CmpE 150 Introduction to Computing, Fall 2019

Assignment 3 – Due: 05/01/2020, 23.59

In this assignment, you will write a Java program to make some calculations on 3D pixel arrays of some images. We will use the **ppm** image format for reading and writing image files. You will read a **ppm** image file into a 3D array. Firstly, you will write the exact 3D array to another **ppm** file. Second, you will calculate a ***color-channel based average*** and write a black-and-white version of the input image as **ppm**. Third, you will perform ***convolution*** operation using a 2D array (called "filter") on the input image and write the result as another **ppm** image. Finally, you will check the values of the neighboring pixels to see if they are within a given **range** and modify these pixels to be equal if they are in the same range. This is called **color quantization**. Then you will write the quantized image to a final **ppm** file.

Each of these steps will be explained in detail. Please read this description very carefully before starting to write any code. Your program will run using arguments and a special integer argument called "mode" will be the indicator of what your program will perform.

PPM Image Format

The PPM (or Portable Pix Map) image format is encoded in human-readable text. Below is a brief synopsis of the format, which will aid you in being able to load and save a PPM image.

Sample ppm file:

```
P3
4 4
255
0   0   0    100 0   0        0   0   0     255    0 255
0   0   0     0 255 175       0   0   0       0     0   0
0   0   0     0   0   0       0 15 175      0     0   0
255 0 255   0   0   0         0   0   0     255   255 255
```

Image Header

The first three lines are defined as the image **header**. They provide an overview of the contents of the image. PPM headers consist of four entries, which will be defined using the example:

```
P3
4 4
255
```

- P3 defines the **image format**; that is, what type of PPM (full color, ASCII encoding) image this is. For this assignment, it will *always be P3*.
- Next comes the number of **columns** and the number of **rows** in the image. The example is a 4 pixel by 4 pixel image

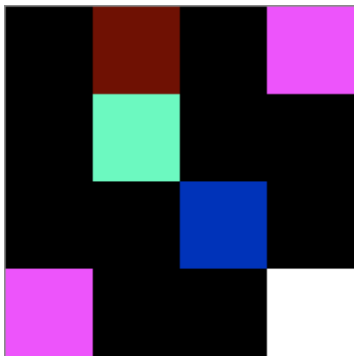
- Finally, we have the **maximum color value**. This defines the scale of values possible for the color intensities. This can be any value, but a common value is 255, meaning the **red**, **green**, and **blue** values for a pixel can range from a value of 0 up to 255. We will again always use 255 for this assignment.

Image Body

The **image body** contains the actual picture information as a series of RGB (red, green, and blue) values. Each pixel of the image is a tiny, colored square. In the PPM file, each pixel is defined by a triplet of values representing how much red, green, and blue (RGB) are present. So, the first pixel, which has a value of **0 0 0**, is black, and the last pixel, **255 255 255**, is white. By varying the levels of the RGB values you can come up with any color in between.

Note that color values must be separated by a space, but after that any additional whitespace is ignored by the image viewer. In the sample **ppm** above we used additional whitespace ('\t') to format the pixel values so that it is easy for a human to understand.

The example image above would look something like this:



Keep in mind, each square is one pixel, so the real thing is much smaller (the rendered image was blown up by 5000%).

How to View PPM Files

To view PPM files, you can use the open-source and free software **IrfanView,** which you can download from **here**.

Part 1: Read the contents of the PPM file into a 3D array

You are given an example 128 x 128 image file named "**input.ppm"**. You will read the contents of this file to a 3D integer array.

Create a 3D integer array (representing the RGB values of the pixels in the image) using the header information of the PPM file (number of columns and number of rows). Fill the 3D array with the values given in the body part of the PPM file. Remember that the body of the PPM file gives information about the RGB values of pixels. First entry of the example above is **0 0 0,** meaning that the pixel on the top left has **0** value for color **red, 0** value for color **green,** and **0** value for color **blue**, this states that the pixel on the top left is **black**. In the 3D array that you create, Third dimension

will represent the **color channel**. See Figure 1 for a nice visual representation of 3D arrays for images.

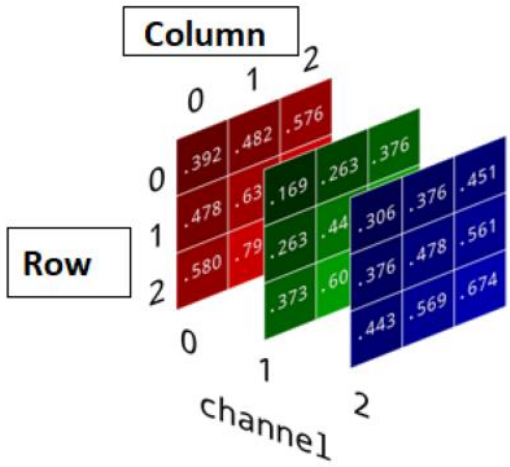


Figure 1: A visual representation of an image in a 3D array. (Values are only illustrative; they are not the real pixel values in the example image given to you.)

We can see from Figure 1 that the third dimension represents the **color channel.** For example:

- pixelArray[0][0][0] represents the value of the **red channel** of the top left pixel.
- pixelArray[0][0][1] represents the value of the **green channel** of the top left pixel.
- pixelArray[0][0][2] represents the value of the **blue channel** of the top left pixel.

Therefore, the size of the 3D array that you create should be [rows][columns][3].

Part 2: Write the contents of the 3D pixel array to a PPM file

Your program will take the "mode" argument as 0, and an input PPM image file.

Argument1: 0

Argument2: input.ppm

After reading the PPM image into the 3D array, create a new file named "**output.ppm**", and write the pixel values to the new PPM file by following the rules of the PPM format. Insert a '\t' character between RGB triplets. After writing the pixel array to the PPM file, you should be able to double click and view the same image in IrfanView.

This part of the project gives you **20 points.**

Part 3: Calculate the color-channel average values and convert the colored image to black-and-white.

Your program will take the “mode” argument as 1, and an input PPM image file.

Argument1: 1

Argument2: input.ppm

In this part, you will calculate the average color values of each pixel using the three color channels (RGB). If you assign the calculated average value to all three color channels for all the pixels in the image, then you obtain the black-and-white (grayscale) version of the image. For example:

- If [0][0][0] is 11, [0][0][1] is 13, and [0][0][2] is 12, average of these values is 12.
- You should assign all three channels the same average value 12. Meaning that:
    - [0][0][0] = 12
    - [0][0][1] = 12
    - [0][0][2] = 12

If you perform this for all the pixel values in the image, you get the black-and-white version. **You should perform integer division** since the pixel values in the images must be integers. Write the new black-and-white version to as another PPM image named “**black-and-white.ppm”**. The original image and your output image should be like below:

Original

Black-and-White

This part of the project gives you **20 points.**

Part 4: Apply convolution to the original image using an input 2D array as “filter”

In this part, you will apply **convolution** to the input image. Your program will take the “mode” argument as 2, an input PPM image and a 3x3 filter (kernel) as a text file.

Argument1: 2

Argument2: input.ppm

Argument3: filter.txt

Convolution is the process of adding each element of the image to its local neighbors, weighted by the filter (kernel). See Figure 2 for example steps of convolution calculation.

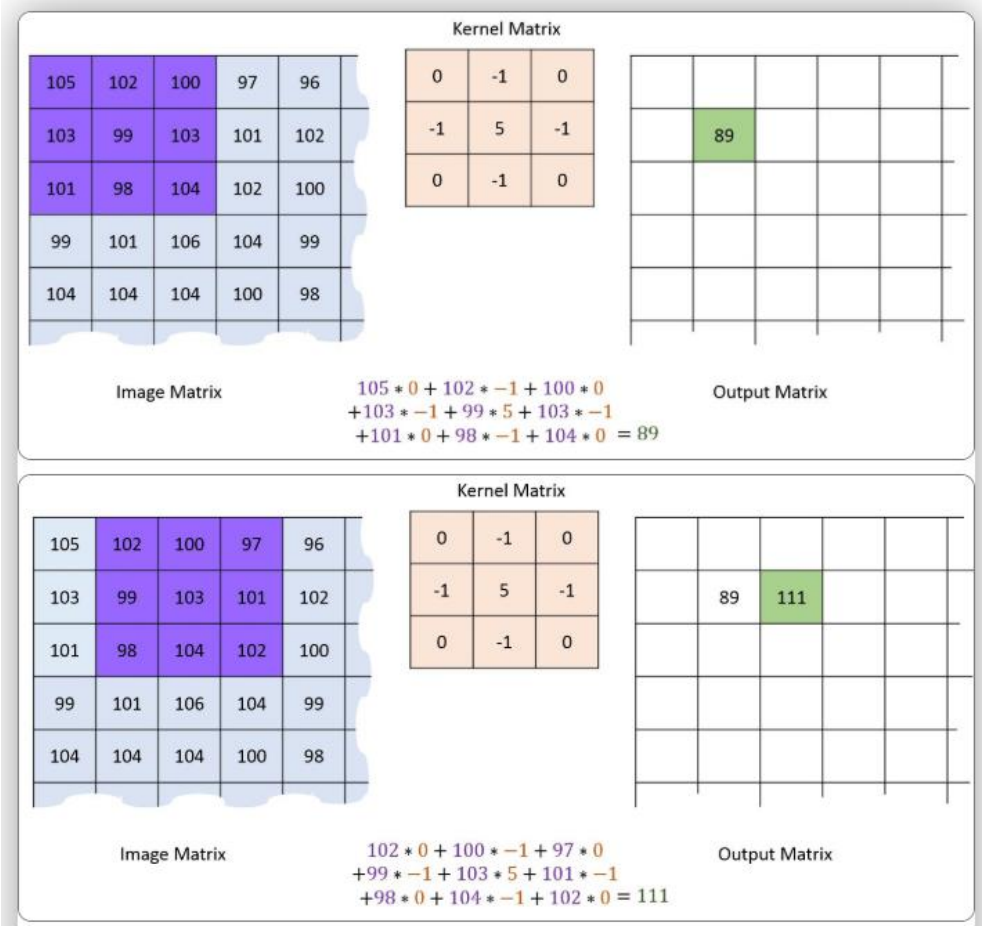


Figure 2: Two steps of an example convolution

In Figure 2, we have a 3x3 convolution filter (kernel), applied to an image. We see that we are taking the weighted sum of the pixel values and writing them to the "middle" pixel of where we are applying the filter. Notice that the output matrix will be smaller than the original image matrix. This is because we are losing the information on the top, bottom, leftmost and the rightmost edges of the image pixel array.

An example "filter.txt" file is given below:

```
3x3
1 0 -1
1 0 -1
1 0 -1
```

Dimensions of the filters are always odd numbers such as 3x3, 5x5, 7x7 etc. Note that filters reduce the size of the input image since we are losing some pixel values on the edges.

You will read the pixel RGB values of the input PPM file to a 3D array, and then read the filter from the "filter.txt" file to a 2D array. Then you will apply convolution to ***each color channel of the image separately.*** Finally, you will save the output file to "**convolution.ppm".**

If we convolve the original image with the 3x3 vertical filter, we get a version of the image with the vertical edges emphasized:

1	0	-1
1	0	-1
1	0	-1

Vertical

If we convolve the image with the 3x3 horizontal filter, we get a version with the horizontal edges emphasized:

1	1	1
0	0	0
-1	-1	-1

Horizontal

These filters are known and special filters in image processing, they are called **vertical** and **horizontal *edge detection*** filters. As you can see, applying these operations to pixels gives us the ability to look at images from different filters.

This part of the project gives you **30 points.**

Part 5: Perform color quantization by looking at the neighboring pixels

In this final part, you will traverse the pixels of an input image one by one, check if the values of a pixel's neighbors are within a given range, and make their values equal if they are in the same value range.

Your program will take the "mode" argument as 3, an input PPM image file, and an integer that represents the **range** that the program will look at.

Argument1: 3

Argument2: input.ppm

Argument3: <an integer value representing **range**>

After reading the input PPM image to a 3D array, you will traverse the whole 3D pixel array element by element, and look at the values of the neighbors of each element. A ***color region*** is defined as the "neighboring elements that have the pixel value within the same **range**". If the neighbors of an element have the value within the same range as the element itself, then you will make those pixel values equal to the value of the element.

To be more precise, you will look at six neighbors of each element (be careful at the boundaries of the image).

If x, y, and z are the coordinates of a 3D array, then you should check for the following neighbors:

- x+1 y z
- x-1 y z
- x y+1 z
- x y-1 z
- x y z+1
- x y z-1

If any of these neighbors, (and also the neighbors of these neighbors, and so on…) are within the range of the pixel value with the element contained in x, y, z coordinate, then all of these elements' pixel values should be the same as the value in x, y, z. (As you can see, this is a perfect scenario for recursion!).

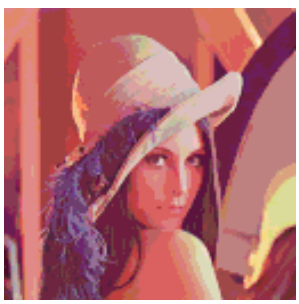
- Assume that the **range** value is 5, If pixelArray[5][5][1] = 13, then we will look at its neighbors and check if their values are between 13 – 5 and 13 + 5. If the value is within the range, then the pixel value of the neighbor will also be 13. If it is not in the range, then we will skip that neighbor.
- In short, if a pixel has the value **N,** then we will check if its neighbors' values are within the range (**N – range)** and (**N + range).** If they are, then we will make their values also **N.**
- **You should start from color channel 0 (red), go row by row, and then to channel 1 (green), go row by row, and then finally to channel 2 (blue), go row by row and finish.**

In accordance with the range value, intensity of quantization will change. If we increase the range, we will lose more information and the image quality will drop significantly. Check the examples below:


Original


Range = 8


Range = 15


Range = 25

After performing the operation, save the quantized image as **"quantized.ppm".**

You should use the "range" value from 1 to 25, for higher values you may get a **stack overflow** error! If you get stack overflow errors for values between 1 and 25, just implement your program for **range** values that you do not get stack overflow errors.

This part of the project gives you **30 points.**

Implementation Details:

1. Your program should have at least two static methods in addition to your main method. Try to write your program as modular as possible (without overusing methods).

2. You can and should make use of all the subjects you have learned in class such as while loops, conditional statements, arrays, recursion etc.

3. Your program will work using arguments. You will use an integer value indicating the working **mode** of your program. Mode 0 is Part 2, Mode 1 is Part 3, Mode 2 is Part 4 and Mode 3 is Part 5. Mode argument is an integer value from 0 to 3 and it will be the first argument of your program.

4. To test and run your program with arguments, click Run -> Run Configurations -> Arguments -> Program Arguments. Enter the necessary arguments here, and then click run at the bottom of the window.

5. Please use the **exact same input and output file names given in this description file**, or else you may lose points while grading. Your working directory should be the project folder in Eclipse. Read and write files directly by name such as "input.txt" and not "src/input.txt" or "C:/users/PC/input.txt"

6. We will give you the expected output files. You can use a tool like diff checker to check if your outputs are correct. It will not be feasible to manually check if your outputs are correct with 128 x 128 image files.

7. Image sizes and convolution filter sizes may change so do not make any assumptions on the sizes. (We can test your program with an image or filter with different size, not always 128 x 128 images or 3 x 3 filters).

Submission: You will submit a project report and your code over Moodle. Project report should consist of five sections. These are:

1. Problem Description: In this section, you should describe the problem in your own words.

2. Problem Solution: In this section, you should specify the concepts (methods, loops, conditionals etc.) that you use in your program. Explain each one (i.e. why you need it, what you accomplish by using it, so on.) broadly.

3. Implementation: This section will include your whole code with comments. You need to pay attention to indentation in order to improve readability.

- Do not forget to explain each variable that you use (i.e. int count = 0; // count is the number of items).
- Before each method, specify what the method does (i.e. /* This method … */)

4. Output of the program: A screenshot of your program input and output should be put in this section. One example run for each functionality of your implementation is enough.

5. Conclusion: You should evaluate your work here. State whether you have solved the problem correctly. If not, state what is missing, what could have been improved, and so on.

6. Your .java file should be named with your initials and your student number together (e.g., OS2013800027). If you have Turkish characters as your initials, please change them to non- Turkish. (Example: ÖS2013800027 should be OS2013800027) You will submit these over Moodle as a single zip file where the file name is your student number. Your zip file should consist of your .java file and your report in .doc or .pdf format. Do not use any Turkish characters in your code, class/variable names, or .java file names.

Partial Submission: If you cannot complete the assignment, you should still submit your code as well as your report. Try to implement as much as you can. In your report, explain which parts you were able to complete and which parts you were not. Partial credit will be given depending on your output. If you can produce the correct output for multiple tests but not all, you will still get partial credit.

Late Submission: Late submission is possible with penalty = -10*(daysLate)^2.

Evaluation Procedure: We will evaluate your code based on the program output for many test cases. You can create test cases on your own and compare the outputs.

Good luck.

Input-Output Examples using the simple PPM file given above:

Input arguments: 0 input.ppm

Output ppm file:

```
P3
4 4
255
0  0  0   100 0  0       0  0  0    255   0 255
0  0  0    0 255 175     0  0  0     0    0  0
0  0  0    0  0  0       0 15 175    0    0  0
255 0 255  0  0  0       0  0  0    255  255 255
```

Input arguments: 1 input.ppm

Output ppm file:

```
P3
4 4
255
0 0 0      33 33 33    0 0 0       170 170 170
0 0 0      143 143 143       0 0 0       0 0 0
0 0 0      0 0 0       63 63 63    0 0 0
170 170 170      0 0 0       0 0 0       255 255 255
```

Input arguments: 2 input.ppm filter.txt

Input "filter.txt" file:

```
3x3
1 0 -1
1 0 -1
1 0 -1
```

Output ppm file:

```
P3
2 2
255
0 0 0    85 85 85
111 111 111    0 0 0
```

Another input "filter.txt" file:

```
3x3
1 1 1
0 0 0
-1 -1 -1
```

Output ppm file:

```
P3
2 2
255
33 33 33    111 111 111
85 85 85    0 0 0
```

Additional examples will be given.