Elec4622 Lab 2, 2022 T2

David Taubman

May 31, 2022

1 Introduction

In this laboratory, you will become familiar with the implementation of FIR filters in two dimensions. While a basic framework is provided for you, there is much for your to modify yourself. It is imperative that you fully understand what is going on to the point where you are confident that you could implement everything yourself from scratch.

2 FIR Filters in Floating Point Arithmetic

In this exercise, you will explore convolution and boundary extension, using floating point arithmetic. You will also get an easy (really) introduction to C++ classes.

1. Start by downloading the "filtering_example.zip" file from the class web-site; unzip and compile the example program.

   - Notice that the executable file gets written to the location "\elec4622\bin". As noted during the last laboratory session, it is best for you to write all your executable files to a common "bin" directory and include this in the executable path[1]. You should change this to one of "\elec4622\slot1\bin" or "\elec4622\slot2\bin", as explained in Week 1.
   - Make sure you can run the program on the command line and from the debugger, processing the "pens_rgb.bmp" image from Lab1. As explained in Week 1, you should put the image into "\elec4622\slot1\data" or "\elec4622\slot2\data." Resist the temptation to put the images into the "bin" directory with the executables. If your execution path is set up properly, you can change directory in the command shell to the "data" directory and run everything from there.
   - Study the "image_comps.h" header file to understand how the "my_image_comp" class works. Be sure to read the notes appearing below its definition.

[1] You can do this by editing/creating the "path" environment variable under the "User Variables" section in Control Panel. The simplest way to find this is to click on the Windows launchpad icon at the bottom left or your Windows 10 desktop, start typing "environment ..." and select the option for setting user environment variables, rather than system environment variables.

2. Modify the program to implement a filter with the following PSF's:

$$h_1 = A \cdot \begin{pmatrix} 0 & 1/3 & 1/2 & 1/3 & 0 \\ 1/3 & 1/2 & 1 & 0.5 & 1/3 \\ 1/2 & 1 & 1 & 1 & 1/2 \\ 1/3 & 1/2 & 1 & 1/2 & 1/3 \\ 0 & 1/3 & 1/2 & 1/3 & 0 \end{pmatrix}$$

$$h_2 = B \cdot \begin{pmatrix} 1/4 & 1/2 & 1/2 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1 & 1 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1 & 1 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 1/4 & 1/2 & 1/2 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$h_3 = B \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/4 & 1/2 & 1/2 & 1/4 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 1 & 1 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 1 & 1 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1/4 & 1/2 & 1/2 & 1/4 \end{pmatrix}$$

where A and B are chosen to ensure that the filters have a DC gain of 1. Save the filtered images obtained using h_2 and h_3 and use the Media Interface "crop_n_shuffle" module to display the difference between the two images. Make sure you understand what you are looking at.

3. Construct an unsharp masking filter, using h_1 as the underlying low-pass filter. Specifically, create a filter with PSF

$$h_{\text{unsharp}}[\mathbf{n}] = \delta[\mathbf{n}] + \alpha(\delta[\mathbf{n}] - h_1[\mathbf{n}])$$

where α controls the degree of sharpening. See what happens as you adjust α – you might want to make it a third command-line argument to the program.

   - At this point you will discover that you need to pay more attention to converting floats back into bytes when writing the result to the output BMP file.

4. Make sure you know what boundary extension method is currently being implemented by the program.

   - See what happens to the unsharp masking result when you use zero-padding as the boundary extension method.
   - Try modifying the boundary extension policy to use symmetric extension.

3 Vertical Filtering with Vectorization

Preliminary note on vectorization: Vector instructions are very important to achieving high throughput in multi-media signal processing on modern CPUs. In this exercise we consider only SSE2-family instructions that work with 128-bit vectors, but AVX2-family instructions are much more powerful, processing 256 bits at once, and AVX-512 has also started to appear in recent years. Due to their importance, modern compilers have evolved to be able to quite a decent job of auto-vectorizing simple codebases and this option should be enabled by default for "Release" builds. While this is great, it prevents you from learning about the benefits of vectorization yourself, since you will want to start on simple examples that the auto-vectorization tools can handle well. For this reason, you should disable auto-vectorization in the build settings, when performing timing comparisons here.

1. Start by downloading the "vertical_filtering.zip" file from the class web-site, unzip and compile the example program.

   - Make sure you understand the differences between the "my_aligned_image_comp" class and the "my_image_comp" class from the previous exercise.
   - Compile the program first to use the direct vertical filtering implementation and then to use the vectorized implementation. You do this simply by changing the "#ifdef" statement in the "main()" function. Check that both versions produce the same output.

2. Modify the program slightly so as to perform the filtering step many times over, so you can reliably measure the execution speed using something like:

   ```
   timer vertical_filtering pens_rgb.bmp out.bmp
   ```

   For the most meaningful results, compile the program in the "Release" mode – use "Build → Configuration Manager" to switch between the "Debug" and "Release" modes.

3. Try modifying the program to add a horizontal filtering stage – i.e., horizontally filter the vertically filtered image, using the same underlying 1D filter.

   - To do this, you will need to create at least a single line buffer to hold intermediate results – as suggested in Chapter 2 of the lecture notes.
   - You should start by implementing the horizontal filtering step only using the direct method. If time permits, you could consider a vectorized implementation of this step as well.