1 Overview

1.1 Description

In this assignment, we were tasked with creating a number of filters to process and modify digital images.

The first three filters, NoRed, NoGreen, and NoBlue, simply remove their respective color from the image. The next three, RedOnly, GreenOnly, and BlueOnly, do the opposite and remove all colors except one. BlackAndWhite turns a color image into grayscale. VerticalReflect and HorizontalReflect mirror the image over the respective axes. Grow doubles the the width and length of the image, while Shrink halves them. Threshold simplifies the image to use only 8 colors by setting each component to either fully on or off.

1.2 Goals

In completing this assignment, my goal was to understand how some of the image processing techniques actually work. Implementing the more complex neighborhood filters was valuable in understanding how larger features in images can be modified.

Furthermore, this assignment allowed me to practice good coding style and documentation, while also learning how to write reports for my projects.

2 Solution Design

NoRed: To implement this effect, the program simply iterates over all the pixels in the image and sets the red component of each pixel to 0.
**NoGreen:** This effect is implemented in the same way as the **NoRed** effect. The only difference is that instead of modifying the red components, it modifies the green ones.

**NoBlue:** This effect is implemented in the same way as the **NoRed** effect. The only difference is that instead of modifying the red components, it modifies the blue ones.

**RedOnly:** To apply this effect, the program iterates over all the pixels in the image and keeps the red component as-is and sets the other to 0. To avoid redundant code, the way this is implemented is by applying the **NoGreen** and **NoBlue** filters to the input image.

**GreenOnly:** This effect is implemented in the same way as the **RedOnly** effect. The only difference is that instead of applying the **NoGreen** filter, it applies the **NoRed** filter.

**BlueOnly:** This effect is implemented in the same way as the **RedOnly** effect. The only difference is that instead of applying the **NoBlue** filter, it applies the **NoRed** filter.

**BlackAndWhite:** To apply this effect, the program iterates over each pixel in the image and calculates the “brightness” of each pixel by taking the mean of the red, green, and blue, components. It then sets each component of the pixel to this brightness value.

**VerticalReflect:** To apply this effect, the program makes a new two-dimensional array of the same size as the input image. It then iterates through each pixel and copies it the blank array, but to location across the vertical axis from the original location. To do this, it maps each original x value to width \(- x - 1\).

**HorizontalReflect:** This effect is implemented in the same way as the **VerticalReflect** effect. The only difference is that instead of modifying the x coordinate, it maps the original y value to height \(- y - 1\).

**Grow:** To implement this effect, the program creates a new two-dimensional array that is double the width and height of the input image. The program then iterates over each pixel in the
input image and copies the value to a square of four pixels in the blank array. This effectively double the width and height of the image.

**Shrink:** To implement this effect, the program creates a new two-dimensional array that is half the width and height of the input image. The program then iterates over each pixel in the empty array sets it to the componentwise mean of the pixels in a square of four pixels from the input image. This effectively halves the width and height of the image.

**Threshold:** For this effect, the program iterates over each pixel in the input image and sets each component to either fully on (255) or fully off (0) depending on whether the current value is greater or less than the threshold value, which is specified by the user. This results in the final image only being composed of eight colors, rather than millions.

**Smooth:** For this effect, the program first creates a new two-dimensional array that is the same size as the input image. Then, while iterating over each pixel in the input image, it finds the bounds of a box centered on that pixel and with a side length that the user has specified. It truncates the box so that it does not contain pixels outside the bounds of the image. It then sets the blank pixel to the componentwise mean of the pixels in the box from the input image. This results in the image being blurred. The size of the square specifies the “blurriness.”

**DeNoise:** The implementation of this effect is very similar to the Smooth filter. The only difference is that instead of taking the mean of the pixels in the square, this filter takes the median for each component. This removes any random noise from the image.

**Erode:** The implementation of this effect is very similar to the Smooth filter. The only difference is that instead of taking the mean of the pixels in the square, this filter takes the minimum for each component. This grows darker areas and shrinks lighter ones.

**Dilate:** The implementation of this effect is very similar to the Smooth filter. The only difference is that instead of taking the mean of the pixels in the square, this filter takes the minimum for each component. This grows lighter areas and shrinks darker ones.
3 Solution Evaluation

3.1 Assumptions

For the most part, the filters in the program do not make any assumptions about the image contents and dimensions. The only requirement is that the images have positive widths and heights, and that all the pixel values are valid.

For the neighborhood filters, which are Smooth, DeNoise, Erode, and Dilate, it is expected that the user-inputted neighborhood radius is smaller than the width and height of the input image. If it does not meet this condition, while the program will not crash or throw an error, the output may not be what the user expects.

3.2 Imperfect Cases

For some filters, depending on the properties of the image or the user input, undesirable or imperfect results can occur. Although these will not cause the program to fail or throw an error, the output may be unexpected.

BlackAndWhite: I assumed that this filter is intended to turn a picture into grayscale. Others implementations may convert the picture to strictly only use black and white. To convert a pixel into grayscale, I treated each component as having an equal contribution to brightness. Since the human eye responds more to green than to red or blue, weighting the components before taking the mean might have led to an improved result.

Shrink: If either the width or height is one, the program will output an invalid image. The dimension that was one in the input will be zero in the output. This is unavoidable as there is no way to shrink such an image, since shrinking it only on the larger dimension would lead to distortions.

Threshold: In this filter, if a component of a pixel is equal to the user-inputted threshold, it will be set to fully on. While this is a valid solution, other implementations may set equal components to fully off. This may lead to some discrepancies in the output when the same image is processed using different implementations.
Smooth, DeNoise, Erode, Dilate (neighborhood filters): For this class of filters, pixels near the edge are treated differently from pixels in the center. This is because the “neighborhood” used in these filters is a square with a user-inputted size, and must be truncated so it does not extend beyond the edge of the image. This means that pixels near the edge, where the neighborhood must be clipped, have a smaller sample, especially for filters like Smooth and DeNoise. This results in the edges being slightly less processed than the centers.

3.3 Extra Credit

In addition to the required filters, I implemented four additional filters: Smooth, DeNoise, Erode, and Dilate. These filters take into account a pixel’s surrounding pixels to calculate its final value. This allows for much more interesting effects since larger, multi-pixel features in the image can be modified.

Using some of these filters, I created this image: [http://imgur.com/6OFB9i1](http://imgur.com/6OFB9i1) (original: [http://7-themes.com/data_images/out/68/7003841-green-landscape-wallpaper-23020.jpg](http://7-themes.com/data_images/out/68/7003841-green-landscape-wallpaper-23020.jpg)). My goal was to make the image look like a watercolor painting. I found the landscape online and ran Dilate with radius two and then ran DeNoise twice, with radius two, then five.


4 Testing

4.1 White-box Testing

One way to test the filters is to construct small image arrays such that when processed through a filter, they will have a predictable output. These tests, however, rely on the tester knowing the inner workings of the implementations. My tests were written by selecting an input array, manually calculating the expected output, and comparing it to the program generated output. Although most of my tests would work with other implementations, some of the filters mentioned in 3.2 may fail. For this reason, I did not write automated tests for the neighborhood filters, since any errors would be visually apparent.

4.2 Black-box Testing

A more effective way to test the filters is by giving it some input images as observing the outputs qualitatively. This method checks the functionality and not the exact correctness of the situation,
and so is more useful in this case. Most of these tests would work with other implementations as well.

**NoRed, NoGreen, NoBlue:** The best image to test these effects is 14.jpg from the provided test images. For each filter, the color it removes will be changed to black, and the white square will become its inverse color. In this test, it is very easy to see if the filters are working properly.

**RedOnly, GreenOnly, BlueOnly:** The best image to test these effects is 14.jpg from the provided test images. For each filter, all square except the white and the color it keeps will be turned to black. The filters color will remain unchanged, and the white square will turn to this color. This also provides a reliable way to test these filters.

**BlackAndWhite:** The most useful image to test this filter is 13.jpg from the provided test images. Since all the colors in a row, except white, have equal brightness, after the filter is run, all the bars should look alike. The white bar will remain unchanged.

**VerticalReflect, HorizontalReflect:** The best image to test these filters is 11.jpg from the provided test images. Any asymmetric image would work, but a face is best since it is easy to recognize changes. These filters can be tested by seeing if Dr. Lin is flipped, or facing the other way, after applying the effects.

**Grow, Shrink:** The best image to test these filters is 11.jpg from the provided test images. Any detailed image would work, but a face is best since it is easy to recognize the same details across images. These filters can simply be visually checked to confirm that they do not distort the image and preserve most details.

**Threshold:** The most useful image to test this filter is 21.jpg from the provided test images. If the threshold is set at 127, each section of the image will become mostly one color. The center will become white and the dark gaps will become black. It is easy to then count the number of colors to verify that it is 8.

**Smooth, DeNoise:** The most useful image to test this filter is 6.jpg from the provided test images, although any detailed image would work. If the radius is set to 5, the Smooth filter will blur and blend the colors. The DeNoise filter has a similar output, but picks the dominant color from each section instead of blurring them all together. If the radius is increased, more detail will be lost.

**Erode, Dilate:** To test these filters, I created an image (http://imgur.com/k621BF9) for which the effects would be obvious. When Erode is run, the black circle becomes bigger, while the white circle shrinks. For Dilate, the black circle shrinks, while the white one becomes bigger. This matches the expected outputs of these filters.