

COMPENG 4TN4

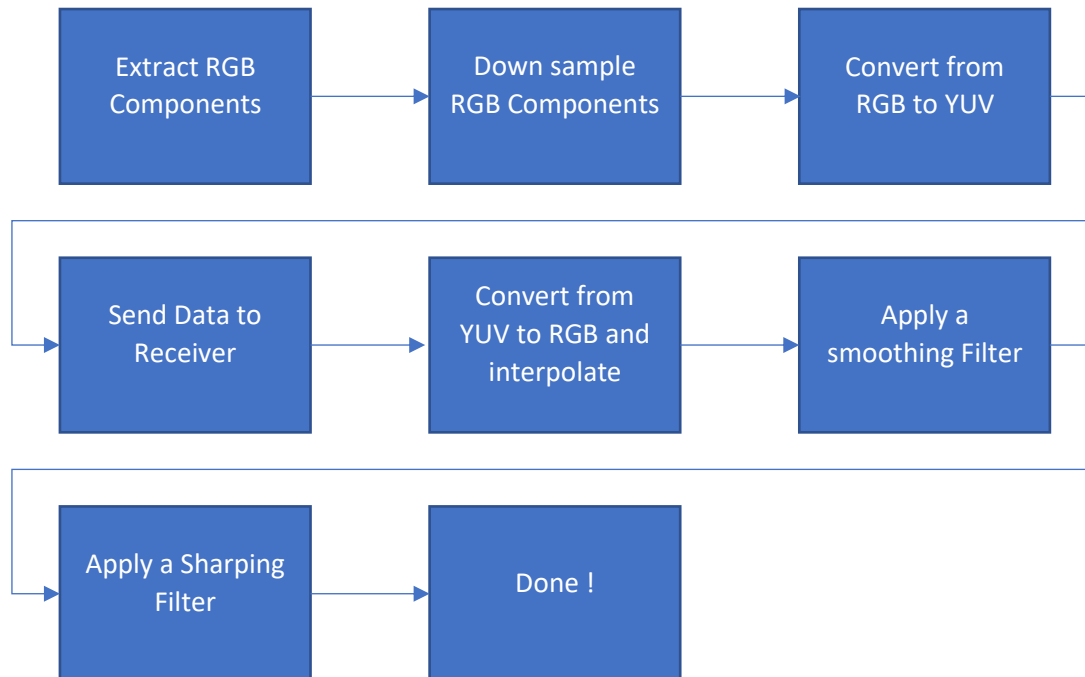
Dr. Xiaolin

Phase1

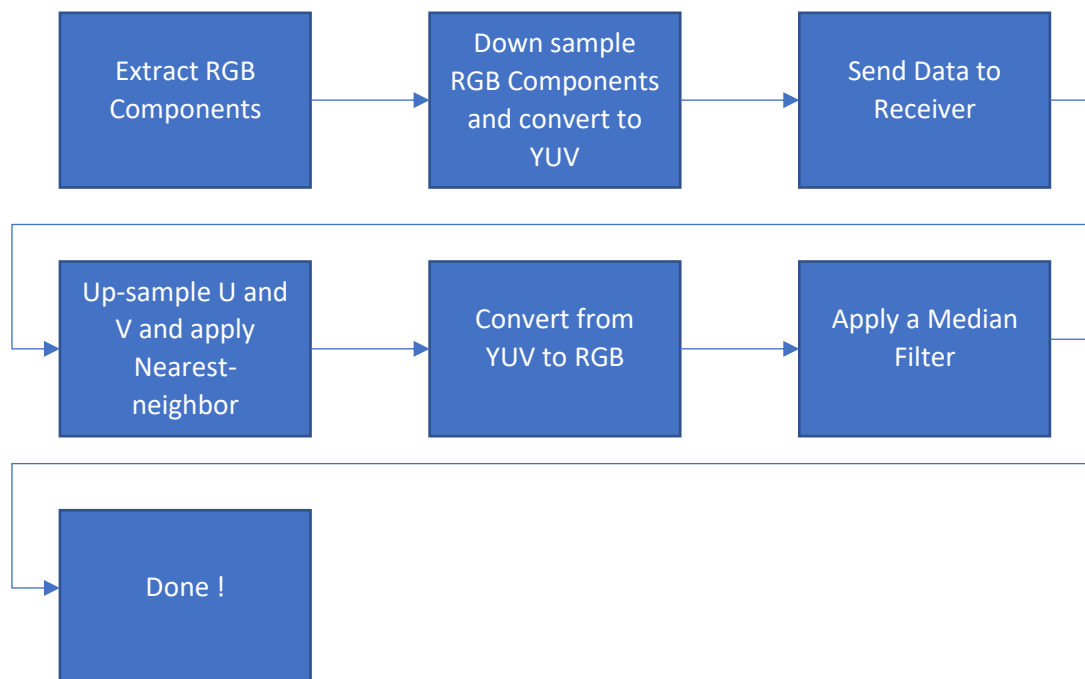
February 19th, 2023

Sawoud Al-los - 400188404

To achieve the requirements for this project, the following algorithm had to be implemented:



The above was the writer's initial plan, but upon implementation, the author managed to combine two of the steps, while adding an extra step, to effectively optimize the algorithm by one stage, below shows the implemented algorithm:



This paper will go through each stage and if possible, show the state of the image components.

The Chosen Image

This was the image that was used, it is a screenshot from the game *Black Mesa*, the reason it was used was because of all the details present in the image, such as the three jets at varying distance, the dam with the water fall, and the communications towers in the background, it was also selected due to its color and contrast variety through the image.

Also, this image was chosen due to the fact that it was a 1920x1080, as that resolution is the standard FHD resolution by the majority of televisions, monitors and smartphones, as well as other displays. But it was decided to trim the image by 1 pixel row and column to make it an 1919x1079 image this was done to ensure the algorithm works for images that are not divisible by 2 or 4.

Below is the image used:



Figure 1: 1919x1079 Image

The Extraction of the RGB Components

The code snippet below shows how the Image is read and then stored into an array that contains all of its RGB values, for the sake of convenience, the RGB components are split into three arrays.

```

OGImage_array = imread('OGImage.jpg');
[width,height,depth] = size(OGImage_array);
Y_downsample_factor = 2;
UV_downsample_factor = 4;
Y = zeros(int32(width/Y_downsample_factor),int32(height/Y_downsample_factor));
U = zeros(int32(width/Y_downsample_factor),int32(height/Y_downsample_factor));
V = zeros(int32(width/Y_downsample_factor),int32(height/Y_downsample_factor));

R = OGImage_array(:,:,1);
G = OGImage_array(:,:,2);
B = OGImage_array(:,:,3);

```

Figure 2: RGB Extraction

Down sampling and YUV conversion

In order to convert images for RGB to YUV, we must multiply the each RGB component by the YUV matrix below:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Figure 3: RGB to YUV matrix

Below is the code:

```

%% This section performs the downsampling as well as the YUV conversion in the same loop
disp('YUV conversion and Downsampling started...');
for y = 2:Y_downsample_factor:width
    for x = 2:Y_downsample_factor:height
        %if ((mod(x,Y_downsample_factor) == 0 && mod(y,Y_downsample_factor) == 0))
        Y(y/Y_downsample_factor,x/Y_downsample_factor) = int8(0.299*R(y,x) + 0.587*G(y,x) + 0.114*B(y,x));
        if (mod(x,UV_downsample_factor) == 0 && mod(y,UV_downsample_factor) == 0)
            U(y/UV_downsample_factor,x/UV_downsample_factor) = int8(-0.147*R(y,x) - 0.289*G(y,x) + 0.436*B(y,x));
            V(y/UV_downsample_factor,x/UV_downsample_factor) = int8(0.615*R(y,x) - 0.515*G(y,x) - 0.100*B(y,x));
        end
    end
end

```

Figure 4: YUV Conversion code

The image above performs the conversion as well as it down samples both the Y,U and V matrices by only performing every other operation for Y and performing every 4th operation for U and V, this is done for optimization purposes. Below is the result:

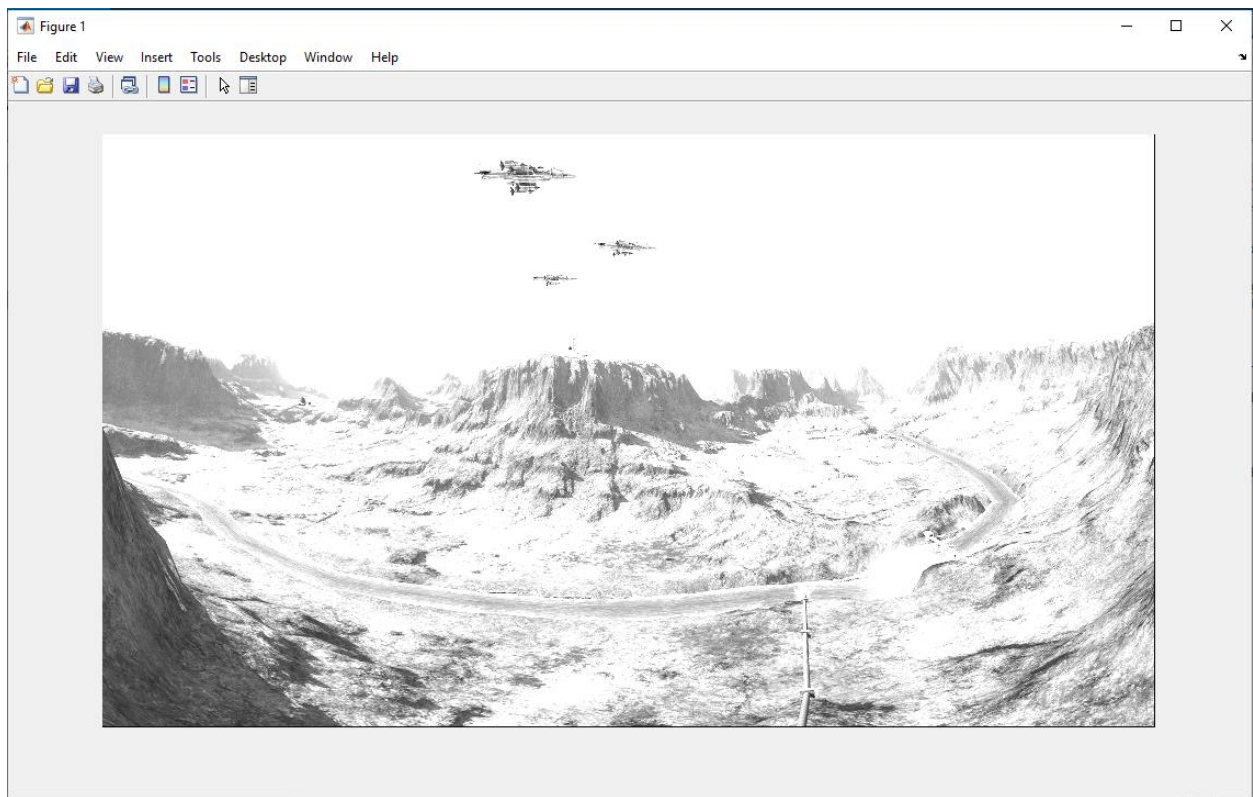


Figure 5: Y Component of The Image



Figure 6: U Component of The Image

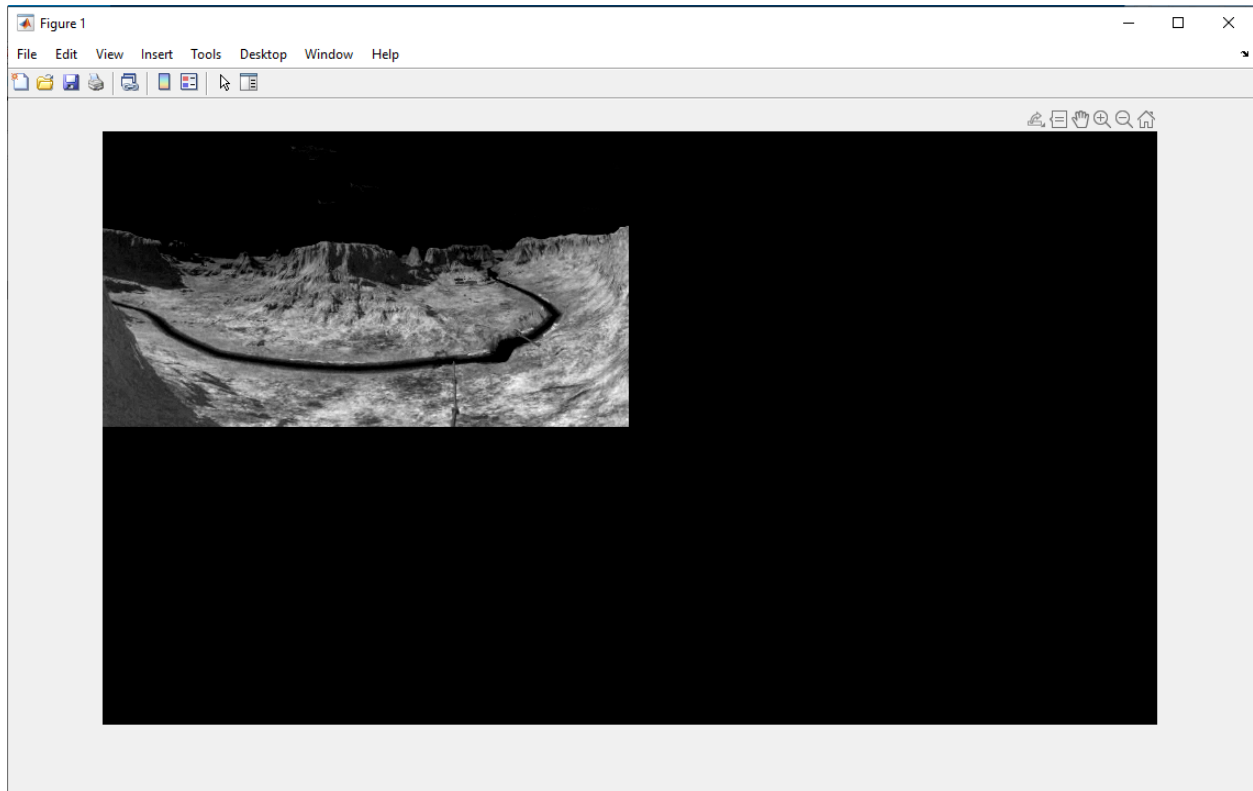


Figure 7: V Component of The Image

As we can see from each image, the Y component is housing all the brightness components of the image, the U and V images are not so clear in what they represent and why they can be subsampled by a higher factor, but if we look at the approximation equations for U and V we can see that U is the difference between Y and B, V is the difference between Y and R, this means that these channels are the actual channels that house the colour info. We can see the U channel having a lot of bright spaces along the sky and the water due to the fact that blue has a higher presence in those regions, and V image looks like we removed the river and the sky from it, as there is little red component in these regions, but along the terrain there is a much higher presence of red colour and that is why the U and V images almost look like negatives of each other, in the case of this image.

U and V Upsampling

After the data gets sent to the receiver module, we want to start the process of reconstructing the image. What we first need to do is upsample the U and V matrices, but since the U and V matrices had much less data to worry about in the first place, and their affect on the image is a smaller compared to the Y matrix, we can use a quick and lazy method of interpolation to optimize the application, we use nearest-neighbor interpolation, in which we take the values of one element, and we share it with adjacent pixels, the image below shows how this operation works:

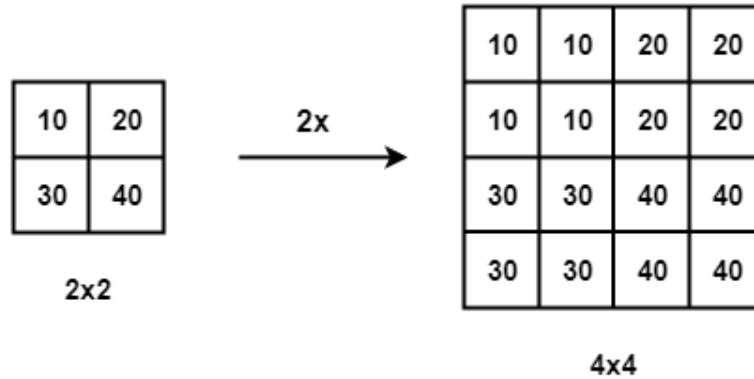


Figure 8: Nearest-Neighbor Interpolation Example

Below is the code for that section:

```

%% Data gets sent to receiver
disp("Data sent to receiver ---->");
U_UPSAMPLED = zeros(int32(width/Y_downsample_factor),int32(height/Y_downsample_factor));
V_UPSAMPLED = zeros(int32(width/Y_downsample_factor),int32(height/Y_downsample_factor));

%% U and V get upsampled by 2 since they were down sampled by 4
disp("UV upsampling started,with interpolation");
for y = 2:(width/UV_downsample_factor) - 1
    for x = 2:(height/UV_downsample_factor) - 1
        U_UPSAMPLED(y*Y_downsample_factor,x*Y_downsample_factor) = U(y,x);
        U_UPSAMPLED(y*Y_downsample_factor+1,x*Y_downsample_factor+0) = U(y,x);
        U_UPSAMPLED(y*Y_downsample_factor+0,x*Y_downsample_factor+1) = U(y,x);
        U_UPSAMPLED(y*Y_downsample_factor+1,x*Y_downsample_factor+1) = U(y,x);

        V_UPSAMPLED(y*Y_downsample_factor,x*Y_downsample_factor) = V(y,x);
        V_UPSAMPLED(y*Y_downsample_factor+1,x*Y_downsample_factor+0) = V(y,x);
        V_UPSAMPLED(y*Y_downsample_factor+0,x*Y_downsample_factor+1) = V(y,x);
        V_UPSAMPLED(y*Y_downsample_factor+1,x*Y_downsample_factor+1) = V(y,x);
    end
end
end

```

Figure 9: Nearest-Neighbor Interpolation Implementation

Convert YUV to RGB

The below code performs the conversion from YUV to RGB for every other row and column:

```

%% YUV to RGB conversion occurs
disp("YUV to RGB...");
for y = 2:Y_downsample_factor:height
    for x = 2:Y_downsample_factor:width
        R(y,x) = int8(Y(y/Y_downsample_factor,x/Y_downsample_factor) + 1.140*V_UPSAMPLED(y/Y_downsample_factor,x/Y_downsample_factor));
        G(y,x) = int8(Y(y/Y_downsample_factor,x/Y_downsample_factor) - 0.395*U_UPSAMPLED(y/Y_downsample_factor,x/Y_downsample_factor) - 0.581*V_UPSAMPLED(y/Y_downsample_factor,x/Y_downsample_factor));
        B(y,x) = int8(Y(y/Y_downsample_factor,x/Y_downsample_factor) + 2.032*U_UPSAMPLED(y/Y_downsample_factor,x/Y_downsample_factor));
    end
end
end

```

Figure 10: RGB Conversion Implementation

This is done since the maximum we have for the Y matrix is every other column, so as a result the maximum conversion we can do is for every other column, below the converted image can be seen with all the white dots, that represent uncalculated values periodically throughout the image, they behave like high frequency noise.



Figure 11: RGB Converted Image



Figure 12: RGB Components from The Converted Image

As we can see, we were able to get a good approximation of the original image, we can see that a lot of the artificing we see in our image for this stage is due to the Green component having scanline like artificing. This could be due to the reason that the equation for the Green component involves a great deal of subtractions, i.e. $G = Y - 0.395 * U - 0.581 * V$, compared to the other two channels that are getting the U and V components getting added to them respectively.

Median Filtering and PSNR

For the final image, Median Filtering was used as it would work best at reducing noise while preserving the image, Gaussian filtering was also considered initially, but quickly discarded due to the fact that we have this noise periodically throughout the Image, not just randomly scattered, In median filtering we can select the middle values and ignore the out of place high or low values with a high large enough kernel size, the below image will show the Median filtering code:


```

%% Median Filtering

disp('Filtering Stage...');

RGB_pre_filter(:,:,1) = uint8(R);
RGB_pre_filter(:,:,2) = uint8(G);
RGB_pre_filter(:,:,3) = uint8(B);
RGB_post_filter = zeros(size(RGB_pre_filter));
window_length = 3;
window = zeros(window_length*window_length);
edgex = (int32(window_length/2));
edgey = (int32(window_length/2));
for depth = 1:3
    if(depth == 1)
        disp('Filtering Stage (R/RGB)...');
    elseif(depth == 2)
        disp('Filtering Stage (G/RGB)...');
    else
        disp('Filtering Stage (B/RGB)...');
    end
    for x = edgex:(width - edgex)
        for y = edgey:(height - edgey)
            i = 1;
            for fx = 1:window_length
                for fy = 1:window_length
                    window(i) = RGB_pre_filter(x + fx - edgex,y + fy - edgey,depth);
                    i = i + 1;
                end
            end
            loc = fix((window_length*window_length/2)) + window_length - 1;
            if(window(loc)>255)
                RGB_post_filter(x,y,depth) = uint8(255);
            elseif(window(loc)<0)
                RGB_post_filter(x,y,depth) = uint8(0);
            else
                RGB_post_filter(x,y,depth) = uint8(window(loc));
            end
        end
    end
end
end

```

Figure 13: Median Interpolation

To determine the appropriate size for the Kernel, and which values to take, a lot of trial and error was required to locate the appropriate values, it was determined that a 3 by 3 kernel was the largest that can be used as others increased the runtime by a large amount, to figure out by how much to offset the array, the writer tried an offset of window_size-1, which gave a values of 32.58 for the PSNR, at the cost of causing minor pixelation around the planes and some of the clouds, when the offset was set to window_size, the planes and the clouds a looked little better, as they contained a lot of white so taking the higher values was better, at the cost of increasing pixelation around the mountain tops, as they use softer colors so lower values. So we can keep the offset to window size – 1 to achieve the highest PSNR.









Figure 14: Offset = window size (PSNR = 32.58 dB)



Figure 15: Offset = window size (PSNR = 25.29 dB)

Conclusion

With this the report is completed, below we will provide some other standard images that have been put through the algorithm, feel free to test the algorithm with your own images !

Initial Image	Processed Image	PSNR(dB)
		44.06
		29.01
		39.15