Advanced Programming 2023/24 – Assessment 3 – Group Project

Image Filters, Projections and Slices

Group Name:	Depth-first-search	
Student Name	**GitHub username**	**Tasks worked on**
Alex Njeumi	acse-ann23	2d- SaltPepper 2d- Threshold 3d- Average Intensity 2d - Tests
Yixiao Jing	acse-yj823	2d-Edge Detection 3d-Gaussian Blur 3d-Medium Blur Image, volume classes 3d-MIP
Andrei Danila	edsml-acd23	2D-Gaussian, Box Blur 3D-Gaussian, Median Overall implementation, merging User interface Filter, image classes Optimization
Fan Huang	acse-fh223	2D - Edge Detection 3D - Median Filter 3D - Slicing 3D - MIP
Shrreya Behll	edsml-sb3323	2d-Greyscale, Brightness Dullness, Histogram 3d-Slicing, Merging and Pull Request, Performance class, Graphs for images and volume, Code Reviews and Report
Wenxin Zhang	acse-wz2723	2d-Median Blur 3d-Gaussian Blur 3d-MIP, MinIP, AIP 3d-Volume

Note: Yixiao Jing, one of our team members, was not able to work on the project from Wednesday to Friday, due to illness. However, we the teammates would like to mention that he was communicative and worked in a limited capacity on the first two days of the project.

1 Algorithms Explanation

2D Image Filters

Grayscale:

The **applyGrayscale** function transforms a colored image into grayscale while keeping it in RGB format for compatibility. It calculates the grayscale value for each pixel using the luminosity method, with a weighted average of the RGB values to reflect human perception of brightness. This value is then applied to all RGB channels of each pixel, ensuring the image remains in grayscale. The modified image data is then saved back into the Image object, turning the original color image into grayscale while maintaining its RGB structure.

Brightness:

The **applyBrightnessDullness** method modifies an image's brightness, either brightening or dimming it based on a given adjustment value. If unspecified, it calculates the image's average brightness and adjusts it towards a medium gray (128) level. This involves iterating over all pixels, computing the average brightness, and applying a uniform adjustment across the image. The adjustment ensures pixel values remain within the valid range using the ClampColorValue function. The adjusted image is then updated in the Image object, changing its overall brightness.

Histogram equalisation:

Histogram equalization enhances image contrast in both grayscale and RGB images. For grayscale, the **applyGrayscaleEqualization** function converts the image to grayscale, calculates its intensity histogram and Cumulative Distribution Function (CDF), then remaps the pixel values to distribute intensities more evenly, thereby improving contrast. In the case of RGB images, two approaches exist: RGB to HSV and RGB to HSL. The **applyRGB_HSVEqualization** function works by adjusting the value (V) channel in the HSV color space to improve contrast, maintaining color fidelity upon converting back to RGB. Similarly, the **applyRGB_HSLEqualization** function enhances contrast by modifying the luminance (L) channel in the HSL space, ensuring color integrity and enhanced visibility under different lighting conditions. These methods collectively improve visual quality by equalizing pixel intensity distribution across various color spaces.

Thresholding:

The **applyThreshold** function operates on an image, selectively altering pixel values based on a specified threshold and chosen channel—luminance (L), value (V), or grayscale (G). Through a pixel-wise traversal, it converts RGB values to the appropriate color space—HSL for luminance, HSV for value, or directly assesses intensity for grayscale. By comparing the threshold against the relevant channel's value, pixels are classified as either exceeding or falling below the threshold, leading to their adjustment to white or black, respectively. This process facilitates the isolation of regions of interest within the image, enhancing its interpretability or aiding in subsequent image analysis tasks.

Salt and pepper noise:

The **applySaltPepperNoise** function injects salt-and-pepper noise into an image to simulate digital imperfections, useful for testing algorithm robustness. It operates on an Image object, altering a specified percentage of pixels. The algorithm selects pixels uniformly at random across the image's entire area. Each selected pixel is randomly assigned to become either white (salt) or black (pepper), based on a 50/50 chance. The percentage of pixels to modify is determined by the input parameter, guiding the algorithm on how extensively to apply the noise. This method effectively creates a realistic noise pattern, challenging algorithms to maintain performance despite the added visual noise, thus simulating conditions algorithms might face in real-world applications.

Median blur:

The **applyMedianBlur** function uses a median blur algorithm to reduce "salt-and-pepper" noise in images, preserving edge clarity by applying a specified kernel size to each pixel. For each pixel, it gathers neighboring pixel values within the kernel's radius, then finds the median of these values, a step that effectively ignores extreme outliers. This median replaces the original pixel value, ensuring that edges remain sharp while noise is minimized. The process, applied to all image channels, enhances image quality without significant blurring, making it ideal for computer vision preprocessing where edge detail is important. This method updates the image with the denoised data, offering a balanced approach to noise reduction and edge preservation.

Box blur:

The **applyBoxBlur** function applies a simple box blur effect to an image. The function operates by iterating over each pixel in the input image and computing the average colour value of the pixels within a specified radius (**blurSize**) of the current pixel. This process involves summing the colour values (across all channels) of each pixel within the neighbourhood defined by **blurSize** and then dividing by the total number of pixels considered to form the blurred pixel value. This averaging process smoothens the image by blending the colours of adjacent pixels, effectively reducing the image's detail and noise. The function updates the original image data with the newly computed blurred data.

Gaussian blur:

The **applyGaussianBlur** function smooths images by utilizing a Gaussian blur, defined by a kernel size and a standard deviation (sigma). It creates a Gaussian kernel, with odd dimensions for symmetry, emphasizing closer pixels through Gaussian distribution weights, which mimic a natural blur. The kernel, normalized to prevent altering the image's brightness, is applied to blend each pixel with its neighbors, softening the image uniformly. This technique smooths sharp edges and details, similar to a soft-focus lens effect, and handles edge pixels by adjusting the kernel's application. The updated, blurred image data replaces the original, achieving noise reduction and detail softening efficiently.

Edge detection filters:

The **applyEdgeDetection** function uses convolution operators operatorX and operatorY to highlight image edges by detecting intensity changes. By applying these operators across the image and calculating the gradient's magnitude for each pixel, it identifies edges. The function's adaptability comes from the ability to switch these operators, allowing the implementation of various edge detection techniques like Sobel, Prewitt, and Scharr with 3x3 operators, or Roberts' Cross with 2x2 operators. Each method has unique kernels that influence edge detection characteristics, enabling customization to suit specific needs. This approach transforms the original image into one where edges are emphasized, facilitating further image analysis or processing tasks.

1.1 3D Data Volume

3D Gaussian Blur:

The **apply3DGaussianBlur** method diffuses pixel values in three dimensions using a 3D Gaussian kernel, effectively smoothing out noise and details in images or volumes. This kernel, shaped according to the Gaussian formula, focuses more weight on central pixels to mimic a natural blur effect, with its values normalized to preserve overall brightness. Through convolution, each voxel's value is recalculated as a weighted average based on its proximity to the kernel's center. This process is applied uniformly across the volume, steering clear of the edges to prevent boundary issues. The result is a uniformly blurred image where noise is reduced without altering the core structure, making it ideal for applications requiring noise reduction while retaining essential features

3D Median Blur:

The **apply3DMedianFilter** function operates as a 3D noise reduction technique that applies a median filtering process within a volumetric context. By utilizing a cube-shaped kernel that traverses through the volume data, this method seeks to diminish noise and irregularities, focusing on preserving the structural integrity of the volume without the reliance on weighted averages. The core of this approach involves sorting the values within the kernel space and selecting the median value as the representative for the central voxel, ensuring that outliers have minimal impact on the resultant volume. This method is particularly adept at maintaining edges and features, thanks to its non-linear nature, making it highly effective for applications where the preservation of distinct boundaries and features is critical. The operation is carefully bounded to avoid alterations at the edges, thereby avoiding any boundary-induced artifacts.

Maximum intensity projection:

The **generateMIP** function creates a Maximum Intensity Projection (MIP) image from a 3D volume data. At first, a vector mipData, is created to store the MIP image's pixel data, initialized with zeros. To find maximum intensity, for every pixel position (x, y) across all 2D slices, the algorithm searches for the highest intensity value through the depth of the volume. The maximum intensity found for each pixel location is saved into mipData, constructing the MIP image pixel by pixel. Finally, the function saves mipData as a grayscale PNG image. The result is a 2D image where each pixel represents the maximum intensity found at that position across the entire 3D volume.

Minimum intensity projection:

The **generateMinIP** function creates a Minimum Intensity Projection (MinIP) image from 3D volume data. It initializes a vector to store the MinIP image's pixel data, setting all pixels to the highest possible intensity value to start. The function then iterates over each pixel position in the 2D plane. For every (x, y) position, the lowest intensity value found across all slices is stored and constructued the MinIP image. The result is a 2D image where each pixel represents the minimum intensity, highlighting the lowest-intensity features.

Average intensity projection:

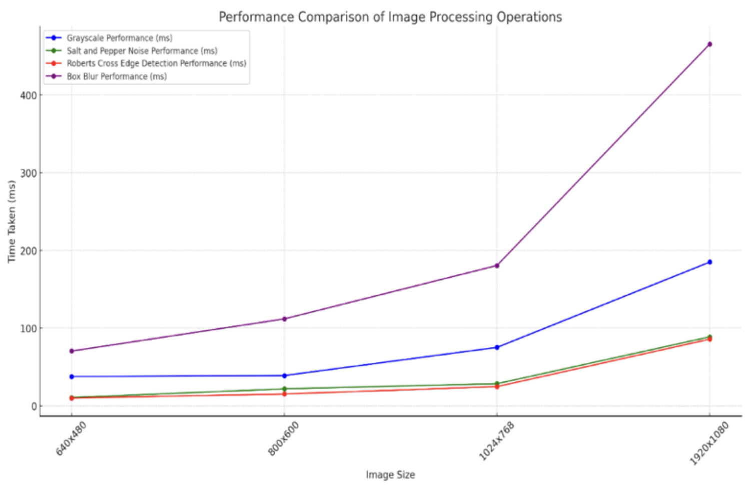
The **generateAIP** function creates an Average Intensity Projection (AIP) image from 3D volume data. A vector is set up to hold the pixel data for the AIP image, initially filled with zeros. The function then goes through each pixel on the 2D plane. For every pixel, it sums up the intensity values from all corresponding pixels across the depth of the volume. After calculating the total intensity sum for each pixel, it computes the average by dividing the total sum by the depth. This average intensity is saved and constructed the AIP image pixel by pixel, saved as a grayscale PNG file. The resulting 2D image illustrates the average intensity for each pixel, providing a holistic view of the internal features based on their average intensities.

Slicing:

The **saveSlice** function extracts and saves a specific slice of a 3D volume as a 2D image, based on the specified axis (X, Y, or Z) and index within the volume. For the Z-axis, it directly accesses the corresponding slice array and saves it. For the X and Y axes, the algorithm constructs the slice by iterating over the volume's depth and either width or height, respectively, collecting pixel values along the specified plane. This collected data is then rearranged into a 2D array that represents the slice. This method allows for the visualization and analysis of specific cross-sections within a volumetric dataset.

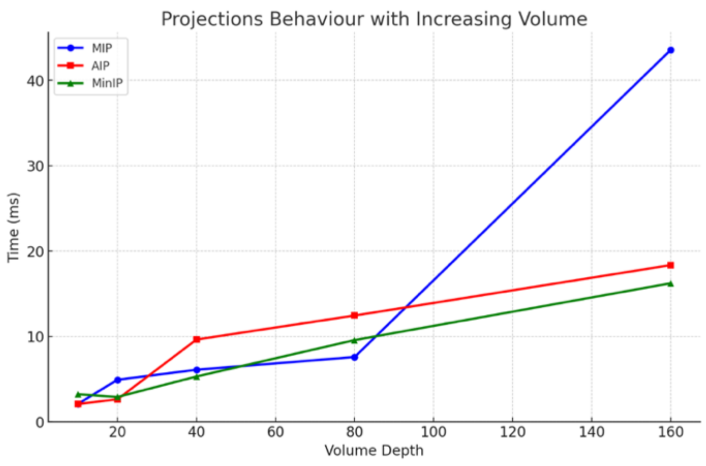
2 Performance Evaluation

2.1 Image size



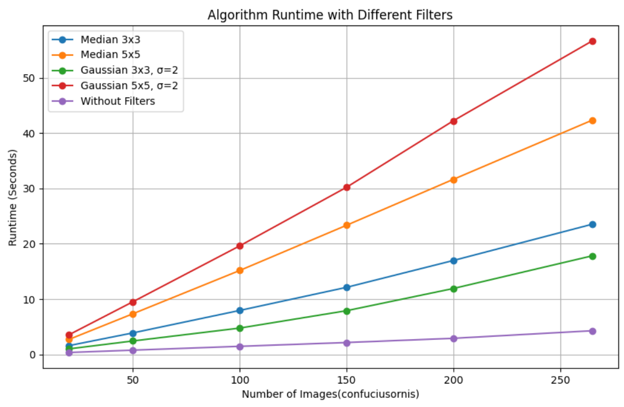
The graph illustrates the performance of four image processing operations—grayscale conversion, salt and pepper noise addition, Roberts Cross edge detection, and box blur—across different image sizes. Notably, the time taken for each operation increases with the size of the image, but the rate of increase varies between the operations. The box blur operation exhibits the most significant increase in time as image size grows, indicating a higher computational complexity compared to the others. Grayscale conversion and salt and pepper noise addition show a more moderate increase in time, suggesting they are less computationally intensive. Roberts Cross edge detection, while starting off with low times for smaller images, also shows a steady increase but remains relatively efficient for larger image sizes. This behaviour suggests that while all operations are affected by image size, the impact on performance varies significantly, highlighting the importance of considering computational complexity when processing images of different sizes.

2.2 Volume size



The graph showcases the performance of three projection techniques—Maximum Intensity Projection (MIP), Average Intensity Projection (AIP), and Minimum Intensity Projection (MinIP)—as the depth of the volume increases. As volume depth grows, the processing time for each technique also escalates, with MIP displaying a notably steeper increase in processing time, particularly as the depth reaches 160. This suggests that MIP may require more computational resources or is less efficient with larger data sets compared to AIP and MinIP. AIP's processing time increases at a moderate rate, indicating a balanced approach between processing speed and data depth. MinIP, while showing an increase in processing time, does so at the least steep curve, hinting at its efficiency or simpler computational demands with larger volumes. This analysis helps in understanding the scalability and efficiency of these projection techniques in handling volumetric data of varying sizes.

2.3 Kernel size



The image shows the trend of the time consumed for applying different filters to a 3D image to implement the projection as the size of the image grows. We compose 3D images of different sizes by stacking different numbers of 2D images. Overall, the time consumed to run the algorithm tends to increase linearly as more images are stacked. Also, for the image datasets we tested, applying a larger kernel size of the filter always requires more runtime because a larger filter size means that we need to perform more computational content for each convolution operation.

Potential Improvements/Changes

- Enhanced testing suite: We recognize the need for enhancing our testing suite to ensure comprehensive coverage and catch any potential bugs or issues. We see particular room for improvement in our 3D testing capabilities.
- Strengthening user interface resilience: At present, it's possible for users to disrupt our user interface through invalid inputs. While we've addressed most scenarios, we've pinpointed a few instances where the interface could be compromised.
- Boosting algorithm performance: While we believe our models are already optimized for efficiency, there's always room for further enhancements by employing more advanced algorithms.
- Significant advancements in 3D median filter performance: We have substantially improved the performance of our 3D median filter. Initially processing 1300 images took around 30 minutes, but we've managed to reduce this to just 30 seconds.