EECS 22: Assignment 4

Prepared by: Alex Chu, Prof. Rainer Doemer

October 31, 2013

Due on Monday 11/18/2013 11:00pm. Note: this is a two-week assignment.

1 Digital Image Processing [100 points + 10 bonus points]

In this assignment, you will learn how to use dynamic memory allocation in your program and how to link against libraries. Based on the program *PhotoLab* for Assignment 3, you will redesign your digital image processing (DIP) operations to accommodate varying image sizes. Then, you will add more DIP operations whose resulting images will differ in size compared to the original one. Thus you can use your *PhotoLab* program to perform the DIP operations on any of your own pictures.

1.1 Introduction

In Assignment 3, you were asked to decompose your *PhotoLab* program into separate modules and compile them into different programs. There were 2 main programs. In the first, the user could load an image from a file, apply a set of DIP operations to the image, and save the processed image in a file by running the *PhotoLab* application by hand. The second was the same as the first except that the DEBUG option was turned on for the compiler. This version of *PhotoLab* displayed more messages to the user and performed all the DIP operations automatically. This assignment will be an extension of Assignment 3.

1.2 Initial Setup

Before you start working on this assignment, do the following:

```
cd  ~/eecs22
mkdir hw4
cd hw4
ln -s  ~eecs22/hw3/UCI_Peter.ppm UCI_Peter.ppm
ln -s  ~eecs22/hw3/halloweenBat.ppm halloweenBat.ppm
ln -s  ~eecs22/hw4/turkey.ppm .
cp ~eecs22/hw4/FileIO.h .
cp ~eecs22/hw4/FileIO.c .
cp ~eecs22/hw4/Image.h .
cp ~eecs22/hw4/index.html ~/public_html/index.html
```

We will extend the *PhotoLab* program based on Assignment 3. Please reuse your **PhotoLab.c** file or the provided **PhotoLab.c** solution as the starting point for this assignment. You may need to copy **PhotoLab.c** from the *hw3* folder or course website to *hw4* first. Copy all the source code files (*.c) and the header files (*.h) to *hw4* except **FileIO.h** and **FileIO.c** Copy the new header and source file for File I/O from the *eecs22* account.
Here,

- **Image.h** is the header file for the definition of the new structure and declarations of the pixel mapping functions we will use in Section 1.3.2;

- **FileIO.h** is the new header file for File I/Os (ReadImage() and SaveImage()).
- **FileIO.c** is the new source file for File I/Os (ReadImage() and SaveImage()).

NOTE:
Again, we will use the PPM image file *UCI_Peter.ppm* for this assignment. Once a DIP operation is done, you can save the modified image as *name.ppm*, and it will be automatically converted to a JPEG image and sent to the folder *public_html* in your home directory. You are then able to see the image in any web browser at: *http://newport.eecs.uci.edu/∼youruserid*, if required names are used. If you save images by other names, use the link *http://newport.eecs.uci.edu/∼youruserid/imagename.jpg* to access the photo.

Note that whatever you put in the *public_html* directory will be publicly accessible; make sure you don't put files there that you don't want to share, i.e. do not put your source code into that directory.

1.3 Add support for different image sizes

In this assignment, we will add support for DIP operations on images with different sizes. In the previous two assignments, our programs defined two constants **WIDTH** and **HEIGHT** as the fixed size of the input image. At that time, our *PhotoLab* program could only manipulate images with the fixed size of 600x475.

In order to add support for varying image sizes, we need to redefine the size of the arrays that we use to store the color intensity information for each pixels. Since the size of the input image cannot be determined at compile time, we cannot use arrays to hold the pixel information. Therefore, we need to use dynamic memory allocation to claim three blocks of memory whose size will be decided at program run time.

Instead of defining three arrays and passing them as the arguments to the DIP operation functions, we will now use pointers to point to an image structure which be dynamically allocated in memory by our program at run time.

1.3.1 Use pointers in one dimensional memory space instead of arrays with two dimensions

We need to use dynamic memory allocation since the size of the image will not be known until we run the program. We will use three pointers to type *unsigned char* for the color intensity values for each pixel instead of three fixed sized arrays. However, pointers only point to a memory space in one dimension. Therefore, we need to map 2-tuple coordinates of pixels to a single value index corresponding to the pixel color information in memory.

For example, we have an image of size 10x5, and three pixels (0, 0), (9, 4), and (6, 4). We assume row major for the image storage in this program. Therefore, the index value for pixel (0, 0) in the one dimensional storage space will be 0; the index value for pixel (9, 4) in the one dimensional storage space will be $49 = 9 + 4 * 10$; and the index value for pixel (6, 4) in the one dimensional storage space will be $46 = 6 + 4 * 10$.

In general, the index value for the pixel (*x*, *y*) in an image of size **WIDTH**x**HEIGHT** in the one dimensional storage space will be ***x* + *y* * WIDTH**.

1.3.2 The Image.c module

Please add one module **Image** (see provided **Image.h**) to handle basic operations on the image.

- **The IMAGE struct**: We will use a *struct type* to aggregate all the information of an image. The following struct is defined in **Image.h**:

```
typedef struct {
unsigned int Width; /* image width */
unsigned int Height;  /* image height */
unsigned char *R; /* pointer to the memory storing all the R intensity values */
unsigned char *G; /* pointer to the memory storing all the G intensity values */
unsigned char *B; /* pointer to the memory storing all the B intensity values */
```

```
}IMAGE;
```

- Define the functions to get and set the value of the color intensities of each pixel in the image. Please use the following function prototypes (provided in **Image.h**) and define the functions properly (in **Image.c**).

```
/* Get the color intensity of the Red channel of pixel (x, y) in image */
unsigned char GetPixelR(IMAGE *image, unsigned int x,  unsigned int y);

/* Get the color intensity of the Green channel of pixel (x, y) in image */
unsigned char GetPixelG(IMAGE *image, unsigned int x,  unsigned int y);

/* Get the color intensity of the Blue channel of pixel (x, y) in image */
unsigned char GetPixelB(IMAGE *image, unsigned int x,  unsigned int y);

/* Set the color intensity of the Red channel of pixel (x, y) in image with value r */
void SetPixelR(IMAGE *image, unsigned int x,  unsigned int y, unsigned char r);

/* Set the color intensity of the Green channel of pixel (x, y) in image with value g
void SetPixelG(IMAGE *image, unsigned int x,  unsigned int y, unsigned char g);

/* Set the color intensity of the Blue channel of pixel (x, y) in image with value b *
void SetPixelB(IMAGE *image, unsigned int x,  unsigned int y, unsigned char b);
```

  The mapping from the 2-tuple coordinates (x, y) to the single index value for the one dimensional memory space will be taken care of in these functions. Please call these functions in your DIP functions for setting / getting the intensity values of the pixels.

- Please add **assertions** in these functions to make sure the input image pointer is valid, and the set of pointers to the memory spaces for the color intensity values are valid too. Last but not least, add assertions to ensure that the coordinates are within the valid ranges for the image.

- Please extend/adjust your **Makefile** accordingly: 1) add the target to generate **Image.o** and 2) add **Image.o** when generating **PhotoLab** and **PhotoLabTest**.

1.3.3 Read and save image files

You may refer to **FileIO.h** for the defined functions for file I/Os.

- **IMAGE *ReadImage(const char *fname):** reads the file with the name *fname.ppm* and returns the image pointer. The color intensities for channel red, green, and blue are stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of the returned IMAGE pointer respectively. The memory space of the image is created in this function by a function call to CreateImage(), see below.

  NOTE: This function returns NULL if loading the image failed. Be sure to handle this error.

- **int SaveImage(const char *fname, IMAGE *image):** saves the color intensities of each red, green, and blue channel stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of *image* into the file with the name *fname.ppm*. This function returns an error code if something went wrong. Handle it by letting the user know that the image was not saved.

Please write two functions to handle the memory allocations and deallocations in **Image.c** (Functions declared in **Image.h**).

Please use the following function prototypes.

```
/* allocate the memory space for the image structure        */
/* and the memory spaces for the color intensity values.    */
```

```
/* return the pointer to the image, or NULL in case of error */
IMAGE *CreateImage(unsigned int Width, unsigned int Height);

/* release the memory spaces for the pixel color intensity values */
/* deallocate all the memory spaces for the image                 */
void DeleteImage(IMAGE *image);
```

IMPORTANT: Note that the ReadImage() function in the FileIO library needs the CreateImage() function to allocate the memory space! Therefore, you will need to define the CreateImage() function correctly before you can use the ReadImage() function.

1.3.4 Modify function prototypes and definitions

Most of our functions need to be refined by taking the IMAGE structure as a parameter which contains all the information about the image.

Your DIP function prototypes should look like below:

- In **DIPs.h**:

```
/* change color image to black & white */
IMAGE *BlackNWhite(IMAGE *image);

/* flip image vertically */
IMAGE *VFlip(IMAGE *image);

/* mirror image horizontally */
IMAGE *HMirror(IMAGE *image);

/* color filter */
IMAGE *ColorFilter(IMAGE *image,
 int target_r, int target_g, int target_b, int threshold,
 double factor_r, double factor_g, double factor_b) ;

/* sharpen the image */
IMAGE *Sharpen(IMAGE *image);

/* edge detection */
IMAGE *Edge(IMAGE *image);

/* add border */
IMAGE *AddBorder(IMAGE *image,
       int border_r, int border_g, int border_b,
       int border_width);
```

- In **Advanced.h**:

```
/* Posterization */
IMAGE *Posterize(IMAGE *image,
   unsigned char rbits,
   unsigned char gbits,
   unsigned char bbits);

/* add noise to image */
IMAGE *AddNoise( int percentage,
  IMAGE *image);
```

```
/* overlay with another image */
IMAGE *Overlay(char fname[SLEN],
     IMAGE *image,
     unsigned int x_offset, unsigned int y_offset);

/*Resize*/
IMAGE *Resize( unsigned int percentage, IMAGE *image);

/*Rotate*/
IMAGE *Rotate(IMAGE *image);

/* Juliaset */
IMAGE *Juliaset(unsigned int W, unsigned int H, unsigned int max_iteration);

/* BONUS: Crop */
IMAGE *Crop(IMAGE *image, unsigned int x, unsigned int y, unsigned int W, unsigned int

/* Test all functions */
void             AutoTest(IMAGE *image);
```

IMPORTANT: Notice the changes made to the return types and the function arguments!
NOTE: By using pointers in one dimensional memory space, you need to modify the statements in your functions for array elements' indexing with the pixel setting / getting functions accordingly. For example:

- In Assignment 3, we got the pixel's color value by indexing the element from the two-dimensional array: *tmpR = R[x][y]*;
- Now, we need to get the pixel's color value by calling the getting function: *tmpR = GetPixelR(image, x, y)*;
- In Assignment 3, we set the pixel's color value by indexing the element from the two-dimensional array: *R[x][y] = ...*;
- Now, we need to set the pixel's color value by calling the setting function: *SetPixelR(image, x, y, ...)*;

By using the setting / getting functions, we can keep the two-dimensional coordinate system as in Assignment 2 and Assignment 3.

Please make sure to include the header file **Image.h** properly in your source code files and header files.

1.3.5 Modify the AutoTest function

Please put the File I/O and memory allocations inside the *AutoTest(IMAGE *image)* function. The only parameter this function will take is an IMAGE struct. Please refer to Section 1.5 for more implementation details.

1.3.6 Modify the Makefile to link against the library

Your own **Makefile** should have at least the following targets:

- *all*: the target to generate the executable program.
- *clean*: the target to clean up all the intermediate files, e.g. object files, the executable programs, and the generated ppm files.
- **.o*: the target to generate the object file *.o from the C source code file *.c.

(a) A second image (150x115)

(b) Original overlay of halloweenBat image at position (100, 150)

(c) New overlay of turkey image at position (165, 325)

Figure 1: The new turkey image, the original overlay image, and the new overlay image.

- *PhotoLab*: the target to generate the executable program *PhotoLab*.
- *PhotoLabTest*: the target to generate the executable program *PhotoLabTest*.

It is recommended to have a few more targets as they will help decompose the structure of your makefile. Compile your source code into *PhotoLab* and *PhotoLabTest* by using your **Makefile**:

```
make all
```

HINT: There are two ways to link against a library (.a) file:

1. use it as a normal object file with full name.
2. use the *-l* option of *gcc* to specify which library to use (e.g. *-lfileio* means using the library *libfileio.a*), and the *-L* option of *gcc* to specify the directory of the library.

The second option is the recommended one.

1.4 Advanced DIP operations

In this assignment, please implement the advanced DIP operations described below in **Advanced.c** (**Advanced.h** as the header file).

Please reuse the menu you designed for Assignment 3 and extend it with the advanced operations. The user should be able to select DIP operations from a menu as the one shown below:

```
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to Black & White
 4:  Flip an image vertically
 5:  Mirror an image horizontally
 6:  Color filter an image
 7:  Sketch the edge of an image
 8:  Sharpen an image
 9:  BONUS: Add Border to an image
 10: Add noise
 11: Overlay
```

(a) Original image

(b) resized to a bigger image (percentage = 175)

(c) resized to a smaller image (percentage = 60)

Figure 2: An image and its resized bigger and resized smaller counterparts.

```
 12: Posterize
 13: Bonus, Surprise Filter
 14: Resize the image
 15: Rotate 90 degrees clockwise
 16: Generate the Julia set image
 17: BONUS: Crop
 18: Exit
Please make your choice:
```

1.4.1 Image Overlay

This function has been modified to take varying image sizes as input. In addition to the overlay picture in the previous assignment, we will now add a second overlay image which is of smaller size. We will overlay the original halloween-Bat in it's original position (100, 150). We will overlay the smaller turkey image at position (165, 325). Please turn in 2 overlay images for this section.

Save the orignal overlay image with the name 'overlaybat' after this step.
Save the new overlay image with the name 'overlayturkey' after this step.

1.4.2 Resize

You need to define and implement the following function for this DIP.

```
/*Resize*/
IMAGE *Resize(unsigned int percentage, IMAGE *image);
```

This function resizes the image with the scale of *percentage*.

- *percentage* $==$ 100, the size of the new image is the same as the original one.
- *percentage* $<$ 100, the size of the new image is smaller than the original one.
- *percentage* $>$ 100, the size of the new image is larger than the original one.

More specifically, we scale *percentage* as follows:

- $Width_{new}$ = $Width_{old}$ * (percentage / 100.00);
- $Height_{new}$ = $Height_{old}$ * (percentage / 100.00);

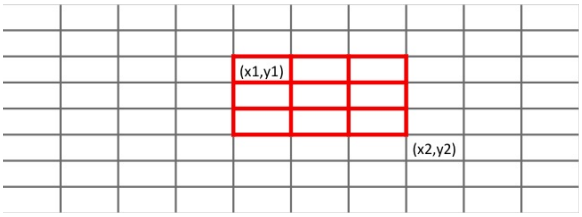


Figure 3: Pixels mapping from the bigger original image to the smaller new image

If *percentage* is greater than 100, we need to duplicate some pixels from the original image to the new larger one. Assume (x', y') are the coordinates for the position of the pixel in the new image while (x, y) are the coordinates for the position of the pixel in the original image. Then, copy the color of the pixel(x, y) in the original image to pixel (x', y') in the new image Note that:

```
x' = x * (percentage / 100.00);
y' = y * (percentage / 100.00);
```

If *percentage* is less than 100, we will have fewer pixels in the new smaller image than in the original image. Therefore, we need to average the values of the color intensities of multiple pixels from the original image. Otherwise, we lose too much information from the original image. We use this average value as the color intensity of the pixel in the smaller image.

To demonstrate, each grid element represents one pixel in the image, as shown in Figure 3. We average the value of the color intensities for all the red edged pixels in the original image (from (x_1, y_1) to ($x_2 - 1$, $y_2 - 1$)) and use this average as the red color intensity of the pixel (x, y) in the new image, where:

```
x1 = x / (percentage / 100.00);
y1 = y / (percentage / 100.00);
x2 = (x + 1) / (percentage / 100.00);
y2 = (y + 1) / (percentage / 100.00);
```

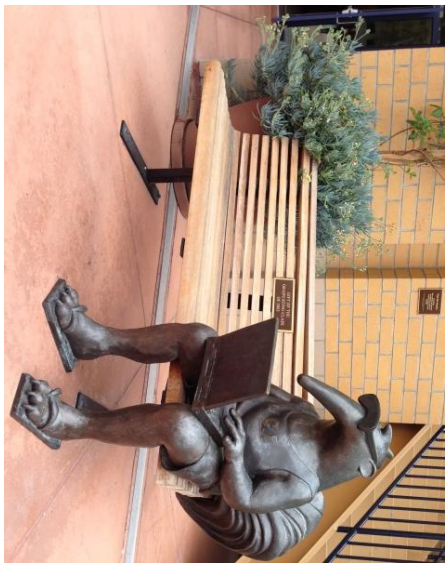
If *percentage* is greater than 100, there are more pixels in the target image than in the original. Thus, some pixel colors will be duplicated in the result.

HINT: For enlarging the image, it is easier to iterate over the target image (not over the original image).
NOTE: The *Resize()* function will consume the input image and return a new image with the new size. Please delete and create the image data structures properly in this function.

Figure 2 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 14
Please input the resizing percentage (integer between 1~500): 175
"Resizing the image" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to Black & White
 4:  Flip an image vertically
 5:  Mirror an image horizontally
 6:  Color filter an image
 7:  Sketch the edge of an image
 8:  Sharpen an image
 9:  BONUS: Add Border to an image
```

(a) Original image



(b) Rotated image

Figure 4: An image and its rotated counterpart.

```
 10: Add noise
 11: Overlay
 12: Posterize
 13: Bonus, Surprise Filter
 14: Resize the image
 15: Rotate 90 degrees clockwise
 16: Generate the Julia set image
 17: BONUS: Crop
 18: Exit
Please make your choice:
```

Save the two images for this operation:

1. 'bigresize': a bigger image with scale *percentage* = 175.
2. 'smallresize': a smaller image with scale *percentage* = 60.

1.4.3 Rotate-90-degree

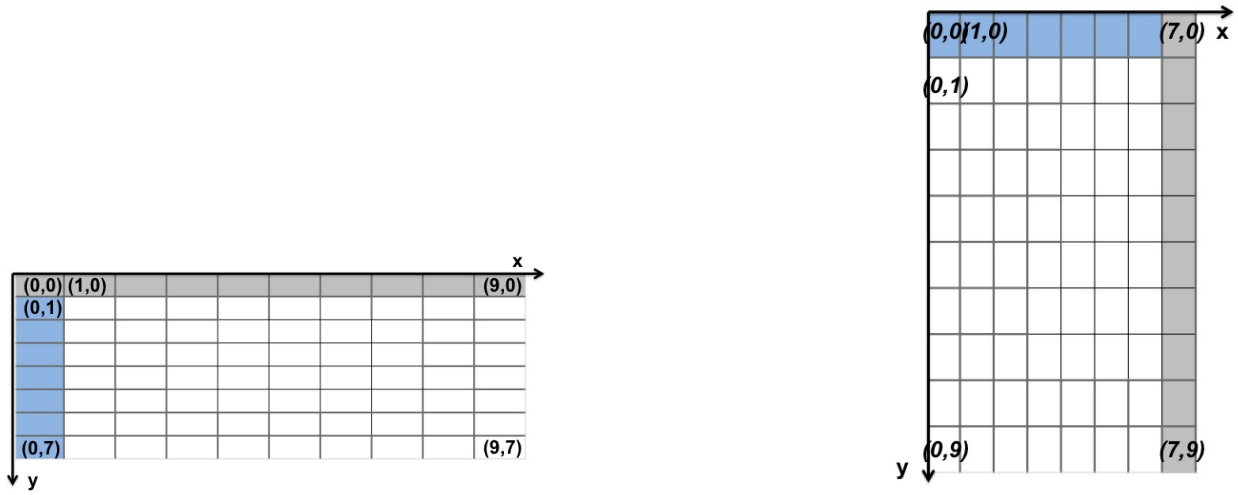
This function rotates the image by 90 degrees clockwise. The size of the image will be the same, but the width (height) of the new image will be the same as its original height (width).

NOTE: As shown in Fig. 5, row pixel indices increase as they go down while the column pixel indices increase as they go to the right. Pixel indices are integer values ranging from 1 to the length of the row or column. The top left pixel's coordinate is (0, 0), and the bottom right pixel's coordinate is (image→Width - 1, image→Height - 1).

First, you need to find how the pixel coordinates map from the original image (x, y) to the rotated image (x', y'). Then, you need to set the color of the pixel at (x', y') in the new image to the color of the pixel at (x, y) in the original image.

You need to define and implement the following function to do this DIP.

```
/*Rotate 90 degrees clockwise*/
```



(a) Coordinates for the original image

(b) Coordinates for the rotated image

Figure 5: An image and its rotated counterpart.

```
IMAGE *Rotate(IMAGE *image);
```

Figure 4 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 15
"Rotate 90 degree clockwise" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to Black & White
 4:  Flip an image vertically
 5:  Mirror an image horizontally
 6:  Color filter an image
 7:  Sketch the edge of an image
 8:  Sharpen an image
 9:  BONUS: Add Border to an image
 10: Add noise
 11: Overlay
 12: Posterize
 13: Bonus, Surprise Filter
 14: Resize the image
 15: Rotate 90 degrees clockwise
 16: Generate the Julia set image
 17: BONUS: Crop
 18: Exit
Please make your choice:
```

Save the image with the name 'rotate' after this step.

1.4.4 Image for Julia Set

The **Julia set** is a mathematical set of points whose boundary is a distinctive and recognizable two-dimensional fractal shape. Images of the Julia set display an elaborate boundary that reveals finer detail at increasing magnifications through progressive and recursive computations. The Julia set has become popular outside mathematics for both its aesthetic appeal and as an example of the creation of a complex structure through the application of simple rules. It is one of the best-known examples of mathematical visualization. (`http://en.wikipedia.org/wiki/Julia_set`)
We are going to write a function to generate an image of the Julia set in this assignment.

- **The algorithm for drawing a picture of the Julia set**
  To draw an image of the Julia set, we will translate the Julia set drawing algorithm from pseudocode to C as an additional DIP function.

  A lot of real-world programming work requires the programmer to reuse and adjust source code written by others to fit their current project. The reference source can vary in many different aspects: the code can have different function prototypes or variable types, be written in pseudocode only, or be written in a completely different programming language. While the reference code provides the basic control flow and structure of the target application, ultimately, the programmer will need to translate the reference code to their working language and ensure the correctness of the implementation.

  We will use the following pseudocode adapted from (`http://lodev.org/cgtutor/juliamandelbrot.html`):

```
for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        /*calculate the initial real and imaginary part of z, based
        on the pixel location and zoom and position values */
        newRe = 1.5 * (x - w / 2) / (0.5 * zoom * w) + moveX;
        newIm = (y - h / 2) / (0.5 * zoom * h) + moveY;

        /* i will represent the number of iterations */
        int i;

        /* start the iteration process */
        for(i = 0; i < maxIterations; i++)
        {
            /* remember value of previous iteration */
            oldRe = newRe;
            oldIm = newIm;

            /* the actual iteration, the real and imaginary
            part are calculated */
            newRe = oldRe * oldRe - oldIm * oldIm + cRe;
            newIm = 2 * oldRe * oldIm + cIm;

            /* if the point is outside the circle with radius 2:
            stop */
            if((newRe * newRe + newIm * newIm) > 4) break;
        }
        /* use color model conversion to get rainbow palette,
        make brightness black if maxIterations reached */
```

```
        color = HSVtoRGB(ColorHSV(i % 256, 255, 255 * (i < maxIterations)));

        /* draw the pixel */
        pset(x, y, color);
    }
```

Here is a brief explanation for this algorithm. The algorithm computes the color for each pixel in the picture based on their coordinates. The algorithm interprets the pixel coordinates as complex numbers, where width represents the real part and the height represents the imaginary part. newRe represents the pixel width coordinate while newIm represents the pixel height coordinate. moveX and moveY are offsets to be added to the pixels while zoom is the magnification of the image. maxIterations is the upper boundary on how many times the loop will execute
First, the coordinates of the pixel will be scaled. We will make the following assumptions for this implementation: zoom = 1, moveX = 0, moveY = 0. Then, the x coordinate will be scaled into the range of (-1.5, 1.5) while the y coordinate will be scaled into the range of (-1, 1). After several other initializations (which will be mentioned later), the algorithm starts a for loop to do some computation. The number of iterations of the for loop is decided by the value of x*x+y*y and a predefined maximum indicated by *max_iteration*. The color of the pixel is decided by the number of the iterations actually executed. The color = HSVtoRGB() and pset() function will draw the pixel(x, y) with the color that is determined by the loop iterations.

To implement this algorithm in C, we need to define variables with proper types and design the scaling equations.

Moreover, we need to specify the actual color that is represented by a specific iteration number. We suggest to use 16 colors in the Juliaset image. Thus, we first change the "color = HSVtoRGB(...)' line into *color = iteration % 16*, and then use the variable *color* as the index to get the color intensity values from an array *palette* for different colors.

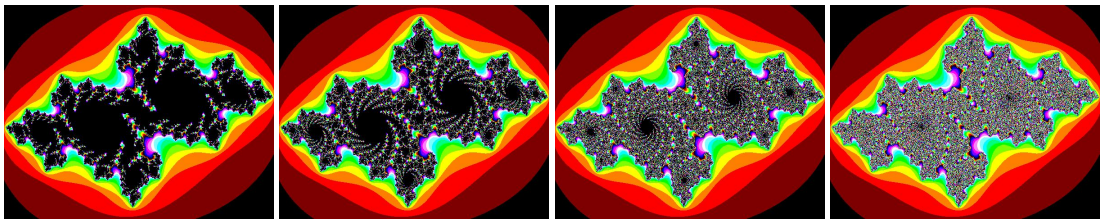
The following definition can be used in the C program:

1. In Constants.h, add the following line:

   ```
   #define MAX_COLOR 16
   ```

2. Inside the Juliaset function, use the following array definition for the color palette:

```
const unsigned char palette[MAX_COLOR][3] = {
        /* r   g   b*/
        {   0,   0,   0 },      /* 0, black            */
        { 127,   0,   0 },      /* 1, brown            */
        { 255,   0,   0 },      /* 2, red              */
        { 255, 127,   0 },      /* 3, orange           */
        { 255, 255,   0 },      /* 4, yellow           */
        { 127, 255,   0 },      /* 5, light green      */
        {   0, 255,   0 },      /* 6, green            */
        {   0, 255, 127 },      /* 7, blue green       */
        {   0, 255, 255 },      /* 8, turquoise        */
        { 127, 255, 255 },      /* 9, light blue       */
        { 255, 255, 255 },      /* 10, white           */
        { 255, 127, 255 },      /* 11, pink            */
        { 255,   0, 255 },      /* 12, light pink      */
        { 127,   0, 255 },      /* 13, purple          */
        {   0,   0, 255 },      /* 14, blue            */
        {   0,   0, 127 }       /* 15, dark blue       */
};
```



(a) max_iteration = 64 (b) max_iteration = 128 (c) max_iteration = 256 (d) max_iteration = 512

Figure 6: Images for the Julia set with different number of iterations.

- **Function Prototype**
  You need to define and implement the following function to do this DIP.

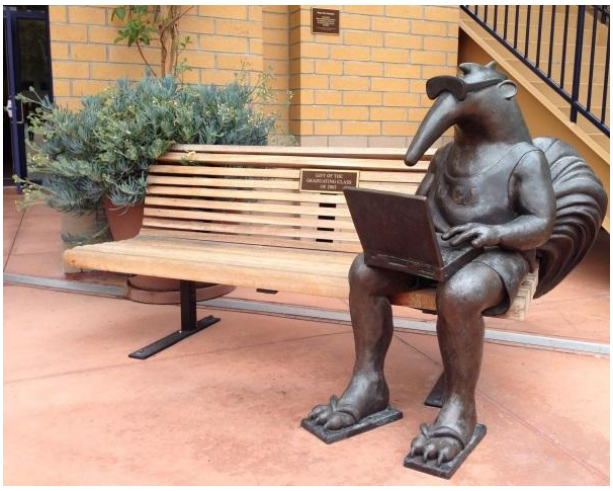
  ```
  /* Juliaset */
  IMAGE *Juliaset(unsigned int W, unsigned int H, unsigned int max_iteration);
  ```

  Here, **W** is the width of the generated image, **H** is the height of the generated image, and **max_iteration** is the maximum iterations for the computation of each pixel. This function will create an image, fill in the pixels with different colors according to the Julia Set rendering algorithm, and return the compiled image at the end.

NOTE: This function does not need a previously read image (you do not need to read UCI_Peter.ppm first or pass IMAGE* data structure to the function). Therefore, you will need to allocate an image within this function to calculate the Julia set.
Once the user chooses this option, your program's output should look like the following:

```
Please make your choice: 16
Please input the width of the Julia set image: 600
Please input the height of the Julia set image: 475
Please input the max iteration for the Julia set calculation: 256
"Julia Set" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to Black & White
 4:  Flip an image vertically
 5:  Mirror an image horizontally
 6:  Color filter an image
 7:  Sketch the edge of an image
 8:  Sharpen an image
 9:  BONUS: Add Border to an image
 10: Add noise
 11: Overlay
 12: Posterize
 13: Bonus, Surprise Filter
 14: Resize the image
 15: Rotate 90 degrees clockwise
 16: Generate the Julia set image
 17: BONUS: Crop
 18: Exit
Please make your choice:
```

Fig. 6 shows four Julia set images with different computation iterations.



(a) Original image

(b) Cropped image

Figure 7: The original image and the cropped anteater image.

Use the following constants for the variables in the pseudocode:

1. zoom = 1;
2. moveX = 0;
3. moveY = 0;
4. maxIterations = 256;
5. cRe = -0.7;
6. cIm = 0.27015;

Save the image with the name "juliaset" after this step.

1.4.5 BONUS: Crop the image

This function crops the image based on a set of user inputs. The user will indicate a starting pixel in the image by entering a x and y offset. Then the user specify the size of cropping by entering how many pixels the user wants to crop in the x and y directions. If the crop amount exceeds the image width or height (or both), the returned image will only crop up to the maximum length of the original image.

NOTE: This means that the picture should only allocate the minimum amount of memory needed to store the image. You need to define and implement the following function to do this DIP.

```
/* BONUS: Crop */
IMAGE *Crop(IMAGE *image, unsigned int x, unsigned int y, unsigned int W, unsigned int H);
```

Figure 7 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 17
Please enter the X offset value: 293
Please enter the Y offset value: 53
```

```
Please input the crop width: 305
Please input the crop height: 415
"Crop" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to Black & White
 4:  Flip an image vertically
 5:  Mirror an image horizontally
 6:  Color filter an image
 7:  Sketch the edge of an image
 8:  Sharpen an image
 9:  BONUS: Add Border to an image
 10: Add noise
 11: Overlay
 12: Posterize
 13: Bonus, Surprise Filter
 14: Resize the image
 15: Rotate 90 degrees clockwise
 16: Generate the Julia set image
 17: BONUS: Crop
 18: Exit
Please make your choice:
```

Save the image with the name 'crop' after this step.

1.5 Test all functions

Finally, you are going to complete the *AutoTest(IMAGE *image)* function to test all functions (as in Assignment 3(. In this function, you are going to call DIP functions one by one and store the results. The function is for the designer to quickly test the program and should require no user intervention. You should supply all the necessary parameters for testing. We will change the function signature for *AutoTest(IMAGE *image)* so that the creation and deletion of the image will be taken care of inside this function.

The function should look like:

```
/* auto test*/
void AutoTest(IMAGE *image)
{
char fname[SLEN] = "UCI_Peter";
char sname[SLEN];

image = ReadImage(fname);
BlackNWhite(image);
strcpy(sname, "bw");
SaveImage(sname, image);
printf("Black & White tested!\n\n");
DeleteImage(image);

...

image = ReadImage(fname);
strcpy(sname, "halloweenBat");
image = Overlay(sname, image, 100, 150) ;
strcpy(sname, "overlaybat");
```

```
SaveImage(sname, image) ;
printf("Overlay with same image sizes tested!\n\n");
DeleteImage(image);

/* one for overlay turkey */
...

image = ReadImage(fname);
image = Resize(175, image);
SaveImage("bigresize", image);
printf("Resizing big tested!\n\n");
DeleteImage(image);

/* one for small resize */
...

image = ReadImage(fname);
image = Rotate(image);
SaveImage("rotate", image);
printf("Rotate 90 degrees clockwise tested!\n\n");
DeleteImage(image);

image = Juliaset(600, 475, 256);
SaveImage("juliaset", image);
printf("Generate the juliaset image tested!\n\n");
DeleteImage(image);

image = ReadImage(fname);
image = Crop(image, 293, 53, 305, 415);
SaveImage("crop", image);
printf("Crop tested!\n\n");
DeleteImage(image);
}
```

Please hard-code "UCI_Peter" as the fname and pass an unallocated IMAGE struct to *AutoTest(IMAGE *image)* when is called in the *main* function.
Your program output should look like the following:

```
crystalcove% ./PhotoLabTest
UCI_Peter.ppm was read successfully!
bw.ppm was saved successfully.
bw.jpg was stored for viewing.
Black & White tested!

...

UCI_Peter.ppm was read successfully!
halloweenBat.ppm was read successfully!
overlaybat.ppm was saved successfully.
overlaybat.jpg was stored for viewing.
Overlay with same image sizes tested!

UCI_Peter.ppm was read successfully!
turkey.ppm was read successfully!
overlayturkey.ppm was saved successfully.
```

```
overlayturkey.jpg was stored for viewing.
Overlay with different image sizes tested!

UCI_Peter.ppm was read successfully!
bigresize.ppm was saved successfully.
bigresize.jpg was stored for viewing.
Resizing big tested!

UCI_Peter.ppm was read successfully!
smallresize.ppm was saved successfully.
smallresize.jpg was stored for viewing.
Resizing small tested!

UCI_Peter.ppm was read successfully!
rotate.ppm was saved successfully.
rotate.jpg was stored for viewing.
Rotate 90 degrees clockwise tested!

juliaset.ppm was saved successfully.
juliaset.jpg was stored for viewing.
Generate the juliaset image tested!

UCI_Peter.ppm was read successfully!
crop.ppm was saved successfully.
crop.jpg was stored for viewing.
Crop tested!
```

1.6 Extend the Makefile

For the **Makefile**,

- extend and adjust it properly with targets for your program with the new module: **Image.c**.
- generate 2 executable programs
  1. *PhotoLab* with the user interactive menu without the DEBUG statements.
  2. *PhotoLabTest*, an executable that just calls *AutoTest(IMAGE *image)* function (in DEBUG mode).

  Define two targets to generate these 2 programs respectively in addition to *all* and *clean*. You may define other targets as needed.

1.7 Use "Valgrind" tool to Find Memory Leaks and Invalid Memory Accesses

Valgrind is a multipurpose code profiling and memory debugging tool for Linux. It allows you to run your program in *Valgrind*'s own environment that monitors memory usage, such as calls to malloc and free. If you use uninitialized memory, write over the end of an array, or forget to free a pointer, *Valgrind* will detect it. You may refer to `http://valgrind.org/` for more details about the *Valgrind* tool.

In this assignment, please use the following command to check the correctness of your memory usages

```
valgrind --leak-check=full program_name
```

If there is no problem with the memory usage in your program, you will see information similar to the following upon completion of your program:

```
==xxxxx==
==xxxxx== HEAP SUMMARY:
==xxxxx==     in use at exit: 0 bytes in 0 blocks
==xxxxx==   total heap usage: 129 allocs, 129 frees, 20,476,437 bytes allocated
==xxxxx==
==xxxxx== All heap blocks were freed -- no leaks are possible
==xxxxx==
==xxxxx== For counts of detected and suppressed errors, rerun with: -v
==xxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

You need to compile your program with the *"-g"* option in *gcc* in order to enable detection of memory usage problems in your program.
If there are problems with your program's memory usage, *Valgrind* will provide you with information about the problem and where to fix it.
For your final submission, your program should be free of warnings and free of any errors reported by *Valgrind*.

2 Implementation Details

2.1 Function Prototypes

For this assignment, you need to define the following functions in **Advanced.h**:

```
/*** function declarations ***/

...

/* Resize */
IMAGE *Resize( unsigned int percentage, IMAGE *image);

/* Rotate */
IMAGE *Rotate(IMAGE *image);

/* Juliaset */
IMAGE *Juliaset(unsigned int W, unsigned int H, unsigned int max_iteration);

/* BONUS: Crop */
IMAGE *Crop(IMAGE *image, unsigned int x, unsigned int y, unsigned int W, unsigned int H);

/* Test all functions */
void AutoTest(IMAGE *image);
```

and in **Image.h**:

```
/*** function declarations ***/

/* Get the color intensity of the Red channel of pixel (x, y) in image */
unsigned char GetPixelR(IMAGE *image, unsigned int x,  unsigned int y);

/* Get the color intensity of the Green channel of pixel (x, y) in image */
unsigned char GetPixelG(IMAGE *image, unsigned int x,  unsigned int y);

/* Get the color intensity of the Blue channel of pixel (x, y) in image */
unsigned char GetPixelB(IMAGE *image, unsigned int x,  unsigned int y);
```

```
/* Set the color intensity of the Red channel of pixel (x, y) in image with value r */
void SetPixelR(IMAGE *image, unsigned int x,  unsigned int y, unsigned char r);

/* Set the color intensity of the Green channel of pixel (x, y) in image with value g */
void SetPixelG(IMAGE *image, unsigned int x,  unsigned int y, unsigned char g);

/* Set the color intensity of the Blue channel of pixel (x, y) in image with value b */
void SetPixelB(IMAGE *image, unsigned int x,  unsigned int y, unsigned char b);

/* allocate the memory space for the image structure         */
/* and the memory spaces for the color intensity values.     */
/* return the pointer to the image, or NULL in case of error */
IMAGE *CreateImage(unsigned int Width, unsigned int Height);

/* release the memory spaces for the pixel color intensity values */
/* deallocate all the memory spaces for the image                 */
void DeleteImage(IMAGE *image);
```

You may want to define other functions as needed.

2.2 Pass in the pointer of the struct IMAGE

In the main function, define the struct variable *image* of type IMAGE. It will contain the following image information: *Width*, *Height*, pointers to the memory spaces for all the color intensity values of the *R*, *G*, *B* channels.
When any of the DIP operations are called in the main function, the address of this *image* variable is passed into the DIP functions. This way, the DIP functions can access and modify the contents of this variable.
In your DIP function implementation, there are two ways to save the target image information. Both options work and you should decide which option is better based on the specific DIP manipulation function at hand.

Option 1: using local variables You can define local variables of type IMAGE to save the target image information. For example:

```
IMAGE *DIP_function_name(IMAGE *image)
{
    IMAGE *image_tmp;

    image_tmp = CreateImage(image->Width, image->Height);

    ...

    DeleteImage(image_tmp);
    image_tmp = NULL;
    return image;
}
```

Make sure you create and delete the image space properly.
Then, at the end of each DIP function implementation, you can copy the data in *image_tmp* over to *image*, or delete the incoming image and return the new one.

Option 2: in place manipulation Sometimes you do not have to create new local array variables to save the target image information. Instead, you can just manipulate on *image.R*, *image.G*, *image.B* directly. For example, in the implementation of BlackAndWhite() function, you can assign the result pixel value directly back to the pixel entry.

NOTE: Please always call SetPixelR (SetPixelG, SetPixelB) function to set the pixel color value and GetPixelR (GetPixelG, GetPixelB) function to read the pixel color value.

3 Budgeting your time

You have two weeks to complete this assignment, but we encourage you to get started early as there is more work than Assignment 3. We suggest you budget your time as follows:

- Week 1:
  1. Design the **Image.c** (**Image.h** as the header file) module.
  2. Change the signature and definitions of the existing functions.
  3. Modify the *AutoTest(IMAGE *image)* function.
  4. Adjust the **Makefile** with the targets for the new module.
  5. Implement one DIP function if possible.
- Week 2:
  1. Implement all the advanced DIP functions.
  2. Complete the *AutoTest(IMAGE *image)* function.
  3. Use *Valgrind* to check memory usages. Fix the code if *Valgrind* complains about any errors.
  4. Script the result of your programs and submit your work.

4 Script File

To demonstrate that your program works correctly, perform the following steps and submit the log as your script file:

1. Start the script by typing the command: *typescript*.
2. Compile and run *PhotoLab* by using your **Makefile**.
3. Choose a few functions to implement (The file names must be 'bw', ..., 'overlaybat', 'overlayturkey', 'bigresize', 'smallresize', 'rotate', 'juliaset' for the corresponding functions).
4. Exit the program.
5. Compile and run *PhotoLabTest* by using your **Makefile**.
6. Run *PhotoLabTest* under the monitor of *Valgrind*.
7. Clean all the object files, generated .ppm files and executable programs by using your **Makefile**.
8. Stop the script by typing the command: *exit*.
9. Rename the script file to *PhotoLab.script*.

NOTE: make sure use exactly the same names as shown in the above steps when saving modified images! The script file is important, and will be checked in grading; you must follow the above steps to create the script file. ***Please don't open any text editor while scripting !!!***

5 Submission

Use the standard submission procedure to submit the following files as the whole package of your program:

- *PhotoLab.c*
- *PhotoLab.script*
- *PhotoLab.txt*

- *Image.c*
- *Image.h*
- *Constants.h*
- *DIPs.c*
- *DIPs.h*
- *FileIO.c*
- *FileIO.h*
- *Advanced.c*
- *Advanced.h*
- *Makefile*

Please leave the images generated by your program in your *public_html* directory. Don't delete them as we may consider them when grading! You don't have to submit any images.