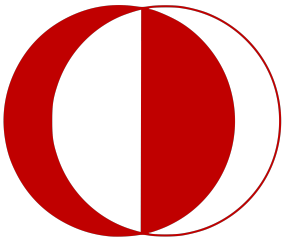
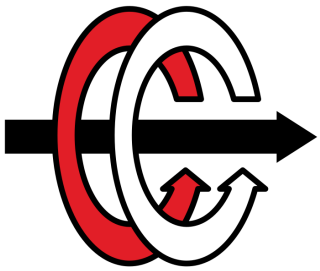



MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

# EE433

# REAL-TIME APPLICATIONS OF DIGITAL SIGNAL PROCESSING

## TERM PROJECT

### GROUP 2


| | |
|---|---|
| ABDULLAH CANBOLAT | 2304244 |
| GÜRAY ÖZGÜR | 2167054 |
| YUNUS BİLGE KURT | 2232395 |


**26.06.2021**

# Contents

# 1 PROBLEM DEFINITION

In this project, two image processing tasks are implemented in LabVIEW. The first task is converting an RGB image to a grayscale image, which is a useful operation in image processing when only luminance information of each pixel is needed in the computations. While performing this operation, user should also see the computation times of both implemented task and LabVIEW's default function. The input image can be read from a file or computer camera can be used. The second task is converting an grayscale image to an RGB image. This task requires a pre-defined colormap for determining RGB values. The colormap should also be changed interactively in the user interface. In addition to implementation of these two tasks, processing the video input and output in real-time is expected, which is also implemented. Throughout this report the implementation of these tasks are explained in detail.

In Section 2, C implemantations of RGB to grayscale and grayscale to RGB conversions are explained, which are later used in LabVIEW implementation. In Section 3, LabVIEW implementation of RGB to grayscale conversion, in Section 3.3, grayscale to RGB conversion, in Section 3.4, and video processing for these conversions, in Section 3.5, are included.

# 2 C IMPLEMENTATION

Listing 1: Header file for the functions

```
1 void rgb2gray(int* src, int total);
2 void gray2rgb(int* src, int total, float* colormap);
```

Two functions that can be seen in Listing 1 are implemented in C for this project. *rgb2gray* function takes an integer pointer *src*, and an integer *total* as inputs and does not return a value. *gray2rgb* function takes an integer pointer *src*, an integer *total*, and a float pointer *colormap* as inputs and does not return a value.

## 2.1 RGB TO GRAY CONVERSION

$$GRAY = 0.299\,R + 0.587\,G + 0.114\,B \tag{1}$$

Listing 2: C implementation of RGB to gray conversion

```
1  void rgb2gray(int* src, int total){
2          int r, g, b;
3          for (int i = 0; i<total; i++){
4          b = *(src+i) & 0xFF; //load blue
5          g = *(src+i)>>8 & 0xFF; //load green
6          r = *(src+i)>>16 & 0xFF; //load red
7          //build weighted average:
8          *(src+i) = (unsigned char)(r*0.299+g*0.587+b*0.114);
9          }
10 }
```

C implementation of RGB to gray conversion can be seen in Listing 2. Owing to the image to array converter block of LabVIEW input images are arrays holding 32 bits for each pixel. In the loop, R, G, B values for each pixel are read and used to determine the grayscale pixel value for total number of pixels iterations. Each pixel is storing 32 bits, first 8 bits are unused, second 8 bits are "R" values, third 8 bits are "G" values and last 8 bits are "B" values. In order to read the "B" value, the operation in the line 4 in Listing 2 is performed for each pixel, we extract the last 8 bits to obtain "B" value. Similarly, by shifting the pointer for 8 and 16 bits, "G" and "R" values are obtained respectively. At the end, new pixel value is assigned using the formula in Equation 1.

IMAQ ColortoImageArray.vi returns a 32-bit integer for each pixel, however, what does the number at each pixel represent is unknown. To determine that, we implemented a little experiment by connecting 2D array to the input of IMAQ IntegertoColorValue.vi and observing its output. Also, using the answer in Reference [1], which is given in the project description, we wrote a similar code to Listing 2 and printed out in the terminal. In LabVIEW, we get the correspondence of 32-bit values shown in Figure 1 and 8-bit R, G, B values which is shown in Figure 2. We write the code in external IDE by printing, R, G, B, respectively, to see whether the red, green and blue values are same. The result is shown in 4 and is the same as LabVIEW output. Therefore, by using the help of external IDE, we justified the placement of R, G, B values in 32-bit integer.

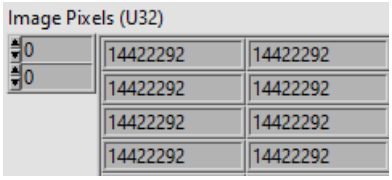



Figure 1: 2D 32-bit values of a picture

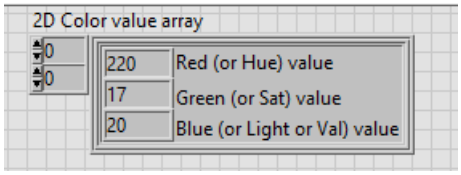



Figure 2: 2D color value array

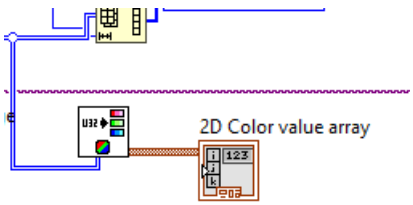



Figure 3: 2D color value array, LabVIEW

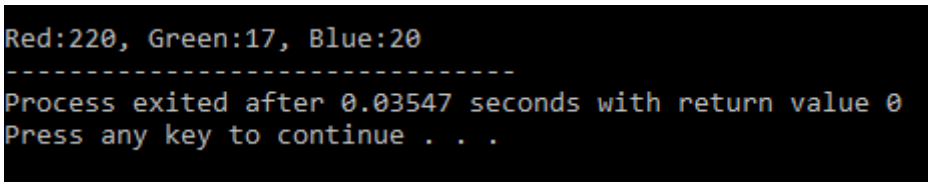



Figure 4: Terminal result of 32-bit to R,G,B values

## 2.2 GRAY TO RGB CONVERSION

Listing 3: C implementation of gray to RGB conversion

```
1 void gray2rgb(int* src, int total, float* colormap){
2         unsigned char r,g,b;
3         for(int i = 0; i<total; i++){
4         r = *(colormap+3*src[i])*src[i];
5         g = *(colormap+3*src[i]+1)*src[i];
6         b = *(colormap+3*src[i]+2)*src[i];
7         *(src+i) = (r<<16) + (g<<8) + b;
8         }
9 }
```

C implementation of gray to RGB conversion can be seen in Listing 3. Owing to the image to array converter block of LabVIEW input images are arrays holding 32 bits for each pixel. In the loop, grayscale value for each pixel are read and with this value a colormap is used to determine the R, G, B values for total number of pixels iterations. Each pixel is storing 32 bits, first 8 bits are unused, second 8 bits will be "R" values, third 8 bits will be "G" values and last 8 bits will be "B" values. In order to write the R, G, B values, the operation in the line 7 in Listing 3 is performed for each pixel, we write the last 8 bits using "B" value. Similarly, by shifting the "G" and "R" for 8 and 16 bits respectively RGB value of each pixel is assigned successfully.

## 2.3 DLL FORMATION

After the preparation of C codes, *image_func.dll* is formed using LabWindows and passed to LabVIEW as explained in the course's LabVIEW tutorial document [2].

# 3 LABVIEW IMPLEMENTATION

LabVIEW implementation will be detailly explained, however, before that in Sections 3.1 and 3.2 the front panel and the block diagram are shown to ease the understanding. Detail explanations can be found in Sections 3.3, 3.4, and 3.5.

## 3.1 FRONT PANEL OF THE PROJECT

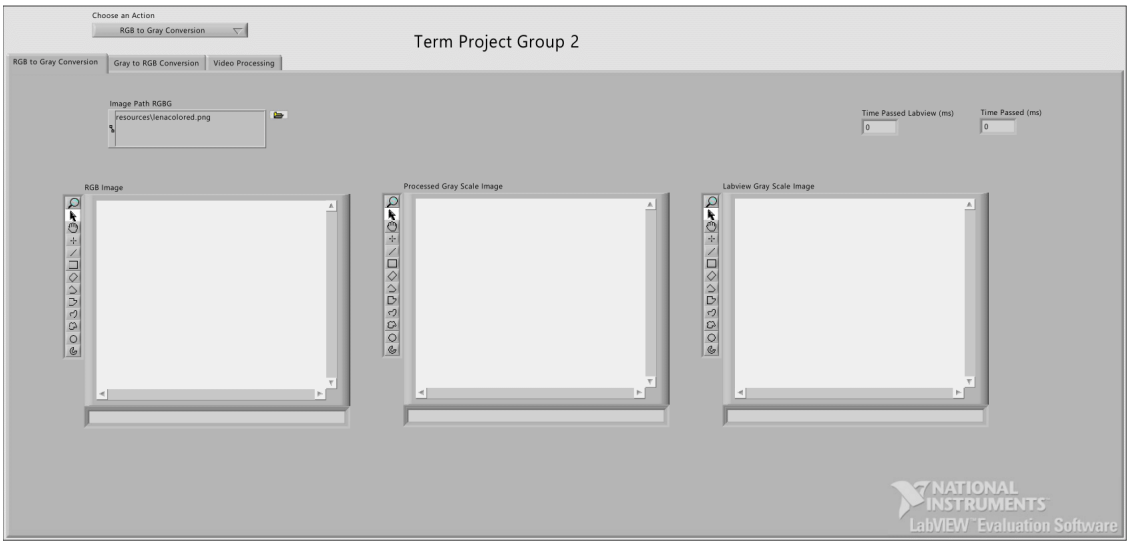



Figure 5: Front panel of "RGB to Gray Conversion"

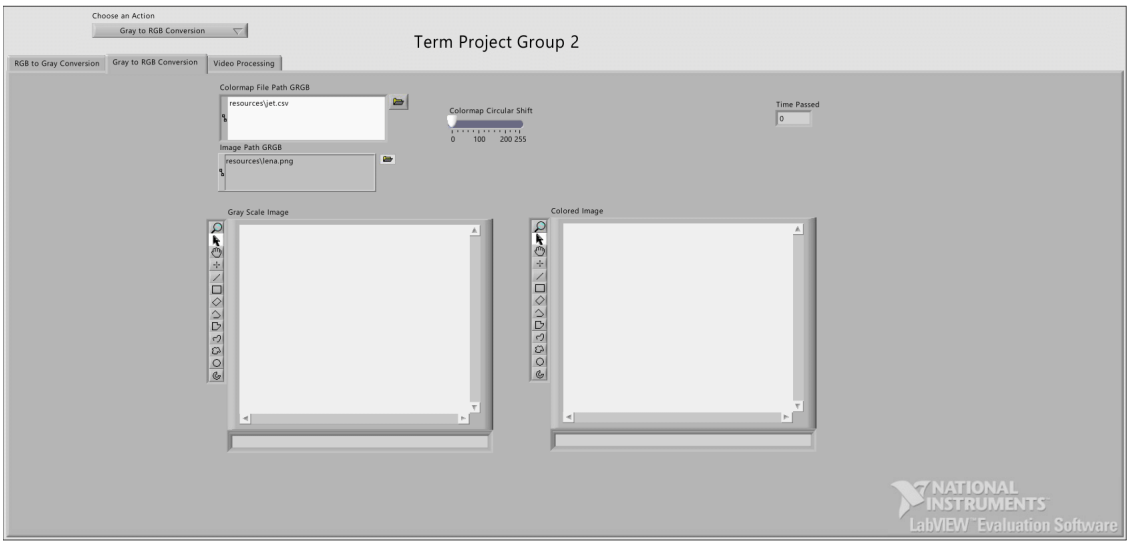



Figure 6: Front panel of "Gray to RGB Conversion"

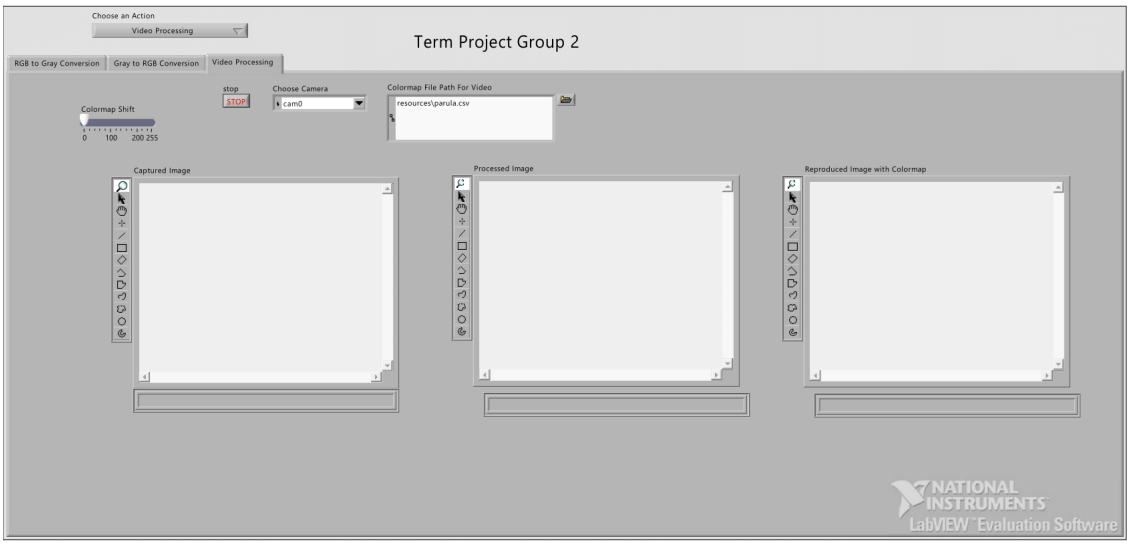



Figure 7: Front panel of "Video Processing"

## 3.2 BLOCK DIAGRAM OF THE PROJECT

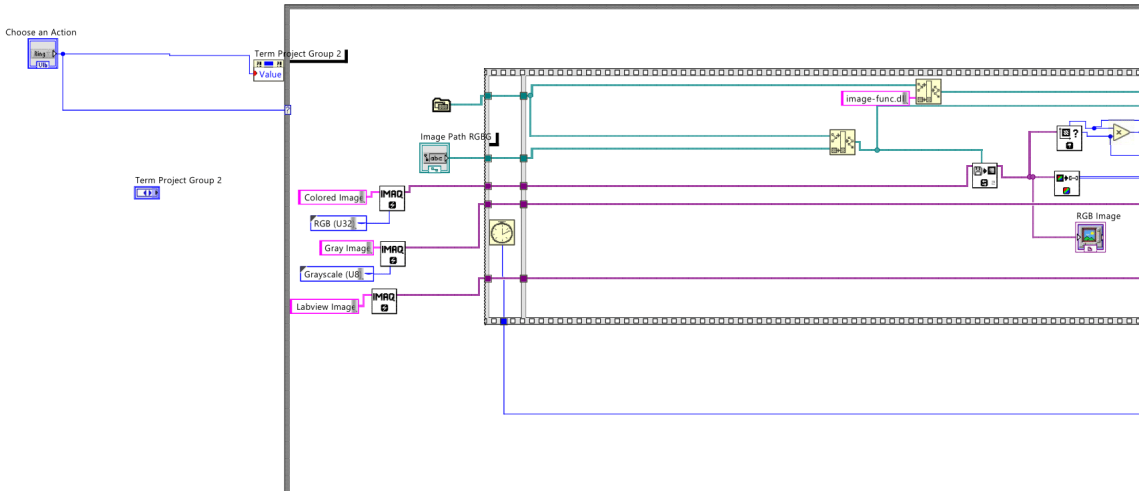



Figure 8: First page of the block diagram of "RGB to Gray Conversion"

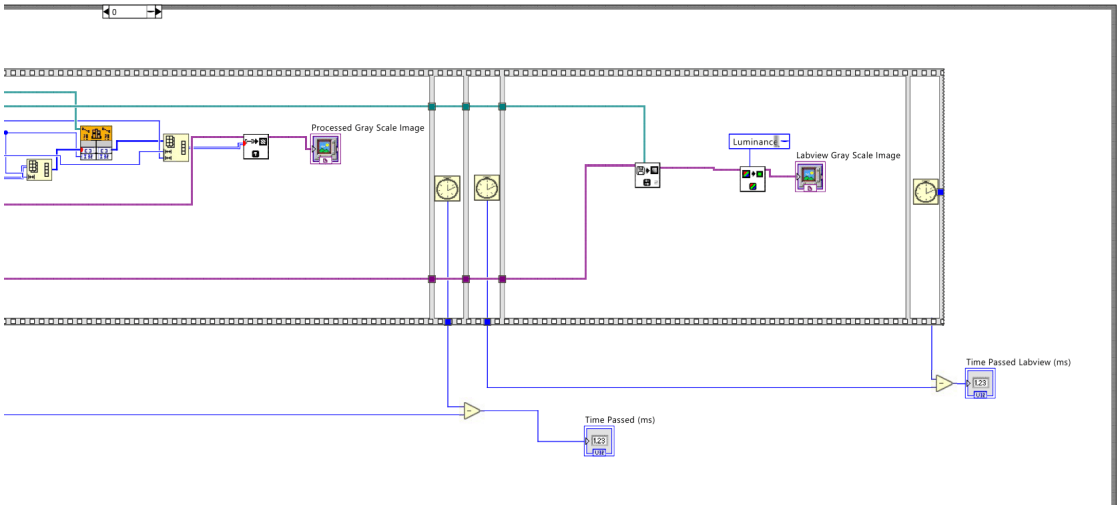



Figure 9: Second page of the block diagram of "RGB to Gray Conversion" for a different photo

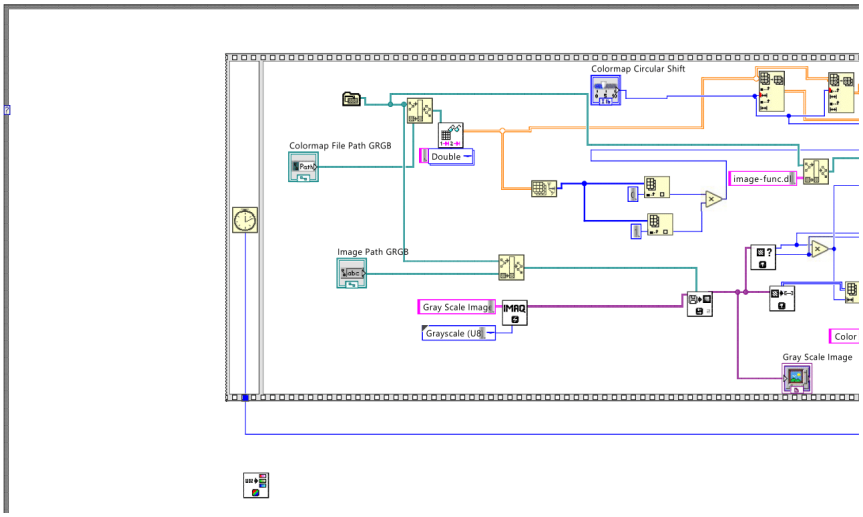



Figure 10: First page of the block diagram of "Gray to RGB Conversion"

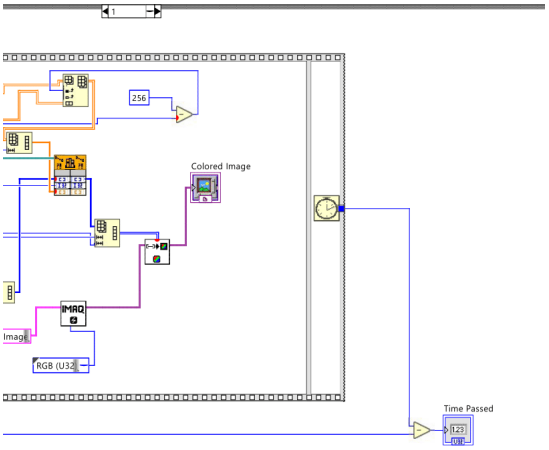



Figure 11: Second page of the block diagram of "Gray to RGB Conversion"

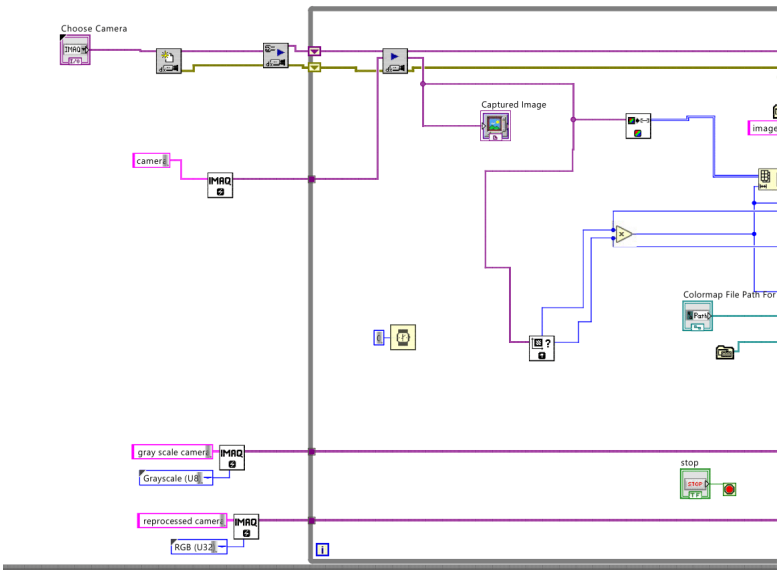



Figure 12: First page of the block diagram of "Video Processing"

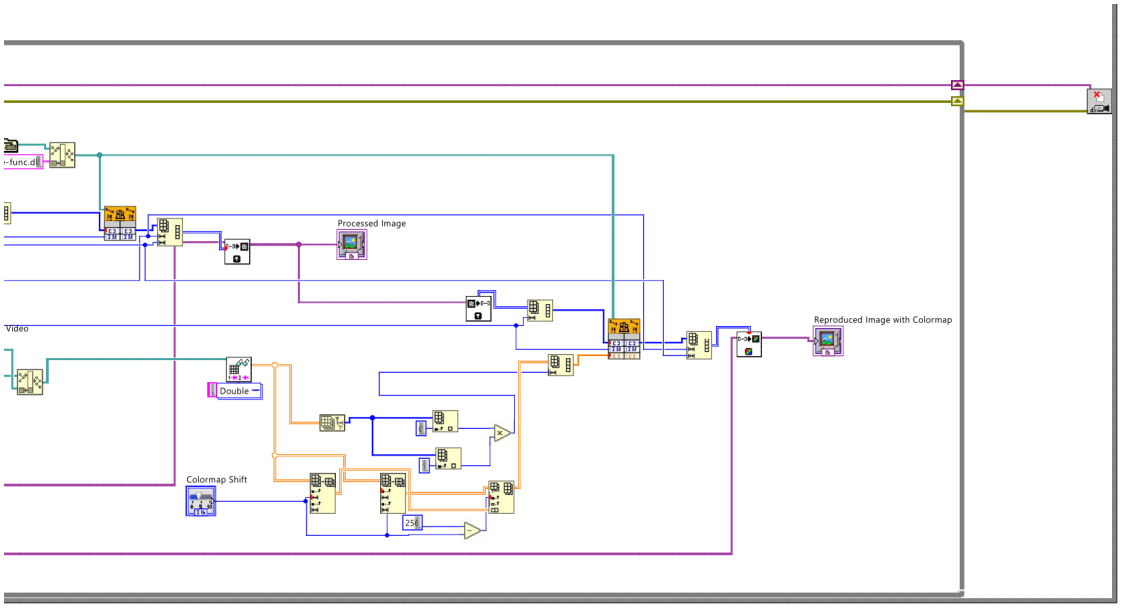



Figure 13: Second page of the block diagram of "Video Processing"

## 3.3 RGB TO GRAY CONVERSION

### 3.3.1 THEORETICAL BACKGROUND

For our LabVIEW implementation, the program should include three different scenarios, that's why a case structure is used in the block diagram as seen from Figure 8. One of the cases is the RGB to grayscale conversion. As explained in Section 2, a C implementation is done accordingly and converted to DLL file with LabWindows, which is then added to LabVIEW by using a call library function node. A call library function node, which is shown in Figure 14, calls a DLL or shared library function directly by indicating the path of the DLL. Here, the return type and the parameters should be specified in LabVIEW. For the image, an array data pointer of the type signed 32-bit integer is used. The screenshot of the panel can be seen from Figure 15. This call function node does the conversion from RGB to grayscale as depicted in 2.1.

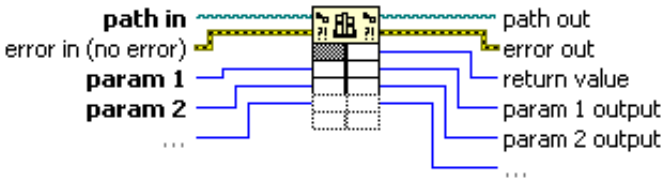



Figure 14: Call Library Function Node

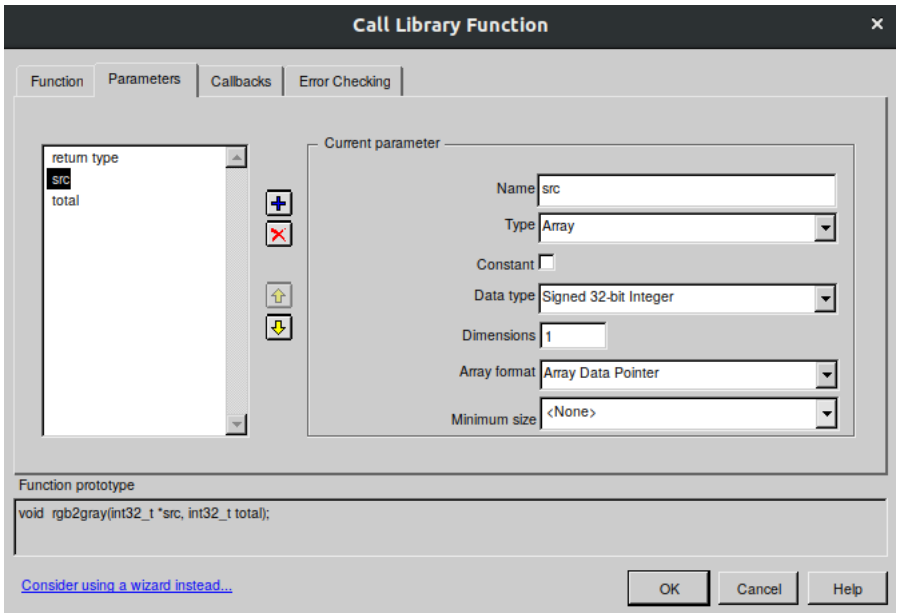



Figure 15: Call Library Function

The flowchart of the code can be given as:

- IMAQ Create, as in Figure 16, creates a temporary memory location for an image, three image initilizations are done for the image, our grayscale conversion and LabVIEWs grayscale conversion.
- IMAQ ReadFile, as in Figure 17, reads an image file to convert.
- IMAQ GetImageSize, as in Figure 18, gives information regarding the resolution of the image, which is used to calculate the total length of the image when it is flattened.
- IMAQ ColorImageToArray, as in Figure 19, extracts the pixels from a color image into a 2D array, whose values are generally unsigned 32-bit integers, which is then flattened and given to call library function node as an input.
- After the conversion, IMAQ ArrayToImage, as in Figure 20, creates an image from a 2D array, and it is shown in the front panel.

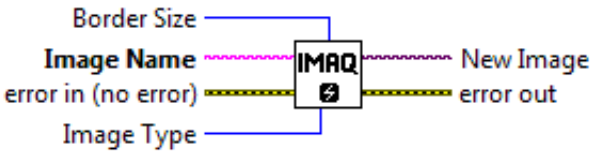



Figure 16: IMAQ Create VI

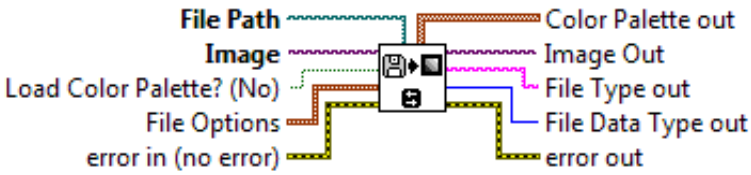



Figure 17: IMAQ ReadFile VI

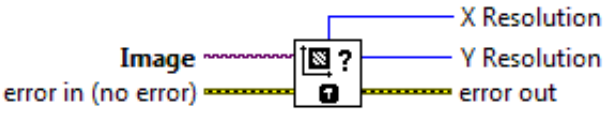



Figure 18: IMAQ GetImageSize VI

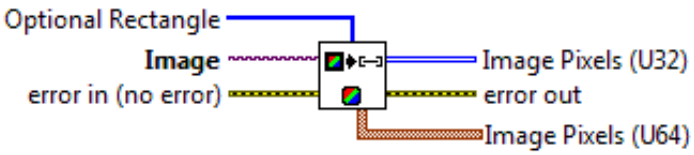



Figure 19: IMAQ ColorImageToArray VI

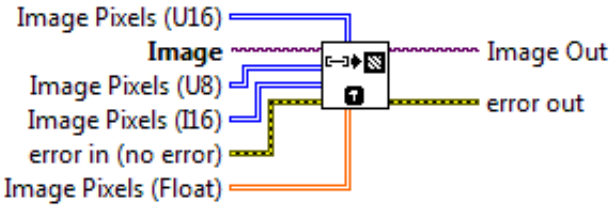



Figure 20: IMAQ ArrayToImage VI

### 3.3.2 RESULTS

A timer is used to calculate the time difference of the beginning and the end of the procedure. Since grayscale only contains luminance (brightness) information and no color information by extracting luminance information with LabVIEW, another grayscale image is obtained. Moreover, the time is calculated in the same manner and compared. As can be seen from Figure 21, at the left, a colored image of a lady, Lena, is used. In the middle of the figure, processed grayscale image is shown, which is obtained in 14 ms. At the right, another grayscale is image is obtained by using built-in functions to extract the luminance information, which took 13 ms to be completed. For another image, the procedure can be repeated and it is shown for a RGB colored disk image to see the effect of conversion in Figure 22.

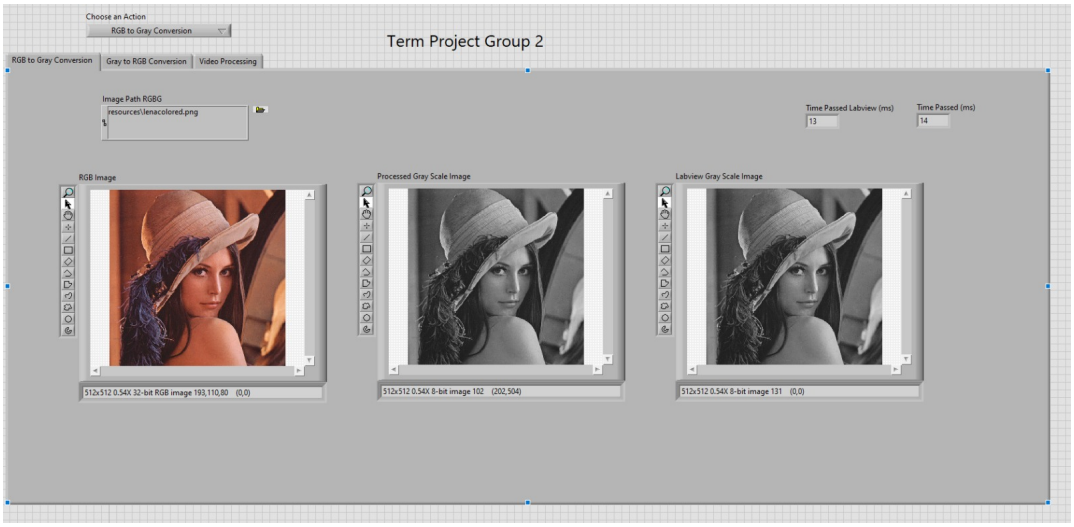



Figure 21: Front panel for results of "RGB to Gray Conversion"

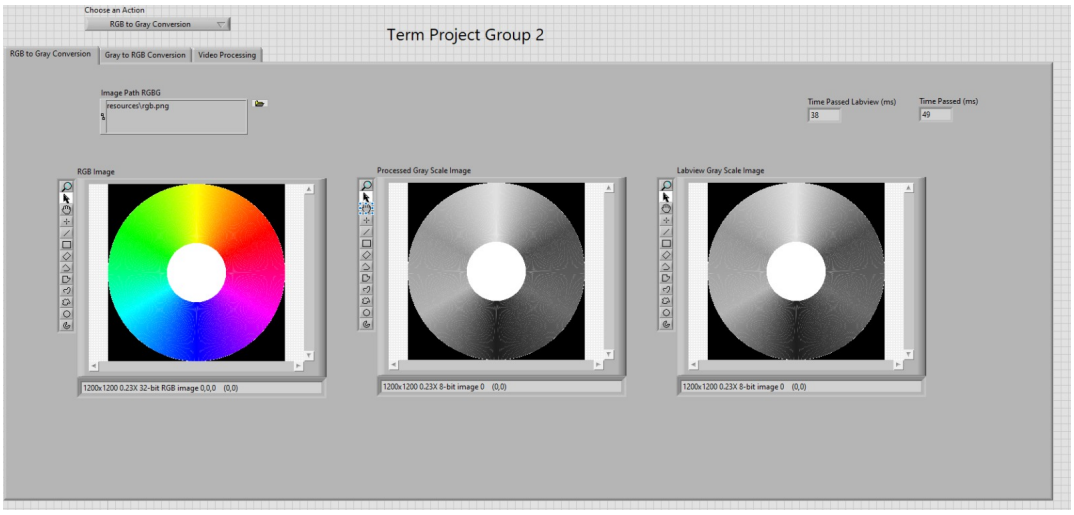



Figure 22: Front panel for results of "RGB to Gray Conversion"

## 3.4 GRAY TO RGB CONVERSION

### 3.4.1 THEORETICAL BACKGROUND

As explained in Section 2, a C implementation of grayscale to RGB is done accordingly and converted to DLL file with LabWindows, which is then added to LabVIEW by using a call library function node. A call library function node, which is shown in Figure 14, calls a DLL or shared library function directly by indicating the path of the DLL. Here, the return type and the parameters should be specified in LabVIEW. For the image, an array data pointer of the type signed 32-bit integer is used. For the colormap, a 4-byte single pointer is used. The screenshot of the panel can be seen from Figure 23. This call function node does the conversion from grayscale to RGB as depicted in 2.2.

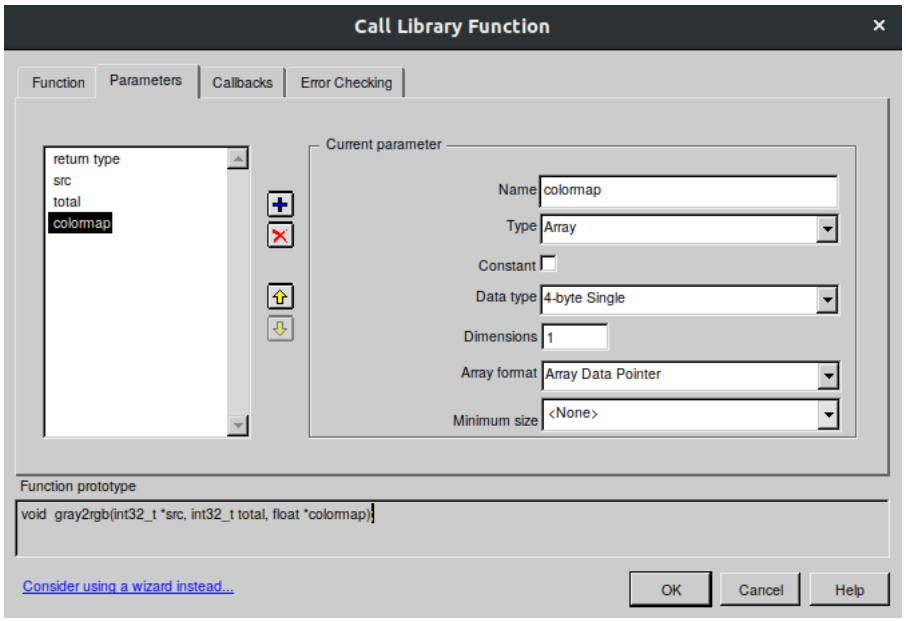



Figure 23: Call Library Function

The flowchart of the code can be given as:

- IMAQ Create creates a temporary memory location for an image, two image initilizations are done for the image, and our RGB conversion.
- IMAQ ReadFile reads an image file to convert.
- A colormap file path is specified and loaded from a .csv file. Furhermore, a circular shift operation is done to obtained different colormaps from a single colormap.
- IMAQ GetImageSize gives information regarding the resolution of the image, which is used to calculate the total length of the image when it is flattened.
- IMAQ ImageToArray extracts the pixels from an image into a LabVIEW 2D array. This array is encoded in 8 bits, 16 bits, or floating point, as determined by the type of input image, which is then flattened and given to call library function node as an input.
- After the conversion, IMAQ ArrayToImage creates an image from a 2D array, and it is shown in the front panel.

### 3.4.2 RESULTS

A timer is used to calculate the time difference of the beginning and the end of the procedure. As can be seen from Figure 24, at the left, a grayscaled image of a lady, Lena, is used. At the right, colored image is shown, which took 59 ms to be completed. For a different colormap, the procedure can be repeated and its effect is shown in Figure 25. For another image, the procedure can be repeated and its effect is shown in Figure 26.

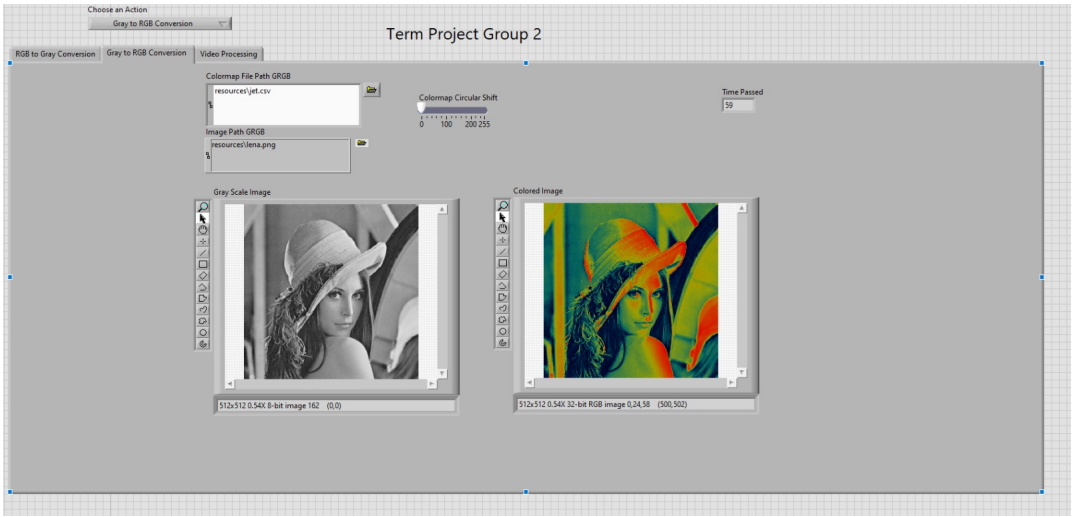



Figure 24: Front panel for results of "Gray to RGB Conversion"

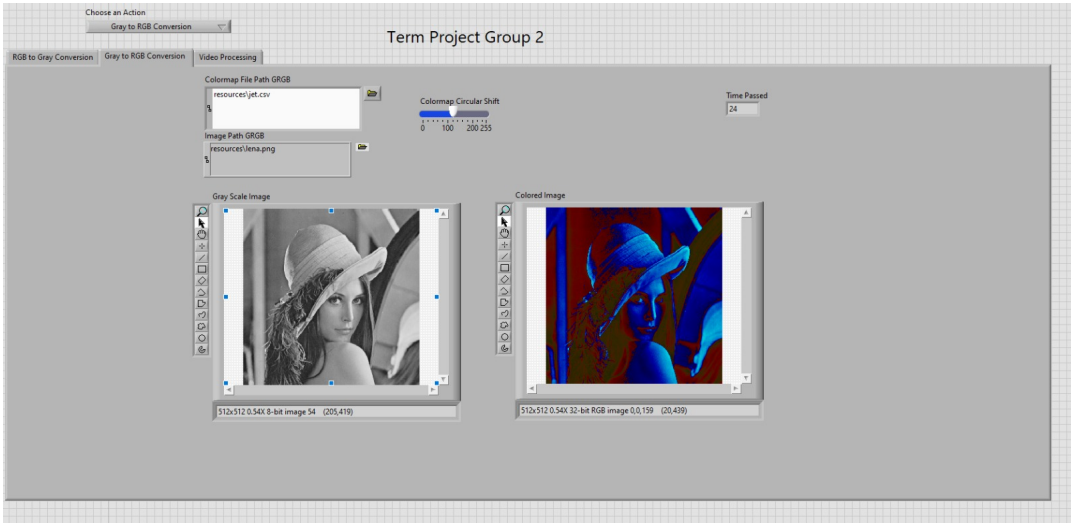



Figure 25: Front panel for results of "Gray to RGB Conversion" for a different colormap

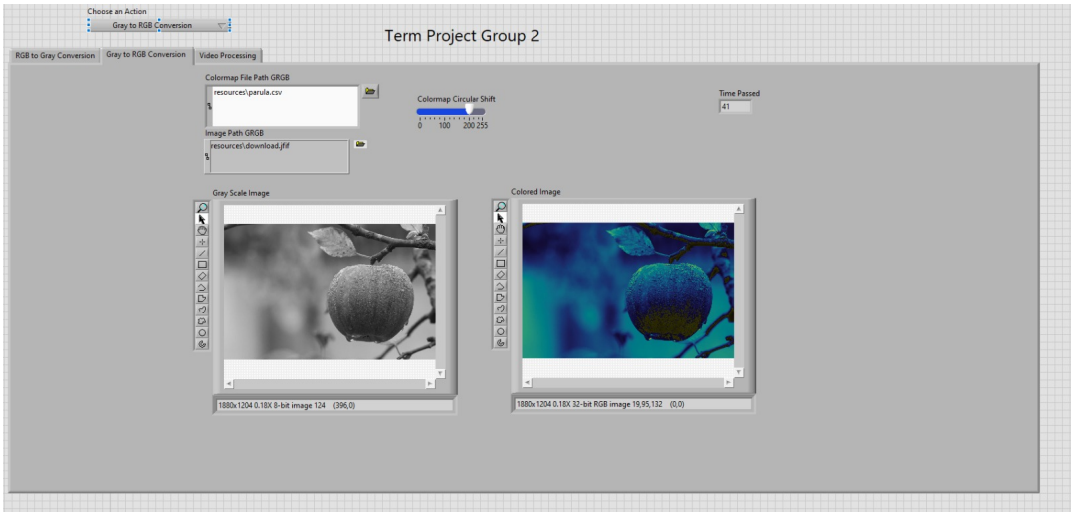



Figure 26: Front panel for results of "Gray to RGB Conversion" for a different photo

## 3.5 VIDEO PROCESSING

### 3.5.1 THEORETICAL BACKGROUND

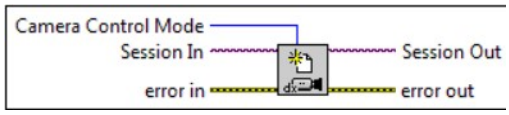



Figure 27: IMAQdx Initialize VI

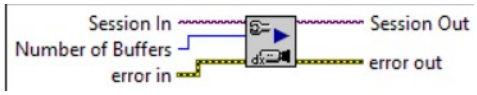



Figure 28: IMAQdx Configure Grab VI

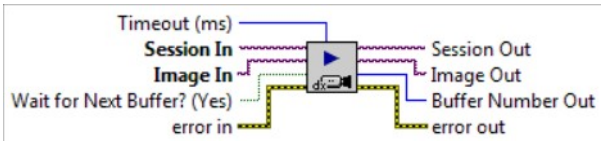



Figure 29: IMAQdx Grab2 VI


Figure 30: IMAQdx Close Camera VI

Main difference of video acquisition is the usage of IMAQ dx blocks. Using four IMAQ dx VI blocks given in Figures 27, 28, 29, and 30 sequentially images can be captured. By putting Grab2 block inside a loop images can be captured, processed and processed image can be shown continuously. After obtaining the images using these blocks, then utilizing the same process that is applied in RGB to Gray conversion in Section 3.3 and using the grayscale image a colormapped image can be obtained as it is done in Section 3.4.

### 3.5.2 RESULTS

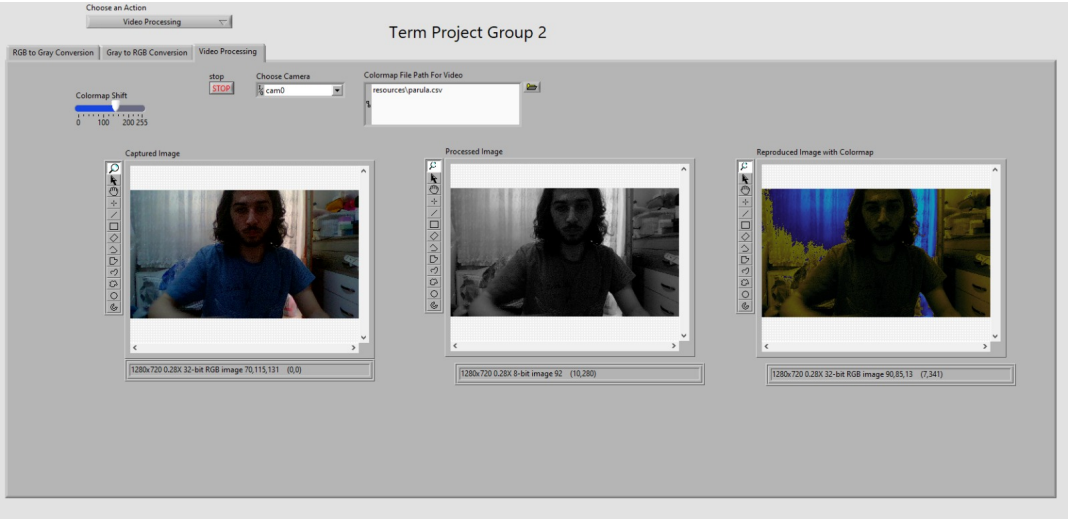



Figure 31: Front panel for results of "Video Processing"

A timer is used to calculate the time difference between beginning and the end of the procedure. As can be seen from the Figure 31 the image captured from the user camera can be converted to a grayscale image which is then converted to an RGB image using a pre-defined colormap. This operation can be done continuously.

# 4 DISCUSSION

In this project, the implementation of two image processing tasks are done in LabVIEW. In the implementation both LabVIEW blocks and C codes are used. It is seen that complicated image processing tasks can be simplified using the power of programming languages. On the other hand, using pre-defined LabVIEW blocks can be faster. In this case, implementation takes more time than pre-defined LabVIEW implementation for RGB to grayscale conversion, however, in some cases finding pre-defined implementations might not be possible or they might be slower. Hence, using LabWINDOWS, DLL files can be created and used in the LabVIEW implementations.

Our observation can be summarized as follows:

- Overall complexity of implementing a function in Labview or in C are nearly same and one should use Labview function if implementing it in C would not be efficient.
- Labview stores Image datatype in a special data structure, probably a linked list, because color values shown in Figure 2 is an array of clusters. Thus, conversion from Image to 2d array takes time in our C implementation, which increases complexity.
- Image loses information of color by turning RGB values to grayscale and using a colormap to get RGB image back may not be helpful for most of the case, because if we investigate Equation 1, we see that if gray value is specified, we have 1 equation with 3 unknowns. Therefore, there will be 2 free variables if color information is not given at the beginning. Thus, grayscale to RGB conversion should be handled carefully with maybe several colormaps for each specific region in the image for reconstruction to be meaningful.
- In videos, images are sampled with constant time intervals. Most of the webcams either take 30 frames per second (FPS) or 60 FPS, depending on the camera. Algorithm complexity of a real time image processing algorithm should consider the constraints of image size taken by the camera and the FPS of the camera. Therefore, any algorithm should be completed within the frames to prevent lagging of the processed video. However, with increasing image sizes the time required for an algorithm increases. Thus, in real world scenarios some image processing algorithms may or may not be implementable in real time, considering the complexity constraint.
-

# 5 REFERENCES

[1] https://forums.ni.com/t5/LabVIEW/Procesing-image-in-Labview-using-a-dll-in-c/m-p/3946407?profile.language=en

[2] https://odtuclass.metu.edu.tr/pluginfile.php/352051/mod_resource/content/1/EE433%20Real-Time%20Applications%20of%20Digital%20Signal%20Processing.pdf