Sidharth Shanmugam

Final Project Report

Machine Vision-Based Anti-Backscatter Lighting System for Unmanned Underwater Vehicles

20 May, 2024

4th Year Project Final Report for the Degree of
MEng in Electronic and Computer Engineering

School of Physics, Engineering and Technology,
University of York, England YO10 5DD

Academic Project Supervisors:
Prof. Paul D Mitchell, Prof. Andy M Tyrrell

Abstract

Underwater imaging is critical for various applications such as marine biology, underwater archaeology, and pipeline inspection. However, backscatter from suspended particles significantly degrades image quality, posing a substantial challenge. This paper addresses the challenge of mitigating underwater backscatter in real-time imaging by developing a backscatter cancellation system using a Raspberry Pi Single Board Computer. The paper proposes a solution that leverages a combination of image processing techniques, including and revolving around the Canny edge detection algorithm, to accurately detect and segment backscatter particles. To evaluate the system, the paper develops a bubble backscatter simulator, and a lossless video recorder for controlled testing and real-world footage analysis. Performance tests revealed an average frame processing latency of 2.6 ms, outperforming systems in previous work that operate on more powerful hardware. Attempts to enhance performance using multiprocessing and a real-time operating system (RTOS) patch, however, resulted in increased latency due to the overhead of Inter-Process Communication (IPC) and frequent kernel context switching. These findings suggest that simpler single-core implementations may offer superior performance for I/O-bound tasks. The results demonstrate significant progress in reducing underwater backscatter, with potential applications across various underwater imaging tasks. Future work will focus on hardware improvements and further software optimisation to refine system performance.

Ethical Statement

After consideration of the University of York's code of practice and principles for good ethical governance, I have identified no related issues in this project.

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my parents for their unwavering support: Amma & Appa, I am eternally grateful and indebted for the sacrifices you have made for me and the guidance you have blessed me with, I owe my successes to the both of you.

I'm extremely grateful to my supervisors Paul Mitchell and Andy Tyrrell, without whom this endeavour would not have been possible, for their support and guidance throughout this project.

I could not have undertaken this journey without the support, guidance, and mentoring of Benjamin Henson, an underwater researcher at the Institute for Safe Autonomy.

I'm also grateful for my friends, providing all the entertainment and core memories that I'll cherish for my lifetime.

And the PostDocs at the Institute for Safe Autonomy, you guys are a blast to work with.

Contents

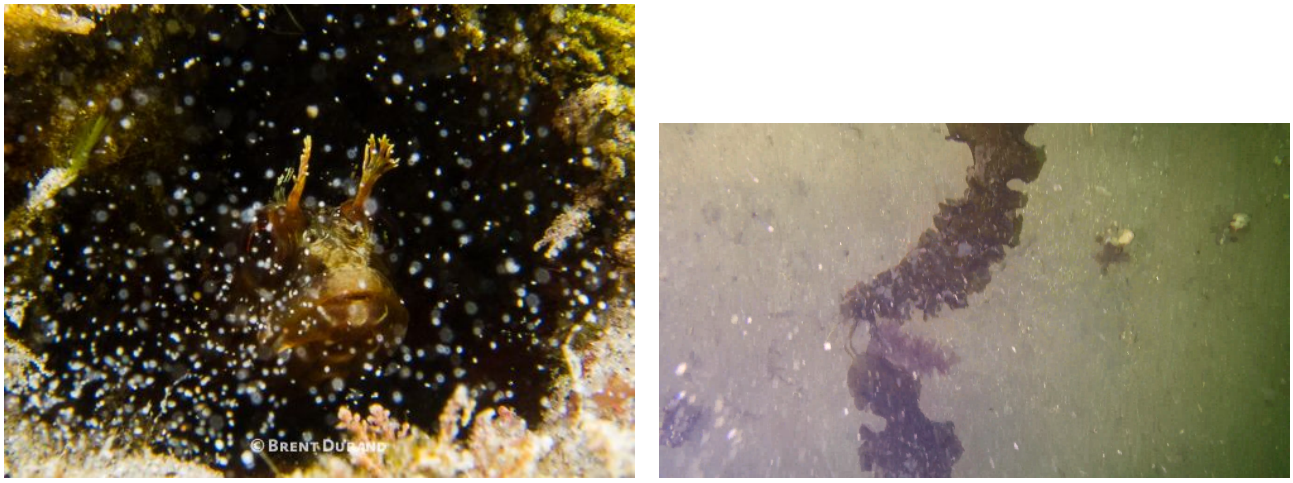
List of Figures

List of Tables

1 Introduction

Underwater imaging capabilities are of great importance across a wide spectrum of disciplines, for instance, marine research and environmental analysis, where scientists dive to several reef locations with waterproof camera systems and quadrats to audit the abundance of coral over time [1]. Unmanned Underwater Vehicles (UUVs), a class of submersible vehicles, are pivotal in advancing these capabilities. With vast arrays of sensors and often remote-controlled or completely autonomous, UUVs enable the end user to conduct exploration missions of extended duration to analyse underwater environments with utmost accuracy even in the most hazardous conditions, impossible for direct human access. These benefits have caused UUVs to become common as a safer and cheaper alternative to manned vehicular operations in almost all underwater imaging-related applications, such as intelligence surveillance and reconnaissance in defence, defect and foreign object inspection in maritime, and oceanography and hydrography in marine research [2].

When capturing images or recording video underwater, one would instantly notice the lack of light at greater sea depths. Coupling a high-power light source next to the camera resolves this issue to ensure a well-lit scene. However, this produces an adverse side-effect called backscatter, shown in Figure 1, where suspended particles in water scatter light in an inhomogeneous manner, reflecting the light emitted by the light source back into the camera, creating exceptionally bright spots and often saturating the image and degrading the quality [3]. Albeit the existence of a few simple and universal techniques to mitigate backscatter, such as bringing the camera closer to the subject, fine-tuning the headlight position such that the subject is illuminated by only the edge of the light cone, or achieving perfect buoyancy to minimise disrupting sand, debris, and bubbles [4], they all lack viability for UUVs due the continuous and arbitrary propellor motion and the constant existence of backscatter-inducing debris floating throughout water bodies.



(a) (b)

Figure 1: (a) Backscatter appears as the particles of sand drift between the aquatic animal and the camera, retrieved from [4]. (b) A captured frame in GoPro footage from a UUV of the seabed with backscatter increasing as the propellors disrupt the sand.

With a specialised camera sensor to capture fast-moving backscatter particles, a single-board computer processing frames using advanced machine vision technologies to detect backscatter, and a specialised projector to project selectively illuminated light patterns, the project's ultimate goal is to develop a novel backscatter-cancelling light source to aid the generation of high-quality underwater images in real-time, thus eliminating the requirement for a camera with better dynamic range compatibility to compensate for the bright regions from backscatter and lens flares. The first objective is to research architectures to develop a reliable backscatter detection system, tieing in with the second objective of researching methodologies to optimise for real-time to ensure predictability and stability, ensuring the projection of backscatter-cancelling light patterns with minimal and consistent latency.

Section 2 summarises the information in related theoretical realms to form the foundation of the subsequent sections. Section 3 first introduces the assumptions and requirements before detailing the high-level processes for both the lighting system and toolsets for testing. Section 4 showcases the system and toolset implementations, which Section 5 then quantifies by evaluating the real-time performances. Section 6 reflects on these evaluations, including a reflection on project execution and a discussion of future work.

2 Background Information

An accurate and real-time backscatter-cancelling lighting system must consider five main factors: (a) precise backscatter particle segmentation even in varying environmental conditions by mapping the exact backscatter positions to aid in (b), the accurate projection of the backscatter cancelling light patterns, ensuring the whole scene except backscatter is lit, (c) low system latency by minimising the time it takes between frame capture and the light pattern projection, ensuring the projection is still correct even with fast-moving backscatter, interlinked with (d), adequate system throughput with parallel processing to increase system responsiveness for reaching the low latency goal, and finally, (e) the ability to control and establish the system frame rate to ensure system predictability.

This section aims to gather background information on the five aforementioned system factors. Beginning with an investigation of previous work on this project in Section 2.1, followed by drawing comparisons and exploring limitations with Section 2.2, which introduces work for an automotive-aimed system. Section 2.3 explores work related to underwater imaging for precise environmental analysis, drawing on a new approach for backscatter segmentation and researching methodologies for tracking. Section 2.4 studies computer systems and architectures for this project, also discussing approaches to mitigate real-time limitations from previous work. Finally, a summary in Section 2.5 consolidates all explored aspects, linking to the wider context and constraints with a decision-making description of the chosen technologies.

2.1 Previous Work in Underwater Anti-Backscatter Lighting Systems

Previous work on this project in [5] by Shepherd introduces a backscatter-cancelling lighting system with goals similar to what this paper proposes. Shepherd presents a two-stage process, which is fundamental to their system, illustrated in Figure 2. The first stage is 'detection', where the specialised projector illuminates the whole scene with a low-brightness, solid white output intending to forcefully induce backscatter particles for detection with machine vision techniques. The second stage is 'pattern projection', where the system overlays black ellipses called 'holes' at the positions of detected backscatter particles over a full-brightness, solid white projection to eliminate backscatter.

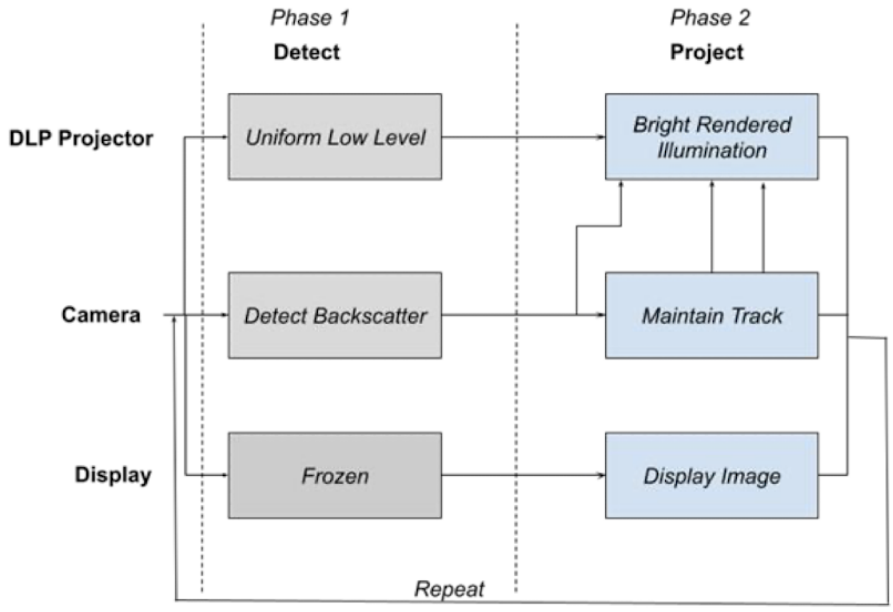
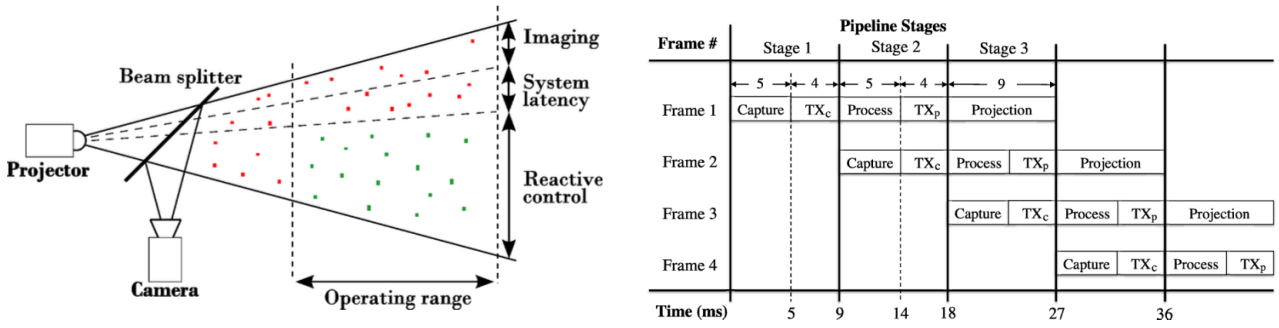


Figure 2: An illustration from the paper in [5] showing the high-level system stages.

Shepherd designs the backscatter segmentation system based on a simple blob detection algorithm [6], where the term 'blob' denotes a group of connected pixels in a binary image. By applying several thresholds, the algorithm first converts the input image into a binary representation to then extract connected pixels and calculate the centre positions of each. After grouping the centre positions from the binary images, the algorithm finally estimates the final centroid and radii for each 'blob'. While this blob detection algorithm benefits from a computationally simple approach, it suffers from a drastic drawback as Shepherd describes "at least one property needs to be shared between the blobs such as size or shape to allow for the isolation of these blobs". The underwater environment is unpredictable, where backscatter particles can never share the same characteristics, making this approach unviable.

2.2 Automotive Headlights for Illumination Through Rain and Snow

Similar to the underwater backscatter that this paper explores and attempts to eliminate, the work in [7] by De Charette et al. outlines an automotive headlight system to illuminate around the backscatter caused by rain and snow whilst driving. The parallax issue observed by Shepherd, where the displacement between the camera's and projector's viewpoint causes a perceived offset in object positioning, is resolved in De Charette et al. using a beamsplitter arrangement, illustrated in Figure 3a, such that the incident visual beam is split 50:50 between the projector and camera, thus eliminating parallax displacement by co-location.



(a) (b)

Figure 3: From [7]: (a) An illustration of the system configuration including a beamsplitter for parallax elimination through co-location, (b) a visualisation of the parallel pipeline stages for their system.

The automotive backscatter-cancelling headlight system utilises an image processing pipeline that applies a background subtraction, albeit not specifying the exact method, followed by blurring and thresholding to isolate the bright water spots. The system employs the connected-components algorithm to segment the backscatter particles, this algorithm scans an image to group its pixels into components based on pixel connectivity, i.e., all pixels in a connected component share similar pixel intensity values and are in some way connected [8]. Unlike Shepherd's system which employs a blob detection algorithm, this system is resilient to unique backscatter particle characteristics, allowing for segmentation regardless of fixed parameters despite also being a computationally simple approach. However, the connected-components algorithm may be inherently limited when given low-quality input images as it may not correctly connect pixels due to distortion and noise.

Although not specifying the exact methods, the automotive system in De Charette et al. employs backscatter particle tracking with a predictive algorithm to project backscatter-cancelling light patterns by accounting for the movement during the system processing period, denoted by the system latency time. Due to the linear and vertical movement of rain and snowfall, one can assume they are using a simple linear interpolative approach with the assumption the particles are moving at a constant speed, to predict the future particle position. To increase system throughput, the work in De Charette et al. proposes a three-stage parallel processing pipeline, illustrated in Figure 3b. Stage 1 represents the task of capturing the frame from the

camera, with TX_c denoting the image data transfer from the camera to the computer. Stage 2 represents the image processing steps, with drop detection, prediction, and projection pattern generation, with TX_p denoting the computer-to-projector data transfer. The third and final stage represents the refresh time of the projector. This system, once primed, will run with a three-frame latency: as the n^{th} frame is being captured, the $n^{th} - 1$ frame is being processed, and the $n^{th} - 2$ frame is being projected. While an increased latency is counter-intuitive, the delay effect is heavily mitigated by the predictive functionality, which can be further improved by precisely clamping the execution and TX duration for each stage.

The system in De Charette et al. takes, on average across 5500 frames, 4.214 ms to transfer data from the camera to the host computer, a 3.2 GHz Intel Xeon processor with 8 GB of RAM running Windows Vista 64-bit, 4.081 ms to process the image, 4.214 ms to send the data to the projector, and 9 ms for the projector to output. Thus, total system latency is 21.51 ms, with an accuracy of 68.9% for rain. The work in [9] by Tamburo et al. expands on the system by De Charette et al. with a more powerful desktop PC for image processing and an FPGA-based controller for the specialised projector. This system achieves an average frame processing time of 0.3 ms, with a total response time ranging between 1 ms to 2.5 ms.

2.3 Underwater Gas Seepage Bubble Quantification for Environmental Analysis

Aside from the work by Shepherd, De Charette et al., and Tamburo et al., public documentation of research related to real-time efforts for underwater backscatter-cancelling lighting systems is non-existent. However, numerous environmental analysis fields rely on highly specialised and precise systems for quantifying underwater bubbles to investigate gas seepages from the sea floor. As underwater backscatter can comprise a mixture of compositions, ranging from bubbles and sand to all sorts of marine debris, it will be beneficial to explore gas escape measurement and monitoring systems as they require high precision, extensive range, strong anti-interference, and low cost under complex underwater conditions [10].

The work in [11] by Thomanek et al. presents a novel approach for a highly precise automated gas bubble imaging system that employs the Canny edge detection algorithm. The Canny algorithm, which prefers a single channel (greyscale) input, first smoothens the input using Gaussian convolution before comparing each pixel's values relative to its neighbours such that when the gradient is above a certain threshold, it sets the bordering pixel a value of '1', otherwise '0', resulting in the formation of edges around objects [12]. In contrast to the system by Shepherd, the Canny approach in Thomanek et al. works around the object characteristics limitation of the simple blob detection algorithm.

The paper compares the Canny algorithm approach with the simple image binary thresholding method, where only the pixels with intensities within a certain threshold are passed through, ultimately establishing Canny as the more accurate choice for segmentation, even in areas of

uneven illumination, illustrated in Figure 4. However, the drawbacks of the Canny approach, as highlighted by Thomanek et al., include a significant increase in computing time, and the necessity for implementing morphological techniques to address bubble transformation issues such as expansion and contraction during segmentation. Nevertheless, considering the advancements in computing power over the past 14 years since this paper was published, modern computers should now be easily capable of handling this workload with relative ease.

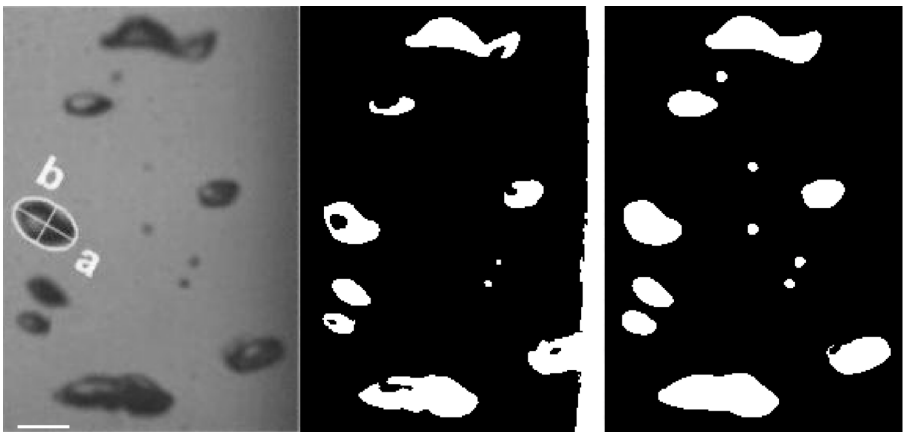


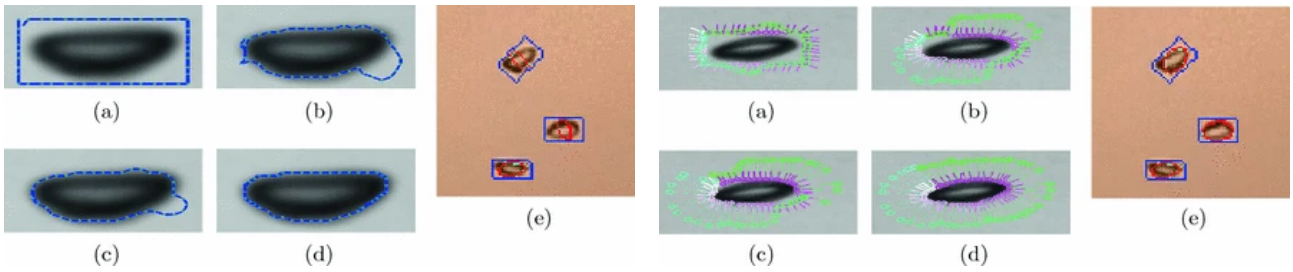
Figure 4: An illustration from the paper in [11] showing the original input frame in the left-most image, bubble segmentation using simple binary thresholding in the middle, and bubble segmentation using the Canny edge detection algorithm in the right-most image.

While this paper by Thomanek et al. provides valuable insight and serves as an excellent starting point, it primarily emphasises system construction and quantifying gas flux measurements. Consequently, certain crucial details, such as the precise methodology employed for achieving the highlighted bubbles post-Canny segmentation in Figure 4, receive less attention. However, one can infer that Thomanek et al. may have employed a logical algorithm to fill closed-loop edges. The study by Zelenka in [13] builds upon the foundational system by Thomanek et al. aiming to address the sporadic false detection issue that is inherent in the Canny edge detection algorithm, introducing robust algorithms capable of achieving precise bubble stream quantification with accurate bubble fitting.

Zelenka introduces a snake-based active contour model developed by Kass et al. in [14] for a more stable and precise method for bubble detection. As explained by Kass et al., a snake embodies an energy-minimising spline that's guided by external forces and influenced by image characteristics that collectively drive the snake towards prominent features within the image, such as lines and edges, all while the snake dynamically adheres to nearby edges due to the active contour model. The classic snake method, illustrated in Figure 5a, uses a bounding box from the baseline localisation for initialisation, shown in blue, from which the snake algorithm optimises with each iteration on a low-quality image sequence. The figure exemplifies the unreliability of the classic snake method due to light areas within the centre of the bubble, as the red outline borders this region instead of the bubble itself. Zelenka, therefore, introduces a novel enhancement to the classic snake method to improve performance in varying lighting

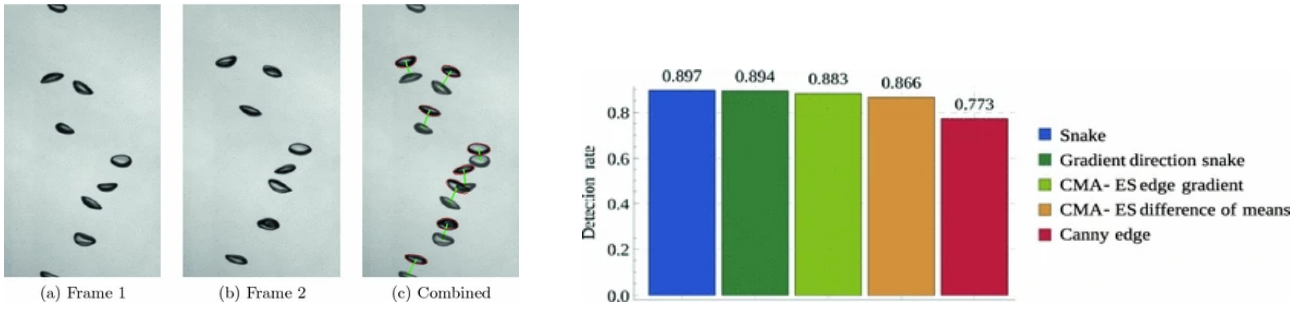
conditions by utilising gradient information of an image to compute external energy terms that guide the snake.

Zelenka also employs the Covariance Matrix Adaptation - Evolution Strategy (CMA-ES) based approach for ellipse fitting in the paper, The CMA-ES algorithm begins by initialising ellipse parameters and defining an objective function to evaluate the fit, then iteratively updates a distribution over the parameter space, generating candidate solutions and optimising the fit by adjusting the distribution parameters, repeating until convergence, yielding the parameters of the best-fitting ellipse. Zelenka compares all of these bubble segmentation methods using a sequence of 20 GoPro-captured images, with circa 10 bubbles visible per frame and manual ground truth. The results, illustrated in Figure 6b, show a clear improvement across all methods compared to the baseline Canny approach, which achieved a detection rate of 77.3%. The classic snake approach achieved the best detection rate of 89.7% and the worst rate was achieved with the CMA-ES approach.



(a) Classic snake algorithm. (b) Gradient-based snake algorithm.

Figure 5: An illustration of the classic snake method in (a) and the gradient snake method in (a) from the work in [13]. Sub-subfigures from (a) to (d) for both methods show every third step in the snake optimisation process, and (e) shows the final result.



(a) (b)

Figure 6: From [13]: (a) demonstration of the system across two subsequent frames at 100 FPS, with the red border denoting bubble detection and the green lines denoting bubble tracking, (b) a comparison of bubble detection rates for different segmentation approaches.

Thomanek et al. introduced a novel approach for bubble tracking, leveraging the 'least distance assumption'. This assumption entails realising that the distance travelled by a bubble between two successive frames is typically smaller than the distance to its closest neighbouring bubble. While this assumption may not hold for scenarios involving overlapping bubbles, extremely high

bubble concentrations, or cases where the travel distance exceeds the distances to neighbouring bubbles, it, however, facilitates a computationally straightforward method for determining bubble positions in successive frames and estimating the bubble rise velocity from the sea floor. Despite the computation simplicity, this method may not present a fair trade-off due to the backscatter originating from bubbles, which can be highly concentrated, overlapping, and subject to rapid movement, especially in the presence of UUV propellers. Zelenka's paper introduces a novel approach for tracking bubbles by employing a Kalman filter [15] to predict bubble positions in subsequent frames based on initial detections. The Kalman filter utilises a recursive algorithm to estimate the state of a dynamic system, in this case, the bubble's position, by incorporating both prediction and measurement update steps. Additionally utilising, the Hungarian method [16], which finds the optimal assignment of a set of tasks to a set of agents, minimising the total cost or maximising the total profit of the assignments, for minimum weighted matching between predicted and newly detected bubble positions, optimising the association process. While it remains uncertain whether the Kalman filter-based method is immune to the limitations associated with the 'least distance assumption', Zelenka does validate its capacity as a highly reliable tracking mechanism. An illustration of the whole system from Zelenka is illustrated in Figure 6a, showing both bubble detection and tracking.

2.4 System Building Blocks

2.4.1 Real-Time Linux Kernel with the PREEMPT-RT Patch

Shepherd's work highlighted the necessity of employing a Real-Time Operating System (RTOS) to mitigate the jittering effect. This phenomenon led to unpredictable bursts of reduced frame rates and lag. An RTOS facilitates the compliance of a 'hard' real-time system, which can provide a guarantee of the maximum time a task necessitates for completion. Although RTOS products are available for most computer systems, they often come with numerous limitations that necessitate circumvention, especially when compared to a general-purpose operating system like Linux. Similar to Shepherd's system, in Linux, when a task in user space, where normal user processes operate, is interrupted, the scheduler will preemptively schedule another task if the interrupt handler can awaken it once it returns. However, many sections in kernel space, where the kernel's code and data reside and execute, do not support preemption due to the presence of spinlocks. Spinlocks are non-blockable and non-sleepable programmatic loops that safeguard critical sections of code. Consequently, the preemption logic breaks down when a user space task invokes a kernel-specific function, thereby transitioning to kernel space upon receiving an interrupt.

Figure 7, depicted in the material from [17] by Bootlin, showcases the challenge of user space task promotions to kernel space, highlighting how kernel-based spinlocks contribute to unpredictable jitter, as denoted by the green question mark. Apart from the kernel space incompatibility, Linux already facilitates user space preemption through task priority-based real-time scheduling, enabling RTOS functionality. The PREEMPT-RT kernel patch modifies all kernel-

space code, allowing it to become preemptible and deterministic. While insufficient in transforming Linux into a 'hard' RTOS, the PREEMPT-RT patch represents a positive stride toward minimizing task-switching latencies. Using a test suite that contains programs to test various real-time Linux parameters, the material in [18] quantifies system latencies for the PREEMPT-RT patch, drawing comparisons with the non-RT kernel. The results are in Figure 8, displaying the maximum latency measurements obtained with the standard kernel, registering at 301 µs, whereas the PREEMPT-RT kernel yielded a significantly reduced latency of 83 µs. This confirms a notable 3.63x reduction in latency achieved through the implementation of the kernel patch.

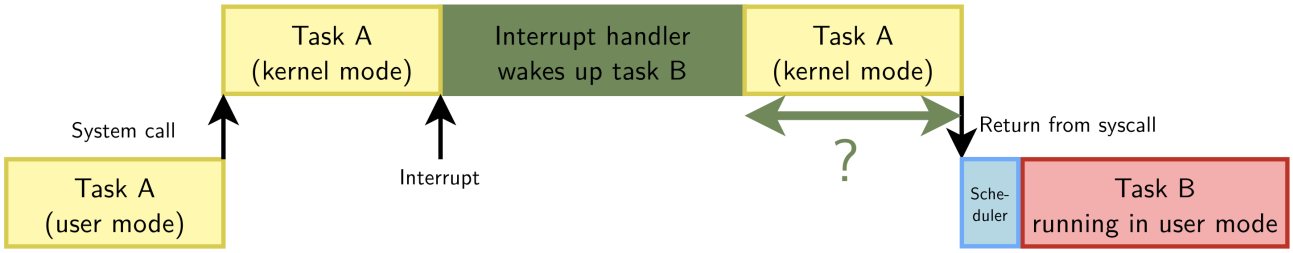
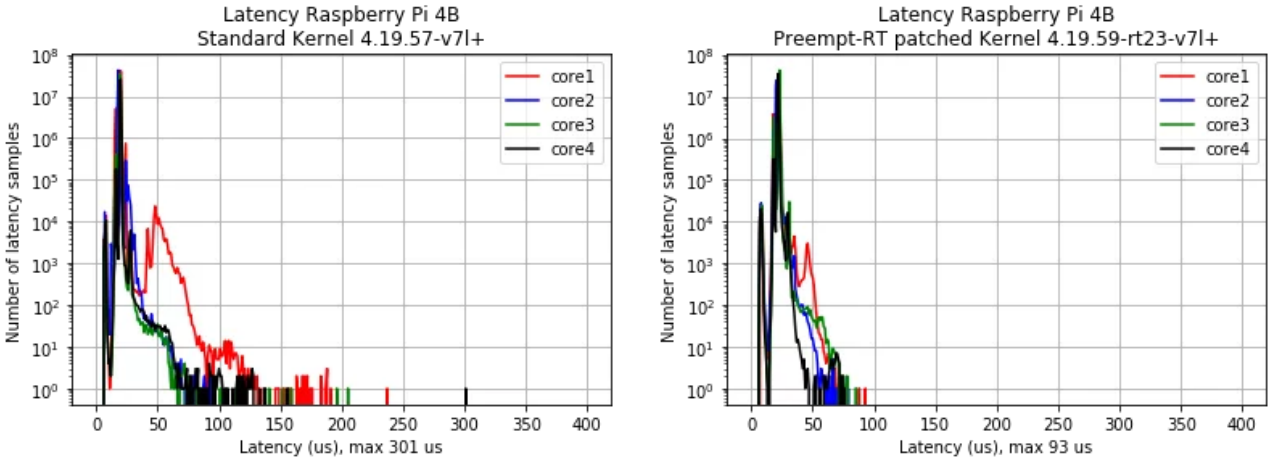


Figure 7: An illustration from the paper material in [17] of kernel-space latency from an interrupt in the Linux kernel.



(a) (b)

Figure 8: Illustration from [18] presents the latency measurements obtained following a Cyclictest execution from the RT-Tests suite. It contrasts the results between (a) the standard Linux RPi kernel and (b) the Linux RPi kernel with the PREEMPT-RT patch applied.

2.4.2 Computing Platforms

Field Programmable Gate Arrays (FPGAs), semiconductor devices structured around a matrix of configurable logic blocks (CLBs) interconnected via programmable interconnects [19], are ideal solutions for computationally demanding image processing tasks, owing to their inherent hardware parallelism and low latency characteristics. In contrast, CPUs execute computations sequentially, processing algorithms step by step, restricting execution to one operation at a time. Despite the availability of highly efficient intellectual property (IP) cores for FPGA-based logic

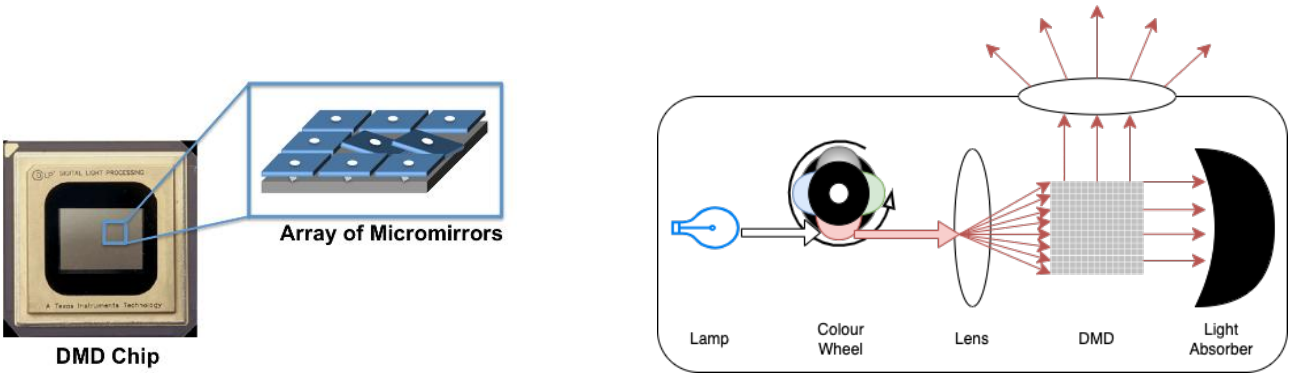
acceleration, development and prototyping times may increase due to the intricate low-level hardware intricacies involved. Additionally, FPGAs typically incur higher costs compared to traditional CPU-based computing systems.

On the other hand, the Raspberry Pi (RPi) company has been at the forefront of designing high-performance, cost-effective, single-board, and modular computers based on the Arm architecture and running the Linux operating system since 2012 [20]. The standard RPi product lineup provides an excellent non-FPGA, CPU-based route for swift and straightforward development and deployment. Recent RPi models boast a strong feature set, including up to 8GB of onboard RAM, dedicated GPU and video decoding capabilities, HDMI output interfaces, dual-band WiFi with Bluetooth support, Gigabit Ethernet connectivity, USB 3.0 and 2.0 ports, MIPI-compatible interfaces for cameras and displays, and an array of 40 GPIO pins.

2.4.3 Specialised Light Source

A Digital Light Processing (DLP) projector plays a crucial role in projecting backscatter-cancelling light patterns. DLP chipset, developed by Larry Hornbeck of Texas Instruments in 1987, comprises a Digital Micromirror Device (DMD), which houses millions of reflective aluminium mirrors, typically only a few microns wide. The DMD serves as a Micro-Opto-Electro-Mechanical System (MOEMS) Spatial Light Modulator (SLM), utilising digital signals to precisely control the angle of each mirror. This capability enables the modulation and attenuation of an incident light beam with remarkable precision.

Figure 9 illustrates how a DLP projector employs this specialised technology: Initially, a beam emitted from a high-intensity white lamp is directed towards a rotating colour-tinted lens wheel, which typically includes red, green, and blue filters, as well as a clear lens for unfiltered light passage. Subsequently, an internal lens diffracts the coloured beam and directs it onto the DMD. Here, the angle and duration of each microscopic mirror's activation dictate the colour and intensity of individual pixels. The DMD selectively redirects the desired beams of each pixel through a diffraction lens to exit the projector. In instances where a particular pixel's beam is not required, the corresponding mirrors on the DMD remain inactive, directing the beam into a light-absorbing region to prevent leakage and ensure image integrity.

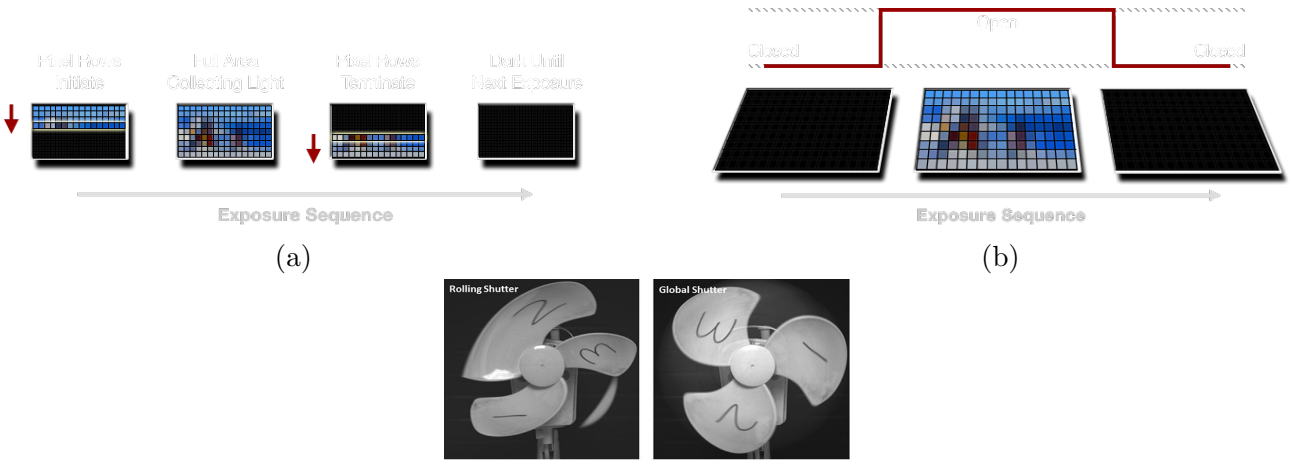


(a) (b)

Figure 9: (a) An illustration of the microscopic mirror array within a DMD chip [21]. (b) A cross-sectional view illustrating the internals of an example projector employing DLP technology.

2.4.4 Specialised Camera Sensor

The cameras found in many traditional and consumer-grade electronic devices, including smartphones, typically utilise a CMOS sensor equipped with a rolling shutter. Although these sensors are smaller and more cost-effective, they introduce distortion effects when capturing fast-moving subjects, attributed to their line-by-line scan image-capturing characteristic. Unlike the line-by-line scanning characteristic of a rolling shutter, a global shutter sensor captures a snapshot of the scene using all pixels simultaneously, rather than activating pixels sequentially from the top of the sensor and working its way down. Figures 10a and 10b depict the distinctions in image capture between a rolling shutter and a global shutter, while Figure 10c showcases the significant disparities between each shutter type. The Raspberry Pi company offers a global shutter camera equipped with a 1.6MP Sony IMX296 sensor, boasting plug-and-play compatibility with Raspberry Pi computers.



(a) (b)

(c)

Figure 10: The image capture of a rolling shutter sensor in (a) and that of a global shutter in (b) [22], and (c), the comparison between the sensors when capturing a fast-moving target [23].

2.4.5 The Python Programming Language and OpenCV

OpenCV is an open-source computer vision and machine learning software library with over 2500 optimised algorithms, including a comprehensive set of classic and state-of-the-art computer vision and machine learning algorithms [24]. Originally developed in C++, OpenCV offers interfaces for Python, Java, and MATLAB, ensuring compatibility across a wide range of platforms. It targets real-time vision applications by leveraging CPU accelerators like MMX and SSE instructions for parallel data processing, as well as GPU accelerators. OpenCV provides simple and high-level wrappers for advanced computer vision algorithms and tools, therefore, significantly reducing development and prototyping time with minimal performance trade-offs. Python, with its easy-to-read and write syntax and robust debugging capabilities, serves as an ideal programming language for interfacing with OpenCV and implementing computer vision tasks.

2.5 Summary

From the background research, the best methodology for precise backscatter segmentation is the Canny edge detection approach due to its resilience in varying particle appearances, a limitation of Shepherd's simple blob detection approach, and robustness in detecting edges even in uneven illumination conditions, exemplified by Thomanek et al. drawing comparisons with simple image thresholding. Zelenka's use of the gradient-based Snake method, despite its capability to segment bubbles with bright spots inside them, is unnecessary for this project's objective, as the focus lies on eliminating just the backscatter within sediments, the bright spot within bubbles, sand, and other marine debris, rather than the entire sediment particle.

When considering the computing platform, a Raspberry Pi single-board computer (SBC) running a Linux OS, and developing the system with Python and OpenCV, is preferable over an FPGA for its cost-effectiveness, ease of development, and compatibility with a wide range of peripherals and software, aspects which are all essential for this project for prototyping in short time constraints. While a Raspberry Pi SBC that runs an ordinary Linux OS is not inherently a 'hard' real-time system like an FPGA implementation, implementing the PREEMPT-RT Linux kernel patch offers improved determinism and reduced latency compared to a standard Linux OS, crucial for ensuring timely and predictable response in the system.

A DLP projector is the preferred choice for projecting backscatter-cancelling light patterns due to its precise control and modulation capabilities, essential for accurately overlaying patterns to eliminate backscatter. Opting for a global shutter camera sensor over a rolling shutter sensor mitigates distortion effects caused by fast-moving subjects, ensuring accurate image capture in dynamic underwater environments.

3 Design

This section aims to provide an overview of the system's equipment and considerations driving the system and toolset designs, along with the approaches to overcome technical challenges. Beginning with an exploration of the hardware setup and the constraints established by Shepherd, Section 3.1 outlines the key components of the prototype system, identifying the requirements and high-level designs for system tools and the system itself. Section 3.2 introduces the requirements and designs for a tool to develop a backscatter-generating simulation for a synthetic ground truth to verify the system, and Section 3.3 discusses that for a lossless test footage recording tool for the system, to generate physical, real-life, testing data. Finally, Section 3.4 provides a holistic understanding of the system's architecture and design philosophy, including image processing pipelines and optimisations for real-time, setting the stage for the subsequent implementation and testing phases.

3.1 System Equipment & Design Constraints

The system follows from the hardware constraints and assumptions that were set by Shepherd. In this prototype system, illustrated in Figure 11, two custom-fabricated and watertight tubes are fastened with a plywood plate. The larger of the two tubes houses the Optoma ML550 projector, which features an LED lamp enabling a brightness capability of up to 500 lm, illuminating a 1 cm DLP for a native resolution of 1280x800, with a maximum 120 Hz vertical and 100 kHz horizontal scan rate capabilities. The smaller tube houses the RPi 5 (8 GB RAM variant) SBC, using a Broadcom BCM2712 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU to feature an advertised 2-3x performance upgrade from the RPi 4 that Shepherd used.

Connected to this RPi, via a 4-lane Mobile Industry Processor Interface (MIPI) transceiver, is an RPi global shutter (GS) camera, featuring the Sony IMX296LQR-C 1.58MP sensor. Instead of a GS sensor, Shepherd employed the RPi High-Quality (HQ) sensor, however, due to the reasons which Section 3.3 outlines, the GS sensor replaces this. The GS camera uses a PT361060M3MP12 lens, with an adjustable F1.2 aperture, a 6 mm adjustable focal length for a minimum object distance of 0.2 m, and a field of view of 63°. All components are mounted within their respective tubes using 3D-printed mounts, such that the separation between the camera and projector lenses is 12 cm. At the backside of each tube is a metallic cap with watertight cable couplers to pass through the RPi and projector power cables, and an ethernet cable for the RPi.

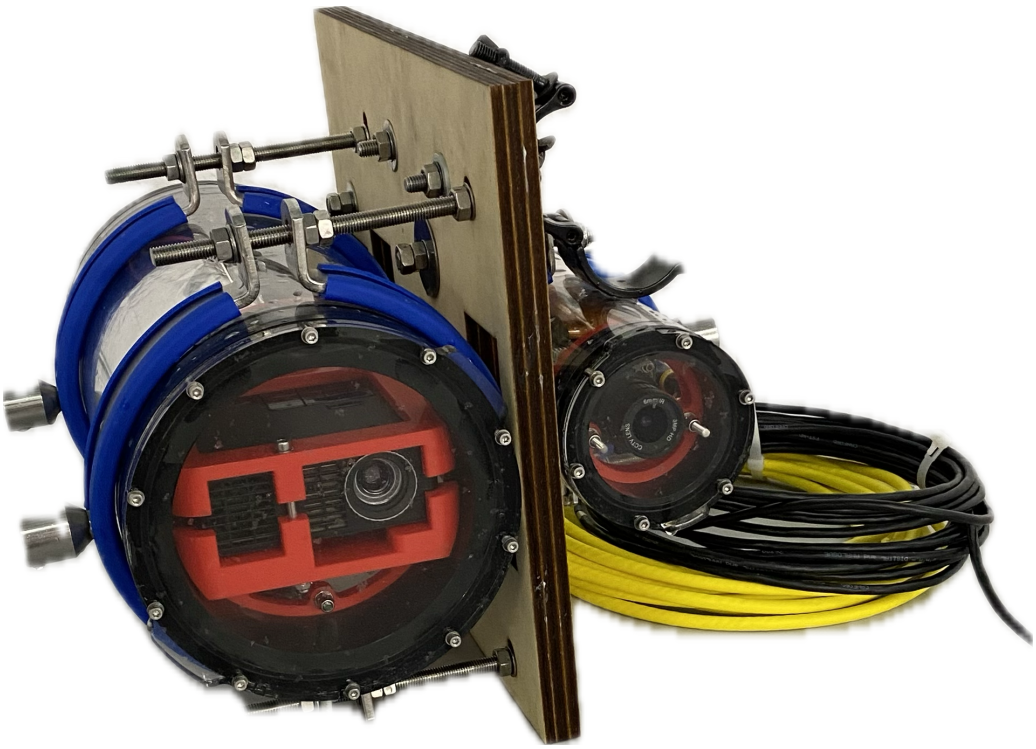


Figure 11: A picture of the submersible system.

The design relies on two assumptions: (a) the system is to run below the ocean twilight zone, which is approximately 200 m below the surface, ensuring the scene is only lit by the system and not other sources such as sunlight, and (b) there are no internal reflections from the housing, such as from the plastic tubes and tube lenses, ensuring all noticeable reflection to be backscatter particles. Due to its watertight construction, disassembling and reassembling the housing is a laborious process, often requiring the reapplication and testing of seals, which can extend the downtime to multiple days. Therefore, conducting tests using pre-recorded footage or synthetic simulations of backscatter is preferred to minimise disruptions and maintain operational efficiency.

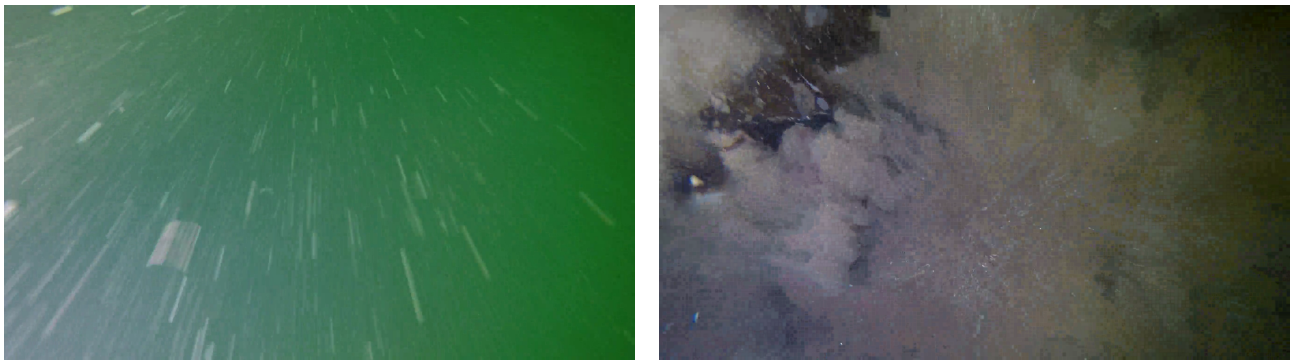
3.2 Simulating Backscatter for Synthetic Ground Truth

Given the substantial weight of the submersible, manoeuvring the system into position within an underwater testing tank presents significant challenges during testing. Moreover, the availability of the underwater testing tank at the Institute for Safe Autonomy wasn't fully realised until the later stages of this project. Consequently, there is a need for software capable of synthetically generating a simulation of backscatter particles. Since this simulation is entirely software-based, it must produce ground truth data comprising a dataset, assigning each backscatter particle with a unique ID, allowing for the tracking of coordinates across every frame of the simulation. Comparing the detected backscatter positions from the system with the true positions derived from the simulation ground truth enables the accurate assessment of the system's accuracy and performance.

Employing a straightforward model involving bubbles rising from the bottom of the screen can form a sufficient simulation of backscatter particles in all axis directions: horizontal (x), vertical (y), and towards the camera (z). Each bubble, representing a backscatter particle, can be depicted as a white circle against a black background for simplicity, with user control over these exact colours. The model should also take into account the bubble expansion due to the pressure difference as they ascend underwater. As a result, a separate model to simulate bubble distance from the camera location may not be necessary if the bubble's radius adjustment based on height is implemented effectively. It's also crucial to model horizontal bubble movement to capture the full range of particle motion.

3.3 Recording Lossless Video for Physical Testing

At the start of this project, Ben provided me with a GoPro recording from a submersible capturing the seabed. This footage was intended for use as test material to develop the system. However, due to the heavy video compression applied by the GoPro, coupled with its rolling shutter sensor, the video contains a large number of compression artefacts and significant motion blur, as illustrated in Figure 12. These issues significantly complicate the task of identifying individual backscatter particles for segmentation. Where the system in Figure 11 originally employed an RPi HQ sensor, which is a rolling-shutter Sony IMX477R stacked, back-illuminated sensor with 12.3 megapixels, in Shepherd's work, the limitations posed by the GoPro footage proved the necessity for a GS sensor, prompting the exploration and subsequent implementation of the RPi GS camera. In order to capture test footage using this system, there must be a program in place to record the 'raw' footage from the camera, directly from the sensor inside the submersible without any lossy compression.



(a) (b)

Figure 12: Frames from the GoPro footage: (a) showing the motion blur of backscatter particles, and (b) showing the pixelation artefacts (depending on your display, you may have to look closely to see the pixelation effect).

When recording in raw format, the RPi GS camera captures in full 1456x1088 sensor resolution at the maximum compatible framerate of 60 frames per second, using the native RAW10 SBGGR10 Bayer format. The SBGGR10 format is constructed of four channels, one red, one blue, and two green, with a 10-bit depth for each. Equation 1 calculates the total number of

bytes per frame and Equation 2 calculates the total number of bytes per second when recording in this raw format, where R_w and R_h are the resolution width and height, D_b is the bit depth per channel, N_{ch} is the number of channels, F_b is the number of bits and F_B the number of bytes per frame, F_{rate} is the frame rate, and finally, S_b is the number of bits and S_B the number of bytes per second.

$$\begin{aligned} F_b &= (R_w \times R_h \times D_b \times N_{ch}) \\ &= (1456 \times 1088 \times 10 \times 4) \\ F_b &= 63\,365\,120\,\text{bit} = 63.365\,12\,\text{Mbit} \\ F_B &= 7\,920\,640\,\text{B} = 7.920\,64\,\text{MB} \end{aligned} \tag{1}$$

$$\begin{aligned} S_b &= (F_b \times F_{rate}) \\ &= (63365120 \times 60) \\ S_b &= 3\,801\,907\,200\,\text{bit} = 3.801\,907\,2\,\text{Gbit} \\ S_B &= 475\,238\,400\,\text{B} = 475.2384\,\text{MB} \end{aligned} \tag{2}$$

The calculations reveal an approximate transfer rate of 475 MB per second when recording the raw footage, presenting a significant challenge due to the substantial data throughput requirement. It would be impossible to transfer at that data rate to the boot drive due to CPU bottlenecks, and most importantly, boot drive write-speed bottlenecks, with the drive being a USB 3.1 stick, despite being a faster option than the default RPi SBC SD-card-based boot drive. To address this challenge, it is imperative to implement a memory buffer in RAM to temporarily store the recording data. This buffer will only offload data to the boot drive once the recording concludes ensuring the CPU is solely reserved for recording. Additionally, employing lossless encoding techniques is essential to reduce the file size, and to convert the recording footage from the Bayer SBGGR10 format to a format natively supported on a wide range of systems ensuring easy file transfers and playback. Equally important is the inclusion of functionality within the recording system to modify resolution for downscaling and to adjust the framerate, effectively further reducing the output filesize without introducing video artefacts, ultimately enhancing the overall efficiency of the system.

3.4 Underwater Backscatter Cancellation System

The introduction to Section 2 mentions five main factors for an accurate and real-time backscatter cancellation lighting system. To consider the first factor, which is the precise backscatter particle segmentation even in varying environmental conditions by mapping the exact backscatter positions, there must be an image processing pipeline, Figure 13 illustrates this pipeline from a high-level perspective. The pipeline must begin with a greyscale filtering stage to reduce im-

age dimensionality by categorising pixels by brightness, ensuring a single-channel output for reduced processing complexity in the subsequent pipeline stages. The next stage applies a Gaussian blur to reduce noise and small-scale variations in pixel intensities, to improve the edge detection performance such that only edges of interest are detected by the algorithm. Post-noise-smoothening, a histogram equalisation stage can improve the dynamic range, enhancing image contrast by redistributing pixel intensities to fully utilise the intensity range. The histogram equalisation stage, albeit theoretically, can drastically improve the edge detection algorithm due to the improved contrast between objects of interest. Application of the Canny algorithm will extract all edges, ideally of the backscatter in isolation due to the previous stages, aiding in the next stage, which is backscatter segmentation. The segmentation stage will utilise a simple methodology, such as highlighting closed-loop edges, in order to isolate and pinpoint the backscatter particles.

The next key factors are low system latency and high system throughput, both interlinked to ensure the system is highly responsive, enabling the projection is still accurate by minimising the processing time after the moment of frame capture. Given Python's high-level abstraction, there is a theoretical limit to how close to real-time the system can operate, a limit that is, in theory, significantly higher than a low-level C-based or an FPGA system. For this system, I will be targetting a total processing duration, which is the time it takes to capture the frame, apply the image processing pipeline, and finally segment the backscatter particles, of 33.33 ms, which roughly equates to a framerate of 30 FPS. Employing a parallel processing pipeline, similar to the implementations by De Charette et al. and Tamburo et al., can, again, in theory, increase system throughput. Unfortunately, The Python interpreter is not fully thread-safe and thus utilises the Global Interpreter Lock (GIL), which is a global mutual exclusion lock, to prevent multiple processor threads from executing at once and causing race conditions. As a result, a Python-based processing pipeline will only ever process image frames sequentially, requiring the bypass of GIL to ensure parallel processing. Figure 14 illustrates the process distribution for the parallel image processing pipeline, such that when the system is segmenting the particles in frame #2, frame #3 receives the Canny algorithm, frame #4 receives a histogram equalisation, frame #5 receives a Gaussian blur, and frame #6 receives a greyscale filter, all in parallel while the system captures frame #7.

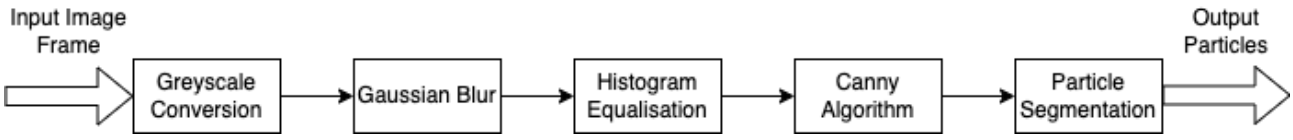


Figure 13: Design of the image processing pipeline for the system.

Parallel Image Processing Pipeline						
Time	Stage 1 Frame Capture	Stage 2 Grayscale	Stage 3 Gaussian Blur	Stage 4 Histogram Equalisation	Stage 5 Canny Algorithm	Stage 6 Particle Segmentation
t	Frame 1					
t + 1	Frame 2	Frame 1				
t + 2	Frame 3	Frame 2	Frame 1			
t + 3	Frame 4	Frame 3	Frame 2	Frame 1		
t + 4	Frame 5	Frame 4	Frame 3	Frame 2	Frame 1	
t + 5	Frame 6	Frame 5	Frame 4	Frame 3	Frame 2	Frame 1
t + 6	Frame 7	Frame 6	Frame 5	Frame 4	Frame 3	Frame 2
t + 7	Frame 8	Frame 7	Frame 6	Frame 5	Frame 4	Frame 3
t + 8	Frame 9	Frame 8	Frame 7	Frame 6	Frame 5	Frame 4
t + 9	Frame 10	Frame 9	Frame 8	Frame 7	Frame 6	Frame 5
...	...	...	...	...	...	...
t + n	Frame n + 5	Frame n + 4	Frame n + 3	Frame n + 2	Frame n + 1	Frame n

Figure 14: Illustration of the design for parallel processing in the image processing pipeline for the system.

In addition to parallel multiprocessing, the system must implement a real-time kernel with the PREEMPT-RT patch. Shepherd's use of the fully-fledged Raspberry Pi OS, which includes a desktop environment and a full software suite, may have been the reason which led to the jitter they experienced, primarily due to the OS scheduler preemption type which prioritises a smooth graphical user interface over system latencies, and also due to the vast number of bundled software with could have been running in the background and consuming CPU time. To ensure minimal OS and software overhead, I will be using the 64-bit Raspberry Pi OS Lite, which does not come with a desktop environment or any non-dependency software. Applying the PREEMPT-RT patch to the OS will, in theory, reduce stage durations due to the reduction in system latency, and additionally, will also reduce latency randomness, ultimately improving system real-time and predictability. The final key factor is the ability to control the system frame rate, essentially by adding a delay to ensure processing at the established rate. For this design, the implementation must track the duration of each stage accurately to compute the necessary delay duration.

4 Implementation

Following Section 3, this section aims to implement the designs, starting with establishing remote communications with the RPi SBC, later developing toolset programs, and finally, the system itself.

4.1 Interfacing With Raspberry Pi SBC

Due to security reasons, also noted by Shepherd, it is not possible to connect the RPi SBC to the Eduroam University network. This incurs a major roadblock, preventing remote communication to the RPi via SSH and communications to the internet from the RPi. Two aspects

will ensure complete and seamless connectivity of the system: (a) SSH connection from the laptop to the RPi to enable remote connectivity for configuration and control, and (b) internet connection from the RPi to a valid network to enable the installation and update of the OS and other software. The security policies in place across all University-owned networks block communication between devices. Therefore, there must be a separate link between the RPi and the device to initiate an SSH connection. As Figure 15 illustrates, SSH communications will occur over an Ethernet connection between the RPi and a personal computer, and internet communications will occur over Wi-Fi to a separate University of York-owned network called 'mydevices', which is intended for internet access from games consoles and specific smart home devices, and can be connected to by registering the RPi SBC's MAC address via a web portal and then entering a default Wi-Fi WPA2 Personal password on the RPi.

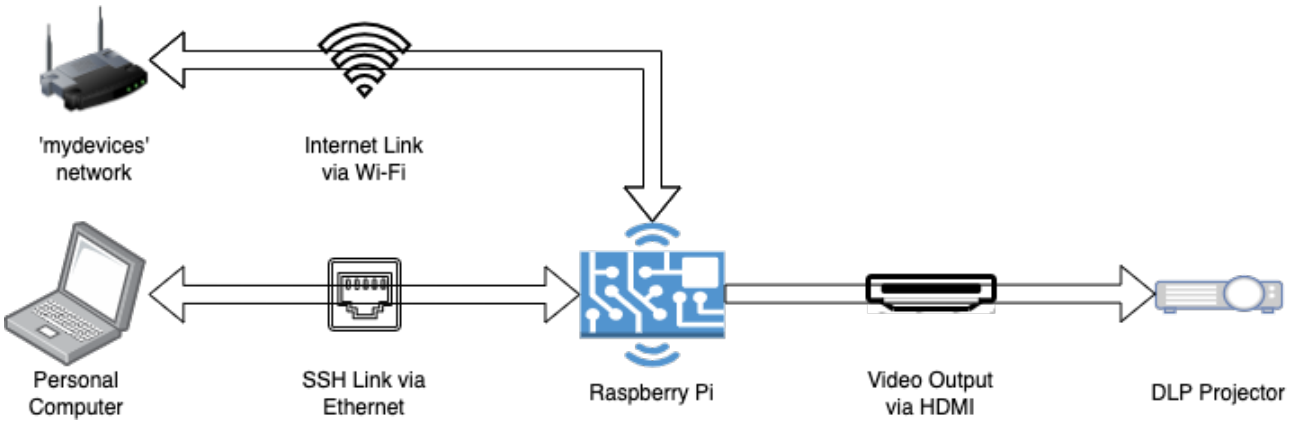


Figure 15: Illustration of the Raspberry Pi SSH and Wi-Fi communication configuration, including video output interfacing with HDMI.

The RPi must have a set hostname for easy identification, and through NetworkManager, configuring 'wlan0' to connect to the 'mydevices' access point for internet access, and to prevent random and intermittent SSH disconnections and unresponsiveness, configuring 'eth0' to a static IP address in the format 192.168.0.xxx/24 and disabling IPv6. Creating a new network on the personal computer (PC) using the Ethernet interface with DCHP disabled, IP address set to 192.168.0.1/24, and subnet mask set to 255.255.255.0, ensures a bidirectional communication channel via Ethernet between the PC and RPi, essential for SSH. Unlike Shepherd's system, which uses the Raspberry Pi OS, this system utilises the Raspberry Pi OS Lite which lacks a desktop environment. Using SSH with X11 forwarding with an X11 display server software, such as XQuartz, on the PC, graphical user interface windows will render on the PC, enabling quick system debugging. Due to the lack of a desktop environment, video output to the DLP projector will be possible by directly writing to the Framebuffer, which is a portion of memory in RAM containing a bitmap that drives the video display output, ensuring display output rates quicker than through a GUI window such as in Shepherd's system.

4.2 Backscatter Simulator for Synthetic Ground Truth

This software, code in Appendix A.2, utilises Python 3.11.2 just like the rest of the Python programs in this project, the 'Pygame' package, which is a game library, to render the graphical user interface, the 'random' package to generate randomised particle velocities, the 'uuid' package to generate a unique ID for each particle in the simulation, and the 'csv' package to generate an export dataset consisting of the positions and radii of each particle at every simulated frame. Defining a set of constants at the start to control the simulation parameters, which Table 2 describes, the script also consists of a 'Bubble' class which denotes a backscatter particle, with the class fields in Table 3 and the class methods in Table 4, all tables in Appendix A.1.

The program flow, illustrated in Figure 16, begins by initialising the Pygame module and creating the graphical window and the simulation clock. If the program is not set to generate the bubble backscatter particles continuously, the program will initialise the bubble backscatter particles, ensuring that the bubble limit is reached before entering the simulation loop. Within the simulation loop, as each iteration denotes a new frame, the program initialises the simulation window background with the colour defined by the user in the constant. Next, only if the constant particle generation feature is enabled by the user, the program initialises the bubbles to the maximum count possible. Then, the program logs the position and radius of each backscatter particle in the frame before drawing every bubble and moving them. The program then runs a probability check, only if true will it randomise the velocities of every single bubble. The program then removes the particles that are outside of the simulation window before exporting an image capture of the frame. The reason behind exporting a PNG image file of each frame instead of a singular video file is to simulate the frame-by-frame capture logic of the Backscatter Cancellation System from the RPi GS Camera, Section 4.4 explains this in further detail. Finally, the Pygame-provided clock functionality limits the simulation to the user-defined constant and stores the time delta, which is the duration in seconds from the last frame, before iterating once again. When the user inputs a 'quit' signal by closing the window, the program creates a CSV file, with the help of the 'csv' Python module, by offloading the metric data from each frame before the program gracefully terminates.

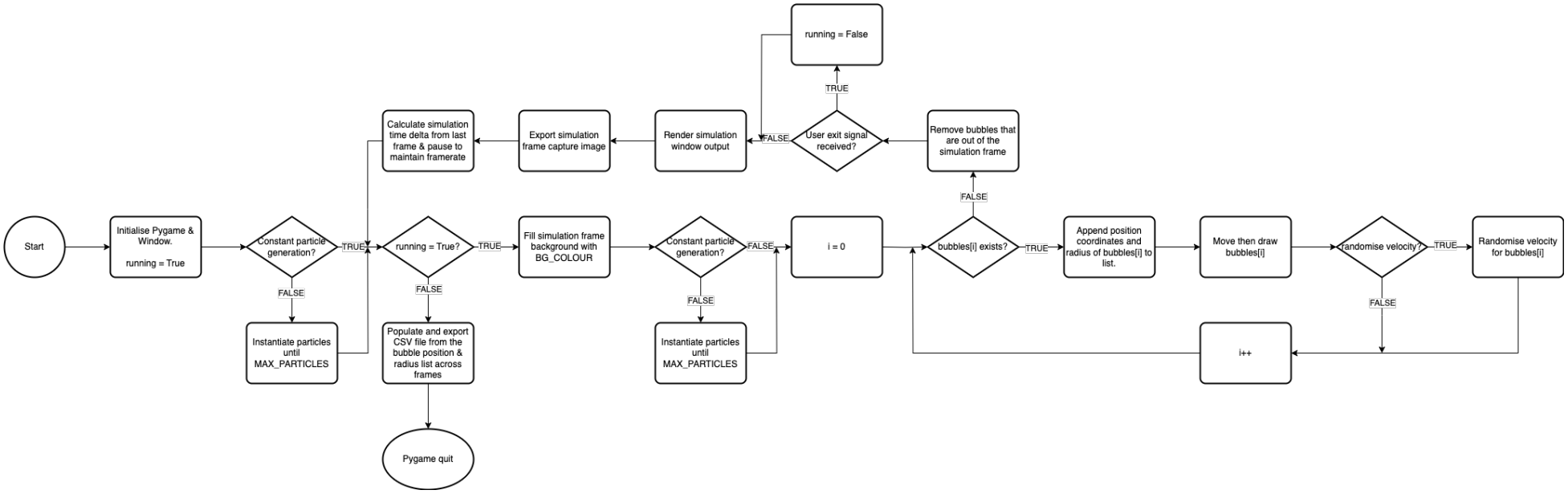


Figure 16: Flowchart of backscatter simulation software process.

4.3 Lossless Video Recorder for Physical Testing

As Section 3.3 introduces, this software, Python scripts in Appendix B.2, records test footage of bubbles from the underwater testing facility at the Institute for Safe Autonomy using an RPi 5 and an RPi GS Camera, from the submersible system itself. The software also drives a DLP projector to project a fully illuminated white light source, ensuring a well-lit scene and to induce backscatter, via HDMI using the Framebuffer to eliminate the need for a desktop environment. Aside from the main entry point script, the software consists of three self-built modules: (1) ProjectorManager, which interfaces the software with the DLP projector via the Framebuffer, (2) the CameraManager, which interfaces the RPi Camera with the software, and finally, (3) the PreviewManager, which produces the GUI window to output the recording previews and program status. The program uses the 'Numpy' Python package to generate bitmap arrays of pixel values to drive the DLP projector via the Framebuffer. The 'Picamera2' library, although only available as a beta release at the time of writing, provides the program with high-level access to the RPi Cameras and RPi SBC's built-in imaging hardware. The 'subprocess' package spawns the FFmpeg process to encode the recording, with the assumption that FFmpeg is present on the system's RPi SBC, and finally, the program uses the 'OpenCV' package to access machine-vision algorithms for image processing and to generate the preview GUI window. The tables in Appendix B.1 list and describe the program configuration constants, and the fields and methods for each class.

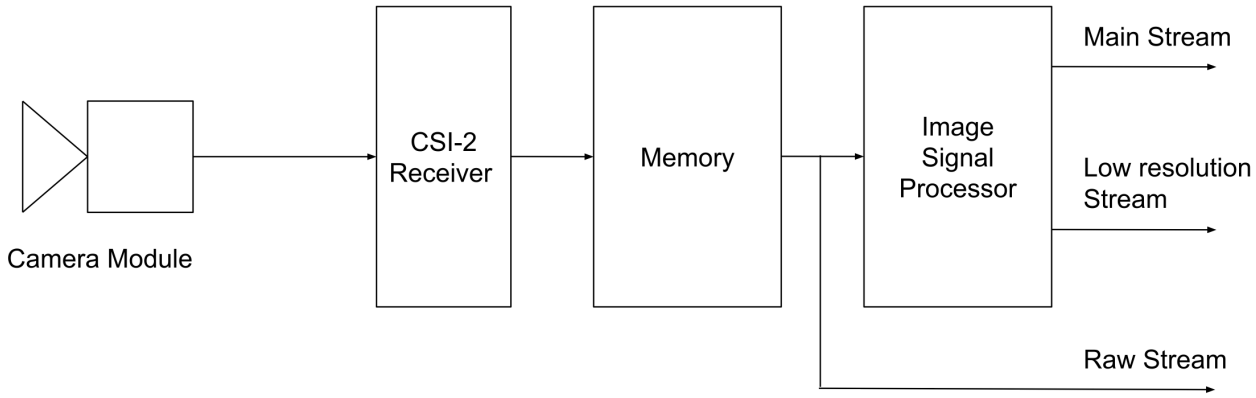


Figure 17: Illustration from [25] of the Raspberry Pi camera system.

As the Picamera2 library manual from [25] illustrates, the RPi camera module delivers the output data stream, which is in native sensor format and not human-viewable, to the CSI-2 receiver hardware block onboard the RPi SBC via a ribbon cable. This hardware block transfers the incoming data into memory, ready for delivery to applications. In the initial prototyping stage for this system tool's implementation, I was receiving and recording the 'raw' stream. However, this data required a lot of computationally intensive processing to convert to a human-viewable format, in addition to the high data throughput requirement of 475 MB per second, recording at a steady rate was impossible even with a memory buffer. The Image Signal Processor (ISP) onboard the RPi SBC reads the 'raw' stream from memory and

efficiently cleans and processes the pixel data stream, ultimately producing a human-viewable output in either, depending on user set configuration, the RGB or YUV format. Instead of parsing the 'raw' stream, I chose to process the 'main' output stream in the final version of the software, configuring the exact resolution and framerates, in addition to a BGR888 output format such that the data is natively compatible with the OpenCV image processing libraries.

Figure 18 illustrates the flow of the software. The program begins with an initialisation, initialising the preview window, which passes through to the remote SSH connection on the PC via X11 forwarding, the projector via the Framebuffer, and the Picamera2 library, which applies a configuration to the RPi GS camera module and starts it. The main loop of the program then begins, retrieving the ISP-processed camera sensor array from the 'main' output stream, and also retrieving the statuses of whether the projector light is on and whether the camera is recording. The sensor array frame is sent to the preview window, along with the statuses that the program overlays on the camera frame, finally logging any user-input keypresses. Using the Framebuffer, which is a memory location accessible through a directory, the program can toggle the light source by writing a bitmap array of white pixel values to turn on, and a bitmap array of black pixel values to turn off. If the program detects an 'L' keypress, the projector light source toggles. With an 'R' keypress, the program toggles the recording status. If the recording needs to start, the program initialises a memory buffer, configures the Picamera2 module to write to this Framebuffer along with a 'Null' encoder, which ensures that the output is unencoded and 'raw', and finally sends a signal to start recording. If the recording needs to stop, the program sends a signal to the Picamera2 module to stop the current recording, and using FFmpeg, which is installed on the RPi SBC, the program applies lossless encoding using the FFV1 encoder to convert the bitstream into an '.MKV' file, reducing the filesize and converting to a common video file format, for easy file transmission and so that any general-purpose software can open and this footage for playback. When the program receives an 'E' key input, it will send the shutdown signal to the Picamera2 module, which gracefully terminates the camera module, it will turn off the projector light source by clearing the Framebuffer and sends a 'destroy windows' signal to OpenCV which will consequently close all windows gracefully, and finally, the program exits the main loop and quits. If the program does not detect a keypress, or if it has finished handling the keypress for all inputs except the exit signal, the main loop iterates such that the entire process repeats.

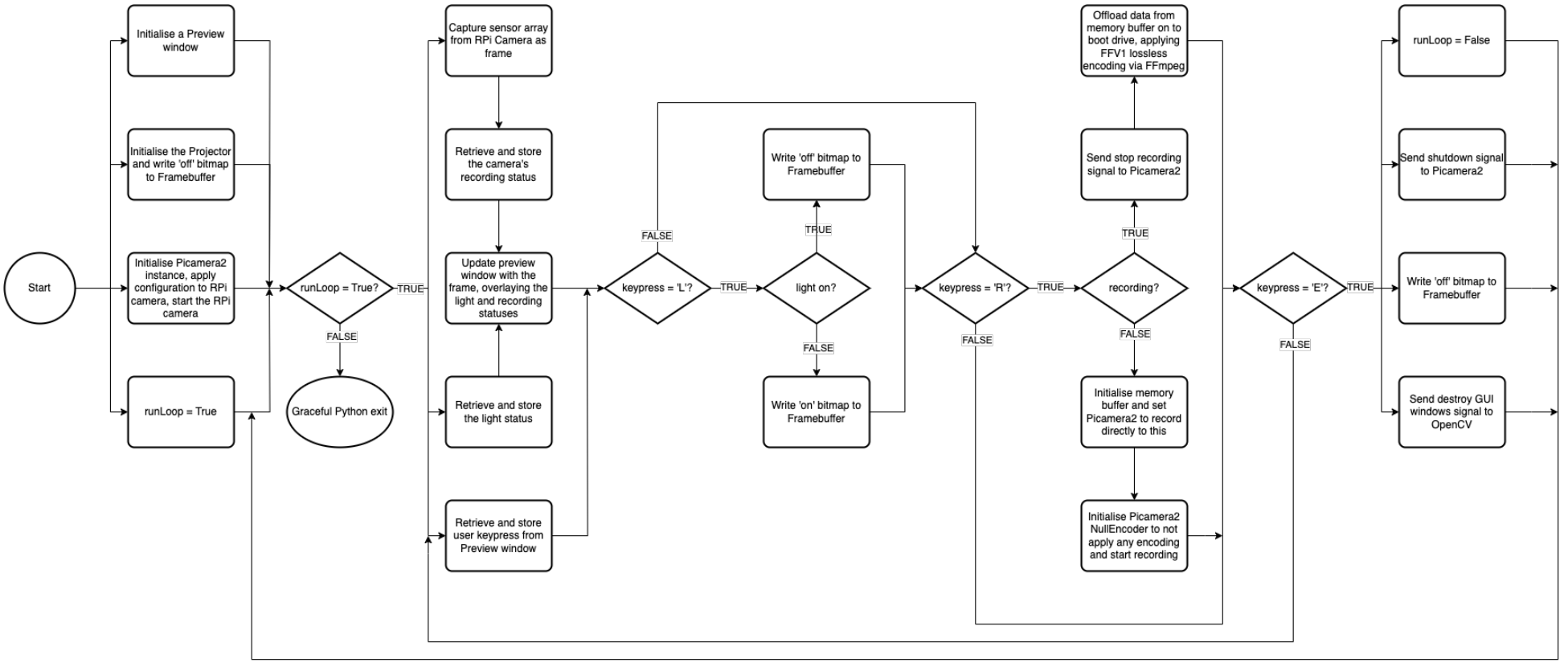


Figure 18: Flowchart of lossless video recorder software process.

4.4 Underwater Backscatter Cancellation System

The backscatter cancellation software, as Section 3.4 describes, with code and references for methods and fields in Appendix C, applies an image processing pipeline to segment backscatter particle regions to then generate light patterns for selective illumination using the DLP projector. Similar to the Lossless Video Recorder implementation in Section 4.3, this system follows the object-oriented paradigm for modular code, breaking down the implementation into four self-built modules, aside from the entry point: (1) CaptureManager, which contains multiple different classes for input types, for example, a Raspberry Pi Camera interface similar to the CameraManager in Lossless Recorder, and a video file stream, which allows test footage inputs. (2) BSManager, which contains the image processing pipeline and backscatter segmentation functionalities, (3) TimeManager, which contains functionality to precisely time code, and finally, (4) WindowManager, very similar to PreviewManager in the Lossless Recorder program, contains functionality to output frames to a GUI for debugging. Instead of utilising the Picamera2's recording functionality for the RPi Camera Stream class, the system captures each frame individually at each iteration so enable fine control over framerates. Due to time constraints and issues with system offsets/parallax, which Section 5 will explain in further detail, I couldn't implement functionality to output the light patterns by directly driving the DLP projector with parallax and offset mitigation, and therefore, skipping the implementation of logic to limit the framerates, such that this system outputs entirely via the GUI windows. Just like the Lossless Recorder program, this system uses the OpenCV package for both the image processing functions and for GUI windows. The program also uses Numpy package for bitmap operations concerning the image frames, the Pandas package for large data set operations concerning the metrics that the system tracks, the 'psutil' package to set the program 'nice' value, which is the software priority for the OS scheduler, and finally, the 'time' package, where the system harnesses the performance counter, which is a system clock with the highest available resolution, to measure code execution durations.

As Figure 19 illustrates, the program begins by setting the software priority level, this priority level is also called the 'niceness', which has the lowest compatible value of -20, denoting the highest priority, and the highest compatible value of 0, denoting lowest priority, such that the lower priority processes demands less CPU processing time and are nicer to other processes in the system. Next, the program sets the correct input stream type before initialising the graphical preview windows and entering the main system loop. Within this loop, the program captures a frame from the input stream at the start of each iteration, and for each frame, applies the image processing pipeline to segment bubbles. After a greyscale conversion, Gaussian blur, and histogram equalisation, all using OpenCV functions, the program configures the Canny edge detection algorithm following the zero-parameter approach by [26]. This approach by Rosebrock, simplifies the upper and lower Canny algorithm thresholds into a single parameter, sigma, which denotes a multiplier for the median pixel intensity of the input frame, such that the upper threshold is sigma above the median, and the lower threshold is sigma below the median. While not exactly zero-parameter as Rosebrock states, as there is still a sigma parameter, this

approach ensures that the edge detection adapts to the specific characteristics of the input image even in different conditions, providing effective edge maps without manual parameter tuning. The system passes the Canny output, a bitmap of detected edges, to the segmentation method which then begins by finding contours, which are curves that join all the continuous points along a boundary, using OpenCV's `findContours()` method. Next, the program sends each contour detection to OpenCV's `minEnclosingCircles()` method, calculates the smallest enclosing circle for a set of contour points, and returns the center and radii of each circle. These minimum enclosing circles (MECs) will be the black 'holes' in the white light patterns, ensuring the backscatter particles are dark whilst the system fully illuminates the surrounding environment. The program renders all of the output image frames, including those from the intermediate image processing pipeline stages if the debug window option is 'True', whilst logging the user-input keypresses. If the user enters the 'E' key, the program breaks from the main loop and enters the shutdown sequence. The program tracks the execution duration of each image processing stage with the performance counter for each frame, including metrics on the number of MEC detections, and the total frame processing time. When the program reaches the shutdown sequence, it extracts all of the metrics and generates a CSV export file, then exits all GUI windows, sends a shutdown signal to the RPi camera module, and finally exits completely.

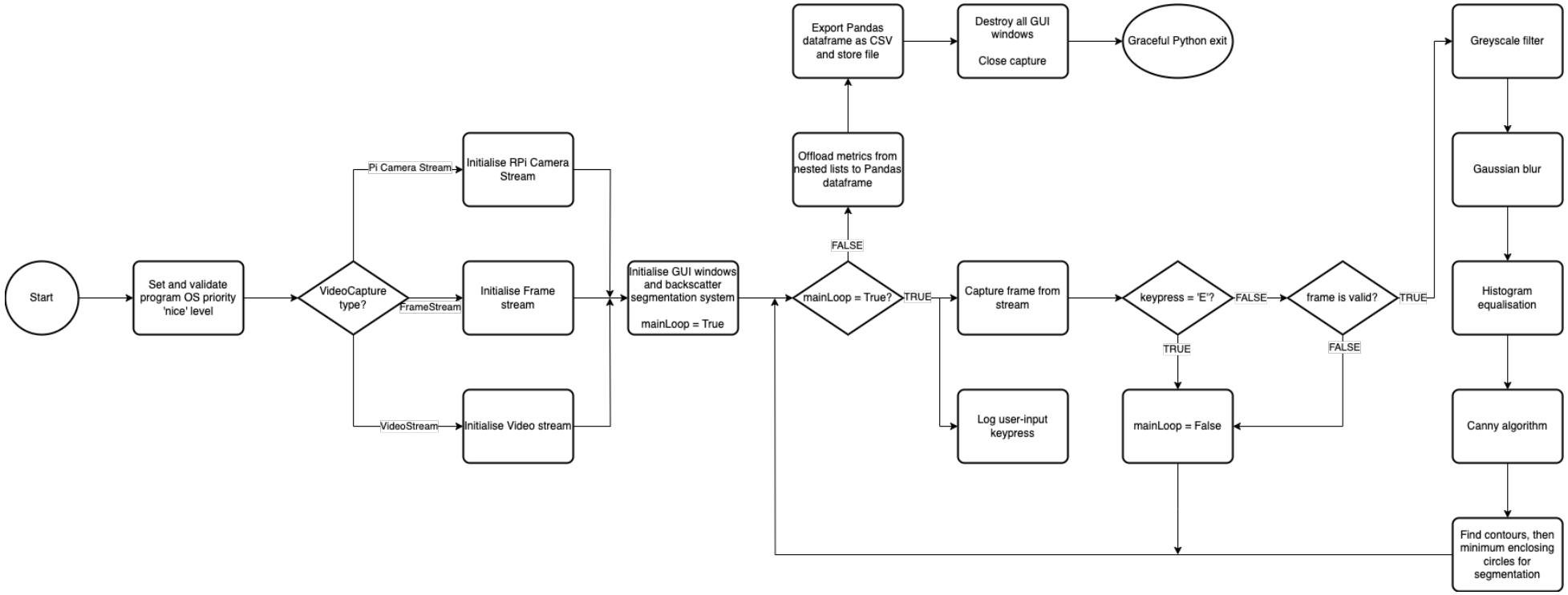


Figure 19: Flowchart of backscatter cancellation system software process.

4.4.1 Implementing Multiprocessing

The functionality of the multiprocessing system follows very closely to the aforementioned base system, which only utilises a single thread due to the Python GIL limitations. Appendix D contains the references for methods and fields as well as the code for this implementation. The 'multiprocessing' Python package provides an interface to support spawning subprocesses with concurrency, effectively side-stepping the GIL, and ultimately enabling the system to fully leverage multiple processors on a given machine, unlike Shepherd's system which employs 'multithreading' and, therefore, is subject to the GIL for standard CPython execution. The single-thread version of the system is split into eight stages, denoted by classes that inherit the 'multiprocessing' Process, which is an object that spawns as a separate process: (1) Capture, which first initialises the input stream, then enters an infinite loop that iterates until the input stream ends whilst returning the capture frame and any user-input keypresses. (2) Greyscale, which simply applies the greyscale filter. (3) Histogram Equalisation, which also simply applies histogram equalisation. (4) Gaussian Blur, again simply applying a Gaussian blur. (5) Canny Algorithm, which applies the Canny edge detector using the 'zero-parameter' approach by Rosebrock. (6) Segmentation, which first finds the contours before applying the minimum enclosing circle function to segment the bubbles. (7) Project, which originally intended to output the light patterns to the Framebuffer, however due to the aforementioned constraints and limitations, instead outputs to a graphical preview window. Finally, (8) Logging, which exports the metrics into a Pandas DataFrame, then into a CSV document.

Due to the asynchronicity of the processes, the system employs queues to send data between stages in a thread-safe manner, preventing instances of data corruption with shared memory locations such as standard variables due to clobbering, where a process overwrites data while another was reading the data. The Capture stage will start filling its output queue with frames, and if no frames are left, or if the process receives a 'quit' signal from the user, it enqueues with a special code instead of a frame. This queue is input to the next stage, which is Greyscale, and this stage begins to dequeue, applying the greyscale filter, then enqueues to its output queue, and if it receives a quit code instead of a frame, it will shutdown itself after forwarding the code. This logic repeats across all the other stages.

4.5 Applying the PREEMPT-RT Patch

The latest version of Raspberry Pi OS Lite at the time I built the real-time kernel, which was on the 15th of March 2024, used the 6.6.20 Linux kernel version. Therefore, I applied the PREEMPT-RT patch with the corresponding version label of 6.6.20-rt25. I selected the 'Fully Preemptible Kernel (Real-Time)' option, also enabling the RCU priority boosting with a 500ms delay boosting for the RT kernel and enabled the 'Multi-core scheduler support' option for both RT and the non-RT kernel. Finally, I built both kernels, copied the libraries and both kernel image files into the correct RPi SBC directories, making sure to take a backup of the existing kernel, and finally, in the boot configuration file, I selected the correct kernel filename, being

able to interchange between either of the two kernels, after rebooting.

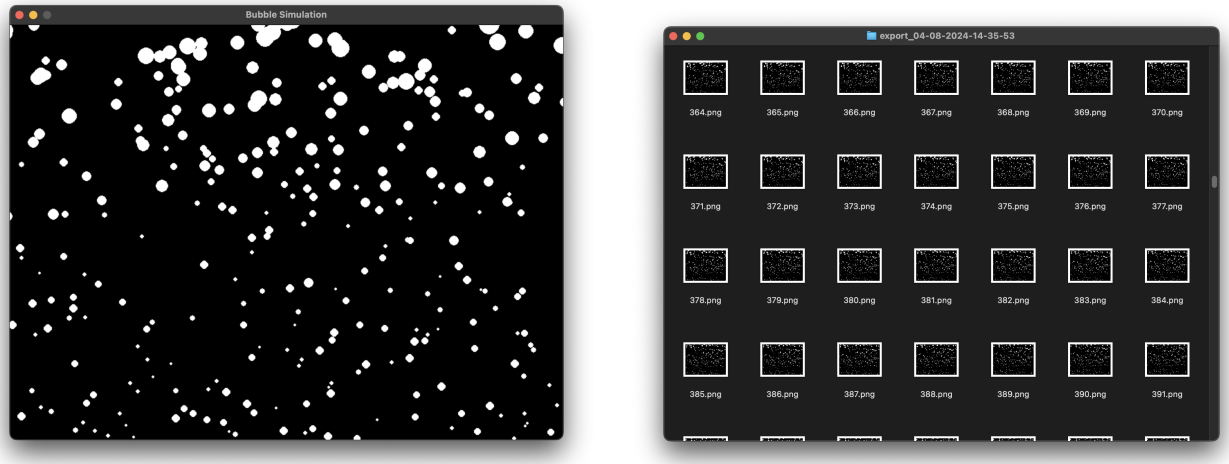
5 Testing & Evaluation

In previous work, Shepherd employs a five-stage testing method: (1) calibration for parallax mitigation, (2) testing stationary backscatter, (3) testing with different reflective materials inducing backscatter, (4) motion and synchronisation of the system, and (5), underwater testing. Shepherd carries out these tests using a temporary testing rig, outside the submersible system, consisting of the projector, camera and RPi SBC fastened to a wooden plate, and employing the submersible system with the end open for underwater testing by partial submersion in a water barrel. Unfortunately, due to the large offsets and parallax between the camera and projector, the project shifts its focus from physical testing to software testing and real-time performance analysis. Following the details on implementation in Section 4, this section aims to describe the testing process for testing both the system and the toolsets, including an evaluation of the backscatter cancellation system performance.

5.1 Testing the Systems

5.1.1 Bubble Backscatter Simulator

The bubble backscatter simulator program works very well as a simple backscatter simulation. As Figure 20 illustrates, the program successfully renders a preview graphical user interface, enabling the user to view the bubbles as they generate and move, and the program also correctly exports each frame of the simulation as a PNG file, emulating the frame-by-frame capture loop behaviour of the backscatter cancellation system. Furthermore, the program exports a CSV dataset of the individual backscatter particle positions and radii from each frame, classing each particle with a unique ID, for the first few particles in the dataset. Despite this, a major drawback that I found from this program is the fact that it doesn't simulate varying backscatter appearances, such as morphological transformations with shape warping, and colour changes, to verify if the system can detect varying backscatter parameters, as all particles are exactly circular and white. In addition, the program does not simulate varying background conditions as the background is exactly black, resulting in the inability to verify whether the system can correctly isolate and segment just the particles and not any regions of interest in the background.

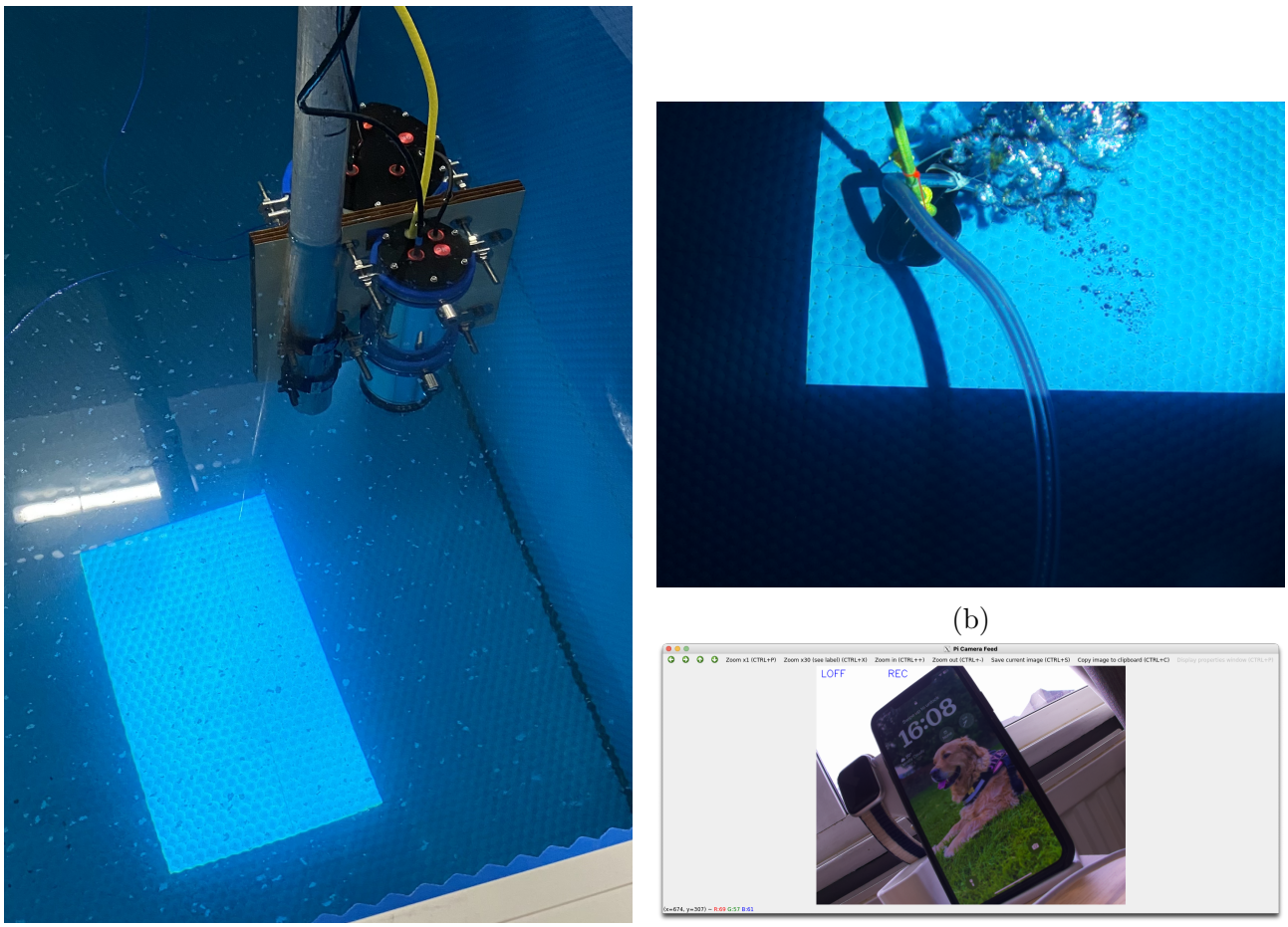


(a) (b)

Figure 20: Screenshots validating the Bubble Backscatter Simulator program: (a) showing the preview GUI window, and (b) showing the list of frames in the output directory.

5.1.2 Lossless Video Recorder

The lossless video recorder proved an invaluable tool that performed almost flawlessly, as Figure 21 illustrates. A change I had to make was to modify the OS to ensure the console would not output to the HDMI Framebuffer, as this was causing interference with my Framebuffer-driven light source. Unfortunately, as the figure also illustrates, there was a major misalignment issue, where the camera is only able to a small corner in the bottom-left of the projector-lit scene. While a simple cropping function can resolve most of this, which I have implemented in the backscatter cancellation system, albeit reducing the system's useable region, the system will still be affected by the camera's internal reflection, which reduces the field of view and adds distortion. Nevertheless, the recordings were very helpful in verifying the system in a non-synthetic environment without requiring re-deployment underwater.

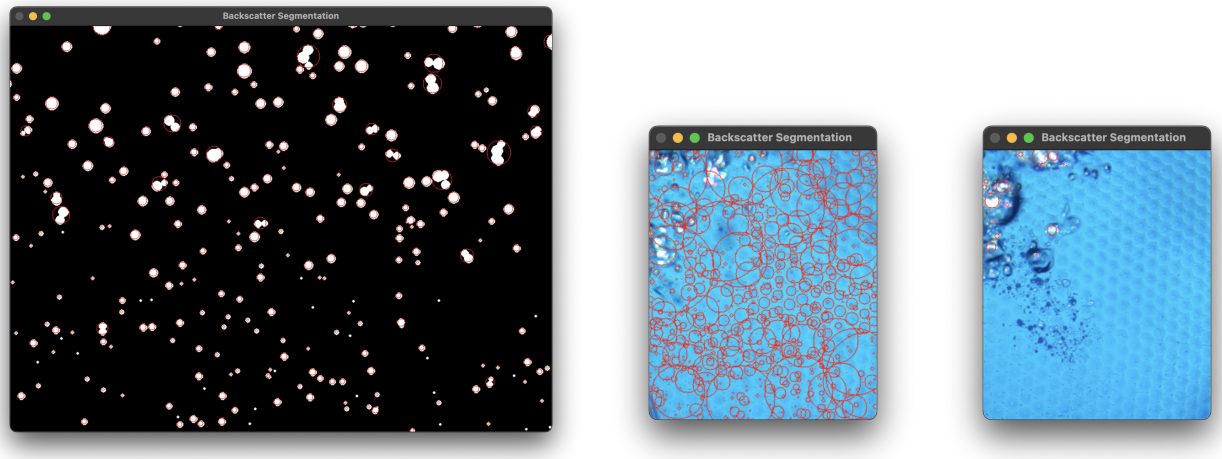


(a) (b) (c)

Figure 21: Figures validating the Lossless Video Recorder program: (a) showing the submersible recording underwater at the Institute for Safe Autonomy testing tank, (b) an extracted frame from recorded footage, and (c) showing the preview GUI window, rendered remotely with X11 forwarding.

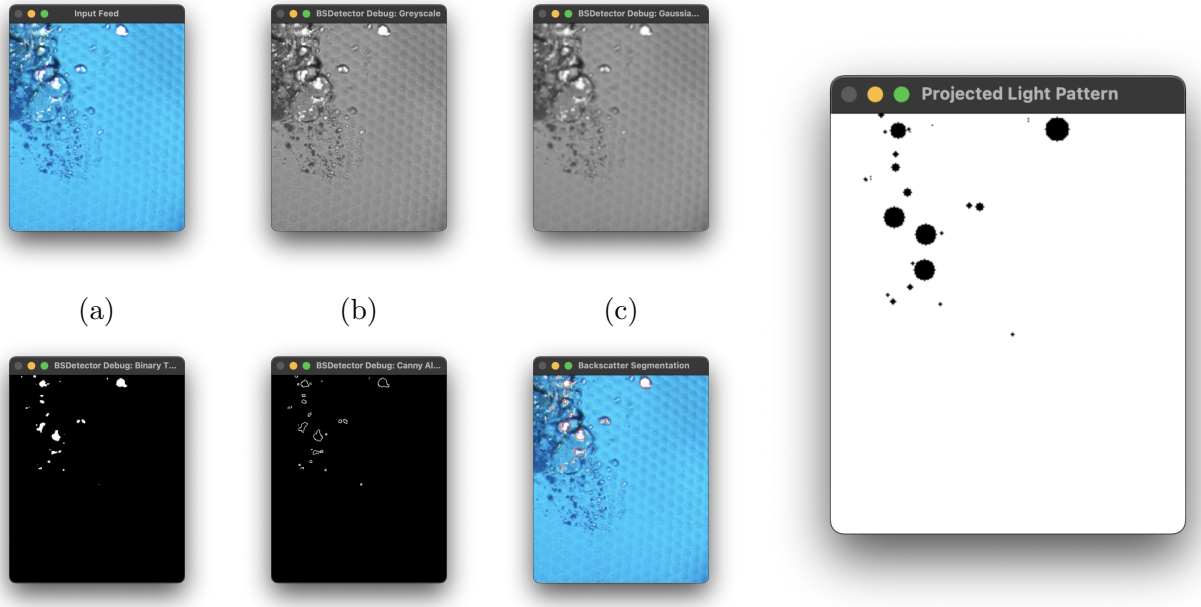
5.1.3 Backscatter Cancellation System

I first verified the backscatter cancellation system using the output from the backscatter simulation and it worked flawlessly, identifying and segmenting the bubbles correctly using the minimum enclosing circle logic, as Figure 22a illustrates. However, when using the underwater test footage, the system started to incorrectly detect the background, mainly the textured padding of the testing tank, as Figure 22b illustrates. I identified that this was caused by the histogram equalisation stage, where the entire image frame's pixel intensities are spread, increasing the contrast substantially. I therefore replaced this stage with an OpenCV binary thresholding implementation, ensuring that the system only passes certain pixel intensities. The results after this processing pipeline stage replacement were much better, as Figure 22c illustrates. Figure 23 illustrates the final system's image processing pipeline flow.



(a) (b) (c)

Figure 22: Screenshots validating the Backscatter Cancellation System program: (a) showing the backscatter segmentation (red circles) from the Backscatter Simulator output, (b) incorrect segmentation of backscatter from the test footage, and (c), which shows the correct backscatter segmentation using the same footage.



(a) (b) (c) (d) (e) (f) (g)

Figure 23: Screenshots previewing the image processing pipeline stages of the Cancellation System program, in order from left to right: (a) the input, (b) greyscale filtering, (c) Gaussian blur, (d) binary thresholding, (e) the Canny algorithm, (f) segmentation with minimum enclosing circles, finally, (g), the projection.

5.2 Performance Evaluation

Due to time constraints, I couldn't implement logic in the backscatter cancellation system which exports a dataset of detected backscatter particles, which I could've then used to compare with

the simulation export, the synthetic ground truth, to quantify the accuracy performance of the system. Therefore, this section entirely focuses on the real-time performance, by analysing the execution time for each image processing pipeline, and the total execution time for each frame with the underwater testing tank footage input to the RPi SBC system with the system software 'nice' value set to the lowest value (-20), ensuring the highest process priority, ultimately quantifying the impact of multiprocessing and the PREEMPT-RT kernel patch.

Process	**Avg. Duration (µs)**				**Std. Deviation (µs)**			
	Std, SP	RT, SP	Std, MP	RT, MP	Std, SP	RT, SP	Std, MP	RT, MP
Frame Capture	1,480	20,100	12,120	27,730	3199	9629	14990	11600
Greyscale Filter	193.4	2370	220.2	212.0	31.19	57.67	91.02	80.16
Gaussian Blur	159.2	319.2	339.5	378.8	148.2	457.4	653.0	599.0
Binary Threshold	19.79	29.70	64.10	81.34	5.144	61.15	57.45	111.3
Canny Algorithm	601.9	795.3	879.20	975.3	95.46	550.3	724.4	1427
Find Contours	110.8	135.8	180.0	252.2	43.16	184.2	134.0	389.2
Find MECs	41.88	47.14	45.60	47.05	30.84	36.67	92.65	41.89
Tot. Duration (µs)	2,607	21,620	13,850	29,670	3253	9732	15140	11920

Table 1: Table of real-time metrics for the system using the standard (std.) and real-time (RT) kernel, in single-core mode (SP), and with multiprocessing (MP), all values rounded up to the nearest 4 significant figures.

Table 1 transcribes the real-time metrics for the system, across 402 frames of real test footage recorded from the testing tank, listing the execution duration for all the image processing pipeline stages using the standard kernel, real-time kernel, multiprocessing, and in the standard single core mode under the GIL. The data shows a 431% increase in total latency from the single-core to the multi-core program with the standard kernel, and 37& increase of that in the real-time kernel, a 729% increase when moving from the standard to the real-time kernel with single-core, and a 114% increase when moving from that in the multiprocessing system. The data also contains the standard deviation of the individual processing stage and total system latencies to measure data variability, showing a 199% increase in fluctuations when moving from the standard to real-time kernel in the single-code environment, and a 21% decrease of that in the multiprocessing environment.

In contrast to the theory that previous sections cover regarding the reduction of latency with multiprocessing and a real-time OS, the data illustrates the complete opposite. The multiprocessing system employs queues, a form of Inter-Process Communication (IPC), to preserve data whilst processing asynchronously across multiple processes. However, IPCs introduce vast computational overhead due to the requirement of data serialisation and deserialisation when transmitting across processes and during OS context switching. Furthermore, while multiprocessing offers parallelism, it still cannot overcome the synchronous nature of I/O bound processes, as the data quantifies showing a 719% increase in frame capture durations between the standard single-core and standard multiprocessing systems. The PREEMPT-RT patch, similarly in theory, should've reduced system latency, but most importantly it should've at

least resulted in more constant latency measurements. However, the data proves the opposite. The PREEMPT-RT patch increases the frequency of preemption points in the kernel, resulting in more context switches, and when paired with the IPCs, slowing down the system rate and increasing latency.

In summary, the PREEMPT-RT patch greatly alters the kernel behaviour, and due to time constraints, I was unable to successfully profile, research and analyse the effects and pinpoint the exact cause for the latency increase, however it is clear that trying to modify the kernel behaviour from such an abstracted level causes more negative responses, than doing so from a low system level, proving that it is better to leave these tasks to the OS, which is specialised in correctly optimising processes much better than me. To conclude, my single-core RPi SBC system achieves a total processing latency of just 2.6 ms, a great improvement from the system in De Charette et al., which takes 4.1 ms to process frames on a much more powerful desktop computer, and achieving the 33.33 ms (30FPS) target that I had originally set in Section 3.4.

6 Conclusion

This project set out to design, implement, and evaluate a real-time backscatter cancellation system for underwater imaging, leveraging an RPi SBC for portability and accessibility. The project began by developing the toolsets for the final system, including a backscatter simulation to generate a synthetic ground truth to test and validate the final system, and a lossless recording script to capture underwater footage to test the final system without requiring underwater deployment. These toolsets provided crucial assistance in the final system's development.

The project achieves the objective of precise backscatter particle segmentation, even in varying environmental conditions, by employing an image processing pipeline centred around the Canny edge detection algorithm, with an adjustment to replace the histogram equalisation stage with binary thresholding, and a simple minimum enclosing circle segmentation method. The project explores methodologies to reduce system latency with a real-time kernel and multiprocessing for increasing system throughput to achieve the latency targets, uncovering mixed results, including the adverse effects of the real-time kernel patch for Linux and Python multiprocessing. However, the single-core system achieves a commendable 2.6 ms average frame processing latency, a clear improvement over previous research.

Underwater testing of the system, with the lossless recording program, uncovered the drastic parallax and distortion effects due to the submersible housing construction, which resulted in the project's shift from focusing on the development of a final working system, to real-time software optimisation, to balance the short project time duration. Therefore, driving the DLP projector, including functionality to control and establish a fixed system frame rate, was no longer an actionable item. Appendix E illustrates the evolution of the project schedule using Gantt charts.

In conclusion, this project successfully delivers a functional and efficient backscatter cancellation

system, with notable improvements in real-time processing speed. The insights gained from the performance evaluation highlight the intricacies of optimising such systems and suggest that, for specific applications, simpler single-core implementations may offer superior performance compared to more complex multiprocessing or RTOS-enhanced solutions. Future work must first focus on completing this system's objectives for DLP-driven projections, fine-grain framerate controls, a more realistic backscatter simulation, and verifying the system's performance with the synthetic ground truth. After that, the following research can drastically improve the system: (a) FPGA implementation for accelerated image processing and DLP projector driving by harnessing the inherently multithreaded architecture, (b) improved submersible housing, mitigating the component offsets with high-precision alignment and a beamsplitter to eliminate parallax by co-location, and finally, (c), a predictive system to track and estimate the future location of backscatter particles to mitigate backscatter movement against system latencies, with options to incorporate machine-learning technologies, using a more realistic backscatter simulation for training data.

References

[1] University of Hawai'i, "Practices of Science: Underwater Photography and Videography." [Online]. Available: https://manoa.hawaii.edu/exploringourfluidearth/physical/ocean-depths/light-ocean/practices-science-underwater-photography-and-videography [Accessed: 2024-02-13]

[2] Yannick Allard and Elisa Shahbazian, *Unmanned Underwater Vehicle (UUV) Information Study.* Defence Research & Development Canada, Atlantic Research Centre, Nov. 2014. [Online]. Available: https://apps.dtic.mil/sti/pdfs/AD1004191.pdf [Accessed: 2024-02-14]

[3] Sidharth Shanmugam, "Initial Report: Machine Vision-Based Anti-Backscatter Lighting System for Unmanned Underwater Vehicles," University of York, York, UK, Tech. Rep., Mar. 2024.

[4] Brent Durand, "Easy Ways to Eliminate Backscatter in your Photos," Oct. 2013. [Online]. Available: https://www.uwphotographyguide.com/eliminate-backscatter-underwater [Accessed: 2024-02-14]

[5] Katie Shepherd, "Machine Vision Based Underwater Anti-Backscatter Lighting System," MEng Project Report, University of York, York, UK, 2023.

[6] OpenCV, "OpenCV: Cv::SimpleBlobDetector Class Reference." [Online]. Available: https://docs.opencv.org/4.9.0/d0/d7a/classcv_1_1SimpleBlobDetector.html [Accessed: 2024-05-01]

[7] R. De Charette, R. Tamburo, P. C. Barnum, A. Rowe, T. Kanade, and S. G. Narasimhan, "Fast Reactive Control for Illumination Through Rain and Snow," in *2012 IEEE International Conference on Computational Photography (ICCP).* Seattle, WA, USA: IEEE, Apr. 2012, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/6215217/ [Accessed: 2024-05-01]

[8] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart, "Connected Components Labeling," 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm [Accessed: 2024-05-01]

[9] R. Tamburo, E. Nurvitadhi, A. Chugh, M. Chen, A. Rowe, T. Kanade, and S. G. Narasimhan, "Programmable Automotive Headlights," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, vol. 8692, pp. 750–765. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10593-2_49 [Accessed: 2024-05-01]

[10] Y. Zhang, Y. Yu, X. Rui, Z. Feng, J. Zhang, Y. Chen, L. Qi, X. Chen, and X. Zhou, "Underwater bubble escape volume measurement based on passive acoustic under noise factors: Simulation and experimental research," *Measurement*, vol. 207, p.

112400, Feb. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224122015974 [Accessed: 2024-02-24]

[11] K. Thomanek, O. Zielinski, H. Sahling, and G. Bohrmann, "Automated gas bubble imaging at sea floor - a new method of in situ gas flux quantification," *Ocean Science*, vol. 6, no. 2, pp. 549–562, Jun. 2010. [Online]. Available: https://os.copernicus.org/articles/6/549/2010/ [Accessed: 2024-02-24]

[12] J. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986. [Online]. Available: https://ieeexplore.ieee.org/document/4767851 [Accessed: 2024-05-02]

[13] C. Zelenka, "Gas Bubble Shape Measurement and Analysis," in *Pattern Recognition*, X. Jiang, J. Hornegger, and R. Koch, Eds. Cham: Springer International Publishing, 2014, vol. 8753, pp. 743–749. [Online]. Available: https://link.springer.com/10.1007/978-3-319-11752-2_63 [Accessed: 2024-05-02]

[14] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," *Int J Comput Vision*, vol. 1, no. 4, pp. 321–331, Jan. 1988. [Online]. Available: http://link.springer.com/10.1007/BF00133570 [Accessed: 2024-05-02]

[15] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960. [Online]. Available: https://asmedigitalcollection.asme.org/fluidsengineering/article/82/1/35/397706/A-New-Approach-to-Linear-Filtering-and-Prediction [Accessed: 2024-02-28]

[16] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics*, vol. 2, no. 1-2, pp. 83–97, Mar. 1955. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/nav.3800020109 [Accessed: 2024-02-28]

[17] Bootlin, "Understanding Linux real-time with PREEMPT_RT training," Jan. 2024.

[18] Mauro Riva, "Raspberry Pi 4B: Real-Time System using Preempt-RT (kernel 4.19.y)," Sep. 2019. [Online]. Available: https://lemariva.com/blog/2019/09/raspberry-pi-4b-preempt-rt-kernel-419y-performance-test [Accessed: 2024-03-01]

[19] "What is an FPGA? Field Programmable Gate Array." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html [Accessed: 2024-02-23]

[20] Raspberry Pi Ltd, "Raspberry Pi - About us." [Online]. Available: https://www.raspberrypi.com/about/ [Accessed: 2024-02-23]

[21] "How does a DLP projector work?" [Online]. Available: https://www.projectorscreen.com/blog/How-does-a-DLP-projector-work [Accessed: 2024-02-24]

[22] RED Digital Cinema, "Global & Rolling Shutters." [Online]. Available: https://www.red.com/red-101/global-rolling-shutter [Accessed: 2024-02-24]

[23] “Rolling Shutter vs Global Shutter sCMOS Camera Mode.” [Online]. Available: https://andor.oxinst.com/learning/view/article/rolling-and-global-shutter [Accessed: 2024-02-24]

[24] OpenCV, “About.” [Online]. Available: https://opencv.org/about/ [Accessed: 2024-03-01]

[25] Raspberry Pi Ltd, “The Picamera2 Library,” Apr. 2024. [Online]. Available: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf [Accessed: 2024-04-01]

[26] A. Rosebrock, “Zero-parameter, automatic Canny edge detection with Python and OpenCV,” Apr. 2015. [Online]. Available: https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/ [Accessed: 2024-05-16]

Appendix A Bubble Backscatter Simulator

A.1 Description of Class Fields & Methods

Name	Description
WINDOW_WIDTH	The maximum width of the simulation window.
WINDOW_HEIGHT	The maximum height of the simulation window.
SIMULATION_FPS	The frame rate to simulate.
MAX_VELOCITY_X	The maximum possible x-axis velocity for particles.
MIN_VELOCITY_X	The minimum possible x-axis velocity for particles.
MAX_VELOCITY_Y	The maximum possible y-axis velocity for particles.
MAX_VELOCITY_X	The minimum possible y-axis velocity for particles.
VELOCITY_CHANGE_RATE	The rate at which the velocity changes.
MAX_RADIUS	The maximum possible spawn-time particle radius.
MIN_RADIUS	The minimum possible spawn-time particle radius.
HEIGHT_RADIUS_GROW_MULTIPLIER	The particle radius growth rate as the particle rises.
BG_COLOUR	The background colour of the simulation window.
PARTICLE_COLOUR	The particle colour the simulation backscatter.
MAX_PARTICLES	The maximum possible particles in each frame.
CONSTANT_PARTICLE_GENERATION	Option to constantly spawn particles.
PARTICLE_RANDOMISE_VELOCITY_PROB	The probability of particle movement randomisation.

Table 2: Simulation parameter constants and their descriptions from the Bubble Backscatter Generator program.

Name	Description
id	The unique ID of the particle, generated by `uuid.uuid4()`.
x	The x-axis coordinate position of the particle.
y	The y-axis coordinate position of the particle.
radius	The radius of the particle.
velocity_x	The current x-axis velocity of the particle.
velocity_y	The current y-axis velocity of the particle.
target_velocity_x	The target future x-axis velocity of the particle.
target_velocity_y	The target future y-axis velocity of the particle.

Table 3: Fields and their descriptions from the Bubble class of the Bubble Backscatter Generator program.

Name	Description
`__init__()`	Initialises a new backscatter `Bubble` object, generating a unique `id` with `uuid.uuid4()`, a random `x` starting position, `y` starting position at the bottom of the screen, random `target_velocity_x` and `target_velocity_y` velocities which are set equal to the initial velocities, `velocity_x` and `velocity_y`.
`randomiseVelocities()`	Generates a random value for `target_velocity_y` and `target_velocity_x`, with the value between `MIN_VELOCITY_Y`, `MAX_VELOCITY_Y`, and `MIN_VELOCITY_X`, `MAX_VELOCITY_X`.
`move()`	Updates the particle's `x` and `y` positions based on its `velocity_x` and `velocity_y` and the simulation time since the last frame. Also updates the particle's velocities, based on the difference between the current velocity and the target velocity, with the `VELOCITY_CHANGE_RATE` multiplier.
`draw()`	Draws the particle based on its `x` and `y` positions, and `radius`, with the colour from the `PARTICLE_COLOUR` constant, using the Pygame library's `pygame.draw.circle()` function.

Table 4: Methods and their descriptions from the Bubble class of the Bubble Backscatter Generator program.

A.2 Python Code

```
import pygame
import random
import uuid
from datetime import datetime
import csv
import os

WINDOW_WIDTH = 800          # Width of simulation window
WINDOW_HEIGHT = 600         # Height of simulation window
SIMULATION_FPS = 60         # Simulation frame rate

MAX_VELOCITY_X = 50         # Maximum possible x-axis velocity for particles
MIN_VELOCITY_X = -50        # Minimum possible x-axis velocity for particles
MAX_VELOCITY_Y = 200        # Maximum possible y-axis velocity for particles
MIN_VELOCITY_Y = 50         # Minimum possible y-axis velocity for particles
VELOCITY_CHANGE_RATE = 0.1  # Rate at which velocity changes

MAX_RADIUS = 6          # Maximum possible spawn-time particle radius
MIN_RADIUS = 2          # Minimum possible spawn-time particle radius
HEIGHT_RADIUS_GROW_MULTIPLIER = 0.05    # Grow multiplier as particle rises

BG_COLOUR = (0, 0, 0)   # Background colour (default is black)
PARTICLE_COLOUR = (255, 255, 255)   # Particle colour (default is white)

MAX_PARTICLES = 300                     # Maximum particles on screen
CONSTANT_PARTICLE_GENERATION = True    # Whether to constantly maintain MAX_PARTICLES
PARTICLE_RANDOMISE_VELOCITY_PROB = 0.01 # Probability or particle movement randomisation

class Bubble:
    """ A bubble-based backscatter particle. """
    def __init__(self):
        self.id = uuid.uuid4()                      # Initialise ID
        self.x = random.randint(0, WINDOW_WIDTH)    # Randomly initialise x-axis spawn
```

```
    position
34          self.y = WINDOW_HEIGHT                      # Initialise y-axis spawn position to
    bottom
35          self.radius = random.randint(MIN_RADIUS, MAX_RADIUS)    # Randomly initialise radius
36          self.randomiseVelocities()                  # Randomly initialise velocities
37          self.velocity_x = self.target_velocity_x    # Randomly initialise velocities
38          self.velocity_y = self.target_velocity_y    # Randomly initialise velocities
39
40      def randomiseVelocities(self):
41          """ Randomly update x and y-axis particle velocities. """
42          self.target_velocity_y = random.uniform(MIN_VELOCITY_Y, MAX_VELOCITY_Y)  # Update
    vertical velocity randomly
43          self.target_velocity_x = random.uniform(MIN_VELOCITY_X, MAX_VELOCITY_X)  # Update
    horizontal velocity randomly
44
45      def move(self, delta_t):
46          """ Update position and radius based on velocity and position. """
47          self.y -= self.velocity_y * delta_t     # Update vertical position
48          self.x += self.velocity_x * delta_t     # Update horizontal position
49
50          # Gradually adjust velocities towards target velocities
51          self.velocity_y += (self.target_velocity_y - self.velocity_y) * VELOCITY_CHANGE_RATE
52          self.velocity_x += (self.target_velocity_x - self.velocity_x) * VELOCITY_CHANGE_RATE
53
54          # As a bubble travels upwards, it must get bigger
55          height_factor = (WINDOW_HEIGHT - self.y) / WINDOW_HEIGHT        # Calculate height
    factor
56          self.radius += HEIGHT_RADIUS_GROW_MULTIPLIER * height_factor    # Grow radius based on
     height factor and multiplier
57
58      def draw(self, screen):
59          """ Draw the bubble-based backscatter particle on screen. """
60          pygame.draw.circle(screen, PARTICLE_COLOUR, (int(self.x), int(self.y)), self.radius)
61
62  if __name__ == "__main__":
63      # Initialise pygame module
64      pygame.init()
65
66      # Initialise pygame window and set window title
67      screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
68      pygame.display.set_caption("Bubble Simulation")
69
70      # Create clock object to track time
71      clock = pygame.time.Clock()
72
73      # Initialise runtime variables
74      running = True          # Stores whether the simulation is running
75      delta_t = 0             # Stores the time in seconds since the last frame
76      bubbles = []            # Array of bubbles which are on the screen
77      bubble_positions = {}   # Dictionary storing individual bubble positions in each frame
78
79      # Log timestamp when simulation starts
80      simulation_start_ts = datetime.now()
81
82      # Generate export filename
83      export_filename_generic = "export_" + simulation_start_ts.strftime("%m-%d-%Y-%H-%M-%S")
84
85      # Create folder for the frame image export
86      if not os.path.exists(export_filename_generic):
87          os.makedirs(export_filename_generic)
88
89      if not CONSTANT_PARTICLE_GENERATION:
```

```
        while len(bubbles) < MAX_PARTICLES:
            bubbles.append(Bubble())

    # Initialise frame counter
    frame_num = 0

    while running:
        # Fill window with the background colour
        screen.fill(BG_COLOUR)

        # Generate new bubbles randomly and constantly
        if CONSTANT_PARTICLE_GENERATION:
            while len(bubbles) < MAX_PARTICLES:
                bubbles.append(Bubble())

        # Log bubble positions and radius in this frame
        for bubble in bubbles:
            # If this bubble has not been previously tracked...
            if bubble.id not in bubble_positions:
                # ... Initialise an empty list to track in subsequent frames
                bubble_positions[bubble.id] = {'positions': [], 'radius': []}
            # Append the list with the bubble's position in this frame
            bubble_positions[bubble.id]['positions'].append((bubble.x, bubble.y))
            bubble_positions[bubble.id]['radius'].append(bubble.radius)

        # Update and draw bubbles
        for bubble in bubbles:
            bubble.move(delta_t)
            bubble.draw(screen)

        # Randomly update bubble velocities
        if random.random() < PARTICLE_RANDOMISE_VELOCITY_PROB:
            for bubble in bubbles:
                bubble.randomiseVelocities()

        # Remove bubbles that are out of the screen
        bubbles = [bubble for bubble in bubbles if bubble.y > -bubble.radius]

        # Event handling
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Update the contents of the full display
        pygame.display.flip()

        # Export frame as png
        filename = os.path.join(export_filename_generic, f"{frame_num}.png")
        pygame.image.save(screen, filename)

        # Limit FPS and store time in seconds since last frame
        delta_t = clock.tick(SIMULATION_FPS) / 1000

        # Increment frame number counter
        frame_num += 1

    # Generate CSV dataset of simulated bubble positions
    export_filename_csv = export_filename_generic + ".csv"

    with open(export_filename_csv, mode='w', newline='') as file:
        writer = csv.writer(file)
        # Write header
```

```
152         writer.writerow(['Bubble ID', 'Frame #', 'X Position', 'Y Position', 'Radius'])
153         # Write bubble positions and radius
154         for bubble_id, data in bubble_positions.items():
155             for frame, (x, y) in enumerate(data['positions']):
156                 radius = data['radius'][frame]
157                 writer.writerow([bubble_id, frame, x, y, radius])
158 
159     pygame.quit()
```

Listing 1: The Python code for the Bubble Backscatter Simulator, commit version: `34b9a82` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/bubble-backscatter-simulator/blob/main/app.py

Appendix B Lossless Video Recorder

B.1 Description of Class Fields & Methods

Name	Description
`REC_FPS`	The recording frame rate.
`REC_WIDTH`	The horizontal resolution of recording.
`REC_HEIGHT`	The vertical resolution of recording.
`FB_WIDTH`	The horizontal resolution of the Framebuffer.
`FB_HEIGHT`	The vertical resolution of the Framebuffer.
`FB_DEPTH`	The bit depth of the Framebuffer.

Table 5: Software configuration constants and their descriptions from the Lossless Raspberry Pi Camera Recorder program.

B.1.1 The 'ProjectorManager' Module

Name	Description
`width`	The horizontal resolution of the Framebuffer device, same as `FB_WIDTH`.
`height`	The vertical resolution of the Framebuffer device, same as `FB_HEIGHT`.
`height`	The Framebuffer bit depth, same as `FB_DEPTH`.
`status`	Stores `True` if the projector light is 'on', `False` if 'off'.

Table 6: Fields and their descriptions from the Projector class of the 'ProjectorManager' module from the Lossless Raspberry Pi Camera Recorder program.

Name	Description
`__init__()`	Initialises a new `Projector` object, populating the Framebuffer fields for `FB_WIDTH`, `FB_HEIGHT`, and `FB_DEPTH`. Initialising the bitmaps for light 'on' and 'off', and setting the light status to 'off'.
`getStatus()`	A simple 'getter' function for the `status` class field.
`on()`	Writes the 'on' bitmap to the Framebuffer memory location, thus turning on the projector light source.
`off()`	Writes the 'off' bitmap to the Framebuffer memory location, thus turning off the projector light source.
`toggle()`	Writes the 'on' bitmap to the Framebuffer memory location if the `status` stores 'off', and vice-versa, thus toggling on the projector light source.

Table 7: Methods and their descriptions from the Projector class of the 'ProjectorManager' module from the Lossless Raspberry Pi Camera Recorder program.

B.1.2 The 'CameraManager' Module

Name	Description
`picam2`	Stores the initialised `Picamera2` instance.
`recording`	Stores `True` if the recording is in progress, otherwise stores `True`.
`recording_filename`	Stores the filename which the recorded file will be saved as.
`recording_start_ts`	Stores the timestamp of when the recording was started.
`recording_end_ts`	Stores the timestamp of when the recording was stopped.

Table 8: Fields and their descriptions from the Camera class of the 'CameraManager' module from the Lossless Raspberry Pi Camera Recorder program.

Name	Description
`__init__()`	Initalises and configures the `Picamera2` instance by setting the recording resolution from `REC_WIDTH` and `REC_HEIGHT`, the frame rate from `REC_RATE`, capture stream as the 'main' output stream, and an ISP output format as 'BGR888', and finally starts the RPi Camera module.
`getStatus()`	A simple 'getter' function for the `recording` class field.
`captureFrame()`	Retrieves and returns the camera's sensor output array.
`startRecording()`	Sets the `recording` class field to `True`, then initialises a memory buffer, logging the timestamp for `recording_start_ts` before finally initialising the 'Null' encoder and starting the recording.
`stopRecording()`	Sets the `recording` class field to `False`, then stops the 'Picamera2' recording, before applying the lossless FFV1 encoding to the footage bitstream, then finally writing to a file and closing the memory buffer.
`toggleRecording()`	Starts a recording if not already recording, and vice-versa, thus toggling the Pi Camera recording.
`shutdown()`	Stops recording if there is one in progress before finally sending a graceful shutdown signal to the 'Picamera2' module to stop and disconnect the Pi Camera.

Table 9: Methods and their descriptions from the Camera class of the 'CameraManager' module from the Lossless Raspberry Pi Camera Recorder program.

B.1.3 The 'PreviewManager' Module

Name	Description
`__init__()`	Initalises a new OpenCV-based GUI window.
`getKeypress()`	Retrieves a keypress using the OpenCV GUI functions.
`shutdown()`	Destroys the OpenCV GUI window.
`showFrame()`	Renders the image frame on the GUI window and overlays the status variable text using OpenCV.

Table 10: Methods and their descriptions from the Preview class of the 'PreviewManager' module from the Lossless Raspberry Pi Camera Recorder program.

B.2 Python Code

B.2.1 Entry Point: 'app.py'

```
""" Records video from a Raspberry Pi Camera. """

from ProjectorManager import Projector
from CameraManager import Camera
from PreviewManager import Preview

# Recording parameters
REC_FPS = 30            # Max available framerate is 60 FPS
REC_WIDTH = 728         # Set to half of full sensor width
REC_HEIGHT = 544        # Set to half of full sensor height

# Framebuffer parameters
FB_WIDTH = 1920
FB_HEIGHT = 1080
FB_DEPTH = 16

if __name__ == "__main__":
    # Create preview window instance
    preview = Preview()

    # Initialise framebuffer - reset fb with light source off
    light = Projector(FB_WIDTH, FB_HEIGHT, FB_DEPTH, False)

    # Initialise Pi camera
    camera = Camera(REC_WIDTH, REC_HEIGHT, REC_FPS)

    # This is the main runtime loop
    while True:
        # Retrieve a frame from the Pi camera for the preview window
        preview.showFrame(
            camera.captureFrame(),
            camera.getStatus(),
            light.getStatus()
        )

        # Read key press
        keypress = preview.getKeypress()

        # Toggle light when 'l' key is pressed
        if keypress == ord('l'):
            light.toggle()

```

```
43
44          # Toggle recording when 'r' key is pressed
45          if keypress == ord('r'):
46              camera.toggleRecording()
47
48
49          # Exit program when 'e' key is pressed
50          if keypress == ord('e'):
51              # Send shutdown request to camera instance
52              camera.shutdown()
53
54              # When exiting, reset framebuffer with zero array
55              light.off()
56
57              # Send shutdown request to preview instance
58              preview.shutdown()
59
60              # Exit
61              break
62
63      # Exit from the script
64      exit()
```

Listing 2: The Python code for the 'app.py' entry point to the Lossless Raspberry Pi Camera Recorder, commit version: `30efb5a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/picam-video-recorder/blob/main/app.py

B.2.2 ProjectorManger: 'ProjectorManager.py'

```
1  import numpy as np
2
3  class Projector:
4      """
5      A simple object which controls the framebuffer to drive a HDMI-interfaced light source.
6      """
7
8      def __init__(self, width, height, depth, status):
9          self.width = width
10         self.height = height
11         self.depth = depth
12         self.status = status
13
14         # Initialise framebuffer outputs
15         self.light_off_output = np.full(
16             (self.height, self.width),  # fill into array of this size
17             0,                          # fill array with this value (black/off = 0)
18             dtype=np.uint16
19         ).reshape(-1)                   # flatten to 1d array for writing to framebuffer
20
21         self.light_on_output = np.full(
22             (self.height, self.width),  # fill into array of this size
23             (2 ** self.depth - 1),      # Maximum value for given colour depth (white)
24             dtype=np.uint16
25         ).reshape(-1)                   # flatten to 1d array for writing to framebuffer
26
27         # Initialise the light to the given status
28         if self.status:
29              self.on()
30         else:
31              self.off()
32
```

```
33      def getStatus(self):
34          """ Return whether or not light is on. """
35          return self.status
36
37      def on(self):
38          """ Turn on the light. """
39          try:
40              with open('/dev/fb0', 'wb') as buf:
41                   buf.write(self.light_on_output.tobytes())
42                   self.status = True
43          except Exception as e:
44              print("Error: couldn't turn on FB light:", e)
45
46      def off(self):
47          """ Turn off the light. """
48          try:
49              with open('/dev/fb0', 'wb') as buf:
50                  buf.write(self.light_off_output.tobytes())
51                  self.status = False
52          except Exception as e:
53              print("Error: couldn't turn off FB light:", e)
54
55      def toggle(self):
56          """ Toggle the light. """
57          # Turn off if already on
58          if self.status:
59              self.off()
60          # Turn on light if already off
61          else:
62              self.on()
```

Listing 3: The Python code for 'ProjectorManager.py' in the Lossless Raspberry Pi Camera Recorder, commit version: `30efb5a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/picam-video-recorder/blob/main/ProjectorManager.py

B.2.3 CameraManager: 'CameraManager.py'

```
1  import io
2  import os
3  import subprocess
4  from picamera2 import Picamera2
5  from picamera2.encoders import Encoder
6  from picamera2.outputs import FileOutput
7  from datetime import datetime
8
9  class Camera:
10     """
11     A wrapper to interface with the Pi camera with implementations to capture a frame for
       preview, start and stop a recording.
12     This interface will record and capture the 'main' stream which is set up as the processed
       'raw' feed.
13     """
14
15     def __init__(self, width, height, rate):
16
17         # Initialise Pi camera instance
18         self.picam2 = Picamera2()
19
20         # Pi camera configuration:
21         self.picam2.video_configuration.size = (width, height)      # Capture video at defined
        resolution/size
```

```
22          self.picam2.video_configuration.controls.FrameRate = rate   # Capture video at defined
     framerate
23          self.picam2.video_configuration.encode = 'main'             # Use 'main' stream for
    encoder
24          self.picam2.video_configuration.format = 'BGR888'           # Use this format for
    OpenCV compatibility
25          self.picam2.video_configuration.align()                     # Align stream size (THIS
    CHANGES THE RES SLIGHTLY!)
26
27          # Apply the configuration to the Pi camera
28          self.picam2.configure("video")
29
30          # Print camera configurations to confirm correct set up
31          print(self.picam2.camera_configuration()['sensor'])
32          print(self.picam2.camera_configuration()['main'])
33          print(self.picam2.camera_configuration()['controls'])
34
35          # Start the Pi camera
36          self.picam2.start()
37
38          # Variable to store whether recording is in progress
39          self.recording = False
40
41          # Initialise variable to store recording filename and timestamps
42          self.recording_filename = ""
43          self.recording_start_ts = None
44          self.recording_stop_ts = None
45
46      def getStatus(self):
47          """ Return whether or not recording is in progress. """
48          return self.recording
49
50      def captureFrame(self):
51          """ Capture a frame from the camera for constructing a frame-by-frame preview feed.
    """
52          return self.picam2.capture_array("main")
53
54      def startRecording(self):
55          """ Start recording from the Pi camera to the memory buffer. """
56          # Only start is not already recording
57          if not self.recording:
58              # Set recording status
59              self.recording = True
60
61              # Initialise a memory buffer to store video recording
62              self.recording_membuff = io.BytesIO()
63              self.recording_output = FileOutput(self.recording_membuff)
64
65              # Generate starting timestamp
66              self.recording_start_ts = datetime.now()
67              self.recording_start_ts_str = self.recording_start_ts.strftime("%m-%d-%Y-%H-%M-%S"
    )
68
69              # Generate filename
70              self.recording_filename = "recording_" + self.recording_start_ts_str
71
72              # Reset recording end timestamp
73              self.recording_stop_ts = None
74
75              # Initialise a 'null' encoder to record without any encoding
76              self.recording_encoder = Encoder()
77
```

```
78              # Start the recording
79              self.picam2.start_recording(
80                  encoder=self.recording_encoder,  # Null encoder for 'raw' footage
81                  output=self.recording_output     # Record directly to memory
82              )
83
84              # Log to console
85              print("Started recording:", self.recording_filename)
86          else:
87              # Log to console if user requests recording start when already in progress
88              print("Cannot start recording, existing recording is in progress!")
89
90      def stopRecording(self):
91          """ Stop recording, unload from mamory buffer to disk and convert to MKV with the
    lossless FFV1 codec. """
92          # Only stop recording if already in progress
93          if self.recording:
94              # Stop the recording
95              self.picam2.stop_recording()
96
97              # stop_recording() entirely stops the Pi camera with stop()
98              self.picam2.start()
99
100             # Generate stopped timestamp
101             self.recording_stop_ts = datetime.now()
102             self.recording_stop_ts_str = self.recording_stop_ts.strftime("%m-%d-%Y-%H-%M-%S")
103
104             # Log to console
105             print("Stopped recording:"
106                 + self.recording_filename
107                 + " (Stopped at: "
108                 + self.recording_stop_ts_str
109                 + ")"
110             )
111             print("Please wait, unloading recording from memory buffer...")
112
113             # Save to file
114             with open(self.recording_filename, "xb") as file:
115                 file.write(self.recording_membuff.getbuffer())
116
117             # Reset recording start timestamp
118             self.recording_start_ts = None
119
120             # Close recording memory buffer to discard data and clear RAM space
121             self.recording_membuff.close()
122
123             # Generate encoded recording filename
124             self.recording_filename_mkv = self.recording_filename + ".mkv"
125
126             # Log to console
127             print("Please wait, encoding raw recording to lossless mkv...: " + self.
    recording_filename_mkv)
128
129             # Get the aligned resolution of the recording
130             (aligned_width, aligned_height) = self.picam2.camera_configuration()['main']['size
    ']
131
132             # FFmpeg command to convert video from BGR888 to MKV (lossless FFV1 codec)
133             ffmpeg_command = [
134                 'ffmpeg',                                   # call FFmpeg
135                 '-f', 'rawvideo',                           # force raw data type
136                 '-s', f'{aligned_width}x{aligned_height}',  # width and height
```

```
                '-pix_fmt', 'bgr24',                        # set BGR888 format
                '-i', self.recording_filename,              # filename of raw file
                '-c:v', 'ffv1',                             # lossless FFV1 codec
                '-y',                                       # overwrite output file if exists
                '-loglevel', 'error',                       # only show errors in console
                '-stats',                                   # show progress stats
                self.recording_filename_mkv                 # output filename
            ]

            # Run FFmpeg command as a subprocess
            subprocess.run(ffmpeg_command)

            # Delete raw file
            os.remove(self.recording_filename)

            # Log to console
            print("Write complete, it is safe to exit or record again: " + self.
    recording_filename_mkv)

            # Set the recording status to false to allow program exiting
            self.recording = False
        else:
            # If not already recording, log error to console
            print("Cannot stop recording, program is currently not recording anything!")

    def toggleRecording(self):
        """ Toggle the Pi camera recording. """
        # Turn off if already on
        if self.recording:
            self.stopRecording()
        # Turn on light if already off
        else:
            self.startRecording()

    def shutdown(self):
        """ Gracefully stop the Pi camera instance. """
        # If recording in progress, stop it
        if self.recording:
            print("Exit request received while recording - recording will be stopped.")
            self.stopRecording()

        # Stop the Pi camera instance.
        self.picam2.stop()

        # Log to console
        print("Camera instance has been gracefully stopped.")
```

Listing 4: The Python code for 'CameraManager.py' in the Lossless Raspberry Pi Camera Recorder, commit version: `30efb5a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/picam-video-recorder/blob/main/CameraManager.py

B.2.4 PreviewManager: 'PreviewManager.py'

```
import cv2 as cv

class Preview:
    """ Preview window which renders frame-by-frame to an OpenCV window. """

    def __init__(self):
        cv.namedWindow("Pi Camera Feed")

```

```
9       def getKeypress(self):
10          """ Waits 1ms for a keypress and returns it. """
11          return cv.waitKey(1)
12
13      def shutdown(self):
14          """ Destroys all OpenCV windows for a clean exit. """
15          cv.destroyAllWindows()
16
17      def showFrame(self, frame, recording, light):
18          """ Updates the window with the new frame, adds status text overlays. """
19
20          # Generate a text-based status for the light source
21          match light:
22              case True:
23                  light_status = "LON"
24              case False:
25                  light_status = "LOFF"
26
27          # Generate a text-based status for recording
28          match recording:
29              case True:
30                  recording_status = "REC"
31              case False:
32                  recording_status = ""
33
34          # Add the two status text together
35          overlay_text = light_status + "    " + recording_status
36
37          # Overlay the text to the frame
38          frame_overlay = cv.putText(
39              img=frame,
40              text=overlay_text,
41              org=(10, 25),
42              fontFace=cv.FONT_HERSHEY_SIMPLEX,
43              fontScale=0.75,
44              color=(255,0,0),
45              thickness=1,
46              lineType=cv.LINE_AA
47          )
48
49          # Render the frame
50          cv.imshow("Pi Camera Feed", frame)
```

Listing 5: The Python code for 'PreviewManager.py' in the Lossless Raspberry Pi Camera Recorder, commit version: `30efb5a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/picam-video-recorder/blob/main/PreviewManager.py

Appendix C Standard Backscatter Cancellation System

C.1 Description of Class Fields & Methods

Name	Description
VIDEO_CAPTURE_SOURCE	Set with directory path to input a list of individual frames, file path to input a video file, or integer '0' to use the RPi Camera input.
VIDEO_CAPTURE_WIDTH	The horizontal resolution of recording.
VIDEO_CAPTURE_HEIGHT	The vertical resolution of recording.
VIDEO_CAPTURE_CROP_HEIGHT	The vertical resolution to crop for the region of interest.
VIDEO_CAPTURE_CROP_WIDTH	The horizontal resolution to crop for the region of interest.
BS_MANAGER_DEBUG_WINDOWS	Whether to render debug windows showing the post-processed frame after intermediate processing stages.

Table 11: Software configuration constants and their descriptions from the Backscatter Cancellation program.

C.2 Python Code

C.2.1 Entry Point: 'app.py'

```
from datetime import datetime
import numpy as np
import pandas as pd
import psutil
import cv2
import os

from CaptureManager import FrameStream, VideoStream, PicameraStream
from WindowManager import Window
from BSManager import Detector
from TimeManager import Timer

OS_NICE_PRIORITY_LEVEL = 0
X11_XAUTHORITY_PATH = '/home/sid/.Xauthority'

### VIDEO CAPTURE SOURCE
#   Input a directory path to feed in a series of frame images,
#   named as an integer denoting the frame number and must be
#   in PNG format.
#
#   Input a file path to feed in a video file.
#
#   Input integer '0' to use Picamera2 capture_array() to capture
#   feed frame-by-frame.
# VIDEO_CAPTURE_SOURCE = 0
# VIDEO_CAPTURE_SOURCE = "./import_04-08-2024-14-35-53/"
# VIDEO_CAPTURE_SOURCE = "../Tank Recordings/recording_04-22-2024-14-36-49.mkv"
VIDEO_CAPTURE_SOURCE = "../Tank Recordings/recording_04-22-2024-14-16-08.mkv"

### VIDEO CAPTURE PARAMETERS
#   WIDTH & HEIGHT:
#       These are the recording parameters which dictate capture
#       resolution.
#
```

```
#       When wanting to use the frame-by-frame output from the
#       bubble-backscatter-simulation program, set these values
#       to the same as the ones input to that program (800x600).
#
#       When wanting to use a pre-recorded video source, these
#       values will be updated to match the correct resolution
#       of the video. Ensure they are similar to avoid confusion.
#
#       Want wanting to use the Pi Camera feed, these values will
#       be used when configuring the camera resolution parameters,
#       however, the camera will align the stream size to force
#       optimal alignment, so the resolution may be slightly
#       different.
#
#   CROP WIDTH & HEIGHT:
#       Apply a crop to isolate the region of interest, mitigating
#       the submersible housing offsets
VIDEO_CAPTURE_WIDTH = 800
VIDEO_CAPTURE_HEIGHT = 600
VIDEO_CAPTURE_CROP_HEIGHT = (0, 320)
VIDEO_CAPTURE_CROP_WIDTH = (430, 700)

### PREVIEW WINDOW NAMES
#   These constants store the names for each GUI window
INPUT_PREVIEW_WINDOW_NAME = "Input Feed"
SEGMENTATION_PREVIEW_WINDOW_NAME = "Backscatter Segmentation"
PROJECTOR_PREVIEW_WINDOW_NAME = "Projected Light Pattern"

### BSMANAGER PARAMETERS
#   CANNY_THRESHOLD_SIGMA: Threshold for the zero-parameter
#   Canny implementation - (https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/)
#
#   BS_MANAGER_DEBUG_WINDOWS: Whether or not to display the intermediate
#   step visualisation.
CANNY_THRESHOLD_SIGMA = 0
BS_MANAGER_HISTOGRAM_EQUALISATION = True
BS_MANAGER_DEBUG_WINDOWS = True
BINARY_THRESHOLD_THRESH = (200, 255)


if __name__ == "__main__":
    # Required to run X11 forwarding as sudo
    os.environ['XAUTHORITY'] = X11_XAUTHORITY_PATH

    # Set the OS priority level
    p = psutil.Process(os.getpid())
    print("Current OS priority: ", p.nice())
    p.nice(OS_NICE_PRIORITY_LEVEL)
    print("New OS priority: ", p.nice())

    # Generate CSV export filename
    export_filename_csv = "export_" + datetime.now().strftime("%m-%d-%Y-%H-%M-%S") + ".csv"

    # Initialise capture stream
    match VIDEO_CAPTURE_SOURCE:
        # If int 0 then set up Pi camera stream
        case 0:
            # Initialise FrameStream
            stream = PicameraStream(VIDEO_CAPTURE_WIDTH, VIDEO_CAPTURE_HEIGHT)
        # If not int 0 then check if it is a valid path
        case _:
```

```
            # If the path is a directory, then FrameStream
            if os.path.isdir(VIDEO_CAPTURE_SOURCE):
                # Initialise FrameStream
                stream = FrameStream(VIDEO_CAPTURE_SOURCE)
            # If the path is a file, then VideoStream
            elif os.path.isfile(VIDEO_CAPTURE_SOURCE):
                # Initialise VideoStream
                stream = VideoStream(
                    source=VIDEO_CAPTURE_SOURCE,
                    crop_w=VIDEO_CAPTURE_CROP_WIDTH,
                    crop_h=VIDEO_CAPTURE_CROP_HEIGHT,
                    loop=True
                )

    # Initialise window to display the input
    input_feed_window = Window(INPUT_PREVIEW_WINDOW_NAME)

    # Initialise window to display the particle segmentation
    segmentation_window = Window(SEGMENTATION_PREVIEW_WINDOW_NAME)

    # Initialise window to display the projector output
    projector_window = Window(PROJECTOR_PREVIEW_WINDOW_NAME)

    # Initialise the backscatter detector
    detector = Detector(
        canny_threshold=CANNY_THRESHOLD_SIGMA,
        thresh_threshold=BINARY_THRESHOLD_THRESH,
        debug_windows=BS_MANAGER_DEBUG_WINDOWS
    )

    # Initialise variable to track frame processing duratione (sec)
    total_frame_processing_time = 0

    # Initialise a Pandas DataFrame to log real-time metrics
    rt_metrics_df = pd.DataFrame(
        columns=[
            'Capture Duration (s)',
            'Greyscale Conversion Duration (s)',
            'Gaussian Blur Duration (s)',
            'Histogram Equalisation Duration (s)',
            'Binary Thresholdind Duration (s)',
            'Canny Algorithm Duration (s)',
            'CV2 findContours() Duration (s)',
            'CV2 minEnclosingCircle() Duration (s)',
            'Number of MECs on screen',
            'Total frame processing time (s)'
        ]
    )

    while True:
        # Start timer to calculate capture duration
        timer = Timer()

        # Read a frame
        frame = stream.read()

        # Stop timer
        capture_duration = timer.stop()

        # Detect keypress
        keypress = cv2.waitKey(1)

```

```
        # Exit if the 'e' key is pressed
        if keypress == ord('e'):
            break

        # While there are frames...
        if frame is not None:
            # Update the input visualisation window
            input_feed_window.update(frame)

            # Start timer for the total frame processing duration
            timer = Timer()

            # Detect the particles and retrieve real-time metrics
            particles, metrics = detector.detect(frame)

            # Stop the total frame processing duration timer
            frame_processing_time = timer.stop()

            # Create a black mask for the segmentation preview
            particle_mask = np.copy(frame)

            # Draw white circles on the black mask for each MEC
            for particle in particles:
                cv2.circle(
                    particle_mask,
                    particle[0],
                    particle[1],
                    (0, 0, 255),
                    1
                )

            # Display the black mask with white circles
            segmentation_window.update(particle_mask)

            # Create a white mask for the projector preview
            projector_mask = np.ones_like(frame) * 255

            # Draw black circles on the black mask for each MEC
            for particle in particles:
                cv2.circle(
                    projector_mask,
                    particle[0],
                    particle[1],
                    (0, 0, 0),
                    -1
                )

            # Display the white mask with black circles
            projector_window.update(projector_mask)

            # Prepend the capture time to the metrics
            metrics = [capture_duration] + metrics

            # Append metrics list to end of dataframe
            rt_metrics_df.loc[len(rt_metrics_df)] = metrics
        else:
            # break out of the while loop when there are no more frames
            break

    # Export dataframe as CSV
    rt_metrics_df.to_csv(
        path_or_buf=export_filename_csv,
```

```
        encoding='utf-8'
    )

    cv2.destroyAllWindows()
```

Listing 6: The Python code for the 'app.py' entry point to the Backscatter Cancellation System program, commit version: `46b962a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/main/app.py

C.2.2 CaptureManager: 'CaptureManager.py'

```
import os
import cv2
import logging
from queue import Queue
from threading import Thread
from natsort import natsorted


class PicameraStream:
    """ Streams from a Picamera frame-by-frame. """

    def __init__(self, width, height):
        # Try to import and use Picamera2 library - this will fail on non-RPi platforms
        try:
            # Pylint - Ignore the missing import error on non RPi platforms
            from picamera2 import Picamera2     # type: ignore

            # Initialise Pi camera instance
            self.picam2 = Picamera2()

            # Pi camera configuration:
            self.picam2.still_configuration.size = (width, height)      # Capture stills at defined resolution/size
            self.picam2.still_configuration.format = 'BGR888'           # Use this format for OpenCV compatibility
            self.picam2.still_configuration.align()                     # Align stream size (THIS CHANGES THE RES SLIGHTLY!)

            # Apply the configuration to the Pi camera
            self.picam2.configure("video")

            # Print camera configurations to confirm correct set up
            logging.debug(self.picam2.camera_configuration()['sensor'])
            logging.debug(self.picam2.camera_configuration()['main'])
            logging.debug(self.picam2.camera_configuration()['controls'])

            # Start the Pi camera
            self.picam2.start()

            # Return the aligned stream resolution
            (aligned_width, aligned_height) = self.picam2.camera_configuration()['main']['size']
            return (aligned_width, aligned_height)

        # If the Picamera2 module is not found, then the program is probably not running on a RPi
        except ImportError:
            logging.error("Picamera2 module not found. Make sure you are running this on a Raspberry Pi.")
        # No other exceptions should be encountered, if they are then log it
```

```
45          except Exception as e:
46              logging.error("Unforseen error encountered: ", e)
47
48      def read(self):
49          """ Capture camera sensor array for constructing a frame-by-frame feed. """
50
51          success, frame = True, self.picam2.capture_array("main")
52          return success, frame
53
54      def exit(self):
55          """ Gracefully stop the Pi camera instance. """
56
57          # Stop the Pi camera instance.
58          self.picam2.stop()
59
60          # Log to console
61          logging.info("Camera instance has been gracefully stopped.")
62
63  class FrameStream:
64      """ Streams a video represented by a set of PNG images for each frame. """
65
66      def __init__(self, path):
67          # Initialise path to folder
68          self.path = path
69
70          # Initialise queue to store frames in memory
71          self.q = Queue()
72
73          # Populate queue with image frames
74          self._load_frames()
75
76      def _load_frames(self):
77          """ Internal method that populates the image frame queue. """
78
79          logging.debug("Beginning to populate frame queue...")
80
81          # Sort all of the frame files inside the folder
82          frames = natsorted(os.listdir(self.path))
83
84          # For each file in the sorted list of frame files...
85          for file in frames:
86              # Ensure that the file is a PNG
87              if file.endswith('.png'):
88                  # Extract the frame number from the filename
89                  frame_num = int(os.path.splitext(file)[0])
90                  # Generate the filepath
91                  frame_path = os.path.join(self.path, file)
92                  # Read in the file
93                  frame = cv2.imread(frame_path)
94                  # Store the file in the queue along with the frame number
95                  self.q.put(frame)
96
97          # Log status
98          logging.debug("Frame queue has been populated.")
99
100     def read(self):
101         """ Dequeues the next frame in the sequence. """
102
103         if not self.empty():
104             return self.q.get()
105         else:
106             return None
```

```
107
108     def empty(self):
109         """ Returns true if there are no more frames to stream. """
110
111         return self.q.empty()
112
113 class VideoStream:
114     """ Streams an input video file. """
115
116     def __init__(self, source, crop_w=None, crop_h=None, loop=False):
117         # initialise the OpenCV stream
118         self.capture = cv2.VideoCapture(source)
119         self.crop_w = crop_w
120         self.crop_h = crop_h
121         self.loop = loop
122
123         # check if capture is accessible
124         if not self.capture.isOpened():
125             logging.error("Cannot open video stream!")
126             raise Exception("Cannot open video stream!")
127
128         # calculate FPS and FPT of the capture
129         self.target_fps = self.capture.get(cv2.CAP_PROP_FPS)
130         self.target_fpt = (1 / self.target_fps) * 1000
131
132     def read(self):
133         _, frame = self.capture.read()
134         if frame is not None:
135             if (self.crop_w is not None) and (self.crop_h is not None):
136                 frame = frame[self.crop_h[0]:self.crop_h[1], self.crop_w[0]:self.crop_w[1]]
137         elif self.loop is True:
138             self.capture.set(cv2.CAP_PROP_POS_FRAMES, 0)
139             _, frame = self.capture.read()
140         return frame
141
142     def exit(self):
143         self.capture.release()
144
145 class t_VideoStream:
146     """ Streams an input video file using threading. """
147
148     def __init__(self, source, queueSize=4096):
149         # initialise the OpenCV stream
150         self.capture = cv2.VideoCapture(source)
151         # initialise parameter that stops video stream
152         self.stopped = False
153         # initialise the queue for pushing frames
154         self.queue = Queue(maxsize=queueSize)
155
156     def start(self):
157         # start a thread to read frames from stream
158         t = Thread(
159             target=self._update,
160             args=()
161         )
162         # allow thread to be killed when main app exits
163         t.daemon = True
164         # start the thread
165         t.start()
166         return self
167
168     def _update(self):
```

```
        while True:
            # stop reading if stream is stopped
            if self.stopped:
                return

            # otherwise, keep reading and queuing until queue is full
            if not self.queue.full():
                # read the next frame from the file
                success, frame = self.capture.read()

                # check if we have reached the end of video stream
                if not success:
                    self.stop()
                    return

                # push the frame to the queue
                self.queue.put(frame)

    def read(self):
        # return a frame from the queue
        return self.queue.get()

    def stop(self):
        # indicate that thread should be stopped
        self.stopped = True
        self.capture.release()

    def empty(self):
        # returns True if queue is empty
        if self.queue.qsize():
            return True
        return False
```

Listing 7: The Python code for 'CaptureManager.py' in the Backscatter Cancellation System program, commit version: `1bf8131` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/main/CaptureManager.py

C.2.3 BSManager: 'BSManager.py'

```
import cv2
import numpy as np

from WindowManager import Window
from TimeManager import Timer

### DEBUG WINDOW NAMES
#   These constants store the names for each GUI window
GREYSCALE_DEBUG_WINDOW_NAME = "BSDetector Debug: Greyscale"
GAUSBLUR_DEBUG_WINDOW_NAME = "BSDetector Debug: Gaussian Blur"
CANNY_DEBUG_WINDOW_NAME = "BSDetector Debug: Canny Algorithm"
HISTEQU_DEBUG_WINDOW_NAME = "BSDetector Debug: Histogram Equalisation"
TRESHOLD_DEBUG_WINDOW_NAME = "BSDetector Debug: Binary Thresholding"

class Detector:
    """ Backscatter detection logic (V1): (a) edges are detected using the Canny algorithm, (b) the detected edges are segmented using a simple method - minimum enclosing circle (MEC), (c) the centre coordinates and radius of the detected particles (MECs) are returned. """

    def __init__(self, canny_threshold, thresh_threshold, debug_windows=True):
        # Zero-parameter threshold for canny (https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/)
```

```
20          self.canny_threshold = canny_threshold
21
22          self.thresh_threshold = thresh_threshold
23
24          # Whether or not to print the intermediate step visualisation
25          self.debug_windows = debug_windows
26
27          # Initialise the debug windows if enabled
28          if self.debug_windows:
29              self.greyscale_window = Window(GREYSCALE_DEBUG_WINDOW_NAME)
30              self.gausblur_window = Window(GAUSBLUR_DEBUG_WINDOW_NAME)
31              self.canny_window = Window(CANNY_DEBUG_WINDOW_NAME)
32              # self.histequ_window = Window(HISTEQU_DEBUG_WINDOW_NAME)
33              self.threshold_window = Window(TRESHOLD_DEBUG_WINDOW_NAME)
34
35      def _greyscale(self, frame):
36          """ (Internal) Applies a greyscale filter to the frame. """
37
38          # Log start timestamp
39          timer = Timer()
40
41          # apply the single-channel conversion with greyscale filter
42          greyscale = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
43
44          # Calculate process duration
45          duration = timer.stop()
46
47          # output to the debug window if enabled
48          if self.debug_windows:
49              self.greyscale_window.update(greyscale)
50
51          # return the greyscaled frame
52          return greyscale, duration
53
54      def _threshold(self, frame):
55          """ (Internal) Applies a binary threshold to the frame. """
56
57          # Log start timestamp
58          timer = Timer()
59
60          # apply binary threshold
61          _, thresh = cv2.threshold(
62              frame,
63              self.thresh_threshold[0],
64              self.thresh_threshold[1],
65              cv2.THRESH_BINARY
66          )
67
68          # Calculate process duration
69          duration = timer.stop()
70
71          # output to the debug window if enabled
72          if self.debug_windows:
73              self.threshold_window.update(thresh)
74
75          # return the thresholded
76          return thresh, duration
77
78      def _histequ(self, frame):
79          """ (Internal) Applies a histogram equalisation to the frame. """
80
81          # Log start timestamp
```

```
82          timer = Timer()
83
84          # apply histogram equalisation
85          histequ = cv2.equalizeHist(frame)
86
87          # Calculate process duration
88          duration = timer.stop()
89
90          # output to the debug window if enabled
91          if self.debug_windows:
92              self.histequ_window.update(histequ)
93
94          # return the histogram equalised frame
95          return histequ, duration
96
97      def _gausblur(self, frame):
98          """ (Internal) Applies a Gaussian blur to the frame. """
99
100         # Log start timestamp
101         timer = Timer()
102
103         # apply the Gaussian blur
104         gausblur = cv2.GaussianBlur(frame, (5,5), 0)
105
106         # Calculate process duration
107         duration = timer.stop()
108
109         # output to the debug window if enabled
110         if self.debug_windows:
111             self.gausblur_window.update(gausblur)
112
113         # return the Gaussian blurred frame
114         return gausblur, duration
115
116     def _canny(self, frame):
117         """ (Internal) Applies the Canny algorithm to the frame. """
118         # Log start timestamp of Canny process
119         timer = Timer()
120
121         # compute the median single-channel pixel intensities
122         gaus_median = np.median(frame)
123         # compute threshold values for canny using single parameter Canny
124         lower_threshold = int(max(0, (1.0 - self.canny_threshold) * gaus_median))
125         upper_threshold = int(min(255, (1.0 + self.canny_threshold) * gaus_median))
126
127         # perform Canny edge detection
128         edges = cv2.Canny(
129             frame,
130             lower_threshold,
131             upper_threshold
132         )
133
134         # Calculate the Canny process duration
135         duration = timer.stop()
136
137         # output to the debug window if enabled
138         if self.debug_windows:
139             self.canny_window.update(edges)
140
141         return edges, duration
142
143     def _segmentation(self, edges):
```

```
144         """ (Internal) Finds contours using cv2.findContours() then calculates the minumum
    enclosing circles using cv2.minEnclosingCircle(). """
145
146         # Log start timestamp of findContours()
147         timer_findContours = Timer()
148
149         # RETR_EXTERNAL only retrieves the extreme outer contours
150         # CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and
151         #   diagonal segments and leaves only their end points
152         contours, _ = cv2.findContours(
153             edges,
154             cv2.RETR_EXTERNAL,       # RetrievalModes
155             cv2.CHAIN_APPROX_SIMPLE  # ContourApproximationModes
156         )
157
158         # Calculate findContours process duration
159         findContours_duration = timer_findContours.stop()
160
161         # Log start timestamp of minEnclosingCircle()
162         timer_segmentContours = Timer()
163
164         # List to store the particle information (centre coords + radius)
165         particles = []
166
167         for contour in contours:
168             # Find the minimum enclosing circle for each contour
169             ((x, y), radius) = cv2.minEnclosingCircle(contour)
170             # Find the centre
171             centre = (int(x), int(y))
172             # Find the radius
173             radius = int(radius)
174             # Store the information
175             particles.append((centre, radius))
176
177         # Calculate findContours process duration
178         segmentContours_duration = timer_segmentContours.stop()
179
180         # return the segmented particle information
181         return particles, findContours_duration, segmentContours_duration
182
183     def detect(self, input):
184         """ Detects the backscatter particles. Returns the particle coordinates and radius,
    and the real-time metrics. """
185
186         # single channel conversion using greyscaling
187         frame, greyscale_duration = self._greyscale(input)
188
189         # apply Gaussian blur noise reduction and smoothening, prep for Canny
190         frame, gausblur_duration = self._gausblur(frame)
191
192         # apply histogram equalisation to improve contrasts for better Canny
193         # frame, histequ_duration = self._histequ(frame)
194
195         # apply a binary threshold to isolate bright backscatter
196         frame, thresh_duration = self._threshold(frame)
197
198         # apply the Canny algorithm to the frame
199         edges, canny_duration = self._canny(frame)
200
201         # segment the particles
202         particles, findContours_duration, segmentContours_duration = self._segmentation(edges)
203
```

```
        # compile metrics into a list
        metrics = [
            greyscale_duration,
            gausblur_duration,
            # histequ_duration,
            0,
            thresh_duration,
            canny_duration,
            findContours_duration,
            segmentContours_duration
        ]

        # Get the total frame processing time, add it to metrics with total particles
        metrics = metrics + [len(particles)] + [sum(metrics)]

        # return detected particles
        return particles, metrics
```

Listing 8: The Python code for 'BSManager.py' in the Backscatter Cancellation System program, commit version: `46b962a` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/main/BSManager.py

C.2.4 TimeManager: 'TimeManager.py'

```
import time

class Timer:
    """ Time durations accurately using the performance counter. """

    def __init__(self):
        """ Start the timer on initialisation. """
        self.start()

    def start(self):
        """ Starts the timer. """

        self.start_timestamp = time.perf_counter()
        self.end_timestamp = 0
        self.duration = 0

    def stop(self):
        """ Stops the timer. """

        self.end_timestamp = time.perf_counter()
        self.duration = self.end_timestamp - self.start_timestamp

        # return (self.start_timestamp, self.end_timestamp, self.duration)
        return self.duration
```

Listing 9: The Python code for 'TimeManager.py' in the Backscatter Cancellation System program, commit version: `2367e84` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/main/TimeManager.py

C.2.5 WindowManager: 'WindowManager.py'

```
import cv2
import numpy as np

class Window:
    """ A GUI window manager utilising OpenCV. """
```

```
6
7       def __init__(self, name, frame=None):
8           self.name = name
9
10          # created OpenCV named window
11          cv2.namedWindow(self.name)
12
13          # initialise window with input frame or test img
14          if frame:
15              cv2.imshow(self.name, frame)
16          else:
17              # create black image 800x600
18              test = np.zeros((600, 800, 3), dtype=np.uint8)
19
20              # Generate a rainbow gradient along the height (600)
21              for i in range(600):
22                  hue = int(180 * i / 600)  # Vary the hue from 0 to 180
23                  colour = list(
24                      map(
25                          int,
26                          cv2.cvtColor(
27                              np.array(
28                                  [[[hue,255,255]]],
29                                  dtype=np.uint8
30                              ),
31                              cv2.COLOR_HSV2BGR
32                          )[0, 0]
33                      )
34                  )
35                  test[i, :, :] = colour
36
37              # display rainbow test image
38              cv2.imshow(self.name, test)
39
40      def update(self, frame):
41          """ Updates the window with the given frame. """
42          cv2.imshow(self.name, frame)
43
44      def destroy(self):
45          """ Gracefully terminates the window instance. """
46          cv2.destroyWindow(self.name)
```

Listing 10: The Python code for 'WindowManager.py' in the Backscatter Cancellation System program, commit version: `407ac01` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/main/WindowManager.py

Appendix D Multiprocessing Backscatter Cancellation System

D.1 Python Code

D.1.1 Entry Point: 'app2.py'

```
1 from multiprocessing import Process, Queue
2 from datetime import datetime
3 import pandas as pd
4 import numpy as np
```

```
5  import logging
6  import psutil
7  import cv2
8  import os
9
10 from CaptureManager import FrameStream, VideoStream, PicameraStream
11 from WindowManager import Window
12 from TimeManager import Timer
13
14 OS_NICE_PRIORITY_LEVEL = -20
15 X11_XAUTHORITY_PATH = '/home/sid/.Xauthority'
16
17 ### VIDEO CAPTURE SOURCE
18 #   Input a directory path to feed in a series of frame images,
19 #   named as an integer denoting the frame number and must be
20 #   in PNG format.
21 #
22 #   Input a file path to feed in a video file.
23 #
24 #   Input integer '0' to use Picamera2 capture_array() to capture
25 #   feed frame-by-frame.
26 VIDEO_CAPTURE_SOURCE = "./import_04-08-2024-14-35-53/"
27
28 ### VIDEO CAPTURE RESOLUTION
29 #   These are the recording parameters which dictate capture
30 #   resolution.
31 #
32 #   When wanting to use the frame-by-frame output from the
33 #   bubble-backscatter-simulation program, set these values
34 #   to the same as the ones input to that program (800x600).
35 #
36 #   When wanting to use a pre-recorded video source, these
37 #   values will be updated to match the correct resolution
38 #   of the video. Ensure they are similar to avoid confusion.
39 #
40 #   Want wanting to use the Pi Camera feed, these values will
41 #   be used when configuring the camera resolution parameters,
42 #   however, the camera will align the stream size to force
43 #   optimal alignment, so the resolution may be slightly
44 #   different.
45 VIDEO_CAPTURE_WIDTH = 800
46 VIDEO_CAPTURE_HEIGHT = 600
47
48 ### PREVIEW/DEBUG WINDOW NAMES
49 #   These constants store the names for each GUI window
50 INPUT_PREVIEW_WINDOW_NAME = "Input Feed"
51 PROJECTOR_PREVIEW_WINDOW_NAME = "Projected Light Pattern"
52 GREYSCALE_DEBUG_WINDOW_NAME = "BSDetector Debug: Greyscale"
53 GAUSBLUR_DEBUG_WINDOW_NAME = "BSDetector Debug: Gaussian Blur"
54 CANNY_DEBUG_WINDOW_NAME = "BSDetector Debug: Canny Algorithm"
55 CONTOUR_DEBUG_WINDOW_NAME = "BSDetector Debug: Detected Contours"
56 HISTEQU_DEBUG_WINDOW_NAME = "BSDetector Debug: Histogram Equalisation"
57
58 ### BSMANAGER PARAMETERS
59 #   CANNY_THRESHOLD_SIGMA: Threshold for the zero-parameter
60 #   Canny implementation - (https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-
      canny-edge-detection-with-python-and-opencv/)
61 #
62 #   BS_MANAGER_HISTOGRAM_EQUALISATION: Whether or not to carry
63 #   out the histogram equalisation step.
64 #
65 #   BS_MANAGER_DEBUG_WINDOWS: Whether or not to display the intermediate
```

```
#   step visualisation.
CANNY_THRESHOLD_SIGMA = 0.33
BS_DEBUG_WINDOWS = False

LOGGING_FILEPATH = ""
LOGGING_FILEPATH_S1 = "export_log-s1.csv"
LOGGING_FILEPATH_S2 = "export_log-s2.csv"
LOGGING_FILEPATH_S3 = "export_log-s3.csv"
LOGGING_FILEPATH_S4 = "export_log-s4.csv"
LOGGING_FILEPATH_S5 = "export_log-s5.csv"
LOGGING_FILEPATH_S6 = "export_log-s6.csv"
LOGGING_FILEPATH_S7 = "export_log-s7.csv"
LOGGING_FILEPATH_S8 = "export_log-s8.csv"

PROCESS_QUEUE_QUIT_SIGNAL = "QUIT"
PROCESS_QUEUE_FQUIT_SIGNAL = "FQUIT"

class S1_Capture(Process):
    """ Acquires and enqueues frames from the capture source. """

    def __init__(self, output_q):
        """
        output_q: Queue to store captured frames and metrics.\n
        """
        super().__init__()
        self.output_q = output_q
        self.capture_duration = 0

    def run(self,):
        # Turn on logging
        logging.basicConfig(
            level=logging.DEBUG,
            format='%(created).6f,%(levelname)s,S1_Capture,%(message)s',
            filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S1,
            filemode='w'
        )

        # Log the start
        logging.info("Process started with PID=%d", self.pid)

        # Initialise capture stream
        match VIDEO_CAPTURE_SOURCE:
            # If int 0 then set up Pi camera stream
            case 0:
                # Log
                logging.debug("Initialising PicameraStream")
                # Initialise FrameStream
                stream = PicameraStream(VIDEO_CAPTURE_WIDTH, VIDEO_CAPTURE_HEIGHT)
                # Log
                logging.debug("PicameraStream initialised")
            # If not int 0 then check if it is a valid path
            case _:
                # If the path is a directory, then FrameStream
                if os.path.isdir(VIDEO_CAPTURE_SOURCE):
                    # Log
                    logging.debug("Initialising FrameStream")
                    # Initialise FrameStream
                    stream = FrameStream(VIDEO_CAPTURE_SOURCE)
                    # Log
                    logging.debug("FrameStream initialised")
                # If the path is a file, then VideoStream
                elif os.path.isfile(VIDEO_CAPTURE_SOURCE):
```

```
128                     # Log
129                     logging.debug("Initialising VideoStream")
130                     # Initialise VideoStream
131                     stream = VideoStream(VIDEO_CAPTURE_SOURCE)
132                     # Log
133                     logging.debug("VideoStream initialised")
134 
135         # Initialise window to display the input
136         input_feed_window = Window(INPUT_PREVIEW_WINDOW_NAME)
137 
138         # Counter to track frames that have been read
139         frame_count = 0
140 
141         # Exit status
142         exit = False
143 
144         while not stream.empty():
145             # Log frame retrieval
146             logging.debug("Retrieving frame %d", frame_count)
147 
148             # Track capture duration
149             timer = Timer()
150             # Capture the frame
151             frame, _ = stream.read()
152             # Stop the timer
153             duration = timer.stop()
154 
155             # Log the retrieval
156             logging.debug("Retrieved frame %d", frame_count)
157 
158             # Enqueue frame and metrics
159             self.output_q.put_nowait((frame, [duration]))
160 
161             # Log the frame enqueue
162             logging.debug("Enqueued frame %d", frame_count)
163 
164             # Update the preview window
165             keypress = input_feed_window.update(frame)
166 
167             # Increment frame count
168             frame_count += 1
169 
170             # Forcefully exit when the 'e' key is pressed
171             if keypress == ord('e'):
172                 exit = True
173                 break
174 
175         # Handle event where capture has finished!
176         stream.exit()
177         if exit:
178             logging.info("User pressed exit key - sending quit signal")
179             self.output_q.put((PROCESS_QUEUE_FQUIT_SIGNAL, None), block=True, timeout=None)
180         else:
181             logging.info("No frames left to capture - sending quit signal")
182             self.output_q.put((PROCESS_QUEUE_QUIT_SIGNAL, None), block=True, timeout=None)
183 
184 class S2_Greyscale(Process):
185     """ Process that applies a greyscale filter to an input. """
186 
187     def __init__(self, input_q, output_q):
188         """
189         input_q: Queue that stores frames that require greyscaling prev. stage metrics.\n
```

```
        output_q: Queue that stores greyscaled frames and metrics.
        """
        super().__init__()
        self.input_q = input_q
        self.output_q = output_q


    def run(self,):
        # Turn on logging
        logging.basicConfig(
            level=logging.DEBUG,
            format='%(created).6f,%(levelname)s,S2_Greyscale,%(message)s',
            filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S2,
            filemode='w'
        )

        # Log the start
        logging.info("Process started with PID=%d", self.pid)

        # Initialise frame counter to 0
        frame_count = 0

        # Initialise window to display debug preview
        if BS_DEBUG_WINDOWS:
            greyscale_window = Window(GREYSCALE_DEBUG_WINDOW_NAME)

        while True:
            # Log input backlog
            # logging.debug("Input backlog of %d", self.input_q.qsize())

            # Log frame retrieval
            logging.debug("Retrieving frame %d", frame_count)
            # Retrieve frame
            frame, metrics = self.input_q.get(block=True)
            # Log the retrieval
            logging.debug("Retrieved frame %d", frame_count)

            # Check if the frame is a quit signal (check whether it's a string first!)
            if type(frame) == str:
                if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
                    # If it is then send quit signal to next stage and break
                    logging.info("Quit signal received - I am now quitting")
                    self.output_q.put((frame, None))
                    break

            # Track capture duration
            timer = Timer()
            # Apply the single-channel conversion with greyscale filter
            greyscale = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            # Stop the timer
            duration = timer.stop()

            # Compile metrics
            metrics.append(duration)

            # Enqueue frame and metrics
            self.output_q.put_nowait((greyscale, metrics))

            # Log the frame enqueue
            logging.debug("Processed and enqueued frame %d", frame_count)

            # Update the preview window
```

```
252             if BS_DEBUG_WINDOWS:
253                 greyscale_window.update(greyscale)
254 
255             # Increment frame count
256             frame_count += 1
257 
258 class S3_HistogramEqualisation(Process):
259     """ Process that applies a histogram equalisation to an input. """
260 
261     def __init__(self, input_q, output_q):
262         """
263         input_q: Queue that stores frames that require hist. equ. and prev. frame metrics\n
264         output_q: Queue that stores hist. equalised frames and metrics.
265         """
266         super().__init__()
267         self.input_q = input_q
268         self.output_q = output_q
269 
270 
271     def run(self,):
272         # Turn on logging
273         logging.basicConfig(
274             level=logging.DEBUG,
275             format='%(created).6f,%(levelname)s,S3_HistogramEqualisation,%(message)s',
276             filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S3,
277             filemode='w'
278         )
279 
280         # Log the start
281         logging.info("Process started with PID=%d", self.pid)
282 
283         # Initialise frame counter to 0
284         frame_count = 0
285 
286         # Initialise window to display debug preview
287         if BS_DEBUG_WINDOWS:
288             histequ_window = Window(HISTEQU_DEBUG_WINDOW_NAME)
289 
290         while True:
291             # Log input backlog
292             # logging.debug("Input backlog of %d", self.input_q.qsize())
293 
294             # Log frame retrieval
295             logging.debug("Retrieving frame %d", frame_count)
296             # Retrieve frame
297             frame, metrics = self.input_q.get(block=True)
298             # Log the retrieval
299             logging.debug("Retrieved frame %d", frame_count)
300 
301             # Check if the frame is a quit signal (check whether it's a string first!)
302             if type(frame) == str:
303                 if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
304                     # If it is then send quit signal to next stage and break
305                     logging.info("Quit signal received - I am now quitting")
306                     self.output_q.put((frame, None))
307                     break
308 
309             # Track capture duration
310             timer = Timer()
311             # Apply the histogram equalisation
312             histequ = cv2.equalizeHist(frame)
313             # Stop the timer
```

```
314             duration = timer.stop()
315
316             # Compile metrics
317             metrics.append(duration)
318
319             # Enqueue frame and metrics
320             self.output_q.put_nowait((histequ, metrics))
321
322             # Log the frame enqueue
323             logging.debug("Processed and enqueued frame %d", frame_count)
324
325             # Update the preview window
326             if BS_DEBUG_WINDOWS:
327                 histequ_window.update(histequ)
328
329             # Increment frame count
330             frame_count += 1
331
332 class S4_GaussianBlur(Process):
333     """ Process that applies a Gaussian blur to an input. """
334
335     def __init__(self, input_q, output_q):
336         """
337         input_q: Queue that stores frames that require Gaus. blurring and prev. frame metrics\
    n
338         output_q: Queue that stores Gaussian blurred frames and metrics.
339         """
340         super().__init__()
341         self.input_q = input_q
342         self.output_q = output_q
343
344
345     def run(self,):
346         # Turn on logging
347         logging.basicConfig(
348             level=logging.DEBUG,
349             format='%(created).6f,%(levelname)s,S4_GaussianBlur,%(message)s',
350             filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S4,
351             filemode='w'
352         )
353
354         # Log the start
355         logging.info("Process started with PID=%d", self.pid)
356
357         # Initialise frame counter to 0
358         frame_count = 0
359
360         # Initialise window to display debug preview
361         if BS_DEBUG_WINDOWS:
362             gausblur_window = Window(GAUSBLUR_DEBUG_WINDOW_NAME)
363
364         while True:
365             # Log input backlog
366             # logging.debug("Input backlog of %d", self.input_q.qsize())
367
368             # Log frame retrieval
369             logging.debug("Retrieving frame %d", frame_count)
370             # Retrieve frame
371             frame, metrics = self.input_q.get(block=True)
372             # Log the retrieval
373             logging.debug("Retrieved frame %d", frame_count)
374
```

```
375             # Check if the frame is a quit signal (check whether it's a string first!)
376             if type(frame) == str:
377                 if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
378                     # If it is then send quit signal to next stage and break
379                     logging.info("Quit signal received - I am now quitting")
380                     self.output_q.put((frame, None))
381                     break
382 
383             # Track capture duration
384             timer = Timer()
385             # Apply the gaussian blur
386             gausblur = cv2.GaussianBlur(frame, (5,5), 0)
387             # Stop the timer
388             duration = timer.stop()
389 
390             # Compile metrics
391             metrics.append(duration)
392 
393             # Enqueue frame and metrics
394             self.output_q.put_nowait((gausblur, metrics))
395 
396             # Log the frame enqueue
397             logging.debug("Processed and enqueued frame %d", frame_count)
398 
399             # Update the preview window
400             if BS_DEBUG_WINDOWS:
401                 gausblur_window.update(gausblur)
402 
403             # Increment frame count
404             frame_count += 1
405 
406 class S5_Canny(Process):
407     """ Process that applies the Canny algorithm to an input. """
408 
409     def __init__(self, input_q, output_q):
410         """
411         input_q: Queue that stores frames to apply Canny with and prev. frame metrics\n
412         output_q: Queue that stores the frame, Canny output edges, and metrics.
413         """
414         super().__init__()
415         self.input_q = input_q
416         self.output_q = output_q
417 
418 
419     def run(self,):
420         # Turn on logging
421         logging.basicConfig(
422             level=logging.DEBUG,
423             format='%(created).6f,%(levelname)s,S5_Canny,%(message)s',
424             filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S5,
425             filemode='w'
426         )
427 
428         # Log the start
429         logging.info("Process started with PID=%d", self.pid)
430 
431         # Initialise frame counter to 0
432         frame_count = 0
433 
434         # Initialise window to display debug preview
435         if BS_DEBUG_WINDOWS:
436             canny_window = Window(CANNY_DEBUG_WINDOW_NAME)
```

```
437
438         while True:
439             # Log input backlog
440             # logging.debug("Input backlog of %d", self.input_q.qsize())
441
442             # Log frame retrieval
443             logging.debug("Retrieving frame %d", frame_count)
444             # Retrieve frame
445             frame, metrics = self.input_q.get(block=True)
446             # Log the retrieval
447             logging.debug("Retrieved frame %d", frame_count)
448
449             # Check if the frame is a quit signal (check whether it's a string first!)
450             if type(frame) == str:
451                 if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
452                     # If it is then send quit signal to next stage and break
453                     logging.info("Quit signal received - I am now quitting")
454                     self.output_q.put((frame, None, None))
455                     break
456
457             # Track capture duration
458             timer = Timer()
459             # compute the median single-channel pixel intensities
460             gaus_median = np.median(frame)
461             # compute threshold values for canny using single parameter Canny
462             lower_threshold = int(max(0, (1.0 - CANNY_THRESHOLD_SIGMA) * gaus_median))
463             upper_threshold = int(min(255, (1.0 + CANNY_THRESHOLD_SIGMA) * gaus_median))
464             # perform Canny edge detection
465             edges = cv2.Canny(
466                 frame,
467                 lower_threshold,
468                 upper_threshold
469             )
470             # Stop the timer
471             duration = timer.stop()
472
473             # Compile metrics
474             metrics.append(duration)
475
476             # Enqueue frame and metrics
477             self.output_q.put_nowait((frame, edges, metrics))
478
479             # Log the frame enqueue
480             logging.debug("Processed and enqueued frame %d", frame_count)
481
482             # Update the preview window
483             if BS_DEBUG_WINDOWS:
484                 canny_window.update(edges)
485
486             # Increment frame count
487             frame_count += 1
488
489 class S6_Segmentation(Process):
490     """ Process that segments edges using minimum enclosing circles (MECs). """
491
492     def __init__(self, input_q, output_q):
493         """
494         input_q: Queue that stores Canny detected edges and prev. frame metrics\n
495         output_q: Queue that stores Canny output edges and metrics.
496         """
497         super().__init__()
498         self.input_q = input_q
```

```
499         self.output_q = output_q
500 
501 
502     def run(self,):
503         # Turn on logging
504         logging.basicConfig(
505             level=logging.DEBUG,
506             format='%(created).6f,%(levelname)s,S6_Segmentation,%(message)s',
507             filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S6,
508             filemode='w'
509         )
510 
511         # Log the start
512         logging.info("Process started with PID=%d", self.pid)
513 
514         # Initialise frame counter to 0
515         frame_count = 0
516 
517         # Initialise window to display debug preview
518         if BS_DEBUG_WINDOWS:
519             contour_window = Window(CONTOUR_DEBUG_WINDOW_NAME)
520 
521         while True:
522             # Log input backlog
523             # logging.debug("Input backlog of %d", self.input_q.qsize())
524 
525             # Log frame retrieval
526             logging.debug("Retrieving frame %d", frame_count)
527             # Retrieve frame
528             frame, edges, metrics = self.input_q.get(block=True)
529             # Log the retrieval
530             logging.debug("Retrieved frame %d", frame_count)
531 
532             # Check if the frame is a quit signal (check whether it's a string first!)
533             if type(frame) == str:
534                 if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
535                     # If it is then send quit signal to next stage and break
536                     logging.info("Quit signal received - I am now quitting")
537                     self.output_q.put((frame, None, None))
538                     break
539 
540             # Track find contours process
541             timer = Timer()
542             # 01 - Find the contours
543             # RETR_EXTERNAL only retrieves the extreme outer contours
544             # CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and
545             #   diagonal segments and leaves only their end points
546             contours, _ = cv2.findContours(
547                 edges,
548                 cv2.RETR_EXTERNAL,       # RetrievalModes
549                 cv2.CHAIN_APPROX_SIMPLE  # ContourApproximationModes
550             )
551             # Stop the timer
552             duration = timer.stop()
553 
554             # Compile metrics
555             metrics.append(duration)
556 
557             # Track find contours process
558             timer = Timer()
559             # List to store the particle information (centre coords + radius)
560             particles = []
```

```
            # 02 - Find minimum enclosuing circles
            for contour in contours:
                # Find the minimum enclosing circle for each contour
                ((x, y), radius) = cv2.minEnclosingCircle(contour)
                # Find the centre
                centre = (int(x), int(y))
                # Find the radius
                radius = int(radius)
                # Store the information
                particles.append((centre, radius))
            # Stop the timer
            duration = timer.stop()

            # Compile metrics
            metrics.append(duration)

            # Enqueue frame and metrics
            self.output_q.put_nowait((frame, particles, metrics))

            # Log the frame enqueue
            logging.debug("Processed and enqueued frame %d", frame_count)

            # Update the preview window
            if BS_DEBUG_WINDOWS:
                # create a black mask
                mask = np.zeros_like(edges)
                # draw contours white white fill
                cv2.drawContours(mask, contours, -1, (255), cv2.FILLED)
                # display window
                self.contour_window.update(mask)

            # Increment frame count
            frame_count += 1

class S7_Project(Process):
    """ Project the backscatter-cancelling light patterns. """

    def __init__(self, input_q, output_q):
        """
        input_q: Queue that stores computed backscatter particles and prev. frame metrics\n
        output_q: Queue that stores metrics.
        """
        super().__init__()
        self.input_q = input_q
        self.output_q = output_q


    def run(self,):
        # Turn on logging
        logging.basicConfig(
            level=logging.DEBUG,
            format='%(created).6f,%(levelname)s,S7_Project,%(message)s',
            filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S7,
            filemode='w'
        )

        # Log the start
        logging.info("Process started with PID=%d", self.pid)

        # Initialise frame counter to 0
        frame_count = 0

```

```
623         # Initialise window to display debug preview
624         projector_window = Window(PROJECTOR_PREVIEW_WINDOW_NAME)
625 
626         while True:
627             # Log input backlog
628             # logging.debug("Input backlog of %d", self.input_q.qsize())
629 
630             # Log frame retrieval
631             logging.debug("Retrieving frame %d", frame_count)
632             # Retrieve frame
633             frame, particles, metrics = self.input_q.get(block=True)
634             # Log the retrieval
635             logging.debug("Retrieved frame %d", frame_count)
636 
637             # Check if the frame is a quit signal (check whether it's a string first!)
638             if type(frame) == str:
639                 if frame == PROCESS_QUEUE_QUIT_SIGNAL or PROCESS_QUEUE_FQUIT_SIGNAL:
640                     # If it is then send quit signal to next stage and break
641                     logging.info("Quit signal received - I am now quitting")
642                     self.output_q.put((frame))
643                     break
644 
645             # Track capture duration
646             # timer = Timer()
647             # Create a white mask for the projector preview
648             projector_mask = np.ones_like(frame) * 255
649             # Process the particles:
650             for particle in particles:
651                 cv2.circle(
652                     projector_mask,
653                     particle[0],
654                     particle[1],
655                     (0, 0, 0),
656                     -1
657                 )
658             # Display the white mask with black circles
659             projector_window.update(projector_mask)
660             # Stop the timer
661             # duration = timer.stop()
662 
663             # Compile metrics
664             metrics.append(len(particles))
665 
666             # Enqueue frame and metrics
667             self.output_q.put_nowait(metrics)
668 
669             # Log the frame enqueue
670             logging.debug("Projected frame %d", frame_count)
671 
672             # Increment frame count
673             frame_count += 1
674 
675 class S8_Logging(Process):
676     """ Project the backscatter-cancelling light patterns. """
677 
678     def __init__(self, input_q):
679         """
680         input_q: Queue that stores each frame's metrics.\n
681         """
682         super().__init__()
683         self.input_q = input_q
684 
```

```
685
686     def run(self,):
687         # Turn on logging
688         logging.basicConfig(
689             level=logging.DEBUG,
690             format='%(created).6f,%(levelname)s,S8_Logging,%(message)s',
691             filename=LOGGING_FILEPATH + LOGGING_FILEPATH_S8,
692             filemode='w'
693         )
694
695         # Log the start
696         logging.info("Process started with PID=%d", self.pid)
697
698         # Initialise frame counter to 0
699         frame_count = 0
700
701         # Generate CSV export filename
702         export_filename_csv = "export_" + datetime.now().strftime("%m-%d-%Y-%H-%M-%S") + ".csv
    "
703
704         # Initialise a Pandas DataFrame to log real-time metrics
705         rt_metrics_df = pd.DataFrame(
706             columns=[
707                 'Capture Duration (s)',
708                 'Greyscale Conversion Duration (s)',
709                 'Histogram Equalisation Duration (s)',
710                 'Gaussian Blur Duration (s)',
711                 'Canny Algorithm Duration (s)',
712                 'CV2 findContours() Duration (s)',
713                 'CV2 minEnclosingCircle() Duration (s)',
714                 'Number of MECs on screen',
715             ]
716         )
717
718         while True:
719             # Log input backlog
720             # logging.debug("Input backlog of %d", self.input_q.qsize())
721
722             # Log frame retrieval
723             logging.debug("Retrieving metrics for frame %d", frame_count)
724             # Retrieve frame
725             metrics = self.input_q.get(block=True)
726             # Log the retrieval
727             logging.debug("Retrieved metrics for frame %d", frame_count)
728
729             # Check if the input is a quit signal (check whether it's a string first!)
730             if type(metrics) == str:
731                 if metrics == PROCESS_QUEUE_QUIT_SIGNAL:
732                     # If it is then export to CSV and break
733                     logging.info("Quit signal received - starting to export")
734                     # Export dataframe as CSV
735                     rt_metrics_df.to_csv(
736                         path_or_buf=export_filename_csv,
737                         encoding='utf-8'
738                     )
739                     logging.info("Successfully exported - now exiting")
740                     break
741                 elif metrics == PROCESS_QUEUE_FQUIT_SIGNAL:
742                     logging.info("Force quit signal received - I am now quitting")
743                     break
744
745             # Log the particles retrieval
```

```
            logging.debug("Logging data for frame %d", frame_count)
            # Add this frame's metrics to the end of the dataframe
            rt_metrics_df.loc[len(rt_metrics_df)] = metrics
            # Log the particle retrieval and processing
            logging.debug("Logged data for frame %d", frame_count)
            # Increment frame count
            frame_count += 1

if __name__ == "__main__":
    # Required to run X11 forwarding as sudo
    os.environ['XAUTHORITY'] = X11_XAUTHORITY_PATH

    # Set the OS priority level
    p = psutil.Process(os.getpid())
    print("Current OS priority: ", p.nice())
    p.nice(OS_NICE_PRIORITY_LEVEL)
    print("New OS priority: ", p.nice())

    q1_2 = Queue()  # Queue between stages 1 and 2
    q2_3 = Queue()  # Queue between stages 2 and 3
    q3_4 = Queue()  # Queue between stages 3 and 4
    q4_5 = Queue()  # Queue between stages 4 and 5
    q5_6 = Queue()  # Queue between stages 5 and 6
    q6_7 = Queue()  # Queue between stages 6 and 7
    q7_8 = Queue()  # Queue between stages 7 and 8

    stages = [
        S1_Capture(output_q=q1_2),
        S2_Greyscale(input_q=q1_2, output_q=q2_3),
        S3_HistogramEqualisation(input_q=q2_3, output_q=q3_4),
        S4_GaussianBlur(input_q=q3_4, output_q=q4_5),
        S5_Canny(input_q=q4_5, output_q=q5_6),
        S6_Segmentation(input_q=q5_6, output_q=q6_7),
        S7_Project(input_q=q6_7, output_q=q7_8),
        S8_Logging(input_q=q7_8)
    ]

    # Start the stages
    for stage in stages:
        stage.start()

    # Wait for stages to finish
    for stage in stages:
        stage.join()

    cv2.destroyAllWindows()
```

Listing 11: The Python code for the 'app2.py' entry point to the Multiprocessing Backscatter Cancellation System program, commit version: `326e3b5` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/multiprocessing/app2.py

D.1.2 CaptureManager: 'CaptureManager.py'

```
import os
import cv2
import logging
from queue import Queue
from threading import Thread
from natsort import natsorted
```

```
7
8
9  class PicameraStream:
10     """ Streams from a Picamera frame-by-frame. """
11
12     def __init__(self, width, height):
13         # Try to import and use Picamera2 library - this will fail on non-RPi platforms
14         try:
15             # Pylint - Ignore the missing import error on non RPi platforms
16             from picamera2 import Picamera2     # type: ignore
17
18             # Initialise Pi camera instance
19             self.picam2 = Picamera2()
20
21             # Pi camera configuration:
22             self.picam2.still_configuration.size = (width, height)      # Capture stills at
       defined resolution/size
23             self.picam2.still_configuration.format = 'BGR888'           # Use this format for
       OpenCV compatibility
24             self.picam2.still_configuration.align()                     # Align stream size (
       THIS CHANGES THE RES SLIGHTLY!)
25
26             # Apply the configuration to the Pi camera
27             self.picam2.configure("video")
28
29             # Print camera configurations to confirm correct set up
30             logging.debug(self.picam2.camera_configuration()['sensor'])
31             logging.debug(self.picam2.camera_configuration()['main'])
32             logging.debug(self.picam2.camera_configuration()['controls'])
33
34             # Start the Pi camera
35             self.picam2.start()
36
37             # Return the aligned stream resolution
38             (aligned_width, aligned_height) = self.picam2.camera_configuration()['main']['size
       ']
39             return (aligned_width, aligned_height)
40
41         # If the Picamera2 module is not found, then the program is probably not running on a
       RPi
42         except ImportError:
43             logging.error("Picamera2 module not found. Make sure you are running this on a
       Raspberry Pi.")
44         # No other exceptions should be encountered, if they are then log it
45         except Exception as e:
46             logging.error("Unforseen error encountered: ", e)
47
48     def read(self):
49         """ Capture camera sensor array for constructing a frame-by-frame feed. """
50
51         success, frame = True, self.picam2.capture_array("main")
52         return success, frame
53
54     def exit(self):
55         """ Gracefully stop the Pi camera instance. """
56
57         # Stop the Pi camera instance.
58         self.picam2.stop()
59
60         # Log to console
61         logging.info("Camera instance has been gracefully stopped.")
62
```

```
63 class FrameStream:
64     """ Streams a video represented by a set of PNG images for each frame. """
65 
66     def __init__(self, path):
67         # Initialise path to folder
68         self.path = path
69 
70         # Initialise queue to store frames in memory
71         self.q = Queue()
72 
73         # Populate queue with image frames
74         self._load_frames()
75 
76     def _load_frames(self):
77         """ Internal method that populates the image frame queue. """
78 
79         logging.debug("Beginning to populate frame queue...")
80 
81         # Sort all of the frame files inside the folder
82         frames = natsorted(os.listdir(self.path))
83 
84         # For each file in the sorted list of frame files...
85         for file in frames:
86             # Ensure that the file is a PNG
87             if file.endswith('.png'):
88                 # Extract the frame number from the filename
89                 frame_num = int(os.path.splitext(file)[0])
90                 # Generate the filepath
91                 frame_path = os.path.join(self.path, file)
92                 # Read in the file
93                 frame = cv2.imread(frame_path)
94                 # Store the file in the queue along with the frame number
95                 self.q.put(frame)
96 
97         # Log status
98         logging.debug("Frame queue has been populated.")
99 
100     def read(self):
101         """ Dequeues the next frame in the sequence. """
102 
103         if not self.empty():
104             return self.q.get()
105         else:
106             return None
107 
108     def empty(self):
109         """ Returns true if there are no more frames to stream. """
110 
111         return self.q.empty()
112 
113 class VideoStream:
114     """ Streams an input video file. """
115 
116     def __init__(self, source, crop_w=None, crop_h=None, loop=False):
117         # initialise the OpenCV stream
118         self.capture = cv2.VideoCapture(source)
119         self.crop_w = crop_w
120         self.crop_h = crop_h
121         self.loop = loop
122 
123         # check if capture is accessible
124         if not self.capture.isOpened():
```

```
125             logging.error("Cannot open video stream!")
126             raise Exception("Cannot open video stream!")
127 
128         # calculate FPS and FPT of the capture
129         self.target_fps = self.capture.get(cv2.CAP_PROP_FPS)
130         self.target_fpt = (1 / self.target_fps) * 1000
131 
132     def read(self):
133         _, frame = self.capture.read()
134         if frame is not None:
135             if (self.crop_w is not None) and (self.crop_h is not None):
136                 frame = frame[self.crop_h[0]:self.crop_h[1], self.crop_w[0]:self.crop_w[1]]
137         elif self.loop is True:
138             self.capture.set(cv2.CAP_PROP_POS_FRAMES, 0)
139             _, frame = self.capture.read()
140         return frame
141 
142     def exit(self):
143         self.capture.release()
144 
145 class t_VideoStream:
146     """ Streams an input video file using threading. """
147 
148     def __init__(self, source, queueSize=4096):
149         # initialise the OpenCV stream
150         self.capture = cv2.VideoCapture(source)
151         # initialise parameter that stops video stream
152         self.stopped = False
153         # initialise the queue for pushing frames
154         self.queue = Queue(maxsize=queueSize)
155 
156     def start(self):
157         # start a thread to read frames from stream
158         t = Thread(
159             target=self._update,
160             args=()
161         )
162         # allow thread to be killed when main app exits
163         t.daemon = True
164         # start the thread
165         t.start()
166         return self
167 
168     def _update(self):
169         while True:
170             # stop reading if stream is stopped
171             if self.stopped:
172                 return
173 
174             # otherwise, keep reading and queuing until queue is full
175             if not self.queue.full():
176                 # read the next frame from the file
177                 success, frame = self.capture.read()
178 
179                 # check if we have reached the end of video stream
180                 if not success:
181                     self.stop()
182                     return
183 
184                 # push the frame to the queue
185                 self.queue.put(frame)
186 
```

```
187     def read(self):
188         # return a frame from the queue
189         return self.queue.get()
190 
191     def stop(self):
192         # indicate that thread should be stopped
193         self.stopped = True
194         self.capture.release()
195 
196     def empty(self):
197         # returns True if queue is empty
198         if self.queue.qsize():
199             return True
200         return False
```

Listing 12: The Python code for 'CaptureManager.py' in the Multiprocessing Backscatter Cancellation System program, commit version: `1e69d8b` from https://github.com/Sidharth-Shanmugam-MEng-Project-2023-24/backscatter-cancellation-system/blob/multiprocessing/CaptureManager.py

D.1.3 TimeManager: 'TimeManager.py'

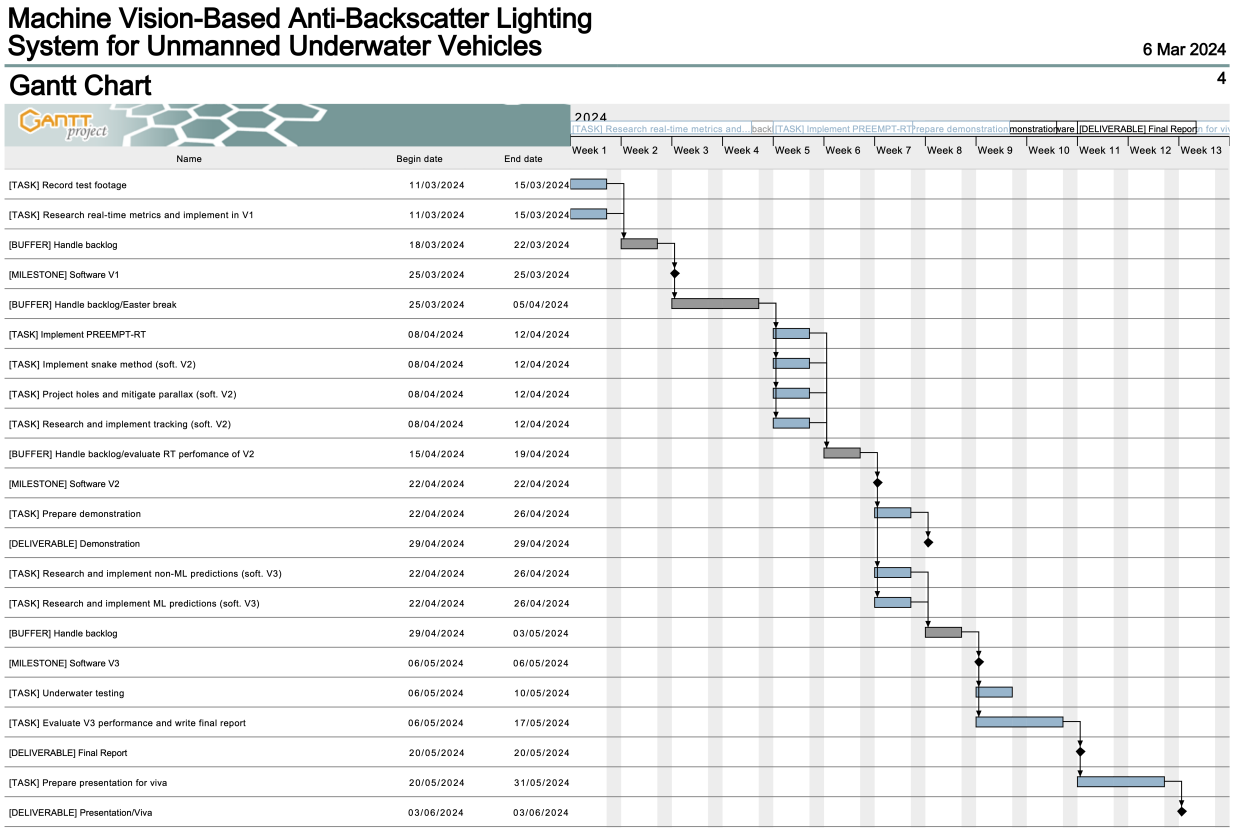
Use the same `TimeManager.py` from the Standard Backscatter Cancellation System.

D.1.4 WindowManager: 'WindowManager.py'

Use the same `WindowManager.py` from the Standard Backscatter Cancellation System.

Appendix E Project Progress Gantt Charts

E.1 08-03-2024



Name	Begin date	End date
[TASK] Record test footage	11/03/2024	15/03/2024
[TASK] Research real-time metrics and implement in V1	11/03/2024	15/03/2024
[BUFFER] Handle backlog	18/03/2024	22/03/2024
[MILESTONE] Software V1	25/03/2024	25/03/2024
[BUFFER] Handle backlog/Easter break	25/03/2024	05/04/2024
[TASK] Implement PREEMPT-RT	08/04/2024	12/04/2024
[TASK] Implement snake method (soft. V2)	08/04/2024	12/04/2024
[TASK] Project holes and mitigate parallax (soft. V2)	08/04/2024	12/04/2024
[TASK] Research and implement tracking (soft. V2)	08/04/2024	12/04/2024
[BUFFER] Handle backlog/evaluate RT perfomance of V2	15/04/2024	19/04/2024
[MILESTONE] Software V2	22/04/2024	22/04/2024
[TASK] Prepare demonstration	22/04/2024	26/04/2024
[DELIVERABLE] Demonstration	29/04/2024	29/04/2024
[TASK] Research and implement non-ML predictions (soft. V3)	22/04/2024	26/04/2024
[TASK] Research and implement ML predictions (soft. V3)	22/04/2024	26/04/2024
[BUFFER] Handle backlog	29/04/2024	03/05/2024
[MILESTONE] Software V3	06/05/2024	06/05/2024
[TASK] Underwater testing	06/05/2024	10/05/2024
[TASK] Evaluate V3 performance and write final report	06/05/2024	17/05/2024
[DELIVERABLE] Final Report	20/05/2024	20/05/2024
[TASK] Prepare presentation for viva	20/05/2024	31/05/2024
[DELIVERABLE] Presentation/Viva	03/06/2024	03/06/2024

Figure 24: Gantt chart schedule as of the 8th of March, 2024.

E.2 15-03-2024

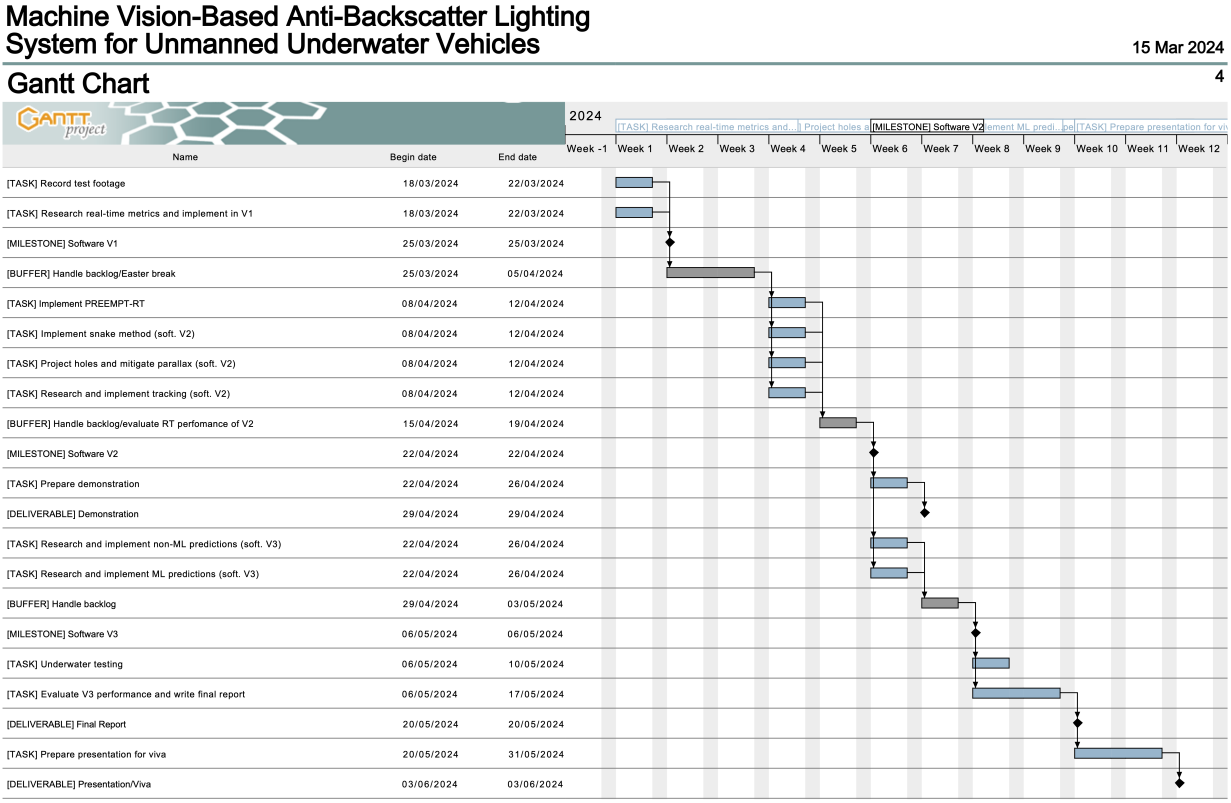


Figure 25: Gantt chart schedule as of the 15$^{\text{th}}$ of March, 2024.

E.3 12-04-2024

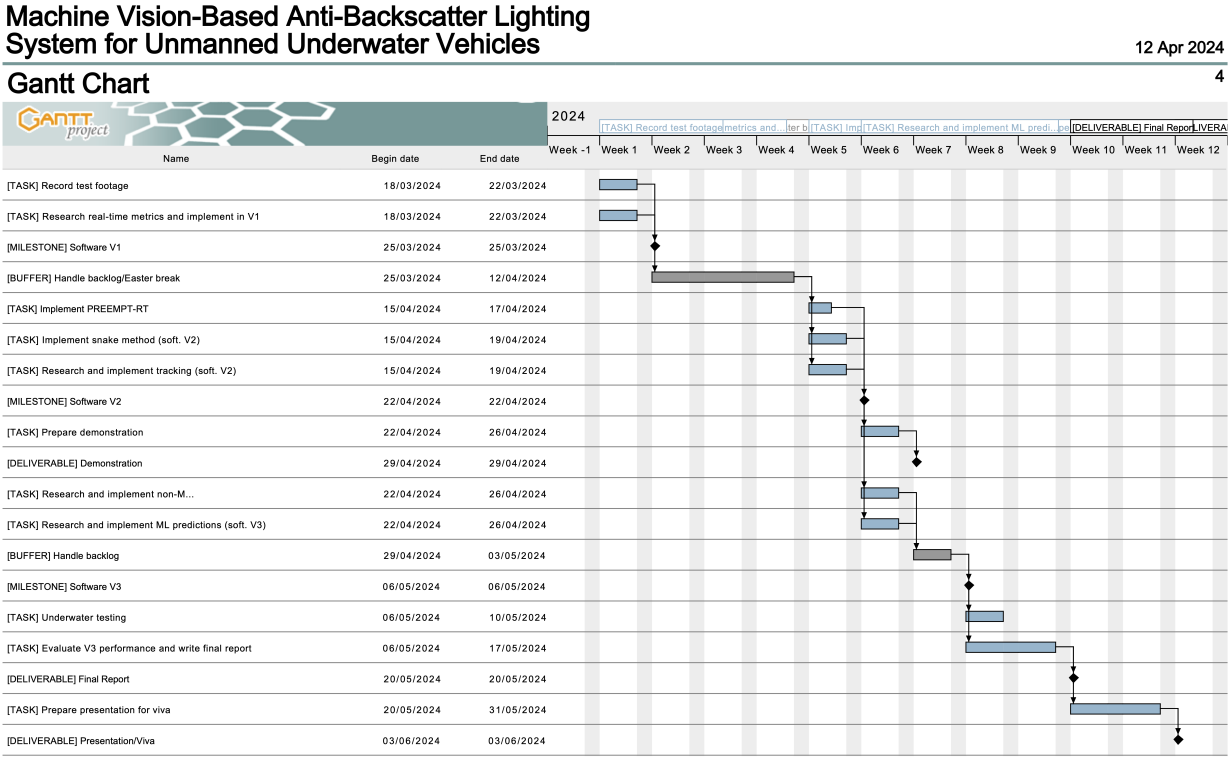


Figure 26: Gantt chart schedule as of the 12$^{\text{th}}$ of April, 2024.

E.4 03-05-2024

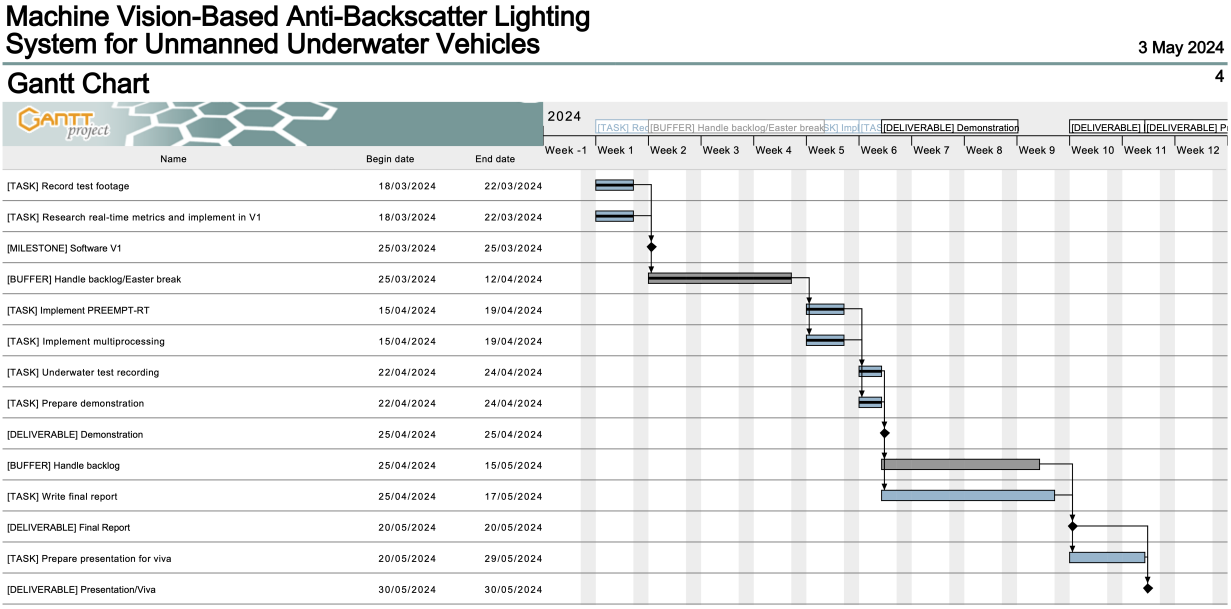


Figure 27: Gantt chart schedule as of the 3rd of May, 2024.