A

Design Methodology

This section introduces the design methodology and theoretical tools that structured the development of the prototype. Tools include the Basic Design Cycle, function analysis, morphological charting, and SCAMPER, as described in the Delft Design Guide [38].

A.1. Design Perspective

The approach for the development process has been shaped by the methods outlined in the Delft Design Guide [38], with particular emphasis on the principle of Design for the Majority. This perspective advocates for developing products that are accessible, relevant, and sustainable for people at the Base of the Pyramid (BoP), the majority of the world's population who often live in economically disadvantaged conditions.

Most existing water monitoring technologies are designed with users at the Top of the Pyramid (ToP) in mind, what relates to wealthier regions with robust infrastructure, technical support, and high financial capacity. These technologies are often too costly, maintenance-intensive, or complex for deployment in low-resource settings. As a result, they fail to meet the needs of local water management professionals in developing areas.

This research specifically adopts a BoP design mindset to ensure that the proposed water level monitoring system is affordable, accessible, available, reliable, sustainable, and culturally acceptable. Key principles influencing the design include:

- **Affordability**: The product must be cost-effective, with minimal reliance on patented technology.
- **Accessibility**: The system should be deployable in remote or under-resourced environments.
- **Availability**: Materials and skills required to build or maintain the system should be locally accessible.
- **Reliability**: The system must operate autonomously with minimal maintenance, supporting community-led usage and repair.
- **Sustainability**: Designs should have low environmental impact, for example through self-sufficient power sources.
- **Acceptability**: Local values, working practices, and preferences must be acknowledged and integrated.

The design perspective serves as a tool to focus the development process on end-user needs. In this research, the primary end users are water management officers operating in low-resource regions, where reliable, low-cost monitoring systems are lacking. However, since the outcome of this project is a proof-of-concept through a minimal viable product (MVP), it will serve as a starting point for future iterations and refinements. Therefore, the project supervisors are also considered interim users during this development phase. Their technical and contextual expertise plays a pivotal role in aligning the

development process with real-world applicability and guiding design decisions when direct user input from the target region is not available.

In addition to the design perspective, a function analysis was performed to map the system's purpose within its context. This analysis structured the translation of user needs into concrete system requirements and supported the development of the requirements list outlined in Section A.4.

A.2. Function Analysis

To complement the design perspective and ensure that all essential functions of the system are logically connected to its components, a Function Analysis was conducted. This method supports the development of both new and existing products by identifying all system functions and linking them to physical or conceptual elements, referred to as ”organs” [38].

This approach encourages a problem-solving mindset by abstracting the system's objectives into functional blocks, helping to focus on ”what the system should do” rather than prematurely locking in design choices.

Figure A.1 illustrates the high-level functional decomposition for the real-time water level monitoring system developed in this project.

The function tree begins with the overarching purpose of the system and breaks it down into five key subsystems:

- Sensor system, responsible for acquiring data from the environment
- Processing system, which performs image analysis and stores data
- Communication system, enabling remote access to measurements
- Power system, designed for off-grid, autonomous operation
- Casing / General design, addressing physical robustness, cost-effectiveness, and maintenance simplicity

Each of these subsystems is further decomposed into specific functional requirements, such as operating during day- and nighttime, data transmission via Wi-Fi and low-frequency maintenance. This analysis helped ensure that each design decision was tied to a clearly identified system function, and it provided a reference framework for validating the prototype during field testing.

The function analysis served not only to clarify design objectives but also supported the creation of the system requirements (Section A.4) and informed the morphological chart and component selection process described later in this chapter.

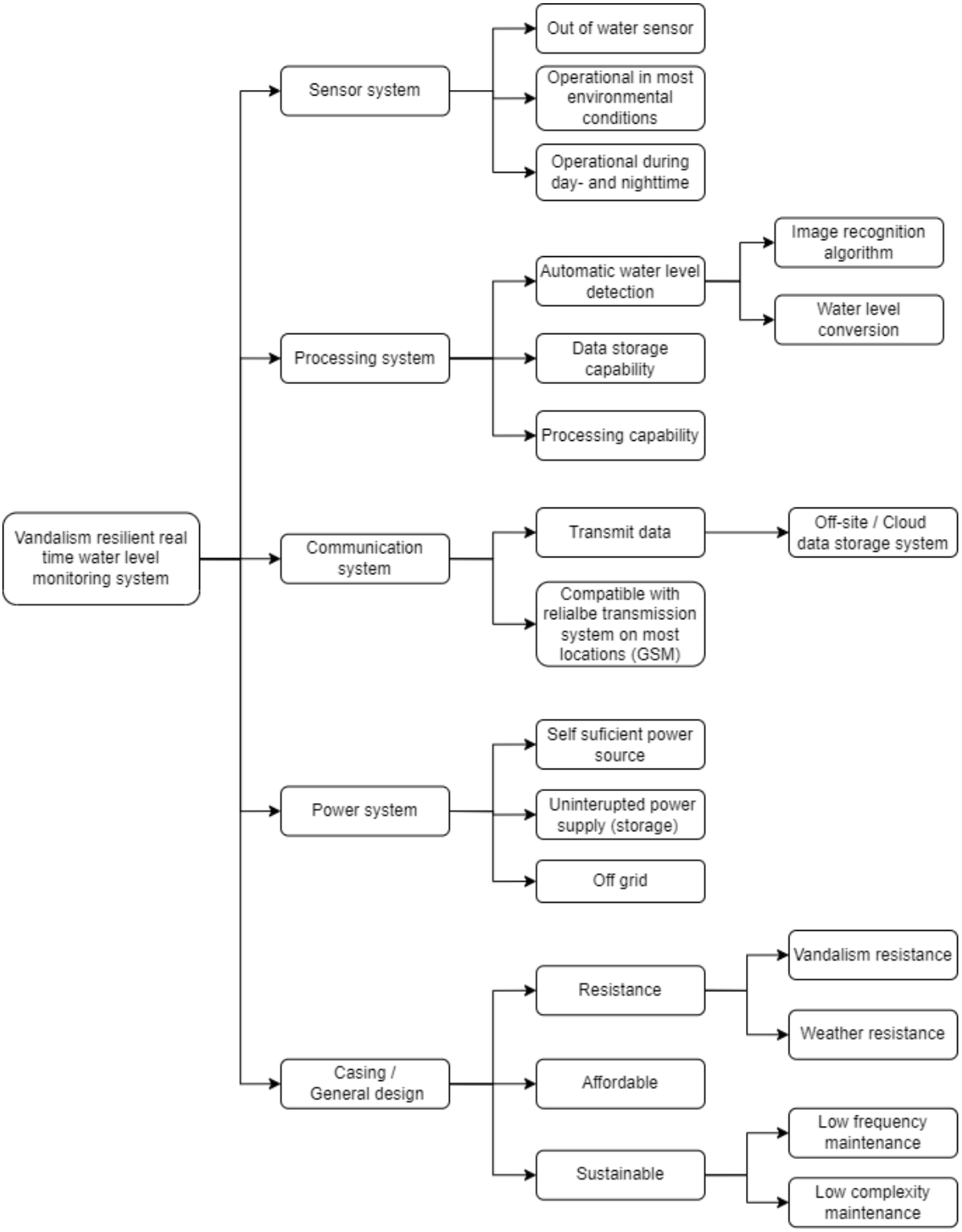


Figure A.1: Function analysis

A.3. Basic Design Cycle

The development process of this research project followed the principles of the Basic Design Cycle, as described in the Delft Design Guide (Boeijen and Daalhuizen, 2013). This model outlines the core reasoning steps in design and promotes an iterative approach, helping designers refine both their understanding of the problem and the development of viable solutions over time.

The cycle consists of the following five key phases:

- **Analysis** – The problem context is studied in depth to clarify user needs, constraints, and key design requirements. This stage provided the foundation for the Function Analysis (Section A.2) and the List of Requirements (Section A.4).
- **Synthesis** – Based on the insights from the analysis, potential solutions and configurations are generated. These are structured using methods such as the Morphological Chart (Section A.5).
- **Simulation** – Early concepts are simulated through low-complexity prototyping, desk testing and reasoning-based evaluations. These simulations help identify weak points and guide refinement before costly iterations.
- **Evaluation** – Prototypes are evaluated based on their performance against predefined criteria (e.g., water level accuracy, processing speed, power efficiency).
- **Decision** – The outcome of the evaluation stage determines whether the solution is accepted, adjusted, or returned to an earlier step in the cycle for further refinement.

These steps are not followed strictly in a linear order but are revisited as needed. Iteration is a core aspect of this model, supporting a deeper alignment between user needs and technical feasibility over time.

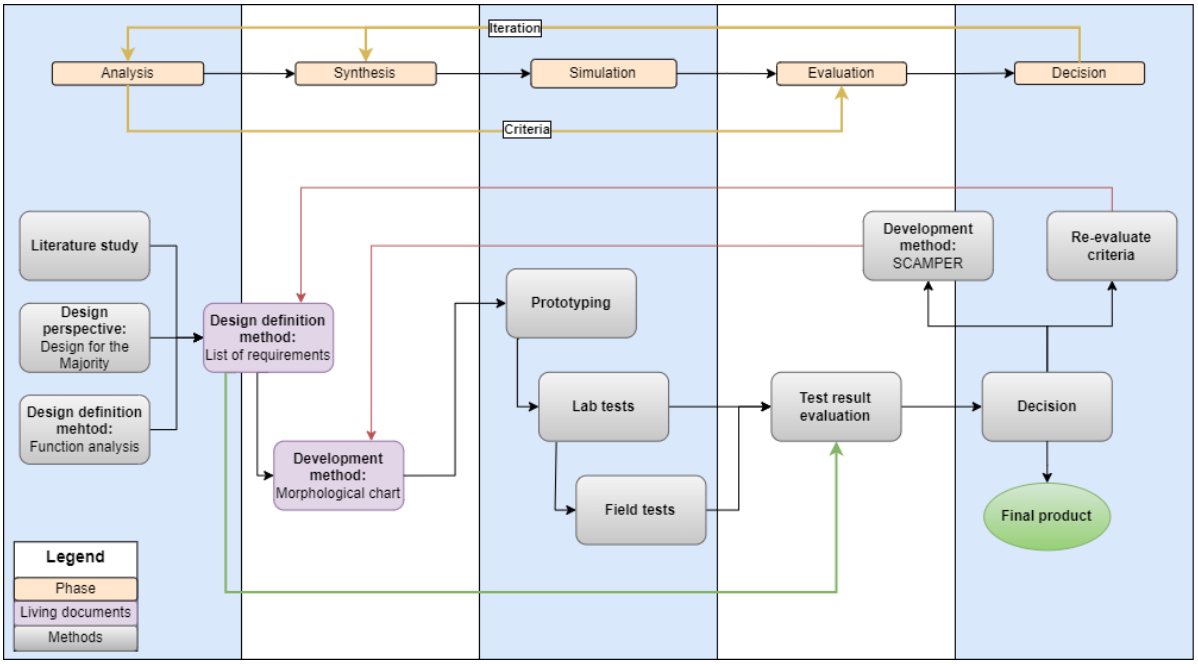


Figure A.2: Application of the Basic Design Cycle in this project.

As shown, the cycle is structured around core design phases (orange), supported by living documents such as the List of Requirements and the Morphological chart (purple), and driven by key design methods (grey). The iterative nature of the process is highlighted by arrows looping between testing, evaluation, and refinement, ensuring that the final prototype reflects both the system requirements, and the user-focused design perspective introduced earlier.

A.4. List of Requirements

To translate user needs and environmental constraints into concrete, testable conditions, a structured List of Requirements (LoR) was created. This method, outlined in the Delft Design Guide [38], plays a central role in anchoring the design process and preventing inconsistent decisions.

The LoR serves as a living document, which acts as both a boundary-setting tool during concept development and a reference point for prototype validation. In the context of this research, the list was directly informed by:

- The design perspective (Section A.1)
- The outcomes of the function analysis (Figure A.1)
- Feedback and scoping sessions with the project team

The list distinguishes between demands (must be met) and wishes (desirable but non-essential) using the MoSCoW prioritization framework:

- **Must-have**: critical for functionality
- **Should-have**: strongly desirable
- **Could-have**: optional improvements
- **Won't-have (for now)**: future scope

The full list of requirements is detailed in this section, and many are explicitly revisited in the performance evaluation and results analysis (Annex C).

The following design requirements were established in cooperation with the project team and guided the development of the prototype water-level monitoring system. These criteria reflect both the functional needs and operational constraints of deploying a robust, low-cost, off-grid system in flood-prone, remote regions such as Kenya. Where applicable, requirements will be validated through field testing, while others are scoped for future development or longer-term operation.

To structure and prioritize the system design requirements, the MoSCoW method was employed. This method categorizes requirements into four groups — Must-have, Should-have, Could-have, and Won't-have (for now) — based on their criticality to the successful functioning of the prototype:

- **Must-have (M)** requirements are essential for the system to function as a valid proof-of-concept. These requirements define the core capabilities without which the device would fail to meet its primary objectives.
- **Should-have (S)** requirements are important but not vital. While not essential for basic operation, they add significant value and improve performance or robustness.
- **Could-have (C)** requirements are desirable but non-essential features. These elements are not strictly required for the prototype, but their development could enhance the system's usability, resilience, or long-term sustainability.
- **Won't-have (W)** requirements are acknowledged but excluded from the current scope due to time, technical, or budgetary constraints. Many of these are identified as potentials for future work, to be explored in follow-up studies or as part of system refinement beyond this proof-of-concept phase.

This prioritization approach allows the project team to focus on delivering a functional and testable prototype within the available resources and timeframe, while also laying the groundwork for future development directions.

Source: Agile Business Consortium [1]

ID	Type	MoSCoW	Description
1	Flood protection	M	The device must remain fully operational and stay dry during floods with a return period of 1/100 years or smaller. The environmental conditions for these events should be verified through a review of site-specific flood risk assessment during the testing phase.
2	Measurement availability	M	Measured water level data must be processed and made available to users within 5 minutes of data collection.
3	Continuous measurement	M	The device must be capable of performing measurements and processing at a minimum frequency of once every 5 minutes under extreme operational conditions, ensuring accurate data collection during critical events.
4	Cost effectiveness	M	The total material cost of the device must not exceed €400, including all components, as confirmed through cost analysis.
5	All-condition measurement	M	The device must measure water levels accurately during both day and night, regardless of ambient light conditions, as verified in a range of lighting scenarios during the testing phase.
6	Water level detection accuracy	S	The device must detect water levels with an accuracy of ±2 cm, as validated during field experiments during the testing phase.
7	Secure installation	S	The device must be installed securely, ensuring the sensor does not shift from its designated frame position for a minimum of 6 months, even under external disturbances.
8	Weather resistance	S	The device must withstand all extreme weather conditions (including heat, wind, and rain) prevalent in Kenya, with a return period of <1/100 years, as demonstrated through environmental stress tests (environmental stress tests not within scope of this project?).
9	Vandalism resistance	S	The device must incorporate design elements that deter and resist vandalism, with specific vandal-proofing measures validated during testing, and final criteria determined during the testing phase.
10	Energy independence	C	The device must operate independently (using power storage or a combination of power generation and storage) for at least 6 months without requiring external intervention.
11	Data storage backup	C	The device must be able to store measured data for up to 48 hours during connection interruptions, ensuring no loss of critical measurements.
12	Data accuracy filtering	W	The system must filter or flag statistical outliers before transmitting data, with criteria for outliers to be defined and finalized during the testing phase.
13	Maintenance-free operation	W	The device must function without any maintenance for a continuous period of at least 6 months, as validated during operational testing (operational testing not within scope of this project?).

Table A.1: List of prioritized system requirements based on the MoSCoW method.

A.5. Morphological Chart

To explore a wide range of conceptual solutions, a Morphological Chart was used. This method, described in the Delft Design Guide (Boeijen and Daalhuizen, 2013), is especially useful in early-stage design when multiple functional requirements need to be addressed simultaneously.

The Morphological Chart breaks down the system's main function (to deliver a vandalism-resilient, real-time water level monitoring system)[1] into multiple subfunctions, such as flood protection, measurement accuracy, power supply, and vandalism resistance. For each subfunction, a list of alternative technical solutions was generated and visualized in a matrix format.

Steps followed in creating the chart:

1. Define the main function and subfunctions. These functions are derived from the Function Analysis (Section A.2).
2. List solution options. Ideas were generated from literature, team brainstorming, and BoP-oriented design principles.
3. Evaluate feasibility. Solutions were assessed based on relevance to the List of Requirements (Section A.4), resource constraints, and alignment with low-power, low-cost principles.
4. Select a preferred combination. A configuration of components was selected to form the initial prototype, with additional paths marked for future iteration.

Cells highlighted in green represent the options included in the current prototype. Those unhighlighted represent unimplemented ideas. This visualization not only illustrates the decision-making process but also demonstrates the modularity of the design. Multiple components can be substituted or improved independently without altering the full system architecture.

The chart also served as a living document during testing and was revisited during the SCAMPER-based evaluation phase (Section A.6).

MoSCoW	Criterion	Option 1	Option 2	Option 3	Option 4	Option 5
M	1. Flood Protection	Away from river	Above river (bridge/stilts)	Floating in river	Submersible with sealed casing	
M	2. Water Level Detection	High resolution camera	Zoom lens	Close to river/gauge		
M	3. Measurement Availability	High processing power	Low computational algorithm	Small datatype upload format		
M	4. Continuous Measurement	Event-triggered measurement	Duty-cycling	Dynamic threshold (historical)	Machine learning anomaly detection	
C	5. Data Accuracy Filtering	Integration of filtering workflow	Post-transmission correction			
S	6. Energy Independence	Power storage capacity	Power generation (solar/wind/hydro)	Storage + Generation	Low power demand	
W	7. Maintenance-Free Operation	High lifetime materials	Adaptive algorithm	Operational via online connection	Automatic lens cleaning	Debris shielding for sensor
S	8. Secure Installation	Secured to manmade structure	Secured on tree	Secured out of reach		
M	9. Weather Resistance	Waterproof	Integrated fan (lens condensation)	Impact-resistant casing (wind/storm)	Compact wind-resistant design	Heat-resistant electronics
W	10. Data Storage Backup	Large SD card (processing+storage)	Additional SD card	Local-to-cloud sync	On-device ring buffer storage	
S	11. Cost Effectiveness	Optimized modular design				
S	12. All-condition Measurement	NoIR lens + IR Light	IR camera	Flashlight	Thermal night vision	
M	13. Vandalism Resistance	Camouflage	Hidden design	Discouraging (beehive/police)	Installed in monitored areas	Compact

Included in current prototype | **Not included** in current prototype

Figure A.3: Morphological chart

[1] Decide on including vandalism resistance in project or not.

A.6. SCAMPER Method

To guide idea generation and concept refinement throughout the design process, the SCAMPER method was employed. This creative thinking technique is based on seven structured heuristics, each prompting a specific question to stimulate innovation or identify opportunities for improvement. It was particularly useful during the evaluation and testing phases, where results revealed areas for improvement or simplification.

The SCAMPER framework includes the following heuristics [38]:

- **Substitute** – What materials, components, or technologies could be replaced with simpler, cheaper, or more sustainable alternatives?
- **Combine** – Are there functions or parts that could be integrated to reduce complexity or improve efficiency?
- **Adapt** – How could elements from other systems or contexts be adapted to suit this design?
- **Modify** – What modifications (e.g., in shape, scale, performance) could improve usability or functionality?
- **Put to Another Use** – Could existing components or algorithms serve secondary purposes (e.g., environmental monitoring)?
- **Eliminate** – Which features or parts can be simplified, reduced, or removed to lower cost or maintenance?
- **Reverse** – Could processes be re-ordered or reversed to improve reliability or reduce resource use?

The SCAMPER method was applied iteratively after each testing round to reflect critically on the prototype's performance against the List of Requirements (Section A.4) and adjust system configurations where appropriate. For example, under the Substitute and Reverse heuristics, the original 5V power system was replaced with a 12V configuration to enable the use of a higher-powered 12V LED for improved nighttime visibility. This was evaluated alongside a 5V LED alternative. However, after testing, the system was reverted to the original 5V setup. This decision was based on the need to better align with the system's low-power design goals and overall energy efficiency requirements, as outlined in the criteria for off-grid operation.

Together, the design methodologies discussed in this section formed the foundation for a structured, iterative, and user-centered development process. The discussed methods ensured that both the conceptual framework and practical constraints were considered. The main report describes how these theoretical foundations were translated into a physical prototype through hardware development and testing.

The actual SCAMPER results can be found in the development log (Annex G).

B

Design choices

B.1. Operating system Raspberry Pi

The design of the water level detection system was influenced by insights gained from the literature review. Existing systems typically either relied on high-computational deep learning approaches, or simpler wildlife cameras that stored raw images without real-time processing. To fill this gap, this research deliberately focused on creating a lightweight, autonomous algorithm that could run efficiently on a low-power device such as the Raspberry Pi Zero 2 W.

Alternative hardware options, such as the Realtek AMB82 Mini, were initially explored. However, due to the researcher's greater familiarity with Python environments and the need to develop a working Prove Of Concept (POC) within a limited timeframe, the decision was made to build the system around the Raspberry Pi platform.

Throughout the development process, algorithmic choices were guided by the need to minimize memory usage, processing demand, and energy consumption. Deep learning models were avoided, as their computational intensity would have conflicted with the low-power and low-cost objectives.

B.2. Reference object

The reference object underwent several design iterations during development. The initial prototype consisted of a small foam panel affixed to a shovel, which proved too unstable for consistent imaging and served only to validate the basic camera functionality. In the second iteration, a hollow metal rail painted white and marked at 10 cm intervals was used; a mounting handle allowed attachment to riverbanks or concrete structures (as tested at TU Delft gutter). Although this rod performed well for early algorithm versions, its narrow profile yielded only a few pixels in width when imaged at greater distances, reducing detection reliability. To address this, the final reference object was constructed from a large matte PVC board, providing sufficient pixel area for each intensity box and thereby smoothing the computed difference signals and reducing noise. Other candidate materials evaluated during this process are listed below.

	Material	IR Reflection (850 nm)	Waterline Visibility	Tape / Paint Adhesion	Water Resistance	Weight / Stability	Cost	Remarks
	Stainless steel pipe	Good (diffuse)	Good (if matte/brushed)	Good	Excellent	High (stable)	High	Durable, industrial-grade
	Aluminium pipe	Very good (specular)	May be worse (glossy)	Moderate – primer recommended	Good	Low to moderate	Medium	Lightweight, but risk of IR glare
	White PVC pipe/sheet	Good (diffuse)	Good (clear contrast)	Very good	Good	Low	Low	Budget option, easy to work with
	Retroreflective tape (3M/Avery)	Very good (IR-specific)	Reflection under water unclear	Self-adhesive, primer may help	Good above water	n/a	Medium	Use only above water for IR detection
	Aluminium foil / metal tape	Very strong (glare)	Poor (glossy)	Poor	Moderate	n/a	Low	Not recommended, except for temporary use
	Matte black tape	Poor (low reflection)	Excellent contrast with tape	Good	Depends on type	n/a	Low	Ideal for markings
	Concrete + white paint	Good (depends on paint)	Good (sharp transition via absorption)	Good with primer on rough concrete	Very good (immersible)	n/a	Low–medium	Ideal for fixed setups; use matte white or epoxy paint

Figure B.1: Design considerations reference object.

B.3. Algorithm development

Over the course of five prototype iterations, the water-level detection algorithm evolved from a simple global-threshold approach to a robust, multi-mode statistical pipeline, with each stage informed by systematic feedback loops between hardware and software.

In the first prototype, Otsu's method was employed to segment an improvised foam staff gauge from its background. Although this yielded a clear binarization in some frames, it proved unstable under changing lighting and reflections; large bright regions such as sky and wet concrete were often misclassified, and underwater portions of the gauge confused the thresholding.

In the second prototype, the refined Otsu workflow added component-size filtering and tight cropping around the gauge. Yet noise persisted, runtime remained high, and manual cropping imposed a significant setup burden. These results motivated moving away from whole-object segmentation toward methods that directly target the waterline's characteristic intensity jump.

In Prototype 3, a "box-differencing" algorithm was implemented: two fixed-height boxes slide down the vertically aligned gauge, computing the difference in mean pixel intensity at each position. After smoothing the resulting profile and suppressing edge peaks, the highest local maximum reliably indicated the waterline. Although this approach markedly outperformed Otsu's method, exploration of a continuous image-subtraction (CIS) variant revealed that rapid-burst capture was beyond the Pi's capacity and that CIS suffered from noise due to vegetation and lighting shifts. Manual cropping remained an obstacle, however, underscoring the need for automated region-of-interest definitions.

Prototype 4 broadened the pipeline to four detection modes (raw intensity, HSV value, hue, and saturation) and introduced a Kolmogorov–Smirnov (KS) statistic as an alternative to mean-difference. For each mode, differences were smoothed, normalized into a probability distribution, and peak-scored, with the highest-scoring mode selected per image. While the KS method consumed more energy, it proved more resilient to glare.

By Prototype 5, the best detection routines were refactored into modular Python packages (wd__capture, wd__image_processing, etc.) and deployed as a systemd service to run headless from boot. Field testing over 270 cycles confirmed that both the mean-difference and KS-test methods met "few-centimeter" accuracy targets on a resource-constrained Pi Zero 2 W, with the service reliably cycling and logging results.

B.4. Hardware development

Over the course of five iterative prototypes, the hardware platform evolved from a bare-bones test setup into a fully integrated, field-ready enclosure. At each stage, lessons learned in bench and field trials drove targeted improvements in reference-object stability, illumination, power management, and environmental resilience.

To optimize processing, the camera output was downsampled. Rather than capturing the full 3280 × 2464 pixel frame, the Raspberry Pi Camera was configured to output a 640 × 480 pixel image. Technically, this is achieved by dividing the full sensor area into nine equal sections and selecting only the central compartment. Importantly, this preserves the resolution of the field of view while substantially reducing image file size, memory usage, and processing requirements.

The initial goal was to verify basic camera-based staging and data logging on the Raspberry Pi Zero 2 W with minimal investment of time and resources. A generic 20.000 mAh USB power bank provided 5 V/2.4 A to both the Pi and a TP-Link M7200 portable LTE router, enabling remote image capture even where Wi-Fi was unavailable. The Pi and camera module V2 were housed in an off-the-shelf Pi Zero plastic case. This assembly was simply taped onto a fence next to the gutter at Tu Delft. Although sufficient to test the Otsu-threshold algorithm in Prototype V1, the open case offered no protection from rain.

Addressing the glare issues, Prototype V2 introduced a circular polarizer was attached in front of the lens aperture, and the Pi case remained unchanged. Power wiring remained the same. This iteration proved the value of a stable, high-contrast gauge, though reflections under variable illumination still induced noise.

To enable reliable night-time captures, Prototype V4 incorporated a MOSFET-driven IR LED emitter (ILH-IN01-85SL-SC211-WIR200) mounted on an LED-star heatsink with thermal interface fluid. A small PCB bearing the MOSFET driver and a 1.2 Ω/5 W current-limiting resistor was wired between the powerbank's second USB output and the LED; cable lugs and heat-shrink provided safe installation and modularity. A custom 850 nm band-pass filter and high-quality circular polarizer were added via a photography filter holder attached the lid. The Pi and MOSFET board were attached to a 60 × 90 mm prototyping PCB, replacing the off-the-shelf case. This arrangement permitted visual confirmation of IR operation, reduced day/night variability, and maintained modularity for rapid assembly.

In the final iteration, all components; Raspberry Pi Zero 2 W (in an aluminium heatsink case), IR-LED driver PCB, powerbank, USB hub, and camera; were consolidated onto a single prototype. The reference object became a 600 × 500 mm matte-white foam-PVC board mounted on 15 mm plywood with stainless-steel screws and saddle brackets, providing a broad, non-buoyant target. A 4 mm PVC foam board was cut to fit inside a weather-resistant box and lined with Velcro for rapid component placement. Adjustable steel L-bracket hinges and a wing-nut lock allowed precise tilt alignment against a mounting structure. Tension straps and a rear wooden block anchor provided secure fastening. These modular fixtures enabled consistent camera orientation.

By Prototype V5, the hardware package balanced modularity, weather resistance, and low-power off-grid operation: IR illumination drew only 10 mAh/h in regular bursts; the 13 400 mAh powerbank enabled 26 h runtime; and the camera, filters, and PCB remained firmly secured within a single sealed enclosure. Each design decision, whether selecting a MOSFET for LED switching or adopting Velcro-lined platforms for rapid reconfiguration, was driven by field robustness and low-cost simplicity.

All design decisions can be found in a chronological fashion in the development log in Annex G.

C

Criteria evaluation

This section evaluates how the developed water level monitoring system performed against the project's design requirements. A critical assessment is made per criterion, supported by the field test data. Where relevant, additional insights for future improvement are identified.

C.1. Measurement Performance

Measurement Availability (Requirement 2 + 3 — Must-have)
The system was designed to ensure that measured water level data is processed and made available within 5 minutes after collection. Field observations indicate that the system processed each capture well within the available timeframe (~20 s). During the test, the Raspberry Pi handled image capture, processing, and local storage without noticeable delay.

Continuous measurement was stable across the 42-hour deployment. No system failures or crashes were recorded, and the algorithm operated autonomously as intended. While the test duration was limited to two days, the initial signs are promising. However, longer deployment periods are needed to confidently confirm robustness for continuous operation under more variable and challenging conditions.

Water Level Detection Accuracy (Requirement 6 — Should-have)
The water level detection system achieved a Mean Absolute Error (MAE) of 1.97 cm and a Root Mean Squared Error (RMSE) of 2.79 cm using the mean-difference method, compared to manual references. The KS-test method performed slightly better, with an MAE of 1.97 cm and an RMSE of 2.49 cm. A clear bias was observed in both methods: 1.57 cm for mean-difference and 1.02 cm for KS-test. This bias is likely caused by surface reflectance and water transparency effects, particularly under daylight conditions.

95% of the detected water levels fell within 4.64 cm of the reference when using the KS-test method, and 4.17 cm using the mean-difference method. While the system does not yet meet the 2 cm accuracy design goal (only 64–65% of detections fall within that margin), these results indicate a solid baseline. Post-processing corrections could likely reduce the bias and improve accuracy further, bringing the system within specification and supporting its strong potential for low-cost monitoring applications.

All-Condition Measurement (Requirement 5 — Must-have)
The device successfully captured water level measurements during both daytime and nighttime. Interestingly, the results show slightly better precision during night periods. This likely results from the fact that under infrared-only illumination, the waterline becomes visually more distinct due to reduced ambient reflections and longer shutter times, leading to sharper transitions in the images. Daytime conditions, especially around dusk or in the presence of glare, posed more challenges. Despite this, the device remained operational and continued to detect waterlines throughout all lighting conditions, fulfilling the requirement.

Flood Protection (Requirement 1 — Must-have)
One key design advantage is that the system can be installed at a significant distance from the river, reducing the risk of flood damage. During field testing, the device was mounted approximately 8.63 meters from the riverbank, a safe distance under normal and moderate flood conditions. While extreme flood scenarios were not encountered, the elevated and flexible installation potential meets the flood resilience requirement conceptually. It would be valuable in future research to investigate how system performance varies with different distances to the reference object, as greater distances might influence detection robustness.

Secure Installation (Requirement 7 — Should-have)
During the two-day test, the device remained firmly in place, and no movement or misalignment was observed. Nevertheless, the mounting system was designed for short-term field testing, not extended deployments. Further work should focus on developing a more versatile and durable mounting solution, ideally in close collaboration with end users to ensure practicality and ease of use in diverse field conditions.

Weather Resistance (Requirement 8 — Should-have)
The device remained operational during light rainfall events experienced during the field test. No water ingress or malfunction was observed. However, the prototype has not yet been exposed to extreme weather conditions such as heavy storms, strong winds, or high temperatures. Full validation of weather resistance thus remains an open item for future testing.

Vandalism Resistance (Requirement 9 — Should-have)
No incidents of vandalism occurred during field deployment. However, this is likely due to the protected nature of the test site. Vandalism resistance was approached mainly through design choices: keeping the device as small as possible, maintaining a low material value to minimize incentive, and enabling elevated or hidden installation. While these strategies are promising, the actual effectiveness of the design against vandalism was not formally tested and remains to be evaluated in more exposed environments.

C.2. Operational Aspects

Cost Effectiveness (Requirement 4 — Must-have)
The total material cost of the prototype was approximately €290, comfortably below the €400 ceiling defined in the requirements list. The choice of affordable components, open-source design, and low-power architecture ensured cost efficiency.

Energy Independence (Requirement 10 — Could-have)
The device operated autonomously for approximately 42 hours on a single battery cycle. However, the goal of achieving at least six months of energy independence was not yet met. Solar panel integration in combination with a power management system is foreseen as a promising next step to achieve this target, but it was beyond the scope of the current research phase.

Data Storage Backup (Requirement 11 — Could-have)
The system successfully backed up data to a USB storage device during GUI operation. However, USB backup functionality was not yet implemented under CLI operation, which is the preferred field setting. This is recognized as a small technical gap, and future work could easily address this with minor script adaptations.

Data Accuracy Filtering (Requirement 12 — Won't-have) The current prototype does not implement active filtering of statistical outliers. Outlier handling and automatic quality control remain identified areas for future development, to enhance system reliability before transmitting data to external servers.

Maintenance-Free Operation (Requirement 13 — Won't-have)
While the prototype operated autonomously for ~42 hours without maintenance, the original design goal of six months maintenance-free operation was not tested. Achieving true long-term maintenance-free functionality would require improved robustness, weatherproofing, and energy supply upgrades, forming important directions for future work.

C.2.1. Summary of Performance

This section summarizes the degree to which the developed water level monitoring system met the predefined design requirements. Each requirement is assessed based on field test results and development observations. For clarity, performance is indicated using green (met), orange (partially met), and red (unmet).

ID	Requirement	MoSCoW	Assessment	Status
1	Flood protection	Must-have	Device can be installed at safe distance from river; no flood exposure during test but conceptually validated	Met
2	Measurement availability	Must-have	Measurement cycle within target interval	Met
3	Continuous measurement	Must-have	Stable operation over 42h test without crashes	Met
4	Cost effectiveness	Must-have	Total cost (€290) well below €400 limit	Met
5	All-condition measurement	Must-have	Functioned both during day and night; better precision at night	Met
6	Water level detection accuracy	Should-have	95% accuracy ±4cm (outside ±2 cm)	Unmet
7	Secure installation	Should-have	Stable for short-term field test; mounting system needs development for long-term use	Partly met
8	Weather resistance	Should-have	Light rain tolerated; no extreme weather tested	Partly met
9	Vandalism resistance	Should-have	No vandalism occurred; design minimizes risk but effectiveness untested	Partly met
10	Energy independence	Could-have	42h on battery; 6-month target not achieved, solar future option and power management system to be implemented	Unmet
11	Data storage backup	Could-have	Worked in GUI mode; not yet in CLI operation	Partly met
12	Data accuracy filtering	Won't-have	Outlier filtering not yet implemented	Unmet
13	Maintenance-free operation	Won't-have	Short test duration; maintenance-free goal untested	Unmet

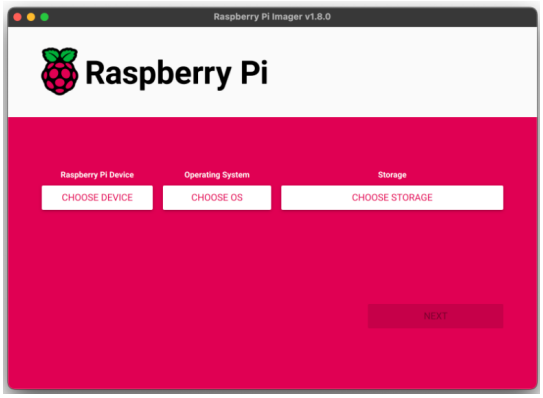
Table C.1: Summary of requirement evaluation with MoSCoW classification and performance status.

D

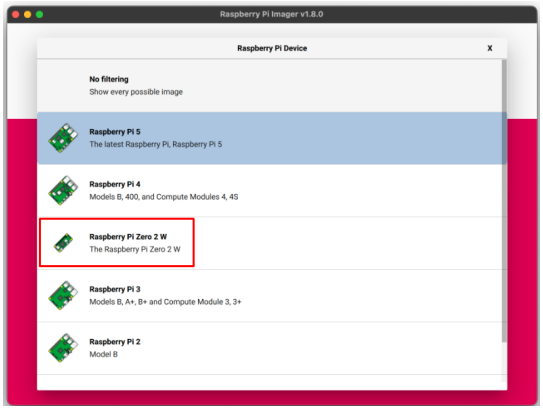
System Setup Manual

D.1. Operating System

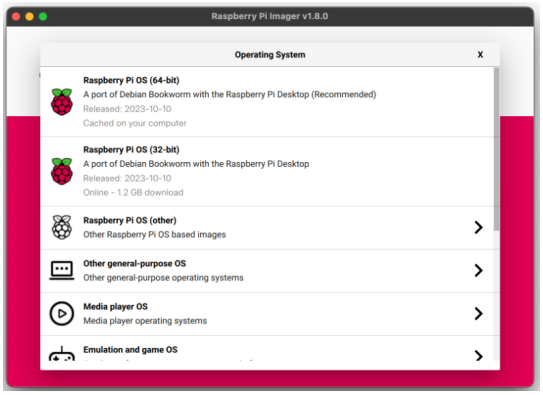
The Raspberry Pi OS is flashed onto the microSD card using the official *Raspberry Pi Imager*, available via raspberrypi.com. After launching the Imager, select the Raspberry Pi Zero 2 W (Figure D.1b), followed by the 64-bit Debian Bookworm with desktop OS image (Figure D.1c). Insert the microSD card using a card reader (Figure D.1d). Before writing, optional settings such as username, Wi-Fi, hostname, locale, keyboard layout, and SSH/VNC access can be preconfigured. Once applied, the OS is written to the card, producing a ready-to-use environment.



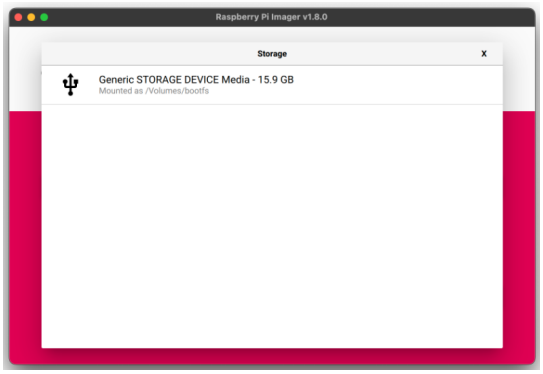
(a) Select target device.



(b) Select processor module.



(c) Select operating system.



(d) Select target storage.

Figure D.1: Overview of the SD-card imaging process.

D.2. First-Boot Configuration: Swapfile and Updates

To allow initial GUI-based configuration, the Pi Zero 2 W can be connected to an external monitor (HDMI → mini-HDMI) and controlled via USB hub (micro-USB → USB-A) with keyboard and mouse. After booting into the desktop, the default 100 MB swapfile is increased to 1024 MB to improve stability under image-processing load; care should be taken to avoid excessive SD-card wear (by increasing the swapfile size too much) (source). This can be achieved using the terminal as follows:

1. Disable swap temporarily:

   ```
   sudo dphys-swapfile swapoff
   ```

2. Edit the swap configuration file:

   ```
   sudo nano /etc/dphys-swapfile
   ```

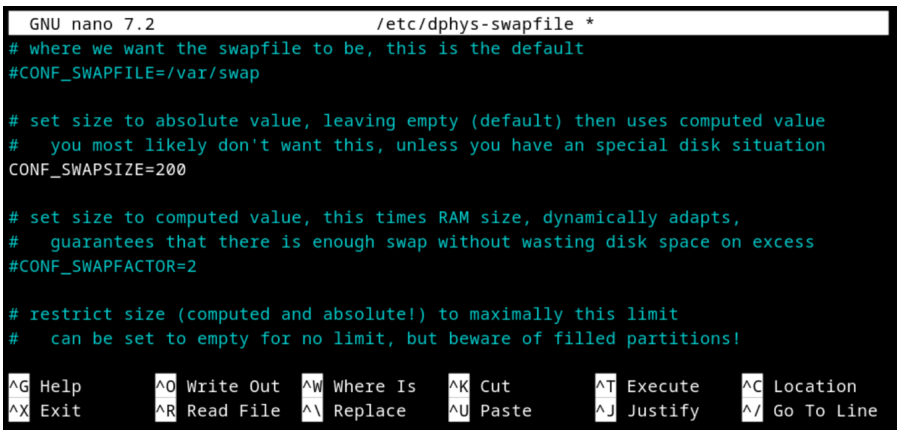


Figure D.2: Swapfile configuration.

3. Increase swap size: Change the line `CONF_SWAPSIZE=100` to `CONF_SWAPSIZE=1024`. Save changes with `CTRL + O`, `CTRL + X`, and confirm with `Y + ENTER`.
4. Apply the new swap size:

   ```
   sudo dphys-swapfile setup
   ```

5. Re-enable swap:

   ```
   sudo dphys-swapfile swapon
   ```

6. Reboot the system:

   ```
   sudo reboot
   ```

After reboot, the system is stable for all GUI and processing tasks. Regular package updates are then performed:

```
sudo apt update
sudo apt upgrade
```

D.3. Package Installation

The water-level detection software depends on both system-level camera drivers and a set of Python libraries. Before operation, ensure the following packages are installed on Raspberry Pi OS:

System & camera features

- libcamera-apps
- libcamera-dev
- v4l-utils

Core Python libraries

- python3-picamera2 (Picamera2 API & libcamera bindings)
- python3-pillow (PIL image handling)
- python3-numpy (array operations)
- python3-matplotlib (plotting backend)
- python3-scipy (signal processing & statistics)
- python3-requests (HTTP for OCR API)
- python3-psutil (system metrics: CPU, memory)
- python3-rpi.gpio (GPIO control)

This can be achieved by running the following line in the terminal command prompt:

```
sudo apt update
sudo apt install -y libcamera-apps libcamera-dev v4l-utils \
python3-picamera2 python3-pillow python3-numpy \
python3-matplotlib python3-scipy python3-requests \
python3-psutil python3-rpi.gpio
```

D.4. Auto-Start Service

To enable the water-level detection algorithm to launch automatically after any reboot, create a `systemd` service unit. This ensures the system is self-recovering: if it loses power or crashes, it will come back online and resume measurements without manual intervention. In combination with a power-management board, energy use can be minimized by only powering the Pi when a measurement cycle is due (see Section 8).

Create and edit the service file with:

```
sudo nano /etc/systemd/system/<filename>.service
```

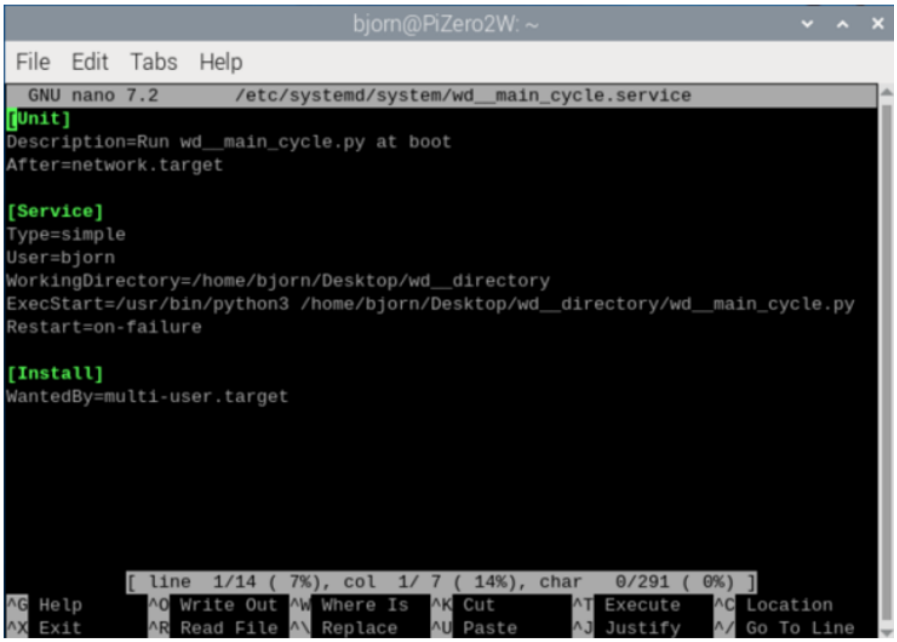


Figure D.3: Example of .service file

This file will be available on `<GitHub link>`.

The service file begins with a [Unit] section that names the task and tells systemd to wait until the network is online before starting. In the [Service] section, it specifies that systemd should run the Python interpreter on your wd__main_cycle.py script as the pi user, from the project's working directory, and automatically restart it if it ever crashes. Finally, the [Install] section hooks the service into the normal multi-user boot sequence so that enabling it causes the script to launch on every reboot without any further intervention.

After placing the service unit file into /etc/systemd/system/, the systemd manager must reload its configuration so that the new unit becomes available. The service can then be configured to start automatically on boot or prevented from doing so, and it supports manual start and stop operations as well as status inspection. This ensures that the water-level detection algorithm recovers automatically after power interruptions and can be managed interactively for troubleshooting or maintenance.

The relevant commands are:

```
sudo systemctl daemon-reload
sudo systemctl enable <filename>.service
sudo systemctl disable <filename>.service
sudo systemctl start <filename>.service
sudo systemctl stop <filename>.service
sudo systemctl status <filename>.service
```

E

Remote access manual

To facilitate remote access and file transfer, a VNC software and SFTP client are configured.

E.1. Remote access GUI mode; RealVNC

1. Enable SSH and VNC interfaces via **Menu** → **Preferences** → **Raspberry Pi Configuration** → **Interfaces**, selecting **Enabled** for both **SSH** and **VNC**.

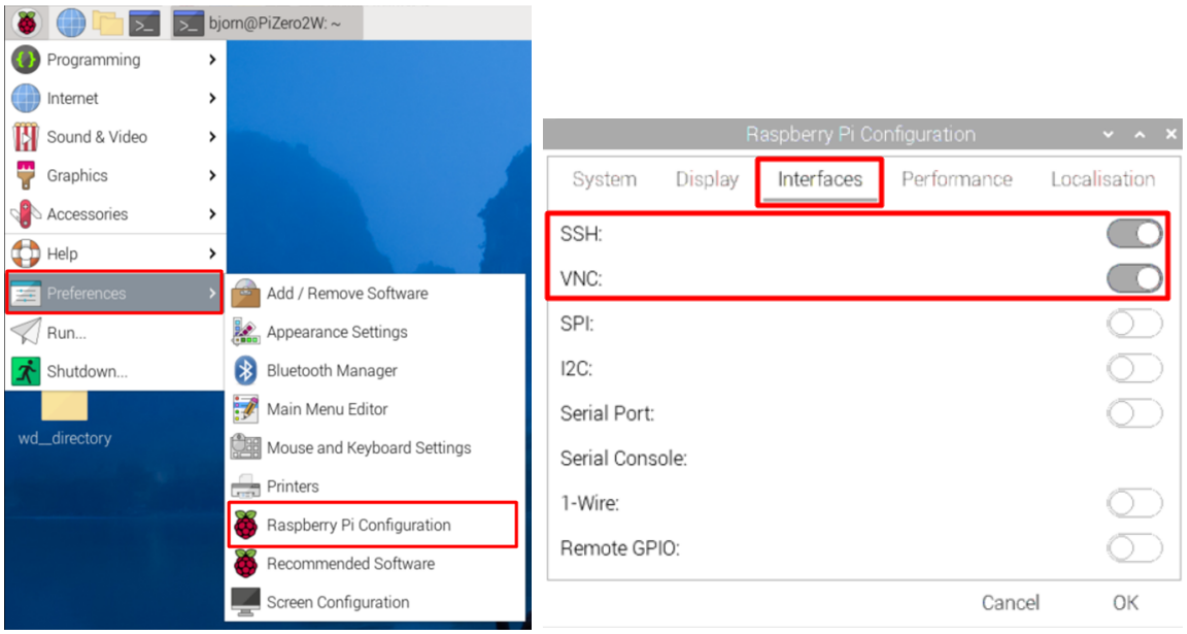


Figure E.1: Enable SSH and VNC settings in RPI environment.

2. From a workstation, download and install RealVNC Viewer (`https://www.realvnc.com/download/`).
3. In VNC Viewer, create a new connection using the Pi's IP address (visible by hovering over the network icon on the RPi desktop) as the VNC Server address.
4. Authenticate with the Pi user credentials configured during OS imaging; upon success, the Pi desktop is displayed for remote control.

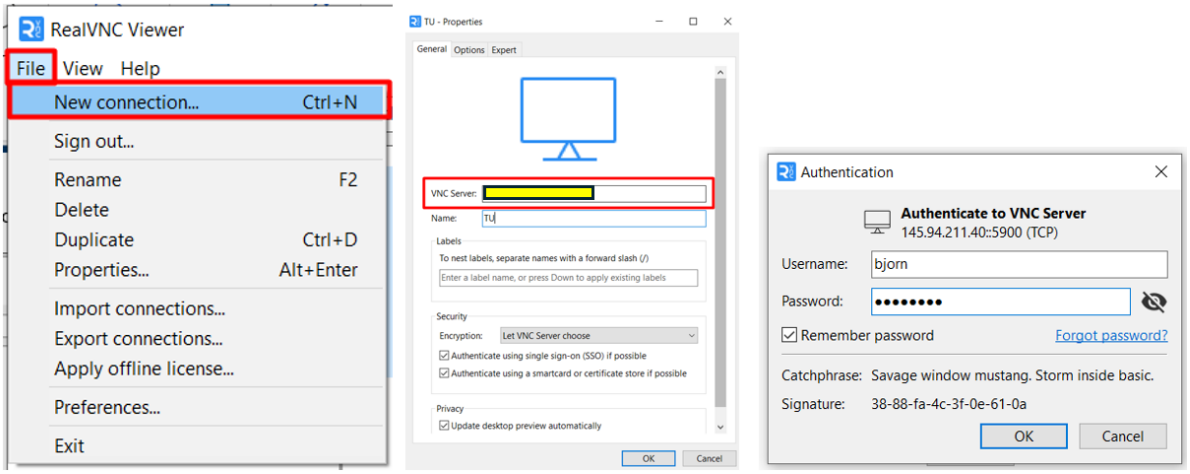


Figure E.2: Setting up a remote connection using RealVNC.

Now the system can be controlled form a pc assuming both systems are using the same internet connection.

E.2. Remote access CLI mode; SSH & PuTTY

In deployments where power savings dictate operating without a monitor or GUI, the Raspberry Pi must be accessed remotely via SSH over Wi-Fi or a mobile hotspot. The following steps describe enabling the SSH server, configuring wireless network credentials, and establishing an SSH connection from a Windows workstation using PuTTY(source PuTTY).

Initial SSH enablement (with monitor & keyboard)

1. Connect the Pi to a monitor and USB keyboard.
2. Open a terminal and run:

   ```
   sudo raspi-config
   ```

3. Navigate to **Interfacing Options** → **SSH**, select **Enable**, then **OK**.

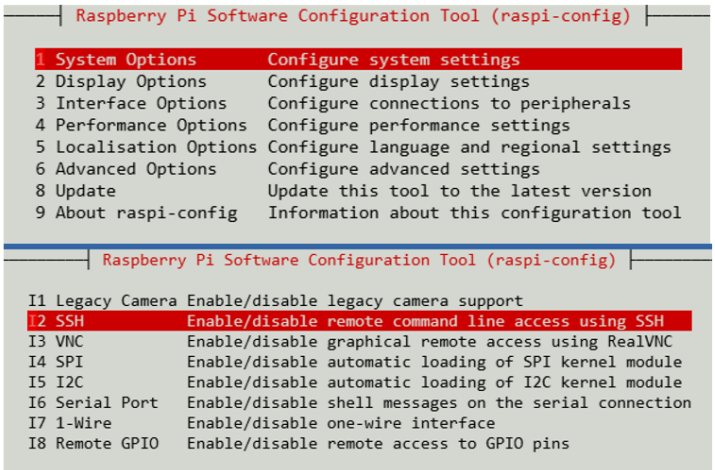


Figure E.3: Set interfacing options.

Configure Wi-Fi credentials:

4. In the same `raspi-config` tool, go to **System Options → Wireless LAN**.

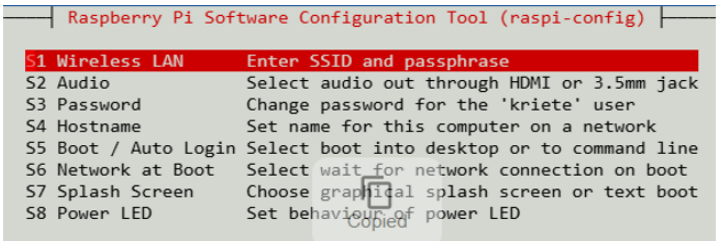


Figure E.4: Configure Wireless LAN connection.

5. Enter your hotspot's SSID and passphrase when prompted.
6. Exit and allow the Pi to reboot (if prompted).

Determine the Pi's IP address on the hotspot:

7. Open a terminal and run:

   ```
   ip a
   ```

8. Note the address listed under the Wi-Fi interface.

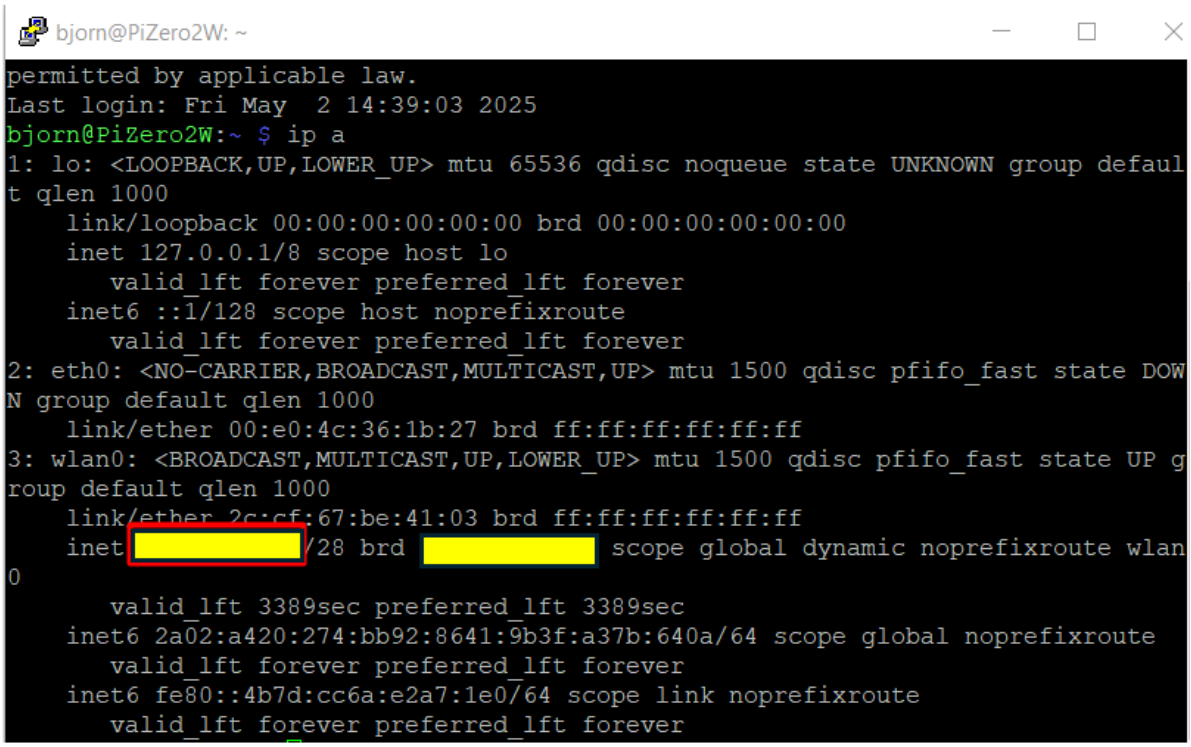


Figure E.5: Location of IP-address

Establish the SSH connection:

9. Download and install PuTTY from `https://www.putty.org/`
10. Launch PuTTY.
11. In **Host Name (or IP address)** enter the Pi's IP from step 8.
12. Ensure **Port** is set to 22 and **Connection type** is SSH.

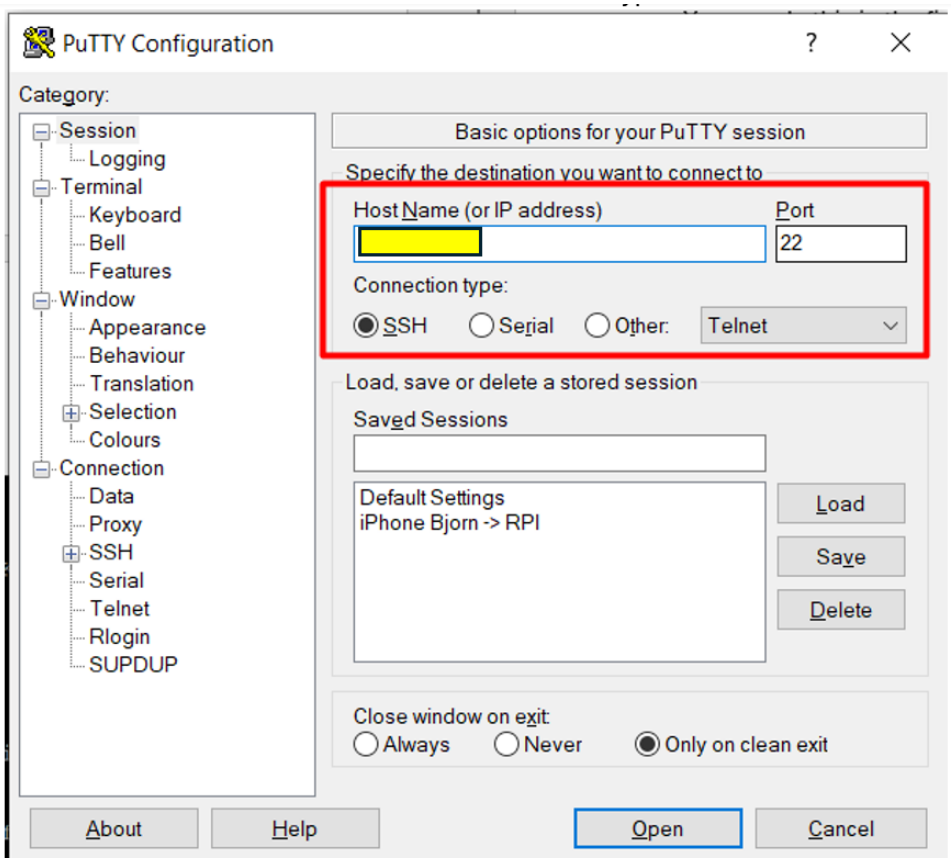


Figure E.6: Insert Host name/ IP-address

13. Click **Open**, then log in with your Raspberry Pi username and password.

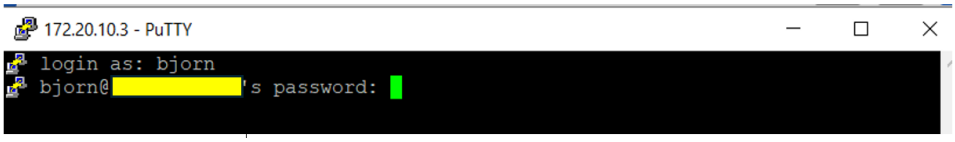


Figure E.7: Login interface for PuTTY

Once connected, you can manage the Pi entirely via the terminal in CLI mode.

E.3. SFTP file transfer via WinSCP

To enable reliable file transfer, the SFTP client WinSCP is configured.

1. Install WinSCP on the workstation (`https://winscp.net/`).
2. Launch WinSCP and configure a New Site with:
   - File protocol: **SFTP**
   - Host name: `Pi IP address`
   - Port number: `<standard>`
   - User name: `<username>` (as set in the OS imager)
   - Password: `<password>` (as set in the OS imager)
3. Save the site and connect; navigate to the project output directory (e.g. `/home/pi/wd_directory/output/`) to upload or download images, logs, and CSV files.

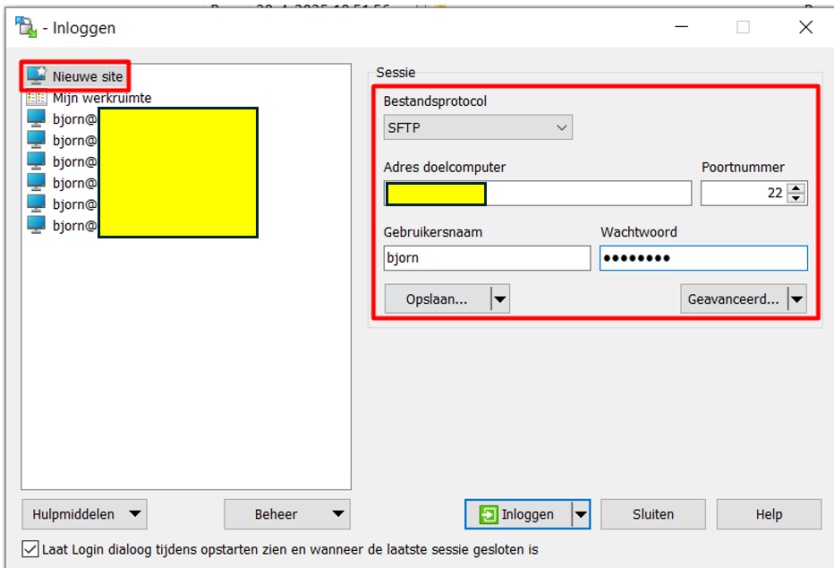


Figure E.8: Setup file transfer configuration.

Once connected, WinSCP can also be used to upload the entire project directory (cloned from the GitHub repository) to the Pi. Simply clone the wd__directory locally on your workstation (e.g. via git clone (GitHub link)), then in WinSCP's local pane navigate to that folder and drag it into /home/pi/ on the remote pane. This transfers all scripts, modules, and configuration files in one step, ready for execution.

F

Operation and calibration manual

F.1. Calibration

Accurate water-level detection requires that the optical axis and algorithm parameters be tuned to the installed reference object. With the enclosure sealed and the reference board fixed, calibration proceeds in two stages: physical alignment and parameter extraction.

First, a live preview is used to finalize camera orientation. In GUI mode (using RealVNC), `Camera_preview.py` (in `wd__Calibration/`) is launched so that the system orientation can be adjusted to the reference object. Once the mount is locked in place, the preview window is closed.

Next, rotation and crop parameters are determined by running `get_crop_angle_parameters.py`, which guides the user through three interactive clicks:

1. Rotation: two clicks on the waterline (left then right) compute the angle required to horizontalize the image.
2. Cropping: four clicks define the rectangular region of interest around the board.
3. Preview: a final overlaid image confirms that the waterline is level and the crop box fully encloses the board.

Once the rotation and crop values are confirmed, they are entered into `wd__config_cycle.py` under `"processing_params"` and `"crop_params"`. Additionally, the interval between measurement cycles governed by the `cycle_rest_seconds` parameter in `wd__config_cycle.py`, should be reviewed and adjusted to suit the deployment requirements (9 minutes in this research). This ensures that the system waits the correct amount of time between successive captures without needing manual intervention.

Verification is then performed by executing a single detection cycle, ideally from within the Thonny IDE (standard Python terminal on RPI OS) so that the initial processing output can be observed. All results (captured images, overlay plots, and CSV logs) are saved to `wd__directory/wd__results/`, and Thonny's console displays informational and error messages to facilitate debugging.

After calibration is confirmed, the Pi must be switched to a CLI mode (text-console) to save power. This is accomplished by running `sudo raspi-config` in a terminal, navigating to:

```
Sudo raspi-config
System Options
Boot
Console Text Console
```

Next, select "Finish" and agree to reboot by selecting "Yes".

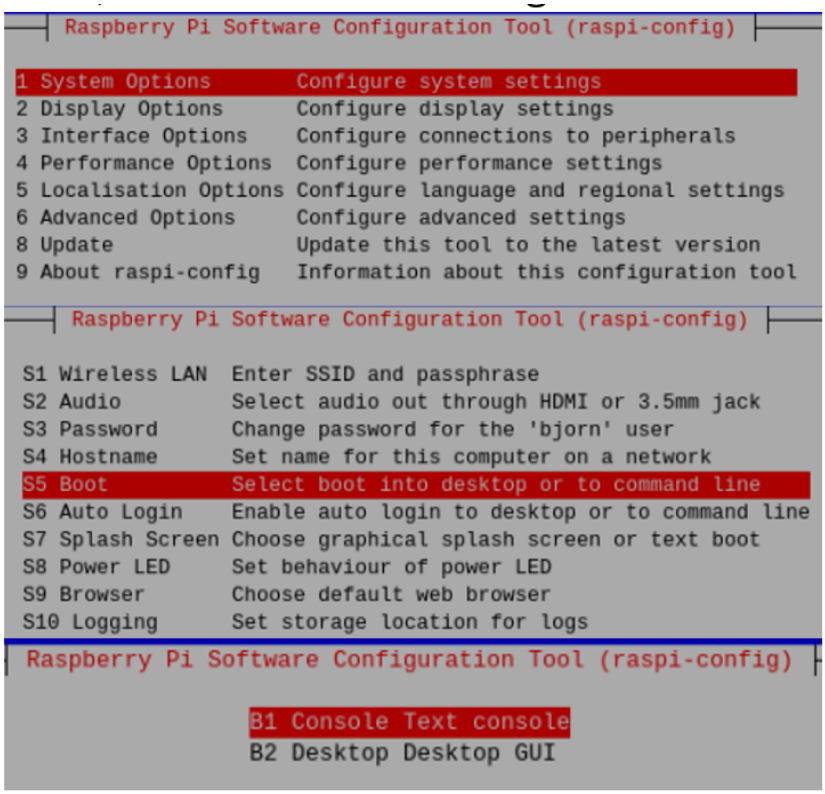


Figure F.1: Switching RPI boot-up configuration

Once the system restarts, SSH access via PuTTY (Section 5.2.5) restores terminal control. The status and activity of the detection service may then be checked with the appropriate `systemctl` commands, and if network posting is enabled, successful uploads can be verified on the OpenRiverCam server.

F.1.1. Device setup checklist

1. Clear al data storages / charge all batteries
2. Setup mobile hotspot
   - `<host name>`, `<host password>`
   - Connect laptop with mobile hotspot
   - Turn on Raspberry Pi and seal device
3. Mount device facing Reference Object
4. Run: `Camera_preview.py` in `wd__Calibration` using Thonny
   - Adjust device orientation so that OI is in the frame
   - Secure camera orientation
   - Adapt lens focus using lens tool
   - Exit out
5. Run: `get_crop_angle_parameters.py` in `wd__Calibration` using Thonny
   - Rotation: two clicks on the waterline (left then right) compute the angle required to horizontalize the image.
   - Cropping: four clicks define the rectangular region of interest around the board.
   - Preview: a final overlaid image confirms that the waterline is level and the crop box fully encloses the board.

6. Open: `wd__config.py` in `wd__directory`
    - Insert rotation and crop parameters
    - Insert cycle wait time under the `cycle_rest_seconds` parameter
    - Save & Exit out
7. (Test)Run: `wd__main_cycle.py` in `wd__directory`
    - Check for errors
    - Check behaviour LED
    - Check cropbox file in `wd__results`
    - Check 4modes file in `wd__results`
    - Check raw images
    - Check writing to usb
    - Check sleep parameter
8. Switch to no GUI mode
    - Open terminal

```
Sudo raspi-config
1 system options
S5 boot
B1 Console Text console
Finish
```

    - Reboot now: no
9. Turn on reboot service file
    - `sudo systemctl enable wd__main_cycle.service`
    - `sudo systemctl status wd__main_cycle.service`
10. Reboot
11. Check IR led if program is running correctly
12. (optionally) Utilize a SSH connection via PuTTY to validate operation

F.1.2. Disable device checklist

1. Take down device
    - Move to safe location (dry and not above water)
2. Make SHH connection via PuTTY
3. Check, stop and disable algorithm
    - `sudo systemctl status wd__main_cycle.service`
    - `sudo systemctl stop wd__main_cycle.service`
    - `sudo systemctl disable wd__main_cycle.service`

F.2. Operation

Once calibration is complete and the Pi has been configured to boot into CLI mode, measurement cycles proceed automatically at the interval specified by the `cycle_rest_seconds` parameter in `wd__config_cycle.py`. In normal operation, the system illuminates the reference board, captures an image, runs the detection algorithm, uploads result if enabled, and then sleeps until the next cycle without any further user intervention.

If a cycle needs to be triggered on demand the detection service may be restarted manually. In the terminal, the following command forces an immediate cycle.

```
sudo systemctl restart wd_main_cycle.service
```

To confirm that the service is running correctly and view recent log entries, the operator can issue:

```
sudo systemctl status wd_main_cycle.service
```

Each cycle writes its outputs to the file system: raw photographs appear under `~/output/raw_images/`; annotated PNGs with the detected waterline are saved in `~/results/`; and a line is appended to the CSV log at `~/results/algorithm_results.csv`. These files can be investigated or retrieved remotely over the RealVNC connection by first pausing the detection service and reverting to GUI mode. To do this, stop the service:

```
sudo systemctl stop wd_main_cycle.service
```

then run `sudo raspi-config` to switch the boot target back to the GUI mode and reboot. Once the Pi restarts, connect via RealVNC and use WinSCP or a similar SFTP client to access and copy the output files. Once the transfer is complete, CLI mode is restored by reconfiguring the boot target and restarting the service:

```
sudo systemctl start wd_main_cycle.service
```

Periodic maintenance includes verifying that the battery maintains sufficient charge, inspecting the reference board for any debris or alignment shift, and checking the enclosure's seals and mounting hardware. By following these procedures, the system delivers reliable, unattended water-level measurements while still allowing straightforward manual checks and data retrieval when needed.

G

Development LOG

7 DEVELOPMENT LOG

7.1 PROTOTYPE V1:

7.1.1 Hardware V1

Components

Function	component
Power supply	ISY powerbank 20.000mA, 2.4A, 5V
Processing	Raspberry Pi Zero 2W
Camera	RPI Camera module V2
Internet connection	Tp-Link portable router
Casing	RPI Zero standard casing

Staff gauge v1

Piece of foam temporarily attached to a spade using tiewraps.

Power source

The powersource used in the first prototype is a Lithium-polymeer battery capable of outputting 2x 5 volts with 2.4A current. This power source is taken as a first option as it was available by the researcher and able to stably power both the operating system (RPI) and the portable router. It contains a capacity of 20.000mA which is more than enough to do preliminary testing of a few hours at a time. The battery weighs about 375g, and dimensions are 73.5x24x152mm. The battery costs about 20€.

Operating system

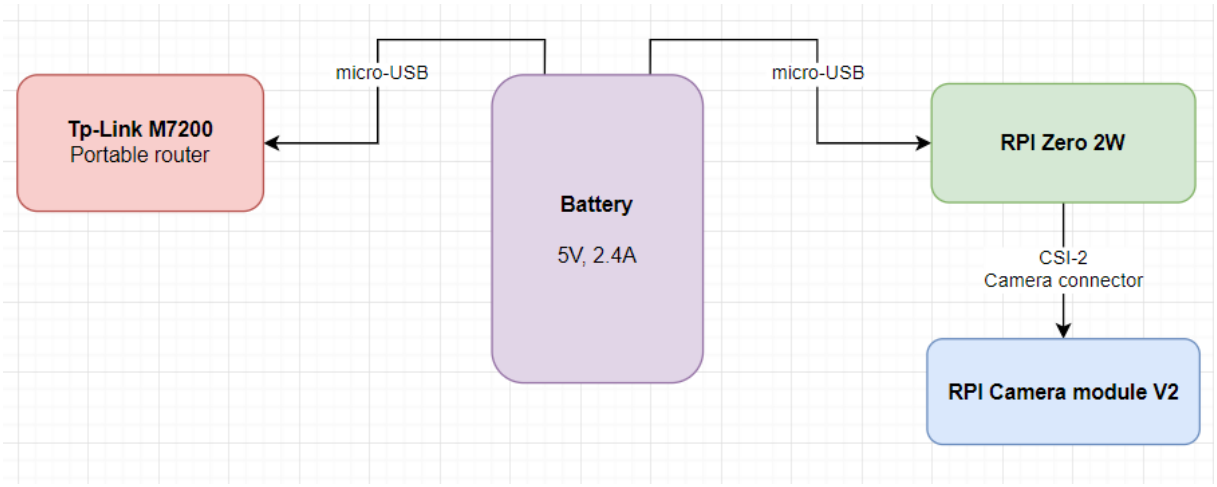
The raspberry pi zero 2W contains a quad-core 64-bit ARM Cortex-A53 processor clocked at 1GHz and 512MB of SDRAM. The device offers 2.4GHz 802.11 b/g/n wireless LAN and Bluetooth 4.2, along with support for Bluetooth Low Energy (BLE). It has a built in wireless LAN and is dimensioned 65mm x 30mm. The board has a microSD card slot, a CSI-2 camera connector, a USB On-The-Go (OTG) port and an unpopulated footprint for a HAT-compatible 40-pin GPIO header. It can be powered via a micro USB socket and output via a mini HDMI port (Raspberry Pi, 2024). The RPI zero 2 W costs about 20€.

Camera module

The camera module V2 is an 8-megapixel module and dimensioned around 25 x 24 x 9 mm. it weighs 3g and is capable of capturing video in 1080p47, 1640 x 1232p41 and 640x480p206. It uses a Sony IMX219 sensor with a resolution of 3280 x 2464 pixels. It comes with adjustable focus and a focal length of 3.04mm. Horizontal Field of View (FoV) is 62.2 degrees and Vertical Field of View (FoV) is 48.8 degrees. Its maximum exposure time is 11.76 seconds, and the product is available in a NoIR version, where the IR filter is removed (raspberrypi, 2025). The camera module V2 costs about 18€.

Portable router

The TPL-link M7200 is a portable router that supports 4G FDD/TDD-LTE and 3G DC-HSPA signals. It contains a battery with a capacity of 2000mAh which translates to roughly 8 hours of Wi-Fi connection. It can be powered via a micro usb-port and allows 1 sim card. Its capable of 150 Mbps download speed and 50 Mbps upload speed. Its dimensioned 94 x 56.7 x 19.8mm (tp-link, 2025) (tp-link, 2025). It costs about 50€.



7.1.2 Algorithm V1

Nr.	File name	link	Description
V1_1	Scripts/Otsu_cropping_th-factor_21112024_V4.ipynb	Algorithm v1	Otsu method + preprocessing
V1_2	Scripts/Timelapse_5mins.py	Timelapse v1	Timelapse image capture

V1_1

Algorithm component V1_1 utilises packages numpy for basic python computations, Pillow and scipy.ndimage for image processing, matplotlib for visualisation, os for managing paths and time to analyse computation demand of the script. The OpenCV package is refrained from because of its size, which is not recommended while using devices running on low processing power like the Raspberry Pi Zero 2 W.

The Otsu thresholding method is a widely used technique in computer vision for image segmentation. Its purpose is to determine an optimal threshold value that separates pixels into two classes: the object of interest and the background. [https://medium.com/@vignesh.g1609/image-segmentation-using-otsu-threshold-selection-method-856ccdacf22]

The algorithm analyses the histogram of pixel intensities in the image and identifies the threshold value that minimizes the variance within each class. After applying this threshold, the image is binarized, which simplifies analysis using basic coding techniques [https://medium.com/@vignesh.g1609/image-segmentation-using-otsu-threshold-selection-method-856ccdacf22].

Medium source based on this paper:
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4310076&tag=1

Description	Formula
Threshold value	k
Probability of pixel in image - Background class 0: pixel i below k - Foreground class 1: pixel i above k	$p_i = n_i/N$
Probability of picking point from class 0	$\omega_0 = \sum_{i=0}^{k} p_i$
Probability of picking point from class 1	$\omega_1 = \sum_{i=k}^{255} p_i = 1 - \omega_0$
Mean levels of each class	$\mu_0 = \sum_{i=1}^{k} ip_i$ $\mu_1 = \sum_{i=k}^{255} ip_i$
Variance of each class	$\sigma_0^2 = \sum_{i=0}^{k} (1-\mu_0)^2 p_i/\omega_0$ $\sigma_1^2 = \sum_{i=0}^{k} (1-\mu_1)^2 p_i/\omega_1$
Within class variability	$\sigma_W^2 = \omega_0\sigma_0^2 + \omega_1\sigma_1^2$
Between class variability	$\sigma_B^2 = \omega_0(\mu_0 - \mu_T)^2 + \omega_1(\mu_1 - \mu_T)^2$ $= \omega_0\omega_1(\mu_1 - \mu_0)^2$
Total variance	$\sigma_T^2 = \sum_{i=1}^{L} (i - \mu_T)^2 p_i$
Otsu Function	$\sigma_B(k) = \omega_0(k)\sigma_0^2(k) + \omega_1(k)\sigma_1^2(k)$

[https://medium.com/@vignesh.g1609/image-segmentation-using-otsu-threshold-selection-method-856ccdacf22]

Otsu method functions

```python
# Function for thresholding an image
def threshold_image(im, th):
    return np.where(im >= th, 255, 0).astype(np.uint8)

# Function to compute Otsu's criteria
def compute_otsu_criteria(im, th):
    thresholded_im = im >= th
    nb_pixels = im.size
    nb_pixels1 = np.count_nonzero(thresholded_im)
    weight1 = nb_pixels1 / nb_pixels
    weight0 = 1 - weight1
    if weight1 == 0 or weight0 == 0:
        return np.inf
    val_pixels1 = im[thresholded_im]
    val_pixels0 = im[~thresholded_im]
    var0 = np.var(val_pixels0) if len(val_pixels0) > 0 else 0
    var1 = np.var(val_pixels1) if len(val_pixels1) > 0 else 0
    return weight0 * var0 + weight1 * var1

# Function to find the best threshold with an adjustment factor
def find_best_threshold(im, adjustment_factor=2):
    threshold_range = range(0, np.max(im) + 1, 10)
    criterias = [compute_otsu_criteria(im, th) for th in threshold_range]
    best_threshold = threshold_range[np.argmin(criterias)]
    adjusted_threshold = best_threshold * adjustment_factor
    return min(adjusted_threshold, np.max(im))  # Ensure it does not exceed the max pixel value
```

A adjustment factor is introduced to the script to improve its performance. This adjustment factor is mannualy adjusted to improve the output of this specific image. It is not part of the original method and not automatically an improvement for other images taken with different lighting or environmental conditions. The original code is taken form the following source: [https://medium.com/@vignesh.g1609/image-segmentation-using-otsu-threshold-selection-method-856ccdacf22]

Processing otsu method output and selecting longest vertical object

Utilizing scipy.ndimage, the longest vertical string of connected pixels is localised in the binary image. for optimization purposes, the width of this object is limited to match the general width of the staff gauge. implementing this greatly reduced the noise of other long connected vertical polygons in the binarized images caused by glare or other background pixels that where not correctly classified.

The function [find_longest_vertical_string] labels connected components and stores them in objects. Next, it iterates trough those objects and calculates the height of each object. If a component is found, it determines if it does fit the max width criteria. It outputs the longest vertical component that confirms the boundary conditions of the manually added max width, which should be implemented by the user and is variable per experiment or setup.

```python
# Function to find the longest vertical connected component
def find_longest_vertical_string(binary_image, max_width=50):
    """
    Find the longest vertical connected component in the binary image
    and limit the width for visualization purposes.
    """
    # Label the connected components
    labeled_image, num_features = label(binary_image)
    objects = find_objects(labeled_image)

    max_height = 0
    longest_vertical_component = None

    # Iterate through each connected component and find the vertical one with the greatest height
    for obj in objects:
        y_slice, x_slice = obj
        height = y_slice.stop - y_slice.start
        width = x_slice.stop - x_slice.start

        if height > max_height and height > width:
            max_height = height
            longest_vertical_component = (y_slice, x_slice)

    # If a component is found, adjust the width to be at most `max_width`
    if longest_vertical_component:
        y_slice, x_slice = longest_vertical_component
        center_x = (x_slice.start + x_slice.stop) // 2  # Find the center of the width
        half_max_width = max_width // 2

        # Limit the x_slice to the central max_width region
        x_start = max(x_slice.start, center_x - half_max_width)
        x_stop = min(x_slice.stop, center_x + half_max_width)
        x_slice = slice(x_start, x_stop)

        longest_vertical_component = (y_slice, x_slice)

    return longest_vertical_component
```

Running the algorithm

- Load and preprocess the image

Opening the image, converting it to grey scale and storing both variants.

- Cropping the image to the area of interest

Depending on the situation, it is recommended to crop the image as much as possible to reduce noise and processing time/power. This step is highly dependent on location and setup.

- Apply otsu thresholding method
- Find longest vertically connected component which fits requirements
- Plot images

For analysis, the original image, grayscale image, cropped image and binary image with the cropped region displayed as an overlay are plotted. This takes significant computational power for the RPI hence this is purely done for analysis and optimisation of the algorithm. In practice, the original image would be posted and stored, together with all information about the processing parameters and waterline detection output.

- A summary file is written to store important information about the run and parameters.

Image capturing script for camera module

Packages that are utilised running this script are os for path management, time for analysis of runtime and computational efficiency and picamera 2 for operating the camera module.
Using the standard configuration settings on the camera module images are taken with an resolution of 3280 x 2464. The script is setup using a timelapse which makes sure that every 10 minues, an image is taken and stored locally. Images names are generated with timestamps using YMDHMS.

7.1.3 Experiment V1

The runtime for the method on the Raspberry Pi Zero 2 W was **356 seconds** when processing pixels individually. By grouping pixels into 10x10 blocks, the runtime was significantly reduced to **61 seconds**, with similar results.

Observations:

- **Sky as noise**: The sky, often light in colour, was frequently detected as an object of interest, introducing noise into the results.
- **Concrete reflections**: Light-coloured concrete surfaces reflected enough light to also be classified as objects of interest, adding noise.
- **Effectiveness of thresholding**: While the thresholding method performed reasonably well in this experiment, its effectiveness may vary under different conditions, such as changes in lighting, cloudiness, or rain.
- **Staffgauge/water line**: the goal of this method is to determine the height of the water level. This means that a clear distinction between the water and staff gauge needs to be determined. In the images can be seen that the Otsu method has difficulties determining what pixel is part of the underwater staff gauge and what part is part of the above water staff gauge. This is caused by disturbed pixel intensities caused bay reflection near the waterline, but also the fact that the camera can detect the underwater part of the staff gauge.
- **Threshold**: during the first test, the threshold of the Otsu method is manually set so it performs the best. However, the best threshold is greatly dependent on environmental conditions like lighting (position of the sun, cloudiness, seasonality, brightness) and precipitation (snow, hail or rain). Also wind and vegetation like leaves flying trough the shot can affect the thresholding.

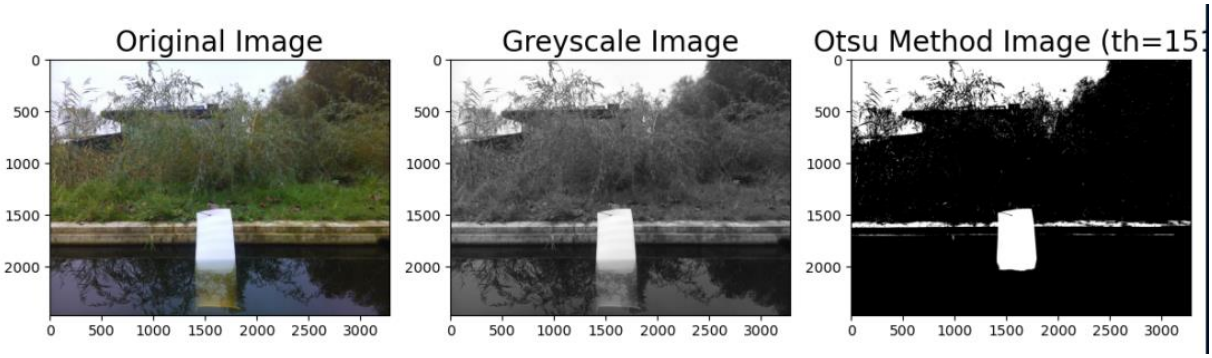


Figure 1: Results of Otsu thresholding method preliminary experiment

The find best threshold function loops trough all possible thresholds between and 255 (range of pixel intensities) and tries to find the best threshold that defines background or object of interest. However this processes can be very computational expensive (while using RPI). Trying every 10th threshold value however sped up the process +- 10x.

10 pixel thresholding **1 pixel thresholding**

```
Time to load and convert image: 0.8919 seconds
Time for thresholding: 27.5857 seconds
Time for plotting: 27.9491 seconds
Vertical height of the water gauge (in pixels): 2080
Time for calculating vertical height: 3.5239 seconds
Total script runtime: 61.0900 seconds
```

```
Time to load and convert image: 0.8207 seconds
Time for thresholding: 296.4523 seconds
Time for plotting: 50.9546 seconds
Vertical height of the water gauge (in pixels): 2080
Time for calculating vertical height: 7.2191 seconds
Total script runtime: 356.1274 seconds
```

7.1.4 Notes on prototype 1 and feedback loop for next prototype

The Otsu thresholding method was not able to differentiate the waterline correctly in a variable environmental conditions. For this method to work, manual adaptation of the thresholding factor is was needed to increase the effectiveness of the thresholding, which is not desirable for an autonomous system. Also reflection of light and clouds on the water surface caused noise in the thresholding method.

For the next design iteration, realising a stable staff gauge and improved detection method have priority. The focus will be on:

- Non floating staff gauge to stabilise the focus area
- Noise reduction in the water level detection algorithm
  - Reflections on water surface
  - Background noise

SCAMPER Heuristics

• Substitute: What materials or processes can be replaced?
  - Staff gauge is not functioning well. Need a more stable gauge that doesn't move in water.
• Combine: What elements can be integrated?
  - A form of noise reduction can be integrated to increase performance (background and reflection noise)
• Adapt: How can the concept be adjusted for improvement?
  - The current otsu thresholding method can be adapted so it is more robust for variable lighting conditions.
• Modify: What aspects (e.g., size or shape) can be altered?
  - The staff gauge has to be altered so it's a more consistent way of detecting the line between water and air.
• Put to another use: Can the concept serve other purposes?
  -
• Eliminate: What features can be simplified or removed?
  -
• Reverse: Can processes be reversed or re-sequenced?
  -

7.2 PROTOTYPE V2

7.2.1 Hardware

Staff gauge v1

Because of an poorly functioning temporary staff gauge v1, a second staff gauge was created. A white hollow piece of aluminium was connected to a handle and marked using water proof marker every 10 cm. the problem with staff gauge v1 was that the foam was too buoyant which caused the staff gauge to float up and not remain in the same location. The new staff gauge was able to be attached to the side of the river bed or experiment setup (gutter) and was much better at creating a stable measure.

7.2.2 Algorithm

A piece of code is introduced that enables making images using a timelapse manner. Using the most updated version of raspberry pi libraries: Dabian bookworm (www.debian.org). utilizing the Picamera2 package, the camera module can be controlled to take images and show previews.

Image capturing

```python
import os
import time
from picamera2 import Picamera2, Preview

def capture_timelapse(output_folder, interval_seconds, duration_seconds):
    """
    Captures timelapse images using the Raspberry Pi camera with timestamps.

    Parameters:
    - output_folder (str): Path to the folder where images will be stored.
    - interval_seconds (int): Time interval between consecutive captures in seconds.
    - duration_seconds (int): Total duration of the timelapse in seconds.
    """
    # Create the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Initialize the PiCamera2
    picam2 = Picamera2()

    # Configure the camera for still capture
    camera_config = picam2.create_still_configuration()
    picam2.configure(camera_config)

    # Start the camera
    picam2.start()

    # Calculate the number of frames to capture
    num_frames = duration_seconds // interval_seconds

    print(f"Starting timelapse: {num_frames} frames, {interval_seconds}s interval")
    print(f"Images will be saved to: {output_folder}")
```

```python
    try:
        for frame in range(1, num_frames + 1):
            # Generate a filename with a timestamp
            timestamp = time.strftime("%Y%m%d_%H%M%S")
            filename = os.path.join(output_folder, f"Delft_{timestamp}.jpg")

            # Capture the image
            metadata = picam2.capture_file(filename)
            print(f"Captured {filename} with metadata: {metadata}")

            # Wait for the next interval
            if frame < num_frames:
                time.sleep(interval_seconds)

    except KeyboardInterrupt:
        print("Timelapse interrupted by user.")

    finally:
        # Stop the camera
        picam2.stop()
        print("Timelapse complete.")

# Run the script
if __name__ == "__main__":
    # Folder where images will be stored
    desktop_path = os.path.expanduser("~/Desktop")
    output_folder = os.path.join(desktop_path, "Timelapse_Images")

    # Timelapse settings
    interval_seconds = 300  # Capture an image every 10 seconds
    duration_seconds = 1800  # Total duration of 5 minutes

    capture_timelapse(output_folder, interval_seconds, duration_seconds)
```

Image processing

Furthermore, the package Pillow (PIL) and scipy.ndimage are used for image processing along side matplotlib to analyse results visually. These packages have a small demand on the minimalistic processing capabilities of the raspberry pi and therefore are preferred over large and more sophisticated libraries like OpenCV.

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import os
from scipy.ndimage import label, find_objects
import time
```

To increase userfriendlyness, a piece of code is introduced to easily switch between the testing environment (PC) and the endproduct/prototype (raspberry pi).

```
# Define the base path for the system (Raspberry Pi or PC)
def get_base_paths(system="pc"):
    if system == "raspberry_pi":
        return {
            "image_path": "/home/bjorn/Desktop/Timelapse_Images",
            "output_path": "/home/bjorn/Desktop/Test_Script_Output"
        }
    elif system == "pc":
        return {
            "image_path": r"C:\Users\bjorn\Desktop\rpi_mirror\Timelapse_Images",
            "output_path": r"C:\Users\bjorn\Desktop\rpi_mirror\Test_Script_Output"
        }
    else:
        raise ValueError("Invalid system. Choose 'raspberry_pi' or 'pc'.")
```

The same method (otsu thresholding) is used to detect the staff gauge in the captured images. To enhance the waterlevel detection capabilities, an additional piece of code is introduced. This piece of code uses scipy to label groups of pixels that are categorized as object of interest by the othsu method. The algorithm detects the largest vertically connected group of pixels and sees this as the staff gauge. other smaller groups of pixels that are labelled as object of interest by the otsu method are hereby filtered out, with the goals of filtering out noise from reflection or mistakenly marked background. To enhance this function, a maximum width is introduced that matches the dimensions of the staff gauge.

```python
# Function to find the longest vertical connected component
def find_longest_vertical_string(binary_image, max_width=50):
    """
    Find the longest vertical connected component in the binary image
    and limit the width for visualization purposes.
    """
    # Label the connected components
    labeled_image, num_features = label(binary_image)
    objects = find_objects(labeled_image)

    max_height = 0
    longest_vertical_component = None

    # Iterate through each connected component and find the vertical one with the greatest height
    for obj in objects:
        y_slice, x_slice = obj
        height = y_slice.stop - y_slice.start
        width = x_slice.stop - x_slice.start

        if height > max_height and height > width:
            max_height = height
            longest_vertical_component = (y_slice, x_slice)

    # If a component is found, adjust the width to be at most `max_width`
    if longest_vertical_component:
        y_slice, x_slice = longest_vertical_component
        center_x = (x_slice.start + x_slice.stop) // 2  # Find the center of the width
        half_max_width = max_width // 2

        # Limit the x_slice to the central max_width region
        x_start = max(x_slice.start, center_x - half_max_width)
        x_stop = min(x_slice.stop, center_x + half_max_width)
        x_slice = slice(x_start, x_stop)

        longest_vertical_component = (y_slice, x_slice)

    return longest_vertical_component
```

Image processing

The image processing algorithm was executed with extensive debugging information printed during runtime, along with a log file for analysis. The processed images were stored in an output folder for review.

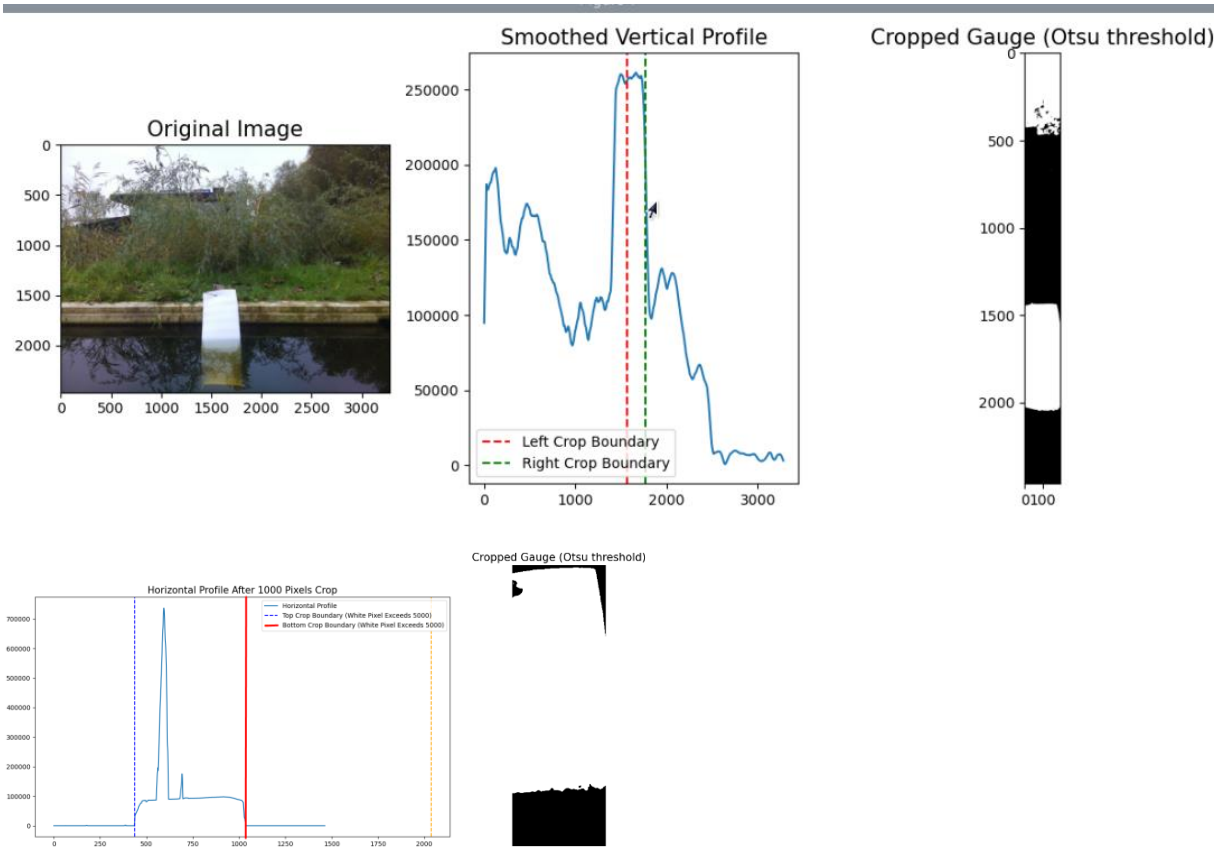
This relatively computationally intensive approach was intended for development and analysis purposes. Future iterations of the product aim to output limited data to conserve power.

Alternative staff gauge detection methods

An alternative approach utilized **peak detection** to identify the staff gauge.

1. The image was cropped to remove the sky.
2. A peak detection algorithm was applied vertically and horizontally to locate the staff gauge.
3. The image was further cropped, and the gauge's vertical pixel count was measured.

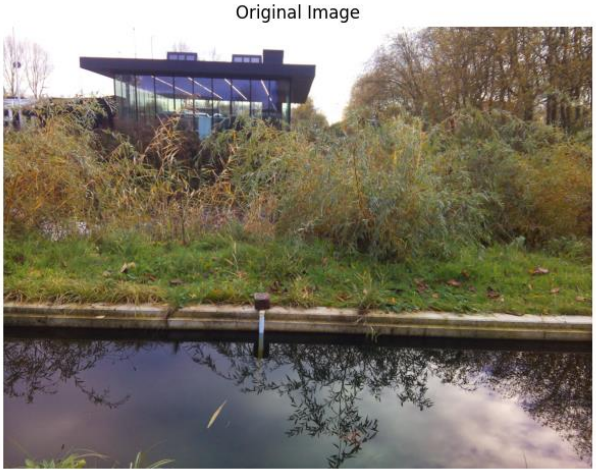
This method required highly accurate thresholding. In cases with significant noise, such as cloud reflections, the cropping and measurements were inaccurate, resulting in unreliable results.



7.2.3 Experiment V2

For this experiment, the gutter was slowly drained over 30 minutes. A timelapse script captured an image every 5 minutes, processed it using the image processing algorithm, and logged key parameters and debugging information into a text file. Images with a blue overlay highlighting the object of interest (staff gauge) were stored in the output folder.

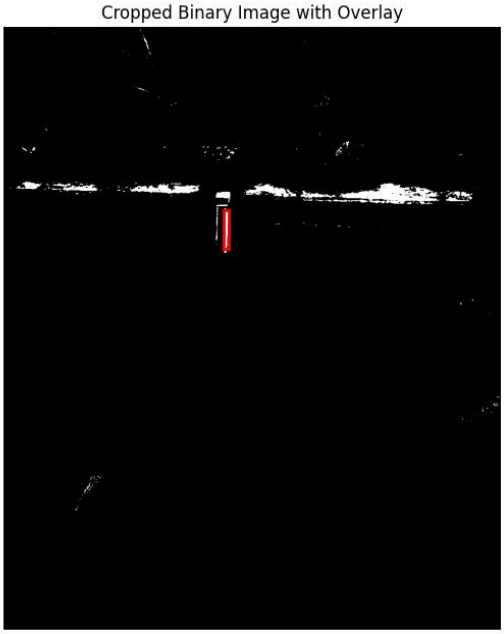
Original Image



Grayscale Image

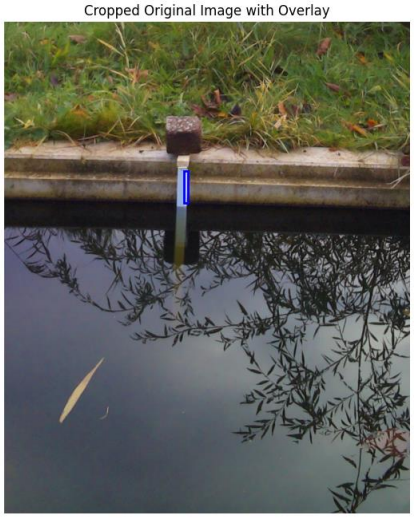
Cropped Grayscale Image

Cropped Binary Image with Overlay



System: pc
Length of the longest vertical object: 79 pixels
Location of the largest vertical object: (Y: 368-447, X: 442-452)
Total script runtime: 6.60 seconds
RUNTIME DETAILS:
Image loading and preprocessing took 0.32 seconds
Thresholding took 0.59 seconds
Finding vertical component took 0.02 seconds
Length of the longest vertical object: 79 pixels
Location of the largest vertical object: (Y: 368-447, X: 442-452)

Cropped Original Image with Overlay



Observations

- Running the script on the Raspberry Pi resulted in a runtime approximately **10 times longer** (around 40 seconds per image) compared to running from pc.
- In the second image taken (2/6), the algorithm incorrectly prioritized a cloud reflection in the water over the staff gauge.
- The dark background of the gutter wall provided good contrast for the staff gauge. However, such contrast may not exist in natural environments.
- Adjustments to the thresholding method were necessary to improve sensitivity. For this experiment, a manual factor of 2x was applied, though this factor heavily depends on environmental conditions, such as lighting and reflection.

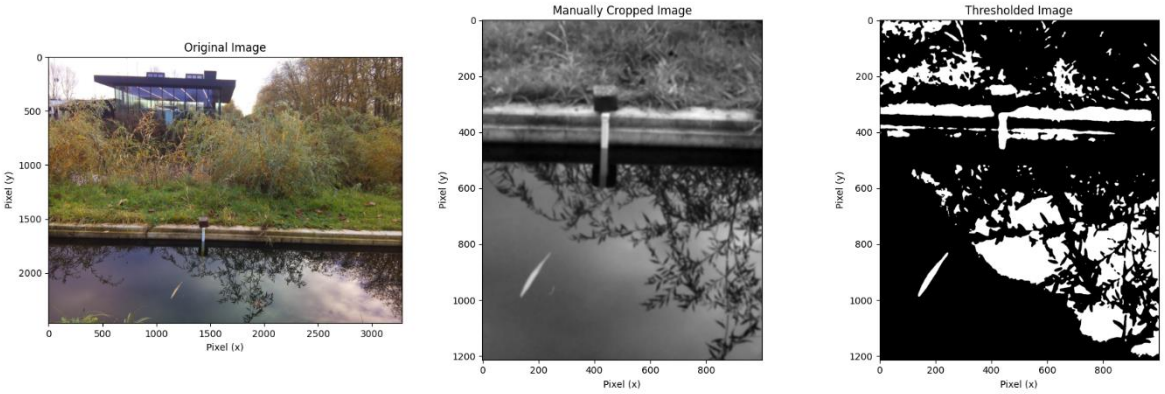


Figure 2: image 1 - threshold factor: 1.2

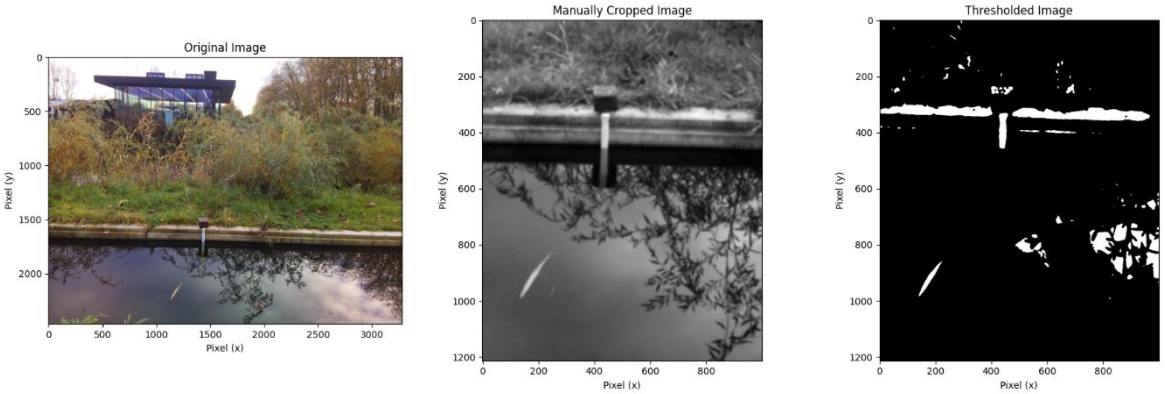


Figure 3: image 1: threshold factor: 1.5

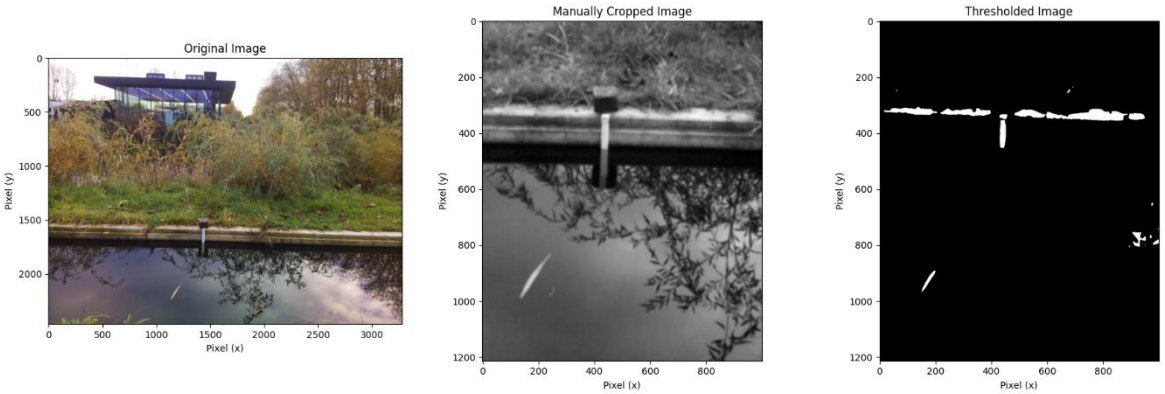


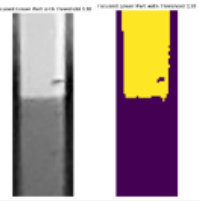
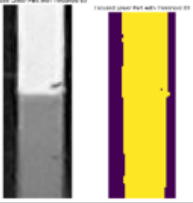
Figure 4: image 1: threshold factor: 2.0

Code adaptations made

To optimize performance of the vertical element algorithm, the entire image is rotated in the preprocessing section of the code so the staff gauge is aligned vertically. This increases the succesrate of finding the longest vertical element as the staff gauge in the binarized image.

To reduce noise like background and reflection on the water surface, the image can be cropped so only the staff gauge is visible. This greatly reduced the noise but introduced new problems. The correct crop for the staff gauge has to be determined manually for every experiment setup. Furthermore, may the camera orientation shift or move due to wind or other factors, the crop would be offset and this would negatively impact the water level detection capabilities.

The way the water level height is being determined by this method is making use of the difference in pixel intensity of the above water staff gauge versus the below water staff gauge. additional testing has been conducted to explore this mechanism. This concluded that the otsu method is not very consistent when detecting the correct waterlevel. The transparency of the water causes the algorithm to sometimes consider the underwater part of the staff gauge as object of interest while it should not.

#	Script output	description
1.		Waterline detected
2.		Waterline **not** detected

The script automaticcaly sumerises all interesting parameters for analysis in a separated file per cycle:

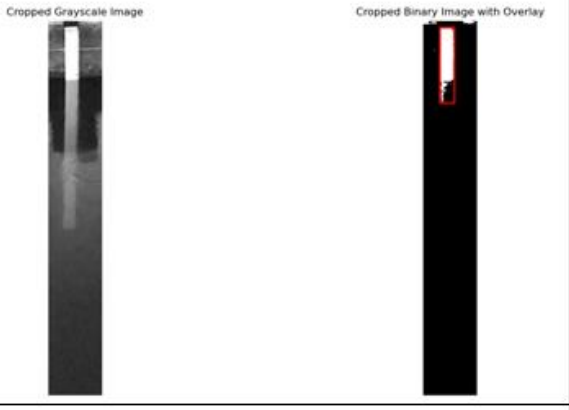
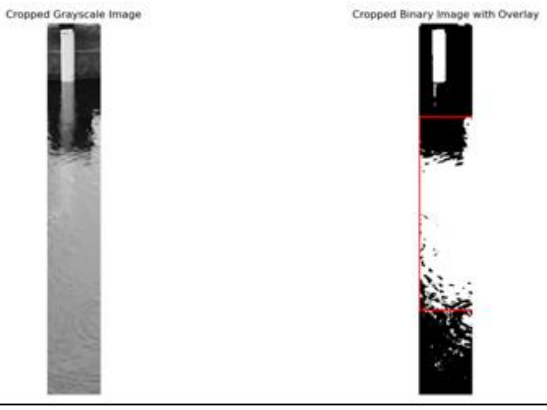
```
SUMMARY REPORT
====================
System: pc
Length of the longest vertical object: 680 pixels
Location of the largest vertical object: (Y: 48-728, X: 0-100)
Total script runtime: 5.10 seconds

RUNTIME DETAILS:
Image loading and preprocessing took 1.07 seconds
Thresholding took 0.03 seconds
Finding vertical component took 0.00 seconds
Length of the longest vertical object: 680 pixels
Location of the largest vertical object: (Y: 48-728, X: 0-100)
Best threshold before iteration: th=150.0
Best threshold found after 16 iterations: th= 75

COMPLETE DEBUG LOG:
====================
System: pc
Processing image: C:\Users\bjorn\Desktop\rpi_mirror\Timelapse_Images\Delft_20241119_154549.jpg
Step 1: Image loading and preprocessing completed.
Step 2: Image cropped.
Step 3: Thresholding completed with best threshold 150.0.
Step 4: Longest vertical component found with height 680 and width 100.
```

Hardware adaptation made

To reduce the glare and reflection on the water surface, the introduction of polarization filter has been tested. Preliminary test has been promising towards noise reduction. Using polarized sunglasses, a simple experiment has been conducted.

Polarized	Regular
	
	

To test the polarization filter, more extensive testing has been conducted in the gutter. Every 10 minutes, 2 images were taken, one with and one without polarization filer, while the gutter was drained to capture different water levels. 4/4 times the polarized setup performed better or ass good as the non polarized setup. Reflection on the water surface is greatly reduced.

Polarized	Regular
	
	

Eventhough this might be prommesing, more extensive testing with larger sample sizes and more variable environmental conditions should be performed to confirm this bias.

7.2.4 Notes on prototype 2 and feedback loop for next prototype

SCAMPER Heuristics

- Substitute: What materials or processes can be replaced?
  - The Otsu method is not ideal for detecting pixel intensity transitions. More options should be explored.
- Combine: What elements can be integrated?
  - A way of capturing images at night should be investigated, while lighting conditions are dark.
- Adapt: How can the concept be adjusted for improvement?
  - Instead of detecting the staff gauge, cropping can be utilized to filter out most of the background noise. A method of detecting intensity changes should be investigated.
- Modify: What aspects (e.g., size or shape) can be altered?
  - A more weatherproof setup should be developed to simplify field testing.
- Put to another use: Can the concept serve other purposes?
  -
- Eliminate: What features can be simplified or removed?
  - Otsu method can be optimized or substituted by a more effective method to detect intensity changes.
- Reverse: Can processes be reversed or re-sequenced?

The improved staff gauge and image capturing capabilities of the current prototype has enabled proper testing. However, the detection method (otsu method) does not seem to be verry effective when it comes down to detecting the water level. With the right manual intervention, it is a useful tool to detect the staff gauge within the natural environment, but this can be simplified by just cropping to the real world coordinates. Testing has made clear that the waterline can be detected making use of the intensity difference between above water staff gauge and below water staff gauge. A more effective way to detect this phenomenon should be explored.

The attachment of an polarized filter are promising. This should be incorporated in future prototype designs. However, a more robust testing setup should be developed to make it more realiable to perform testing outside without putting the electrical components at risk.

7.3 PROTOTYPE V3

7.3.1 Hardware V3

No changes

7.3.2 Algorithm V3

Water level detection algorithm

A alternative to the Otsu method has been developed with a focus on the intensity differences between the dry and wet staff gauge. The requirements for this method are:

- Staff gauge detection
- Water level deterction on the staff gauge as background;

To achieve this, a function is created that logs the mean intensity of a predefined box over the lengt of the staffgauge or image. Using this function, two boxes of x hight are slid down the gauge's image. The mean intensity of the pixels from the top box is compard to the bottom box for every position on the staff gauge. The boxes with the highest contraxt is labelled as the water level.

```python
def calculate_intensity_differences(image_np, box_height=10):
    """
    Calculate the intensity differences between two box_height-pixel-high boxes
    as we slide down the image. Returns the y-coordinate with the max difference.
    """
    height, width = image_np.shape[:2]
    intensity_differences = []

    # Iterate over all possible positions for the top of the first box
    for y in range(height - 2 * box_height):  # Ensure both boxes fit
        box1 = image_np[y : y + box_height, :]  # Top box
        box2 = image_np[y + box_height : y + 2 * box_height, :]  # Bottom box

        # Calculate mean intensity for each box
        mean_intensity_box1 = np.mean(box1)
        mean_intensity_box2 = np.mean(box2)

        # Calculate the absolute intensity difference
        intensity_difference = abs(mean_intensity_box1 - mean_intensity_box2)
        intensity_differences.append(intensity_difference)

    # Find the y-coordinate with the maximum intensity difference
    max_diff_index = np.argmax(intensity_differences)
    max_diff_y = max_diff_index + box_height  # This is the top of the first box
    waterline_y = max_diff_y + box_height // 2  # Shift to the middle between the two boxes
    return waterline_y, intensity_differences
```

This function logs the intensity differences of every box and determines the water line (y-coordinate of the image). the intensity differences can in turn be plotted over the height of the image which results in a graph where the peeks point out the highest pixel intensity difference. This function is adapted so that its unable to result in peek detection where its not probable, such as the border of the image (tests resulted often in a false peek at the border of the image). the second most probable waterline (second highest peek with a minimum distance from the highest peek) is also visualised. This gives insight into what the algorithm is detecting as intensity differences. Often the real waterline and the bottom of the gauge

(slightly visible under water) were labelled as primary or secondary waterline by the algorithm, performinig already much better than the Otsu method.

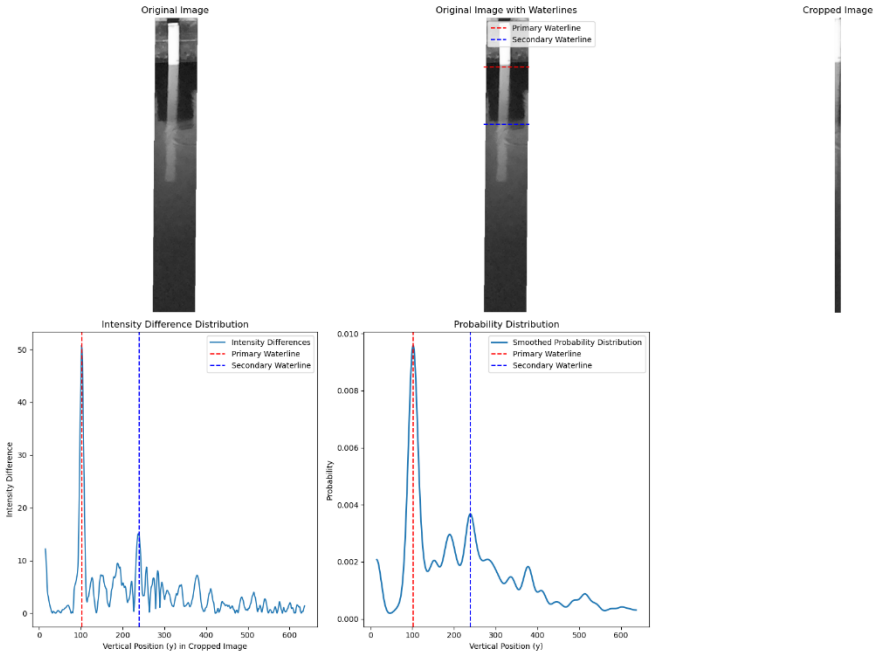
```python
def find_waterlines(intensity_differences, box_height=10, min_distance=50, edge_buffer=10):
    """
    Find the primary and secondary waterlines based on intensity differences,
    ensuring that they are separated by at least `min_distance` and are not
    within `edge_buffer` pixels from the edges.
    """
    # Convert edge_buffer and min_distance to indices in the intensity_differences array
    intensity_differences_filtered = np.copy(intensity_differences)
    intensity_differences_filtered[:edge_buffer] = -np.inf  # Exclude first edge_buffer pixels
    intensity_differences_filtered[-edge_buffer:] = -np.inf  # Exclude last edge_buffer pixels

    # Find the primary waterline (highest intensity difference)
    primary_index = np.argmax(intensity_differences_filtered)

    # Zero out a region around the primary waterline to find the secondary waterline
    low_limit = max(primary_index - min_distance, 0)
    high_limit = min(primary_index + min_distance, len(intensity_differences))
    intensity_differences_filtered[low_limit:high_limit] = -np.inf

    # Find the secondary waterline (next highest intensity difference)
    secondary_index = np.argmax(intensity_differences_filtered)

    # Return the indices of the two most probable waterlines
    if intensity_differences_filtered[secondary_index] == -np.inf:
        raise ValueError("Could not find a valid secondary waterline. Adjust constraints or check input.")

    return primary_index, secondary_index
```

The images are processed by rotating so that the general waterline aligns with the horizontal. This is nessecairy as the image is not transposed to the real world coordinates and the intensity boxes are iterated downwards over the image. furthermore, the image is cropped to the staff gauge as much as possible to filter out noise and converted to greyscale to visulise the intensity of every pixel.

```python
def process_image(image_path, save_path, system="pc"):
    try:
        # Check if file exists
        if not os.path.exists(image_path):
            print(f"ERROR: Image file does not exist at: {image_path}")
            return None

        # Create output directory if it does not exist
        if not os.path.exists(save_path):
            os.makedirs(save_path)

        # Step 1: Load and preprocess the image
        original_im = Image.open(image_path)
        angle = -0.5
        original_im = original_im.rotate(angle, resample=Image.BICUBIC, expand=True)
        original_np = np.array(original_im)  # Keep the original image in color

        # Step 2: Crop the image
        crop_box = (50, 15, original_np.shape[1] - 45, original_np.shape[0] - 45)
        cropped_im = original_im.crop(crop_box)
        cropped_np = np.array(cropped_im)
        crop_offset_x, crop_offset_y = crop_box[0], crop_box[1]

        # Step 3: Convert the cropped image to grayscale for intensity analysis
        cropped_grayscale_np = np.mean(cropped_np, axis=2) if len(cropped_np.shape) == 3 else cropped_np

        # Step 4: Calculate intensity differences on the cropped grayscale image
        box_height = 10  # Must be consistent with calculate_intensity_differences
        waterline_y_in_cropped, intensity_differences = calculate_intensity_differences(cropped_grayscale_np, box_height=box_height)
```

```
# Step 5: Find primary and secondary waterlines
try:
    primary_index, secondary_index = find_waterlines(intensity_differences, box_height=box_height, min_distance=50)
    # Convert indices to y-coordinates in the cropped image
    waterline_y_cropped = primary_index + box_height + box_height // 2
    second_waterline_y_cropped = secondary_index + box_height + box_height // 2
    print(f"Detected waterlines at y={waterline_y_cropped} (primary) and y={second_waterline_y_cropped} (secondary) for {os.pat
except ValueError as e:
    print(f"Error finding waterlines: {e}")
    return None

# Map waterline positions to the original image coordinates
waterline_y_original = map_cropped_to_original_y(waterline_y_cropped, crop_offset_y)
second_waterline_y_original = map_cropped_to_original_y(second_waterline_y_cropped, crop_offset_y)

# Step 6: Create a 2x2 combined plot with the specified format
fig, axes = plt.subplots(2, 2, figsize=(12, 12))
```

Finally the coordinates are transformed from the croppedimage to the original image to enable visualisation plots.



Continuous image subtraction algorithm

(Liu, W.-C. (2024). Evaluation of deep learning computer vision for water level measurements in rivers. Environmental Monitoring and Assessment; paragraph 2.6)

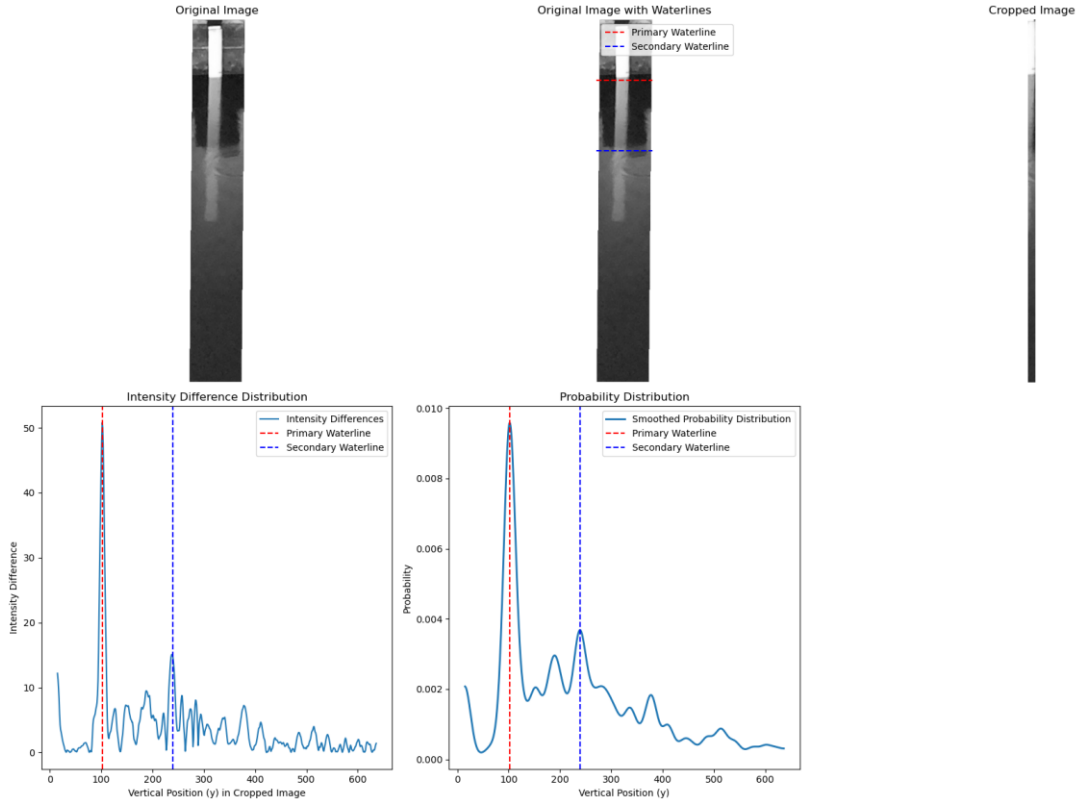
Code to replicate this method is created but the results differ from the work conducted in the article. The article uses time interval of 0.05s for consecutive images. Test images made to test own code are made with an interval of several seconds. This might be the reason that the result is not up to standards.

```
def process_images(img1_path, img2_path):
    """Process two consecutive images to highlight water areas."""
    # Load images as grayscale using Pillow
    img1 = Image.open(img1_path).convert("L")
    img2 = Image.open(img2_path).convert("L")
    print(os.path.basename(img1_path))
    print(os.path.basename(img2_path))

    # Convert images to NumPy arrays
    img1_array = np.array(img1, dtype=np.float32)
    img2_array = np.array(img2, dtype=np.float32)

    # Subtract the images
    diff = np.abs(img2_array - img1_array)

    print(diff)

    # Enhance the difference: Dilate and square pixel values
    enhanced = np.clip(diff ** 2, 0, 255)
    print(enhanced)

    # Binarize the image
    binarized = (enhanced > threshold_value).astype(np.uint8) * 255  # 0 or 255
    print(binarized)
    # Convert the result back to an image
    binarized_image = Image.fromarray(binarized, mode="L")
    return binarized_image

# Iterate through image pairs
for i in range(len(image_files) - 1):
    img1_path = os.path.join(image_dir, image_files[i])
    img2_path = os.path.join(image_dir, image_files[i + 1])

    # Process images
    output_image = process_images(img1_path, img2_path)

    # Save the processed image
    output_image.save(os.path.join(output_dir, f"processed_{i+1}.png"))

print(f"Processed images saved to {output_dir}.")
```

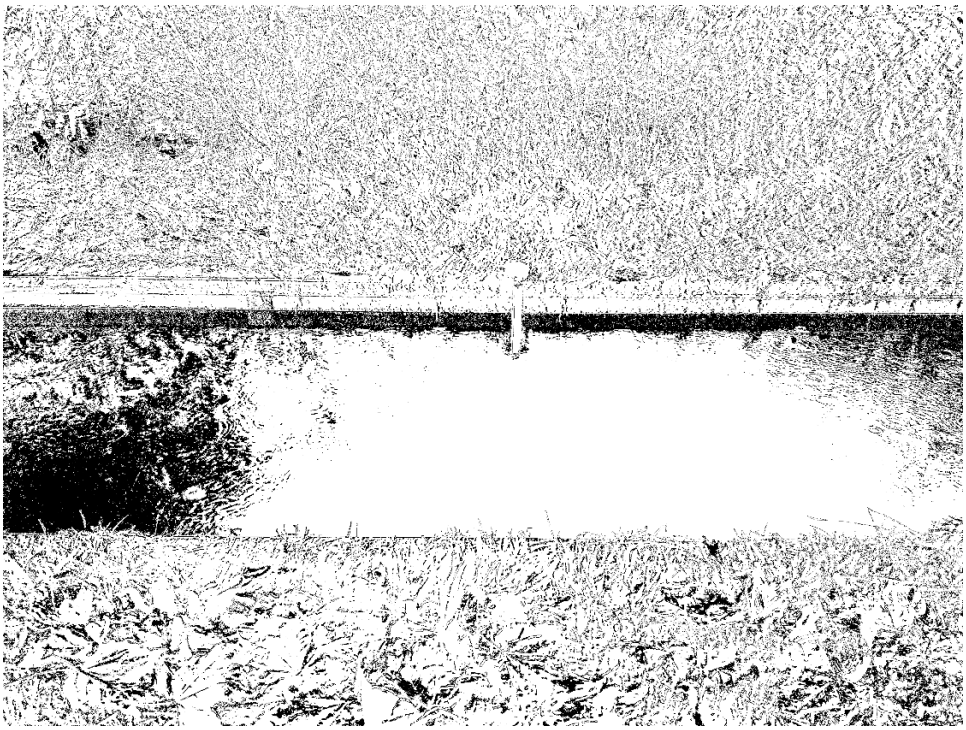
7.3.3 Experiment V3

Water level detection algorithm

The intensity box method works much better compared to the Otsu method. Cropping to the staff gauge greatly reduces background noise but contributes to more work setting up the system. The crop dimentsions have to be determined manually (for now) and precousions have to be made so that the camera orientation does not change (by winds or other environmental impacts).

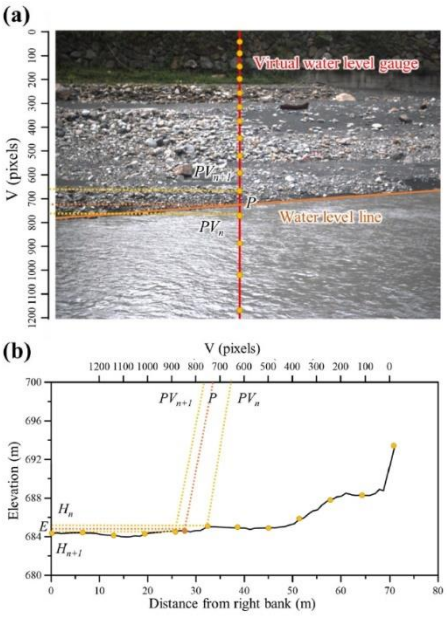


CIS



The method takes the difference between pixel intensity and dilates the difference. After binarization it reconstructs the images showing difference between dynamic and static pixels. The goal is to locate the static bank and dynamic water, but a lot of factors play a role in this workflow. Natural surroundings like vegetation or water puddles on the bank result in noise. Also camera orientation has to be very secure to make sure each pixel between the frames are compared with the same pixel of the previous orientation, linked to the same coordiantes in the real world.

- Interval of consecutive images needs to be smaller to decrease variables other then water dynamics such as moving grass, change in illumination / shading or glare.
- Method works best if riverbank is made up out of stone/earth instead of moving elements like vegetation
- Method works best when water velocity is high



- Method can be paired with digital water level gauge
- Noise can be reduces using an rolling average. **Updated Processing Pipeline (method not tested yet) :**
    - Compute a rolling average of N consecutive frames to smooth out transient changes like grass motion.
    - Subtract the current frame from the rolling average.
    - Enhance and binarize the resulting image to highlight water areas.

7.3.4 Notes on prototype 3 and feedback loop for next prototype

The water level detection method seems promising. This method can be integrated in the device workflow, combining timelapse imaging with automatic processing. The CIS method is probably not best choice for this setup. The raspberry pi is unable to consistently take short interval images [0.05s]. as of now, the device is a raspberry pi connected with an camera module, held together on a cartwood box and some ducttape. It would help to create a more weather proof casing so test are les risky. Problems with damaged camera modules also point to better protection for the sensitive parts of the system. Finaly, different filters can be tested to increase the ability to detect the waterline based on intensity differences. A high quality polarization filter can be utilized and a ir bypass filter can be tested for night measurements. Night measurements also point towards the need of IR illumination.

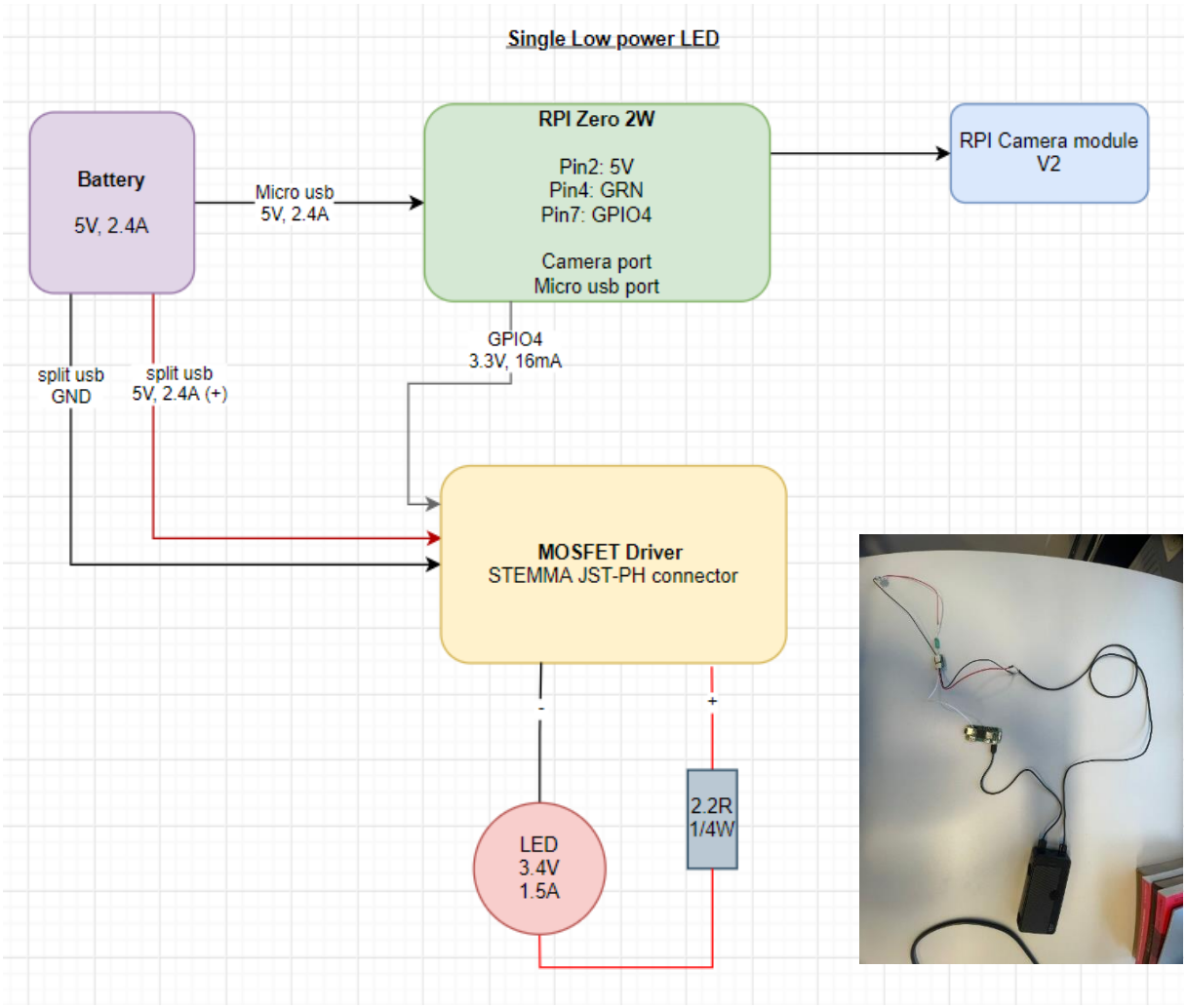
SCAMPER Heuristics

- Substitute: What materials or processes can be replaced?
  - Device casing has to be more practical for experimenting and somewhat waterproof to reduce risks during outside testing.
- Combine: What elements can be integrated?
  - All components of the device should be integrated in a simplistic casing forming the first practical prototype.
- Adapt: How can the concept be adjusted for improvement?
  - Filters such as polarization filters and IR bypass filters can be introduced to the system, together with an IR illumination method.
- Modify: What aspects (e.g., size or shape) can be altered?
  - To connect every component, a breadboard can be used to help with closing the circuit and making sure every component has a safe current and voltage. A USB hub can be added to enable modifications in the script during field tests because this enables the use of a mouse and keyboard.
- Put to another use: Can the concept serve other purposes?
  -
- Eliminate: What features can be simplified or removed?
  - The otsu method can be eliminated, together with the very preliminary casing of the system which mostly consisted of cartwood box and ducttape and elastic bands.
- Reverse: Can processes be reversed or re-sequenced?

7.4 PROTOTYPE V4

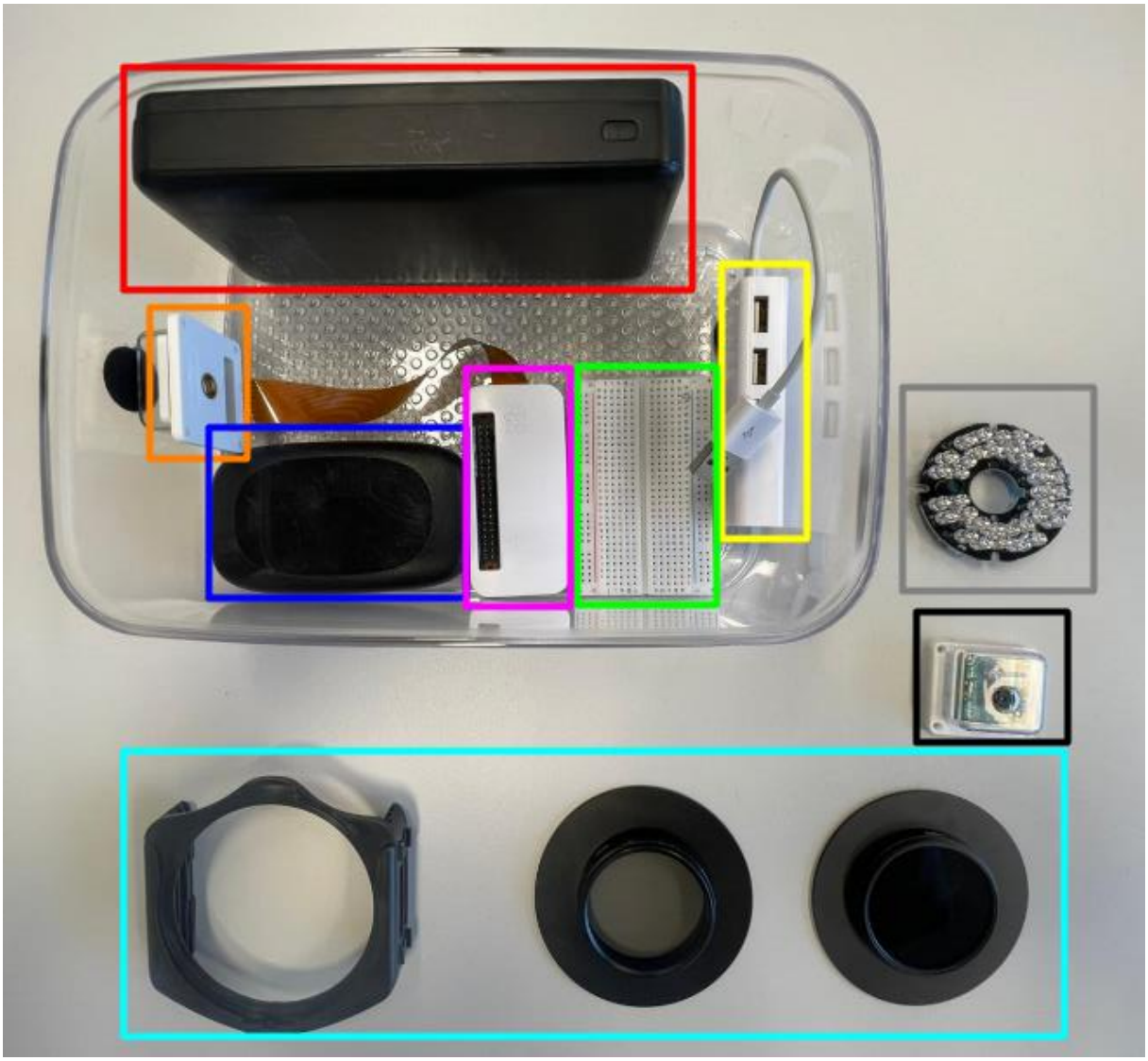
7.4.1 Hardware V4

To power a IR led, a mosfet driver is used as transistor. This is required as the raspberry pi gpio can only output a low current signal, and not power the led by itself. The led needs to be powered separately. To save power, the led only turns on at night and during the image capturing. So realistically its only turned on for 1 or 2 seconds every cycle (cycle being every 10 minutes usually).



The led is protected by a resistor (1.2Ohm, 5W), and powered separately form the powerbank. The powerbank has a shared capacity but is able to output 2.4A 5V from each output.

Polarization filter and an 850nm bypass filter has been acquired together with a filter holder used in photograpy. These filters are relatively cheap and easy to acquire. A 12V led ring has been purchased from Ali express, able to output 0.3A * 12V = +- 4W. however this cant currently be powered as the current power supply is only capable of outputting 5V.



- Box
  - Temporary prototype
- **ISY Powerbank**
  - 20.000 mAh
  - 2x usb output [5V, 2.4A]
- **Rpi NoIR camera module V2**
  - 8 MP
  - 3280 x 2464 pixel static
  - 1080p30 video
  - No IR filter
- **Tp-Link M7200 portable router**
  - 2.000 mAh (charge by battery)
- **Rpi Zero 2W**
  - 5V input
  - 3.3v, 5V output
  - 512MB RAM

- **Breadboard**
- Usb / ethernet port
  - For connecting mouse and keyboard to RPI if VNC fails
- **IR LED ring**
  - 12 V (not suitable for this setup)
- **Rpi camera module V2**
  - Optional for daytime image capturing
  - 8 MP
  - 3280 x 2464 pixel static
  - 1080p30 video
- **Filter holder + Polarized filter + 850nm pass trough filter**
  - Polarized filter for reducing noise from glare on water surface
  - IR pass trough filter for reducing noise from light pollution at night

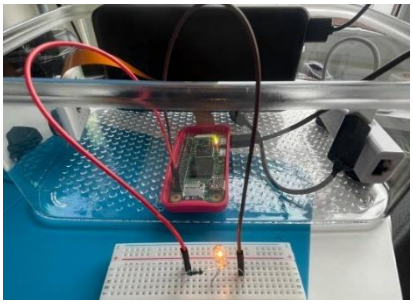
A food container is used to act as a temporary casing for the project. Currently the focus is on daylight capturing and night time capturing, where during the night, the IR led has to illuminate the project area to be able to detect the water level.

Daytime Capturing
The current setup is capable of capturing bursts of images at predefined intervals during the daytime. The first image of each burst is stored in the local memory and processed by the waterline detection algorithm. The results of the algorithm are also stored locally. Subsequently, the device uploads the raw image and the detected water level to the OpenRivenCam server. Finally, the program enters sleep mode until the next predefined cycle (every 10min). A high-quality polarization lens is available to reduce noise caused by glare on the water surface.

Night-Time Capturing
Currently, the prototype cannot process images at night. While a NoIR camera is operational, the IR LED required to illuminate the project location is not yet functional. However, a lens to filter out light waves shorter than 850 nm is already available to minimize noise from other light sources.

Project Difficulties at This Point
Desk research suggested using a 12V IR Led ring to illuminate the project area at night, considering that the object of interest is outdoors and approximately 5-10 meters from the camera. However, this would require an additional power source, as the current one only outputs 5V. Also switching to a 12V setup would contradict the criteria of being low power.

Preliminary experiments with simple LEDs powered via the GPIO pins of the Raspberry Pi encountered challenges. The GPIO pins are designed to transmit utility signals, such as turning LEDs on or off, but are not ideal for powering features. However, powering an LED using the constant output pin did work.



What's Next for the Coming Week?
To enable night vision, we plan to obtain a 5V IR led ring to illuminate the project area at night. This IR Led ring will be powered directly by the Raspberry Pi, eliminating the need for an additional power source. A MOSFET driver will be used to control the Led ring, ensuring it only turns on during the image-capturing bursts (2 seconds every 10 minutes) to reduce power consumption and minimize disturbance to wildlife.

To determine the required **W/sr (watts per steradian)** for your **3.3V, 1.5A IR LED** to illuminate a water body at a distance of **5 meters** with a beam angle of **50°**.

Required radiant intensity

$$\boldsymbol{I} = \boldsymbol{E} \cdot \boldsymbol{d}^2$$

I = radiant intensity (power per steradian needed to achieve sesired irradiance at specified distance)

E = minimal detectable irradiance (not specified by rpi, estimate at least 0.1 [W/m2]

D = distance = +- 5 [m]

$$I = 0.1 \cdot 5^2 = 2.5\, W/sr$$

Required power

$$\boldsymbol{P = I \cdot \Omega}$$

P = total power [W]

Ohm = Solid angle [R]

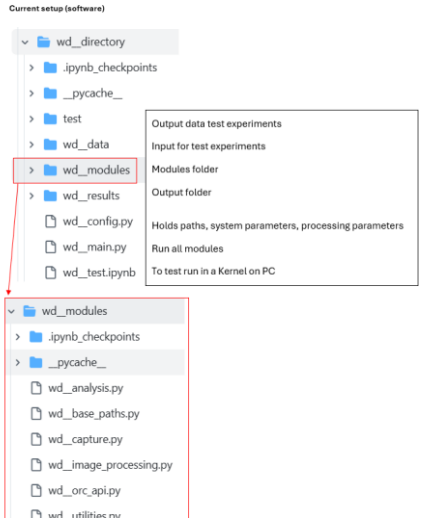
$$\boldsymbol{\Omega = 2\pi(1 - cos(\theta / 2))}$$

$$\Omega = 2\pi(1 - cos(50° / 2))$$

$$P = 2.5 \cdot 0.588 = 1.47\ W$$

7.4.2 Algorithm V4

Organised code and enhanced user friendliness of script. no functionality changes were made to the code.
The algorithm is restructured using different .py files and organised in folders. This enables better work dynamics and code can be more easily adapted.



7.4.3 Experiment V4

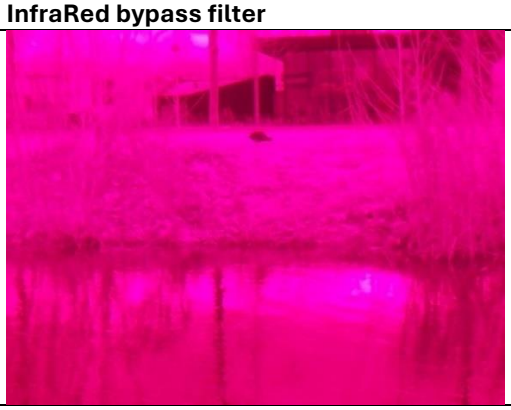
IR light testing

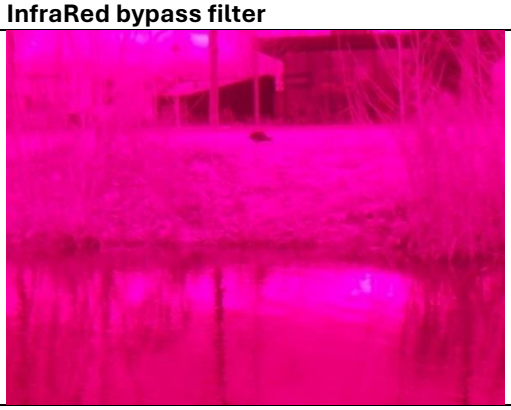
The IR Led seems to be working as it is able to illuminate a small dark room. However, the intensity of the LED seems too low to illuminate a waterbody in a natural environment. For now, one LED is connected in the system. The powerbank that is used as a power source at the moment has two outputs both capable of emitting 2.4A of current and 5 Volts. This makes it non logical to connect the led's in series, as the current would be shared across the series and would not achieve the leds maximum output (1.5 A), as the 2.4A would be shared between them (max 1.2A). However, a test experiment could be setup that tests one led on max capacity (1.5A) and two leds on reduced capacity (1.2A). A beam casing that focuses the IR Light on the target area could be interesting for testing as well and could be combined with the function of keeping the filters dry.

After experimenting it seems that the single led only draws 1A on ~3 volts, which mean the led only emits with 3Watts. According to preliminary estimates, this should be enough but it is not close to illuminating the project area enough. For night measurements, upscaling of the IR led is necessary.

Filter testing

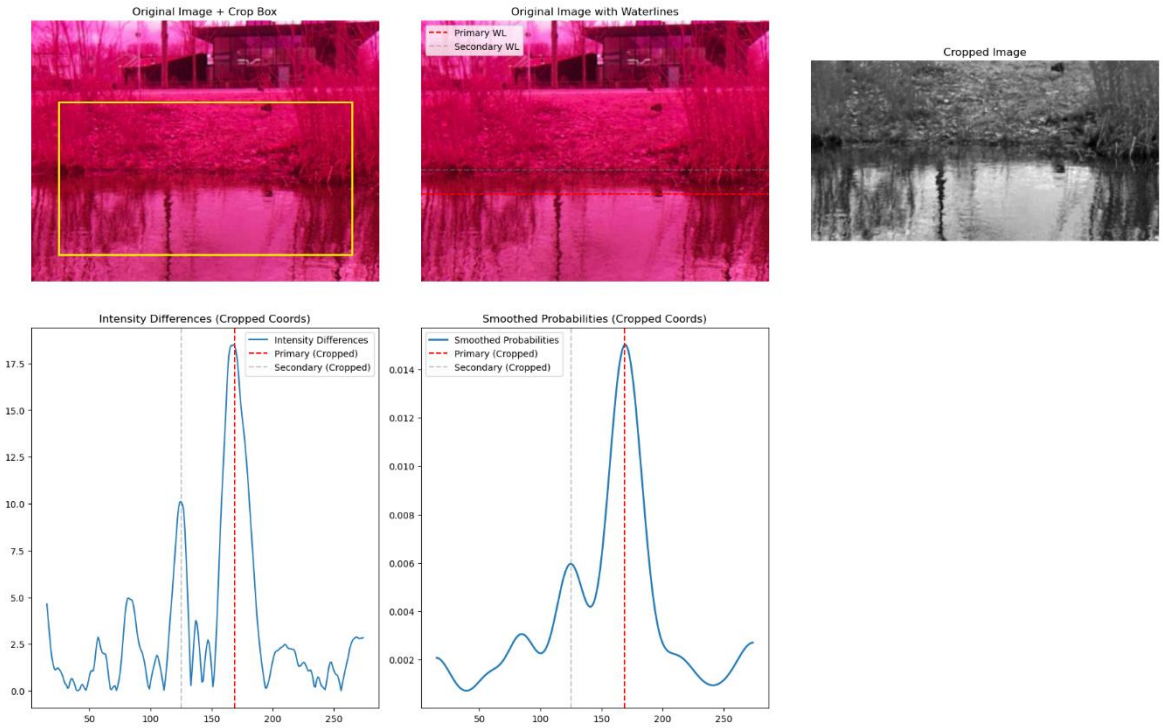
Different combinations of filters have been experimented with. The experiment took place in the afternoon about 14:00 while it was cloudy and dark. The polarization filter should reduce the noise from reflections on the water surface. The IR bypass filter should only pass +850nm wavelength. This should stabalize the image as it only receives the light that the IR Light is emitting and reduce noise from other light sources at night time. During the experiment a NoIR camera module was utilized.

No filter	**InfraRed bypass filter**
	
The contrast of the image utilizing no filters is the highest of all the experiment setups	The contrast decreases when using the IR filter during daylight
IR + Polarization	**Pol**
	

The contrast decreases from the IR filter, the polarization filter smooths out the white reflection noise a little	The white reflections on the grass and water surface are smoothed out but the contrast is decreased compared to no filter.
No filter	**InfraRed bypass filter**
	
The contrast of the image utilizing no filters is the highest of all the experiment setups	The contrast decreases when using the IR filter during daylight
IR + Polarization	**Pol**
	
The contrast decreases from the IR filter, the polarization filter smooths out the white reflection noise a little	The white reflections on the grass and water surface are smoothed out but the contrast is decreased compared to no filter.

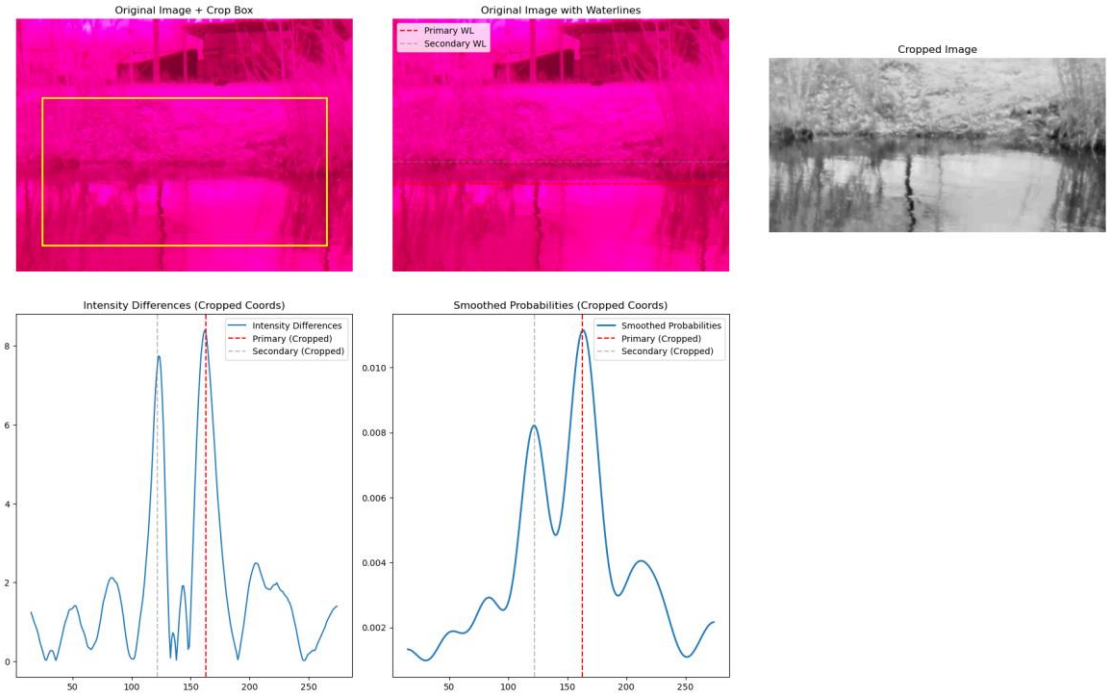
None

Secondary waterline is mostly correctly placed on real waterline. But the contrast of the reflection /shadow of the slope have higher contrast. The contrast peeks on 10.0 & 17.5.



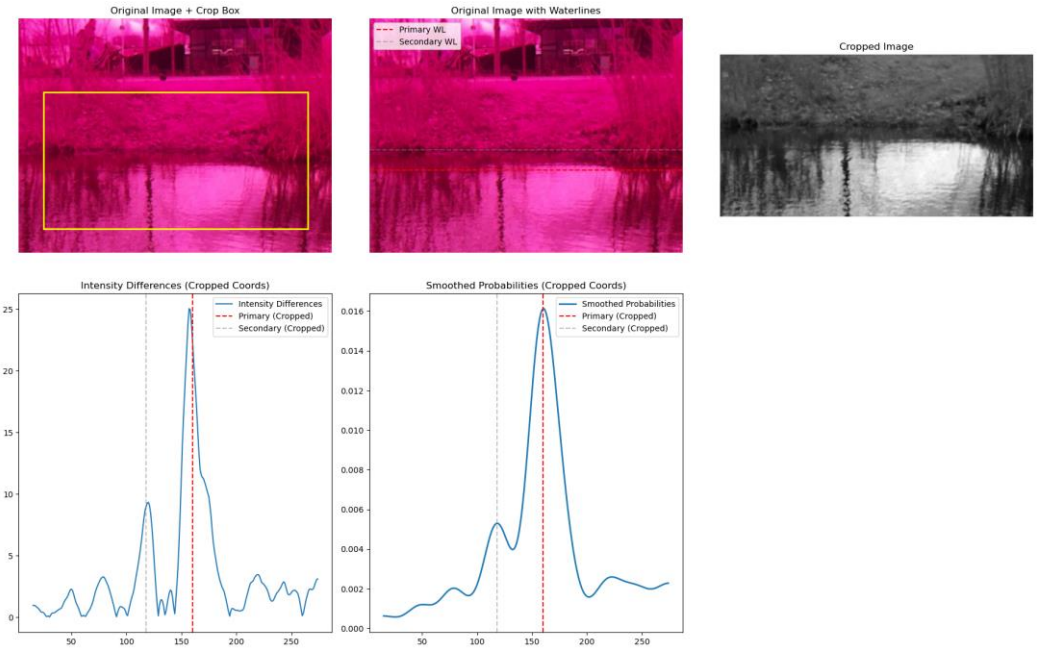
IR

The IR filter image catches the same contrast peeks, however the contrast seems lower overall. The contrast peeks only at +- 8.0.



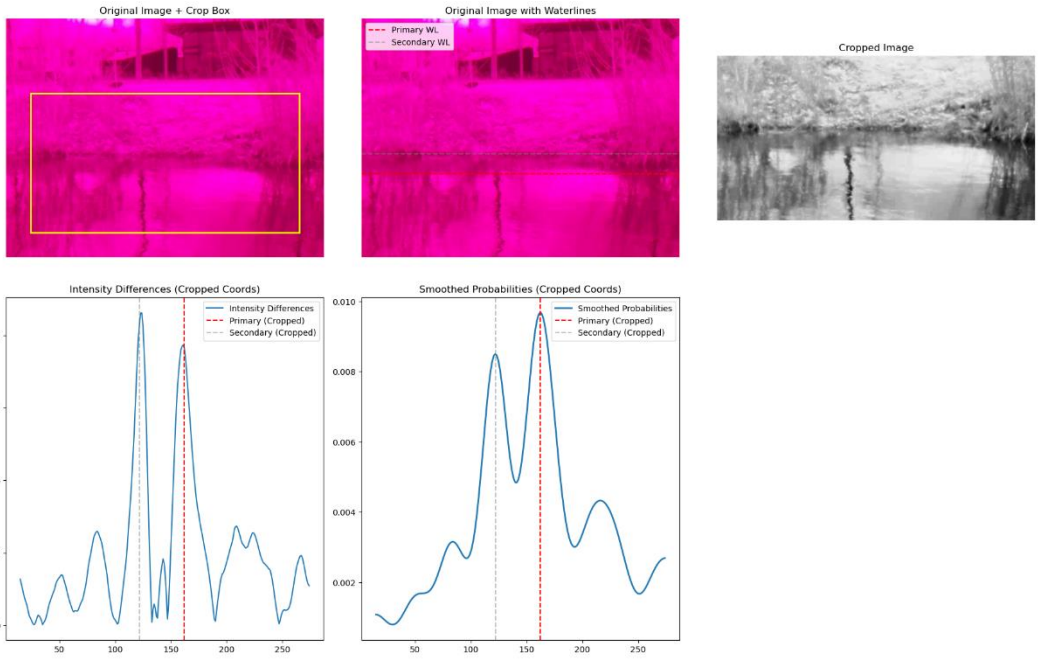
Polarized

The same results occur using the polarized filter, however the contrast is higher than the original image peeking at 10.0 and 25.0.



Polarized & InfraRed

Both filters result in the lowest contrast and the primary and secondary peeks are very similar in intensity. Both peek around 8.

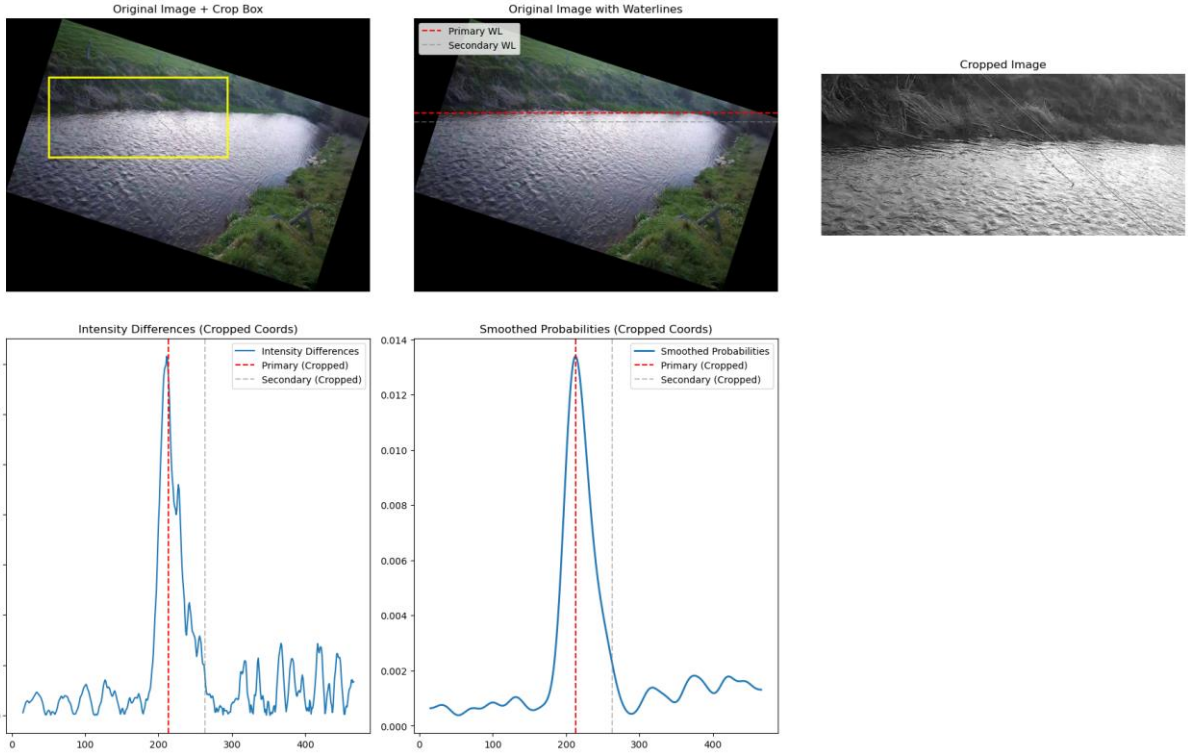


Overall, the polarized images have the highest contrast over all images. However, more testing should be done to test the consistency of this behaviour on different times and illumination conditions. Also, a very wide intensity comparison box is taken resulting in a diffusive contrast. Taking a narrower comparison box should result in a more distinctive intensity difference.

7.4.4 Test data hommerich

A dataset of 11.146 images taken of a stream in Hommerich, Limburg is provided for analysis of the algorithm. The images were taken from December 2022 until April 2023 every hour. At night images were taking using a flashlight.

A random selection of 100 images (taken during daylight without flashlight) were selected and run trough the current algorithm. Preprocessing steps that were taken were rotation of -18 degrees, so the waterline mostly aligned with the horizontal. This way, no further transformations had to be made to run the algorithm of water level detection.



In hindsight, a too wide of an intensity comparison box is selected to determine the water level. A stretch of +- 5 meter is cropped of the original image to run trough the steps of the algorithm, which smooths out the pixel intensity of the comparison boxes by a lot. Next time, a narrower slice should be selected in the preprocessing process.

The statement that should guarantee the secondary waterline to be at least 50 pixels away from the primary waterline caused that the secondary waterline was not mapped to a peek but just the highest value at least 50 pixels away from the primary waterline, this was fixed later.

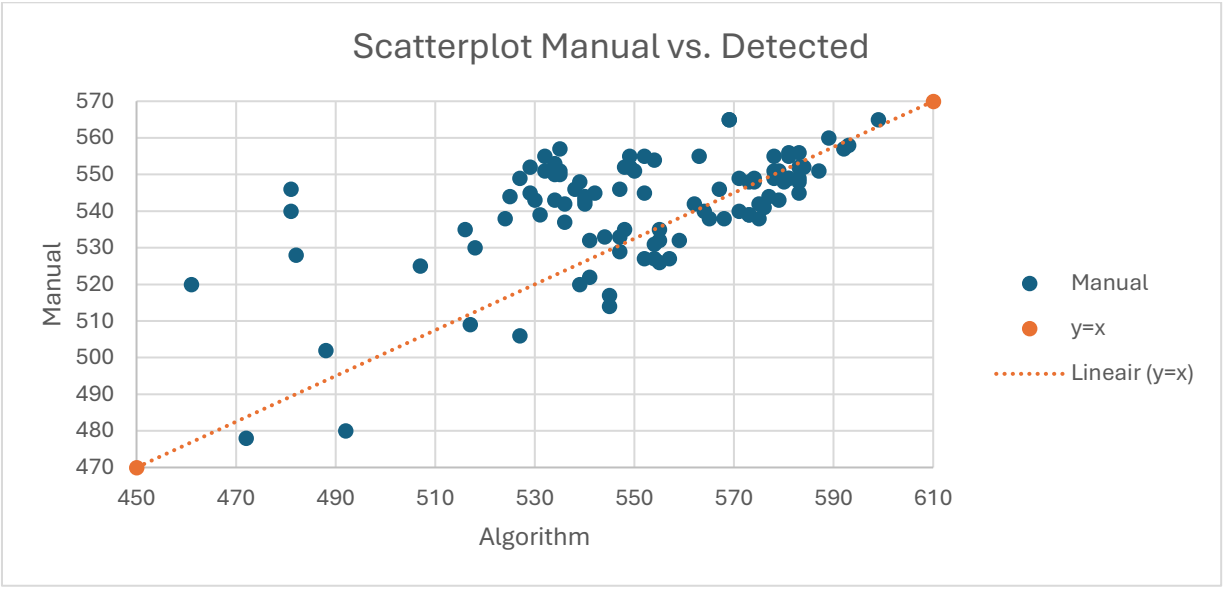
Using a script, an effort was made to map the real waterline to compare the algorithm outputs to. This was accomplished by manually clicking two points in the image. The mean y coordinate of this line is labelled as the real water level (for now). This pixel y coordinate can be compard to the detected water

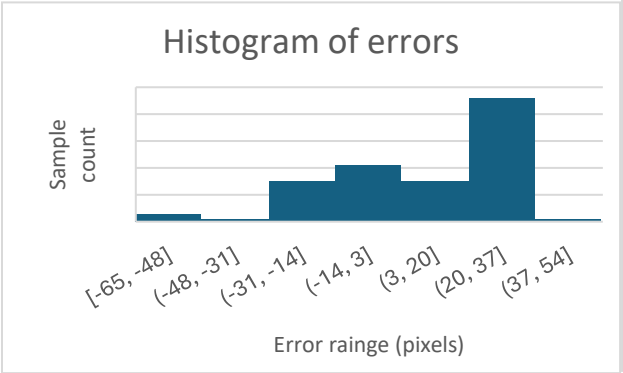
level from the algorithm. After visual analysis, this way of determining the real water level is way to inconsistent an inaccurate to realy derive any conclusions. However, the metohod of comparing the detected waterline with the 'real' water level is developed.

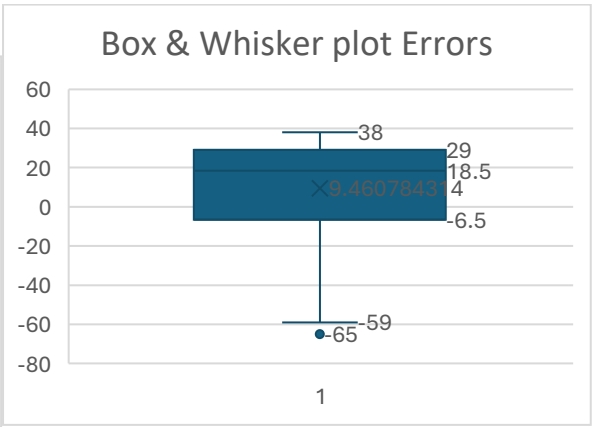
image	Algorithm	Manual	Error
schedule_20220831_045046.jpg	569	565	4
schedule_20220831_062436.jpg	549	552	-3
schedule_20220831_045046.jpg	569	565	4
schedule_20220831_062436.jpg	549	552	-3
schedule_20220831_102356.jpg	593	558	35
schedule_20220831_102646.jpg	589	560	29
schedule_20221220T085601.jpg	599	565	34
schedule_20221228T074600.jpg	584	552	32
schedule_20221228T123100.jpg	587	551	36
schedule_20221229T080100.jpg	548	552	-4
schedule_20221229T124600.jpg	583	552	31
schedule_20221230T080101.jpg	592	557	35
schedule_20230105T123101.jpg	539	548	-9
schedule_20230106T104601.jpg	540	543	-3

...

MAE	RMSE	Bias
21.38235	3.075839	9.460784



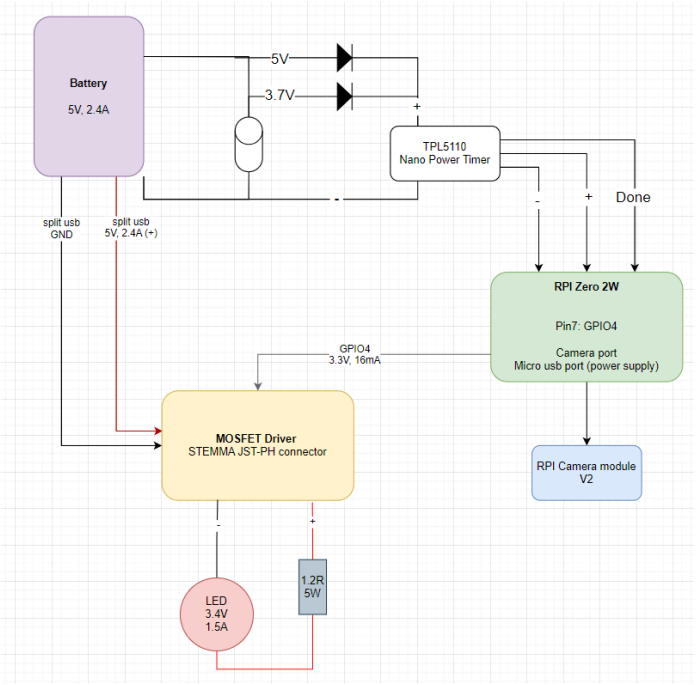




7.4.5 Power consumption experiment

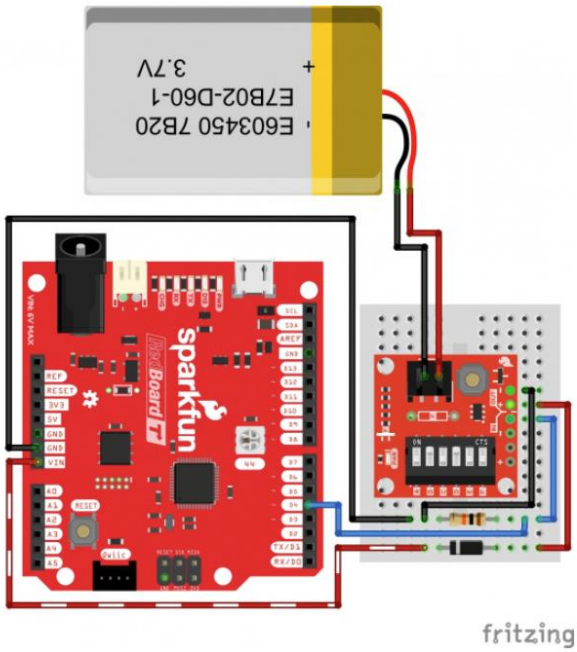
The TPL5110 nano power timer is a board that controls power consumption of a system. It runs at +- 85 nA when in standby mode and controls the delay of which the device can stay in standby mode. When it gets a done signal, it turns power from the powersource off for x amount of time. X is regulated by a series of resistors.

The first tryout was to connect the main power source and an separate power source for the timer. This however resulted in instable voltagges to the pi (between 4.2 and 5.0). When the pi goes below 4.7V it has difficulties booting up. The boot up process is repeated indefinitely. See figure below for board setup. The reason for this instability is the use of diodes, that regulate the direction of the current. The 5V current cant flow in the opposite direction of the current from the battery, hence the diode is used. However, the diode has a forward voltage drop of +- 0.8V, which probably caused the instability.



The second try was to connect the nanotimer directly between the main power source and the pi. However, also this setup was very inconsistent. While introducing a power measurement tool (multi tool or usb power measurement tool), the nano timer shuts down and therefore shuts down the boot up of the pi. The Voltage is stable at the input pins of the timer, but gets unstable after it passes the internal mosfet of the timer board and get distributed trough the jumper wires.

The cause of the instability within the timer could be the result of short circuiting of some sort because during testing, some smoke and burning smell was detected. 2 new times are bought to test this setup and the voltage instability.



findings

- The powerbank has a powersaving mode that shuts down the powerbank after low power demand (80nA is low power) after about a minute. This makes it so that a boot up after standby is impossible
    - A workaround is connecting an usb hub in between the power source and the nano timer. The little led in the hub keeps the battery on while in resting mode. This takes power but for now it's a good way of testing the nano timer.
- When the delay is set < the boot up time, the nanotimer shuts itself down before getting the done signal. It keeps booting in cycles but never finishes.
- N14007 diode has Forward Voltage drop of 0.8V, which causes instability in power supply which results in insucessfull boot up of pi.
- The pi internal timer starts 30 seconds after turning on the power supply. When a rest time of 120 seconds is programmed before sending the done signal, it wil take +-150 secons to shut down after turning the power on.

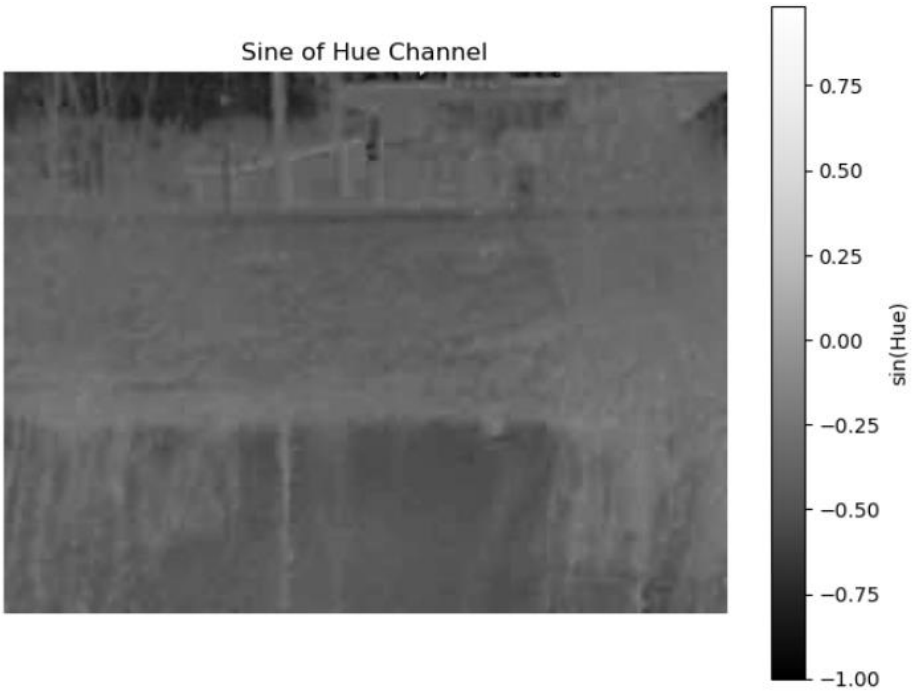
Nano power timer seem to be working if only using a simple script that produces a print statement and a resting state before forwarding the done signal.

7.4.6 Colour schemes experiment

NOIR camera

This is the original image displayed in RGB format. Each pixel consists of red, green, and blue values that together define its color.

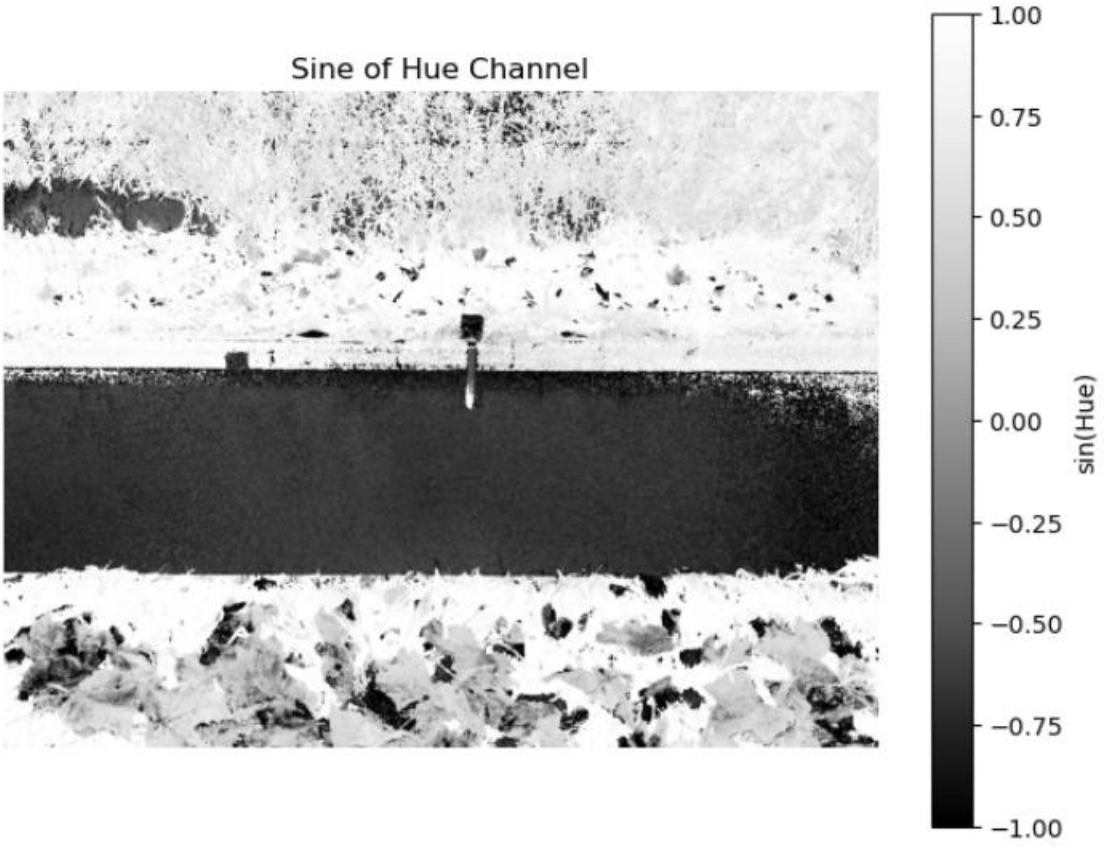
The Value (V) channel represents the brightness of each pixel. Higher values correspond to brighter regions, while lower values indicate darker areas.

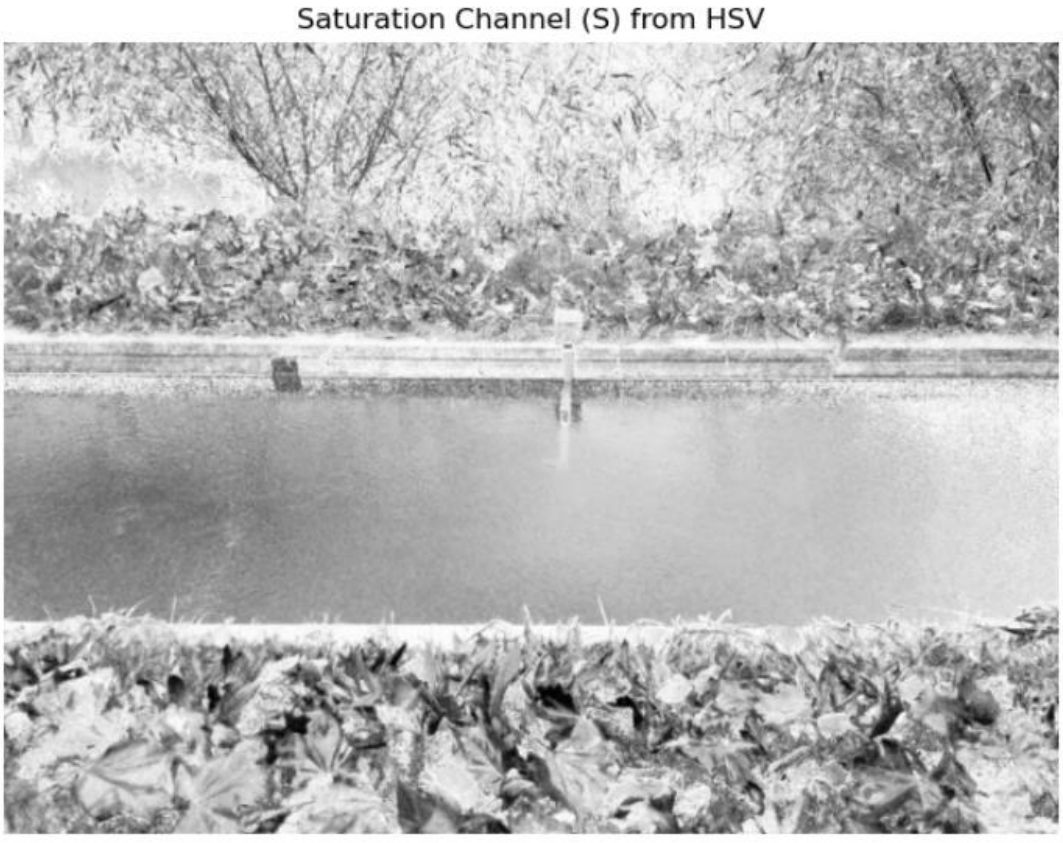


The Hue (H) channel represents color tones as angles in a circular color space. By applying a sine transformation, we can visualize periodic variations in hue, making color transitions clearer.

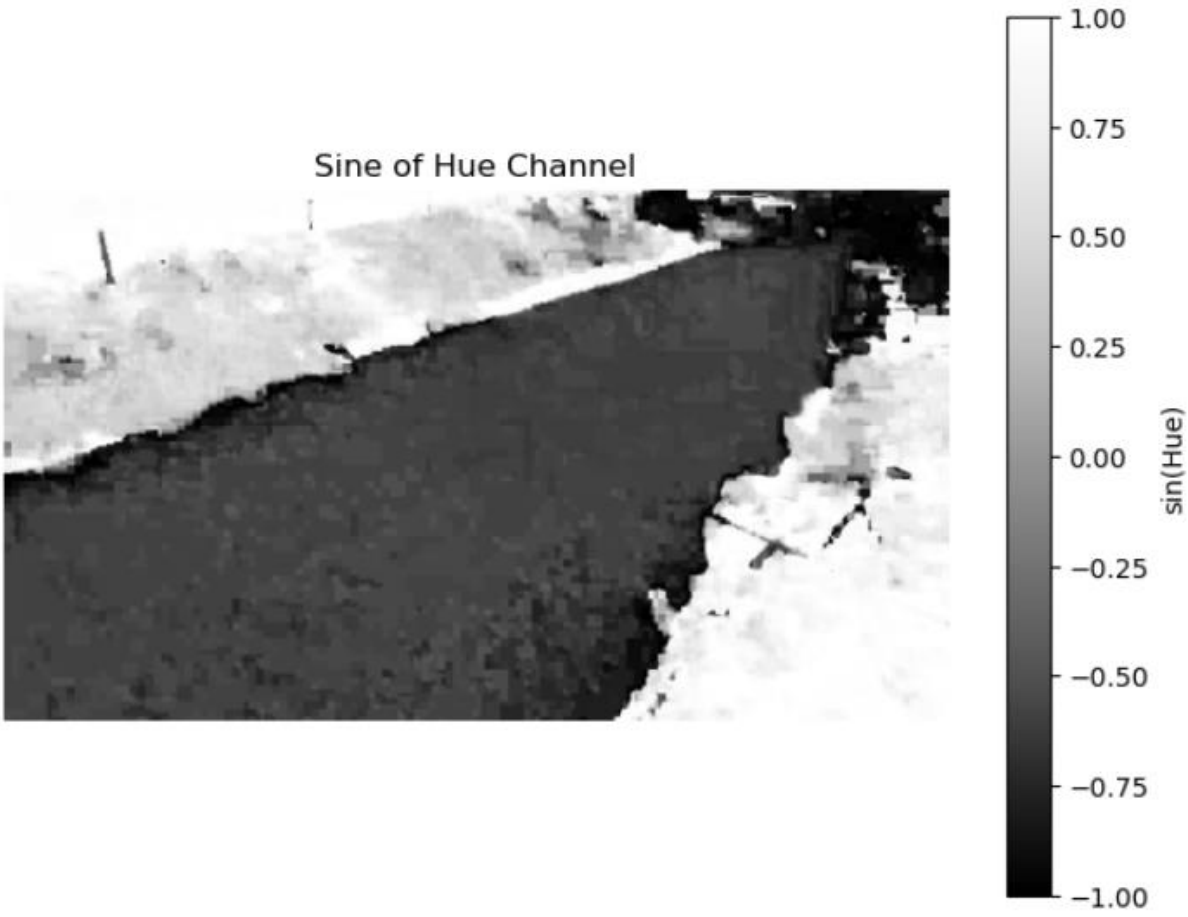
The Saturation (S) channel indicates color intensity. A higher saturation value means a more vivid color, while a lower value results in a duller, grayer color.

Normal camera


Sine of Hue Channel
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
sin(Hue)


Saturation Channel (S) from HSV

Hommerich dataset


Sine of Hue Channel
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
sin(Hue)


Saturation Channel (S) from HSV

7.4.7 Testing differen color scemes on hommerich dataset

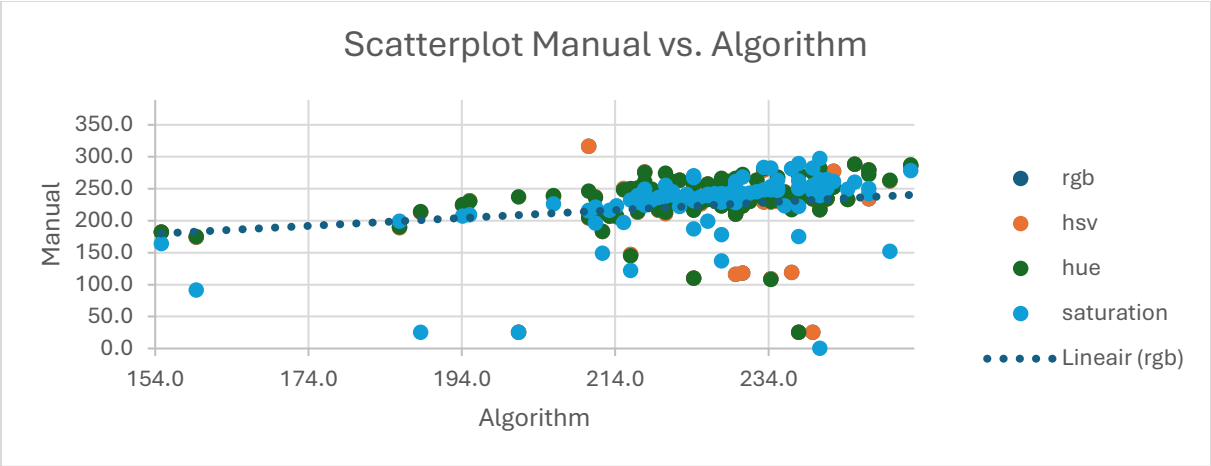
The algorithm and manual water level labelling process is run to analyse the consistency of different colour settings as input for the waterlevel detection algorithm. From first estimation, HUE seems to have the highest contrast between land and water, as green and blue has a higher contrast compared to pixel intensity. However, after running the script on a sample size of 100 randomly selected images from Hommerich dataset (during daytime), there seems to be no clear difference between the accuracy of the different color settings.

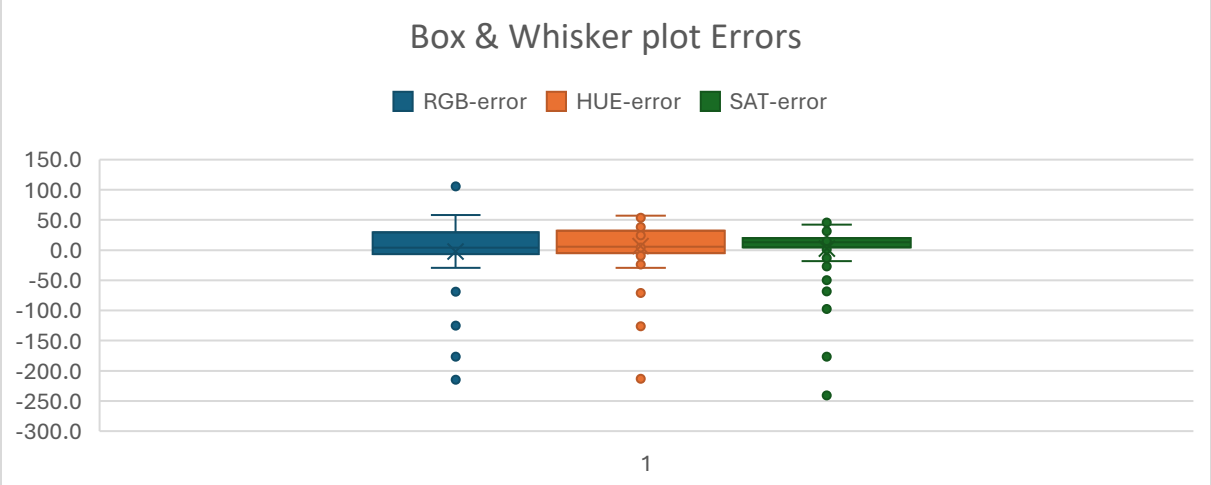
Two tests were conducted. Firstly, the raw images were cropped to a small portion of the entire frame to save computation demand. A manual labelling algorithm was run to determine the 'real' water level visually. After that the algorithm was run on the cropped dataset. These results were compared.

	HSV-error	HUE-error	SAT-error
MAE	28.3174	22.8026	27.1788
RMSE	5.3	4.8	5.2
Bias	-2.3	6.8	2.3

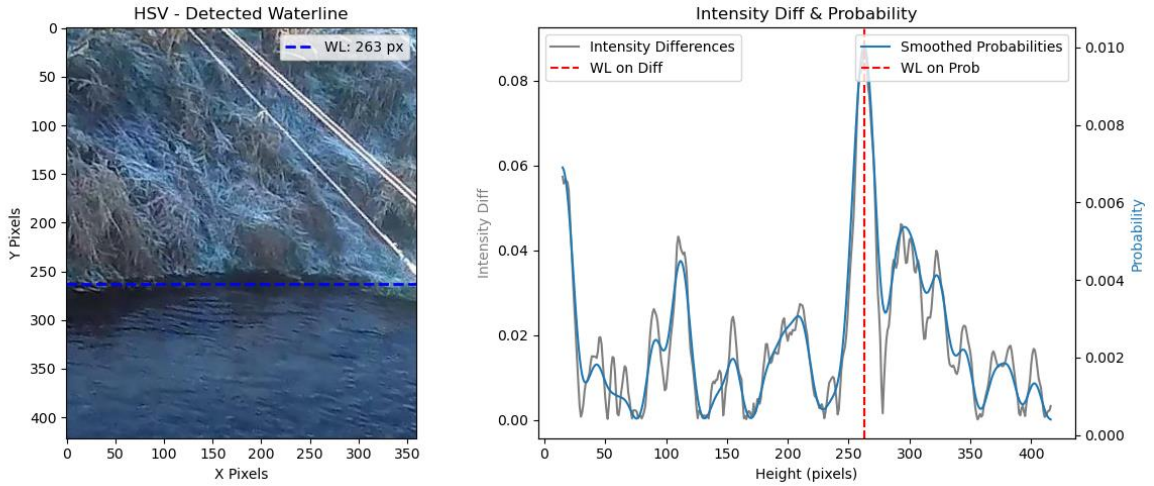
Here the hypothesis and expectation of the HUE being the most effective collour setting for land/water contrast was not met. The mean estimated error and root mean squared error were not significantly different.

In the scatterplot we see a decent correlation between algorithm results and manual estimation of the waterline. However, there are quite a lot of outliers.

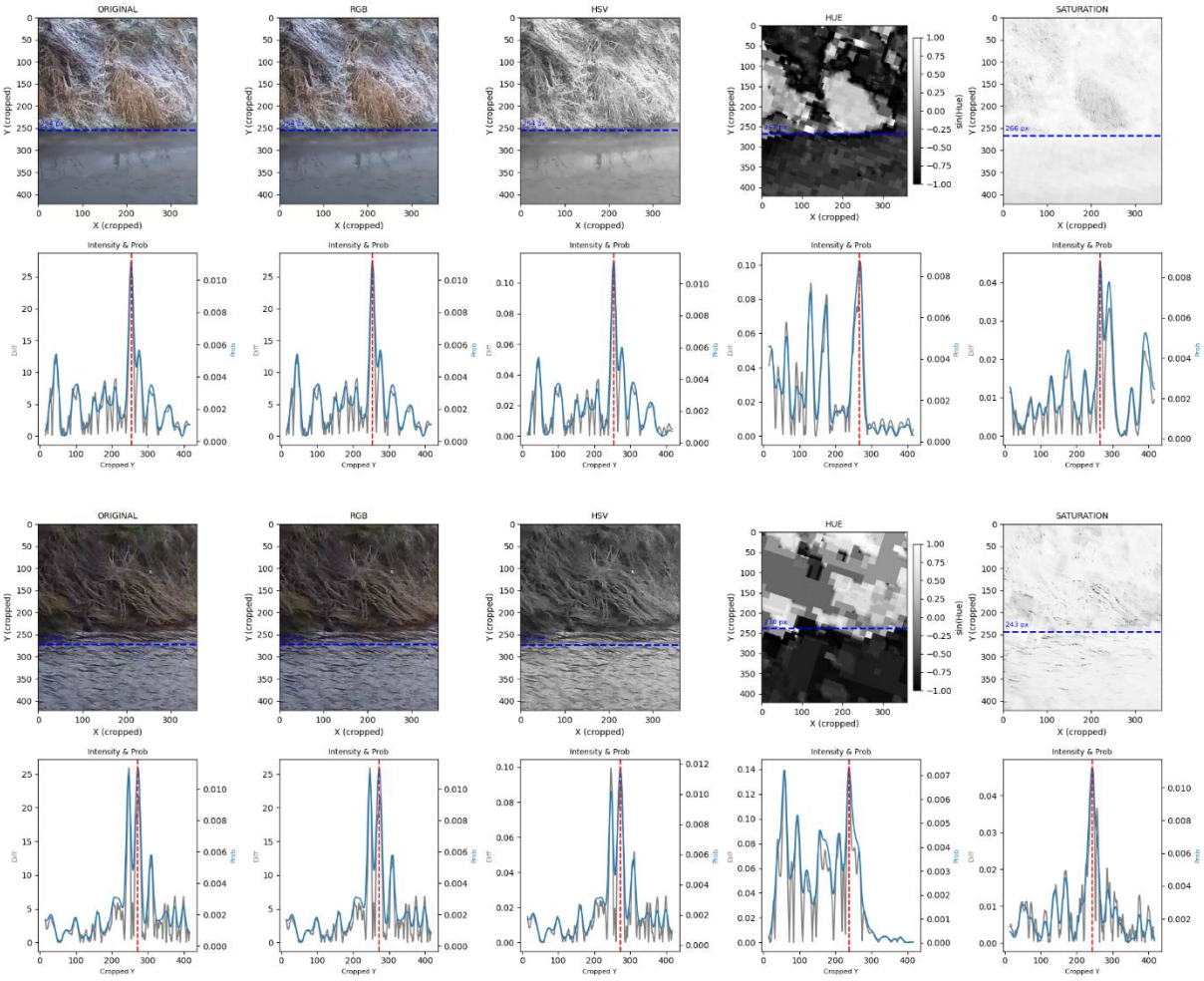




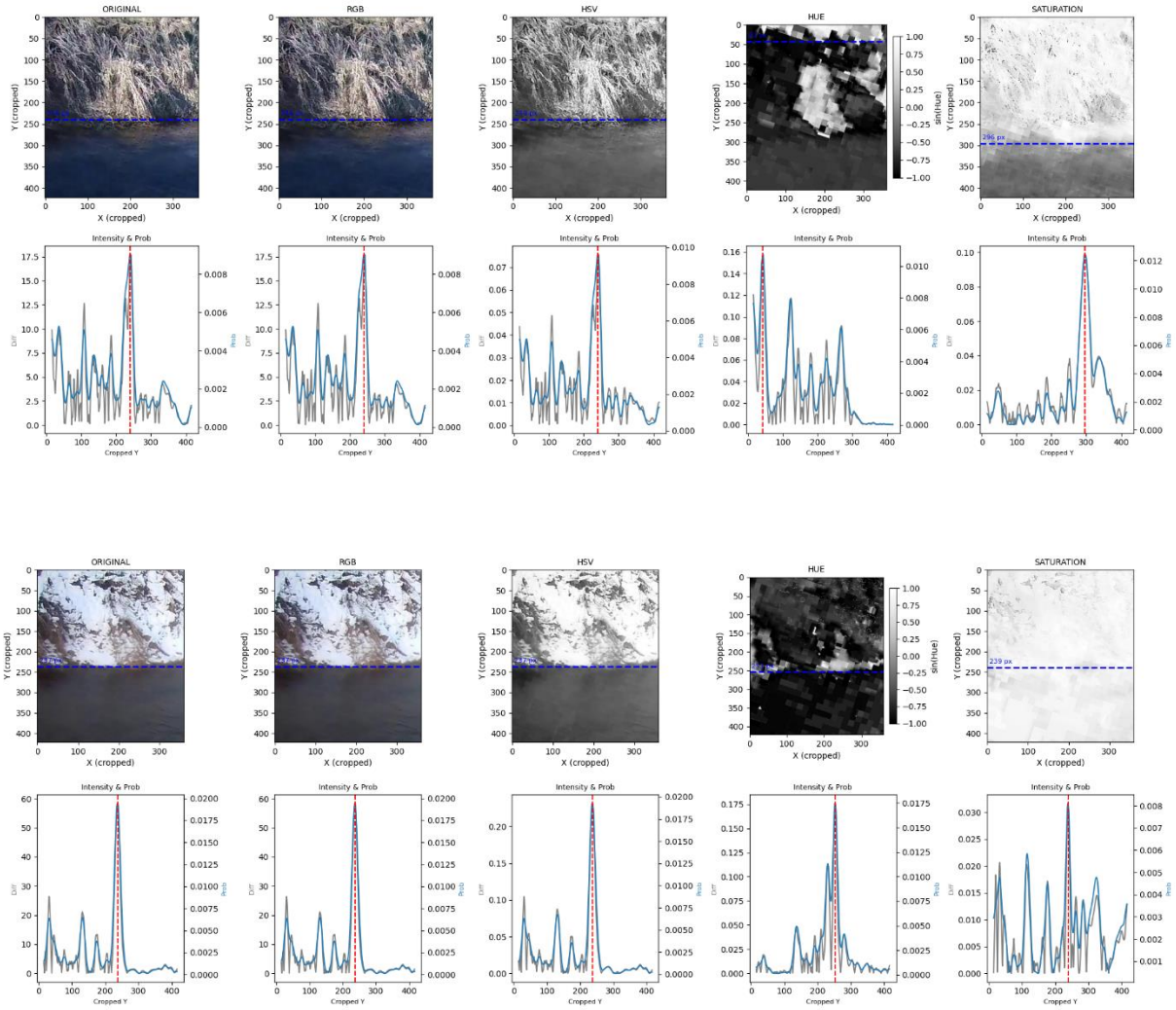
Pixel scale:



Sometimes, the water level detection algorithm performs realy well on all color settings. Other times, only one or none of the methods depict a representation of the real water level



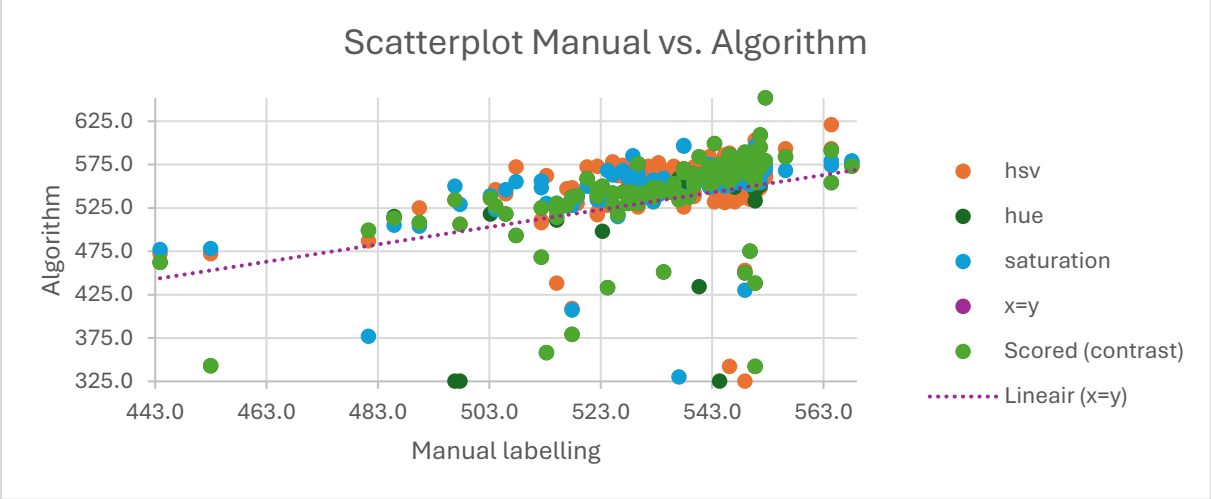
Under certain environmental conditions (frost or low sun angle), the hue image is very distorted or diffuse.

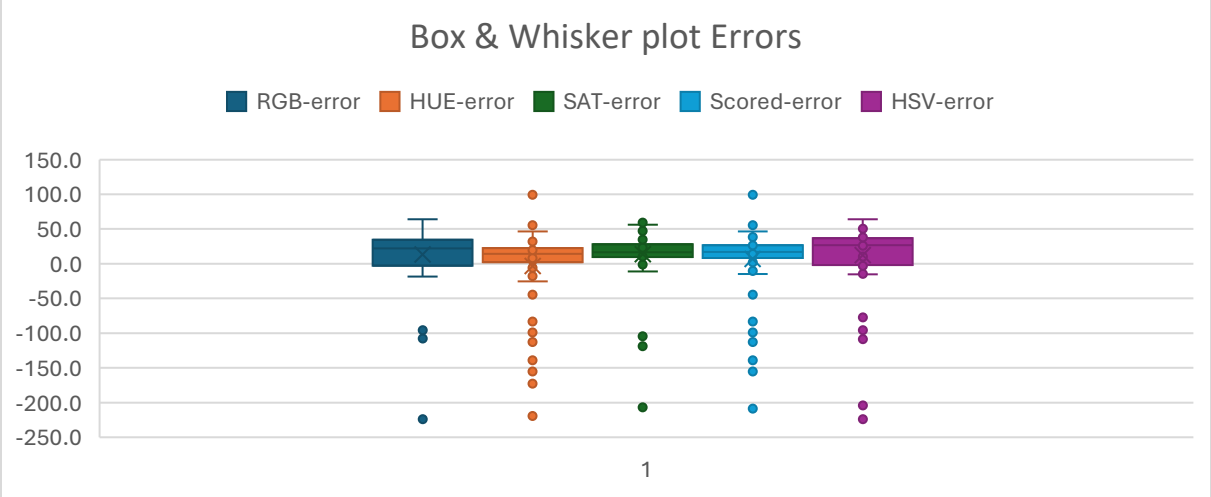


- Image_data_variants_dynamic cropping.ipynb
    - Utilises a dynamic cropping and rotating script to preprocess images and then run the waterlevel detection algorithm
- Image_data_variants_cropped_input.ipynb
    - Takes preprocessed images to reduce computation time and power demand. Runs the same script and should give the same results
- General images get visualized using RGB color coding. Each pigment gets a value between 0-255 which determines the color of the pixel. Besides RGB, a couple other color codings are available like the following. Hue, corresponds to where the color lies on the traditional color wheel. This means it ranges from 0-360* or 0-1. In coding it is recommended to connect the ends so that 1* 360* are similar instead of 359* apart. Saturation refers to the purity of a color. It measures how intens and vivid a color appears. Combine the last two and we get HSV (hue saturation value), which is in itself a alternative color coding.
- Using a labaling algorithm, the ‘real waterlinie’ is again assigned visually on a random set of 100 images from the Hommerich dataset. These were compared with the algorithm output for every color coding and with a scored output. The scored output selects the detected waterline with the highest contrast in pixel intensity of the intensity boxes across all color codings. However,

looking at the statistics, this scored method has not significantly increased the water line detection capabilities as shown in the statistics.

	RGB-error	HSV-error	HUE-error	SAT-error	Scored-error
MAE	26.18	29.61	34.50	24.65	29.83
RMSE	5.12	5.44	5.87	4.96	5.46
Bias	13.19	12.40	-3.05	13.72	6.74
Pearson r	0.39	0.34	0.50	0.50	0.53
R^2	0.15	0.11	0.25	0.25	0.28





7.4.8 Power monitoring experiment

The current setup is tested in standby mode (no active script running) and active detection mode (script running on repeat). The results concluded that with the current setup (powerbank without the use of a nano timer), experiments can be conducted as long as the battery can be changed every day.

Standby mode:		~275 mA/h	
Active mode:	running script constantly:	~522 mA/h	
Realistic use case:	running every ~10min:	~333 mA/h	~60h on 20.000mA battery

The current IR LED runs on a current draw of ~1A. however, because its only lit 2seconds every 10 minutes. The power consumption is not significant.

Several tests were conducted running the script. The inconsistency of the power consumption points to the possibility of more acting variables such as temperature, device efficiency after running updates ect.

7.4.9 Notes on prototype 4 and feedback loop for next prototype

The base for the hardware and algorithm is set. The main issue for now before we can transition to the next phase (fieldwork) is a robust and inclusive design so the prototype can be tested in a real world environment. A power saving mode is under development, but not successful yet. The nano power timer is causing power issues and the system is not yet capable of running the algorithm from bootup without any manual intervention. These things need to be addressed before we are able to transition to the testing phase.

SCAMPER Heuristics

- Substitute: What materials or processes can be replaced?
  - The device casing is not yet waterproof and has to be more robust for real world deployment. Also the option of switching to a 12V power source should be explored so night time measurements can be tested.
- Combine: What elements can be integrated?
  - The NoIR camera module should be tested for day measurements so only one camera module has to be present in the prototype. Is the NoIR camerea module as effective as the standard one?
- Adapt: How can the concept be adjusted for improvement?
  - The need for low power and power saving mode has to be improved so the system is only active when It has to. The nano power timer is now still giving issues that need to be resolved. Also a way of accessing the system outside would be great (portable display for example).
- Modify: What aspects (e.g., size or shape) can be altered?
  - The new casing has to be modified so rain is not damaging the camera or causing droplets or condensation that influence the image capturing.
- Put to another use: Can the concept serve other purposes?
  - The current device is able to be used as a monitoring system for security or the consumer market like a diy ring doorbell or birdhouse camera.
- Eliminate: What features can be simplified or removed?
  -
- Reverse: Can processes be reversed or re-sequenced?

7.5 PROTOTYPE 4B

Hessel has provided me with an python environment and script to transform the image to a 3D referenced system. A cross section is measured in the field and these coordinates are linked to the image. poligons run across this cross section and make pixel distributions across the entire cross section. These distributions can be compared to find the land/water line.

This function is optional and additional to the project. it gives insight on how to process the data on a separate server. The project however focusses more on taking measurements locally and on board processing.

7.5.1 Algorithm

A chi-squared method is applied to the script results to try to find the most probable land/water line. However, after running a couple of images trough the wle script, the chi2 does not seem to perform verry well (using RGB). As of right now, the WD script performs better. The WD differs from the WLE on a couple of aspects:

WD;

- Sweeps row by row over the ROI
- Measures mean intensity difference of the intensity boxes
- Score based on highest intensity difference
- Creates a 1D profile mean intensity difference

WLE:

- Creates polygons over an 3D cross section
- Displays intensity distribution (histograms) for every polygon
- Creates an CDF displaying ordered intensity counts in every polygon
- The cdf can be compared to show differences in texture/brightness/color of surfaces

Some adaptions are made to the code from pyorc:

- Color coding is changed so HUE is used. This seems to work better than RGB
- Added chi2 for wle
- Applied a piece of code that applies a penalty when both waterlines are not corresponding t othe same water height. This optimalisation step makes sure it detects land/water levels that imply an horizontal water surface.