 7, p. 207, Oct 2021.

[9] W. A. Mustafa and M. M. M. A. Kader, "A review of histogram equalization techniques in image enhancement application," *Journal of Physics: Conference Series*, vol. 1019, p. 012026, jun 2018.

[10] D. Alleysson, S. Susstrunk, and J. Herault, "Linear demosaicing inspired by the human visual system," *IEEE Transactions on Image Processing*, vol. 14, no. 4, pp. 439–449, 2005.

[11] D. Menon, S. Andriani, and G. Calvagno, "Demosaicing with directional filtering and a posteriori decision," *IEEE Transactions on Image Processing*, vol. 16, no. 1, pp. 132–141, 2007.

[12] I. Boyadzhiev, K. Bala, S. Paris, and F. Durand, "User-guided white balance for mixed lighting conditions," *ACM Transactions on Graphics*, vol. 31, p. 1–10, Nov 2012.