Advanced Programming 2023/24 – Assessment 3 – Group Project

Image Filters, Projections and Slices

Group Name:	**Ziggurat**	
Student Name	**GitHub username**	**Tasks worked on**
Tianju (Tim) Du	edsml-td323	2D Filter, 3D Filter, Filter Test, Performance Test
Melissa Sim	acse-mys20	Grayscale, Brightness, Histogram equalisation, Convert RGB HSL, Projections, optimised projection, Image Class, Unit Test for Image class, projections and helper functions
Wenhao Hong	acse-wh623	Volume and Slice Class, Interaction test, Median Blur
Javonne Porter	acse-jp2923	Some 2d-filters, testing, profiling, performance testing, optimisation
Xiaoye Zhang	acse-xz4019	Team leader, integrate code, code review, 2D filter, 3D filter, Volume class, Image class, Slice class, Projection class, tests, GitHub Actions, Doxygen.
Wenxin Li	edsml-wl123	Slice Class, Performance test

1. Algorithms Explanation

1. 1 2D Image Filters

1.1.1 Grayscale:

Check if RGB or single channel. If single channel, returns the input image. If RGB, calculates pixel intensity using the provided luminosity formula and sets a value for each pixel.

1.1.2 Brightness:

In manual mode, it iterates through each pixel, adding or subtracting the specified value to adjust intensity. In automatic mode, it calculates the current average intensity of the image and determines the necessary adjustment value to achieve an average intensity of 128. This adjustment is applied to all pixels by recursively calling the function.

1.1.3 Histogram equalisation:

The input image's pixel intensity histogram is computed. Then, the cumulative distribution function (CDF) is calculated by summing histogram counts from lowest to highest intensity levels. Intensities are adjusted based on their corresponding CDF probabilities, and original intensities are mapped to new values within the [0, 255] range to redistribute them.

1.1.4 Thresholding:

Thresholding filters convert pixels to black or white depending on intensity. We simply loop pixel by pixel for grayscale images converting based on the pixel intensity. We look at the HSV/HSL images and V/L channels for colour images to find intensity.

1.1.5 Salt and pepper noise:

A salt and pepper filter converts pixels to either black or white at a user-specified proportion. We simply loop over images pixel by pixel, converting depending on the results of random number generators.

1.1.6 Median blur:

Median blur is an image processing algorithm that reduces random noise in images while preserving edges and details. It uses an odd-sized kernel to ensure a central pixel for each operation. By making a histogram in the range of 0 to 255, and finding the median inside the histogram, the algorithm calculates the median of a pixel's intensity within a sliding window. The window slides across the image, handling border pixels by replicating them to fill the window.

1.1.7 Box blur:

The boxBlur function implements the box blur effect, which can effectively reduce noise and make the image look softer by averaging each pixel of the image and its surrounding pixels. The function uses a two-step process: first, blurring is applied horizontally, then blurring is applied vertically, thus achieving an even treatment of the entire image. The filter kernel size must be odd to ensure symmetry, but for even-sized kernels, the function adapts by including an extra pixel in the calculation. Boundary processing is performed by a "clamping" strategy, which ensures the validity of all pixel values in the process. Through this step-by-step process, boxBlur provides an efficient way to soften the image and reduce noise while keeping the computational complexity low.

1.1.8 Gaussian blur:

The Gaussian blur function applies the Gaussian kernel, which is centred around each pixel of the image. The kernel size must be odd to ensure symmetry around the central pixel. The kernel itself is generated based on the Gaussian distribution, normalised so that its values sum to one. A “clamp” is used to adjust the indices to point to the nearest valid pixel inside the image to address boundary conditions. During convolution, pixel intensities are weighted by the kernel and summed to produce

the blurred effect, with final pixel values clipped to the 0-255 range suitable for 8-bit images. The result is a new image with a smooth, blurred appearance.

1.1.9 Edge detection filters:

The edge detection methods implemented are the Sobel, Prewitt, Scharr, and Roberts Cross operators, each designed to highlight significant image intensity changes indicative of edges.
Sobel Operator: This operator utilises 3x3 kernels to approximate the image intensity gradient, emphasising areas of high spatial frequency.
Prewitt Operator: Similar to the Sobel operator, it employs different coefficients and provides a simpler gradient approximation. Scharr Operator: Offers a more rotationally symmetric approach to edge detection and is sensitive to diagonal movements. Roberts Cross Operator: A simpler operator using a 2x2 convolution kernel, it focuses on highlighting rapid intensity changes.
Otherwise, for vh_anatomy_sp15.png, after applying blur and then applying the edge detection, you can get a really good result, given that the blur has the ability to reduce the noise and help to detect the edge.

1.2 3D Volume Filters and Projections

1.2.1 3D Gaussian Blur:

The two functions for applying a 3D Gaussian Blur differ in their approach: one uses a 3s kernel, and the other utilizes a separable approach with a 1D kernel applied the along the axis. The 3D kernel function performs a convolution operatiit computes the weighted sum of the surrounding voxels using a 3D Gaussian kernel for each voxsian kernel. This is computationally intensive because it requires iterating over all voxels within the kernel's volume for each voxel in the input volume. The complexity is a function of the number of voxels in the kernel cubed (kernelSize^3) multiplied by the number of voxels in the input volume. On the other hand, the 1D kernel function takes advantage of the separability property of the Gaussian kernel. This significantly reduces the number of computations because for each axis, it only needs to consider a line of voxels rather than a volume. The complexity here is a function effectively 3 * kernelSize * number of input voxels. The 1D kernel approach is faster because it greatly reduces the number of multiplications and additions needed to compute the blur. Instead of a cubic increase in computations with respect to the kernel size, it only increases linearly, making it more efficient especially for larger kernels.

1.2.2 3D Median Blur:

There are two functions to apply 3D Median Blur. One with 3D kernel to move across voxel. Another optimised one using a histogram and a sliding window for the 3D median filter. In optimised version, when the window slides across the image, only the histogram entries for the voxels entering and exiting the window are updated. This avoids the need to sort all the voxels within the kernel window each time it moves, which would be computationally expensive, especially for large kernel sizes (lookup complexity of O(1)). Moreover, the histogram-based method with a sliding window is faster for larger kernel sizes because it leverages the fact that in a sliding window, most of the voxel values are shared between consecutive windows. Thus, it only needs to update the histogram for the small portion of the voxel values that change, which is far more efficient than re-sorting all the values at each step. As a result, the time complexity of this optimized algorithm is approximately O(n). In contrast, the second approach without the histogram is computationally costly because it performs a sorting operation (with a complexity of O(n log n), where n is the number of voxels in the window) for each voxel in the volume, making it much slower for large kernel sizes. The two functions generate the exact same test images.

1.2.3 Maximum and Minimum intensity projection (MIP, mIP)

Projection functions iterate through each pixel of the output image, traversing the depth dimension of the volume stack. For MIP and mIP, they update the maximum/minimum pixel intensity values until reaching the last 2D image, setting the output pixel accordingly.

1.2.4 Average and Median intensity projections (AIP, MedIP)

For AIP sum all intensity values and calculate the mean by dividing by the number of stacks. For MedIP stores values in an array, sorting them using quicksort recursively to find the median based on array size and middle/middle two values and set the output pixel.

1.2.5 Slice:

The slice class slices the Volume object as required, e.g. YZ plane and XZ plane slices. Given the slice index (x and y), it calculates the required pixel positions by traversing the Volume object pixel values and applying the corresponding algorithm, and finally returns a one-dimensional array and a two-dimensional image using the stbi library methods.

2 Performance Evaluation

2.1.1 Image Size

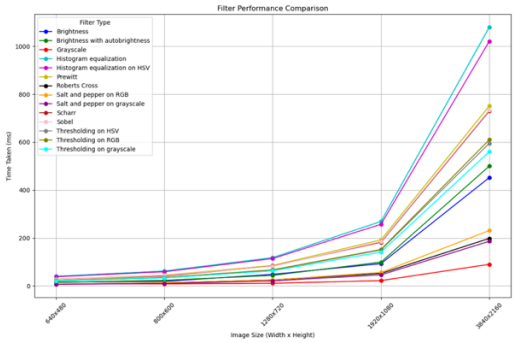

Figure.1

As shown in the chart, time taken by applying grayscale filter with larger image sizes increases the least obvious. Similar patterns can also be observed in the performances of salt and pepper filter and Robert Cross edge detection. Time taken by other filters shows more significant rises with the increase of image size, while histogram equalization shows the most rapid growth. This is because grayscale only involves a simple iteration through the image, and the randomness of noise addition in salt and pepper is not quite computationally intensive. Other methods such as Sobel and Scharr edge detection require more complex computing operations (e.g. kernel convolution). And histogram equalization involves double pass of the whole image and multiple computational steps, making it scale more significantly with image size.

2.1.2 Volume Size- Projection Time Performance

The median operation is the slowest due to sorting. Implementing partial sorting algorithms or using histograms to calculate the median could improve runtime. This approach was implemented, and speed tests demonstrated that using histograms runs faster for larger stacks. Therefore, a threshold was introduced to determine which function to utilize based on the volume depth. Max and min operations are slower than average due to individual pixel checks and conditions.

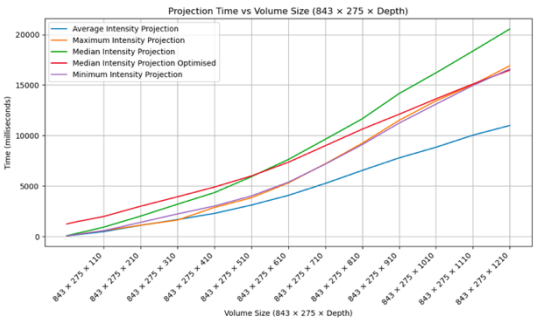

Figure.2

Early stopping criteria could optimize these functions by halting when the maximum value reaches 255 and the minimum value reaches 0, representing the highest and lowest possible values, respectively. For all functions, loop unrolling could speed up operations by simultaneously processing multiple pixel depth values for comparison, storage, and summation (for average) at once.

2.1.3 Kernel Size

2D kernel

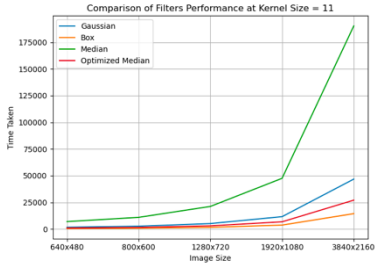

Figure.3

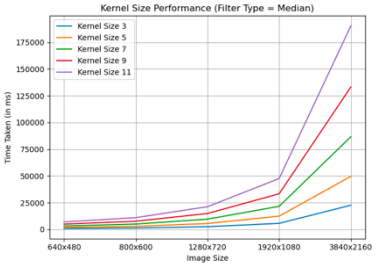

Figure.4

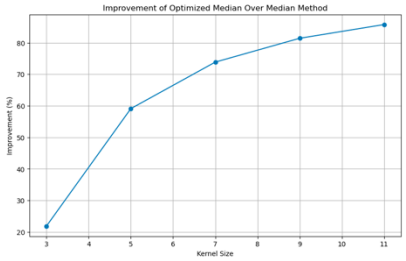

Figure.5

In general, the running time of the filter increases with image size and kernel size(Figure.4). Due to the sorting feature of the Median filter usually is the slowest among the three. With a fixed kernel size of 11, the time required to apply the original version of the Median filter is much higher than that of Gaussian and Box due to the sorting algorithm. After realising the slow speed of Median Blur, we improved the sorting algorithm by using the histogram to sort the values and sliding windows. The performance improves together with a larger kernel and image size, which reaches to 80 % (Figure.5). improvement and outperforms the Gaussian blur (Figure.3) at kernel size 11.

3D kernel

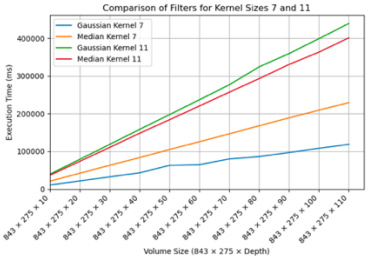

Figure.6

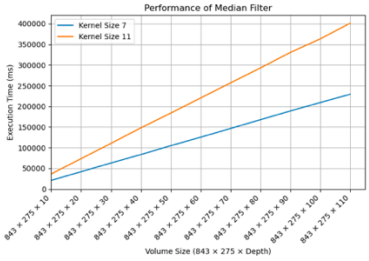

Figure.7

In general, a similar pattern is shown by applying 3D kernels compared to 2D kernels. Due to the sorting steps in the Median filter, it is generally slower than the Gaussian filter. However, we noticed that when the kernel size increases to 11, Gaussian filter is slower than Median filter, which may because the step of creating a 3D kernel in Gaussian blur consumes more time with larger kernel sizes. After optimised both functions, the performance significantly improved.

2.2 Profiling with Gperf

We noticed that `medianIntensityProjection` was slow. Our initial guess for the bottleneck was our sorting algorithm. We implemented versions of the code using QuickSort, and MergeSort but found our initial QuickSort implementation was faster 3.54x faster. Profiling showed that calling the `getSlice` and 'getPixel' function was the main factor to slowing down the code. A function `getVoxel` was developed to counter. The `getPixel` approach took ~371s, and the new approach with `getVoxel` to 0.02s. A histogram to calculate the median was implemented for other functions which showed faster runtimes than sorting. Knowing that we had already optimised the main bottleneck as much as we could, we then decided to combine all the approaches, selecting the most efficient solution based on the number of slices specified.

Potential Improvements/Changes

The current project has low memory utilization, for each new project will create a separate pointer, resulting in memory buildup and other problems, subsequent optimization can use smart pointers to improve memory utilization. For large size images, the project processing speed is too slow, subsequent optimization can use multi-threading to accelerate the image processing process.