Advanced Programming 2023/24 – Assessment 3 – Group Project

Image Filters, Projections and Slices

Group Name:	**Fibonacci**	
Student Name	**GitHub username**	**Tasks worked on**
Yibin Gao	edsml-kg23	Histogram Equalization, Threshold, Salt and Pepper Noise, Github version control, report
Ruihan He	edsml-rh323	Edge Detection (Prewitt, RobertsCross, Scharr, Sobel), report, readme
Yu Yin	acse-yy923	3D Projection, 2D gray scale, Github Action manager, report
Wenyi Yang	acse-wy1023	3D blur filters, Github To Do manager, report, optimisation
Sara	acse-sl4623	2D Blur (Box, Gaussian, Median), report, optimisation
Manawi	acse-mk1923	Volumne Slicing，2D Brightness Adjustment, report

1 Algorithms Explanation

1.1 2D Image Filters

1.1.1 Grayscale:

The grayscale algorithm converts color images to shades of gray by using 0.2126R + 0.7152G + 0.0722B. While this method effectively maintains luminance, it may result in the loss of crucial color information for some applications. Enhancements could involve adjusting weights based on image content or integrating techniques that retain color to improve visual details for specific purposes.

1.1.2 Brightness:

It processes each pixel, adjusting its value while ensuring it remains within the valid color range of 0-255. The auto-brightness feature calculates the image's average brightness and adjusts it to achieve a target of 128, aiming to balance images that are too dark or too light. However, this method may oversimplify by uniformly adjusting all pixels, potentially causing a loss of detail in shadows or highlights. To enhance results, dynamic range adjustment based on histogram analysis or localized brightness adjustments could be considered for preserving details.

1.1.3 Histogram equalisation:

The histogram equalization function enhances image contrast for both grayscale and color images. For grayscale images, it evens out pixel intensities by using the cumulative distribution function (CDF). In the case of color images, it converts the color space to HSV or HSL (based on hsv parameters), equalizes the luma (or value) channel, and then reverts back to the original color space. While this method enhances visibility and detail in images, it may lead to color distortion in high-contrast areas. To address this issue, adaptive histogram equalization can be used to apply adjustments locally rather than globally. This approach offers a more refined enhancement that maintains natural appearances and reduces noise amplification.

1.1.4 Thresholding:

The threshold function converts an image into a binary form using a specified threshold value. In grayscale images, each pixel is assigned either black (0) or white (255) based on whether its intensity falls below or above the threshold. For RGB images, the function initially transforms each pixel's color to HSV or HSL (depending on the `useHsv` flag), then applies the threshold to the brightness or luminance value, converting the pixel to black or white. This simplification process enhances specific features in the image while diminishing detail and color information.

1.1.5 Salt and pepper noise:

It randomly turns selected pixels completely black or white. This function works with both grayscale and RGB images, applying changes directly to pixel values in grayscale or across all color channels in RGB. The time complexity of these 5 algorithms is $O(n)$,where n represents the number of pixels in the image. Each pixel undergoes processing only once in a linear pass, resulting in a linear time complexity relative to the image size. They can enhance algorithm speed by incorporating parallel computing methods.

1.1.6 Median blur:

The median blur filter creates a copy of the image then iterates through each pixel (including the boundary pixels) and for each channel it stores the neighborhood of each pixel based on the precomputed offsets. If the neighborhood pixel falls out of the image bounds then the indices are adjusted to take the nearest pixel value within bounds. Once the neighborhood of the current pixel is built, the median of the neighborhood vector is computed using quickSelect sorting. QuickSelect sorting has, on average, a time complexity of $O(n)$, but its worst case is $O(n^2)$. In addition, iterating through every pixel already has time complexity $O(height * width)$. The filtering algorithm was optimized by precomputing the offsets before iterating through the pixel values.

1.1.7 Box blur:

The box blur filtering is implemented similar to the above median blurring. However, the algorithm computes the sum for each image channel then assigns the average to each channel. The algorithm scales better as there's no sorting need and is optimized by precomputing the offsets.

1.1.8 Gaussian blur:

The Gaussian blur filter calls upon a custom utility function to generate a 1D Gaussian kernel which is then applied sequentially to the x-axis and the y-axis by calling upon another custom utility function to apply the convolutions. The blurred image is then copied back to the original image; hence the function modifies the image in place. By separating the convolution into two 1D convolutions we aimed to reduce the computational complexity. Despite the fact that the 1D convolution utility function uses nested loops to iterate over the image dimensions, channels and kernel size, given it uses a 1D kernel, its time complexity is proportional to kernel size instead of kernel size squared if we had implemented a 2D kernel.

1.1.9 Edge detection filters:

Our edge detection library supports Roberts Cross, Prewitt, Scharr, and Sobel operators. Its modular design ensures ease of use and flexibility, allowing for simple updates and additions. However, issues with dynamic memory allocation and basic error handling, impact efficiency and reliability for large images or complex operations. Additionally, the Scharr filter amplified noise in the original image, which was mitigated by applying a median blur filter beforehand, this can be observed on Figure 1. However, this introduces fake edges, underscoring the trade-off between edge detection and noise reduction. Noticing that when the original image does not have noise, the output would be noisy after applying a blur filter to edge detector.

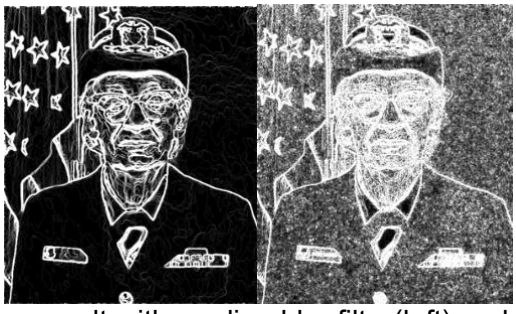

Figure 1. Scharr operator result with median blur filter(left) and without blur filter(right).

1.2 3D Data Volume

1.2.1 3D Gaussian Blur:

Just as 2D Gaussian Blur, a 3D Gaussian convolution can be decomposed into three consecutive applications of 1D Gaussian convolution. By doing this, the total number of operations is reduced to $3n$ from n^3.

1.2.2 3D Median Blur:

For 3D Median Blur, we use the histogram algorithm to find the median. The histogram is an array of 256 elements, where each element's index corresponds to a possible intensity value, and the value at each element represents the number of voxels in the volume that have that intensity. To find the median, iterate over the histogram from the start, cumulatively summing the counts. When the sum reaches half the total number of voxels in the window, the current index in the histogram corresponds to the median value. This method avoids the $O(n \log n)$ complexity of sorting, offering a way to find the median in linear time relative to the number of unique values (which is constant in the case of 8-bit images). This method is particularly suitable for our task where the range of data values is limited and there is a frequent need to update and calculate the median.

1.2.3 Maximum intensity projection:

When implementing Maximum Intensity Projection (MIP), our goal is to select the maximum intensity value for each pixel in multiple 2D images to produce a single 2D image. We first

set the index adjustment function to ensure that the user-specified index range is valid. Then, we traverse each pixel position, compare the corresponding voxel values, and keep the maximum value. One challenge with this approach is how to efficiently process large amounts of data. We address this issue by processing one slice at a time and updating the resultant image incrementally, thus avoiding unnecessary memory footprint.

1.2.4 Minimum intensity projection:

Minimum Intensity Projection (MinIP) is implemented in a similar process to MIP, but aims to highlight the darkest (minimum intensity) structures. For the implementation, we used a similar logical structure to MIP, but chose the minimum instead of the maximum for comparison. We initialize the resultant image using std::numeric_limits<unsigned char>::max() to ensure that any actual voxel values will be less than this initial value, thus correctly implementing the choice of minimum.

1.2.5 Average intensity projection:

Average Intensity Projection (AIP) aims to generate an image by calculating the average value of the corresponding voxels in all slices. To implement AIP, we employ an accumulator array to store the sum of the intensities at each pixel position, and then compute the average by dividing the sum by the number of slices. The main challenge in the implementation of AIP is to maintain the accuracy of the numerical computation, especially when dealing with a large number of slices. For this reason, we employ integer-type accumulators and perform type conversion in the final step to ensure the accuracy of the results.

1.2.6 Slicing:

XY Plane Slicing (sliceXY) generates 2D slices from a 3D volume by iterating over pixel positions at a specific depth. YZ and XZ Plane Slicing (sliceYZ, sliceXZ) produce 2D slices by iterating over depth and height (YZ) or width and depth (XZ), mapping 3D voxel values to reveal internal structures. Key challenges include accurately mapping 3D coordinates to 2D and ensuring efficiency and stability in slicing operations. Solutions involve direct access to voxel values and optimized iteration strategies. The software offers a user-friendly interface for selecting slicing planes and coordinates, and utilizes std::vector for dynamic slice management and std::filesystem for improved file handling, enhancing performance and usability.

2 Performance Evaluation

2.1 Image size

On LHS, we observe the median blurring is the most computationally expensive (using 78% of total 2D blurring run time) due to sorting the neighbourhood values using quickSelect. On RHS, we observe how the blurring filters scale relative to image dimensions (assuming square images).

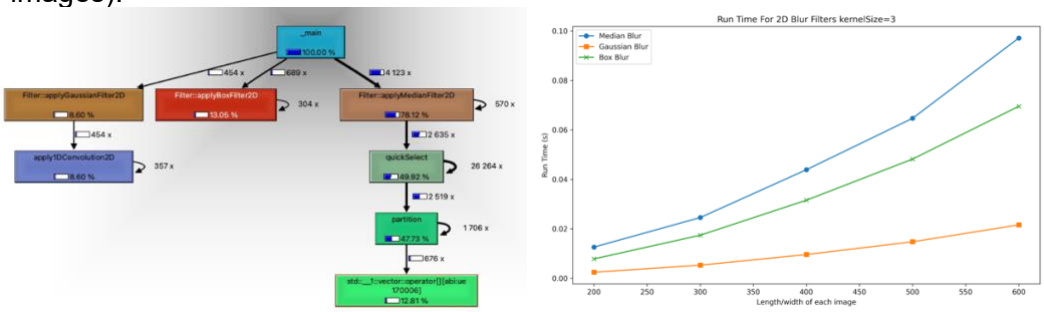

Figure 2. Performance measurement on 2D image blur filters.

2.2 Volume size

On LHS, although we did extra optimization for the 3D median filter, it's still the most computationally expensive. On RHS, as we just increased the number of images in the Z direction to change the volume, so the increase in computation is linear, and the slope of the Gaussian filter is much smaller than the median filter.

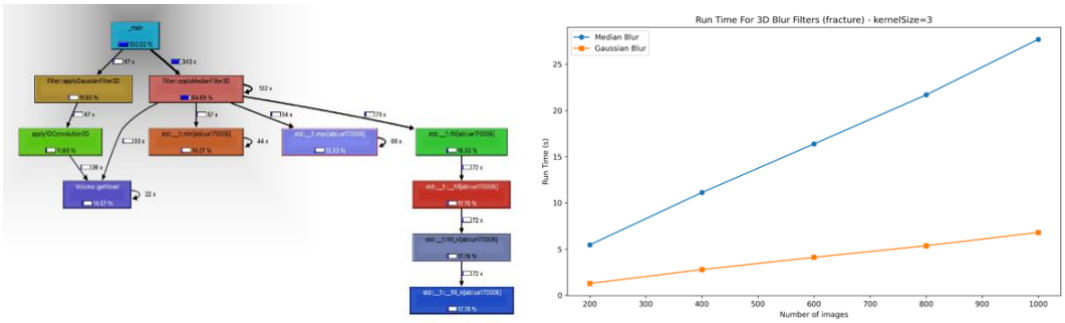

Figure 3. Performance measurement on 3D image blur filters.

	XY	XZ	YZ
Confuciusornis	0.01542130s	0.00327167s	0.0300489s
Fracture	0.00286475s	0.01359970s	0.0294475s

Table 1. Runtime on Volume datasets of slice on 3 axes.

Memory access and cache efficiency vary across XY, XZ, and YZ slicing due to dataset dimensions and layout. XY slicing benefits from contiguous memory, with performance tied to slice size. The confuciusornis dataset's larger XY dimensions mean more data processing, explaining XY performance differences. XZ slicing is relatively cache-friendly, though it involves jumping between slices. Despite a larger depth suggesting slower performance for the fracture dataset, confuciusornis's smaller depth ensures faster performance. YZ slicing, with its complex, non-sequential access, incurs significant cache misses, especially in the fracture dataset due to its greater depth. These dynamics highlight how dataset dimensions critically influence slicing efficiency.

2.3 Kernel size

We note that median blurring on images scales the worst relative to kernel size given the added step of sorting the neighborhood vector. The Gaussian blur is the most efficient algorithm given our implementation uses sequential convolutions of a 1D kernel both for 2D and 3D images.

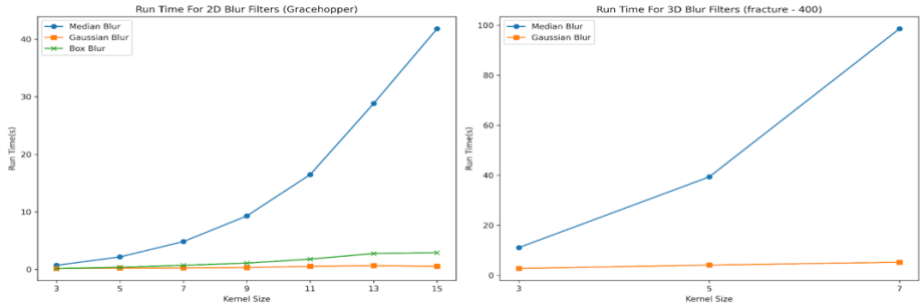

Figure 4. Performance on blur filters for 2D and 3D with various kernel size.

3 Potential Improvements/Changes

For the histogram algorithm, we can consider using a sliding window approach to dynamically update the histogram and use a more efficient search algorithm rather than iterating the entire histogram to find the median.

For 2D/3D blur filters, when dealing with boundaries, we now simply use the strategy of copying the pixel values of the edges to the outside to fill the filter window. Further consideration could be given to implementing and supporting user-selectable padding strategies, such as zero padding, reflect padding, and wrap around. For box blurring, we could have also considered sequential 1D convolutions using a predefined mean kernel to reduce computational complexity.

For future iterations, we recommend enhancing error handling for improved debugging, developing an advanced user interface for real-time result previews, and optimizing memory management to better accommodate large images.