IMAGE PROCESSING

A Web App written in JavaScript that combines various image processing techniques and methods built with React.

Luke Ellul

Table of Contents

Contents

Introduction

This project uses JavaScript and the power of the browser to manipulate images using the techniques provided in class. ReactJS is used to design the UI. The README file located in the root directory of the project contains installation instructions and some other technicalities.

The main directory is divided into the following directories:

- **src:** contains the source code of the web app
- **build**: contains static files (JavaScript, HTML and CSS files) that can be served to the user
- **public:** contains assets used by the web app such as images used
- **node_modules**: contains all the dependencies that are used by the project (this directory appears after you issue **npm install**)

The **src** directory contains a file called **App.js**. This is like the **main** file in a traditional app. It combines all the components together of the web app to show you the page you see when you run the web app.

Now, the way the app works is you have a **state** that is shared among all components. This state is a simple JavaScript object. The state holds various information about the web app but most importantly, it holds information about the image data of the original image and the image data of the processed image. By image data, we mean an array representing the pixels of an image. So that's the state.

The various components representing different parts of the web app, such as buttons, sliders, etc…they dispatch actions that update the **state**. For example, the grayscale button gets the image data of the original image from the state, processes that image data, and updates the state with the processed data. The actual original image and the processed image are (magically) connected to the state and they are each connected to their image data property. So, when the processed image data in the state is updated with new image data, the processed image changes accordingly (because it's connected to the image data). Same with the histogram chart that keeps updating itself with every modification to the processed image. The histogram is connected to the image data in the **state** and therefore, when the state is updated the histogram is updated too!

This magic stuff is actually handled in the **redux** folder in the **src** directory. The **ui** directory contains **React** components that render the UI. For example, **PointProcessing.js** in the **ui** folder contains components that render the buttons that do point processing modifications on the image.

Techniques

All the techniques listed in the Assessment spec except for Line/Edge detection and Morphology were implemented in the web app. The reason why I left the last two out is because I ran out of time.

These techniques are stored in the **Functions** folder located in the **src** directory. Each technique is represented as a function that takes an array of image pixel values as an argument, along with other parameters, and returns a processed array of image pixels. These functions are then used by the components mentioned above.

It's important to note that an image in this context is not represented as a 2-dimensional array but rather as a 1-dimensional array of numbers. Each number represents an RGB value (or alpha). So 1 pixel is represented by 4 numbers, the first number represents the **R**ed intensity, the second number the **G**reen intensity, the third number the **B**lue intensity and the fourth number represents the alpha value of that pixel. For example, this array represents an image with 4 pixels:

[23,45,67,255,23,87,34,255,34,89,09,255,23,56,57,255]

To process this 1-d array like a 2-d array the following technique is used:

Let's say we want to read the blue component's value of a pixel at column 200, row 50 of a 2-d array of an image. We get that value out of the 1-d array as follows:

blueComponent = **imagedata**[((**50** * (**width** * 4)) + (**200** * 4)) + **2**];

Where **imagedata** is the 1-d array and **width** is the width of the image.

Point Processing

Functions that perform point processing operations are located in **PointProcessing.js** in the **PointProcessing** directory.

Invert

The invert function just loops over each pixel component and subtracts it from 255.

Grayscale

I used the average method for the grayscale function. The function loops over each pixel and for each pixel it calculates the average of all its color components, then it sets each color component of that pixel to the calculated average value.

Binarize

The binarize function accepts a threshold and checks of each component of every pixel is above the threshold. If it is, then it sets the component to 255, else it sets it to 0.

Otsu Binarize

The Otsu Binarize function performs automatic binarization by finding the threshold automatically and applying the above **Binarize** function. To find this threshold we have to find the Otsu Level and this is done using the **otsuLevel** function that takes the histogram counts of an image as input and outputs the Otsu level. The code for finding the Otsu Level was converted to JavaScript from Matlab code found on this Wikipedia article about the Otsu Method: https://en.wikipedia.org/wiki/Otsu%27s_method.

The histogram counts of an image are found using the **histogramGrayLevel** function that is found in **HistogramProcessing.js** under the **Histogram** directory.

Log Transform

Log Transform is performed by iterating over each pixel and setting it’s components’ values to a constant **c**, multiplied by the log of the pixel intensity plus 1.

Power Transform

Power Transform is applied in a similar fashion to Log Transform but instead, we multiply a constant **c** with the intensity of a pixel raised to a parameter **y**.

Histogram Processing

Image Histogram

To show the histogram of an image the function **histogramGrayLevel** is used which is located in the file **HistogramProcessing.js** under the directory **Histogram**. The **histogramGrayLevel** function takes image data array as input and outputs an array with 255 elements. Each element represents a pixel intensity (or pixel gray level) and each element in the array is set to the number of pixels in the image that contain that pixel intensity (or pixel gray level).

Histogram Equalization

The **histogramEqualization** function calculates the histogram equalization of an image and takes an image data array as an argument. To calculate the histogram equalization, we first find the sum of the histogram counts by applying the **histogramGrayLevel** function to the image data argument. We then get the intensity levels of the image by finding the unique values in the image data array. We then iterate through all the intensity levels.

For each intensity level, we get the sum of the histogram counts minus the histogram counts that come after the current intensity level. We then iterate though each pixel in the image and if the value of the current pixel is the same as the current intensity, we set the pixel's value to the calculated sum divided by the sum of all the histogram counts.

Convolution

The functions that deal with convolution are located in the file **Convolution.js**. The function **convolution** accepts the image width, the image height, a mask, and the image data array as its arguments. The mask is a 2-d array of numbers. The function starts by reversing the order of the arrays in the mask and reversing the numbers of each array in the mask in order to flip the mask, since we're applying convolution.

We then apply convolution over the image data array using the iterative method in the spatial domain. Basically, the function iterates over every pixel in the array and sets an accumulator to 0. We then multiply the pixel's value with the value at the center of the mask and add this to the accumulator. Then, each neighboring pixel of the current pixel (by neighboring I mean pixels that are in range to the current pixel as the mask) is multiplied with the equivalent mask's value and each value is added to the accumulator. After the whole area of the mask is processed, the value of the accumulator is assigned to the current pixel.

The **BoxFilterMask** function and the **GaussianMask** functions generated a box filter mask and a gaussian mask respectively.

Average Filter

The **averageFilter** function applies convolution on the image data array by calling the **convolution** function with the mask generated by calling the **BoxFilterMask** function.

Gaussian Filter

The **gaussianFilter** function applies convolution on the image data array by calling the **convolution** function with the mask generated by the **GaussianMask** function.

Fourier Transform

The functions in the file **Fourier.js**, also located in the **Convolution** directory deal with processing an image in the frequency domain. The function **fourier** returns the frequencies that represent an image. These functions are only there for providing a basis in working with image processing in the frequency domain in JavaScript. They aren't used by the web app.

Noise Addition and Removal

Functions that deal with noise addition and removal are located in **noise.js** which is located under the **Noise** directory.

Adding Salt and Pepper Noise

The function **addSaltandPepper** adds salt and pepper noise to an image and accepts an image data array as an argument. The function iterates over each pixel in the image. For each pixel, a random number between 0 and 255 is generated. If this number is say, less than 3, then we set the current pixel's value to 0. If the random number is say, larger than 251, we set the current pixel's value to 255. If the random number falls between 3 and 251, we leave the pixel value unchanged.

Filtering

The **filter** function applies filtering on the image in order to remove noise. The function accepts a function that takes an array as argument and returns an element in that array. The rest of the arguments include the image width, the image height and the image data. The function iterates over each pixel. For each pixel a set of neighboring pixel values is created. This set is then sorted and the function which is given as an argument is called with the sorted array. The function returns an element from the sorted array. For example, the median filter function returns an element in the middle of the sorted array. The value returned by the function is then set to the current pixel.

The **median** filter calls the **filter** function with a function that returns the middle value in an array. The **maximum** filter calls the **filter** function with a function that returns the last item in an array. The **minimum** filter calls the **filter** function with a function that returns the first item in an array.

Segmentation on Binarized Images

The functions that deal with image segmentation are located in the file **Segmentation.js** located in the directory **Segmentation**. To segment an image, the connected components algorithm is used, specifically the **Two-Pass** algorithm. The algorithm works in the following way:

First, we apply binarization on the image. This can be accomplished by using Otsu's method. Each pixel in the image is then assigned a value of 0 or 1 based on whether it's a background pixel or a foreground pixel.

Then we set a variable holding the label counter.

We also create a set that keeps track of label counters and their equivalent labels.

We iterate over each pixel.

If the neighboring pixels have different pixel values, we set the current label counter to the current pixel and we increment the label counter.

If the neighboring pixels have the same value as that of the current pixel, we set the label of the current pixel to the minimum label of the neighboring pixels. We also record the equivalent labels by updating the set of equivalent labels.

If a neighboring pixel has a value the same as the current pixel, we set the label of the current pixel to the label of the neighboring pixel.

After the first pass, we iterate over each pixel once more and set each label to the minimum of its equivalent labels.

The **TwoPass** function in **Segmentation.js** does the above steps and also assigns colors to the different components.

The algorithm explained above was inspired from this Wikipedia article: https://en.wikipedia.org/wiki/Connected-component_labeling