CSC 21100 Homework #2

Purpose

The purpose of this paper is to investigate and describe the processes involved in encoding/decoding digital and a limitation of our computer system. We will also describe the processes involved in converting analog, or real-world signals, into digital, or abstract representations for the computer to store/manipulate. The material we investigate today will, in turn, help us better understand how real-world data is transformed into computable data and the data structures involved in storing and manipulating the resulting data on computer systems.

Introduction

Modern households contain many computers. from the kitchen to the living room, computers are in constant work. The dish washer, printer, and hand-held calculator on your living room stool are examples of systems that contain a computer. These computers record the physical phenomena surrounding them by sampling how the properties of matter alter over time. The sampled data is called "Analog data" and devices, or electronic circuits, that perform these tasks are called Analog devices.

To get a better idea of how computers record, transforms, and reproduces real world data, we investigate the processes involved in recording a picture from a camera. We will apply a linear transformation on the picture by manipulating its color intensities and contrast, finally, we blend two pictures together. Also, we will demonstrate how we can manipulate digital data to compensate for the limitations of hardware through a non-linear transformation, or gamma correction. When we are satisfied with our transformations, then the resulting digital data will be sent to the monitor, at which point it will be converted by to analog signal for display.

Our journey begin in the real-world, or physical world. In this physical world, there exists matter, which falls under 4 different states, solid, liquid, gas, or plasma. Though plasma may just be an extension of the gas phase under certain rare or specific conditions, we, nevertheless, distinguish it from the other 3 states of matter. Materials such as plastic, ice, sand, metal, or wood, fall under the solid state. Items such as, water and soda beverages fall under the liquid state. The air in the atmosphere, or the wind surrounding us are in the gas phase, and finally, turbulence caused by heat or total ionization of atoms fall under the plasma phase.

The reason why I mention these states is because analog devices must exist in one of these states. We can define an analog device as physical matter in one of the 4 phases, but usually in the first-three phases of matter. Most of the time, matter exists in a single state. However, some particles can exist in two different states of matter. In fact, the property of the real world our analog device will record, photons, or light, falls under two different phases of matter. This duality nature of light was described by Albert Einstein in 1905 through his "law of the photoelectric effect", one of his many great accomplishments.

Heinrich Hertz was the first to show that light, which was considered as an electromagnetic wave, or energy, can eject electrons from materials. Hertz began a revolution that touched multiple cores of science, particularly physics and chemistry. Einstein later defined light as "photons", and he described them as discrete quantities, or quanta, which he then used to solve a paradox between quantum mechanics and Newtonian mechanics. This paradox was called "blackbody radiation", and it was discovered by Max Planck, another great scientist whom influenced Einstein's work greatly.

Before describing how cameras record analog signals, we need to define how the properties of matter changes. As we noted before, matter exists in one of the four phases, solid, liquid, gas, or plasma. When matter transitions from one state to the other, it is effectively changing its nature, or properties. But how does matter transition from one phase to another or change its properties? The answer is energy. When a particle in one of the four phases of matter absorbs energy, then a chemical reaction is triggered. To understand how this chemical processes causes a system's properties to change, we must observe the chemical equation.

Let's take for example the situation that occurs under the halogenation of alkanes. In other words, let's see what happens when an alkane, which is a molecule containing a sequence of carbon and hydrogen bonds, such as ethane, reacts with a halogen, or a group 7 element, such as chlorine:

$$CH_3CH_3 + Cl_2 \text{ ----- hv (UV light) ---> } CH_3CH_2\text{-}Cl + HCl$$

Therefore, if we mix ethane gas with chlorine gas and shine a beam of UV light at the system, then they will become chloroethane (or ethyl chloride) gas and hydrochloric acid gas. Ethyl chloride has a boiling point of 12.3^{o}C while ethane has a boiling point of -89^{o}C. Therefore their properties has changed, although they are still in the gas phase of matter at room temperature. Also, hydrochloric acid has its own boiling point which has different properties from everything else in the system. This is an example of how energy can shift the properties of matter, and this is the foundation of chemistry, and of course many other fields of science.

How can be explain this phenomena or chemical reaction? Well, we can say the heat from light excited the molecules ethane and chlorine. Eventually, the heat from the light was enough to break the activation energy of this reaction and because both chloride atoms (each of the Cl in CL_2) have such a high affinity for electrons (because they are so electronegative and seek to fill their final orbital with an electron), the vibration caused the chlorine molecule to approach the electron dense ethane molecule, therefore causing the ethane molecule to break apart and eject a hydrogen out and filling one of the chloride's empty orbital with the electron it had once bonded to the hydrogen atom. The second chloride took over the remaining electrons that once was shared with the previous chloride and now has a full orbital but with a net negative charge. Therefore, the hydrogen, which has an empty orbital and a net positive charge is attracted to the net negative chloride and forms a bond to produce HCl.

Let's observe another example of matter changing properties. Let's see what happens when a highly substituted alcohol reacts is mixed with an alkyl halide (a group 7 element) at room temperature:

$$(CH_3)_3C\text{-}OH + HCl \text{ <-- hv (light from the sun) --> } (CH_3)_3C\text{-}OH_2^+ + Cl^-$$

This reaction, which proceeds to the right and maintains dynamic equilibrium between the molecules on the left-hand and right-hand side of the reaction is another example of an alteration to the properties of the matter on the left side of the reaction, otherwise known as the reagents or starting product of the reaction. In this case, the product is not very sable due to the fact that the net molecules have a plus and minus charge. Therefore the product will exist in an ionic-bond state, while the original reagents were held by a strong covalent bonds. Most reactions maintain a dynamic equilibrium between the reagents and product. Even the reactions that have a strong tendency to proceed to the right (from reagent to product), though it might not be as obvious.

So we now know how matter's properties are altered. If we introduce enough energy to start a chemical reaction, or to break the activation energy, then the system will be transformed into another with different properties. We can record these transformations through chemical reactions. Of course applying chemistry is the only way to make such record these changes. For example, chemistry is the study of electron movement in chemicals (or chemical electrons), while other fields such as electrical engineering apply engineering principles to the movement of electrons (or electron engineering). Therefore, when we speak of the foundations of science, we can sort of say that science is the study of electrons!

Now we wonder how computers digitalize these measurements for storage and manipulation. The best way to understand how this data is transformed into signals that computers understand is to think of photovoltaic reactions. These type of reactions are just another subset of chemical reactions studied. We start by analyzing an electrochemical reaction. We know that metals tend to give off their electrons easily, while halogens in group 7, or other non-metal atoms near that group except the stable atoms at group 8, tend to pull electrons to themselves. Therefore, let's take a metal such as iron and mix it with a non-mental and very electron negative molecule such as chlorine. Iron is a metal which has 8 electrons in its valence shell, however it is not stable like the group 8 non-metal noble gases due to the fact that its d-orbital is not completely filled. More information on how this chemical orbitals work and why iron, with 8 valence electrons, does not have a full outer set of valence electrons can be further investigated in organic chemistry (Organic Chemistry I if taken at CCNY) or physical chemistry (Physical Chemistry II if course is taken at CCNY). So if we mix iron with chlorine we get the following reaction:

$$Cl_2 + 2Fe^{2+} \rightarrow 2Cl^- + 2Fe^{3+}$$

To explain how this reaction takes place, a "half-reaction" mechanism was proposed:

$$Fe^{2+} \rightarrow Fe^{3+} + e^- \text{ (Cathode Cell, Oxidized)}$$

$$Cl_2 + 2e^- \rightarrow 2Cl^- \text{ (Anode Cell, Reduced)}$$

Because we are dealing with ionic interactions which give up electrons or accept electrons, what if we separate these two reactions by a salt bridge? In doing so, the electrons from the half-cell containing the iron reaction will need to travel across the salt bridge to reach the chlorine half-cell. This movement

across is called current, and this is how batteries work. These sort of reactions are what forms the foundations of computers. To Describe this process in terms of computers, we monitor how the voltage changes across time. Of course the process is much more involved because we need to actually consider a special kind of reactions that occurs between a semiconductor and non-metal, or other semiconductors. However, the concept is very similar.

The voltage changes over time depending on how current flows. If we have a greater flow of current, then we have higher voltage. If the flow of current is low, then we have less voltage. This concept is best described by Faraday's law: V = IR, where V is voltage, I is current, and R is the resistance against the flow of current. Depending on the nature of the reaction and the concentration of the reagents in our reaction, we notice that the voltage of each half-reaction changes. Therefore, contributing to a change in the overall voltage of the system, and of course we must also account for the external factors.

Computer systems are made of many transistors, which undergo these sort of half-reactions. depending on the net voltage in each transistor, which is determined by the combination of the half-reaction equations into the net equation, then that transistor is rather in the "High" state or "Low" state. Another way to represent this is by 1 or 0. Therefore, a transistor can take on two states:, 0 or 1. To account for these states, we can define the transistor a set consisting of two elements, 0 and 1, and at any given time, this set can output a single element, 1 or 0, to describe itself. Therefore, given that the transistor can only be in a single state at once, we can represent all the possible states the transistor is capable of achieving as a permutation of 2 elements chosen 1 at a time. That is:

$$T_{states} = P(2, 1) = {}_2P_1 = 2! = 2$$

Because a computer is made up of many transistors, we say that the transistor is the smallest unit of information the computer can work with. Because the transistor can only define a single state out of two possible states at any given time, we define the transistor as a binary unit. Therefore, the smallest piece of information a computer system, which is made up of many transistors can work with is a binary unit, and we call this a bit. Therefore 1-bit of information, as far as the computer is concerned, can be used to encode 2 or describe two possible states. To determine how many bits of information we are working with, generally speaking, we can Daniel Bernoull's formula as:

$$Bits = Log_2X$$

Where Bits is how much information you have or are getting, and X is the number of discrete states you are encoding or representing. Therefore, a computer will use 1-bit of information to represent two pieces of data. Depending on the configuration of the transistor, if the half-reactions taking place in its cells produces very low or no net voltage, then the resulting electromagnetic field may not be strong enough to force the current to pass through its bridge. Therefore, we denote such a state "0". However, in the opposite case where current passes through the transistor, we denote it as "1". If we arrange our transistors in a certain way, then we can hold onto its last state for later use. Putting together many such arrangements of transistors, gives us a way to store many units of data, and each unit is capable of two possible states. Therefore, by the rule of product, if we have 8 such units in a sequence, then the resulting system is capable of

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256 \text{ states}$$

Therefore, if we have an 8-bit system, then we can work at most with 256 different elements or objects at any given time. Now let's see how this sort of system allows us to build a very simple camera.

In the general sense, and simplistic view, a camera has three major components. Each component digs deep into a separate core of science, optics, chemistry, and electrical engineering. Just like the reactions we mentioned earlier, the process which records an image from the physical world depends on a chemical reaction taking place, because only when electrons move can be describe the properties of matter as changing and record the magnitude of change. A simple camera would concise of a material which contains many cells, or units, where electrochemical reaction scan take place. To begin the process, light reflects off the surface of surface of the particles in the physical world. These particles absorb a small fraction, or sometimes the entire, light. Eventually, the reflected light rays will hit the lens of the camera.

The lens of the camera is geometrically oriented in such a way that the light reflects directly to the surface which contains the electrochemical cells. The number of these cells determine the quality/resolution of the camera. When a light ray hits one of these cells, it triggers an electrochemical reaction similar to the electrochemical reaction we described above. The result reactions produces a small voltage, which depends on the energy of the light ray. Because the particles in the physical world tend to absorb different amount of light, we expect the voltage levels to be slightly different. The voltage level delivered by the electrochemical cells determine how bright the pixel corresponding to that cell will be. The range of values a decent camera would produce is between 0 and 16777215, or 2^{24}-1. These levels are divided into three color values ranging from 0 to 255 (or 2^8-1), which we denote as red, green, and blue. This workings of this process is called "analog", because it samples the real world for some sort of property change and triggers its results periodically, or over cycles.

Therefore, our analog compartment triggers a certain voltage based on the properties of the real-world. This voltage is then interpreted as a sequence of "high" or "low" states in a sequence of transistors, depending on whether or not current passes through them. The sequence of 1 (high) or 0 (low) states these transistors form is then interpreted as a value in a particular range. If the transistors that are reconfigured made up a system consisting of 24-bit of information, then the configuration they make up is interpreted as one of the numbers between 0 and 2^{24}-1, inclusive of 0 and 2^{24}-1. Therefore, if the resulting configuration resulted in the pattern:

0000 1111 0000 1111 0000 1111

Then it would be interpreted as number 986895 out of 2^{24}-1. Since we decided to interpret this intensity number as three different 8-bit color channels: red, green, and blue, we use a group of 8 patterns to represent each color value. Therefore, we would interpret this number as:

0000 1111 0000 1111 0000 1111

This binary representation can be converted into the 3-tuple values:

<15, 15, 15>

Therefore, we say that this signal as produced a pixel value consisting of the following intensity values of each of the color channels:

Red = 15

Blue = 15

Green = 15

and we would get a R,G,B value for each electrochemical cell on our surface. Therefore, if we only have 1 electrochemical cell, then we can say that we have a 1x1 pixel image, and this very small image would consist of a single red dot. Now let's say we have 100 such electrochemical cells. Let's say we arrange them such that there are 10 cells per row, and there are 10 such rows. Then we can say that each cell of the surface corresponds to one of the pixels of a 10x10 image. This, in simple terms , is the concept behind a very simple camera system.

Now we know that a computer consists of many, or millions, of transistors arranged in particular ways, such that each configuration corresponds to some sort of representation of a number, such as the configuration our simple camera caused when a reaction took place in one of its cells. When we save data to our computers, we are basically saving a configuration of transistors for later, of course there are other ways to save our configurations, such as on CD-ROM drives and hard drives, and each of these will have a similar switching or pattern system imprinted on them to produce the expect high and low sequences we saved. Therefore, when we load saved data, we are basically reading a configuration pattern we registered earlier.

Many operating systems also add an alpha channel for transparency. This alpha channel turns our RGB color values into RGBA values. If we assign 8-bits per color channel, then we must use 32-bits, or 4 bytes, to represent a single pixel. My monitor is set to a resolution of 1366 x 768, though it can go more, I like this resolution because the items on my screen are just the right size for visibility. Therefore, my monitor requires 1366x768x4 = 4,196,352 bytes of memory to display the desktop. This is equal to approximately 4 megabytes of memory if this is a 2D image!

So how are picture from the camera manipulated through a computer to produce some kind of desired effect in the record image?

Let's say we have an image called **car_pic1.jpg**, and we would like to decrease its color intensity. To do so, we must load the jpeg file into our computer's memory. Upon doing so, we must decode the jpeg file into the standard RGB representation. Why decode? because RGB representations of images, such as the one above (which is 800x450) takes a lot of space on our storage devices. Therefore, we compress our RGB representation into another representation, in this case we jpeg, which is much smaller in size. Some compression methods can reduce the image size without losing quality. These compression methods are called "lossless compression". Jpeg, however, is a lossy compression standard. Therefore,

we will lose some quality on compression and de-compression of our data. But this is perfectly fine because the pros far outweighs the cons.

How does the jpeg standard work? In a brief summary, we can implement jpeg compression by following the scheme:

1. Load RGB image into memory.

2. Color space transform the RGB data into YUV444 space. Therefore, we are maintaining the color channels exactly. This phase is lossy due to round off errors (because we working with float factors).

3. Sample down our YUV444 data into YUV420. Therefore, we are maintaining our Y (luminance) channel exactly per pixel, but sampling Cr and Cb (chrominance red and blue) only once every four luminance sample.

4. Apply Discrete Cosine Transform (DCT), which is mathematically equivalent to rotating our data points. In actual implementation we would take an integer based approach called Fast Discrete Cosine Transformation (FDCT).

5. Quantize our data points. This can be done by just dividing every point by a scalar factor (for example dividing every element by a 2, or 3, etc .. usually up to 32 for h264). This effectively zeroes out some of the data points, therefore, this is our most lossy phase.

6. Zig-Zag re-arrange our quantized data for preparation of run-length encoding.

7. Run-Length encode our zero runs. This is the first compression step where represent a run of zeros by a single value.

8. Entropy encode the run-length data, such as Huffman encoding the data.

9. Write the file structures headers data, quantization table, and Huffman tables for each of the Luminance and chrominance channels (one table for both chrominance channels).

To decode jpeg images, we run the process above backwards. See the Last few pages of this paper for a very simple jpeg decoding algorithm I wrote in C.

Going back to how we process our jpeg image through a computer, we have the following steps:

1. Load **car_pic1.jpg** into memory.

2. Decode the compressed data back into the RGB representation.

3. Apply image transformation algorithm to the RGB data.

4. Save image data into a file or send the data to the monitor for display.

Let's say we would like to decrease the brightness and add contrast to our image, then we can do so by apply the following mathematical function:

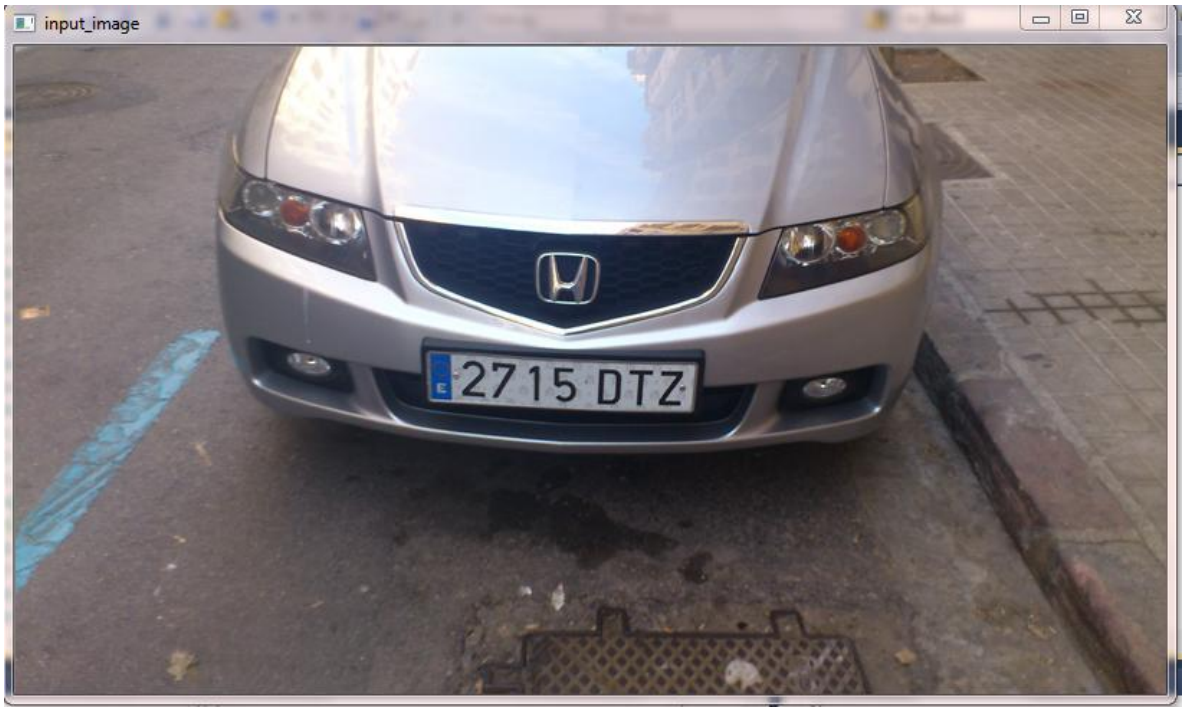
$$g(x) = a*f(x) + b$$

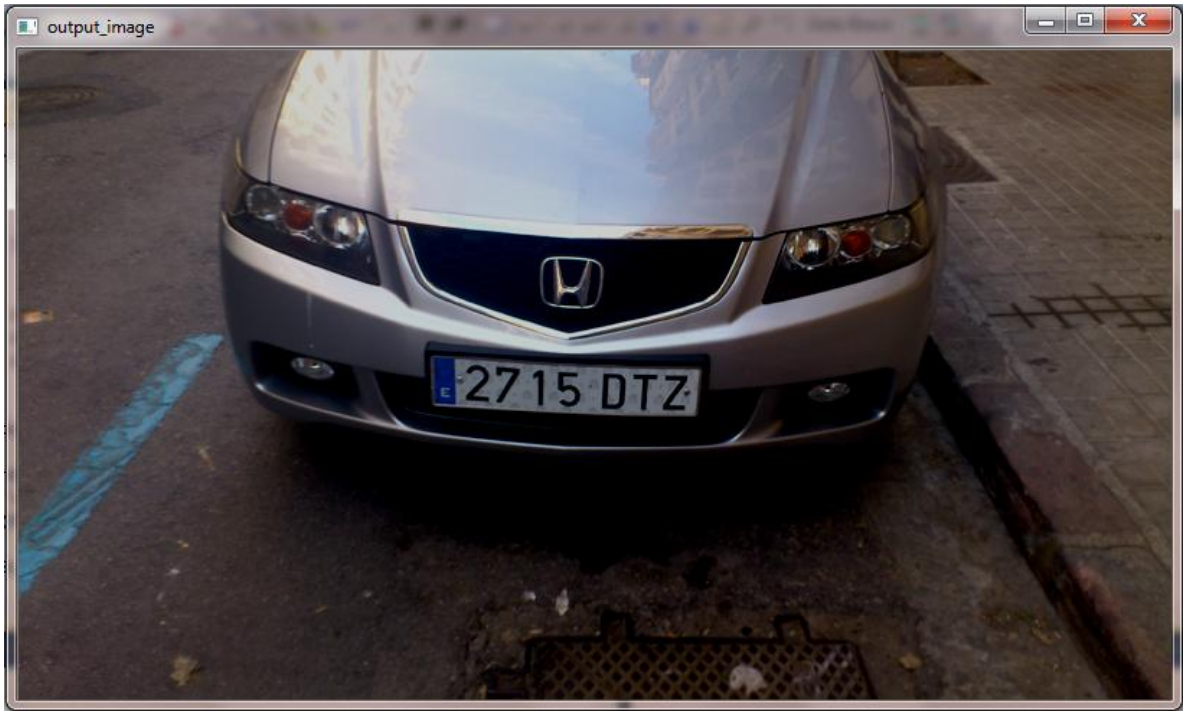
Where g(x) is the output image, f(x) is the input image, a is a scalar value that determines the contrast factor, and b determines the brightness of our final image. Therefore, f(x) will supply each pixel of the input image to our algorithm, which will then multiply the pixel by a and add b to it. An implementation of such an algorithm can be done in C++ as:

```
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <stdio.h>
#include <iostream>

using namespace std;
using namespace cv;


using namespace cv;

int main( int argc, char** argv )
{

    Mat img = imread("car_pic1.jpg");

    namedWindow("input_image", CV_WINDOW_AUTOSIZE);
    imshow("input_image", img);

    int rows = img.rows;
    int cols = img.cols;

    if (img.isContinuous())
    {
        cols = rows * cols; // Loop over all pixels as 1D array.
        rows = 1;
    }

    FILE *output_file;
    output_file = fopen("img_ppm.ppm", "w");
    fprintf(output_file, "P3\n");
    fprintf(output_file, "# desktop snapshot to ppm by Haleeq Usman2\n");
    fprintf(output_file, "%d ", img.size[1]);
    fprintf(output_file, "%d\n", img.size[0]);
    fprintf(output_file, "255\n");

    int a = 0, b = 0;
    unsigned int red, green, blue, tmp_red, tmp_green, tmp_blue;
```

```
    for (int i = 0; i < rows; i++)
    {
        Vec3b *ptr = img.ptr<Vec3b>(i);
        for (int j = 0; j < cols; j++)
        {
            Vec3b pixel = ptr[j];

            red = pixel.val[2];
            green = pixel.val[1];
            blue = pixel.val[0];

            // Let apply our first filter: af(x) + b,
            // where a = contrast and b = gain (aka brightness)
            // let's save our output image as a ppm.
            a = 1;
            b = - 50;
            tmp_red = a*red;
            tmp_green = a*green;
            tmp_blue = a*blue;

            red = tmp_red > 255? 255 : tmp_red < 0? 0 : tmp_red;
            green = tmp_green > 255? 255 : tmp_green < 0? 0 : tmp_green;
            blue = tmp_blue > 255? 255 : tmp_blue < 0? 0 : tmp_blue;

            fprintf(output_file, " %u %u %u", red, green, blue);

            if(j % 800 == 0 && j > 0)
                fprintf(output_file, "\n");

            // Let's apply the transformation to the image data we have in memory for
            // display
            ptr[j].val[2] = a*ptr[j].val[2] + b > 255? 255 : a*ptr[j].val[2] + b < 0? 0 :
              a*ptr[j].val[2] + b;
            ptr[j].val[1] = a*ptr[j].val[1] + b > 255? 255 : a*ptr[j].val[1] + b < 0? 0 :
              a*ptr[j].val[1] + b;
            ptr[j].val[0] = a*ptr[j].val[0] + b > 255? 255 : a*ptr[j].val[0] + b < 0? 0 :
              a*ptr[j].val[0] + b;
        }
    }
    fclose(output_file);

    // Let's display our output image
    namedWindow("output_image", CV_WINDOW_AUTOSIZE);
    imshow("output_image", img);
    waitKey(0);

    return 0;
}
```

This program would produce:





As another example, let's say we have two images and would like to apply a linear blend effect to them, then we can apply the following mathematical operation:

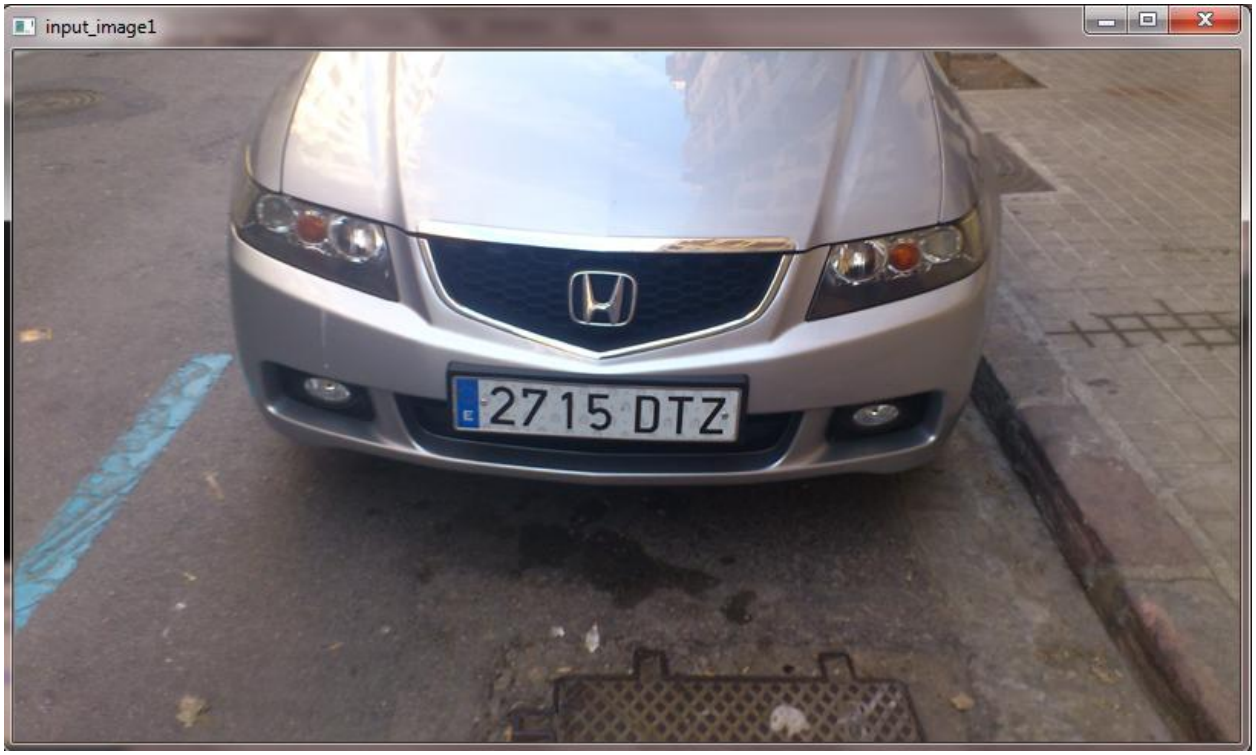
$$g(x) = (1-a)*f_0(x) + a*f_1(x)$$

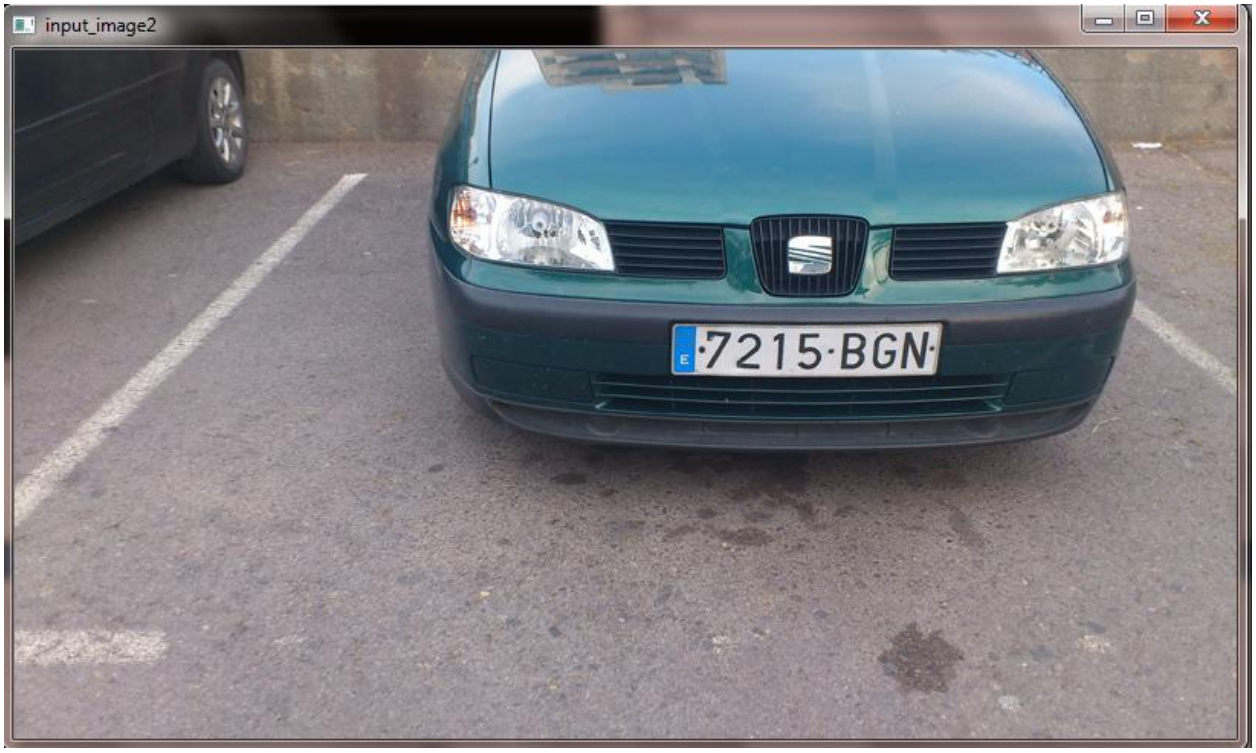
Where g(x) is the output image as before, $f_0(x)$ is the first input image, $f_1(x)$ is the second input image, and a is the blend factor. Therefore, if we wanted to blend two images together by 50%, then we can do something like:

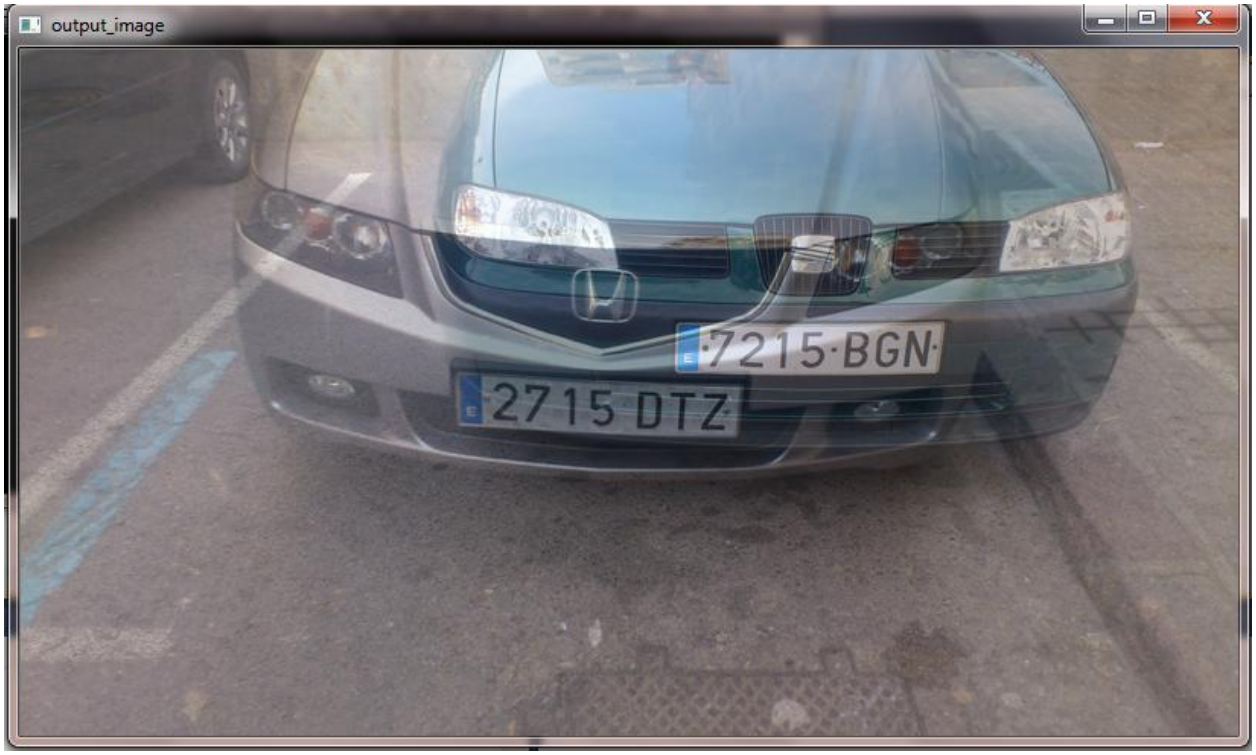
```
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <stdio.h>
#include <iostream>

using namespace std;
using namespace cv;


using namespace cv;

int main( int argc, char** argv )
{

    Mat img = imread("car_pic1.jpg");
    Mat img2 = imread("car_pic2.jpg");

    namedWindow("input_image1", CV_WINDOW_AUTOSIZE);
    namedWindow("input_image2", CV_WINDOW_AUTOSIZE);
    imshow("input_image1", img);
    imshow("input_image2", img2);

    int rows = img.rows;
    int cols = img.cols;

    if (img.isContinuous())
    {
        cols = rows * cols; // Loop over all pixels as 1D array.
        rows = 1;
    }

    FILE *output_file;
    output_file = fopen("img_ppm.ppm", "w");
    fprintf(output_file, "P3\n");
    fprintf(output_file, "# desktop snapshot to ppm by Haleeq Usman2\n");
    fprintf(output_file, "%d ", img.size[1]);
    fprintf(output_file, "%d\n", img.size[0]);
    fprintf(output_file, "255\n");

    float a = 0;
    unsigned int red, green, blue, tmp_red, tmp_green, tmp_blue;
    for (int i = 0; i < rows; i++)
    {
        Vec3b *ptr = img.ptr<Vec3b>(i);
        Vec3b *ptr2 = img2.ptr<Vec3b>(i);
        for (int j = 0; j < cols; j++)
        {
            Vec3b pixel = ptr[j];

            red = pixel.val[2];
            green = pixel.val[1];
            blue = pixel.val[0];
```

```
            // Let apply our linear blend filter: g(x) = (1-a)*f0(x) + a*f1(x),
            // where a = blend factor between 0 and 1,
            // if a = 0 then g(x) = f0(x), therefore blends to f0(x)
            // if a = 1 then g(x) = f1(x), therefore blends to f1(x)
            a = 0.5;

            ptr[j].val[2] = (1-a)*ptr[j].val[2] + a*ptr2[j].val[2];
            ptr[j].val[1] = (1-a)*ptr[j].val[1] + a*ptr2[j].val[1];
            ptr[j].val[0] = (1-a)*ptr[j].val[0] + a*ptr2[j].val[0];
        }
    }
    fclose(output_file);

    namedWindow("output_image", CV_WINDOW_AUTOSIZE);
    imshow("output_image", img);
    waitKey(0);

    return 0;
}
```

This would produce something like:







When we send the digital data back to the monitor, then the monitor produces analog signals corresponding to the intensity encoded into the digital data it received. This process is non-linear and leads to an exponential factor ranging in from 2.0 to 2.4. To compensate for this mismatch, we can apply a the mathematical function:

$$g(x) = f(x)^{\gamma}$$

To the digital data before sending it out to the monitor for display. The factor "y" in the equation above is denoted as the gamma correction factor. Most monitors will fall in the range of 2.0 to 2.4, I believe.

As mention earlier, Jpeg is another computational process performed by the computer to compress digital images even further. The decompression and compression processes can get quite involved. For example, here is a very simple jpeg decoder I wrote. It is not very complete because it does not support many of the jpeg features, which are mandatory by the jpeg standard. However, it should be able to decode grayscale images and images which contain a single color tone per 8x8 macroblock:

```
#ifndef _GLOBALS_H_
#define _GLOBALS_H_
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <vector>
#include <windows.h>

#define Y_dc_Hidx 0
#define Y_ac_Hidx 1

#define CbCr_dc_Hidx 2
#define CbCr_ac_Hidx 3

#define MCU_BLOCK_Y_DC 0
#define MCU_BLOCK_Y_AC 1
#define MCU_BLOCK_CB_DC 2
#define MCU_BLOCK_CB_AC 3
#define MCU_BLOCK_CR_DC 4
#define MCU_BLOCK_CR_AC 5

typedef enum MARKER {
	SOI =  0xFFD8, // Start of Image
	APP0 = 0xFFE0, // Application specific information
	DQT = 0xFFDB, // Define Quantization Table
	DHT = 0xFFC4, // Define Huffman Table
	SOS = 0xFFDA, // Start of Scan
	COM = 0xFFFE, // Comment
	SOF0 = 0xFFC0, // Start of Frame (Baseline DCT
	SOF2 = 0xFFC2, // Start of Frame (Progressive DCT)
	E0I = 0xFFD9, // End of Image
	EOB = 0x0000 // End of block
};
typedef struct treeNode {
	struct treeNode *left;
	struct treeNode *right;
	int val;
};
typedef struct huffTableProp {
```

```
	int num_nodes;
	int start_level;
	int node_shift;
};
typedef struct {
	unsigned short width;
	unsigned short height;
	unsigned char num_components;
	unsigned char comp_precision;
	unsigned char sampling_y;
	unsigned char sampling_cb;
	unsigned char sampling_cr;
	unsigned char dqt_y[8][8];
	unsigned char dqt_cbcr[8][8];
	struct treeNode dht_y_dc;
	struct treeNode dht_y_ac;
	struct treeNode dht_cbcr_dc;
	struct treeNode dht_cbcr_ac;
	int **mcu_y;
	int **mcu_cb;
	int **mcu_cr;
} jpegImage;

jpegImage jpeg_image;

treeNode *huff_table;
huffTableProp huffTreeInfo;
FILE *fp;
unsigned int spaces = 0;

#endif


#include "globals.h"

int read_short(unsigned short &short_val) {
	unsigned char short_buf[2];
	int read_amt = 0;
	short_val = 0;
	if((read_amt = fread(short_buf, sizeof(short_buf), 1, fp)) > 0) {
		short_val = short_val & 0 | short_buf[0] << 8 | short_buf[1];
	}
	return read_amt;
}

int read_byte(unsigned char &byte_val) {
	return fread(&byte_val, 1, 1, fp);;
}

int unread_byte(unsigned char last_char) {
	unsigned char buf;
	return ungetc(last_char, fp);
}

int read_bits(short num_bits = 0) {
	static unsigned char buf, last_buf;
	static int remainder = 0, store_last_buf = 0;
```

```
	int nbits_val = 0, dummy = 0;

	if(remainder == 0) {
		if(store_last_buf == 1) {
			last_buf = buf;
		} else {
			store_last_buf = 1;
		}

		if(read_byte(buf) == 0) {
			return -1;
		}

		printf("\ncurrent byte: %02x last byte: %02x\n", buf, last_buf);
		//printf("\nreading [%d] bits", num_bits);
		if(last_buf == 0xFF && buf == 0x00) {
			// skip this byte because it's a stuff byte
			if(read_byte(buf) == 0)
				return -1;
			printf("STUFF BYTE - SKIPPING 00, CURRENT = %02x\n", buf);
		}

		remainder = 8;
	}

	remainder--;
	if(num_bits == 0) {
		return buf >> remainder & 1;
	} else {
		int sign = 1;
		nbits_val = nbits_val | buf >> remainder & 1; // upper bit
		if(nbits_val == 0)
			sign = -1;
		num_bits--;
		for(dummy=0; dummy<num_bits; ++dummy) {
			nbits_val = nbits_val << 1 | read_bits();
		}
		return sign*nbits_val;
	}
}

void set_zero(int *arr, int size) {
	int i;
	for(i=0; i<size; i++)
		arr[i] = 0;
}

void setImageWidth(unsigned short width) {
	jpeg_image.width = width;
}

void setImageHeight(unsigned short height) {
	jpeg_image.height = height;
}

void setNumComponents(char num_components) {
	jpeg_image.num_components = num_components;
}
```

```
void setCompPrecision(char comp_precision) {
	jpeg_image.comp_precision = comp_precision;
}

void setYSampling(char sampling) {
	jpeg_image.sampling_y = sampling;
}

void setCbSampling(char sampling) {
	jpeg_image.sampling_cb = sampling;
}

void setCrSampling(char sampling) {
	jpeg_image.sampling_cr = sampling;
}

void print_spaces() {
	for(int i=0; i<spaces; i++) {
		printf(" ");
	}
}

void processAPP0() {
	char file_format[5];
	unsigned short len;
	unsigned char major_version;
	unsigned char minor_version;
	char *remaining_bytes;

	spaces +=5;

	read_short(len);
	len = len - 2;

	fread(file_format,1, 5, fp);
	read_byte(major_version);
	read_byte(minor_version);
	len = len - 7;

	remaining_bytes = (char *)malloc(len);
	if(fread(remaining_bytes, 1, len, fp) == len) {
		printf("Skipped %d bytes\n", len);
	}

	print_spaces();
	printf("File Format: %s %d.%d\n", file_format, major_version, minor_version);
	spaces -=5;
}

void processCOM() {
	unsigned short len;
	char *data;

	spaces +=5;
	read_short(len);
	len = len - 2;
```

```
	data = (char *)malloc(len);
	if(fread(data, 1, len, fp) == len) {
		data[len] = '\0';
		print_spaces();
		printf("%s\n", data);
		spaces -=5;
	}
}

void processDQT() {
	unsigned short len;
	unsigned char dqt_id;
	int i = 0, x = 0;
	unsigned char tmp_dqt_cbcr[8][8], tmp_dqt_y[8][8];
	int k = 0, j, i1, j1;
	int zigzag[] = {
			0,  1,  8, 16,  9,  2,  3, 10,
			17, 24, 32, 25, 18, 11,  4,  5,
			12, 19, 26, 33, 40, 48, 41, 34,
			27, 20, 13,  6,  7, 14, 21, 28,
			35, 42, 49, 56, 57, 50, 43, 36,
			29, 22, 15, 23, 30, 37, 44, 51,
			58, 59, 52, 45, 38, 31, 39, 46,
			53, 60, 61, 54, 47, 55, 62, 63
		};
	spaces = 5;
	print_spaces();

	read_short(len) - 2;
	read_byte(dqt_id);
	if(dqt_id == 0) {
		for(x=0; x<8; x++) {
			for(i=0; i<8; i++) {
				read_byte(tmp_dqt_y[x][i]);
			}
		}
		// Let's re-order the Luminance DQT
		for(int i = 0; i < 8; i++) {
			for(j = 0; j < 8; j++) {
				i1 = zigzag[k] / 8;
				j1 = zigzag[k] % 8;
				jpeg_image.dqt_y[i1][j1] = tmp_dqt_y[i][j];
				k++;
			}
		}

		printf("Luminance DQT:\n");
		for(int x=0; x<8; x++) {
			print_spaces();
			for(i=0; i<8; i++) {
				printf("%4d", jpeg_image.dqt_y[x][i]);
			}
			printf("\n");
		}
		read_byte(dqt_id);
		if(dqt_id != 0xFF) {
			for(x=0; x<8; x++) {
				for(i=0; i<8; i++) {
```

```
                        read_byte(tmp_dqt_cbcr[x][i]);
                    }
                }
                // Let's re-order the Chromance DQT
                k = 0;
                for(int i = 0; i < 8; i++) {
                    for(j = 0; j < 8; j++) {
                        i1 = zigzag[k] / 8;
                        j1 = zigzag[k] % 8;
                        jpeg_image.dqt_cbcr[i1][j1] = tmp_dqt_cbcr[i][j];
                        k++;
                    }
                }

                printf("\n");
                print_spaces();
                printf("Chromance DQT:\n");
                for(int x=0; x<8; x++) {
                    print_spaces();
                    for(i=0; i<8; i++) {
                        printf("%4d", jpeg_image.dqt_cbcr[x][i]);
                    }
                    printf("\n");
                }
            } else {
                unread_byte(dqt_id);
            }
        } else {
            for(x=0; x<8; x++) {
                for(i=0; i<8; i++) {
                    read_byte(tmp_dqt_cbcr[x][i]);
                }
            }
            // Let's re-order the Chromance DQT
                k = 0;
                for(int i = 0; i < 8; i++) {
                    for(j = 0; j < 8; j++) {
                        i1 = zigzag[k] / 8;
                        j1 = zigzag[k] % 8;
                        jpeg_image.dqt_cbcr[i1][j1] = tmp_dqt_cbcr[i][j];
                        k++;
                    }
                }

                printf("Chromance DQT:\n");
                for(int x=0; x<8; x++) {
                    print_spaces();
                    for(i=0; i<8; i++) {
                        printf("%4d", jpeg_image.dqt_cbcr[x][i]);
                    }
                    printf("\n");
                }
            read_byte(dqt_id);
            if(dqt_id != 0xFF) {
                for(x=0; x<8; x++) {
                    for(i=0; i<8; i++) {
                        read_byte(tmp_dqt_y[x][i]);
                    }
```

```
                        }
                        // Let's re-order the Luminance DQT
                        for(int i = 0; i < 8; i++) {
                                for(j = 0; j < 8; j++) {
                                        i1 = zigzag[k] / 8;
                                        j1 = zigzag[k] % 8;
                                        jpeg_image.dqt_y[i1][j1] = tmp_dqt_y[i][j];
                                        k++;
                                }
                        }

                        printf("\n");
                        printf("Luminance DQT:\n");
                        for(int x=0; x<8; x++) {
                                print_spaces();
                                for(i=0; i<8; i++) {
                                        printf("%4d", jpeg_image.dqt_y[x][i]);
                                }
                                printf("\n");
                        }
                } else {
                        unread_byte(dqt_id);
                }
        }
        spaces -=5;
}

void processSOF0() {
        unsigned short len;
        spaces +=5;

        read_short(len);
        read_byte(jpeg_image.comp_precision);
        read_short(jpeg_image.width);
        read_short(jpeg_image.height);

        print_spaces();
        printf("Precision per component: %d | Dimension: %dx%d\n",
jpeg_image.comp_precision, jpeg_image.width, jpeg_image.height);
        spaces -=5;
}

void addValueToNode(treeNode *node, int level, int count, int value) {
        int i = level - 1;
        if (i >= 0) {
                if ((huffTreeInfo.node_shift + count >> i & 1) == 0) {
                        if (node->left == NULL) {
                                node->left = (treeNode *)malloc(sizeof(treeNode));
                                node->left->left = NULL;
                                node->left->right = NULL;
                                node->left->val = -1;
                        }

                        addValueToNode(node->left, i, count, value);
                } else {
                        if (node->right == NULL) {
                                node->right = (treeNode *)malloc(sizeof(treeNode));
                                node->right->left = NULL;
```

```
                        node->right->right = NULL;
                        node->right->val = -1;
                }

                addValueToNode(node->right, i, count, value);
        }
    } else {
        node->val = value;
    }
}

void processDHT() {
    unsigned short len;
    unsigned char dht_id = 0;
    unsigned char huff_len_count[4][16];
    unsigned char **huff_vals = (unsigned char **)malloc(4*sizeof(unsigned char *));
    int total_len = 0, n = 0, i = 0, z = 0, dht_loop = 0;
    spaces +=5;

    read_short(len);
    read_byte(dht_id);

    for(dht_loop = 0; dht_loop < 4; dht_loop++) {
        total_len = 0;
        switch(dht_id) {
        case 0x00: // Class = 0 (DC), Id = 0 (Luminance)
            n = Y_dc_Hidx;
            huff_table = &jpeg_image.dht_y_dc;
            break;
        case 0x01: // Class = 1 (DC), Id = 0 (Chrominance)
            n = Y_ac_Hidx;
            huff_table = &jpeg_image.dht_cbcr_dc;
            break;
        case 0x10: // Class = 0 (AC), Id = 1 (Luminance)
            n = CbCr_dc_Hidx;
            huff_table = &jpeg_image.dht_y_ac;
            break;
        case 0x11: // Class = 1 (AC), Id = 1 (Chrominance)
            n = CbCr_ac_Hidx;
            huff_table = &jpeg_image.dht_cbcr_ac;
            break;
        }
        huff_table->left = NULL;
        huff_table->right = NULL;
        huff_table->val = -1;;
        huffTreeInfo.num_nodes = 0;
        huffTreeInfo.node_shift = 0;
        huffTreeInfo.start_level = -1;
        int set_start_level = 1;
        int tmp_level_node_num = 0;

        printf("\n\n n = %d | id = %02x\n\n", n, dht_id);
        for (i = 1; i <= 16; ++i) {
            read_byte(huff_len_count[n][i - 1]);
            total_len += huff_len_count[n][i - 1];

            if(set_start_level == 1) {
                if(huff_len_count[n][i - 1] > 0) {
```

```
                                        huffTreeInfo.start_level = i;
                                        set_start_level = 0;
                                }
                        }
                }
                huff_vals[n] = (unsigned char *)malloc(total_len*sizeof(unsigned char));
                for (int i = 1; i <= 16; ++i) {
                        print_spaces();
                        printf("Codes of length %02d %s (%03d total):", i, (i > 1 ? "bits" :
"bit "), huff_len_count[n][i - 1]);
                        if (huff_len_count[n][i - 1] > 0) {
                                if(huffTreeInfo.start_level < i) {
                                        huffTreeInfo.node_shift = huffTreeInfo.node_shift +
tmp_level_node_num << 1;
                                }
                                tmp_level_node_num = 0;
                                for (z = 0; z < huff_len_count[n][i - 1]; ++z) {
                                        read_byte(huff_vals[n][i - 1]);
                                        addValueToNode(huff_table, i, z, huff_vals[n][i - 1]);
                                        huffTreeInfo.num_nodes++;
                                        tmp_level_node_num++;
                                        printf(" %02x", huff_vals[n][i - 1]);
                                }
                        }
                        printf("\n");
                }
                // Let's read the id for only the first two loops.
                if( dht_loop < 3) {
                        read_byte(dht_id);
                }
                print_spaces();
                printf("Total number of codes: %03d\n\n", total_len);
        }
        spaces -=5;
}

short searchHuffTable(int dht_id, int bit) {
        static treeNode *last_node;
        treeNode *active_table;

        // set the huffman table to search through
        if(last_node == NULL) {
                switch(dht_id) {
                case Y_dc_Hidx:
                        printf("\nIn table = Y_dc_Hidx\n");
                        active_table = &jpeg_image.dht_y_dc;
                        break;
                case Y_ac_Hidx:
                        printf("\nIn table = dht_y_ac\n");
                        active_table = &jpeg_image.dht_y_ac;
                        break;
                case CbCr_dc_Hidx:
                        printf("\nIn table = dht_cbcr_dc\n");
                        active_table = &jpeg_image.dht_cbcr_dc;
                        break;
                case CbCr_ac_Hidx:
                        printf("\nIn table = dht_cbcr_ac\n");
                        active_table = &jpeg_image.dht_cbcr_ac;
```

```
                        break;
                }
        } else {
                active_table = last_node;
        }

        if(bit == 0) {
                printf("0");
                if(active_table->left->val == -1) {
                        printf(".");
                        last_node = active_table->left;
                } else {
                        printf(". = %02x\n", active_table->left->val);
                        last_node = NULL;
                }
                return active_table->left->val;
        }

        printf("1");
        if(active_table->right->val == -1) {
                printf(".");
                last_node = active_table->right;
        } else {
                printf(". = %02x\n", active_table->right->val);
                last_node = NULL;
        }
        return active_table->right->val;
}

void processSOS() {
        unsigned short hdr_len;
        char *hdr_data;
        int dht_id, bit;
        short val_len;

        read_short(hdr_len);
        hdr_len -= 2;
        hdr_data = (char *)malloc(hdr_len*sizeof(char));
        dht_id = Y_dc_Hidx;

        // Let's skip the header info (size = hdr_len)
        if(fread(hdr_data, 1, hdr_len, fp) == hdr_len) {
                printf("Skipped %d bytes\n", hdr_len);

                /* Let's allocate memory for the ac/dc/mcu counters and macroblocks */
                // dc_count/ac_count are used to keep track of how many dc or ac
                // componets we have added to the current macroblock.
                // mcu_count is used to keep track of the macroblock number we are in.
                // mcu_type is used to keep track of the macroblock type we are in.
                // dummy is used for as the iterator for the loops.
                // done_decoding is used to signify whether or not
                // we are done decoding the macroblocks
                int dc_count, ac_count, mcu_count, mcu_type, dummy, done_decoding;
                int mcu_size, ac_run, ac_size, ac_code;
                dc_count = 0;
                ac_count = 0;
                mcu_count = 0;
                ac_run = 0;
```

```
		ac_size = 0;
		ac_code = 0;
		dummy = 0;
		done_decoding = 0;
		mcu_type = MCU_BLOCK_Y_DC;
		unsigned char *z3;
		mcu_size = ceil((double)jpeg_image.width/8 * jpeg_image.height/8);

		jpeg_image.mcu_y = (int **)malloc(mcu_size*sizeof(int *));
		jpeg_image.mcu_cb = (int **)malloc(mcu_size*sizeof(int *));
		jpeg_image.mcu_cr = (int **)malloc(mcu_size*sizeof(int *));


		// initialize each macroblock
		for(dummy=0; dummy<mcu_size; dummy++) {
			jpeg_image.mcu_y[dummy] = (int *)malloc(64*sizeof(int));
			jpeg_image.mcu_cb[dummy] = (int *)malloc(64*sizeof(int));
			jpeg_image.mcu_cr[dummy] = (int *)malloc(64*sizeof(int));
			set_zero(jpeg_image.mcu_y[dummy], 64);
			set_zero(jpeg_image.mcu_cb[dummy], 64);
			set_zero(jpeg_image.mcu_cr[dummy], 64);
		}

		// Let's begin processing the compressed data
		while(mcu_count < mcu_size) {
			bit = read_bits();
			if(bit == -1) {
				printf("[ERROR]: Jpeg image data is corrupt. [REASON]: Data
prematurely terminated!\n");
				return;
			}
			val_len = searchHuffTable(dht_id, bit);
			// check for (EOB) End of block
			if(val_len == EOB) {
				// Move onto next MCU
				switch(mcu_type) {
					// (Y) Lumninance DC/Lossless
				case MCU_BLOCK_Y_DC:
					mcu_type = MCU_BLOCK_Y_AC;
					dht_id = Y_ac_Hidx;
					++dc_count;
					break;
					// (Y) Lumninance AC/Lossy
				case MCU_BLOCK_Y_AC:
					mcu_type = MCU_BLOCK_CB_DC;
					dht_id = CbCr_dc_Hidx;
					// reset the ac/dc counters for next mcu
					ac_count = 0;
					dc_count = 0;
					break;
					// (Cb) Chrominance blue DC/Lossless
				case MCU_BLOCK_CB_DC:
					mcu_type = MCU_BLOCK_CB_AC;
					dht_id = CbCr_ac_Hidx;
					++dc_count;
					break;
					// (Cb) Chrominance blue AC/Lossy
				case MCU_BLOCK_CB_AC:
```

```
                                mcu_type = MCU_BLOCK_CR_DC;
                                dht_id = CbCr_dc_Hidx;
                                // reset the ac/dc counters for next mcu
                                ac_count = 0;
                                dc_count = 0;
                                break;
                                // (Cr) Chrominance red DC/Lossless
                        case MCU_BLOCK_CR_DC:
                                mcu_type = MCU_BLOCK_CR_AC;
                                dht_id = CbCr_ac_Hidx;
                                ++dc_count;
                                break;
                                // (Cr) Chrominance red AC/Lossy
                        case MCU_BLOCK_CR_AC:
                                mcu_type = MCU_BLOCK_Y_DC;
                                dht_id = Y_dc_Hidx;
                                // reset the ac/dc counters for next mcu
                                ac_count = 0;
                                dc_count = 0;
                                // move onto the next set of macroblocks
                                mcu_count++;
                                break;
                        }
                } else if(val_len > 0) {
                        printf("\nReading next %02x bits as value...\n", val_len);
                        // determine which macroblock type we are in
                        switch(mcu_type) {
                                // (Y) Lumninance DC/Lossless
                        case MCU_BLOCK_Y_DC:
                                jpeg_image.mcu_y[mcu_count][ac_count + dc_count] =
read_bits(val_len);
                                printf("Setting mcu_y dc value: %02x\n",
jpeg_image.mcu_y[mcu_count][ac_count + dc_count]);
                                dc_count++;
                                // move onto the AC values
                                mcu_type = MCU_BLOCK_Y_AC;
                                dht_id = Y_ac_Hidx;
                                break;
                                // (Y) Lumninance AC/Lossy
                        case MCU_BLOCK_Y_AC:
                                //ac_code = read_bits(val_len);
                                ac_run = val_len >> 4;
                                ac_size = val_len & 0x0F;
                                printf("AC_CODE = %02x\n", val_len);
                                printf("AC_RUN = %02x\n", ac_run);
                                printf("AC_SIZE = %02x\n", ac_size);
                                printf("skipping next %d ac entries\n", ac_run);
                                printf("current[0] ac_count = %d\n", ac_count);
                                ac_count += ac_run;
                                printf("current[1] ac_count = %d\n", ac_count);
                                jpeg_image.mcu_y[mcu_count][ac_count + dc_count] =
read_bits(ac_size);
                                printf("Setting mcu_y ac value: %02x\n",
jpeg_image.mcu_y[mcu_count][ac_count + dc_count]);
                                ac_count++;
                                // if 63 AC values, then move onto next block
                                if(ac_count >= 63) {
                                        ac_count = 0;
```

```
                                        dc_count = 0;
                                        mcu_type = MCU_BLOCK_CB_DC;
                                        dht_id = CbCr_dc_Hidx;
                                }
                                break;
                                // (Cb) Chrominance blue DC/Lossless
                        case MCU_BLOCK_CB_DC:
                                jpeg_image.mcu_cb[mcu_count][ac_count + dc_count] =
read_bits(val_len);
                                printf("Setting mcu_cb dc value: %02x\n",
jpeg_image.mcu_cb[mcu_count][ac_count + dc_count]);
                                dc_count++;
                                // move onto the AC values
                                mcu_type = MCU_BLOCK_CB_AC;
                                dht_id = CbCr_ac_Hidx;
                                break;
                                // (Cb) Chrominance blue AC/Lossy
                        case MCU_BLOCK_CB_AC:
                                //ac_code = read_bits(val_len);
                                ac_run = val_len >> 4;
                                ac_size = val_len & 0x0F;
                                printf("AC_CODE = %02x\n", val_len);
                                printf("AC_RUN = %02x\n", ac_run);
                                printf("AC_SIZE = %02x\n", ac_size);
                                printf("skipping next %d ac entries\n", ac_run);
                                printf("current[0] ac_count = %d\n", ac_count);
                                ac_count += ac_run;
                                printf("current[1] ac_count = %d\n", ac_count);
                                jpeg_image.mcu_cb[mcu_count][ac_count + dc_count] =
read_bits(ac_size);
                                printf("Setting mcu_cb ac value: %02x\n",
jpeg_image.mcu_cb[mcu_count][ac_count + dc_count]);
                                ac_count++;
                                // if 63 AC values, then move onto next block
                                if(ac_count >= 63) {
                                        ac_count = 0;
                                        dc_count = 0;
                                        mcu_type = MCU_BLOCK_CR_DC;
                                        dht_id = CbCr_dc_Hidx;
                                }
                                break;
                                // (Cr) Chrominance red DC/Lossless
                        case MCU_BLOCK_CR_DC:
                                jpeg_image.mcu_cr[mcu_count][ac_count + dc_count] =
read_bits(val_len);
                                printf("Setting mcu_cr dc value: %02x\n",
jpeg_image.mcu_cr[mcu_count][ac_count + dc_count]);
                                dc_count++;
                                // move onto AC values
                                mcu_type = MCU_BLOCK_CR_AC;
                                dht_id = CbCr_ac_Hidx;
                                break;
                                // (Cr) Chrominance red AC/Lossy
                        case MCU_BLOCK_CR_AC:
                                //ac_code = read_bits(val_len);
                                ac_run = val_len >> 4;
                                ac_size = val_len & 0x0F;
                                printf("AC_CODE = %02x\n", val_len);
```

```
                                printf("AC_RUN = %02x\n", ac_run);
                                printf("AC_SIZE = %02x\n", ac_size);
                                printf("skipping next %d ac entries\n", ac_run);
                                printf("current[0] ac_count = %d\n", ac_count);
                                ac_count += ac_run;
                                printf("current[1] ac_count = %d\n", ac_count);
                                jpeg_image.mcu_cr[mcu_count][ac_count + dc_count] =
read_bits(ac_size);
                                printf("Setting mcu_cr ac value: %02x\n",
jpeg_image.mcu_cr[mcu_count][ac_count + dc_count]);
                                ac_count++;
                                // if 63 AC values, then move onto next block
                                if(ac_count >= 63) {
                                        ac_count = 0;
                                        dc_count = 0;
                                        mcu_type = MCU_BLOCK_Y_DC;
                                        dht_id = Y_dc_Hidx;
                                }
                                break;
                        }
                }
        }

        /*
        // Let's re-order the Luminance DQT
        int zigzag[] = {
                0,  1,  8, 16,  9,  2,  3, 10,
                17, 24, 32, 25, 18, 11,  4,  5,
                12, 19, 26, 33, 40, 48, 41, 34,
                27, 20, 13,  6,  7, 14, 21, 28,
                35, 42, 49, 56, 57, 50, 43, 36,
                29, 22, 15, 23, 30, 37, 44, 51,
                58, 59, 52, 45, 38, 31, 39, 46,
                53, 60, 61, 54, 47, 55, 62, 63
        };

        jpegImage jpg_img_tmp;
        int k = 0, j, i1, j1;
        for(int i = 0; i < 8; i++) {
                for(j = 0; j < 8; j++) {
                        i1 = zigzag[k] / 8;
                        j1 = zigzag[k] % 8;
                        jpg_img_tmp.dqt_y[i1][j1] = jpeg_image.mcu_y[i][j];
                        k++;
                }
        }
        */

        // We are done decoding the huffman values. Let's output the DCT
        // macroblocks
        int dummy2, dummy3, **mcu_block;
        char component_label[3];
        printf("\nBASELINE DCT MCU/Macroblocks:\n\n");
        for(dummy=0; dummy<mcu_size; ++dummy) {
                printf("DCT MCU/Macroblocks #%d:\n", dummy + 1);
                for(dummy2=0; dummy2<3; ++dummy2) {
                        switch(dummy2) {
                        case 0:
```

```
                                        mcu_block = jpeg_image.mcu_y;
                                        component_label[0] = 'Y';
                                        component_label[1] = ' ';
                                        component_label[2] = '\0';
                                        break;
                                case 1:
                                        mcu_block = jpeg_image.mcu_cb;
                                        component_label[0] = 'C';
                                        component_label[1] = 'r';
                                        component_label[2] = '\0';
                                        break;
                                case 2:
                                        mcu_block = jpeg_image.mcu_cr;
                                        component_label[0] = 'C';
                                        component_label[1] = 'b';
                                        component_label[2] = '\0';
                                        break;
                                }
                                printf("    %2");
                                for(dummy3=0; dummy3<64; ++dummy3) {
                                        printf("%4d ", mcu_block[dummy][dummy3]);
                                        if((dummy3 + 1) % 8 == 0) {
                                                printf("\n");
                                                if((dummy3 + 1) != 32) {
                                                        printf("    %2");
                                                }
                                        }
                                        if((dummy3 + 1) == 32) {
                                                printf("%s =%2", component_label);
                                        }
                                }
                                printf("\n");
                        }
                        printf("\n\n");
                }

        }
}

int main() {
        MARKER marker;
        unsigned char byte_val = 0;
        unsigned short short_val = 0;
        int continue_reading = 1;

        int read_amt = 0;

        //fp = fopen("huff_simple023.jpg","rb");
        fp = fopen("5.jpg","rb");
        if(fp) {
                printf("file opened!\n");

                while(read_short(short_val) > 0 && continue_reading) {
                        switch(short_val) {
                        case SOI:
                                printf("\nread (%02x): SOI\n", short_val);
                                break;
                        case APP0:
```

```
                        printf("\nread (%02x): APP0\n", short_val);
                        processAPP0();
                        break;
                case DQT:
                        printf("\nread (%02x): DQT\n", short_val);
                        processDQT();
                        break;
                case SOF0:
                        printf("\nread (%02x): SOF0\n", short_val);
                        processSOF0();
                        break;
                case DHT:
                        printf("\nread (%02x): DHT\n", short_val);
                        processDHT();
                        break;
                case SOS:
                        printf("\nread (%02x): SOS\n", short_val);
                        processSOS();
                        //continue_reading = 0;
                        break;
                case COM:
                        printf("\nread (%02x): COM\n", short_val);
                        processCOM();
                        break;
                default:
                        printf("\nread (%02x): Unknown or skipping...\n", short_val);
                }
        }
    }
    printf("\n");
    return 0;
}
```