Advanced Programming 2023/24 – Assessment 3 – Group Project

Image Filters, Projections and Slices

Group Name:	**Xavier Initialization**	
Student Name	**GitHub username**	**Tasks worked on**
Daniel Seal	edsml-ds423	Image class, 2D Image blur
Yongwen Chen	acse-yc3321	Grayscale, Edge detection
Zeqi Li	acse-zl123	Histogram equalisation, Thresholding, 2D UI
Jing-Han Huang	edsml-jh123	Brightness, Salt and Pepper Noise
Wenbo Yu	acse-wy1223	3D data volume, filters, projections, and slices, UI
Ning Guan	edsml-ng323	Edge detection

1 Algorithms Explanation

1.1 2D Image Filter:

The *Image* class is the main driver for all the 2D filter operations. *Image* stores a pointer to an image's pixels in a 1D buffer in memory and creates a *data* attribute on instantiation, storing the pixels in a (h, w, c) accessible tensor. All the 2D filters have an *apply* method that takes in a copy to an *Image* object and performs in-place pixel updates on the raw 1D buffer. Each of the 2D filters inherits from an abstract *Filter2D* class that has a virtual *apply* method, that any derived class must override, forcing consistency across all our filters.

1.1.1 Grayscale:

For each raw pixel, a new grayscale value is calculated using Y = 0.2126R + 0.7152G + 0.0722B. The greyscale apply will only work for RGB (3 channel) images. If an image input is only 1 channel, it is assumed to already be grayscale and will return the original image with no updates.

1.1.2 Brightness:

The Brightness iterates through each pixel, adjusting its brightness by adding or subtracting a specified value within the range of -255 to 255. For each pixel, a new value is calculated using Y = current pixel + brightness. This Y is clamped between 0 and 255 to prevent over or underflow.

1.1.3 Histogram equalisation:

For grayscale images, the filter directly alters the image data based on the cumulative distribution function (CDF) of pixel intensities to spread out the most frequent intensity values, thereby improving contrast. For colour images, the algorithm first decides whether to work in the HSV or HSL colour space. In HSV mode, it equalizes the V (value) channel to adjust contrast while preserving colour hues and saturation levels. In HSL mode, it focuses on the L (lightness) channel for contrast adjustment, again preserving hues and saturation. The algorithm computes a new CDF specific to the V or L channel, maps the original values to this equalized histogram, and then converts the modified channel back into RGB.

1.1.4 Thresholding:

The Thresholding filter converts an image into a binary image based on a specified threshold. For grayscale images, pixels below the threshold are set to 0 (black), and those above are set to 255 (white). For colour images, the method first checks if they should be processed in HSV or HSL colour space. It then converts the colour to the specified space, applies the threshold based on the V (value) channel in HSV or the L (lightness) channel in HSL, and converts pixels below the threshold to 0 (black) and those above to 255 (white).

1.1.5 Salt and Pepper Noise:

The Salt and Pepper Noise algorithm introduces random black and white speckles into an image based on a user-defined percentage. It utilises the Fisher-Yates shuffle algorithm to generate random coordinates to inject (randomly) black (0) or white (255) pixel values.

1.1.6 Blur:

The 2D kernels are all pre-computed to avoid O (k x k) overhead during the convolve iteration. During each *apply*, an *Image* object is copied, and the *data* attribute is overwritten with the required padding. Padding values are all set to 0.

Having access to the raw image 1D buffer and the modified *data* attribute is essential to avoid the neighbourhoods of pixels changing as we in-place update the raw image in memory. For odd-sized kernels, the pixel is centred. For even-sized kernels, the pixel is always placed at the top right.

1.1.6.1.1 Median blur:

A pixel's neighbourhood is multiplied by a kernel of 1s (including itself). The subsequent kernel values are then sorted using a quick sort algorithm and the midpoint is taken as the median value. This is the new value for the pixel.

1.1.6.1.2 Box blur:

A pixel's neighbourhood is multiplied by a kernel of 1s (including itself). The subsequent kernel values are then averaged to get the pixel's new value.

1.1.6.1.3 Gaussian blur:

A (k x k) kernel of gaussian weights it precomputed analytically, using the 2D gaussian function. Precomputing the kernel weights is possible because the gaussian function is only dependent on a pixel's position in relation to its neighbours which remains the same for any position in the image (uniform stride). The neighbourhood, containing the weight x pixel values, is then summed to get the new pixel value.

1.1.7 Edge detection filters:

Edge detection is implemented for Sobel, Prewitt, Scharr, and Roberts' Cross algorithms. The user-interface (UI) will suggest converting an image to grayscale and applying gaussian or box blurring prior to edge detection to reduce noise that could affect edge detection accuracy.

1.2 3D Data Volume

The *Volume* class is a driver for three-dimensional (3D) volume data, encapsulating the dimensions (width, height, slices) of the volume and the data itself, with each slice of the volume stored as a pointer vector to an array of unsigned characters.

1.2.1 3D Gaussian Blur

GaussianFilter3D applies a normalised Gaussian kernel to 3D data to improve edge-preserving noise reduction and computational efficiency through kernel separation and precomputation.

1.2.2 3D Median Blur

The 3D Median Blur function reduces noise in volumetric data by replacing each voxel with the median of its neighbourhood, effectively preserving edges while smoothing the volume.

1.2.1 Maximum intensity projection (MIP):

The MIP method iterates through specified slices of a 3D volume, comparing each voxel's value to find the maximum value across the Z-axis. This technique is essential for visualizing high-intensity structures within volumetric data.

1.2.2 Minimum intensity projection (MinIP):

MinIP operates similarly to MIP but focuses on identifying the minimum voxel values across the Z-axis within a specified slice range. This approach is particularly useful for highlighting low-intensity areas.

1.2.3 Average intensity projection (AIP):

The AIP method computes the mean voxel value across the Z-axis for each X-Y position, providing a comprehensive overview of the volumetric data's average intensity. This technique is valuable for general visualization purposes, where detailed structure representation is required.

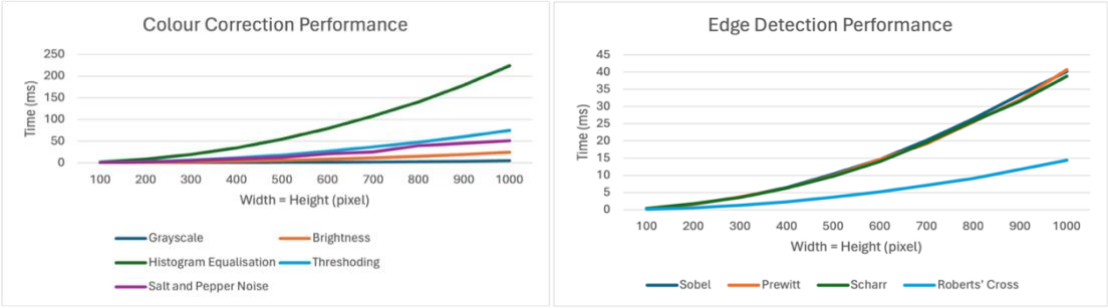
1.2.4 Slicing:

The *Slice* class implements YZ and XZ plane slicing of 3D volumes, enabling visualization of cross-sectional views along specified axes. By adjusting indices for zero-based indexing and efficiently arranging pixel data, these

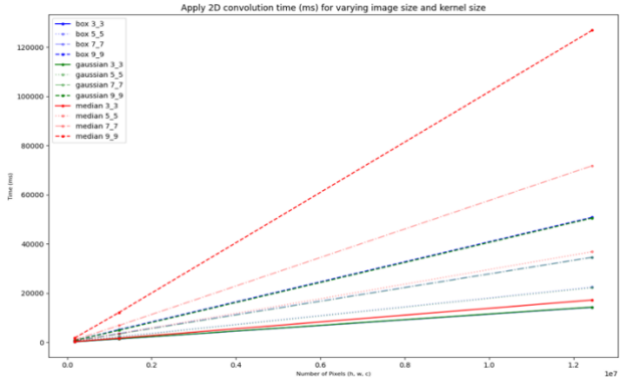
methods facilitate the extraction and saving of slice images, providing insights into the internal structure of volumetric data.

2 Performance Evaluation

2.1 Image size

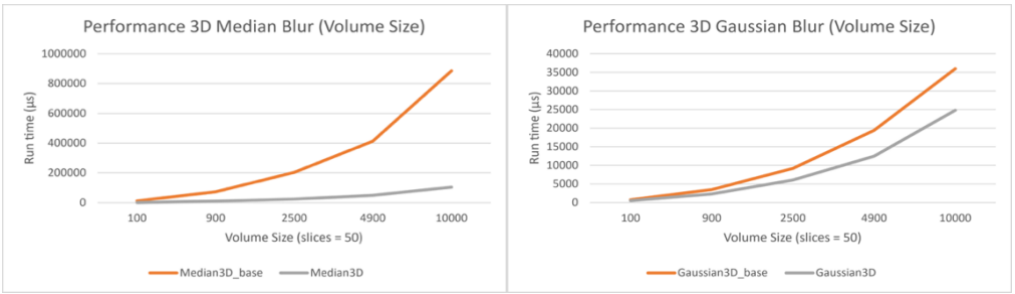


The line charts show the processing time for various image manipulations for different image sizes ranging from 100x100 to 1000x1000 pixels. For Colour Correction (left), histogram equalisation has the highest processing time due to pixel intensity computation overhead, while grayscale, having the least overhead, is the fastest. For edge detection (right), the expected increase in processing time with increases in image size is observed. As expected, the 3x3 kernel algorithms, Sobel, Prewitt, and Scharr algorithms all have ~ equivalent relationships and are slower than the Roberts' Cross algorithm that uses a smaller 2x2 kernel.



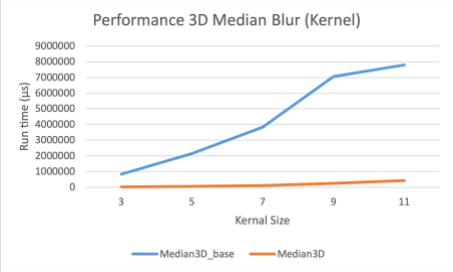
To test the 2D blurring, 3 different images (*tienshan*, *stinkbug* and *gracehopper*) were tested for each technique (quicksort implemented for median blur), see *results/ results_2D_Conv_num_pixels.csv*. As expected, (due to precomputed kernels), the time convergence increases linearly for each blurring technique as the number of pixels increases. As expected, the additional sorting overhead in the median blur does cause the convergence time to diverge to slower times as the kernel size increases.

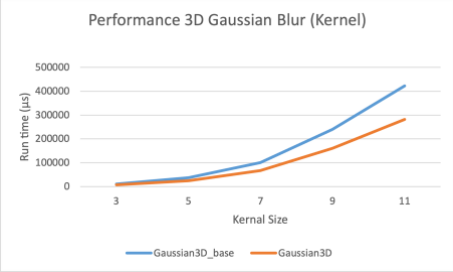
2.2 Volume size

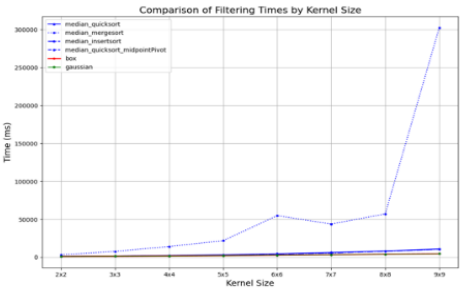


As expected, the runtime increases as volume increases. Before optimising (base), the time taken was a bottleneck for larger volumes, especially for median blur. After optimising (orange) we achieved notable speed up.

2.3 Kernel size







GaussianFilter3D: The optimised code outperforms its base counterpart as it employs a precomputed kernel and optimized convolution. These optimisations prove crucial with increasing volume and kernel sizes.

MedianFilter3D: The optimised version uses a histogram-based approach for median computation, which outperforms Median3D_base in terms of efficiency and avoids higher sorting complexity. In addition, the reuse of individual histogram arrays minimises memory allocation and enhances cache performance, resulting in significant reductions in runtime, especially as volume and core sizes increase, as evidenced by the performance graphs.

2DFilters: Merge, insertion and quick sort were all tested for varying kernel sizes with median blur. Quick sort was the quickest when implemented using the median of three algorithm to find the pivot. Quicksort generally outperforms insertion sort due to its average time complexity of O(n log n), compared to the average time complexity of insertion sort which is O(n^2). Quicksort with median of three is very comparable to box and gaussian even with the additional sorting overhead. This comparable time complexity was expected due to precomputing the kernels before iterating over the image pixels.

3 Potential Improvements/Changes

3.1 Histogram Equalisation

- parallel processing could expedite histogram and CDF calculation.
- Integrating SIMD (Single Instruction, Multiple Data) instructions could accelerate pixel-level operations, making conversions between colour models and the equalisation process faster.
- pre-calculating some values or using lookup tables for repetitive calculations within RGB-HSV/HSL conversions might further enhance performance.

3.2 Convolution

- 2D and 3D convolutions are independent operations that could be threaded. The convolved pixel values could then be set on the raw image buffer sequentially to avoid race conditions.

3.3 Class Design

- The convolution code is currently separate. Particularly, blurring and edge detection in 2D. The *Kernel* class could be extended to handle the edge detection kernels. Extending the *Kernel* class would also ensure that all convolution operations are done correctly with padding to avoid any incorrect calculations at the edges.
- There could also be a base *Convolution* class from which a *Convolution2D* and *Convolution3D* could be derived.

3.4 Memory

- For 3D volumes, we are currently reading all the data into memory. For scaling, it would be preferrable to load the data in batches, perform operations over a batch and then free up the memory allocation before repeating until all batches are processed. For larger volume datasets, this would avoid overloading the memory.