1. Task Distribution

• **Aijia Yang:** Edge detection filters, Basic tests of 2D filters. Questions:4a,4b,4c,4d
• **Antonio Nikoloski:** 3D Tests, Build Filter structure and the whole project structure Questions:3D; Projection testing and developing.
• **Sazina Azam:** Image Sharpening Filter, Designed Test Format Question:3
• **Tyana Tshiota:** (Group Lead): QA, Updated License, Doxygen Documentation, clang Linting Workflow, Colour Correction Filters, ColorConverter Utility Questions: 1a,1b,1c,1d,1e
• **Xiaorui Zhou:** Edge detection filters, tests for the number of channels in all 2D images. Questions: 4a,4b,4c,4d
• **Zewei Zhang:** Blur filters (BoxBlur, GaussianBlur, MedianBlur), Tests for 2D blur filters. Questions: 2a,2b,2c
• **Ziqi Yue:**Build the basic structure and 3D tests. Question:3D

2. Design of the Code

Core Design Principles: The codebase follows OOP principles, ensuring modularity, encapsulation, and polymorphism. Each 2D filter inherits from Filter2D, while 3D filters inherit from Filter3D, enforcing a consistent apply() method for a uniform interface. This design allows seamless integration of new filters with minimal refactoring. Encapsulation keeps each class self-contained, improving code maintainability and security while reducing unintended side effects. The modular approach enhances clarity, reusability, and scalability, making future extensions straightforward without introducing redundancy.

Class Structure: Filter2D and Filter3D serve as abstract base classes. Filter2D handles 2D image processing, while Filter3D extends this to volumetric data. Each derived filter specialises in a distinct operation, grouped by functionality.

Colour Correction Filters: The color correction filters improve image clarity, brightness, contrast, and noise management. GreyscaleFilter2D converts RGB images to grayscale using the luminance formula, handling both RGB and RGBA formats by discarding the alpha channel. BrightnessFilter2D offers Manual and Auto modes, with branchless clamping for efficiency and adaptive brightness adjustment for low-light images. HistogramEqualisationFilter2D enhances contrast using HSV (Value channel) or HSL (Lightness channel) to improve vibrancy while avoiding oversaturation. ThresholdFilter2D enables binary segmentation across multiple color spaces, while SaltPepperFilter2D introduces noise for degradation simulation. The ColorConverter utility standardizes RGB, HSV, and HSL transformations. Encapsulation via an anonymous namespace prevents conflicts, and std::clamp ensures pixel values remain within valid ranges.

Image Blur Filters: Blur filters use spatial convolution to smooth images and reduce noise. The Box Blur averages neighboring pixels using a sliding kernel. The Gaussian Blur applies a weighted convolution with a Gaussian kernel, prioritizing central pixels and reducing high-frequency noise. The Median Blur replaces each pixel with the median of its neighborhood, effectively removing salt-and-pepper noise while preserving edges. All filters handle edges by clamping out-of-bound indices to the nearest valid pixel. Filters operate on a temporary copy of the original image to avoid interference during processing, and the final pixel values are clamped to the range of 0–255 using efficient, branchless operations.

Sharpening Filter: The *SharpenFilter2D* class applies a Laplacian-based sharpening filter to enhance edges in 2D images. It follows OOP principles, inheriting from the *Filter2D* base class for modularity and scalability. The design ensures code reusability, allowing new filters like Gaussian blur to be added easily. A temporary buffer *(std::vector)* prevents data corruption, while clamping pixel values maintains valid color ranges. The filter modifies images in-place, reducing memory overhead and improving efficiency. This approach balances performance, accuracy, and flexibility, making *SharpenFilter2D* a robust solution for sharpening while maintaining a clean and extensible design.

Edge Detection Filters: EdgeDetectionFilter2D, inheriting from Filter2D, implements edge detection algorithms like Sobel, Prewitt, Scharr, and Roberts through polymorphism. It accepts an edgeType string to select the algorithm and overrides the apply method to apply the chosen filter. If the image is in color, it is converted to grayscale first. The applyEdgeKernels function performs convolution by calculating gradient magnitudes for Sobel, Prewitt, and Scharr methods. To handle edge pixels, std::clamp is used to ensure safe indexing when the kernel extends beyond the image boundaries. This prevents out-of-bound errors by clamping the index to a valid range, replicating nearest pixel values at the edges.

3D Filters: Design Discussion: 3D Filters, Orthographic Projections, and Slices

3D Module: The module contains projection, 3D Gaussian blur, 3D median blur, and slices. The module extends image processing capabilities to 3D data, which can be applied in CT scans and medical imaging.

Projection: The Projection class implements orthogonal projections, including Maximum Intensity Projection (MIP), Minimum Intensity Projection (MinIP), Average Intensity Projection (MeanAIP), and Median Average Intensity Projection (MedianAIP). These projections use static methods and reference the Volume object, calculating pixel data along the z-axis for each 2D slice. The design emphasizes statelessness, allowing easy integration of new projection methods.

Gaussian Blur 3D: GaussianBlur3D, inheriting from Filter3D, applies spatial smoothing by convolving with a 3D Gaussian kernel. The kernel's weights are precomputed and normalized based on user-defined size and standard deviation for efficiency. For edge processing, nearest boundary copying is used to minimize artifacts at the data boundaries.

Median Blur 3D: Inherited from Filter3D, this method applies median filtering within a 3D neighborhood to remove impulse noise while preserving edges. It utilizes a fast selection algorithm for efficient median calculation, reducing computational overhead.

Slice: The Slice class allows for extracting detailed cross-sections from 3D volumetric data. By selecting a fixed coordinate (x or y), it traverses the depth to extract pixel rows at corresponding positions. This direct indexing, combined with optimized memory access patterns, ensures high performance and low overhead. The resulting slices provide detailed visualization of internal structures, crucial for analysis in medical imaging and engineering diagnostics.

3. Performance

Optimisation

Colour Correction and Per-Pixel Modifiers: Used branchless clamping in BrightnessFilter2D, replacing conditional logic with arithmetic operations to enhance pixel adjustment efficiency. The SaltPepperFilter2D employs a dual noise strategy, balancing speed and precision by switching between fast and precise noise generation methods. Direct pixel manipulation minimizes memory overhead and accelerates image processing.

Image Blur Filters: The Box Blur and Gaussian Blur exhibit a time complexity of $O(w \times h \times ch \times k^2)$, where k is the kernel radius, while the Median Blur incurs additional overhead due to sorting, resulting in $O(w \times h \times ch \times k^2 \log k^2)$. Space complexity is primarily governed by the storage of temporary data copies ($O(w \times h \times ch)$ to avoid in-place modification artifacts. The Gaussian Blur further allocates memory for the kernel weights $O(k^2)$, while the Median Blur dynamically stores neighborhood values during sorting, introducing transient $O(k^2)$ or $O(k^3)$ memory footprints per pixel.

Sharpening Filter: The SharpenFilter2D applies a Laplacian kernel to enhance edges, with a time complexity of $O(w \times h \times ch)$, where w, h, and ch represent width, height, and channels, respectively. Since each pixel undergoes a fixed 3×3 convolution, the complexity remains linear in the number of pixels. To optimize performance, branchless clamping (std::clamp) is used to ensure pixel values remain within [0, 255], reducing CPU pipeline stalls. Additionally, the use of in-place processing minimizes memory overhead by avoiding unnecessary buffer allocations. Efficient memory access patterns ensure better cache utilization, accelerating image processing.

Edge Detection Filters: The EdgeDetectionFilter2D class has O(w * h * ch) time and space complexity from grayscale, data copy, and edge detection. Our approach modifies pixels directly to cut redundant memory use, reducing cache misses and boosting performance. Bounded indexing for kernel application avoids out - of - bounds checks. Sobel, Prewitt, and Scharr use 3x3 kernels with O(w * h * N^2)(N = 3) complexity. Scharr is more edge - sensitive but costlier. Roberts Cross uses a 2x2 kernel, has O(w * h) complexity, is fast but less accurate. Regarding performance bottlenecks, data copying gobbles up memory and time for large images. Also, std::sqrt for gradient magnitude is costly and hikes the computational load.

3D Filters:

Projection: The projection method has linear complexity (O(N)), which means that only one scan is done for each 3D pixel. It uses a small amount of memory because only one output image needs to be stored.

Blur: 3D Gaussian blur has complexity $O(N \cdot K^3)$, and its complexity is related to the kernel size (k). Median blur is slightly slower due to the need for sorting, and has complexity $O(N \cdot K^3 \cdot \log K^3)$. The memory requirement of both is proportional to the kernel size.

Slicing: The slicing algorithm is very efficient and has linear complexity (O(N)). The complexity of the slice size is directly related to the number of images, and it uses minimal additional memory besides the original data.

Execution Time and Memory usage:

This section evaluates the performance of different image processing filters in terms of execution time and memory usage. Performance is assessed based on computational complexity and empirical execution time measurements. London

Measurement Methods: Execution time was measured using **high-resolution timers** while processing a high-resolution image (5760 × 8640, 31.7MB, JPG). Memory usage was monitored to assess space complexity, particularly for filters requiring additional storage (e.g., Gaussian Blur, Median Blur). Each test was run multiple times for consistency.

Filter	**Execution Time for Large Image(s)**	**Filter**	**Execution Time for Large Image(s)**
Grayscale	**2.245**	Box Blur	**103.909**
Brightness Adjustment	**3.127**	Gaussian Blur	**130.001**
Histogram Equalisation	**2.498**	Median Blur	**414.459**
Thresholding	**2.278**	Sharpening Filter	**8.096**
Salt and Pepper Noise	**2.823**	Edge Detection	**71.595**

2D Filter Performance Bottlenecks & Optimizations

- **Box Blur & Gaussian Blur:** High execution times due to large kernel convolutions. Optimized using separable kernels for efficiency.
- **Median Blur:** Highest execution time (414459 ms) due to sorting operations. Could be optimized with faster median-finding algorithms.

- **Sharpening Filter:** Executes in **8095.82 ms**, maintaining linear complexity **O(w × h × ch)**. Optimization through **branchless clamping and efficient memory access**.
- **Edge Detection:** Performs at **71595.2 ms**, optimized by reducing redundant memory operations and using **clamped indexing**.

Filter	**Execution Time(s)**	**Filter**	**Execution Time(s)**
Projection	**20.910**	3D Median Blur	**842.360**
3D Gaussian Blur	**141.659**	3D Slice	**6.580**

3D Filter Performance Bottlenecks & Optimizations

- The Projection class (e.g., MIP, MinIP, MeanAIP, MedianAIP) achieves linear time complexity of O(N), where N is the total number of voxels (w × h × d). However, for large datasets (e.g., 5760 × 8640 × 100), the execution time of 20.910 seconds indicates a bottleneck in sequential z-axis traversal.
- Memory usage remains low, requiring only O(w × h) for the output 2D image, but cache inefficiency arises from non-contiguous memory access across depth slices.
- The sheer size of 3D data (e.g., millions of voxels) amplifies memory bandwidth demands and cache misses, slowing all filters.

Future Optimisations:

Future optimizations may include multithreading (OpenMP) for 2D and 3D operations, enabling parallel processing of image filtering, projection, and slicing. GPU acceleration (CUDA/OpenCL) will significantly speed up Gaussian blur, median blur, and projection calculations by distributing the workload across thousands of cores. Currently, 3D Blur is relatively computationally expensive. For projection, AI-based super-resolution methods (similar to DLSS) can achieve low-resolution rendering and deep learning upscaling, reducing computational costs and maintaining quality. These improvements will increase efficiency, scalability, and real-time performance of 2D and 3D workflows.

4. Final Discussion

Group Dijkstra successfully delivered the “Image Filters, Projections, and Slices” project in C++, emphasizing modularity, performance, and technical robustness. Key innovations include the ColorConverter utility, designed for seamless RGB, HSV, and HSL conversions, ensuring adaptability in color manipulation tasks. The BrightnessFilter2D supports dual-mode brightness control, offering both automatic and manual adjustments for enhanced flexibility. The SaltPepperFilter2D employs adaptive noise distribution for realistic degradation simulation, improving its effectiveness in testing noise resilience. Image blur filters leverage extend-based boundary handling to achieve effective noise reduction and edge preservation, with planned enhancements such as separable Gaussian convolution, approximate median selection, SIMD vectorisation, and memory-efficient tiling for real-time performance. The Sharpening Filter employs advanced convolution techniques with dynamic weight adjustments to emphasize edges while minimizing artifacts. Edge detection filters utilize a temporary buffer to prevent direct data modification, with future optimizations planned for in-place processing and OpenMP/SIMD parallelisation. For 3D data, optimized Gaussian and median blurring algorithms enable efficient volumetric filtering, while the Slice Filter extracts cross-sections for detailed visualization, crucial in medical imaging. Orthographic filters implement Maximum Intensity Projection (MIP) and Minimum Intensity Projection (MinIP) for accurate volumetric data representation, with potential for expanded projection methods. The project’s technical depth, combined with clear communication and teamwork, ensured effective integration of these features, delivering a robust and modular solution with high performance.

END