Image and Video Processing Assignment 1

Arseni Loika

March 2024

1 Point Operations

1.1 Tone mapping & Linearization

To linearize the image, the function $f(x) = x^{1/2.2}$ needs to be reversed, yielding $f(x) = x^{2.2}$, to be applied to every pixel. The image is also processed to increase brightness (by linear multiplication) and decrease contrast (by exponential operation), as in the code below.

```
ferrari = imread("./ferrari.JPG");
ferrari_db = im2double(ferrari);

% inverted the function
ferrari_corrected = ferrari_db.^2.2;

% optionals
ferrari_adjusted_bright = ferrari_corrected.*2.3; % brightness
ferrari_adjusted_contrast = ferrari_adjusted_bright.^(0.4); % contrast

% plot
% ... (plotting code omitted)
```

Which yields the following results:

Figure 1: Tone Mapping and Linearization Results - original and with inverted Gamma correction

Figure 2: Tone Mapping and Linearization Results - increased brightness and decreased contrast

1.2 Color Correction

To perform pixel-based white balancing of the image, a sample pixel was selected using `impixel()` function. The pixel's channel averages were calculated and used to get new multiplier (alpha) for each channel. Then each channel was modified using that alpha. Details in the code below.

```
wb_pic = imread("white_balance_input.jpg");
wb_pic = im2double(wb_pic);
wb_pic = wb_pic.^2.2;
imshow(wb_pic); title("Original");

% select sample pixel
[x, y, rgb] = impixel(wb_pic);

% equalize the rgb values
rgb_new = (rgb(1) + rgb(2) + rgb(3)) / 3;

% find alpha
alpha_R = rgb_new/(rgb(1));
alpha_G = rgb_new/(rgb(2));
alpha_B = rgb_new/(rgb(3));

% process every pixel's corresponding channels
wb_pic(:,:,1) = wb_pic(:,:,1)*alpha_R;
wb_pic(:,:,2) = wb_pic(:,:,2)*alpha_G;
wb_pic(:,:,3) = wb_pic(:,:,3)*alpha_B;

% show result
imshow(wb_pic); title("Pixel-Based Correction");
```

The code yields the following result:

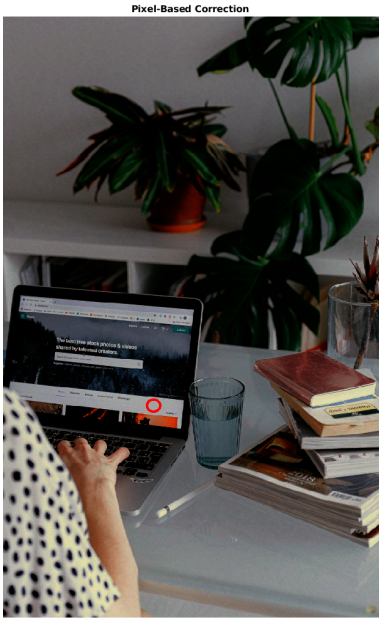

Figure 3: Pixel-Based Balancing, sample pixel selected from the area within red ellipsis

To perform white balancing of the image with the gray-world assumption, the averages of each RGB channel, as well as their overall average, were calculated and used to get new multiplier (alpha) for each channel. Then each channel was modified using that multiplier.

```
% get the average RGB value, make it grey
wb_img_original = imread("white_balance_input.jpg");
wb_img = im2double(wb_img_original);
wb_img = wb_img.^2.2;

% get individual channels
red_channel = wb_img(:,:,1);
green_channel = wb_img(:,:,2);
blue_channel = wb_img(:,:,3);

% get averages of respective channel
avg_red = mean(red_channel, "all");
avg_green = mean(green_channel, "all");
avg_blue = mean(blue_channel, "all");

% get the average across channels
rgb_new = (avg_red + avg_blue + avg_green) / 3;

% calculate respective multipliers
alpha_red = rgb_new / avg_red;
alpha_green = rgb_new / avg_green;
alpha_blue = rgb_new / avg_blue;

% modify the channels
wb_img(:,:,1) = wb_img(:,:,1)*alpha_red;
wb_img(:,:,2) = wb_img(:,:,2)*alpha_green;
wb_img(:,:,3) = wb_img(:,:,3)*alpha_blue;

% plot
% ... (plotting code omitted)
```

The code yields the following result:

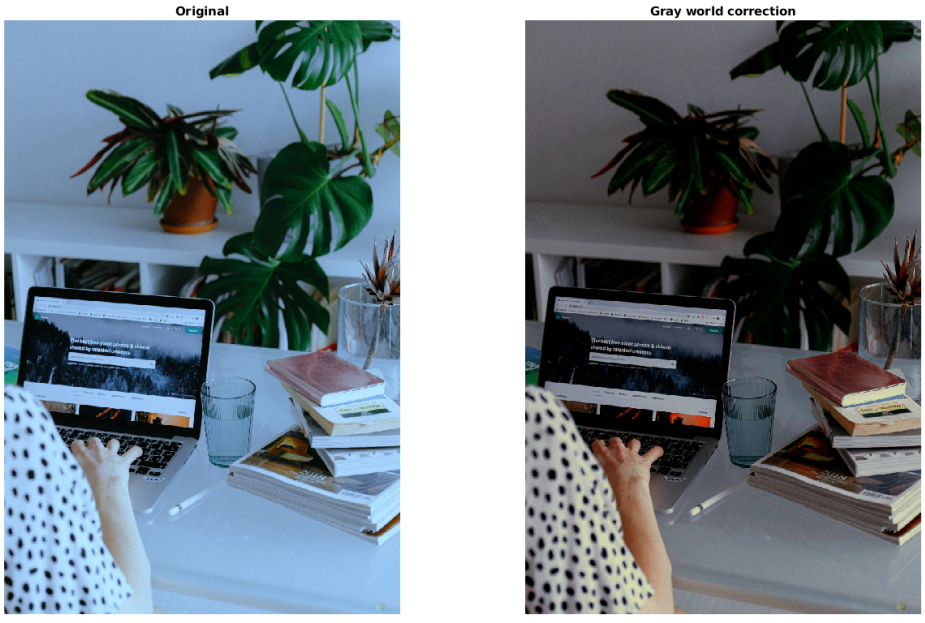

Figure 4: Tone Mapping and Linearization Results

1.3 Histograms

To calculate an image histogram from scratch, first the image is split into RGB channels. Then each channel was processed to convert it into a vector, and the unique intensities were retrieved. Then the cumulative sums were calculated for each channel's intensities, and the results plotted. The details are in the code below.

```
function [freq_reds, freq_greens, freq_blues] = histogram(image_path)
    img = imread(image_path);
    img = im2uint8(img);

    % process the channels - reshape into vectors and get all unique
       values
    red = img(:,:,1);
    red = reshape(red,1,[]);
    unique_reds = unique(red);
    red = sort(red);

    green = img(:,:,2);
    green = reshape(green,1,[]);
    unique_greens = unique(green);
    green = sort(green);

    blue = img(:,:,3);
    blue = reshape(blue,1,[]);
    unique_blues = unique(blue);
    blue = sort(blue);

    L = 1:256;
    freq_reds = zeros(size(unique_reds));
    freq_greens= zeros(size(unique_greens));
    freq_blues = zeros(size(unique_blues));

    for i = 1:256
        % count the occurence of each unique value in their channel
        freq_reds(i) = sum(unique_reds(i) == red);
        freq_greens(i) = sum(unique_greens(i) == green);
        freq_blues(i) = sum(unique_blues(i) == blue);
    end

    % plot
    plot(L, freq_reds, 'r', L, freq_greens, 'g', L, freq_blues, 'b');
       title("Histogram for " + image_path);
end
```

The code yields the following result:

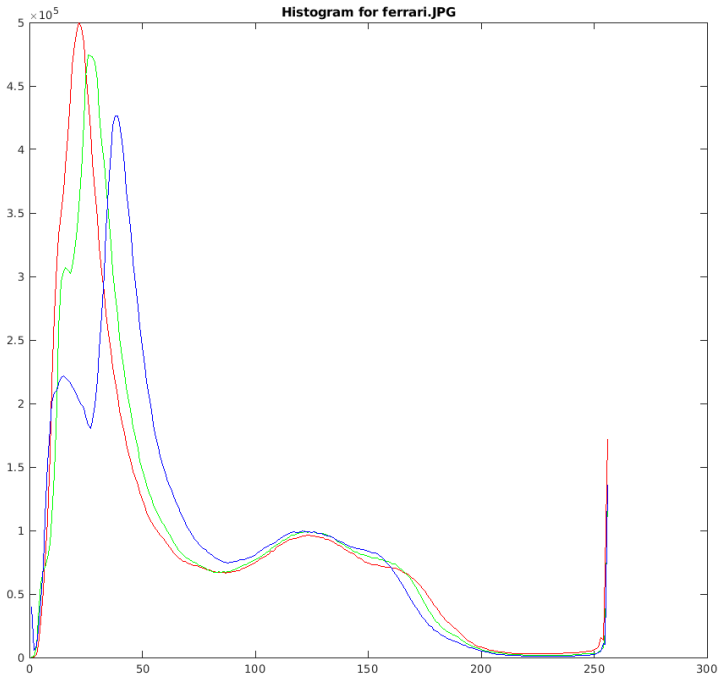

Figure 5: Pixel-Based Balancing, sample pixel selected from the area within red ellipsis

1.4 Histogram Equalization

Global histogram equalization was done by first converting the image to HSV format, then calculating the histogram of the 'value' element. Then, the cumulative sum of the histogram is calculated, and rescaled to be in the range of 0 to 1. It is then scaled back and applied to the image. More details in the code below.

```
function [eq] = histogram_eq_global(img, flag)
    img = rgb2hsv(img);

    % compute the histogram
    [h, ~] = imhist(img(:,:,3));

    % get cumulative density
    cdf = cumsum(h);

    % normalize
    cdf = rescale(cdf, 0, 1);

    % apply to image
    eq = cdf(round(img(:,:,3)*255+1));
    img(:,:,3) = eq;

    % plot the info
    % ... (code hidden)

end
```

The code yields the following results:

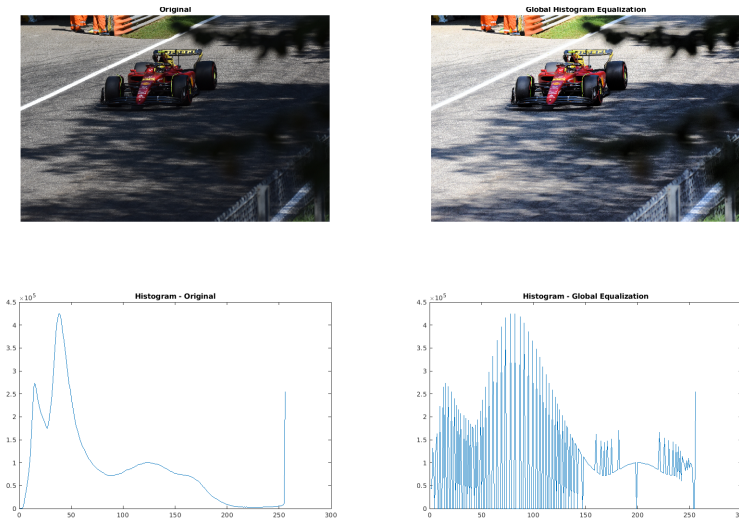

Figure 6: Global Histogram Equalization

For local histogram equalization, the same principle as in global correction was applied, but this time on a moving window of predetermined size, splitting the image into $N \times N$ windows and applying the correction on each independently. Full details in the code below.

```
img = imread("ferrari.JPG");
img = rgb2hsv(img);
out = img;

% define the number of chunks and window size
dims = size(img);
chunks = 5;
window_size = floor(dims(1:2)/chunks);

for i = 0:chunks-1
    for j = 0:chunks-1
        % define the chunk dimensions
        chunk_start = ([i j].*window_size)+1;
        chunk_end = ([i j].*window_size) + window_size;

        % get the chunk from the image
        chunk = out(chunk_start(1):chunk_end(1), chunk_start(2):chunk_end
            (2),:);

        % equalize it
        eq_chunk = histogram_eq_global(chunk, false);

        % update the chunk in the output image
        out(chunk_start(1):chunk_end(1), chunk_start(2):chunk_end(2),3) =
            eq_chunk;
    end
end

out = hsv2rgb(out);
imshow(out); title("Local Equalization with " + chunks + " windows");
```

The code yields the following result:

Figure 7: Local Histogram Equalization

1.4.1 Bonus

Locally adaptive equalization was achieved by applying global method to windows of increasing size, thus taking into consideration larger chunks at each iteration. The modifications to the code are seen below.

```
% define chunk start - constant
chunk_start = ([0 0].*window_size)+1;
for i = 0:chunks-1
    for j = 0:chunks-1
        chunk_end = ([i j].*window_size) + window_size;
        chunk = out(chunk_start(1):chunk_end(1), chunk_start(2):chunk_end
            (2),:);
        eq_chunk = histogram_eq_global(chunk, false);
        out(chunk_start(1):chunk_end(1), chunk_start(2):chunk_end(2),3) =
            eq_chunk;
    end
end
out = hsv2rgb(out);
```

The code yields the following result:

Figure 8: Locally Adaptive Histogram Equalization

1.5 Matting

Matting was completed by computing a relative green level of the cat image, and using it to create a mask which is then used to mask out the green background and then applied to the background image at each channel. Full details in the code below.

```
cat_img = imread("cat.jpg");
cat_img = im2double(cat_img);
background = imread("background.jpg");
background = im2double(background);
[bkg_rows, bkg_cols,~] = size(background);

% resize foreground to match background
cat_img = imresize(cat_img, [bkg_rows, bkg_cols]);

% foreground color channels
cat_red = cat_img(:,:,1);
cat_green = cat_img(:,:,2);
cat_blue = cat_img(:,:,3);

% background color channels
bkg_red = background(:,:,1);
bkg_green = background(:,:,2);
bkg_blue = background(:,:,3);

% get the relative green level
green_lvl = cat_green.*(cat_green - cat_red).*(cat_green - cat_blue);

% set threshold
threshold = 0.3*mean(green_lvl(green_lvl > 0));
filter = green_lvl > threshold;

% replace pixels
cat_red(filter) = bkg_red(filter);
cat_green(filter) = bkg_green(filter);
cat_blue(filter) = bkg_blue(filter);

% final recombined image
recombined = cat(3, cat_red, cat_green, cat_blue);

% plot
imshow(recombined); title("Combined");
```

The code yields the following results:

Figure 9: Background and Foreground images for matting

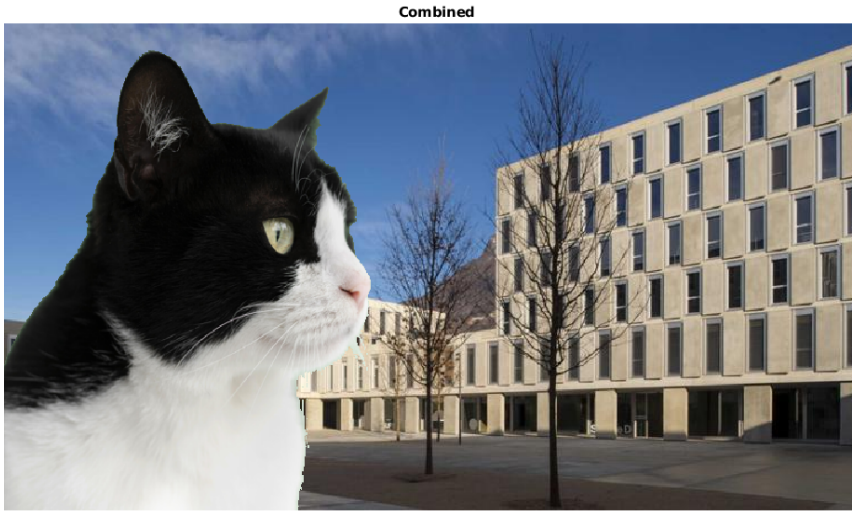

Figure 10: Matting Results

1.6 Bonus

For the bonus, a new pair of images were combined, and then adjusted in contrast and brightness. Details in code below.

```
fgd = imread("drift.jpg");
fgd = im2double(fgd);
fgd = fgd.^2.2;
background = imread("road.jpg");
background = im2double(background);
background = background.^2.2;
[bkg_rows, bkg_cols,~] = size(background);

% ... (repeated code from ex1.5 omitted)

green_lvl = fgd_green.*(fgd_green - fgd_red).*(fgd_green - fgd_blue);
threshold = 0.004*mean(green_lvl(green_lvl > 0));
filter = green_lvl > threshold;

% ... (repeated code from ex1.5 omitted)

recombined = cat(3, fgd_red, fgd_green, fgd_blue);
recombined_bright = recombined.*1.2;
recombined_contrast = recombined_bright.^1.3;
```

The code yields the following results:

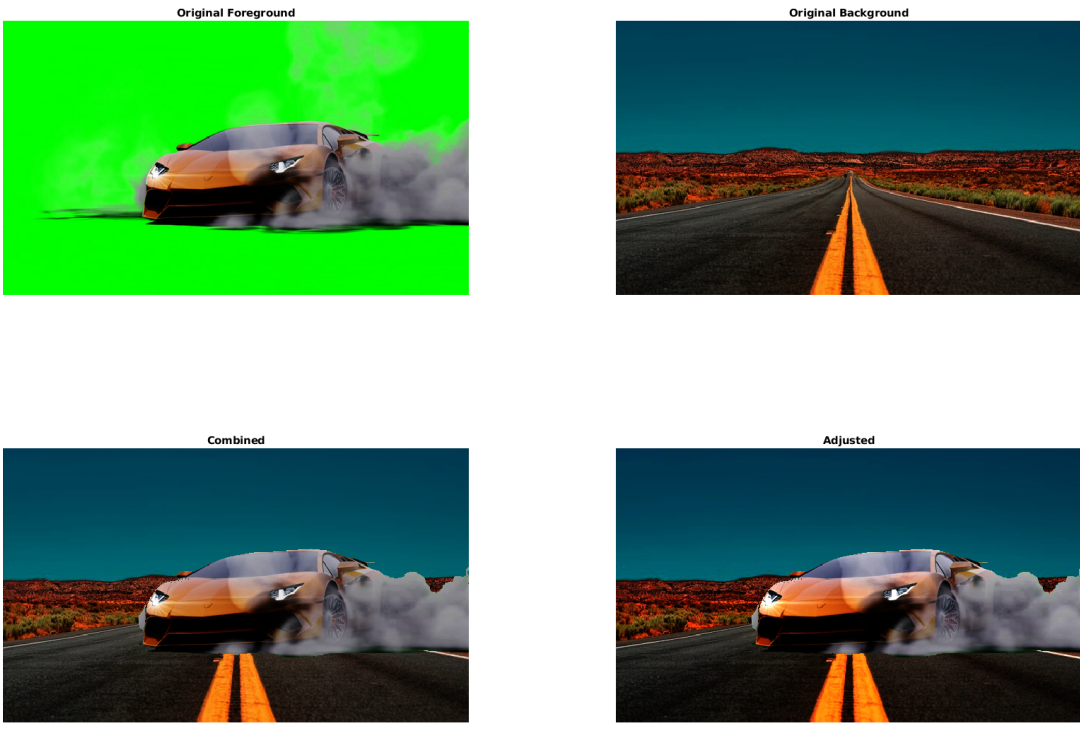

Figure 11: Bonus Intermediate Results