Convolution Matrix Simple Write-Up

Std=C++14

The reason for a simple write-up is I hope that the code can partly speak for itself and that the given task isn't complex from a SWE perspective. Most of my explanation will talk about the reasoning behind the design choices I made.

Matrix.cpp

Matrix object is simply created for use by the convolutional filter. It can access data stored in its indices from the (row, column) notation and have a matrix interface for the user.

Why implement my own Matrix Function

The main reason was I needed a way to:

1. Use a 1D array as the actual data structure, map the index via (row*columns) + column
    a. The other option would be to instantiate vector<vector<int>> matrix which would not be ideal from a memory access perspective: To have Vectors of vectors would mean that the first index of the outermost vector would point to some area in memory that is disconnected to the second index which points to another vector in memory. This discontiguous area of memory means that there would be more jumps which slows down the memory read time. Memory Reads are an expensive operation and we try to minimize jumps as much as possible.
2. Have access to edge case checks
    a. Implementing your own edge case checking abstracts the need away from the user to constantly explicitly check and allows for greater versatility within the convolution matrix.
3. Template Class
    a. Matrix is a template class so it can hold any data type it needs to, this is so that It can store both the "**unsigned char**" and "**short int**" data type, but can be used for more.
4. Matrix print to console
    a. Implement a print matrix function that can print the matrix in the (rows,column) form for visualization and debugging
5. Fill with random numbers
    a. To fill with random numbers of any data type, the fillRand() function allows for that flexibility.
6. Heap Storage
    a. Per instructions, we need Dx and Dy and M to be separate matrixes in memory
7. Min/Max
    a. For-loop to calculate min/max of the matrix for any data type, separately calculated outside of the convolution function.

Driver.cpp

This is the where the main function resides, in the code we check off the instructions and make our edge-case assumptions clear and implement methods that help with user input and the **convolution** function

Convolution Function

How I implemented it and what my assumptions were.

1. The filter is small, it's only [-1, 0, 1]. Which is Right.data – Left.data or Up.data – Down.data.
    a. This means that for every index, we would need to calculate the left and right/ up and down. A simple for loop can calculate this, and I chose to go with the for-loop instead of a DFT because the filter is so small.
2. Edge-Case, Wrap Around.
    a. I think of the relationship in a picture to be spatial-frequency, which means that close together items are on average "closer" together in value. So when I calculate for the horizontal-axis for example I first wanted to reflect the value on the other side of the index. If I was on (5,0) which means that (5,1) is my 1* and my (5, -1) is my -1* on the filter. If I chose to make val(5, -1) = val(5,1) then at (5,0) I would end with '0' and that is not what I want, 0's across the edge of the matrix. So this I why I chose the wrap-around method which allows the matrix to fold itself on the respective axis and my (5, -1) is now mapped to (5, end_of_column). This yields a more accurate resultant change in the resultant convolution matrix.

How to compile and run

Copy and paste this into your linux environment, I'll also provide an already compiled executable file named "convolution.exe".

1. g++ -std=c++14 Driver.cpp
2. ./a.out