Advanced Programming 2023/24 – Assessment 3 – Group Project

Image Filters, Projections and Slices

Group Name:	Radix Sort	
Student Name	**GitHub username**	**Tasks worked on**
Benjamin Duncan	edsml-bd1023	3D orthographic projections, slicing in different planes, ensuring compatibility between windows and mac systems, expanding utility of 3D interface, runtime analysis, report
Boyang Hu	edsml-bh223	2D Gaussian blur, Gaussian blur test, report
Chawk Chamoun	edsml-cc8915	2D Median and Box blur, Meidan and Box blur test, report
Mingsheng Cai	acse-sc4623	3D filters, 3D orthographic projections, 3D slice, all 3D tests, integrating and refactoring project structure, implement IFilter2D and IFilter3D interface,3D orthographic projections algorithm optimisation, 3D runtime analysis, 2D filter padding optimization, expanding and refactoring all 2D functions tests, bash scripts and cmakelist files and executable files, doxygen and latex manual pdf, readme files, publish library documentation online, set up github action and release, report
Moyu Zhang	acse-mz223	Five 2D Colour correction and simple pre-pixel modifiers, expanding utility of 2D interface, integrating 2D code, 2D runtime analysis in modifiers and edge detection, refactor 2D edge detection and Median & Box filter tests, doxygen, readme files, test executable, report
Ryan Benney	acse-rgb123	2D edge detection, edge detection test, report

1 Algorithms Explanation

1.1 2D Image Filters

1.1.1 Grayscale:

The grayscale conversion first retrieves the image's width, height, and number of channels, and then iteratively applies the grayscale conversion formula (0.2126*R + 0.7152*G + 0.0722*B, where R, G and B are the red, green and blue channel values of each pixel) to update each value. The converted grayscale value is stored back in the image data.

1.1.2 Brightness:

Brightness adjustment function uses the brightness variable to adjust the brightness, which is set in the PixelFilter class constructor based on user input. The algorithm iterates over each pixel of the image, adds the brightness value to each RGB channel, and ensures the result is within the range of 0 to 255.

1.1.3 Histogram equalisation:

Histogram equalization function converts RGB into HSL or HSV, uses the brightness values to calculate the histogram and cumulative distribution function, and then adjust the pixel values through histogram equalization. In this process, calculateHistogramAndCDF and applyCDF helper functions are used to calculate the histogram, CDF, and update pixel values.

1.1.4 Thresholding:

Thresholding function uses a threshold value (defined in the class constructor) to convert the image into a binary image. Pixels with grey-values below the threshold are set to 0 (black), while all others are set to 255 (white). This process is particularly effective for grayscale versions of the image or thresholding based on specific colour spaces (such as HSL or HSV).

1.1.5 Salt and pepper noise:

Salt and pepper noise algorithm uses a percentage variable (defined in the class constructor) to control the density of the added noise. The rand() function is used to randomly select a percentage of pixels and set them to either all black or all white.

1.1.6 Median blur:

The Median Filter operates by calculating the median intensity of surrounding pixels within a kernel, typically 3x3, and assigning this median value to the central pixel. This process is iterated across all image pixels, effectively reducing noise. The function getMedianKernel calculates the median, while applyMedianBlur updates each pixel. This method supports RGB and grayscale images and utilizes selection sort for median calculation due to restrictions on sorting algorithms. It offers ZeroPadding, EdgeReplication, and ReflectPadding options.

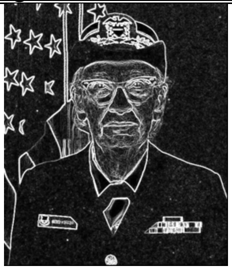
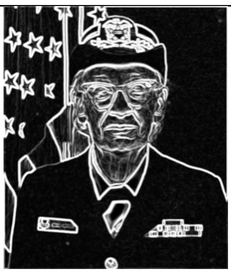
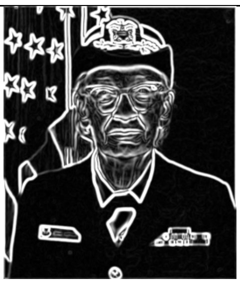
1.1.7 Box blur:

The Box Blur Filter averages the intensity values of pixels surrounding a central pixel within a kernel. The applyBoxBlur function accomplishes this without the need for sorting. test_box.cpp evaluates the blur effect by comparing original and blurred pixel values around specified coordinates. This method effectively smoothens the image by averaging nearby pixel values.

1.1.8 Gaussian blur:

The \`Gaussian2DFilter\` class generates a normalized Gaussian kernel based on specified kernel size and sigma, ensuring the kernel's elements sum to 1. The \`apply\` function then convolves kernel with each image pixel. It multiplies surrounding pixels by the kernel's corresponding elements, summing these values for the new pixel intensity. The degree of blurring is controlled by the ‘sigma’ parameter, which in this case has been set to 2. It also offers three padding methods: "ZeroPadding" fills boundaries with zeros, "EdgeReplication" extends the edge pixels outward, and "ReflectPadding" mirrors the image at boundaries for external pixels.

1.1.9 Edge detection filters:

		
No gaussian blur (lots of noise) Smaller contrast between edges	Gaussian kernel 3x3 sigma 2.0 (noise reduction) higher contrast	Gaussian kernel 7x7 sigma 2.0 (reduced edge sharpness)

The library implements four edge detections algorithms: Sobel, Prewitt, Scharr and Robert's Cross. The first three use 3x3 convolution kernels to identify vertical and horizontal brightness gradients, with Scharr's method emphasising central pixels for enhanced edge detection. Robert's Cross, in contrast, uses smaller 2x2 kernels which reduces computational load but increase noise sensitivity.
The edge detection filters were also applied after blurring, and it was found that Gaussian Blur was superior for pre-processing in edge detection. This is because, unlike Box Blur, it applies a weighted average that focuses on central pixels, preserving edge sharpness while still diminishing noise. However, the choice between blurring techniques and edge detectors must be tailored to the specific image characteristics to optimize the outcome.

1.2 3D Data Volume

1.2.1 3D Gaussian Blur:

In the main interface, the user can choose to apply a gaussian or median filter to the volume they have loaded in. If they choose Gaussian, a Gaussian3DFilter object (stored in Gaussian3DFilter.h) will be initialised with the user-inputted sigma and kernel size. It then calls the 'apply' function, which separately applies the Gaussian filter in the x, y and z directions. In each application, a weighted sum is computed using the Gaussian kernel, which is then accumulated, normalised to a 0-255 range and placed into the original data vector.

1.2.2 3D Median Blur:

The 3D median blur, like the 3D gaussian blur, begins with the initialisation of a Median3DFilter object using a user-inputted kernel size. In Median3DFilter.cpp, the apply function iterates over each voxel in the x, y and z dimensions, adds a precomputed offset using the precomputeNeighborhoodOffsets function, finds the median value within the neighbourhood and assigns the median value as the new voxel value.

1.2.3 Maximum/Minimum intensity projection:

The Projection class contains the key functionality to carry out an orthographic projection on an input 3D data volume. Each function takes in the width, height and depth of the stack which are inherited from the Volume object, and a pointer to the data volume. In each case, the function iterates over each pixel in the x-y plane, finds the index location of that pixel in the flattened 1D vector, and calculates the projection in the z direction.

1.2.4 Average intensity projection:

The library contains two implementations of the average intensity projection. The first method, the mean intensity projection, iterates over each pixel and adds its value to an unsigned long storing the sum of each set of pixels in the z direction. The sums are then divided by the depth of the 3D volume, returning a 2D image of mean values.
The second method, which calculates the median, iterates over each pixel and assigns each pixel in each z direction to a vector of char's, pixelValues. It then uses a conditional statement to find the middle index of the array and uses a helper function to sort and select the pixel value at that index.

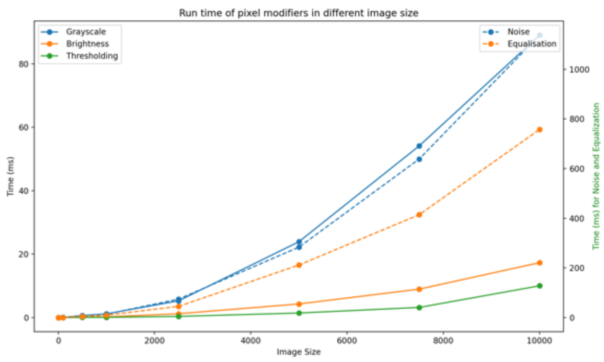
1.2.5 Slicing:

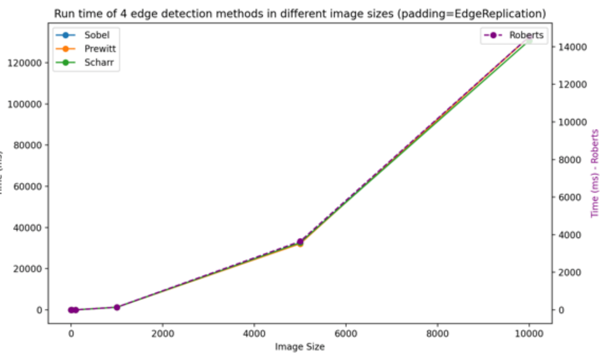
In the main 3D interface, the user can select the plane in which to reproject or extract a subset from the 3D volume. The projection is carried out using the volume.save function, which takes in an output directory to place the reprojected images, a plane (x-y, x-z or y-z) and, if the user specifies, an index corresponding to the ith image in that plane. The function calculates the number of images in the plane and uses the getPlaneSlice function from slice.cpp to locate the data array associated with the index. The stbi_write_png helper function is then used to write the slice array to a .png file in the user-specified directory.

2 Performance Evaluation

2.1 Image size

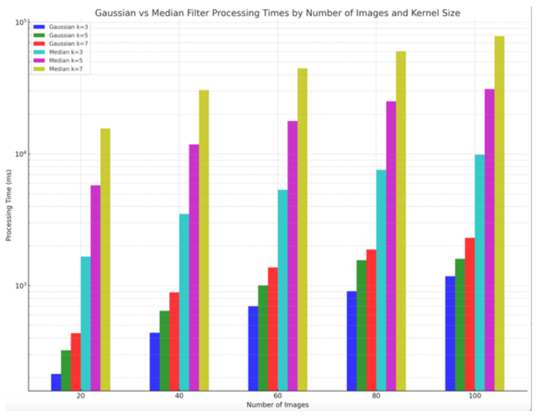
2.1.1 2D filter





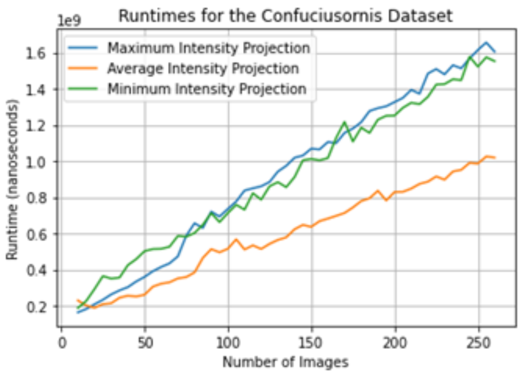
The left graph shows how runtime increases with image size for the pixel modifiers in the library. Salt and pepper noise and histogram equalisation both increase due to the increasing complexity of computing histograms and randomly selecting pixels for larger images.
The right graph shows how runtime increases with image size for the four edge detection methods with a fixed padding strategy. In all four methods, the runtime increases exponentially, with the Roberts method increasing at the slowest rate. This is particularly pronounced for larger image sizes, where the runtime for Roberts is in the order of 1e+4 compared to the 1.2e+5 order of the other methods. This is presumably due to the smaller kernel size of the Roberts which requires fewer complex calculations.

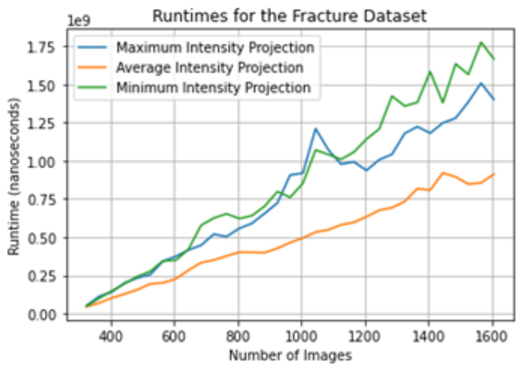
2.1.2 3D filter



The figure presents a comparison of processing times for 3D Gaussian filters and Median filters (compared with the graph in 2.1.4, The Median filter here has optimised with histogram-based approach)across various image sizes and kernel sizes. The processing time for the Gaussian Filter increases linearly with the image size, and an increase in kernel size also leads to a rise in processing time. Conversely, the processing time for the Median Filter escalates more significantly with image size, and the impact of kernel size enlargement on runtime is even more pronounced.

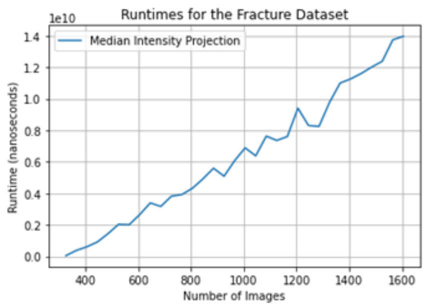
2.2 Volume size

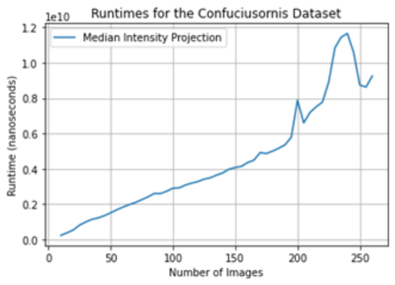




In both datasets, the maximum and minimum projections exhibit similar behaviour, but the minimum runtime records marginally slower runtime in the fracture dataset (1.4022 vs 1.6669 seconds for the full set of 1300 images). For the fracture dataset, the max projection records a gradient of approximately 1.053968 milliseconds per image. The mean projection is significantly more efficient than both, requiring just 0.91181 seconds for the full fracture dataset and recording a gradient of roughly 0.035277 milliseconds per image.

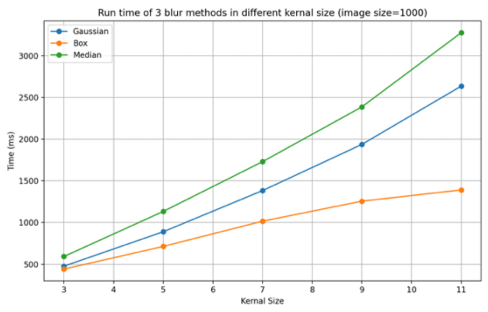
Runtimes for the median projection grow considerably slower as the number of images in the volume scales. For the fracture dataset, the algorithm recorded a rate of change of 108.66 milliseconds per image, reaching a runtime for 13.9597 seconds for the full image stack.





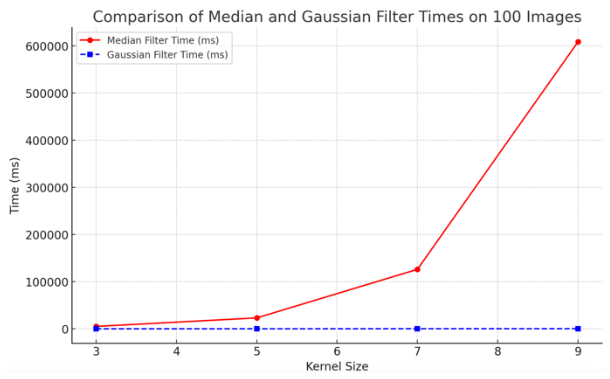
2.3 Kernel size

2.3.1 2D filter



The Median filter is the most computationally expensive, particularly with larger kernels. Its time complexity is not constant and increases significantly with kernel size, likely due to the need for sorting or partitioning within each kernel window. The Box filter is the most efficient than the Median filter. Its computational cost does increase with kernel size, but it's not as steep, indicating a better scaling with size. The Gaussian filter shows a similar trend to the Box filter. But it is more computationally intensive than the Box filter due to the weighted average calculation.

2.3.2 3D filter



The figure illustrates a comparative analysis of the execution time between 3D median and Gaussian filter using a dataset of 100 images from fracture dataset. It is evident from the data that the runtime of the median filter increases exponentially with the size of the kernel, whereas the Gaussian filter exhibits a linear increase in runtime with a significantly smaller magnitude. Specifically, the complexity of the median filter is nonlinear and computationally more intensive due to the necessity of employing sorting or selection algorithms within each kernel window to find the median value. As the kernel size increases, the number of elements each pixel has to process rises sharply, leading to a significant increase in computational load. On the other hand, the Gaussian filter is a linear operation. It involves convolving the image with a Gaussian kernel, which can be decomposed into a series of 1D convolutions, thereby maintaining computational efficiency as the kernel size increases.

3 Potential Improvements/Changes

3.1 Structure:

1) The class volume stores data via pointers and could be altered to store pointers to objects of the image class instead which would mitigate memory issues and align more closely with object-oriented design principles. 2) The structures of 2D and 3D filters are currently distinct, yet their core functionalities are similar, thus integration into a unified interface could be considered. 3) Orthographic projection and slicing functionalities are implemented as static methods; these could be enhanced by storing the data resulting from operations within objects. 4) The functionalities implemented by the basic test classes are relatively simplistic, and the interaction tests among some highly related subclasses require further development.

3.2 Algorithms:

1) The computational complexity of 2D filter operations is high, suggesting a search for more advanced algorithms as substitutes could be beneficial. 2) The current implementation of 2D convolution operations considers padding methods, whereas 3D convolution does not, which presents an opportunity for optimization. 3) The runtime performance of the 2D box blur is currently slow, and utilizing integral image techniques for implementation could offer improvements. 4) The computation speed of the 3D median filter is slow, which could be enhanced by employing histogram-based approaches. 5) The current implementation of orthographic projection is simply achieved through a for loop; exploring alternative methods for acceleration could be advantageous. 6) The project currently does not utilize multithreading; incorporating multithreading to accelerate processing could be considered.