Part 1 Written Report

Alwin Hollebrandse

This program requires the user to provide two command line args. The first is file location that holds all images of interest. The second argument is instruction file. This code will create a new results folder in the current directory that eill have a folder per image that will house all operation outputs. See the “Results” section for example output.

Image File location Arg:

This is the first argument of the program. It should contain a path to where all of the images are located. In thoery, any images that Java’s Buffered Image supports would be supported by my code. An example image can found in the results section under “original”.

Instruction File arg:

This is the second command line argument for the program. It should be a txt file. It is used to build the operation arraylist (discussed under Implentation/General). Accepted Strings that will cause action are (ignoring case): SingleColor, Quantization, SaltAndPepper, Gaussian, LinearFilter, MedianFilter, Histogram, HistogramEqualization. These can be found in the all\_Instructions.txt file.

Implentation:

General:

This program does the following: first it validates the users command line args. It then parses the instructions.txt file into an arraylist that holds the found Strings that match those and explained in the Instrcution File arg section. This code then loops through all of the images found in the first command line argument. It then performs the user requested operations in the following order (skipping operations not requested): SingleColor, Quantization, SaltAndPepper, Gaussian, LinearFilter, MedianFilter, Histogram, HistogramEqualization. If there is an error within one of these operations, the operation will be skipped (or if it severe enough, the whole image will be skipped). If the operation was not requested, the code will check if there are any output files related to said operation already in the results folder for the current image. If there is, it will be deleted. This was implanted such that there will be no confusion when checking results. If the operation was requested, it will be completed and added to ./results/{{current image’s name}}/{{operation output}}. Each of these operations works on the outcome of the previous specified operation.

Each operation loops through all pixels in the image, which is performed in parallel. This was accomplished by creating a ParallelMatrix File. This file creates how ever many threads the system can support and divides the image pixel matrix into segments of equal size. This was done by looping through all the width pixels but jumping by MAX\_THREADS and by looping through all height pixels per width iteration. Each thread starts at an incremented x value. Each thread then calls a code lambda defined in the respective operation’s java file for the current width and height value of the pixel matrix.

The end of the program prints out time metrics collected through the code’s run.

SingleColor:

This code is found under SingleColorScale.java. It takes in the original image and the desired color of the image. Accepted color values are (ignoring case): gray, red, green, and blue. The code calls ParallelMatrix with the following lambda: depending on what color the param was, remove unneeded channels (though the image remains in RBG format). It should be noted that research showed that gray pixels should not all be equal. The following formula was used for gray conversion: (int) (c.getRed() \* 0.299) + (int) (c.getGreen() \* 0.587) + (int) (c.getBlue() \* 0.114);. The output is the new image using only the specified channel. The output is saved as {{user specified color}}.jpg

Quantization:

This code (Quantization.java) takes in the current image, scale, and color (color is used to compute the mean squared error). This code then computes the valid pixel ranges for the given scale. This is done by performing 256/scale, and adding a running count of that number until 256 is met (forming quantizationArray). This code then passes the following lambda to ParallelMatrix: for the provided pixel loop through quantizationArray starting at position 1. For each used number, check if the current color value is less than the current quantizationArray vlue. If it is, set the color to the value of quantizationArray[i-1]. A use of Booleans prevents this from happening more than once per color. The output is saved as Quantization.jpg.

SaltAndPepper:

This section is found under NoiseAdder.java. It accepts the following parameters: current image, noiseType, randomThreshold, mean, and sigma. After validating each param, this code provides this lambda: for the given pixel, check which type of noise is being added. If it is “saltAndPepper”, generate a random number. If this is less than or greater than the random threshold, generate another number. If this number is greater than 0.5, set the pixel color to black. Otherwise set it white. The output is saved as saltAndPepper.jpg.

Gaussian

This section is found under NoiseAdder.java. It accepts the following parameters: current image, noiseType, randomThreshold, mean, and sigma. After validating each param, this code provides this lambda: for the given pixel, check which type of noise is being added. If it is “gaussian”, generate a random gaussian value using the provided mean and sigma. This value is added to each relevant channel and normalized. The output is saved as gaussian.jpg.

LinearFilter

This section is found under Filter.java. This filter methods crops of borders of images that do not have a large enough border for an edge pixel. It accepts the following parameters: current image, filterType, filterWidth, filterHeight, weights, and scalar. After validating each param, this code provides this lambda: for the given pixel, check which type of filter is being applied. If it is “linear”, take the user’s filter mask size and compute the average of each pixel in that filter (where each pixel is first multiplied by the corresponding weight value). The center pixel of this filter is set to that average value. Note that each channel of the pixel is computed independently. The output of this method is saved as linear.jpg.

MedianFilter

This section is found under Filter.java. This filter methods crops of borders of images that do not have a large enough border for an edge pixel. It accepts the following parameters: current image, filterType, filterWidth, filterHeight, weights, and scalar. After validating each param, this code provides this lambda: for the given pixel, check which type of filter is being applied. If it is “median”, take the user’s filter mask size and adds each pixel to a list by the corresponding weight amount of times. The median of this list is then computed. That becomes the value of the filter’s center pixel. The output of this method is saved as median.jpg.

Histogram

This section is found under GraphHistogram.java. It accepts the following parameter: current image. In order to create a histogram, each pixel is looked at (not in parallel since the required semaphore would slow the process to slower than a sequential version) and the gray color value ((int) (c.getRed() \* 0.299) + (int) (c.getGreen() \* 0.587) + (int) (c.getBlue() \* 0.114)) is added to an int array of size 256 in the correct position. This histogram is then saved to histogram.png.

HistogramEqualization

This section is found under HistogramFunctions.java. It accepts the following parameter: current image and histogram. In order to equalize the histogram, a look up table is created by creating an array of size 256. For each value of this array, the value is the histogram value at the same position plus the previous look up value. This lookup table is given to the lambda which does the following: for each pixel, get the current gray value and use that as the key for the lookup table. That is the new value of the pixel. This code has two outputs: equalizedHistogram.png and equalizedImage.jpg.

Example Metrics

Final Metrics:  
Converting to a single color processing time for the entire batch (ms): 21437  
Quantization processing time for the entire batch (ms): 23641  
total meanSquaredError: 2147483647  
Adding salt and pepper noise processing time for the entire batch (ms): 25852  
Adding gaussian noise processing time for the entire batch (ms): 35861  
Linear filter processing time for the entire batch (ms): 32867  
Median filter processing time for the entire batch (ms): 47520  
Histogram creation processing time for the entire batch (ms): 3319  
Equalized histogram creation processing time for the entire batch (ms): 16758  
Total RunTime (without image exporting): 207255  
Real run time: 307348

Results:

Original

A picture containing wall, indoor, food, white

Description automatically generated

singleColor

A picture containing wall, white, indoor, photo

Description automatically generated

Quantization

A picture containing wall, indoor, building

Description automatically generated

salt and Pepper

A picture containing ground, nature, wall, building

Description automatically generated

Gaussian

A picture containing wall, building, white, animal

Description automatically generated

LinearFilter

A picture containing wall, white, indoor, building

Description automatically generated

MedianFilter

A picture containing wall, indoor, white, building

Description automatically generated

Histogram

A screenshot of a social media post

Description automatically generated

HistogramEqualization

A picture containing outdoor, grass, tree, nature

Description automatically generated

equalizeHistogram

A screenshot of a social media post

Description automatically generated