Exercise #3 – Programming Language Control, I/O, and Types

DUE: As specified on Canvas

Please thoroughly read Programming Language Concepts Chapters 6 & 7 and bring any questions you have to class.

This Exercise can be completed in any programming language of your choosing including: Java, C/C++, C#, Python, Lisp, etc. and any libraries of your choice available in your chosen programming language (e.g. OpenCV, boost, …). Please note the language, libraries, and the compiler/interpreter you used and tools used to debug your code and include any build files and basic instructions on how to build and run your code. You may use Windows, the SE workstation, VB-Linux or the PRClab to develop and test your code. Use a new language you want to learn (to make this exercise a more interesting challenge) or use one you are most comfortable with to make it more straight-forward. For image processing, some languages like MATLAB and Octave as well as libraries such as OpenCV for C/C++ and Python offer direct support for image file types such as PPM, JPEG, and PNG, which can be an advantage. As made evident by the example C code, the binary file manipulation is non-trivial even for uncompressed PPM. Either way, pay close attention to the efficiency and readability of your solution. If you are new to image processing and transformation of images, you may find these background notes useful along with this basic brighten and contrast example code.

If you wish to work with a partner, the partner you plan to complete your final comparative or custom programming language research project with, please do so. However, in this case, I will expect you to attempt a solution in two unique programming languages rather than one.

Exercise #3 Requirements:

1) [10 points] Read the DSP programmer's guide chapter 24 PSF section and summarize the value of the PSF (Point Spread Function) for image processing. Comment on how hard or easy this is to implement in the language of your choice and whether efficiency is one of your concerns.

2) [20 points] Implement the PSF transform as described in the programming language of your choice. ***Describe the language you are using*** – e.g. imperative procedural, pure functional, object oriented and why you chose it. Any reason is acceptable, but should include examples of advantages the language has with sound logic to back up your assertions. ***Provide at least three advantages of your choice (this is my best and only well-known programming language is a good example)***. If possible, time your image transform for transform of a small statically initialized buffer of 1000 graymap pixels (this can be done easily in Linux using the ***time*** command – in Windows, this may not be possible, so for windows, you can

provide an argument for why the code is easy to understand or efficient in terms of total instruction count for example).

3) [20 points] In the language of your choice, write a simple routine that will load a PGM (Portable Graymap) into a two dimensional array, computes the negative (255 – pixel value at each location) and writes it back out to another PGM. Test with Irfanview or GIMP to make sure your negative transform works. You may also want to review the input files or resultant transformed output files with a binary editor such as Frhed.

4) [40 points] Apply a sharpen PSF (Point Spread Function) to your two dimensional array from step #2 above using the same language with values as follows:

```
#define K (4.0) // Play with this to see how much sharpening you can get

double PSF[9] = {-K/8.0, -K/8.0, -K/8.0, -K/8.0, K+1.0, -K/8.0, -K/8.0, -
K/8.0, -K/8.0};
```

Apply the PSF to each pixel of interest in your 2D array so that the new pixel, stored in a transformed array is a sum of products with each PSF multiplier applied to each of the eight nearest neighbors and K+1.0, the 4th element in the PSF applied to the pixel of interest. Note that the transformed array is a new image array, but the PSF is applied to each and every pixel in the original array. Pixels should be an 8-bit unsigned type that provides a range of 0 to 255. E.g. here's a pseudo C code example of the basic block to which you must add iteration control (example C code is here, but note this uses a 1D array). You may find this paper and code to be helpful.

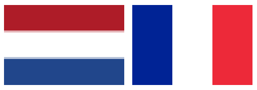
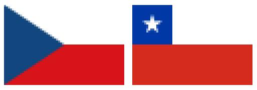
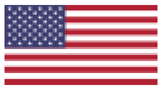
```
unsigned char newpixel=0;

// PSF application to pixel of interest using neighbors from 2D Image
newpixel += PSF[0]*(double)Image[i-1][j-1];
newpixel += PSF[1]*(double)Image[i-1][j];
newpixel += PSF[2]*(double)Image[i-1][j+1];
newpixel += PSF[3]*(double)Image[i][j-1];
newpixel += PSF[4]*(double)Image[i][j]; // note this is K+1.0 multiplier
newpixel += PSF[5]*(double)Image[i][j+1];
newpixel += PSF[6]*(double)Image[i+1][j-1];
newpixel += PSF[7]*(double)Image[i+1][j];
newpixel += PSF[8]*(double)Image[i+1][j+1];

// check to see if you have overflowed the original 8-bit unsigned type
if(temp<0.0) temp=0.0;
if(temp>255.0) temp=255.0;

// set output image to transformed value
newImage[i][j]=(unsigned char)newpixel;
```

If you use Java or C/C++, consider use of a conditional expression in at least one location in your code as well as looping and logical constructs and comment on whether the use of a conditional expression simplifies or obfuscates your code if you use it. Try something new like the conditional expression or dividing the image and using concurrency if you are not using a language new to you (e.g. sharpen_grid.c) – this is not a requirement, but a way to challenge yourself and I'll be looking for good design for efficiency.

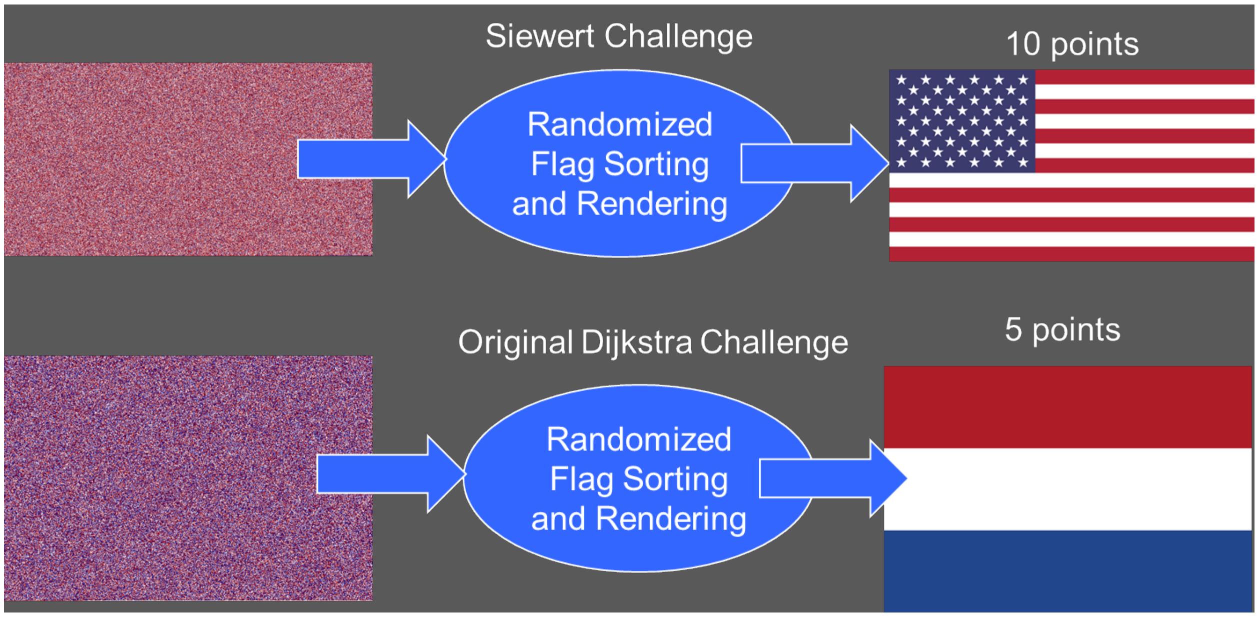
Make sure that your solution can handle the boundary conditions at edges and corners by replicating the edge pixels (i.e. create your array two rows bigger than the image and two columns wider and replicate the value below, above, left, or right of the extended pixel range – e.g. your 400x300 array becomes 402x302) so that your PSF can be applied to all pixels in the image. Note that in my example I simply ignored the first row, last row, first column and last column in my 1D array implementation to deal with this boundary condition. Run your code on this 120K pixel PGM (Aspect Ratio is 4:3, so 400 by 300 pixels) – Cactus-120kpixel.pgm

5) [10 points] Now run your program from step 4 on the 12 megapixel PGM and report your time to transform (4000 by 3000 pixels) – Cactus-12mpixel.pgm using Linux ***time*** or suitable equivalent in Windows (e.g. try using POSIX clock_gettime or gettimeofday). Does your solution scale with resolution as you would expect?

6) [**up to 50 points extra credit – due with Exercise #3 or #4**] – Reference Flags
Edsger Dijkstra gave his students a now famous problem to sort an equal number of randomly arranged red, white and blue balls (consider them pixels in an image) in an array into the Dutch flag. His point was that this seemingly trivial problem is not as easy to code correctly and efficiently as it first appears. Since posing the problem, many solutions have been suggested including some very efficient methods from Dijkstra himself. Many of these can be found in a variety of programming languages on Rosetta Code here. For extra credit, create a new algorithm that sorts an equal number (or properly proportioned number) of red, white, blue elements into another national flag (e.g the Dutch, French, Chilean and Czech Republic are all 2:3 aspect ratio red, white and blue flags whereas the US flag is red, white, blue but 10:19 aspect ratio – as noted here). I have provided reference flags for you to use. Your code should ***start with an array of the proper aspect ratio with randomized but equal numbers of red, white, blue elements for Dutch and French*** which it organizes (sorts) into an array that depicts the flag of interest as best possible using an equal number of each color as a starting condition. ***Note that your code must start with randomly ordered but invariant ratios of red, white and blue pixels which must be sorted to render each flag [5 pts for each equal color flag, 10 pts for each unequal for a total of up to 50 points].***

 (2:3 equal)  (2:3 unequal)  (10:19 ?)

Using a Rosetta code example (as you wish) adapt a PL implementation of your choice to produce the Dutch or French flag so you can display with Irfanview [5 points each]. For more extra credit, modify your implementation so it can render any or all of the flags provided such that your code takes the RAND version of the flag and produces the correctly rendered version. For example, for the American flag, such that the transformation data flow below is implemented. You may find the Wikipedia PPM reference useful as well.


Siewert Challenge
10 points
Randomized Flag Sorting and Rendering
Original Dijkstra Challenge
5 points
Randomized Flag Sorting and Rendering

Overall, provide a well-documented professional report of your findings, output, and tests so that it is easy for a colleague (or instructor) to understand what you've done. Include any C/C++ source code you write (or modify) and Makefiles needed to build your code. I will look at your report first, so it must be well written and clearly address each problem providing clear and concise responses to receive credit.

In this class, you'll be expected to consult the Linux and development tool manual pages and to do some reading and research on your own, but you are welcome to use whatever development and debug tools you are most comfortable with.

Upload all code and your report completed using MS Word or as a PDF to Blackboard and include all source code (ideally example output should be integrated into the report directly, but if not, clearly label in the report and by filename if test and example output is not pasted directly into the report).

Grading Checklist

[10 points] Reading of DSP Programmer's Guide on PSF convolution:

[5 pts] Summary____

[5 pts] Comments on implementation____

[20 points] PSF Transform function:

[5 pts] Programming language selection ______________________________________

[10 pts] PSF code____

[5 pts] PSF transform efficiency analysis_____________________________________

[20 points] Selection of programming language for image file manipulation and processing.

a. [10 pts] Programming language selection rationale______________________
b. [10 pts] PGM file load and write-back

   ____

[50 points] PSF image transformation:

a. [30 pts] Code complete and compiles _________________________________

b. [20 pts] Code works well on 12k pixel image __________________________

[20 points extra credit] PSF image transformation on large image:

a) [5 pts] Code modified for high resolution_____________________________________
b) [10 pts] Code is efficient____
c) [5 pts] Code is optimized ____