EECS 22: Assignment 3

Prepared by: Yasamin Moghaddas, Yang Xiang, Mark Boyher, Prof. Halima Bouzidi, Weiyu Luo, Prof. Rainer Dömer

January 29, 2024

Due on Sunday 02/16/2025 at noon, 12:00 pm.

Contents

1 Digital Image Processing

In this assignment you will learn how to break a program into multiple modules, and compile them into one program. Based on the program *PhotoLab* for Assignment 2, you will be asked to develop some advanced digital image processing (DIP) operations, partition them in separate modules, manipulate images using bit operations, and develop an appropriate **Makefile** to compile your program with DEBUG mode on or off.

1.1 Introduction

In Assignment 2, you were asked to develop an image manipulation program *PhotoLab* by using DIP techniques. The user can load an image from a file, apply a set of DIP operations to the image, and save the processed image in a file by using the *PhotoLab*. This assignment will be based on Assignment 2.

1.2 Initial Setup

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

```
mkdir hw3
cd hw3
cp  ~eecs22/public/PhotoLab_v2.c .
cp  ~eecs22/public/EngPlaza.ppm .
```

NOTE: Execute the above setup commands only ONCE before you start working on the assignment! Do not execute them after you start the implementation, otherwise your code will be overwritten!

We will extend the *PhotoLab* program based on Assignment 2. You must use the provided template file PhotoLab_v2.c file.

Once a DIP operation is done, you can save the modified image as *name*, and it will be automatically converted to a JPEG image and sent to the folder *public_html* in your home directory. Then you are able to see the image in a web browser at: ***http://eecs.uci.edu/∼youruserid***, if required names are used (i.e. 'bw', 'negative', 'colorfilter', 'edge', 'shuffle', 'vflip', 'hmirror', 'pixelate', 'fisheye', 'posterize', 'rotate', 'blur' for each corresponding function). If you save images by other names, use the link ***http://eecs.uci.edu/∼youruserid/imagename.jpg*** to access the photo.

You will also be able to see your images compared with the reference images in a web browser at:
http://eecs.uci.edu/∼youruserid/diff.html. You can use this page to make sure that your images match the expected solution.
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 Decompose the program into multiple modules

Decompose the **PhotoLab_v2.c** file into multiple modules and header files:

- **PhotoLab.c**: this module contains the *main()* function, and the menu function *PrintMenu()* as well as *AutoTest()*.
- **FileIO.c**: this module contains the function definitions of *LoadImage()* and *SaveImage()*.
- **FileIO.h**: the header file for **FileIO.c**, with the function declarations of *LoadImage()* and *SaveImage()*.
- **Constants.h**: the header file in which the constants to be used are defined.
- **DIPs.c**: this module contains the DIP function definitions from Assignment 2, i.e. *BlackNWhite()*, *Negative()*, *ColorFilter()*, *Edge()*, *Shuffle()*, *VFlip()*, *HMirror()*, *Pixelate()*.
- **DIPs.h**: the header file for **DIPs.c**, with the DIP function declarations.
- **Advanced.c**: this module contains the function definitions of new filters in Assignment 3, i.e. *FishEye()*, *Rotate()*, *Posterize()*, and *MotionBlur()*.
- **Advanced.h**: the header file for **Advanced.c**, with the function declarations of *FishEye()*, *Rotate()*, *Posterize()*, and *MotionBlur()*.

NOTE: Advanced.c and Advanced.h will be incomplete if you still DID NOT implement the advanced DIP functions yet.

HINT: Please refer to the slides of *lectures covering Compiler components, translation units, Make and Makefile* for an example of decomposing programs into different modules.

1.4 Compile the program with multiple modules using static shared library

The *PhotoLab* program is now modularized into different modules: **PhotoLab**, **FileIO**, **DIPs** and **Advanced**. In this assignment we are using shared libraries to group the compiled object code files into static libraries. Often C functions and methods which can be shared by more than one application are broken out of the application's source code, compiled and bundled into a library.

As was explained in the lecture, in order to generate the libraries first compile the source code into object files. Use *"-c"* option for **gcc** to generate the object files for each module, e.g.

```
% gcc -c FileIO.c -o FileIO.o -Wall -std=c11
% gcc -c Advanced.c -o Advanced.o -Wall -std=c11
...
```

NOTE: If you encounter this error: `% cc1:  error:  unrecognized command line option "-std=c11"`
Then use this command before using gcc: `% scl enable devtoolset-7 tcsh`

As explained in the Make and Makefile lecture, libraries are typically named with the prefix "lib". Here we want to create a library named *libfilter*:

```
% ar rc libfilter.a DIPs.o Advanced.o
% ranlib libfilter.a
```

Linking with the library:

```
% gcc -Wall PhotoLab.o FileIO.o -lfilter -L. -lm -o PhotoLab
```

Execute the program:

```
% ./PhotoLab
program executes
% _
```

1.5 Using 'make' and 'Makefile'

On the other hand, we can put the commands above into a **Makefile** and use the *make* utility to automatically build the executable program from source code. Create your own **Makefile** with at least the following targets:

- *all*: the target to generate the executable programs.
- *clean*: the target to clean all the intermediate files, e.g. object files, autogenerated images, and the executable program(s). Be careful to only delete intermediates files, not any of your true source files.
- *PhotoLab*: the target to generate the executable program *PhotoLab*.

To use your **Makefile**, use this command:

```
% make all
```

The executable program *PhotoLab* shall then be automatically generated.

Requirement: There must be a rule for each object file depending on the corresponding .c file and any other needed dependency. Dependencies which are not needed will reduce the points.

HINT: Please refer to the slides of *Lecture 11* (Covered on Feb. 02, 2024) for an example on how to create a **Makefile**.

1.6 Advanced DIP operations

In this assignment, you will use **Advanced.c** and **Advanced.h** to implement the advanced DIP operations described below.

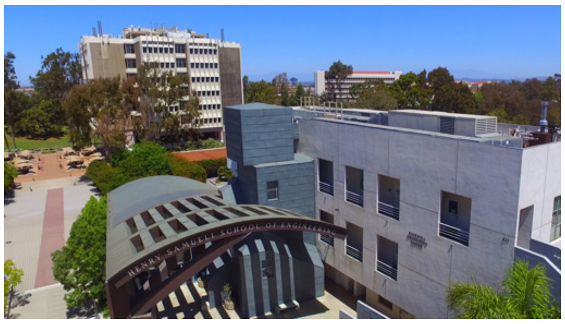
Reuse the menu you designed for Assignment 2 and extend it with the advanced operations. The user should be able to select DIP operations from a menu as the one shown below:

```
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Color filter an image
 6:  Sketch the edge of an image
 7:  Shuffle an image
 8:  Flip an image vertically
 9:  Mirror an image horizontally
 10: Pixelate the image
 11: Create a fisheye image
 12: Posterize an image
 13: Rotate and zoom an image
 14: Motion Blur
 15: Test all functions
 16: Exit
please make your choice:
```

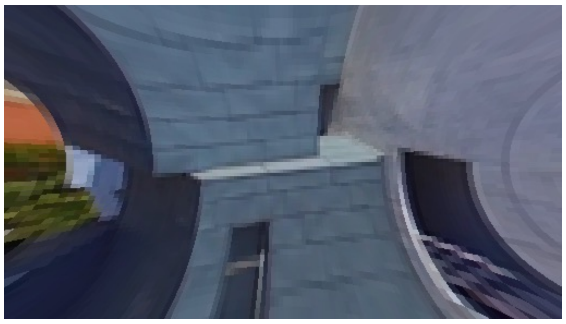
1.6.1 Create a fisheye image

In this operation, you will create a fisheye image. The fisheye effect distorts an image by stretching the picture around a rounded camera lens. As a result, the fisheye lens exaggerates the size of objects in the center of your photograph by creating a wide-angle warping effect. The fisheye effect does not necessarily ”zoom in” in the traditional sense, but it can make the image appear as though it is warped and focused on the central area, with objects near the edges of the image appearing larger and distorted. Additionally, depending on the parameters, the FishEye function may apply some scaling, which could make the image appear either larger or smaller after the distortion is applied.

Note that the fisheye effect’s distortion is not constant; rather, it increases based on the radial distance from the center.



(a) Original image



(b) a image with distortion_factor=0.5 , k=0.5, scaling_factor=1.5

Figure 1: An image and its specific fisheye counterpart.

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

```
/* Create a fisheye image */
void FishEye(unsigned char R[WIDTH][HEIGHT],
                unsigned char G[WIDTH][HEIGHT],
                unsigned char B[WIDTH][HEIGHT],
                double base_factor,
                double k,
                double scaling_factor);
```

For the implementation of the fisheye effect, we provide the following pseudocode:

Algorithm 1 Dynamic Fisheye Effect

```
1:  Function FishEye($R, G, B, base\_factor, k, scaling\_factor$)
2:  Input: $R, G, B$: 2D arrays for red, green, and blue channels of the image
3:      $base\_factor$: Base distortion factor
4:      $k$: Rate of distortion increase with radius
5:      $scaling\_factor$: Scaling factor for controlling the overall image size
6:  Output: The transformed image in $R, G, B$
7:  # Create output arrays
8:  $R\_out, G\_out, B\_out$: Arrays to store the transformed pixels
9:  # Initialize image center
10: $center\_x \leftarrow \frac{WIDTH}{2}, center\_y \leftarrow \frac{HEIGHT}{2}$
11: for each pixel $(x, y)$ in the image do
12:     # Calculate the normalized distance from the center
13:     $dx \leftarrow \frac{x - center\_x}{center\_x}, dy \leftarrow \frac{y - center\_y}{center\_y}$
14:     $radius \leftarrow \sqrt{dx^2 + dy^2}$
15:     # Calculate the distortion factor based on radius
16:     $distortion \leftarrow (1.0 + k \times radius^2)$
17:     # Apply fisheye transformation (polar coordinates)
18:     $theta \leftarrow \text{atan2}(dy, dx)$
19:     $new\_radius \leftarrow \frac{radius \times base\_factor}{distortion \times scaling\_factor}$
20:     # Convert back to Cartesian coordinates
21:     $x\_src \leftarrow center\_x + (new\_radius \times \cos(\theta) \times center\_x)$
22:     $y\_src \leftarrow center\_y + (new\_radius \times \sin(\theta) \times center\_y)$
23:     # Check if source coordinates are within bounds and copy pixel
24:     if $0 \leq x\_src < WIDTH$ and $0 \leq y\_src < HEIGHT$ then
25:         $R\_out[x][y] \leftarrow R[x\_src][y\_src]$
26:         $G\_out[x][y] \leftarrow G[x\_src][y\_src]$
27:         $B\_out[x][y] \leftarrow B[x\_src][y\_src]$
28:     else
29:         $R\_out[x][y] \leftarrow 0$
30:         $G\_out[x][y] \leftarrow 0$
31:         $B\_out[x][y] \leftarrow 0$
32:     end if
33: end for
34: # Copy the result back to the original image arrays
35: for each pixel $(x, y)$ in the image do
36:     $R[x][y] \leftarrow R\_out[x][y]$
37:     $G[x][y] \leftarrow G\_out[x][y]$
38:     $B[x][y] \leftarrow B\_out[x][y]$
39: end for
40: End Function
```

Figure 1 shows an example of this operation where the base_factor is 0.5, k is 0.5 and the scaling_factor is 1.5. Once the user chooses this option, **and before the whole menu is printed out again**, your program's output should look like this:

```
Please make your choice: 11
Enter a value for base factor: 0.5
Enter a value for k: 0.5
Enter a value for scaling factor: 1.5
"FishEye" operation is done!
```

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

1.6.2 Bit Manipulations: Posterize an image

Posterization of an image entails conversion of a continuous gradation of tone to several regions of fewer tones, with abrupt changes from one tone to another. This was originally done with photographic processes to create posters. It can now be done photographically or with digital image processing, and may be deliberate or may be an unintended artifact of color quantization. (`http://en.wikipedia.org/wiki/Posterization`).

We are going to use bit manipulations to posterize the image. As before, a pixel in the image is represented by a 3-tuple (r, g, b) where *r*, *g*, and *b* are the values for the intensities of the red, green, and blue channels respectively. The range of *r*, *g*, and *b* are from 0 to 255 inclusively. As such, we use *unsigned char* variables to store the values of these three values.

To posterize the image, we are going to change the least *n* (where $n \in \{1, 2, 3, \dots 8\}$) significant bits of color intensity values so as to change the tone of the pixels. Basically, we will change the *n*th least significant bit of the color intensity value to be 0, and the least $n-1$ bits to be all 1. For example, assume that the color tuple of the pixel at coordinate (0,0) is (41, 84, 163). Therefore,

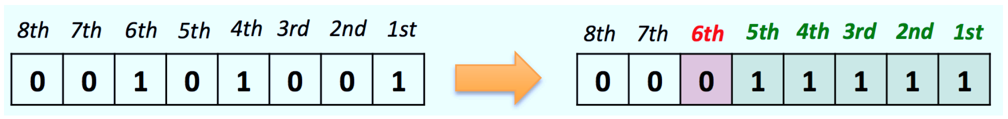
```
R[0][0] = 41;
G[0][0] = 84;
B[0][0] = 163;
```

In binary representation, the color tuple will be:

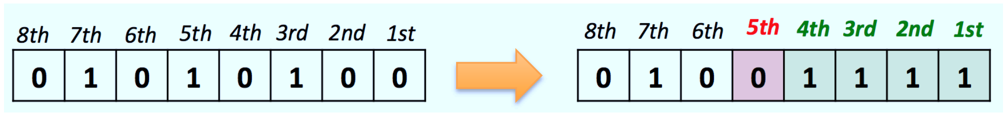
$$R[0][0] = 00101001_2;$$
$$G[0][0] = 01010100_2;$$
$$B[0][0] = 10100011_2;$$

Fig. 2 shows the operation for posterize for different least significant bits of the intensities for the red, green, and blue channels. As illustrated in Fig. 2(a), in order to posterize the least 6 significant bits of the red intensity, we set the 6th bit to be 0, and the 1st to the 5th bits to be 1s. Similarly in Fig. 2(b), to posterize the least 5 significant bits of the green intensity, we set the 5th bit to be 0, and the 1st to the 4th bits to be 1s; and in Fig. 2(c), to posterize the least 4 significant bits of the blue intensity, we set the 4th bit to be 0, and the 1st to the 3rd bits to be 1s.
Function Prototype: You need to define and implement the following function to do this DIP.

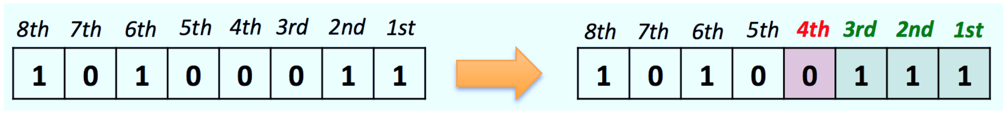
```
/* Posterize the image */
void Posterize(unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT],
    unsigned int rbits,
    unsigned int gbits,
    unsigned int bbits);
```


8th 7th 6th 5th 4th 3rd 2nd 1st
0 0 1 0 1 0 0 1
8th 7th 6th 5th 4th 3rd 2nd 1st
0 0 0 1 1 1 1 1

(a) Posterize the least 6 significant bits of the red channel for pixel(0,0)


8th 7th 6th 5th 4th 3rd 2nd 1st
0 1 0 1 0 1 0 0
8th 7th 6th 5th 4th 3rd 2nd 1st
0 1 0 0 1 1 1 1

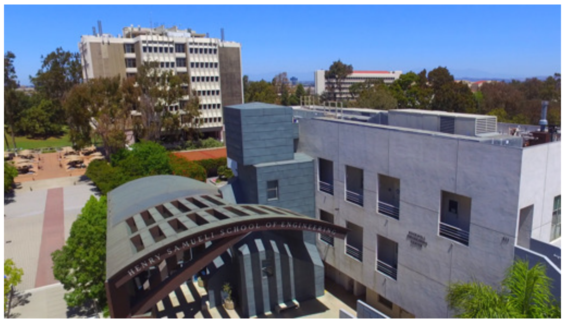
(b) Posterize the least 5 significant bits of the green channel for pixel(0,0)


8th 7th 6th 5th 4th 3rd 2nd 1st
1 0 1 0 0 0 1 1
8th 7th 6th 5th 4th 3rd 2nd 1st
1 0 1 0 0 1 1 1

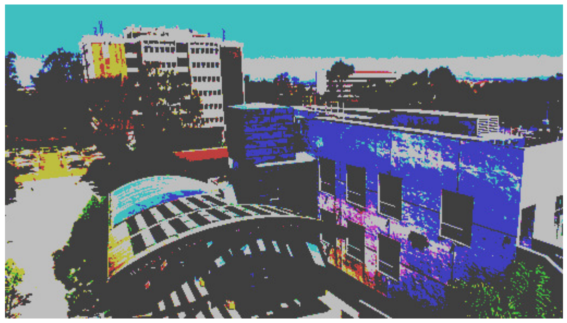
(c) Posterize the least 4 significant bits of the blue channel for pixel(0,0)

Figure 2: The example of posterizing the color channels.

Here, *rbits*, *gbits*, and *bbits* specify the number of least significant bits that need to be posterized. Since the size of *unsigned char* variable is 8 bits, the valid range of *rbits*, *gbits*, and *bbits* will be 1 to 8.
HINT: You will need to use bitwise operators, e.g. '&', '<<', '>>', '|' for this operation.



(a) Image without posterization



(b) Image with posterization, where rbits = 7, gbits = 7, bbits = 7

Figure 3: The image and its posterized counterpart.

Fig. 3 shows an example of our posterized image. Once the user chooses this option, **and before the whole menu is printed out again**, your program's output should look like this:

```
please make your choice: 12
Enter the number of posterization bits for R channel (1 to 8): 7
Enter the number of posterization bits for G channel (1 to 8): 7
Enter the number of posterization bits for B channel (1 to 8): 7
"Posterize" operation is done!
```

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

1.6.3 Rotate and zoom an image

Rotation matrix(`https://en.wikipedia.org/wiki/Rotation_matrix`) is a transformation matrix for executing a rotation in Euclidean space.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Based on the rotation matrix, we can achieve the desired outcome of rotating an image around a specific center point and adjusting the scale of the image through the manipulation of the matrix in the following matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta/\alpha & -\sin\theta/\alpha \\ \sin\theta/\alpha & \cos\theta/\alpha \end{bmatrix} \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} C_x \\ C_y \end{bmatrix}$$

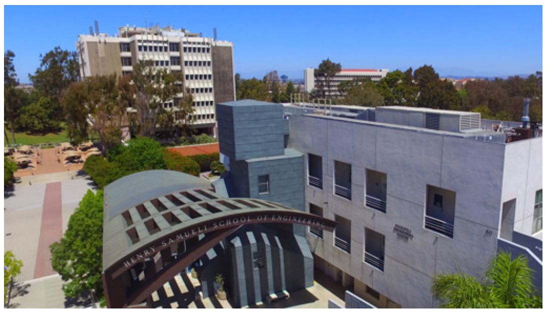
To rotate an image around a specific center point and adjust the scale of the image, you need to apply this matrix to get the new x and y coordinates. α is a scale factor for zooming. C_x and C_y are the coordinates of the center point. Repeat this for every pixel, and for every color channel (red, green, and blue) of the image. You can use `cos()` and `sin()` functions in `math.h`. Remember to convert the angle to radian before you apply the functions.

Note that image rotation often incurs pixels that lie outside of the source and/or the target image. To deal with this situation, have your DIP operation traverse all pixels in the target image and then check whether or not the corresponding source pixel lies within the original image (x' and y' coordinates within [0, WIDTH - 1] and [0, HEIGHT - 1] ranges, respectively). For pixels with source coordinates outside the original image, assume they are black color. For ease of implementation, use black blocks to fill in the pixels that are not assigned values after rotation.

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

```
/* rotate and zoom the image */
void Rotate(unsigned char R[WIDTH][HEIGHT],
            unsigned char G[WIDTH][HEIGHT],
            unsigned char B[WIDTH][HEIGHT],
            double Angle,
            double ScaleFactor,
            int CenterX,
            int CenterY)
```

The rotate image should look like the figure shown in Figure 4(b):


(a) Original Image

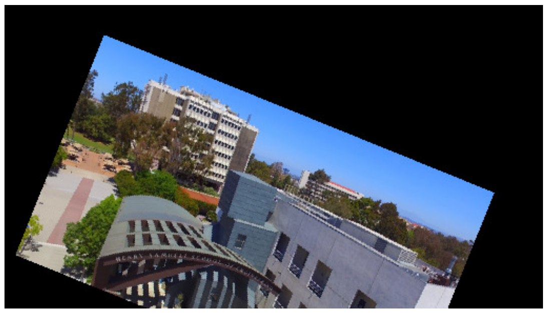

(b) Rotated Image

Figure 4: An image and its rotated counterpart.

Once the user chooses this option, **and before the whole menu is printed out again**, your program's output should look like this:

```
Please make your choice: 13
Enter the angle of rotation: 22
Enter the scale of zooming: 0.78
Enter the X-axis coordinate of the center of rotation: 110
Enter the Y-axis coordinate of the center of rotation: 220
"Rotate" operation is done!
```

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

1.6.4 Motion Blur: Bonus

Any kind of blur is essentially making every pixel more similar to those around it. In a horizontal blur, we can average each pixel with those in a specific direction, which gives the illusion of motion.
For this program, we will calculate each new pixel's value as half of its original value. The other half is averaged from a fixed number of pixels to the right. This fixed number can be called the `bluramount`. The larger the value, the more blurring will occur.

This is applied for the red, green, and blue intensity of each pixel.

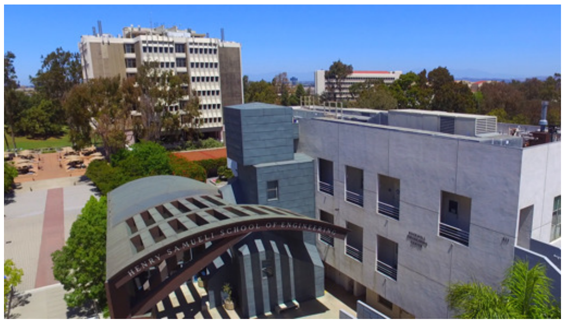
You must also ensure you don't access pixels off the bounds of the image. For example, the third pixel from the right should only average itself (half weight), and the next two (at a quarter weight each).

For this function, prompt the user for the `bluramount` (see the output of the program with this option below for "printing" details).

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

```
/* Make a blurred image*/
void MotionBlur(int BlurAmount,
    unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT]);
```

Here, *BlurAmount* specifies the amount of blurring in the image.


(a) Original image

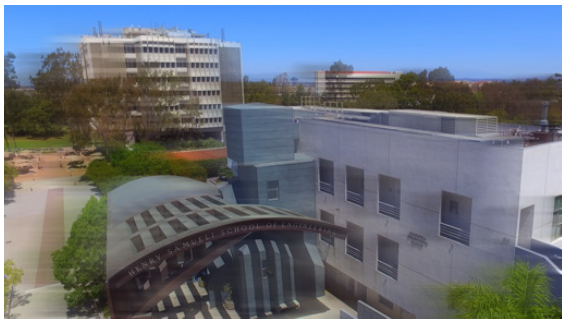

(b) blurred image with motion blur=40

Figure 5: An image and its motion blur.

Figure 5 shows an example of this operation where *blouramount* is 40. Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 14
Please input blur amount: 40
"motion blur" operation is done!
```

Save the image with name ’blur’ after this step.

1.7 Test all functions

Finally, you are going to complete the *AutoTest()* function to test all the functions. In this function, you are going to call DIP and advanced functions one by one and save the results. The function is for the designer to quickly test the program, so you should supply all necessary parameters when testing.

Note that the below DIP function calls should only be included in the test all functions if you actually implement the DIP functions themself.

The function should look like:

```
void AutoTest(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
 unsigned char B[WIDTH][HEIGHT])
{
  char fname[SLEN] = "EngPlaza";

  LoadImage(fname, R, G, B);
  Negative(R, G, B);
  strcpy(sname, "negative");
  SaveImage(sname, R, G, B);
  printf("Negative tested!\n\n");

  LoadImage(fname, R, G, B);
  ColorFilter(R, G, B, 130, 130, 150, 30, 0, 255, 255);
  strcpy(sname, "colorfilter");
  SaveImage(sname, R, G, B);
  printf("Color Filter tested!\n\n");

  ...

  LoadImage(fname, R, G, B);
  Pixelate (R, G, B, 4);
  strcpy(sname, "pixelate");
  SaveImage(sname, R, G, B);
  printf("Pixelate tested!\n\n");

  ...

  LoadImage(fname, R, G, B);
  FishEye(R, G, B, 0.5, 0.5, 1.5);
  strcpy(sname, "fisheye");
  SaveImage(sname, R, G, B);
  printf("FishEye tested!\n\n");

  LoadImage(fname, R, G, B);
  Posterize(R, G, B, 7, 7, 7) ;
  strcpy(sname, "posterize");
  SaveImage(sname, R, G, B);
  printf("Posterize tested!\n\n");

  LoadImage(fname, R, G, B);
  Rotate(R, G, B, 22, 0.78, 110, 220);
  strcpy(sname, "rotate");
  SaveImage(sname, R, G, B);
  printf("Rotate tested!\n\n");

  LoadImage(fname, R, G, B);
  MotionBlur(40, R, G, B) ;
  strcpy(sname, "blur");
  SaveImage(sname, R, G, B);
  printf("MotionBlur tested!\n\n");
}
```

Implement the new (extended) *AutoTest()* function in **Photolab.c**. Since the *AutoTest()* function will call the functions in the **DIPs.c** and **Advanced.c** modules, make sure to include the header files properly. Also, be sure to adjust your **Makefile** for proper dependencies.

1.8 Support for the DEBUG mode

In C programs, *macros* can be defined as preprocessing directives. Define a macro named **"DEBUG"** in your source code to enable/disable the messages shown in the *AutoTest()* function.

When the macro is defined, the main menu will not appear, your program executes only the function *AutoTest()*, prints its associated print statements (the messages in the *AutoTest()* show up) and finishes afterwards.

If the macro is not defined, the program will execute in its regular fashion and the main menu will appear. The printf statements in the `LoadImage()` and `SaveImage()` function will also show up. In this case however, the messages in the function *AutoTest()* will not show up.

You should decide in which function and in which module this **"DEBUG"** macro needs to be added.

1.9 Extend the Makefile

For the **Makefile**,

- extend it properly with the targets for your program with the new module: **Advanced.c**.
- generate two executable programs
  1. *PhotoLab* with the user interactive menu and the **DEBUG** mode off.
  2. *PhotoLabTest* without the user menu, but with only the *AutoTest()* function for testing, and turn the **DEBUG** mode on. Note that we can thus use the same source files to generate two different programs.

  Define two targets to generate these two programs. Use the *"-D"* option for gcc to enable/disable the DEBUG mode instead of defining the "DEBUG" macro in the source code. You may need to define more targets to generate the object files with different DEBUG modes.

2 Implementation Details

2.1 Function Prototypes

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

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

/* create the fisheye image */
void FishEye(unsigned char R[WIDTH][HEIGHT],
             unsigned char G[WIDTH][HEIGHT],
             unsigned char B[WIDTH][HEIGHT],
             double base_factor,
             double k_factor,
             double scaling_factor);

/* posterize the image */
void Posterize(unsigned char R[WIDTH][HEIGHT],
             unsigned char G[WIDTH][HEIGHT],
             unsigned char B[WIDTH][HEIGHT],
             unsigned int rbits,
             unsigned int gbits,
             unsigned int bbits);

/* rotate and zoom the image */
void Rotate(unsigned char R[WIDTH][HEIGHT],
            unsigned char G[WIDTH][HEIGHT],
            unsigned char B[WIDTH][HEIGHT],
            double Angle,
            double ScaleFactor,
            int CenterX,
            int CenterY);

/* motion blur */
void MotionBlur(int BlurAmount,
             unsigned char R[WIDTH][HEIGHT],
             unsigned char G[WIDTH][HEIGHT],
             unsigned char B[WIDTH][HEIGHT]);
```

You may want to define other functions as needed.

2.2 Global constants

The following global constants should be defined in **Constants.h** (they are also declared in *PhotoLab_v2.c*, please don't change their names):

```
static const int WIDTH=512;           /* image width */
static const int HEIGHT=288;          /* image height */
static const int SUCCESS=0;            /* return code for success */
static const int EXIT=16;              /* menu item number for EXIT */
static const int MAX_PIXEL=255;        /* max pixel value */
static const int MIN_PIXEL=0;          /* min pixel value */
static const int SHUFF_HEIGHT_DIV=4;   /* Height division for shuffling */
static const int SHUFF_WIDTH_DIV=4;    /* Width division for shuffling */
```

make sure that you properly include this header file when necessary.

3 Budgeting your time

You have two weeks to complete this assignment, but we encourage you to get started early as there is a little more work than for Assignment 2. We suggest you budget your time as follows:

- Week 1:
    1. Implement all the advanced DIP functions.

2. Implement the *AutoTest()* function.
3. Figure out how to enable/disable the **DEBUG** mode in the source code and add targets to the **Makefile** accordingly.
4. Script the result of your programs and submit your work.
5. Bonus part

- Week 2:

1. Decompose the given template program into different modules, i.e. **PhotoLab.c**, **FileIO.c**, **FileIO.h**, **Constants.h**, **DIPs.c**, **DIPs.h**.
2. Create module **Advanced.c**, **Advanced.h**, and implement an initial advanced DIP function.
3. Create your own **Makefile** and use it to compile the program.

4 Script File

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

1. Start the script by typing the command: *script*.
2. Compile and run *PhotoLab* by using your **Makefile**: type 'make clean', then 'make', then './PhotoLab'.
3. Choose 'Test all functions' (The file names hardcoded in this function must be 'bw', 'negative', 'colorfilter', 'edge', 'shuffle', vflip', 'hmirror', 'pixelate', 'fisheye', 'posterize', 'rotate', and 'blur' for the corresponding function).
4. Exit the PhotoLab.
5. Compile and run *PhotoLabTest*: type 'make PhotoLabTest'.
6. Test the dependencies in your Makefile: type 'touch Advanced.c', then 'make PhotoLab'.
7. Stop the script by typing the command: *exit*.
8. Rename the script filename of *typescript* to *PhotoLab.script*.

NOTE: make sure to 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 for grading; you must follow the above steps to create the script file. ***Do not open any text editor while scripting.***

5 Submission

5.1: **Assignment 3.1 - Autograder**
Submit the following files together to Gradescope under assignment name 3.1. **(DO NOT PLACE these files in a zip file as Autograder is not setup to unzip them.)** Ensure that your assignment passes all tests. If it does not, resubmit the files before the due date.

- *PhotoLab.c*
- *FileIO.c*
- *FileIO.h*
- *Constants.h*
- *DIPs.c*
- *DIPs.h*

- *Advanced.c*
- *Advanced.h*

Please DO NOT change the file names.

5.2: **Assignment 3.2 - Manual Verifications**
Important: Ensure that your software is well documented with clear, concise, and useful comments. Proper comments should explain the logic behind your solution, describe complex code, or clarify non-obvious implementation details. Avoid excessive comments that repeat what the code already conveys, insufficient comments that leave the reader guessing, or irrelevant comments that do not add value.

Important: Also, prepare a pdf file called PhotoLab.pdf (use your local computer operating system) where you briefly explain how you designed your program and any hardships faced (if any).

Insert the following files into one zip file named PhotoLab.zip and submit it to Canvas under Assignment 3.2. **Note that this zip file just adds PhotoLab.script, Makefile and PhotoLab.pdf to the files from Assignment 3.1 above. If needed, you can resubmit these files into a zip file anytime before the due date.**

- *PhotoLab.script*
- *PhotoLab.pdf*
- *Makefile*
- *PhotoLab.c*
- *FileIO.c*
- *FileIO.h*
- *Constants.h*
- *DIPs.c*
- *DIPs.h*
- *Advanced.c*
- *Advanced.h*

6 Grading

6.1: **Assignment 3.1 - Autograder**

- *DIP FishEye: 15 points*
- *DIP Rotate: 15 points*
- *DIP Posterize: 15 points*
- *Menu Operation: 10 points*
- *Autotest with DEBUG Application: 5 points*
- *Motion Blur: 10 points (Bonus - extra credit)*

6.1: **Assignment 3.2 - Manual Verification**

- *Makefile (using both PhotoLab.script and Makefile): 30 points*
- *Source Code Comments and Project Documentation PhotoLab.pdf: 10 points*