Automated Acoustic Detection of Submerged Hydrocarbon Plumes Produced by Seeps and Spills

by

© Ginelle Claire Nazareth

A Thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Master of Engineering

Faculty of Engineering and Applied Science

Memorial University of Newfoundland and Labrador

October 2024

St. John's Newfoundland and Labrador Canada

Abstract

Assessing the impact of an oil spill is especially challenging when it forms a neutrally buoyant submerged plume, as in the case of the Deepwater Horizon well blowout. Autonomous Underwater Vehicles (AUVs) have proven to be an effective platform for detecting, tracking, and sampling these plumes due to their adaptive response capabilities. However, efforts to test AUV spill detection technology at naturally occurring seeps are hindered by challenges posed by seep environments. This thesis consequently focuses on addressing seep-site challenges by enhancing automated acoustic detection of submerged plumes. It proposes a novel plume detection algorithm for forward-looking sonars consisting of five steps: range-gating, segmentation, grid conversion, clustering, and georeferencing. Due to the computational complexity of the clustering step, a custom 'block clustering' algorithm for image data is developed to meet real-time processing requirements. A playback test using field trials data collected at Holyrood Bay demonstrates that the plume detection algorithm successfully identifies high-density clusters. Furthermore, the block clustering is consistently faster than the benchmark algorithm and produces clustering results that are visually more intuitive. The plume detection algorithm was implemented on Memorial University's Explorer AUV and utilized during trials to adaptively detect and sample the Scott Inlet seeps in Baffin Bay.

Acknowledgments

I am deeply grateful to my supervisors, Dr. Neil Bose, Dr. Jimin Hwang, and Dr. Ting Zou for their thoughtful guidance throughout my program and the much-needed course corrections when I lost sight of the bigger picture. In particular, I would like to thank Dr. Neil Bose for the exciting opportunity to be a part of the Scott Inlet Seeps project and experience a true Arctic adventure.

My graduate studies would not have been possible without funding from multiple sources: Memorial University's School of Graduate Studies provided my baseline funding, while the Fisheries and Oceans Canada's Multi-Partner Research Initiative (MPRI) and the Natural Sciences and Engineering Research Council (NSERC) Alliance program provided additional funding.

I would like to thank Dr. Evan Edinger for providing me with data acquired by the SuMo ROV aboard the CCGS Amundsen at the Scott Inlet seeps, and Dr. Jimin Hwang for providing me with sonar data of a micro-bubble plume; these data sources were invaluable for my research. I am also sincerely thankful for Gina Millar and Craig Bulger, who persevered despite overwhelming challenges during the field trials.

There have been countless people supporting me throughout this journey, and I would like to thank my parents, Rodney and Vanessa Nazareth not only for encouraging me to choose this path, but also for being the best role models of lifelong learners. I wish to thank Bill and Karen Tucker for welcoming me as family and giving me a home away from home. Thanks also go to Xi Chen, Irene Tong, Shauna Irani, and Gillian Nazareth for listening, empathizing, and ensuring I never felt alone. And finally, to Kodie Collings, for your help with diagrams, and for your limitless faith and love.

Table of Contents

List of Figures

List of Tables

Glossary

ACE	Automated Control Engine
AUV	Autonomous Underwater Vehicle
CDOM	Chromophoric Dissolved Organic Matter
CFD	Computational Fluid Dynamics
CTD	Conductivity, Temperature, and Depth
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DVL	Doppler Velocity Log
DWH	Deepwater Horizon
INS	Inertial Navigation System
LISST	Laser In-Situ Scattering and Transmissometry
MOODB	MOOS Database
MOOS	Missions Oriented Operation Suite
MOOS-IvP	Missions Oriented Operation Suite - Interval Programming
PAHs	Polycyclic Aromatic Hydrocarbons
PCC	Payload Control Computer
ROV	Remotely Operated Vehicle
VCC	Vehicle Control Computer
WHOI	Woods Hole Oceanographic Institution

Chapter 1

Introduction

1.1 Background and Motivation

The 2010 Deepwater Horizon (DWH) oil spill released an unprecedented 3.19 million barrels of oil into the Gulf of Mexico [1]. Although early ship-based observations indicated that large amounts of oil remained trapped in the deep ocean, the existence of submerged oil plumes was debated at the time. Their existence was confirmed, however, one month following the initial release when submerged oil was detected at a depth of approximately 1000 m [2]. Studies following the DWH disaster showed that submerged oil can persist in the deep sea for more than six months, and is toxic to marine life [3]. Naturally, significant efforts have been put into characterizing, delineating, and understanding the fate of the submerged oil [3].

A range of monitoring platforms are utilized to detect and track submerged oil. Of these, Autonomous Underwater Vehicles (AUVs) are most appropriate for spills occurring over large spatial and time scales [4]. As untethered vehicles, AUVs are not spatially constrained by the surface support vessel, and can also respond to onboard sensor measurements in real-time. These adaptive response capabilities are essential given the time-sensitive nature of spills. Consequently, AUV technology has been developed to adaptively track, characterize, and

sample submerged oil plumes. The collected water samples are sent to laboratories for chemical analysis to definitively confirm the presence of oil [5].

There are several factors that make characterizing submerged oil plumes challenging, including (1) the complex chemical composition of oil, (2) the multi-phase nature of the plume, and (3) the degradation and transport of the plume by natural processes [6]. Given these challenges, the onboard sensors can provide only a likelihood of the existence of oil. Tracking methods designed for AUVs during the DWH spill utilized a single sensor measurement for the decision criterion [2], [7]. Building on this pioneering work, recent research has focused on enhancing autonomy by combining data from multiple sensors [8].

Testing is an essential part of the design process but releasing oil into the ocean for this purpose is both counterproductive and illegal. Dye tracers and bubble plumes have accordingly been identified as environmentally friendly proxies for spilled oil [9],[10]. However, since these proxies do not emulate both the physical and chemical properties of oil, they cannot produce positive detections across a suite of plume detection sensors because each sensor detects a different property of the plume. It is consequently challenging to create a test environment for a multi-sensor system, and an alternative approach is to test the technology at naturally occurring oil and gas seeps. Seeps, however, are more challenging to detect [11], and this thesis consequently focuses on expanding the scope of AUV spill detection capabilities to include the detection of seep sites.

1.2 Research Questions

The following is the core research question of this thesis:

Q. How can the oil spill detection capability of an AUV be enhanced to facilitate testing at a natural seep location?

This thesis is structured around the following sub-questions, which derive from the core research question:

Q1. To what extent can existing oil spill detection methods be utilized at natural seep locations?

Q2. What methods exist for automated acoustic detection of submerged oil and gas plumes?

Q3. How can an AUV use automated acoustic detection to detect both spill plumes and seep plumes?

1.3 Organization of Thesis

Chapter 2: Literature Review first covers the background information required to answer Research Question 1. It includes a comparison of seep and spill plumes, as well as sensors and methods for detecting submerged plumes. Through an analysis of the applicability of spill detection methods in seep areas, it identifies acoustic detection as a viable method requiring further research, thereby motivating Research Question 2. The subsequent review of automated acoustic detection methods for seeps, submerged spills, and pipeline leaks reveals that existing algorithms for processing sonar data are not applicable to both seeps and spills. Thus, a novel plume detection method is required.

Chapter 3: Plume Detector Design, presents a novel plume detection algorithm for a forward-looking scanning sonar, thereby answering Research Question 3. The algorithm creates

a binary image of the sonar data and recognizes the plume as dense clusters of points in the image; its final output is the georeferenced center coordinates and radius of each cluster. Although existing clustering algorithms could be utilized by the plume detector, they are not well suited for this image-based real-time application. Accordingly, a novel 'block clustering' algorithm which combines density-based clustering and image processing methods is also presented.

Chapter 4: Plume Detector Performance validates the proposed plume detection algorithm using field trials data collected by Memorial University's Explorer AUV at Holyrood Bay. The trials were designed to collect acoustic data of a micro-bubble plume, which was utilized as a proxy for a submerged oil plume. During the playback test, the plume detector effectively identified high-density features in several scans of the sonar data. Chapter 4 also contains an analytical comparison between the proposed clustering algorithm and the frequently utilized Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. It shows that the block clustering algorithm produces a better clustering output and outperforms DBSCAN in a computation time performance test. The block clustering algorithm is consequently a better option for this application as real-time performance enables rapid adaptation. Finally, Chapter 4 provides direction on how to tune some of the plume detection algorithm parameters.

The *Conclusion* in *Chapter 5* contains a summary of the thesis and recommendations for future work. It demonstrates that the plume detection algorithm is well suited for applications integrating multiple sensors. More significantly, the block clustering algorithm can be utilized in any image processing application where density-based clustering is required.

1.4 The Scott Inlet Seeps Project

This research was conducted as part of a Memorial University project: *Investigating Scott Inlet Seeps with Autonomous Underwater Vehicles*. The primary objective of the project is to enhance AUV capabilities for oil spill response through the development and testing of new technologies [12]. The project's field trials were conducted in September 2023 at the Scott Inlet seeps in Baffin Bay, and the results of the trials are published in [13]. During these trials, I was involved with several aspects of operation, including mission planning, piloting and data analysis.

Over the course of the Scott Inlet Seeps project, a software suite consisting of new and existing applications was compiled for Memorial University's Explorer AUV. Some applications in the suite were responsible for interfacing with sensors and processing the data onboard; others controlled the mission, and adaptively modified the search plan based on sensor data to collect a water sample at the optimal location. The software suite also included the plume detection software developed for this thesis. A higher-level description of the missions and software is provided in [13], while Appendix C contains the complementary software-focused documentation. The Appendix includes an overview of the individual software components, how they interact with each other, and descriptions of their interfaces. It should be noted that while the software interface design documented in Appendix C is my own work, the higher-level mission design for the Scott Inlet Project is not, and the implementation of the software represents a collaborative effort.

During the Scott Inlet trials, adverse weather conditions and a range of hardware issues significantly limited operational days at the actual seep site, and most dives were consequently

conducted in sheltered water near the seep area. Two dives were conducted at the location where bubble ebullition had been previously observed, but no plume-like features were present in the sonar data that was collected when the vehicle was at depth. Several factors including the INS drift, the AUV's high altitude, the slower scan rate of the sonar, and lower activity of the seep itself, could have caused the AUV to miss the seep plumes. The plume detector software however, worked as intended. The sonar 'saw' the water surface when the vehicle was diving and surfacing, and these detections were correctly classified as clusters by the plume detector. Since the vehicle was not at depth when the detections occurred, the clusters were appropriately ignored by the mission control system. Even though the trials did not produce useful acoustic data, it is important to recognize that the goal of testing the developed software at the Scott Inlet seeps has driven the design and implementation of the plume detection algorithm presented in this thesis.

1.5 Research Contributions

The following are the notable research contributions of this thesis:

1. Identification of the absence of automated plume detection algorithms for forward-looking sonars which are appropriate for both seep and spill plumes.
2. Development of a novel plume detection algorithm for a forward-looking sonar.
3. Development of an innovative clustering algorithm which operates directly on image data. To the best of my knowledge, this clustering algorithm represents a completely original approach.

Chapter 2

Literature Review

2.1 Introduction

The first half of this literature review provides an overview of AUV plume detection technology and the marine environment in which it is used. More specifically, the seep and spill environments are described in Section 2.2, while the plume detection sensors and tracking methods are covered in Section 2.3 and Section 2.4, respectively. Section 2.5 draws from this background information to analyze the extent to which spill plume detection methods for AUVs can be used at seep sites. The analysis reveals challenges created by the seep environment and identifies acoustic detection as a method that can address these challenges. Accordingly, Section 2.6 reviews automated acoustic detection algorithms for submerged plumes from a range of research fields. It finds that none of the algorithms are applicable to both seep and spill plumes, but identifies a promising clustering-based approach, which is developed in the next chapter. Finally, Section 2.7 verifies that the Scott Inlet seeps can be detected using a forward-looking sonar.

2.2 Seep and Spill Plumes

Natural seeps occur primarily along continental margins, where oil or gas from subsurface reservoirs flows through fissures and faults into the ocean [14]. These seeps are considered analogous to terrestrial oases as they provide the basic nutrients for chemoautotrophic microorganisms, which in turn support higher forms of life [15]. While seeps occur naturally, spills are caused by anthropogenic activity and can result in the formation of submerged oil plumes regardless of whether the leaking source is pressurized or not [6]. The focus of this thesis, however, is on spill plumes caused by pressurized leaking sources such as the broken DWH oil well, which exposed limitations in submerged spill detection technology [16].

An understanding of the differences between seeps and spills is required when testing oil spill detection technology at a natural seep site. The primary difference is that hydrocarbons are released into the environment at a much faster rate during an oil spill. For example, the daily release of oil from the damaged DWH well was on the same scale as the annual natural seepage in the Gulf of Mexico [17]. Spills consequently overwhelm marine ecosystems, whereas seeps release oil at a slower rate that allows ecosystems to adapt [18]. Furthermore, unlike spills, seeps have multiple sources with transient fluid flows which vary both spatially and temporally [19]. Flow rates are influenced by environmental factors such as tide-induced pressure variations and bottom current velocities [20]. Most seeps are calm, with a slow seepage of oil or dissolved methane into ambient waters; white bacterial mats on the seafloor are often the sole visible indicator of a methane seep, as eruptive bubble plumes form only where the methane concentration exceeds the saturation level [21].

The deep plume formed in the wake of the DWH well blowout was distinctly different from natural seep plumes [22]. A sudden drop in pressure at the release point resulted in the formation of microscopic oil droplets. Subsequently, a submerged multi-phase plume consisting of oil droplets with a relative neutral buoyancy (< 70 µm diameter), dissolved hydrocarbon compounds, and gas bubbles entrained with seawater was formed [1], [23]. The DWH plume was transported laterally by currents to extend continuously more than 35 km from the source [24]. Seeps, however, release hydrocarbons slowly and diffusely rather than with a high intensity, and cannot create the same type of large multi-phase plume. While the dissolved methane forms a diffused methane-rich area close to the seabed and can cover tens of square kilometers, the vented gas bubbles rise without spreading laterally and have a footprint that is typically less than tens of square meters [22], [25].

At seep sites, microbes in the sediment biodegrade the hydrocarbons even before they are released into the water, contributing to a lowered environmental impact. However, once in the water column, the fate of hydrocarbons produced by seeps and spills is governed by the same environmental and biological processes. A significant amount of oil rises to the surface to form a slick, while soluble components dissolve in the water column, and simpler hydrocarbons such as methane are quickly consumed by bacteria. At the surface, the lighter compounds evaporate and the remaining oil is broken down over time by microbes and weathering processes. Eventually, the remains "fallout" into a neutrally buoyant layer or back into the sediment [26], [27].

2.3 Hydrocarbon Plume Detection Sensors

Oil is a complex substance composed of dozens of major hydrocarbon compounds and thousands of minor ones [28]. Given the complex chemical nature of oil, the size of a submerged spill plume, and its dynamic multi-phase nature, a range of sensors are used to detect, characterize, and track submerged oil [16]. Among these sensors, mass spectrometers provide highly selective and sensitive information about dissolved hydrocarbons; they uniquely identify a range of complex compounds and can detect trace amounts of chemicals with concentrations in the ppb range [16], [29]. However, existing underwater mass spectrometers' capabilities are limited to measuring the lighter hydrocarbons [30].

Fluorometers detect oil based on its tendency to fluoresce, and despite being less selective and sensitive than mass spectrometers, are frequently used because of their low cost, availability, light weight, and usability [16], [29]. Fluorometric measurements of the heavier polycyclic aromatic (multiple-ring) hydrocarbons (PAHs) [31] complement mass spectrometry measurements of their lighter counterparts. Fluorometers require calibration to distinguish oil from other naturally occurring fluorescing substances in the water [29], and laboratory analysis of collected water samples is used to confirm that measured fluorescence peaks were produced by oil [16]. Since fluorometers do not indicate the degree of dispersion, data from a holographic camera or laser in-situ scattering and transmissometry (LISST) sensor can be used to complement fluorometric data [16]. Both instruments can provide estimates of the size distribution of droplets in the water column based on how light is scattered [11], [32]. This droplet size distribution can be used not only to predict the fate of the plume, but also to

determine whether the plume dispersion is due to physical forces or chemical dispersants since dispersants create smaller droplets [16].

Most sensors utilized for submerged oil and gas detection are in-situ sensors which must first come in contact with hydrocarbons to detect them [33]. Sonars, however, are remote sensors which can scan large sectors of the water column to detect acoustic scatterers such as oil droplets and gas bubbles [11], [34]. While they enable plume tracking in fast-moving currents, their primary limitation is that they may not be capable of distinguishing between oil droplets and other targets in the water column [34]. Although theoretically, acoustic frequency-based methods exist to differentiate between fish and bubbles [35], in practice these methods cannot be utilized with the data from commercial sonar systems. The down-sampled version of the complete acoustic waveform provided by commercial sonars is insufficient for a frequency analysis, which requires a high-resolution signal. Thus, a combination of sensors which detect different characteristics of the plume are utilized in practice. This approach not only mitigates the impact of false positives, but enables researchers to correlate the sensors' data and positively identify the plume even in the absence of prior knowledge of the sensor signatures [36].

2.4 AUV Tracking of Submerged Oil Plumes

Approaches to AUV tracking of submerged spilled oil plumes are based on what is known about the plume and the environmental conditions, as well as the capabilities of the AUV itself. Although the tracking algorithms utilized by AUVs during the DWH spill reflected low levels of adaptation, they were successful because researchers first achieved a general localization of the plume using a towed instrument package before planning the AUV missions.

For example, the Woods Hole Oceanographic Institution (WHOI) researchers utilized an automated and reconfigurable, but not autonomous approach with the Sentry AUV to determine the horizontal extent of the submerged oil. The AUV was programmed to descend until it detected oil, and then follow a pre-planned zig-zag path transecting the plume down-current from the wellhead. During the mission, Sentry transmitted the mass spectrometry data back to the surface vessel; researchers monitored this data, which showed distinct 'background' and 'plume' regions, and commanded the vehicle to cut the tracklines short once it indicated that the vehicle was no longer within the plume [7]. The detected neutrally buoyant plume was at times 200 m high and 2 km wide and extended continuously for at least 35 km from the source.

The method utilized with the Sentry AUV proved to be effective but required significant human intervention. Accordingly, Jakuba et. al proposed a Bayesian clustering method through which an AUV can use data from multiple hydrocarbon sensors to semi-autonomously identify a hydrocarbon plume, and validated it using data collected during the DWH spill [8]. While Kukulya et. al have focused on developing and validating an adaptive method for transecting a plume [37], some studies have taken an alternative boundary-following approach [27], [38], [39]. For example, Hwang et. al have developed an algorithm for an AUV to autonomously delineate a submerged oil plume using a forward-looking sonar [38].

2.5 Utilizing Spill Detection Methods at Seeps

While the low hydrocarbon concentrations at natural seeps support life, it makes detection more challenging. This was made evident during a series of trials conducted at the Santa Barbara seeps off the coast of California in 2019. During the trials, an AUV equipped with a

fluorometer collected water samples in areas of higher fluorescence using a simple threshold-based algorithm. Despite confirmation from the AUV's holographic camera of oil droplets at the sampling locations, laboratory analysis of the samples indicated that they did not contain significant quantities of oil; the hydrocarbon concentrations were similar to those in samples from an uncontaminated site. Researchers consequently concluded that further trials in areas with higher concentrations of oil are necessary [5], [11].

Like the approach utilized with the Sentry AUV during DWH, a significant amount of operator oversight was involved during the Santa Barbara trials. An AUV equipped with a sonar was utilized as a pre-screening tool to survey large areas and identify regions where gas bubbles or oil droplets were present. Missions with the primary water-sampler equipped AUV were then planned in areas identified as 'hot spots' using the sonar data. The spatial and temporal variability of fluid flow at the seeps, however, revealed that behaviours with more rapid adaptation were required [11].

Since hydrocarbons are released at seep sites through a distributed network of fissures, the spatial distribution of seep plumes is inherently different from spill plumes. An AUV survey of the Australian Yampi Shelf seeps shows varying methane concentration levels within the same area [19]. This stands in contrast to the distinct plume boundary observed at the DWH site. Furthermore, ship-based geochemical surveys of seeps on the north-west Australian continental margin show pockets with higher hydrocarbon concentrations rather than a continuous plume [40]. In the absence of distinct plume boundaries and the presence of multiple hydrocarbon peaks at seep sites, autonomous methods developed to track spill plume boundaries may not be successful. It would consequently be more viable to focus on testing detection and sampling methods.

This thesis focuses on automated acoustic detection of submerged plumes to address the challenges faced when using spill detection methods at seep sites. The challenges presented by the presence of multiple sources and lower hydrocarbon levels are addressed by integrating a sonar as it detects the higher concentration sources where methane saturation causes bubble ebullition. Furthermore, automating the acoustic detection addresses challenges stemming from spatiotemporal variability by enabling more rapid adaptation. The focus here is specifically on detection, and not tracking, to allow for the method to be tested at seep sites.

2.6 Automated Acoustic Plume Detection

Research involving automated acoustic detection of subsurface oil and gas plumes is directed toward detecting spilled oil, natural seeps, or pipeline leaks (see Table 2.1 for an analytical summary of the relevant research papers). Since the detection of submerged spill plumes using sonars is a relatively new field, the literature in the area is sparse. Furthermore, existing algorithms for tracking spilled oil plumes [38], [41] are not suitable for seep sites; they are not designed for the smaller acoustic footprint of seep plumes, and their detection and tracking mechanisms are very tightly coupled.

Research focusing on automated detection of seep bubbles and gas leaks utilizes bottom-facing sonars to detect the characteristic acoustic flare shape (Fig. 2.1). These approaches rely on the detection of long vertically-oriented shapes or edges [42], [43], [44], [45], [46], or data patterns produced by rising gas bubbles [47], [48], [49]. They cannot, however, be utilized for neutrally buoyant spill plumes which do not have the same vertical flare shape or rising bubbles. Furthermore, mounting the sonar in a forward-looking orientation is better suited for

spill delineation as the plume forms one or more horizontal intrusion layers in the stratified ocean [50].

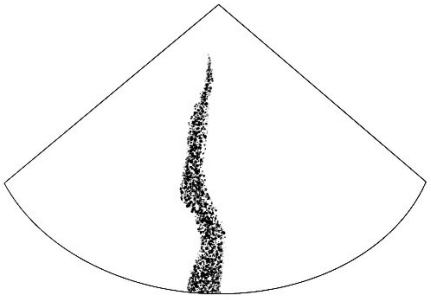


Fig. 2.1: Illustration of a seep plume's rising flare shape in bottom-facing sonar data

Research using machine learning approaches tends to be limited in subsea applications due to the absence of sufficient training data, which results in model over-fitting. To work around this issue, Speck et. al generated synthetic MBES images of bubble plumes using a Computational Fluid Dynamics (CFD) simulator and style-transfer algorithm [51]. These images were then used to train a YOLOv3 neural network, and the resulting object detector successfully detected compressed air bubbles in real MBES data. The process of developing a CFD simulator, however, introduces a level of complexity that is not required here where the primary goal is simply to distinguish real targets from noise. Similarly, networks designed to suppress the model-overfitting issue caused by small sample sets are also relatively complex [52].

In summary, existing automated seep and gas leak detection methods would not transfer well to spill detection. Given the absence of research on automated seep plume detection using a forward-looking sonar, some of the research on more generic automated target detection algorithms for sonars was reviewed. In general, detection algorithms tend to require some knowledge about the object; for example, template matching algorithms utilize the object's size

and shape, whereas statistical classifiers expect a distribution for the highlight or shadow [53]. Thus, the literature on non-model-based object detection is more relevant to plume detection. The work of Johannsson et al. [54], which utilizes a simple computer vision pipeline of smoothing, gradient thresholding, and clustering, provides valuable insight. Based on this literature review, the following chapter develops a plume detection algorithm which identifies the plume as a dense cluster of points in forward-looking sonar data.

Table 2.1: Summary of research involving the automated acoustic detection of subsurface oil and gas plume

Title	Citation	Sonar Type	Sonar Orientation	Objective	Algorithm Approach	Reason for not selecting approach
Oil Plume Mapping: Adaptive Tracking and Adaptive Sampling from an Autonomous Underwater Vehicle	[38]	Scanning Sonar	Forward Facing	Spill Plume Tracking	Identifies scan lines with high intensity detections and then identifies oil patches as consecutive scan lines with the high intensity detections.	Not designed for the smaller footprint of seep plumes. Detection and tracking methods are tightly coupled.
Bubble Plume Tracking Using a Backseat Driver on an Autonomous Underwater Vehicle	[41]	Scanning Sonar	Forward Facing	Spill Plume Tracking	Utilizes the stalled continuity method to identify scan lines with high intensity detections and then uses Gaussian blurring to identify oil patches.	
Automatic Detection and Segmentation on Gas Plumes from Multibeam Water Column Images	[42]	Multibeam Sonar	Bottom Facing	Seep Detection	Detects vertically oriented edges using a Haar classifier and richer texture features using a Local Binary Patterns (LBP) detector	Relies on the detection of long vertically oriented shapes or edges
Subsea pipeline leak inspection by autonomous underwater vehicle	[43]	Multibeam Sonar	Bottom Facing	Pipeline Leak Detection	Detects edges in the sonar image using morphological edge detection	
Comprehensive Detection of Gas Plumes from Multibeam Water Column Images with Minimisation of Noise Interferences	[44]	Multibeam Sonar	Bottom Facing	Seep Detection	Uses morphological features such as height, area, and width to distinguish gas plumes from noise	

Extended Detection of Shallow Water Gas Seeps from Multibeam Echosounder Water Column Data	[45]	Multibeam Sonar	Bottom Facing	Seep Detection	Detects large clusters with a certain height/width ratio and distance from the seafloor	Relies on the detection of long vertically oriented shapes or edges
Automatic gas leak detection system	[46]	Multibeam Sonar	Bottom Facing	Pipeline Leak Detection	Segments the image into foreground and background sections using maximum entropy segmentation and then uses the Hough transform to detect linear features	
A method for undersea gas bubbles detection from acoustic image	[47]	Multibeam Sonar	Bottom Facing	Pipeline Leak Detection	Utilizes a scale-invariant feature transform (SIFT) flow algorithm to estimate the motion characteristics of gas leaks	Relies on data patterns produced by rising gas bubbles, and requires the sonar to be in a bottom-facing orientation
Automatic Detection of Marine Gas Seeps Using an Interferometric Sidescan Sonar	[48]	Interferometric Sidescan Sonar	Bottom Facing	Seep Detection	Detects regions of high intensity and low interferometric (spatial) coherence in the images formed by the two receiver arrays	
Technical Note: Detection of gas bubble leakage via correlation of water column multibeam images	[49]	Multibeam Sonar	Bottom Facing	Seep Detection	Detects data patterns produced by rising gas bubbles using a cross-correlation technique from particle imaging velocimetry	
Supervised Autonomy for Advanced Perception and Hydrocarbon Leak Detection	[51]	Multibeam Sonar	Bottom Facing	Pipeline Leak Detection	Trained a YOLOv3 object detector using synthetic MBES images of bubble plumes generated by a Computational Fluid Dynamics (CFD) simulator.	Requires CFD simulations, which introduces a level of complexity not required for this application

2.7 Bubble Ebullition at the Scott Inlet Seeps

In [55] Nikolovska et. al presented scanning sonar images of single and clustered bubble streams from gas seeps in the eastern Black Sea. These images revealed that it may not be possible to distinguish the acoustic backscatter of a single bubble stream from noise, but the backscatter from a cluster of streams produces distinctive features in the data. Given the plan to test the developed algorithm at the Scott Inlet seeps, video data of these seeps was analyzed to determine if the bubble plumes were sufficiently large for detection.

The Scott Inlet seeps are located in Baffin Bay, primarily along the south wall of Scott Trough, and have been active for at least 45 years [56], [57]. A 2018 research expedition located a prominent methane seep in the area at 71.37812° N, -70.07452° W (Stn0) using a Remotely Operated Vehicle (ROV) (see Fig. 2.2) [58]. Video recordings from this expedition[1] were reviewed to identify seep locations based on the presence of microbial mats and bubble plumes, and these locations are mapped out in Fig. 2.3. Approximately 18 seepage locations were observed during the survey, most evidenced only by microbial mats (see Fig. 2.4). Bubble ebullition occurred at three sites in the south-west region of the survey area, all within 30 m of each other. While only a single bubble stream was observed at one of these sites, the remaining two sites presented large fields of bubbles (see Supplementary Video for [58]); the more active sites should be detectable using a forward-looking sonar.

---

[1] Video data collected using SuMo ROV aboard CCGS Amundsen. Funding source for acquiring the video data: ArcticNet Hidden Biodiversity project, Amundsen Science CFI funding, and NSERC Ship Time grant to E. Edinger et al.

During the expedition, water samples were collected at Stn0 where bubble streams were observed, as well as from eight sites within a 5km radius of it. Although the bottom water methane concentration was highest at Stn0, samples taken over a 24-hour period reveal a high degree of variability, with concentrations ranging from 4 nM to 610 nM [58]. This suggests a large daily variation in methane flux and bubble ebullition activity.

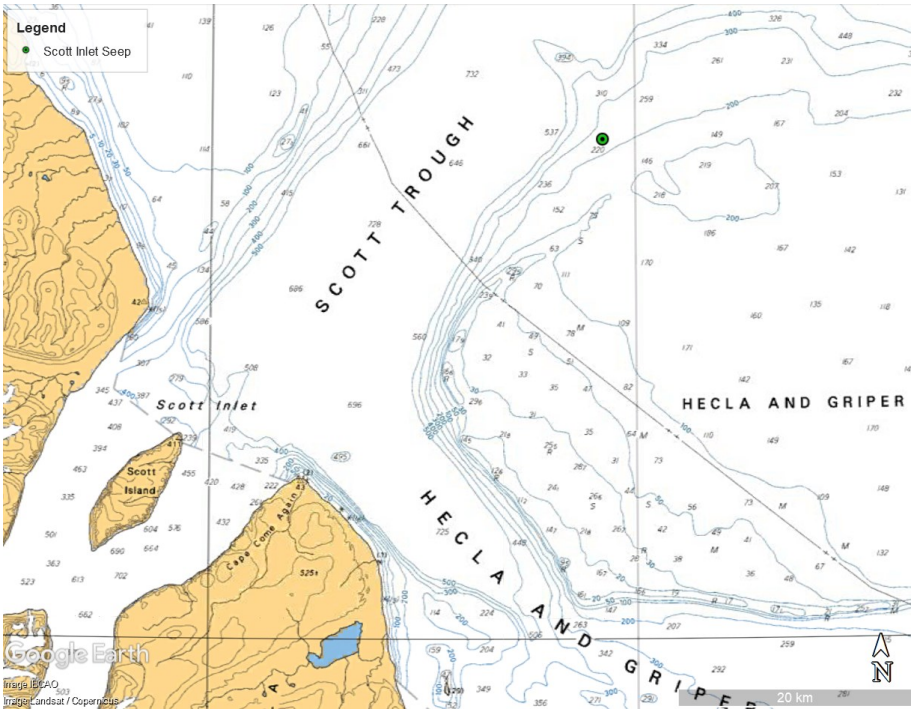


Fig. 2.2: Location of the seep identified by the ROV during the CCGS Amundsen 2018 cruise to Scott Inlet. The nautical chart is published by the Canadian Hydrographic Service [59].

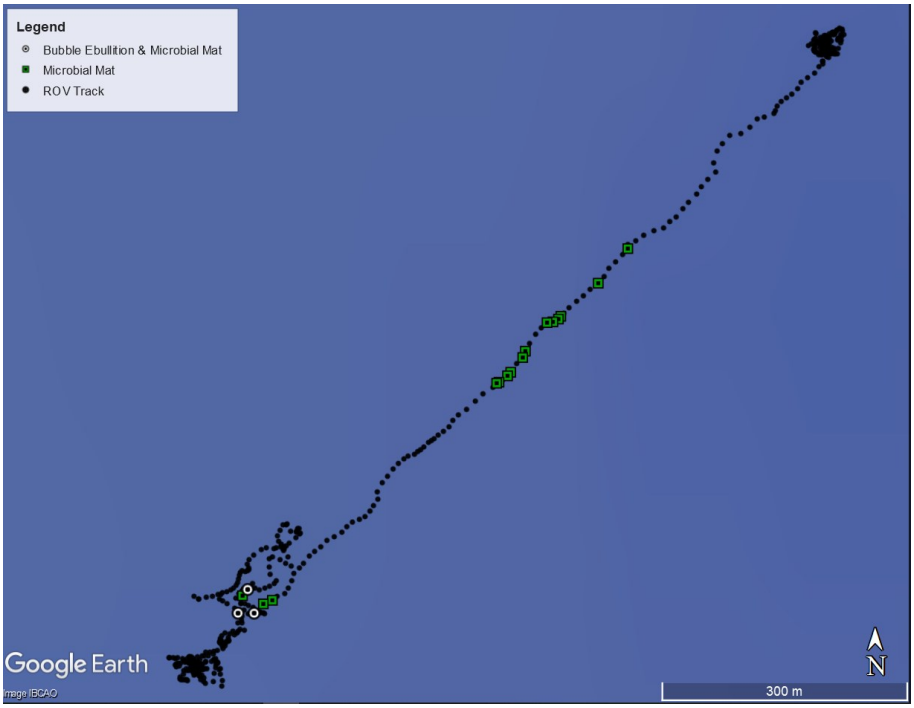


Fig. 2.3: ROV track along with seep locations identified by microbial mats and bubble ebullition[1]

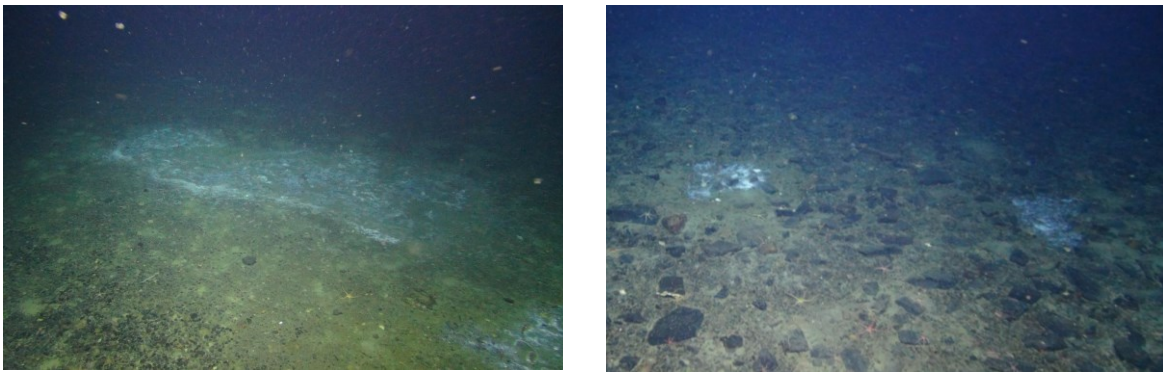


Fig. 2.4: White microbial mats indicating methane seepage through the seafloor at Scott Inlet[1]

2.8 Conclusion

In conclusion, this literature review has identified that developing acoustic detection methods would expand the scope of AUV spill detection technology and enable testing at seep locations. Automated acoustic detection addresses seep-site challenges stemming from the presence of lower hydrocarbon concentrations and multiple sources with spatiotemporally varying flow; the focus is solely on detection, as tracking methods may not be successful at seep sites. The

literature review has also demonstrated that existing plume detection algorithms do not apply to both seep and spill plumes. It has subsequently motivated the development of a clustering-based approach to plume detection using a forward-looking sonar and has verified that this approach would be viable at the Scott Inlet seeps.

Chapter 3

Plume Detector Design

3.1 Introduction

This chapter focuses on the development of a novel plume detection algorithm for a forward-looking scanning sonar. The algorithm is designed for a multi-sensor approach to plume detection in which its output is combined with data from in-situ hydrocarbon sensors. Its objective is to enable rapid adaptation by processing the sonar scans during the mission and identifying areas of interest in which more detailed surveys are required. Due to the inherent limitations of commercial sonars which result in a low discriminative capacity, the algorithm differentiates only between real targets and noise. The task of specifically identifying hydrocarbons in the water column is left to the in-situ sensors. The plume detection algorithm identifies the plume as high-density regions in the sonar data and returns the georeferenced center coordinates of these regions. It consists of five steps: range-gating, segmentation, grid conversion, clustering, and georeferencing.

This chapter begins with a description of the factors which have shaped the plume detector's design decisions (Section 3.2) and then covers the operation of the Ping360 sonar (Section 3.3), the scanning sonar for which the algorithm was designed. Following a brief overview of the

algorithm in Section 3.4, each step is described in a separate section. The source code for the plume detection algorithm is provided in an online GitHub repository [60], as well as in Appendix A and Appendix B.

3.2 Design Considerations

The plume detector's design was shaped by several factors, which are listed here:

1. Real-time performance: Given the plan to use the plume detector at the Scott Inlet seeps, achieving real-time performance was a high priority. The data must be processed faster than the rate at which it is received to enable an adaptive response to a dynamic environment.
2. Configurability: The algorithm needed to be configurable not only for seep and spill plumes of different sizes but also for different platforms, as processing capabilities vary significantly from one system to another.
3. Sensor availability: The plume detector was designed for the Ping360 scanning sonar because it had already been integrated on Memorial University's Explorer AUV. While some of the algorithm's details are consequently specific to the Ping360, they can be easily adapted for any scanning sonar.
4. Automation before autonomy: In this application, a truly autonomous system would need to combine data from different sensors to identify the plume without prior knowledge of the sensor signatures. However, even the most advanced algorithm for hydrocarbon plume identification using multiple sensors remains semi-autonomous as it relies on an operator to assign meaning to the processed data [8]. Additionally, the method is limited to in-situ sensors which produce 1D timeseries data and is not applicable to remote sensing sonars,

which generate 2D acoustic data. Thus, substantial research is required to achieve truly autonomous detection using sonars, and is beyond the scope of this thesis. The focus here is consequently on automating acoustic detection as a first step towards autonomous detection.

3.3 Ping360 Sonar Operation

The Ping360 (Fig. 3.1) is a mechanical scanning sonar (MSS) that can scan the complete 360° area around it, within a 50 m range. It transmits acoustic pulses into the water and then measures the intensity of acoustic reflections over a period to detect objects in the water column. Each of these acoustic transmit and receive cycles is referred to as a 'ping'. The intensity of the reflection produced by an object depends on the density difference between the object and water; gas bubbles, rocks, and metal produce strong acoustic reflections [61].

The beam generated by a scanning sonar ensonifies a narrow area in front of the sonar, much like how a flashlight illuminates the dark. To scan a sector, the sonar mechanically rotates the transducer head which produces the beam [61]. An image of the scanned sector is then produced by compositing the individual 'slices' from each beam, as in Fig. 3.2. The data presented is from a test which was conducted at Lake Barrington, Tasmania, Australia. During the test, the Ping360 sonar was secured to the dock, and scanned an area containing a small micro-bubble plume created by a bubble generator; the test is described in detail in [62]. Warmer colours in the sector scan image (Fig. 3.2) indicate higher intensity acoustic reflections

Fig. 3.1: Ping360 Scanning Sonar

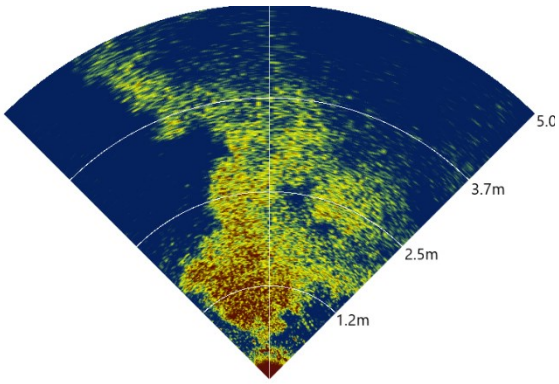


Fig. 3.2: Ping360 sector scan of a micro-bubble plume

The Ping360 is not calibrated by the manufacturer; it provides non-dimensional intensity measurements which do not correspond to any physical quantity, such as the acoustic backscatter intensity [63]. Following each ping, the Ping360 returns an array of intensities representing the strength of the acoustic reflections over the range of the beam. The Ping360 has a relatively slow ping rate, and takes 35 seconds to scan the complete 360° area around it when operating with a 50 m range [64]. As a result, it is more effective for the sonar to scan only a sector in front of the vehicle with a sweeping back-and-forth motion.

3.4 Algorithm Overview

The plume detection algorithm consists of the following five steps:

1. Range-gating: Removes high-intensity noise in the sonar data close to the transducer head.
2. Segmentation: Partitions the dataset into background data and positive detections. Intensity samples that exceed a threshold are considered positive detections.
3. Grid Conversion: Transforms the data storage format from a polar grid to a Cartesian grid. This conversion simplifies subsequent processing because when the data is stored

in the Cartesian grid format, the storage locations correspond to the sampling locations. The data is also down-sampled during the transformation to prepare for the computationally intensive clustering step.

4. Clustering: Identifies high-density groups of positive detections and categorizes them as clusters; detections in low-density areas are considered noise and are discarded. The output is the center and radius of each cluster.
5. Georeferencing: Computes the real-world coordinates of the cluster centers.

The algorithm's execution flow is illustrated in Fig. 3.3. The first two steps are applied to each beam return, while the remaining three steps require the whole sector scan. As a result, the range-gating and segmentation steps are executed as soon as the beam data from each ping is received. Execution stops following segmentation if the sonar is in the process of scanning a sector, and the segmented data is stored. Once the beam data from the port or starboard edge of the sector is received, all five steps of the algorithm are executed, and the whole sector scan is processed.

The algorithm's configurable parameters are identified in Fig. 3.3 using italics. These parameters control the algorithm's output and allow the algorithm to be configured to detect seep or spill plumes. While each of the following sections describes a step in the plume detection algorithm along with its' configurable parameter(s), directions on how to tune the parameters are provided in the following chapter. The tuning method is based on a visual analysis of how parameter changes affect the algorithm's output. It is important to note that the emphasis on automated detection, as opposed to autonomous detection, justifies the manual selection of the algorithm's parameters.

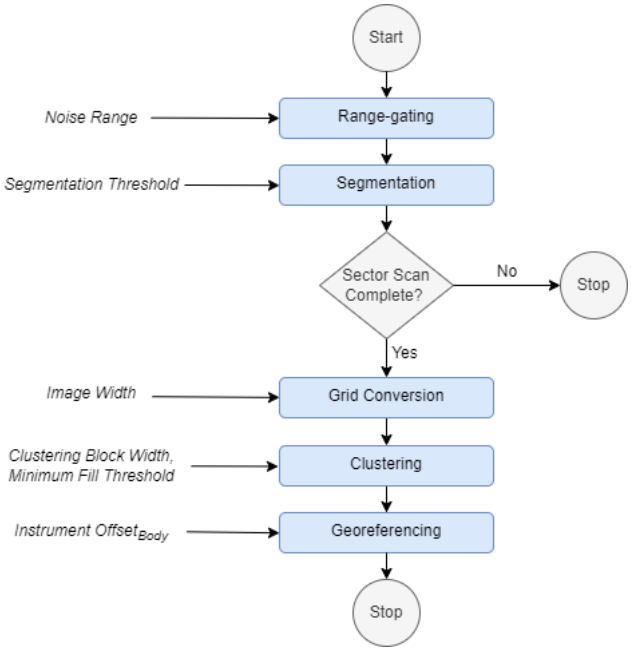


Fig. 3.3: Plume Detection Algorithm Flowchart. The algorithm's configurable parameters are italicized.

3.5 Range-Gating

Sonar data typically contains high intensity noise in the region close to the transducer head. The noise is visible at the bottom of Fig. 3.2 as a completely red region touching the sonar and extending approximately half a meter from it. This high intensity data is an artifact of the sonar's operation; when the transducer emits the acoustic pulse, it causes the sonar itself to vibrate, and these vibrations take some time to die out. The sonar subsequently picks up its own vibrations when it measures the acoustic reflections [65].

Range-gating removes the high-intensity noise close to the transducer head. More specifically, it zeroes intensity samples within a configurable *noise range* from the sonar. An appropriate value for the *noise range* can be determined based on a visual inspection of the sonar's acoustic return. If the sonar settings or the sonar itself is changed, a re-inspection of the data is required to determine if the *noise range* requires adjustment. The *noise range*

(R_{Noise}) is input to the algorithm in meters, and the following equation is utilized to determine the equivalent number of samples within the noise range:

$$N_{Noise} = \frac{R_{Noise}}{R_{Total}} \times N_{Total} \tag{3.1}$$

where:

N_{Noise} = the number of samples within the *noise range*

R_{Noise} = the *noise range*, measured meters

R_{Total} = the total ping range, measured in meters

N_{Total} = the total number of samples in the ping

The Ping360 data message does not directly indicate the total ping range (R_{Total}) referenced in (3.1) but provides parameters from which it can be calculated. Referencing the source code for the Ping360 Ping-Viewer software [65], R_{Total} is calculated as:

$$R_{Total} = \frac{25 \times 10^{-9}\ TcN_{Total}}{2} \tag{3.2}$$

where:

T = the sample period, measured in 25 nano-second increments

c = the speed of sound, measured in m/s

Fig. 3.4 contains a plot of the acoustic intensity data from a single beam containing 1200 samples. In the example, it is assumed that 120 samples (highlighted yellow) fall within the *noise range*; these samples are zeroed during the range-gating process and the resulting dataset is plotted in Fig. 3.5. Note that since the sonar is not calibrated, the acoustic intensity is a non-dimensional 8-bit measurement, which has a maximum value of 255.

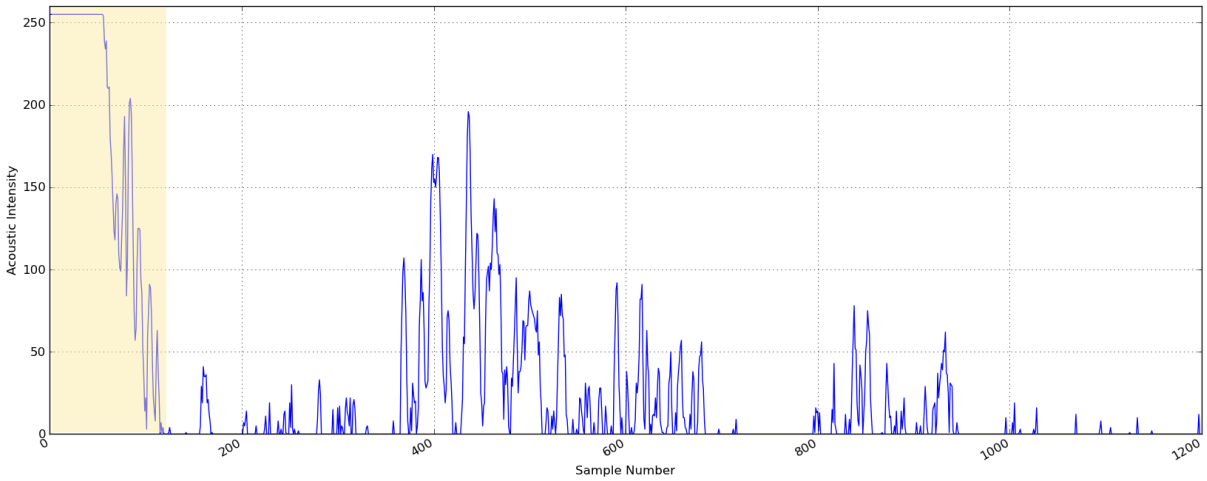


Fig. 3.4: Acoustic Intensity data from a single beam. The region within the *noise range* is highlighted yellow.

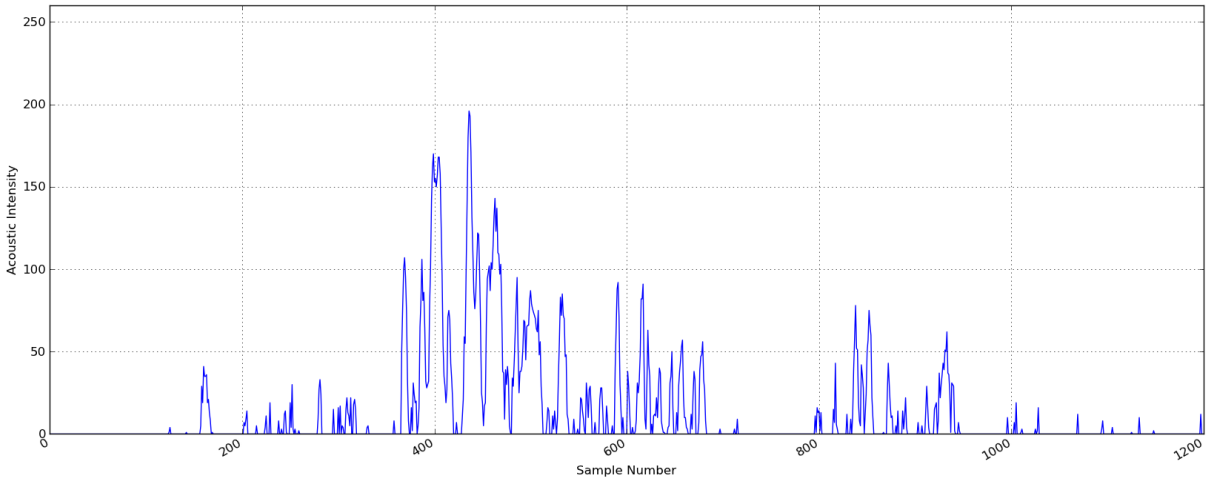


Fig. 3.5: Range-gated acoustic intensity data from a single beam.

3.6 Segmentation

Segmentation partitions the range-gated intensity samples into background data and positive detections; it produces a basic binary dataset which aids subsequent processing. A simple threshold is applied to each intensity measurement along the beam to segment the data (Fig. 3.6). Medium-high intensity measurements above the threshold are considered positive detections, and the corresponding segmented data point is set to '1'; low-intensity samples are indicated by a value of '0' in the segmented dataset (Fig. 3.7).

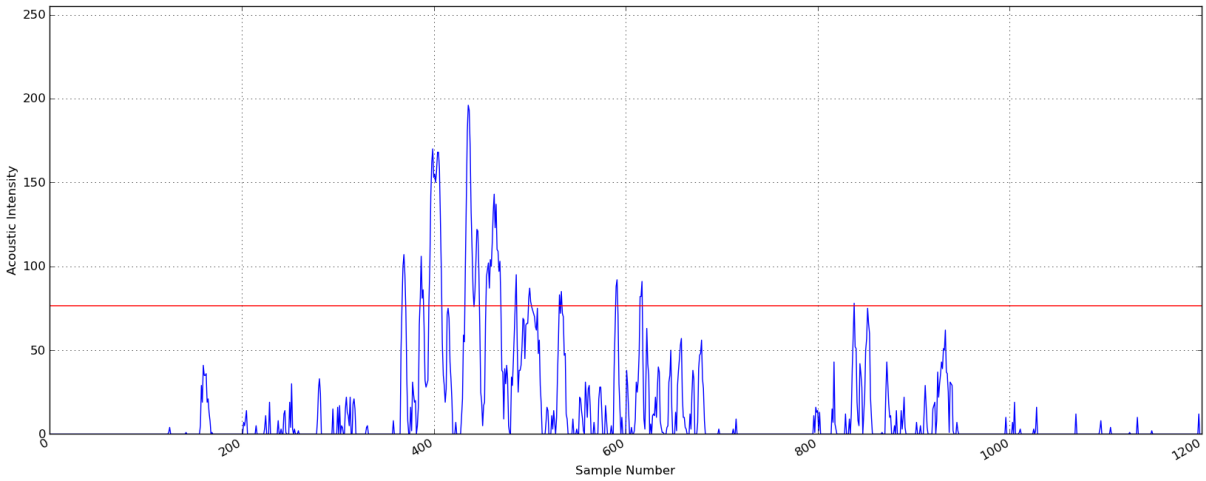


Fig. 3.6: Range-gated data for a single beam (blue) and the segmentation threshold (red).

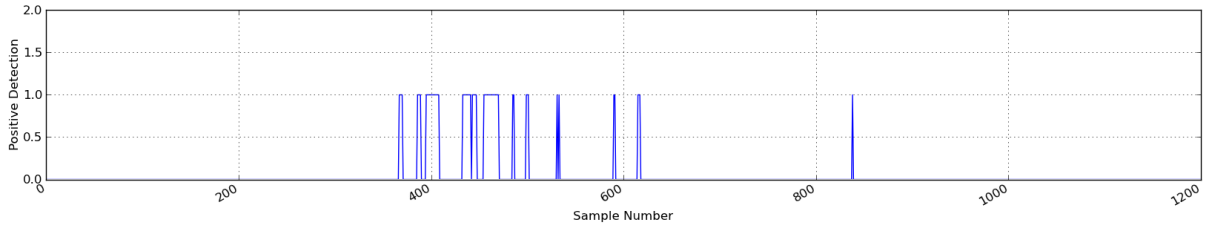


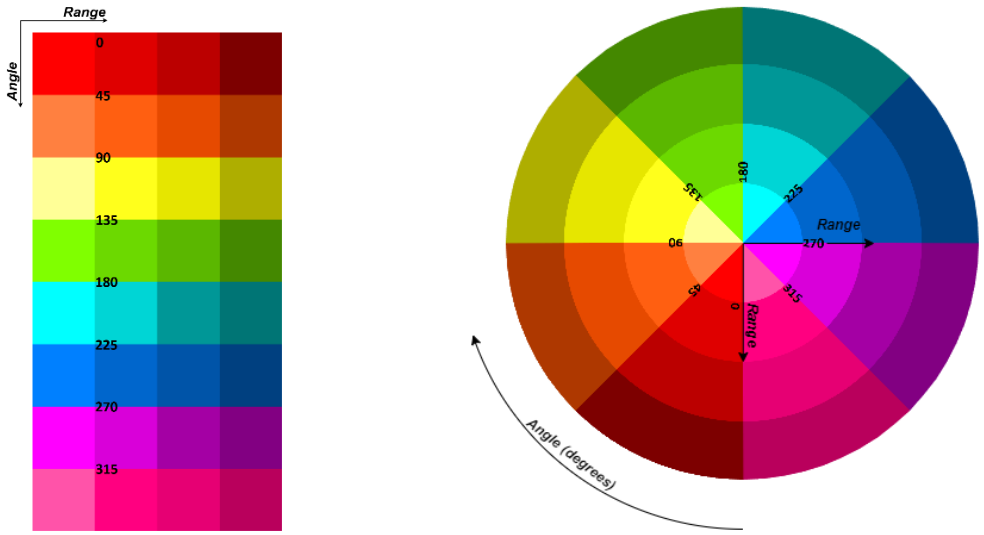
Fig. 3.7: Segmented data for a single beam.

The *segmentation threshold* is a configurable parameter and is specified as a percentage of the sonar's full-scale range (0-255). A threshold of 30% is utilized for the example in the figures above.

3.7 Grid Conversion

Sonar data is collected in polar coordinates, with a range of intensity values along each angular position of the transducer head (Fig. 3.9(a)). Following segmentation, the data is still stored in the polar format, and the grid conversion step maps it onto a Cartesian grid to facilitate real-time processing (Fig. 3.9(b)). The re-mapping is accomplished using an image transformation function which converts from polar to Cartesian coordinate space. Fig. 3.8 illustrates the transformation using a minimal dataset consisting of 8 beams with 4 samples along each beam;

each colour block represents an individual data sample. The input image (Fig. 3.8(a)) is the two-dimensional array of polar data in its storage format - with each beam's angular position along one axis, and the range from the sonar along the other. The input image is 'wrapped' around itself to produce an image that stores the data in Cartesian coordinates (Fig. 3.8(b)). This output image measures range along both axes, and the data storage locations correspond to the data sample locations in physical space. The output *image width* is a configurable parameter and should be selected based on the system's processing capabilities.



(a) Input image: data stored in polar coordinates

(b) Output image: data stored in Cartesian coordinates

Fig. 3.8: Input and output images of a polar to Cartesian coordinate space transformation.

More specifically, the OpenCV warpPolar function [66] was selected for the polar to Cartesian transformation. For each pixel in the output image, the warpPolar function computes the range and angle to identify the source location in the input image. Typically, this location is not a discrete value and lies between pixels. Accordingly, the function interpolates between pixels surrounding the source location to compute the output image pixel

value. Several interpolation options are available, and although nearest neighbour interpolation is the fastest, it also has the highest signal-to-noise ratio. The selected bilinear interpolation method is only marginally slower than nearest neighbour interpolation but produces an image with a significantly better signal to noise ratio [67].

The motivation for converting the data into an image with a Cartesian pixel grid is two-fold. First, the clustering process involves locating data points that lie within defined areas, and these operations are straightforward with a Cartesian grid. To illustrate this, Fig. 3.9(b) shows that the number of data points within a square area of a Cartesian grid is fixed, regardless of the location of the square. In contrast, a square area of a polar grid contains a variable number of data points, determined by the location of the square. This variability exists because the polar grid spacing increases with distance from the center. Thus, locating data points in an area of a polar grid is not a trivial operation, and converting to a Cartesian grid simplifies processing.

(b) Polar Grid (c) Cartesian Grid

Fig. 3.9: Finding data points within square areas in two types of grids

The second advantage of re-gridding is that it allows for simultaneous downsampling of the data before the computationally intensive clustering step. Downsampling is especially

important because sonars generate large amounts of data. For example, a single Ping360 sonar scan of a 90° sector can contain 120,000 samples (1200 samples/beam × 100 beams). In comparison, a 400x400 pixel image produced through the re-gridding process would contain around 31,000 data pixels; this reduces the dataset size by a factor of 4. Note that here, 'data pixels' refers to pixels in the image that lie in the sonar swath, whose values are derived from the sonar data. The area containing the data pixels for a 90° sonar swath is shown in black in Fig. 3.10. The number of data pixels provides an effective measure of the dataset size, as only these pixels have a significant impact on the clustering computations.

Thus, the pixel grid format of an image allows for simpler and faster processing at the clustering stage, thereby enabling real-time processing.

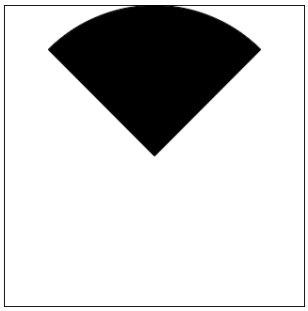


Fig. 3.10: Image data pixels for a 90° swath. The data pixels are shown in black.

3.8 Clustering

3.8.1 Introduction

The objective of the clustering process is to identify high-density groups of positive detections in the segmented image, and simultaneously filter out noise. The scikit library [68] provides several options for clustering algorithms, but only a few discard outliers as noise. Among these, the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm is the

fastest [69]. DBSCAN's worst-case $O(n^2)$ time complexity, however, makes it unsuitable for real-time applications, and a custom 'block clustering' algorithm is consequently developed. Nevertheless, this clustering section also focuses on DBSCAN as it is the benchmark against which the block clustering algorithm will be compared.

Section 3.8.2 defines the plume detector's clustering interface, while Section 3.8.3 provides an overview of DBSCAN and describes how it can be configured to match the defined interface. Section 3.8.4 describes the scenario which produces DBSCAN's worst-case time complexity and explains why workarounds are not suitable for this application. Thus, the motivation for a faster and more stable clustering algorithm is provided, and a customized block clustering algorithm is described in Section 3.8.5. Finally, Section 0 presents a comparison of the two clustering algorithms.

3.8.2 Plume Detector Clustering Interface

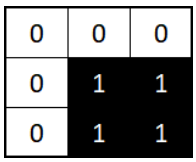
The clustering interface defined here establishes a set of rules to determine which clustering algorithms can be utilized for the plume detector. More specifically, any algorithm which identifies cluster pixels based on the following definitions is suitable:

1. Positive detection pixel: A pixel in the segmented image that has a value of '1' (i.e. a medium/high acoustic intensity point).
2. Clustering block: A square subset of pixels in the segmented image, centered on a positive detection pixel.
3. High-density block: A clustering block in which the number of positive detection pixels exceeds a *minimum fill threshold*.
4. Cluster pixels: Pixels in the segmented image that lie in high-density blocks.

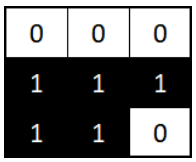
The plume detector requires two input parameters for the clustering algorithm:

1. The *clustering block width*, which determines the size of the clustering block. It is input to the algorithm in meters and converted to pixels based on the number of pixels per meter in the input image.
2. The *minimum fill threshold*, which determines whether a clustering block is a high-density block. It is input to the algorithm as a percentage, allowing it to be specified without any dependence on the size of the clustering block. It is converted to a pixel value through multiplication with the total number of pixels in the clustering block.

The definitions above are illustrated in Fig. 3.11, where each cell in the images represents a pixel, and the black pixels are positive detection pixels. Each image represents a 3x3 pixel clustering block (i.e. the *clustering block width* is set to 3 pixels). With the *minimum fill threshold* set to 50% (4.5 pixels), the first clustering block containing 4 black pixels (Fig. 3.11(a)) is not a high-density block, while the second clustering block containing 5 black pixels (Fig. 3.11(b)) is a high-density block.



(a) Low-density clustering block



(b) High-density clustering block

Fig. 3.11: Illustration of the two types of clustering blocks. A 3-pixel *clustering block width* and 50% *minimum fill threshold* are utilized.

3.8.3 Clustering with DBSCAN

DBSCAN identifies clusters as high-density groups of points separated by areas of lower density and can be configured to match the plume detector's clustering interface.

DBSCAN refers to the area around a point as its 'neighbourhood'. The shape of the neighbourhood is defined by the selected distance metric, and the frequently utilized Euclidean distance produces a ball-shaped neighbourhood [70] (Fig. 3.12(a)). The Chebyshev distance, however, is selected here as it produces a square neighbourhood that matches the shape of the plume detector's clustering block (Fig. 3.12(b)). Formally, the Chebyshev distance is the maximum distance measured over any of the axes. It is also known as the "chessboard" distance, as it corresponds to the minimum number of moves for a chess king to go from one square to another [71] (Fig. 3.13).

(a) Euclidean Neighbourhood (b) Chebyshev Neighbourhood

Fig. 3.12: Neighbourhood shapes produced by two different distance metrics

(a) Euclidean distance (b) Chebyshev distance

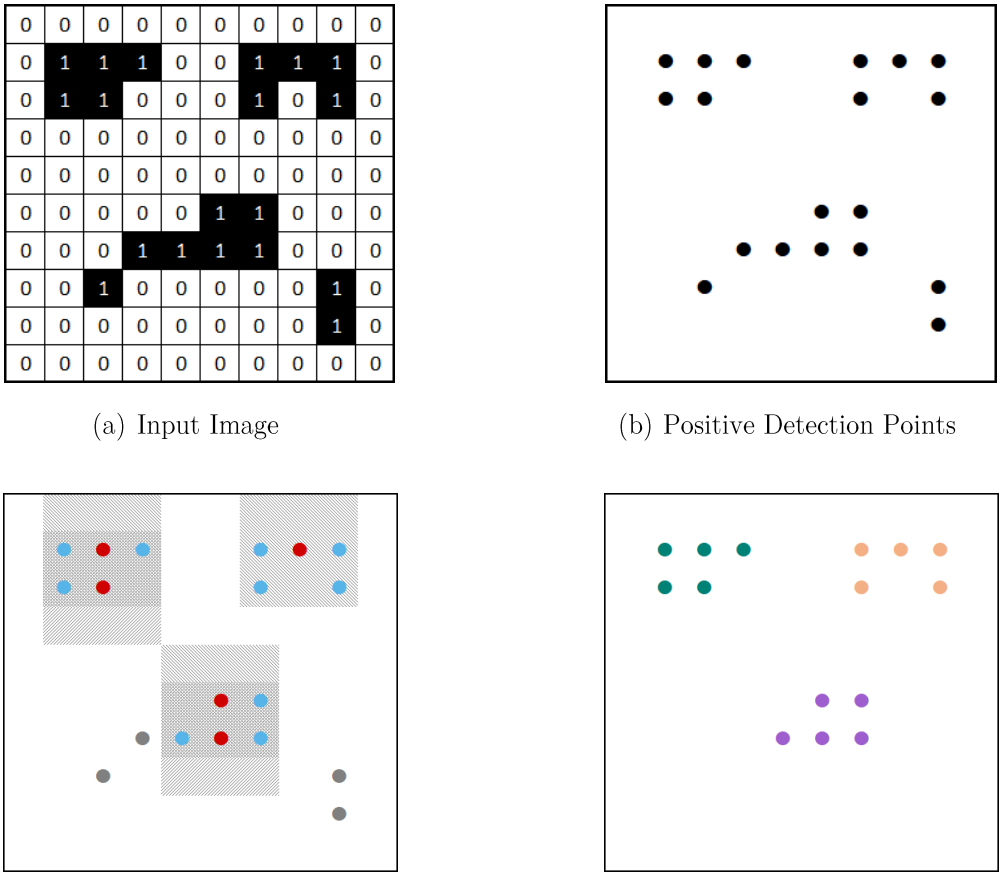
Fig. 3.13: Euclidean and Chebyshev distance measurements

The DBSCAN algorithm takes two parameters.

1. *Epsilon* (ε): A range value defining the size of the neighbourhood around a point. It is measured from the point to the edge of the neighbourhood.
2. *MinPts* : the minimum number of points in a neighbourhood that is required for the formation of a cluster.

DBSCAN defines clusters based on 'core points', which are points that have more than *MinPts* neighbours in their ε neighbourhood. Core points that are within an ε range from each other belong to the same cluster, and all points that are within an ε range of a core point belong to the core point's cluster. Points that are not within an ε range of a core point are considered 'noise' [70].

Fig. 3.14 illustrates the plume detector's clustering process, with DBSCAN as the selected clustering algorithm. The 10x10 pixel segmented image which is the input to the clustering step is shown in Fig. 3.14(a). DBSCAN, however, cannot operate directly on the image and requires the data to be provided as a list of coordinates. Thus, the input to the DBSCAN algorithm is the center coordinates of the positive detection pixels (Fig. 3.14 (b)). Fig. 3.14(c) shows DBSCAN's classification of the data points with ε set to 1.5 pixels and *MinPts* set to 4. The red points are core points and have at least 4 points in their neighbourhood. The blue points are cluster points and are within the neighbourhood of core points. The grey points are noise points and are outside the neighbourhood of the core points. Areas with a diagonal fill indicate the ε neighbourhood of core points; overlapping neighbourhoods have a darker cross-hatch fill. The last image in Fig. 3.14(d) identifies the points in the three resulting clusters with unique colours. The green and purple clusters contain two core points within an ε range of each other, while the orange cluster contains one core point.



(a) Input Image

(b) Positive Detection Points

(c) DBSCAN Classification of Points

(d) DBSCAN Clusters

Fig. 3.14: The plume detector's clustering process with DBSCAN as the selected clustering algorithm

Thus, DBSCAN can be configured to match the plume detector's clustering interface using the Chebyshev distance metric. Only the terminology is slightly different: DBSCAN's ε is half the *clustering block width*; DBSCAN's *MinPts* parameter corresponds to the *minimum fill threshold*; and DBSCAN's core pixels' neighbourhoods correspond to the high-density blocks. Consequently, DBSCAN's cluster points which lie in the core points' neighbourhoods correspond to the plume detector's cluster pixels which lie in the high-density blocks.

3.8.4 The Motivation for a Novel Clustering Algorithm

This section illustrates why DBSCAN is not well suited for this real-time image processing application and thereby provides the motivation for a novel clustering algorithm. The concept

of cluster degeneration is also introduced; it plays an important role in the following chapter, where the performance tests create the worst-case scenario by causing the clusters to degenerate.

Generic implementations of the DBSCAN algorithm have a worst-case $O(n^2)$ time complexity. Improving DBSCAN's time complexity is a large and active area of research because the computation time becomes infeasible for large datasets [72]. The running time of DBSCAN depends on the number of invocations of the range query function, which determines which points are in the neighbourhood of a given point [73]. As a result, DBSCAN's runtime increases significantly with larger values of ε, which produce larger neighbourhoods and consequently require searches through greater portions of the dataset. As ε is increased, fewer and fewer clusters are formed, and the worst-case scenario occurs when all or most of the data points are contained within a single cluster; this is referred to as cluster 'degeneration' [74].

In most applications, degenerated clusters are not very useful, and the original developers of DBSCAN suggest tuning the ε parameter based on the data to avoid degenerate clusters [74]. However, this is not a viable approach in this application for several reasons. First, since the data is not known beforehand, the algorithm should be robust enough to work on a wide range of datasets, rather than require tuning based on the data. Second, approaches that require an operator's oversight are not suitable for real-time automated applications. Finally, not only is the formation of a single cluster a valid outcome in this application, but it can be expected in the presence of a large spill plume.

The worst-case $O(n^2)$ time complexity applies to generic implementations of the DBSCAN algorithm, which are not constrained by the dimensionality of the data. If the data is 2-dimensional, as it is here, a faster grid-based DBSCAN algorithm can be utilized; it simplifies

the range query by organizing the data into a grid and has an improved *O(nlogn)* time complexity [73]. However, the algorithm is relatively complex, and implementations of it are not readily available. Furthermore, since the data to be clustered here is already stored in a grid format, the initial steps of the algorithm that partition the data into a grid would be redundant. A novel clustering algorithm that operates directly on the image data is consequently developed in the following section.

3.8.5 A Novel Block Clustering Algorithm

The proposed block clustering algorithm combines density-based clustering and image processing methods. It is based on Blomberg et. al's method for automatic detection of gas seeps using interferometric side-scan sonars [48]; they consider a potential seep pixel to be a positive detection if three of the four nearest neighbouring pixels are also potential seep pixels. However, like Hwang et. al's spill plume tracking algorithm [41], the block clustering algorithm is designed to look at a larger area as spill plumes have a larger acoustic footprint.

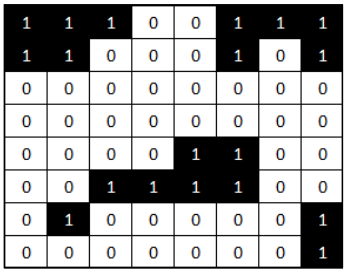
The block clustering algorithm builds upon the plume detector's interface outlined in Section 3.8.2. It defines the following terms, which determine how the clusters are formed:

1. Cluster Region: A set of connected high-density blocks. It determines the spatial extent of a cluster.
2. Cluster: A high-density group of positive detection pixels that lie within a single cluster region.

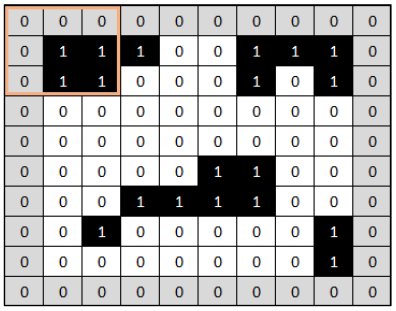
The algorithm's final output is the center coordinates and radius of each cluster. Fig. 3.15 provides a graphical introduction to the block clustering algorithm through images that are created during the clustering process for a small 8x8 pixel image. In this example, the

clustering block width is set to 3 pixels, and the *minimum fill threshold* is set to 50%. The following is a summary of each image:

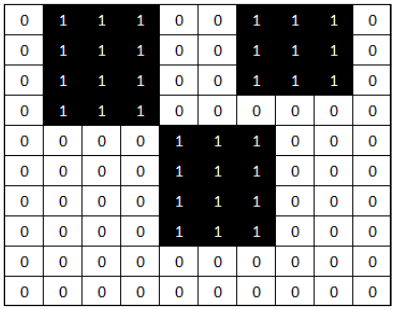
1. Input Image (Fig. 3.15(a)): The image created during the preceding grid conversion step, in which positive detection pixels identify medium-high intensity acoustic returns.
2. Padded Input Image (Fig. 3.15(b)): The input image, padded with a zero-valued pixel border. It is created to aid subsequent processing.
3. High-Density Blocks Image (Fig. 3.15(c)): Identifies the high-density blocks in the padded input image. Recall that a high-density block is a clustering block in which the number of positive detection pixels exceeds the *minimum fill threshold*.
4. Cluster Regions Image (Fig. 3.15(d)): Identifies sets of connected high-density blocks as cluster regions. The pixels in each cluster region are labelled with a unique pixel value and are also displayed with a unique colour.
5. Clusters Image (Fig. 3.15(e)): Identifies the cluster pixels, which are the positive detection pixels that lie within the high-density blocks. Their pixel values correspond to the cluster region in which they lie, and consequently identify the cluster to which they belong.



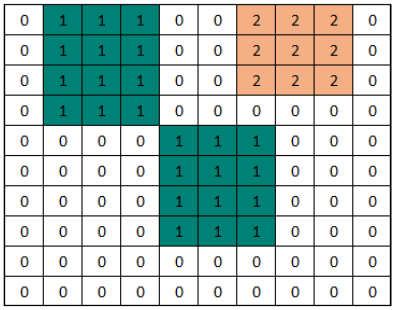
(a) Input Image



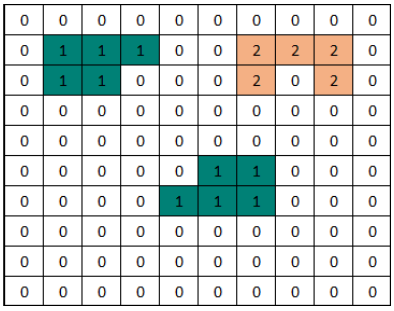
(b) Padded Input Image



(c) High-Density Blocks Image



(d) Cluster Regions Image



(e) Clusters Image

Fig. 3.15: Images created during the clustering process for a small 8x8 pixel image, when using a 3x3 pixel clustering block and 50% *minimum fill threshold*. In (b) the border padding cells are indicated with a grey background, and the clustering block for the first positive detection pixel is highlighted with an orange border.

The details of each step in the clustering process are as follows:

1. Calculate *clustering block width* in pixels.

The *clustering block width* is input to the algorithm in meters and converted to pixels based on the number of pixels per meter in the input image. This calculated floating-point block

width must then be converted to an integer value to ensure that the clustering block encompasses a discrete number of pixels. Additionally, selecting an odd-valued block width allows for the block to be centered on the selected positive detection pixel, with an equal number of pixels on either side of it. Thus, the calculated floating-point block width is rounded up to the nearest odd integer.

2. Created Padded Input Image.

The padded input image is created to ensure that clustering blocks centered on the positive detection pixels at the edge of the input image can be extracted. For example, it is not possible to extract a 3x3 pixel block centered on the positive detection pixel at the top left corner of the input image in Fig. 3.15(a). To address this issue, a zero-valued pixel border is added to the input image and is referred to as the 'border padding'. Within the created padded input image, clustering blocks for all the positive detection pixels can be identified. The clustering block for the first positive detection pixel is highlighted with an orange border in (Fig. 3.15(b)).

The width of the border padding is equal to the number of pixels on one side of the central positive detection pixel in the clustering block. It is calculated as half the *clustering block width*, rounded down to the nearest integer. In the padded input image (Fig. 3.15(b)), the border pixels are highlighted with a grey background; the 1-pixel border width is computed based on the 3-pixel *clustering block width*. This step of adding border padding reflects a typical image filtering workflow [75], which involves operations similar to the clustering block evaluation.

3. Create High-Density Blocks Image.

The high-density blocks image (Fig. 3.15(c)) identifies regions corresponding to high-density blocks in the padded input image with a pixel value of '1'. In other words, when it is overlaid on the padded input image, it identifies all the clustering blocks in which the number of positive detection pixels exceeds the *minimum fill threshold* (Fig. 3.16). The high-density blocks image is created as a first step towards identifying the cluster regions.

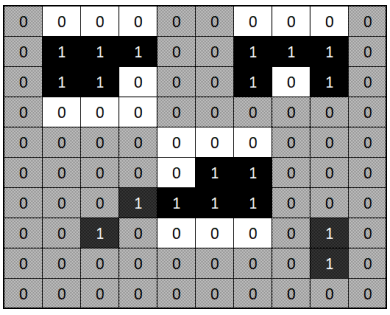
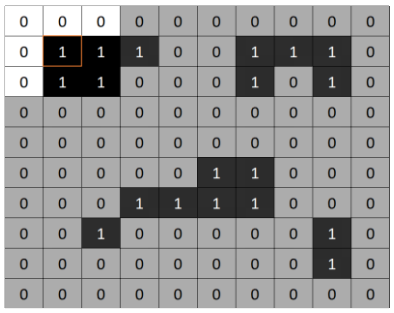


Fig. 3.16: The padded input image overlaid with the high-density blocks image; pixels that do not lie within the high-density blocks are greyed out.

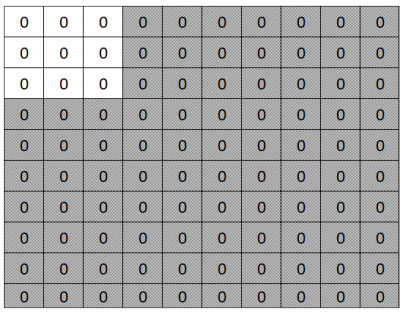
The high-density blocks image is initialized to a two-dimensional array of zeroes the same size as the padded input image. To identify the high-density blocks, the following evaluation is carried out for each positive detection pixel in the padded input image: first, the clustering block centered on the selected positive detection pixel is identified; if it is a high-density block, the pixels within the same set of coordinates in the high-density block image are then set to '1'.

Fig. 3.17 illustrates the high-density blocks image creation process for the first two positive detection pixels. In Fig. 3.17(a) the first positive detection pixel is highlighted with an orange border, and all the pixels except for those in the clustering block around it are greyed out. The block contains 4 positive detection pixels, which is less than the *minimum fill threshold* of 4.5

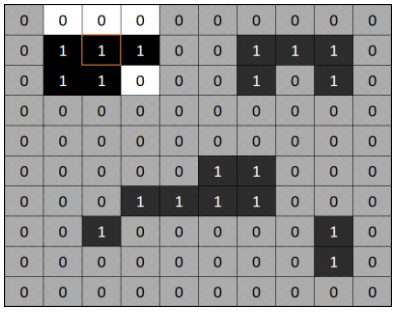
pixels. Thus, the clustering block for the first positive detection pixel is not a high-density block. In Fig. 3.17(b), the corresponding pixels in the high-density blocks image are indicated with a white background; their pixel values are left at '0'. Fig. 3.17(c) indicates the clustering block for the second positive detection pixel. This is a high-density block as there are 5 positive detection pixels, which exceeds the 4.5 pixels threshold. Consequently, the corresponding pixels in the high-density blocks image are set to '1' (Fig. 3.17(d)).



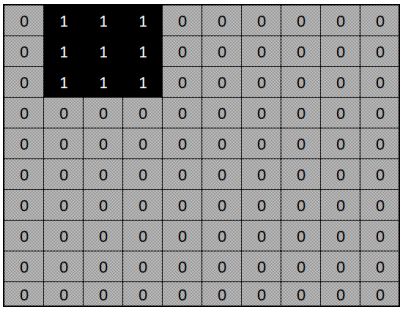
(a) Clustering block for the first positive detection pixel



(b) High-density blocks image after the first clustering block evaluation



(c) Clustering block for the second positive detection pixel



(d) High-density blocks image after the second clustering block evaluation

Fig. 3.17: Illustration of the high-density blocks image creation process for the first two positive detection pixels. In each image, the pixels that do not lie within the selected clustering block are greyed out.

4. Create Cluster Regions Image.

The cluster regions image (Fig. 3.15(d)) identifies the pixels in each cluster region with a unique pixel value. A cluster region is formed by a set of connected high-density blocks and defines the spatial extent of a cluster. Here, two pixels are considered 'connected' if any of their edges or corners are touching. Fig. 3.18 provides a visual example of connectivity; the central black pixel could be connected to any of the eight grey pixels surrounding it.

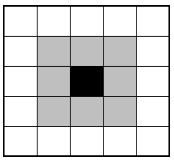


Fig. 3.18: Illustration of connectivity - the black pixel could be connected to any of the eight grey pixels surrounding it.

The problem of uniquely labelling sets of connected regions in images has been well researched. Algorithms that solve the problem are referred to as connected-component labelling algorithms, and implementations are readily available in any image processing library. Accordingly, the high-density blocks image is input to the 'label' function from the scikit-image library [76] to generate the cluster regions image.

5. Create Clusters Image.

The clusters image (Fig. 3.15(e)) contains the cluster pixels, labelled with the pixel value corresponding to the region in which they lie. The cluster pixels are the positive detection pixels that lie within the high-density blocks.

The clusters image is created by multiplying each pixel in the padded input image with the corresponding pixel in the cluster regions image. In mathematical terms, this is an element-wise multiplication of two image arrays. The multiplication is effectively a masking process, as

illustrated in Fig. 3.19. The padded input image is first visualized as a mask in Fig. 3.19(a) by inverting the pixel colours. When this mask is overlaid on the cluster regions image (Fig. 3.19(b)), the cluster pixels are identified as those with non-zero pixel values. Thus, the padded input image is masked with the cluster regions image to retain and label the cluster pixels.

(a) The padded input image, visualized as a mask

(b) The padded input image mask overlaid on the cluster regions image

Fig. 3.19: Generation of the clusters image, visualized as a masking process

6. Calculate cluster centers and sizes.

The block clustering algorithm's final output is the center and radius of each cluster, measured in pixels with respect to the image frame (see Fig. 3.21 for an illustration of the image frame). The center of each cluster is calculated as the average of the x and y coordinates of the n pixels in the cluster.

$$Center_x = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.3}$$

$$Center_y = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{3.4}$$

The radius of the cluster is the Euclidean distance between the center of the cluster and the cluster pixel which is furthest away from it:

$$Cluster\ Radius = Max(\sqrt{\ Center_x - x_i\ ^2 + (Center_y - y_i)^2} \tag{3.5}$$

The block clustering algorithm is summarized as pseudo code in Algorithm 1.

Algorithm 1 *BlockCluster*($input\_img$, $range_m$, $block\_width_m$,$min\_fill_{percent}$)

{ $input\_img$ identifies the positive detections }

{ $range_m$ is the sonar's configured range, measured in meters }

{ $block\_width_m$ is the clustering block width, measured in meters }

{ $min\_fill_{percent}$ is the minimum fill threshold for a high-density block }

$block\_width_{pixels} \leftarrow block\_width_m \times \dfrac{width(input\_img)_{pixels}}{2 \times range_m}$

$block\_width_{pixels} \leftarrow 2 \times floor\left(\dfrac{block\_width_{pixels}}{2}\right) + 1$

$padding \leftarrow floor\left(\dfrac{block\_width_{pixels}}{2}\right)$

$min\_fill_{pixels} \leftarrow block\_width_{pixels}{}^2 \times \dfrac{min\_fill\_percent}{100}$

$padded\_img \leftarrow CopyMakeBorder(input\_img, border\_width = padding, border\_value = 0)$

Initialize *hd_blocks* to 2D array of zeroes the same size as *padded_img*

for each col x in *padded_img* $\notin$ border padding **do**

  for each row y in *padded_img* $\notin$ border padding **do**

    $selected\_pixel \leftarrow$ pixel in *padded_img* at (x, y)

    if *selected_pixel* is a positive detection **then**

      $lower\_left \leftarrow (x - padding,\ y - padding)$

      $upper\_right \leftarrow (x + padding,\ y + padding)$

      $clustering\_block \leftarrow$ pixels in *padded_img* from $lower\_left$ to $upper\_right$

      $fill \leftarrow sum(clustering\_block)$

      if $fill > min\_fill_{pixels}$ **then**

        for each pixel p in *hd_blocks* from $lower\_left$ to $upper\_right$ **do**

          $p \leftarrow 1$

Initialize *cluster_regions* and *clusters* to 2D array of zeroes the same size as *padded_img*

$cluster\_regions \leftarrow LabelledConnectedComponents(hd\_blocks)$

$clusters \leftarrow padded\_img \circ clusters$

for each cluster i in *clusters* **do**

  $center_i \leftarrow Average$(coordinates of pixels $\in$ cluster i)

  $radius_i \leftarrow Max(Range(center_i,$ coordinates of pixels $\in$ cluster $i))$

return $(center_i, radius_i)$ for each cluster i in *clusters*

3.8.6 Clustering Comparison

The plume detector's clustering definitions do not specify which pixels should belong to the same cluster. As a result, while DBSCAN and the block clustering algorithm identify the same cluster pixels, they group these pixels into clusters differently (see Fig. 3.20). DBSCAN uses the core points to define the extent of each cluster, whereas the block clustering algorithm uses the high-density blocks for this purpose. For example, the block clustering algorithm groups the green pixels in Fig. 3.20(a) into a single cluster because the high-density blocks in which they are contained are connected. However, these pixels form two different clusters in the DBSCAN output Fig. 3.20(b) as the core points in the green cluster are not within the ε range of the core points in the purple cluster. In other words, while DBSCAN requires significant overlap in the high-density blocks to merge clusters, the block clustering algorithm does not require any overlap, only connectivity, of the high-density blocks. The block clustering algorithm consequently produces a better clustering output for this application; this will be shown through the data-based clustering output comparison in Section 4.4.1.

0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	2	2	2	0
0	1	1	0	0	0	2	0	2	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

(a) Block Clustering Output

0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	2	2	2	0
0	1	1	0	0	0	2	0	2	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0
0	0	0	0	3	3	3	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

(b) DBSCAN Clustering Output

Fig. 3.20: Clustering Output Comparison

The processing method of the two algorithms is also fundamentally different. When the segmented image data is input to DBSCAN as a list of coordinates, spatial information

regarding the location of the positive detection pixels with respect to each other is lost. DBSCAN's range query is consequently an expensive operation that involves computing distances between coordinates to identify a point's neighbours. The block clustering algorithm, however, is designed specifically for 2D image data and operates directly on the segmented image. As a result, it can take advantage of the fact that the data storage locations correspond to the real-world locations. Its range query is consequently trivial; it only involves identifying the pixels that are stored around the selected pixel. Furthermore, the block clustering algorithm leverages existing tools for the connected-component-labelling problem, which has already been optimized and is solvable in $O(n)$ time [77]. As a result, the block clustering algorithm proves to be consistently faster than DBSCAN during the computation time test in Section 4.4.2.

3.9 Georeferencing

To determine the real-world location of the cluster centers, the cluster center coordinates are georeferenced using three successive transformations. The reference frames utilized in these transformations are illustrated in Fig. 3.21. The local frame's origin is located at a geographic coordinate in the mission area; the frame's x-axis points east while the y-axis points north. The body frame is located at the center of the AUV's Inertial Navigation System, while the instrument frame is located at the center of the Ping360 sonar. The image frame is centered on the top left pixel in the clusters image. The x-axes of the body, instrument, and image frames are parallel to the vehicle's longitudinal axis and rotate with the vehicle.

The clustering algorithm outputs the center coordinates of each cluster measured in the image frame. Thus, the georeferencing algorithm starts with the cluster center coordinate in

the image frame of reference, and first transforms to the instrument frame, next to the body frame, and finally to the local frame. The algorithm assumes that the vehicle is operating and collecting data at a fixed depth, and consequently only computes the 2D transformations.

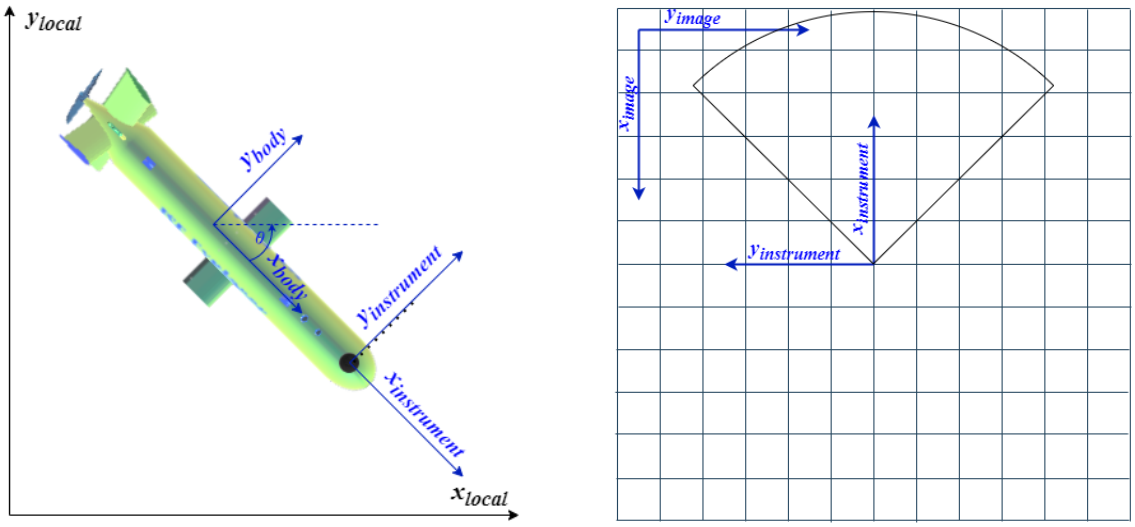


Fig. 3.21: The local, body, instrument, and image reference frames

Given the cluster center coordinates in the image frame ($Cluster_{Image}$), the equivalent coordinate in the instrument frame ($Cluster_{Inst}$) is computed using (3.6). The same transformation is expressed in the expanded form in (3.7), where all variable values are measured in pixels. The 180° rotation inverts the axes' directions, while the added offset is the location of the image frame origin in the instrument frame ($Image\ Origin_{Inst}$):

$$Cluster_{Inst} = R^{-1}(180°) \times Cluster_{Image} + Image\ Origin_{Inst} \tag{3.6}$$

$$\begin{bmatrix} cluster_{x,inst} \\ cluster_{y,inst} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} cluster_{x,image} \\ cluster_{y,image} \end{bmatrix} + \begin{bmatrix} \frac{image\ width\ -1}{2} \\ \frac{image\ height\ -1}{2} \end{bmatrix} \tag{3.7}$$

The transformation of the cluster coordinates from the instrument frame to the body frame ($Cluster_{Body}$) involves only scaling and translation operations as the axes of the two frames are parallel to each other (3.8), (3.9). Since the cluster coordinates in the instrument frame ($Cluster_{Inst}$) is measured in pixels, it is first multiplied by the meters per pixel ratio (k) to convert the units to meters. The location of the sonar, measured in meters with respect to the body reference frame ($Instrument\ Offset_{Body}$) is then added as an offset:

$$Cluster_{Body} = k \times Cluster_{Inst} + Instrument\ Offset_{Body} \tag{3.8}$$

$$\begin{bmatrix} cluster_{x,body} \\ cluster_{y,body} \end{bmatrix} = k \begin{bmatrix} cluster_{x,inst} \\ cluster_{y,inst} \end{bmatrix} + \begin{bmatrix} instrument\ offset_{x,body} \\ instrument\ offset_{y,body} \end{bmatrix} \tag{3.9}$$

$$k = \frac{2 \times sonar\ range}{image\ width} \tag{3.10}$$

Finally, the position of the cluster center in the local frame ($Cluster_{Local}$) is computed. The transformation from the body frame to the local frame consists of a rotation by θ, and offset by the position of the AUV in the local frame (AUV_{Local}):

$$Cluster_{Local} = R^{-1}(\theta) \times Cluster_{Body} + AUV_{Local} \tag{3.11}$$

$$\begin{bmatrix} cluster_{x,local} \\ cluster_{y,local} \end{bmatrix} = \begin{bmatrix} cos\,\theta & sin\,\theta \\ -sin\,\theta & cos\,\theta \end{bmatrix} \begin{bmatrix} cluster_{x,body} \\ cluster_{y,body} \end{bmatrix} + \begin{bmatrix} auv_{x,local} \\ auv_{y,local} \end{bmatrix} \tag{3.12}$$

$$\theta = 90° - AUV\ Heading \tag{3.13}$$

Since the AUV is simultaneously travelling and collecting data, it is in a different location when the data for each beam is collected. Accordingly, the (AUV_{Local}) position in (3.11) is

defined as the location of the AUV when the cluster center was scanned. During operations, the beam angle and the AUV's location at the time of the ping are recorded together each time the acoustic return for a beam is received. The georeferencing algorithm utilizes this data when determining (AUV_{Local}); it first computes the beam angle by converting the ($Cluster_{Inst}$) coordinate to polar form, and then looks up the AUV position recorded with the computed beam angle.

In addition to the cluster center coordinates, the cluster radius is also output. The pixel value produced by the clustering algorithm is converted to meters following the equation below:

$$Cluster\ Radius_{meters} = k \times Cluster\ Radius_{pixels} \tag{3.14}$$

3.10 Conclusion

This chapter has presented a novel plume detection algorithm which identifies the plume as high-density clusters in the sonar data. The algorithm is designed for the Ping360 sonar, with an emphasis on real-time performance and configurability. It consists of five steps: range-gating, segmentation, grid conversion, clustering, and georeferencing. The algorithm can be configured to detect both seep and spill plumes, and its output is determined by the input parameters which are summarized in Table 3.1. Detailed directions on how to tune these parameters are provided in the following chapter.

A block clustering algorithm designed specifically for the segmented image data has also been developed. It identifies high-density blocks in the input image and labels a set of connected blocks as a cluster region. The cluster pixels are then pixels in the input image

which lie in the high-density blocks and are labelled based on the region in which they lie. The block clustering algorithm leverages the fact that the locations of the pixels in the input image directly correspond to the real-world sampling locations. As a result, while processing the range query is complex in DBSCAN implementations, it is trivial for the block clustering algorithm.

Table 3.1: Summary of the Plume Detection Algorithm Parameters

Algorithm Step	**Parameter**	**Description**
Range-Gating	*Noise Range*	Range within which acoustic intensity samples are zeroed due to the presence of high-intensity noise.
Segmentation	*Segmentation Threshold*	Intensity samples above this threshold are considered positive detections. A lower threshold increases the sensitivity of the plume detector, but also makes it more susceptible to noise.
Grid Conversion	*Image Width*	Width of the image that is created by the grid conversion process. A larger width produces a higher resolution image which allows the plume detector to detect smaller targets, but also increases the processing time.
Clustering	*Clustering Block Width*	Defines the size of the square neighbourhood around a pixel. A smaller clustering block width allows the plume detector to detect smaller targets, such as seep plumes, but also makes it less resilient to noise.
	Minimum Fill Threshold	The minimum fill required for a clustering block to be considered a high-density block. Lower settings for the minimum fill threshold allow the plume detector to identify less dense targets.
Georeferencing	*Instrument* $Offset_{Body}$	Distance from the the origin of the body frame to the sonar, measured in the body frame of reference.

Chapter 4

Plume Detector Performance

4.1 Introduction

In this chapter, the performance of the plume detection algorithm is evaluated using data collected by Memorial University's Explorer AUV in Holyrood Bay, Newfoundland, Canada. The Ping360 sonar data was collected during field trials for a spill plume delineation algorithm [41] and has also proven to be useful for testing the algorithm presented here. Section 4.2 describes the test equipment and setup, as well as the software applications utilized for the experiment and data playback test. The plume detection results from the playback test are then presented in Section 4.3. The final analysis in Section 4.4 consists primarily of a performance comparison between DBSCAN and the block clustering algorithm.

4.2 Experiment Setup

4.2.1 Explorer AUV Navigation and Control

Memorial University's Explorer (Fig. 4.1) is a large survey class AUV measuring 5.3m in length and capable of running 16-hour missions. Its primary navigation instrument is the IXSEA PHINS III, which serves as an Inertial Navigation System (INS) and integrates

measurements from other sensors. In a submerged GPS-denied environment, the PHINS is primarily aided by the Paroscientific depth transducer. During the field trials, the AUV was also equipped with an RDI Workhorse 600 kHz Doppler Velocity Log (DVL) which provides additional bottom tracking when the vehicle is within 90m from the bottom. When operating the Explorer in the shallow waters of Holyrood Bay, the DVL typically maintains a constant bottom lock, allowing the PHINS to compute a more reliable estimate of the vehicle's position. This estimate can then be used to georeference the sensor measurements. The Explorer also has a single beam sonar mounted in the nose cone, which it uses for bottom avoidance.

Fig. 4.1: Memorial University's Explorer AUV

Fig. 4.2: The Nikuni KTM65S2 bubble generator setup at the Holyrood wharf

The Explorer's Vehicle Control Computer (VCC) interfaces with the navigation instruments and directly controls the vehicle's trajectory by actuating the planes and thruster. While the VCC executes pre-planned missions, adaptive missions are the domain of the onboard Payload Control Computer (PCC). The PCC is the 'backseat driver' [78] and realizes autonomous control using the MOOS-IvP (Mission Oriented Operating Suite - Interval Programming) software suite. The MOOS-Ivp software suite contains tools for communication, control, and mission planning [79]. Using these tools, the PCC can interface with mission-specific payload

sensors and can adaptively respond to the sensor data in real-time to autonomously control the mission. Accordingly, a Ping360 scanning sonar was mounted on the Explorer AUV and configured to interface with the PCC.

4.2.2 Micro-bubble Plume

A micro-air-bubble plume was utilized as an acoustically similar, environmentally friendly proxy for oil droplets and methane bubbles. A previous study identified micro-air-bubble plumes as a viable proxy for testing an AUV's capability to acoustically detect oil droplets [10]. In the study, several systems were tested, and the Nikuni Karyu Turbo Mixer (KTM) bubble generator was found to successfully discharge pressurized water enriched with a high concentration of micro-bubbles. The micro-bubbles persisted in the water column for several minutes and appeared like oil patches in the Ping360 sonar images.

To create a larger plume, the Nikuni KTM65S2 bubble generator (Fig. 4.2) was utilized since it has a higher flow rate than the model used in [10]. The bubble generator was set up at the edge of the wharf at Holyrood; the hose connected to its outlet extended 100m into the bay, where the bubbles were released from the discharge nozzle at a depth of approximately 20 m (Fig. 4.3).

4.2.3 Mission Description

The Explorer's VCC executed a pre-planned lawnmower mission around the location of the discharge nozzle. The mission was run at 5 m altitude, with a 20 m spacing between each of the 200 m long lines. During the mission, the Ping360 sonar continuously scanned the area for the bubble plume. Since the Ping360 cannot provide complete 360° areal coverage while the AUV is travelling at a speed of 1.5 m/s, the sonar was configured to scan only a 120° sector,

with a 50 m range. The positioning of the sector on the port side of the vehicle reflects the original plume-delineation objective of the trials. The Ping360 ensonifies a large vertical sector of the water column due to its 25° vertical beam width, thereby expanding the search area well beyond the horizontal plane defined by the depth of the vehicle.

The white track-line in Fig. 4.3 indicates the INS' estimate of the AUV's position during the mission; it shows that despite DVL aiding, a significant amount of INS drift occurred. The 120 m drift is apparent after the AUV surfaces as the difference between the INS position when the GPS fix is received, and the GPS corrected position. The INS is typically calibrated on land when the vehicle is powered up. However, due to an INS error that occurred prior to the mission, an in-water INS reset and calibration were required and may have been the cause of the drift.

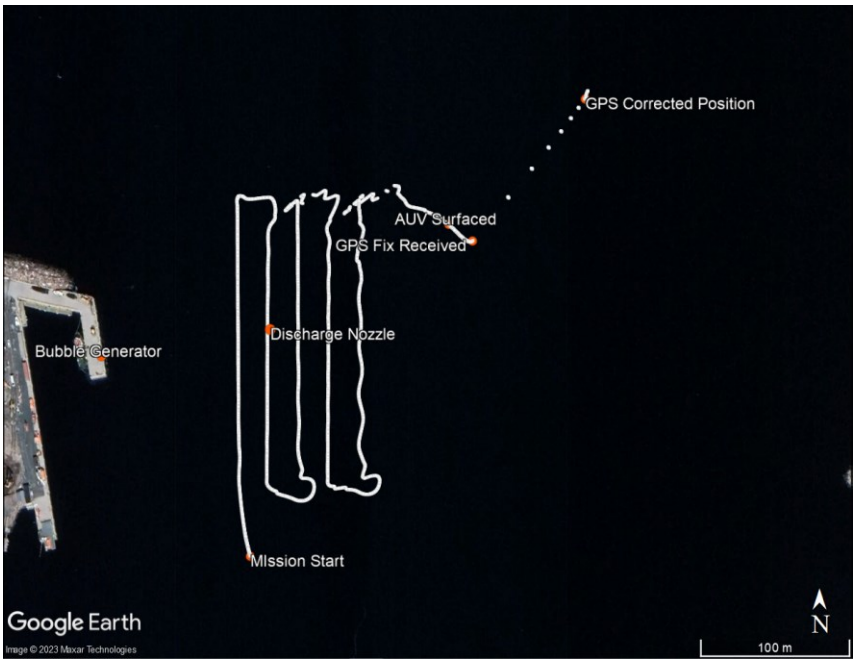


Fig. 4.3. AUV trajectory during the lawnmower mission (white track line), and the approximate locations of the bubble generator and discharge nozzle.

4.2.4 Data Acquisition and Playback Setup

The data acquisition and processing software components were implemented as MOOS applications to ensure that they could be utilized by the PCC for future adaptive missions. MOOS uses a publish-subscribe communication model, with the MOOS database (MOOSDB) at the center of the messaging system. Applications subscribe to the MOOSDB for updates on specific variables and publish generated data to the database. Note that although the MOOS-IvP software suite contains several processes for mission control, they are not relevant here since the test is only concerned with data collection and processing.

During the field trials, the Ping360 sonar was controlled by a custom iPing360Device MOOS application [80] running on the PCC. Sonar data received by the application was stored in the MOOS Database (MOOSDB) along with the navigation data (Fig. 4.4). All data written to the MOOSDB was also saved in the MOOS asynchronous log file.

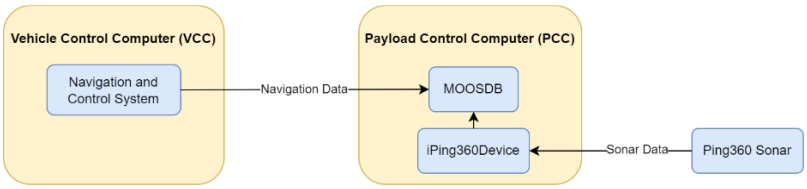


Fig. 4.4.Navigation data from the VCC and sonar data from the Ping360 is stored in the MOOSDB on the PCC

The plume detection algorithm presented in this thesis was implemented as a MOOS application called pPlumeDetector, and its interface is described in Appendix C. Although the pPlumeDetector code is implemented primarily in Python (Appendix A), the computationally expensive clustering step is implemented in Cython (Appendix B), which features Python-like syntax with the performance of C/C++.

pPlumeDetector was tested using uPlayback, a built-in MOOS-IvP software module that enables simulation testing through the replaying of logged data. The MOOS applications running during the playback test and the dataflow between them are shown in Fig. 4.5. uPlayback reads from the asynchronous log created during the mission and writes the logged values to the MOOSDB in the same order with which they were originally written. As a result, the pPlumeDetector application receives the navigation and sonar data as it would if it were running on the vehicle during the mission.

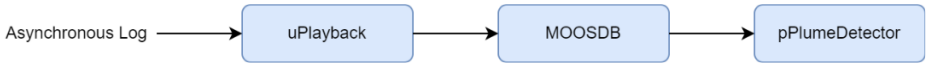


Fig. 4.5.Data flow between MOOS applications in the playback setup

4.3 Results

The proposed plume detection algorithm successfully identified high-density features in several scans of the sonar data during the playback test. Fig. 4.6 provides a visualization of the processing steps for a single sector scan from the experiment. Although the sonar range was set to 50 m, only the data within a 20 m range from the sonar is displayed as it contains all of the significant features in the scan. The parameter settings utilized for the playback test are listed in Table 4.1.

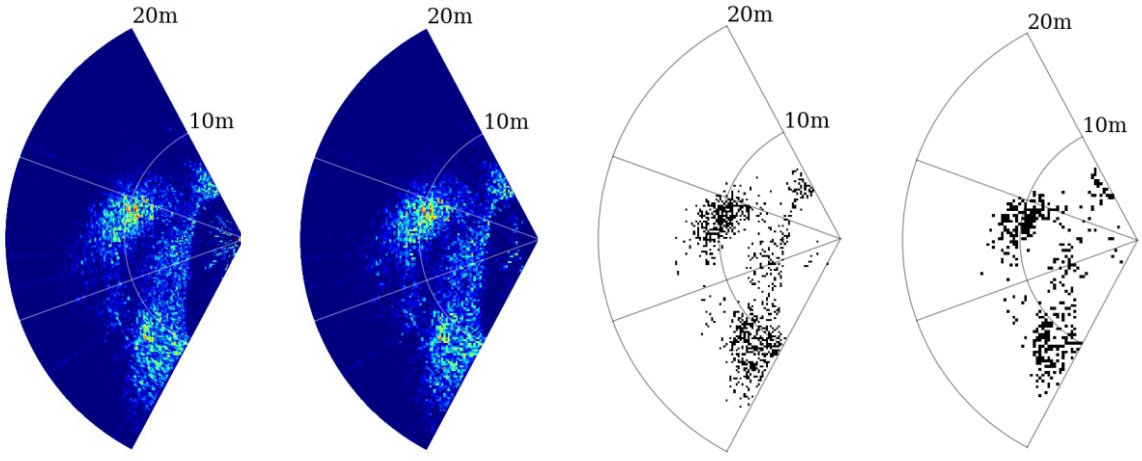
The sonar data collected with a resolution of 24 samples/meter is shown in Fig. 4.6(a); this high resolution data is used as-is for the initial computationally inexpensive steps. First, the range-gating process zeroes all data within 2 m of the sonar head as it typically contains significant noise (Fig. 4.6(b)). Next, segmentation retains samples with intensity values greater than 30% of the sonar's full-scale range. Note that the algorithm stores and processes the data

in the polar grid format for the steps in Fig. 4.6(a), (b) and (c). It is displayed in the Cartesian grid format, measuring range along both axes, for clarity.

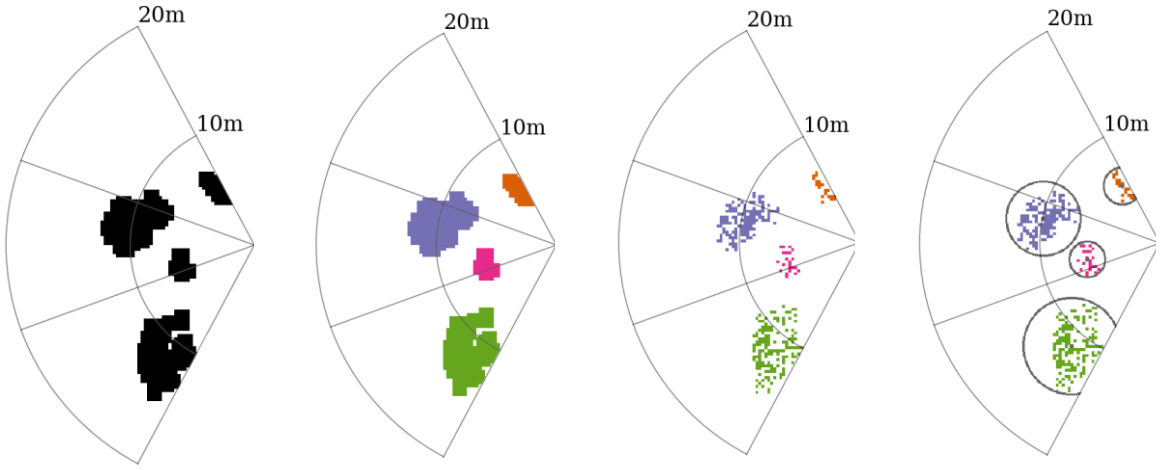
Prior to clustering, the segmented data is warped into a Cartesian grid and simultaneously downsampled to a resolution of 4 pixels/meter. The downsampling is evident from the pixelation in Fig. 4.6(d). From Fig. 4.6(d) onwards, the images are a direct representation of the data, which is stored in the Cartesian grid format.

The clustering process begins with the creation of the high-density blocks image (Fig. 4.6(e)); it identifies 1 m x 1 m blocks in which the fill exceeds 30%. Next, the cluster regions image identifies four sets of connected high-density blocks (Fig. 4.6(f)). Finally, the clusters image is created (Fig. 4.6(g)); it contains only the cluster pixels for the four clusters, labelled with the pixel value corresponding to the region in which they lie.

The output of the clustering process is the center and radius of each cluster. These are illustrated in Fig. 4.6(h), where each grey circle is centered on a cluster center, and has a radius corresponding to the cluster's radius.



(a) Raw Data (b) Range-gated (c) Segmented (d) Re-gridded



(e) High-Density Blocks (f) Cluster Regions (g) Clusters (h) Cluster Centers and Radii

Fig. 4.6: Visualization of the plume detection algorithm processing steps for a single sector scan from the experimental data

Table 4.1: Plume detection algorithm parameter setting for the playback test.

Algorithm Step	**Parameter**	**Value**
Range-Gating	*noise range*	2m
Segmentation	*segmentation threshold*	30%
Grid Conversion	*image width*	400 pixels
Clustering	*clustering block width*	1 m
	minimum fill threshold	30%
Georeferencing	$instrument\ offset_{x,body}$	3 m
	$instrument\ offset_{y,body}$	0 m

The location of each cluster center in the local coordinate frame was computed by the georeferencing algorithm, and is plotted in Fig. 4.7. The plot shows that most of the plume-like feature detections occurred on the last leg of the mission, more than 50 m from the discharge

nozzle. Unfortunately, due to the significant INS drift, the position of the detections cannot be used to verify that the features were indeed produced by the bubble plume. Nevertheless, since the data looks similar to scanning sonar images of a seep plume collected by an ROV [55], it still forms a valid test dataset.

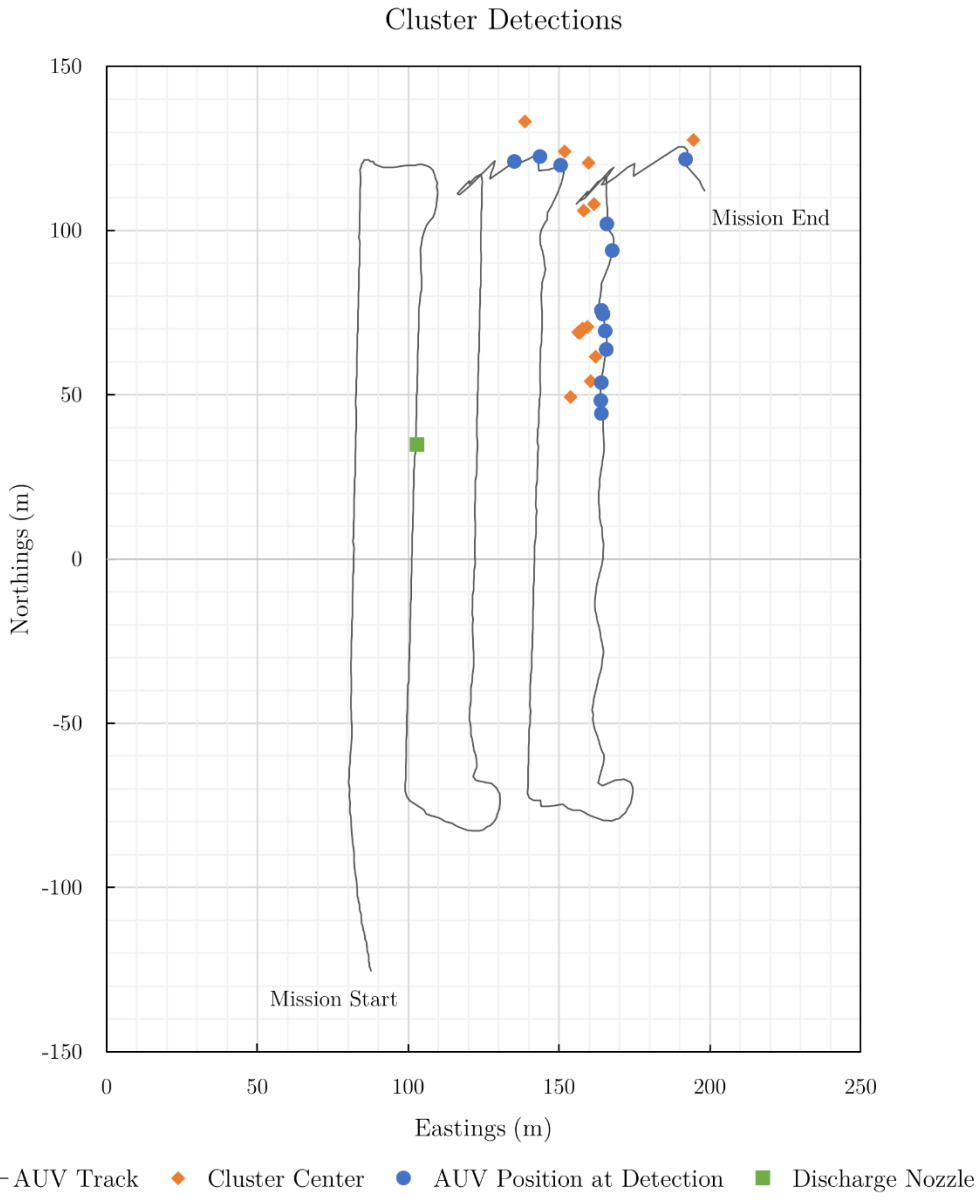


Fig. 4.7: AUV trackline during the mission, along with the cluster detections and location of the AUV at the time of the detections

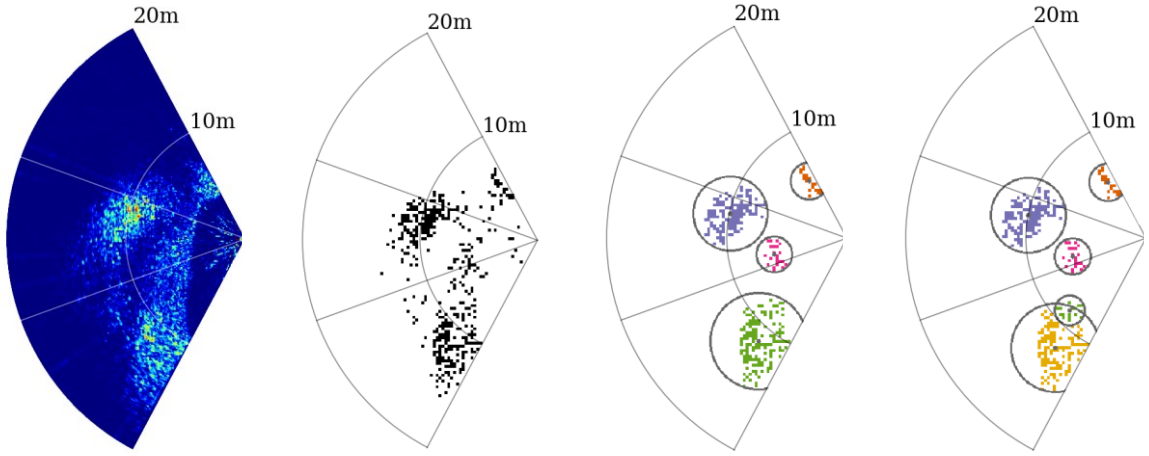
4.4 Analysis

This section consists primarily of a comparison between the block clustering algorithm and DBSCAN, in terms of their clustering outputs and computation times. It also provides direction on how to tune the algorithm's parameters for different use cases.

4.4.1 Clustering Output Comparison

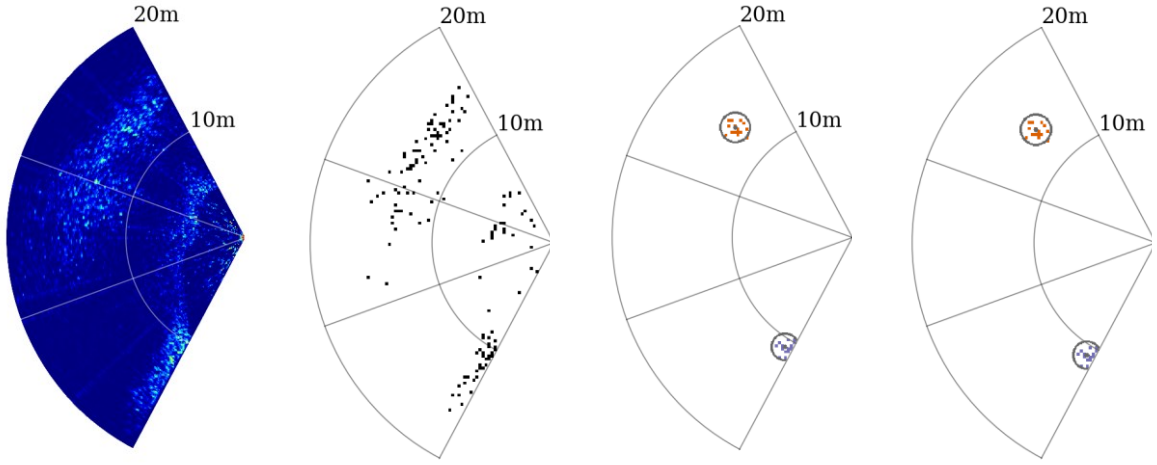
The block clustering and DBSCAN outputs for two scans from the playback test are shown in Fig. 4.8 and Fig. 4.9 below; for both algorithms, the parameter settings listed in Table 4.1 were utilized. A visual inspection of the generated clusters shows that they are similar. Since the plume detector's interface explicitly defines which positive detection pixels are cluster pixels, the number of cluster pixels identified by the two algorithms is the same in both scans.

Due to the different definitions for cluster connectivity, however, DBSCAN produces five clusters in Scan A while the block clustering algorithm produces four clusters. DBSCAN requires significant overlap between two high-density blocks for the positive detection pixels within them to belong to the same cluster. The block clustering algorithm, in contrast, does not require any overlap of the high-density blocks to merge clusters; the high-density blocks must only be connected (see Section 0). As a result, the block clustering algorithm merges clusters more readily, and it can be seen from Fig. 4.8 that the output is more visually intuitive.



(a) Raw Data (b) Re-gridded (c) Block Clustering Output (d) DBSCAN Output

Fig. 4.8: Comparison of DBSCAN and Block Clustering Outputs for Scan A



(a) Raw Data (b) Re-gridded (c) Block Clustering Output (d) DBSCAN Output

Fig. 4.9: Comparison of DBSCAN and Block Clustering Outputs for Scan B

DBSCAN and the block clustering algorithm are both capable of generating clusters with arbitrary shapes, which is advantageous when working with datasets featuring non-flat geometry. In these datasets, the clusters often have curved or irregular shapes and cannot be partitioned effectively using flat, linear boundaries (see Fig. 4.10). While the ability to identify

arbitrarily shaped clusters is beneficial, it also allows for an overlap in the circular cluster boundaries. For example, in DBSCAN's clustering output for Scan A (Fig. 4.8), the yellow cluster at the bottom lies within the boundary of the green cluster and vice-versa.

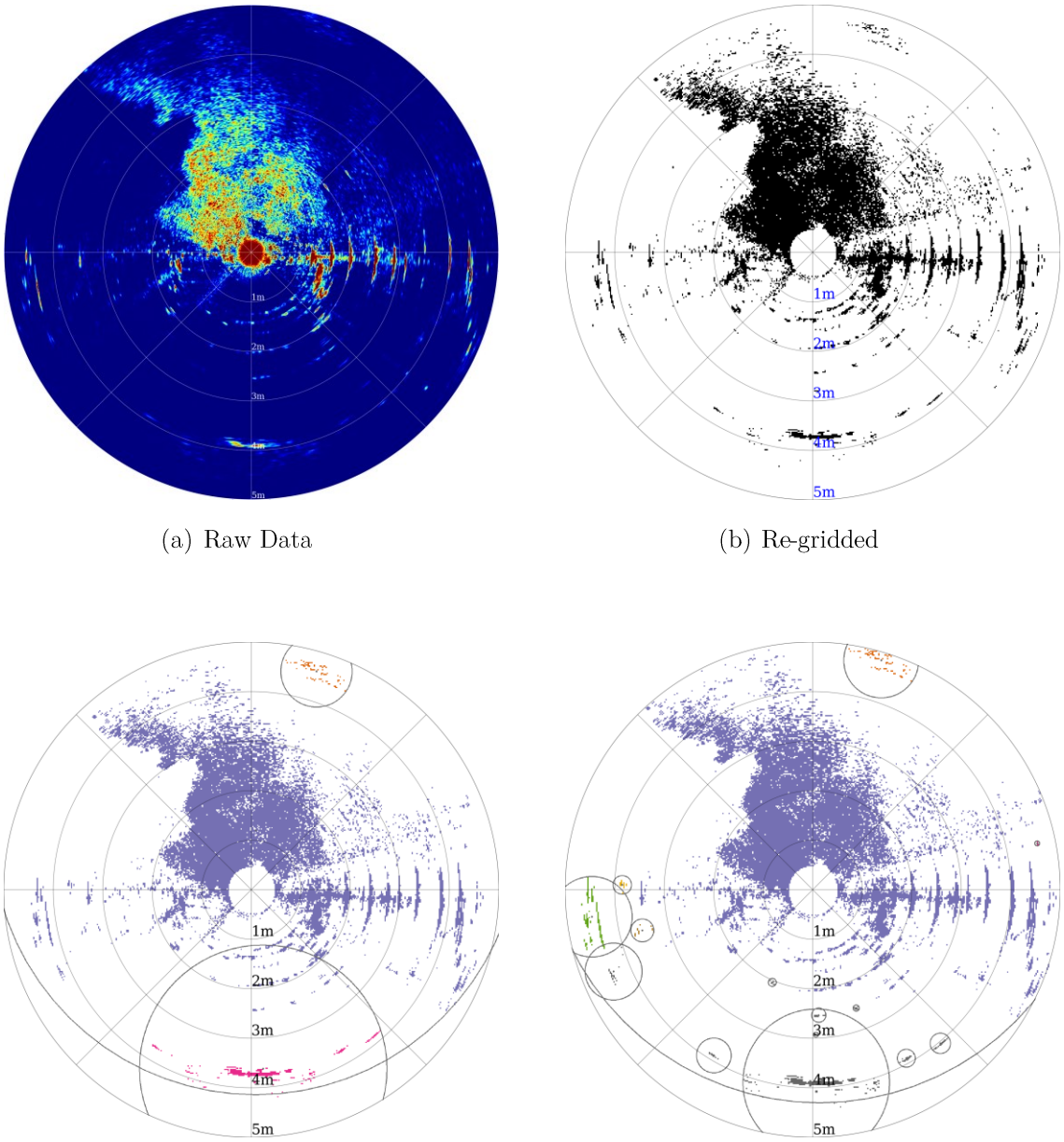
It is important to be aware of the possibility of overlapping cluster boundaries when utilizing the algorithm's output in an adaptive mission. For example, if the total cluster area is calculated as a sum of the clusters' radii-based areas, the computed value may not be an accurate representation of the data; any overlap would produce an overestimate of the total area.

Fig. 4.10: Dataset containing clusters with non-flat geometry

4.4.2 Computation Time Comparison

This computation time analysis studies how the clustering time of DBSCAN and the block clustering algorithm increases with clustering complexity. It uses data from an experiment in [62] since the dataset is more difficult to cluster, and consequently creates a better test environment for performance testing. During the experiment, the Ping360 sonar was secured to a lakeside dock and configured to scan the complete 360° sector around it. A micro-bubble generator positioned in the sonar's field of view generated a bubble plume, which the sonar captured in its acoustic scans (Fig. 4.11(a)). In the scan selected for this test, the plume occupies a large portion of the sonar's field of view despite its small size because the sonar's

range is set to only 5 m. The result is a larger number of positive detections, similar to what might be expected for a spill plume.



(a) Raw Data

(b) Re-gridded

(c) Block Clustering Output

(d) DBSCAN Output

Fig. 4.11: Comparison of DBSCAN and Block Clustering outputs for a Ping360 scan of a micro-bubble plume. A 10% *segmentation threshold*, 1% *minimum fill threshold* and 0.5m *clustering block width* are utilized.

The computation time analysis tests were carried out on a laptop with an Intel Core i7 processor. While both clustering algorithms were implemented primarily in Python, the

computationally expensive sections were implemented in Cython. The algorithm parameters were selected to generate a large number of cluster pixels; a 10% *segmentation threshold* produced significant positive detections (Fig. 4.11(b)), while a *minimum fill threshold* of 1% ensured that most of these detections were categorized as cluster pixels (Fig. 4.11(c)). The clustering complexity was then gradually increased by successively increasing the *clustering block width*, causing the clusters to merge and triggering cluster degeneration. Fig. 4.11(c) and Fig. 4.11(d) show the clustering output for the block clustering algorithm and DBSCAN respectively, with a small *clustering block width* of 0.5 m. It is evident that the clusters have already begun to degenerate, as most of the cluster pixels are contained within the central cluster. Once the block width is increased to 2 m, both algorithms generate only one cluster containing all the cluster pixels (Fig. 4.12).

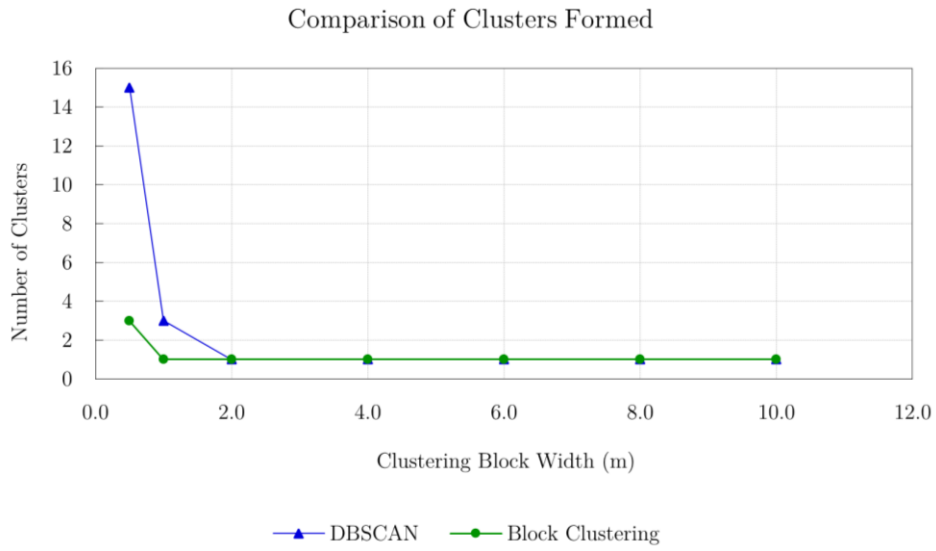


Fig. 4.12. Number of clusters formed by DBSCAN and the block clustering algorithm, with increasing *clustering block widths*

The computation times for both algorithms are plotted in Fig. 4.13 as a function of the *clustering block width*, where the highest block width of 10 m spans the whole image; each

data point in the plot represents the average of 10 test runs. The results show that the developed block clustering algorithm is consistently faster than DBSCAN. On average, it is 2.5 times faster, with the most significant difference observed at a block width of 2 m, where it is approximately 4 times faster than DBSCAN. Notably, this is the same point where the clusters first degenerate.

Due to its faster computation times, the block clustering algorithm offers more flexibility when selecting the *clustering block width*. For example, if we require this data to be processed within 0.5 seconds, DBSCAN's block width would have to be under 1.5 m, while values up to 4.5 m could be utilized for the block clustering algorithm. Furthermore, the block clustering algorithm's computation time increases at a much slower rate within the most likely operation range, which is when the *clustering block width* is less than 5 m, or 50% of the image width. This stability is desirable in a real-time application because it means that if the algorithm is tuned with a certain dataset, similar computation times can be expected with similar data.

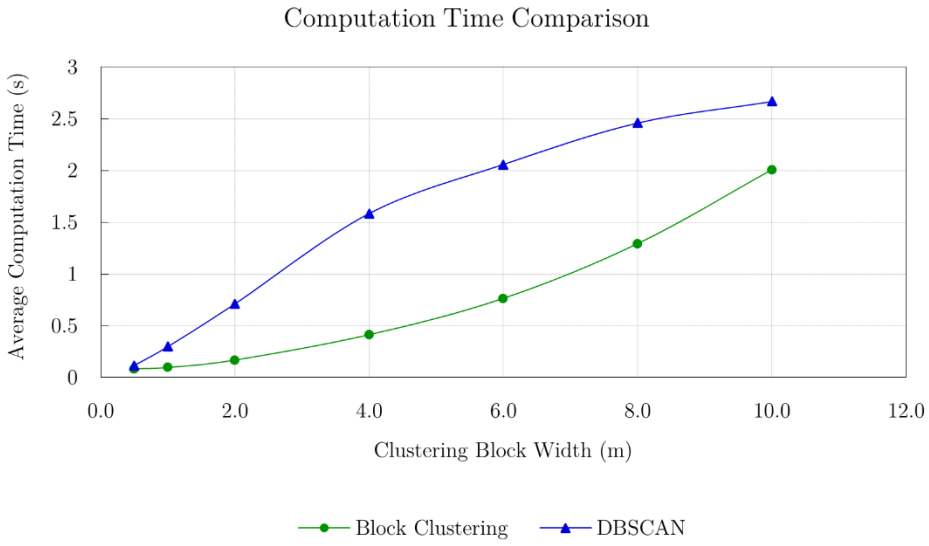
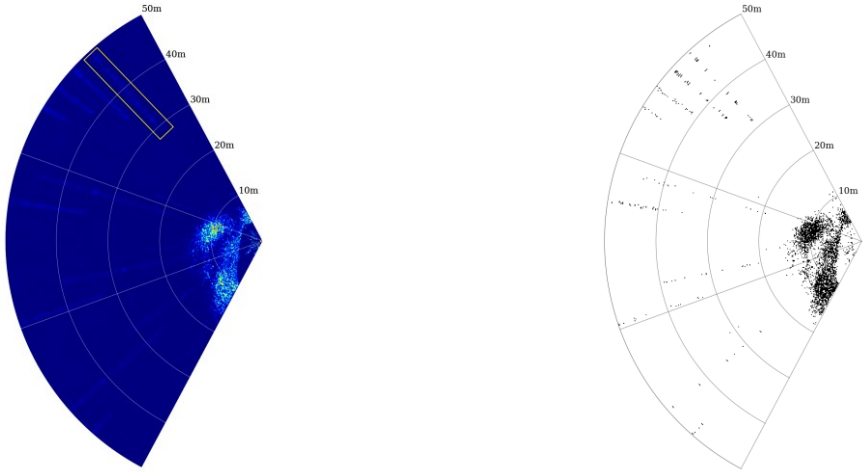


Fig. 4.13: Impact of increasing clustering complexity on the DBSCAN and block clustering computation times

4.4.3 Parameter Selection

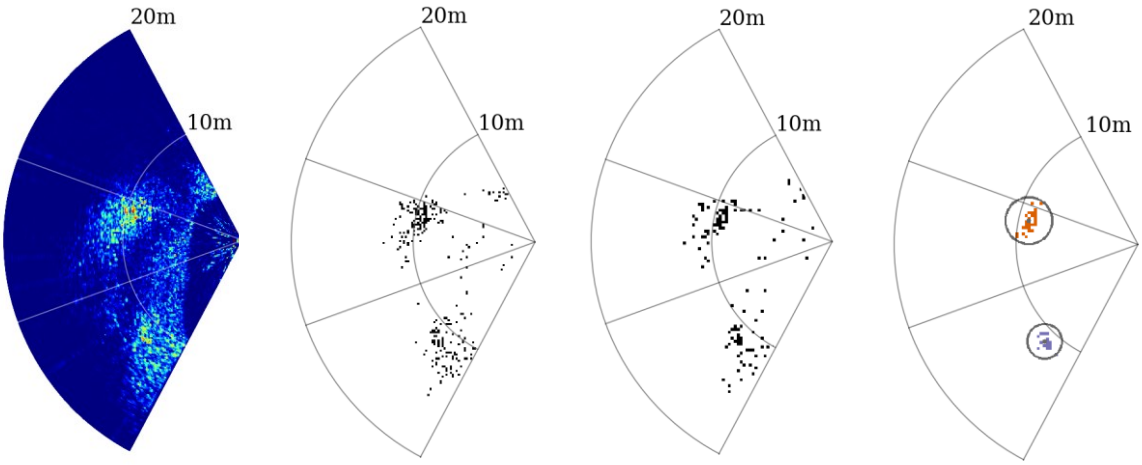
The plume detector's parameters can be configured based on a visual analysis of how changes to the parameters affect the algorithm's output. Since selecting the segmentation and clustering parameters requires some consideration, this section provides direction on how they can be configured for different types of plumes.

The *segmentation threshold* should be configured to retain medium and high intensity returns. A lower and more sensitive threshold is especially important in the seep environment where the bubbles rise quickly forming vertical plumes with a small horizontal footprint. Setting the threshold too low, however, may result in the unwanted detection of artifacts in the sonar data. For example, with the *segmentation threshold* as low as 10%, the radial lines in Fig. 4.14 are also considered positive detections even though they are most likely produced by electrical noise or acoustic interference from other devices on the AUV. Conversely, a higher threshold produces sparse detections and consequently inhibits the formation of clusters, as evidenced by Fig. 4.15 where the threshold is set to 50%. An effective approach is to use the lowest possible threshold which will discard obvious noise and artifacts.



(a) Raw Data (b) Segmented

Fig. 4.14: Segmentation with the threshold set to 10%. In (a), an artifact in the sonar data is outlined with a yellow box.

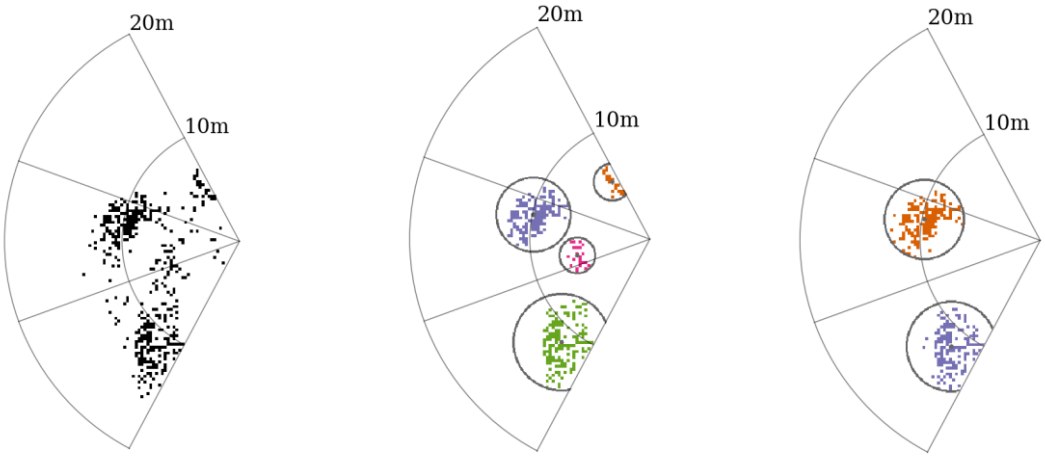


(a) Input (b) Segmented (c) Re-gridded (d) Clustered

Fig. 4.15: Block clustering steps with a 50% *segmentation threshold*, 1 m *clustering block width*, and 30% *minimum fill threshold*.

The *clustering block width* should be selected based on the expected size of the plume and can be increased for larger spill plumes to decrease the likelihood of noise being detected as a target. For example, while four clusters are detected in Fig. 4.16(b) with the block width set to 1 m, only the two larger clusters are detected when the block width is increased to 2 m (Fig.

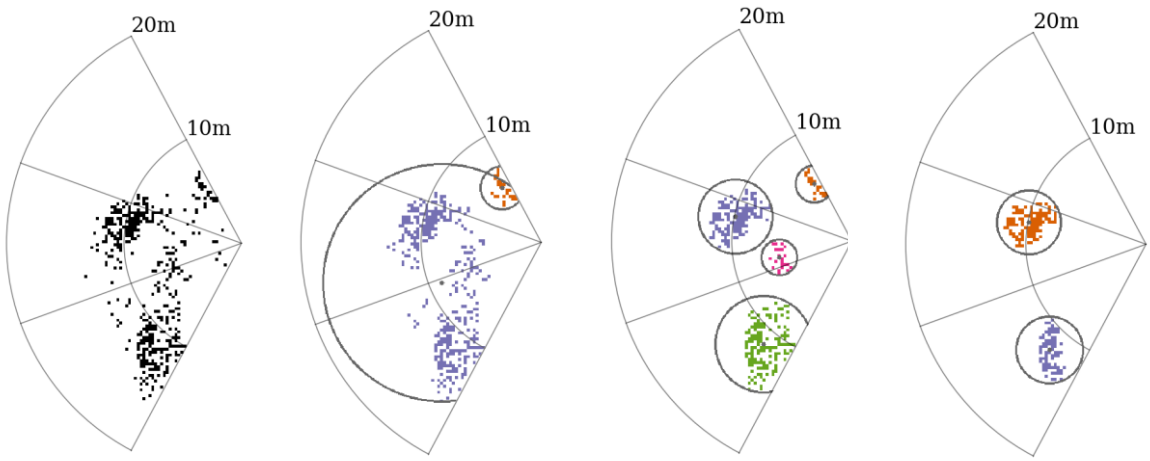
4.16(c)). The benefit of noise resistance provided by a larger block width should, however, be weighed against the possibility of computation time increases. As a result, the block width is best set to the largest reasonable value which maintains real-time performance during tests with the onboard processor.



(a) Re-gridded Image (b) Clustering Output with 1 m block width (c) Clustering Output with 2 m block width

Fig. 4.16: Clustering of a segmented image with different block widths. The minimum fill percent is constant (30%).

The nature and density of the plume determine what is an appropriate setting for the *minimum fill threshold*. A 30% minimum fill was selected for the field trials data because it identifies features similar to those in scanning sonar images of gas seeps [55] (Fig. 4.17(c)). A lower setting of 10% creates a large cluster containing very low-density regions which could, on their own, be noise (Fig. 4.17(b)). With a higher setting of 50%, the generated clusters are not representative of the original data as the medium-density clusters are classified as noise (Fig. 4.17(d)). Higher values for the *minimum fill threshold*, can, however, be used for spill plumes since the neutrally buoyant micro-bubbles spread out laterally forming a large dense plume.



(a) Re-gridded Image (b) Clustering Output with 10% minimum fill (c) Clustering Output with 30% minimum fill (d) Clustering Output with 50% minimum fill

Fig. 4.17: Clustering of a segmented image with different *minimum fill thresholds*. The *clustering block width* is constant (1 m)

4.5 Conclusion

This chapter has demonstrated using field trials data, that the plume detection algorithm accomplishes its primary goal of identifying high-density regions in sonar data. While it is uncertain as to whether the high intensity acoustic reflections were created by the bubble plume, the data is similar to what might be expected for a seep plume and is consequently useful for testing.

A comparative analysis of the block clustering algorithm and DBSCAN shows that they identify the same pixels as cluster pixels, but form the clusters differently due to differences in definitions of connectivity; the block clustering algorithm merges clusters more readily, producing an output that is more visually intuitive. With regards to processing time, the block clustering algorithm is consistently faster than DBSCAN and is consequently a better option than DBSCAN for real-time applications. Finally, the clustering algorithm can be configured

based on the expected characteristics of the plume. Thus, this chapter has also demonstrated that the design considerations related to real-time processing and configurability were addressed.

Chapter 5

Conclusions and Future Work

This thesis has been directed towards exploring how the oil spill detection capability of an AUV can be enhanced to facilitate testing at a natural seep location. The Literature Review in Chapter 2 identifies challenges at seep sites stemming from the presence of multiple sources with spatiotemporally varying flow and lower hydrocarbon concentrations. It subsequently proposes automated acoustic detection as a method that can address these challenges. Finally, by demonstrating that existing algorithms are not applicable to both seeps and spills, it provides the motivation for a novel plume detection algorithm.

Chapter 3 presents a novel plume detection algorithm for a forward-looking scanning sonar. The algorithm identifies the plume as dense clusters of points in the sonar data and returns the georeferenced center coordinates of each cluster. It consists of five steps: range-gating, segmentation, grid conversion, clustering, and georeferencing. For the clustering step, DBSCAN is identified as an existing algorithm that can be configured to match the plume detector's clustering interface. However, since available DBSCAN implementations show significant run-time increases when the clustering degenerates, a block clustering algorithm designed specifically for the segmented image data was developed.

The proposed block clustering algorithm is the most significant contribution of this thesis. By directly processing the segmented image, it works around the issue that causes the increase in DBSCAN's computation time. It first identifies cluster pixels, which lie in high-density blocks, and then forms clusters, which lie in connected blocks. The developed clustering algorithm is not limited to detecting submerged plumes, or even to subsea applications. It can be utilized in any image processing application where density-based clustering is required.

In Chapter 4, the plume detection algorithm is validated using data collected by a Ping360 sonar mounted on the Explorer AUV. The data was collected during a mission in which the AUV surveyed an area containing a submerged micro-bubble plume. The plume detection algorithm successfully identified high-density features in the sonar data during a playback test. However, since the cluster centers are more than 50 m away from the bubble generator's outlet, it is difficult to be certain that the sonar detected the plume during the mission.

Chapter 4 also contains a data-based comparison of the block clustering algorithm and DBSCAN. Comparing the clustering output shows that both clustering algorithms identify the same cluster pixels, but the block clustering algorithm sorts these pixels into clusters in a manner that is more visually intuitive. When comparing processing time, block clustering is faster and more stable than DBSCAN, and is consequently better suited for real-time processing. Finally, Chapter 4 provides direction on how to tune some of the plume detection algorithm's parameters and demonstrates that the algorithm can be configured for a smaller seep plume or a larger spill plume.

The plume detection software was installed on Memorial University's Explorer AUV and utilized during the Scott Inlet trials (see Section 1.4). It should be noted that for these trials, DBSCAN was utilized for the clustering step because it was the faster option at the time; the

block clustering algorithm had not yet been implemented in Cython, and its original Python implementation was slower than scikit's Cython-based DBSCAN implementation. It was possible, however, to ensure real-time processing by configuring DBSCAN with a smaller *clustering block width* value for the smaller seep plumes.

A natural next step for this research would be to collect more test data, as the current dataset is relatively small, and there is uncertainty as to whether the plume was detected. However, the trials at Holyrood with the Ping360 sonar proved to be challenging; there were several missions in which the sonar may have missed the plume due to its slow scan rate, producing a relatively empty dataset. It would consequently be worthwhile to conduct future tests with a Multi-beam Echo Sounder (MBES), which is more likely to capture the plume as it can complete multiple scans within one second [81]. The plume detection algorithm can also be easily adapted for use with an MBES.

The primary limitation of sonars lies in their inability to distinguish between plumes and other objects in the water column. To address this limitation during the Scott Inlet project, a fluorometer was selected to complement the Ping360 and provide more specific information regarding hydrocarbons in the water column. The project took a modular approach, processing the data from the two sensors separately during different phases of the mission. It would be beneficial, however, for future research to explore how sonar data can be fused with in-situ sensor data for plume detection, as it would maximize the data usage. Existing sensor fusion algorithms for AUV-based plume detection are limited to use with in-situ sensors since they correlate the time-series data based on their timestamps [8]. Further research is consequently required to integrate sonar data which requires position-based correlation.

The plume detection algorithm focuses purely on detection, and not tracking, to facilitate testing at natural seep sites. It is worth noting that this decoupling also makes the algorithm more modular, allowing for more varied uses. For example, the georeferenced cluster center data can be easily added to a map containing data from other sensors, and tracking decisions can then be made based on this integrated sensor data map. In contrast, detection and tracking are tightly coupled in existing sonar-based plume tracking algorithms, which does not leave much flexibility for integration with other sensors [38], [41]. The proposed plume detection algorithm is consequently well suited for an integrated multi-sensor approach to plume tracking.

Appendix A

Plume Detector Code

The following is the Python implementation of the plume detector algorithm presented in this thesis; it calls the block clustering function in Appendix B. The git repository is also available online [60].

```
import os
import csv
import copy
import time
import pymoos
import math
import numpy as np
from datetime import datetime
from brping import PingParser
from brping import definitions
from collections import OrderedDict
from skimage import measure
import cv2 as cv
from matplotlib import pyplot as plt
import os
import pyproj
from pPlumeDetector.src.block_cluster import block_cluster_fn
import timeit


class Cluster():
    def __init__(self):
        # Cluster radius is the distance from the cluster center to the furthest point from it
        self.radius_pixels = 0
        self.radius_m = 0
        self.num_points = 0

        # Sonar beam angle at cluster center
        self.angle_degs = 0

        # Vehicle nav data at time that the sonar pinged the cluster center
        self.nav_x = 0
        self.nav_y = 0
        self.nav_heading = 0

        # Cluster center in pixel coordinates
        self.center_row = 0
        self.center_col = 0

        # Cluster center in the different reference frames
        self.instrument_x = 0
        self.instrument_x = 0
        self.body_x = 0
        self.body_y = 0
        self.local_x = 0
        self.local_y = 0
        self.lat = 0
        self.long = 0
```

```
    def serialize(self):
        return OrderedDict([
            ("center_row_pixels", self.center_row),
            ("center_col_pixels", self.center_col),
            ("num_points", self.num_points),
            ("radius_pixels", self.radius_pixels),
            ("center_local_x_m", self.local_x),
            ("center_local_y_m", self.local_y),
            ("radius_m", self.radius_m),
            ("center_lat", self.lat),
            ("center_long", self.long),
            ("nav_x_m", self.nav_x),
            ("nav_y_m", self.nav_y),
            ("nav_heading_deg", self.nav_heading)
        ])

class PPlumeDetector():
    def __init__(self):

        # Algorithm Parameters
        self.threshold = 0.3 * 255
        self.window_width_m = 1.0 #1.0 # Clustering window size
        self.cluster_min_fill_percent = 30 #50
        self.noise_range_m = 2# Range within which data is ignored (usually just noise)
        self.image_width_pixels = 400# Width in pixels of sonar images
        # Distance between the Ping360 and INS center, measured along the vehicle's longitudinal axis
        self.instrument_offset_x_m = 3
        self.num_cluster_outputs = 5

        # Constants
        self.grads_to_rads = np.pi/200 #400 gradians per 360 degs
        self.rads_to_grads = 200/np.pi
        self.max_angle_grads = 400

        self.window_width_pixels = None
        self.cluster_min_pixels = None

        self.comms = pymoos.comms()

        # Vars for storing data from MOOS DB
        self.num_samples = None
        self.num_steps = None
        self.start_angle_grads = None
        self.stop_angle_grads = None
        self.transmit_enable = None
        self.speed_of_sound = None
        self.binary_device_data_msg = None # Encoded PING360_DEVICE_DATA message
        self.device_data_msg = None # Decoded PING360_DEVICE_DATA message
        self.range_m = None # Sonar range
        self.lat_origin = None
        self.long_origin = None

        # Angles at which to processs the sector scan data. Gets set to the start & stop angles
        self.scan_processing_angles = None

        # Matrix containing raw intensity data for a complete scan of pings between the start and stop angles
        self.scan_intensities = None
        self.scan_intensities_denoised = None # Scan intensities with the central noise data removed

        # Matrix containing segmented data (i.e. result from thresholding) from the scan of the entire sonar swath.
        # Row indexes define the sample number and each column is for a different scan angle
        self.seg_scan = None
        self.seg_scan_snapshot = None # Copy of seg scan, taken at start/stop angle

        self.seg_img = None           # self.seg_scan warped into an image (re-gridded to cartesian grid)
        self.clustered_cores_img = None # Image with core cluster pixels (percentage of pixels in surrounding > threshold)
        self.cluster_regions_img = None # Image wth all pixels in clustering windows set to 1  (used for labelling)
        self.labelled_regions_img = None # Cluster regions image, with unique label (pixel value) applied to each region
        self.labelled_clustered_img = None # seg_image * labelled_regions_img
        self.clustered_img = None # Black and white version of labelled_clustered_img
        self.output_img = None
        self.clustered_img_view = None

        self.num_scans = 0
        self.first_scan = True
        self.clustering_pending = False

        # Cluster information for the clusters detected in the current scan
        self.num_clusters = 0
        self.clusters = [] # List of Cluster data structures
```

```
        self.sorted_clusters = [] # List of Cluster data structures, sorted based on cluster radius (largest to smallest)
        self.cluster_centers_string = ""

        # Cluster information of clusters detected since the start
        self.cluster_centers = []

        self.clustering_time_secs = None
        self.total_processing_time_secs = None

        self.data_save_path = None
        self.orig_images_path = None
        self.viewable_images_path = None

        # Most recent nav data
        self.current_nav_x = 0
        self.current_nav_y = 0
        self.current_nav_heading = 0

        # Nav data is saved every time a ping message is received. There is storage for each position of the sonar head,
        # allowing for the nav data at the time of each ping to be stored.
        self.nav_x_history = np.zeros(self.max_angle_grads)
        self.nav_y_history = np.zeros(self.max_angle_grads)
        self.nav_heading_history = np.zeros(self.max_angle_grads)

        #plt.rcParams['figure.constrained_layout.use'] = True
        plt.rcParams['font.family'] = 'serif'

        self.state_string= 'DB_DISCONNECTED'
        self.states = {
            'DB_REGISTRATION_ERROR': -1,
            'DB_DISCONNECTED': 0,
            'DB_CONNECTED': 1,
            'STANDBY': 2,
            'ACTIVE': 3,
        }

        self.status_string = 'GOOD'
        self.statuses = {
            'GOOD': 1,
            'DB_REGISTRATION_ERROR': -1,
            'TIMEOUT': -2,
            'PROCESSING_ERROR': -3
        }

    def run(self):

        # Setup pymoos comms
        print("Initial State: {0}".format(self.state_string))
        self.comms.set_on_connect_callback(self.on_connect)
        self.comms.set_on_mail_callback(self.on_mail)
        self.comms.run('localhost', 9000, 'p_plume_detector')
        #self.comms.run('192.168.56.104', 9000, 'p_plume_detector')

        # Create folder for saving images. Save in /log directory if it exists, otherwise use current directory
        date_time = datetime.strftime(datetime.now(), '%Y%m%d_%H%M%S')
        folder_name = "plume_detector_data_" + date_time
        if os.path.exists("/log"):
            log_dir = "/log"
        else:
            log_dir  = os.path.dirname(__file__)
        self.data_save_path = os.path.join(log_dir, folder_name)
        self.orig_images_path = os.path.join(self.data_save_path, "orig_images")
        self.viewable_images_path = os.path.join(self.data_save_path, "viewable_images")

        try:
            os.mkdir(self.data_save_path)
            os.mkdir(self.orig_images_path)
            os.mkdir(self.viewable_images_path)

        except OSError as error:
            print(error)

        while True:

            # Process scan when ping data from start/end of scan is received
            if self.clustering_pending:

                start_time = time.time()
                # Warp data into image with cartesian co-ordinates, then cluster
                self.seg_img = self.create_sonar_image(self.seg_scan_snapshot)
                self.cluster()
```

```
            self.calc_cluster_centers()
            self.get_cluster_center_nav()
            self.georeference_clusters()
            self.output_sorted_cluster_centers()
            self.save_images()

            # Calculate and output the total processing time
            end_time = time.time()
            self.total_processing_time_secs = end_time - start_time
            print_str = 'Scan ' + str(self.num_scans) + ": Processing time is " + \
                        str(self.total_processing_time_secs) + " secs"
            print(print_str)
            self.comms.notify('PLUME_DETECTOR_TOTAL_PROCESSING_TIME_SECS', self.total_processing_time_secs,
                              pymoos.time())

            self.save_text_data()
            self.clustering_pending = False

        time.sleep(0.02)  # 50Hz


    return

def on_connect(self):
    '''Registers vars with the MOOS DB'''
    success = self.comms.register('SONAR_NUMBER_OF_SAMPLES', 0)
    success = success and self.comms.register('SONAR_NUM_STEPS', 0)
    success = success and self.comms.register('SONAR_START_ANGLE_GRADS', 0)
    success = success and self.comms.register('SONAR_STOP_ANGLE_GRADS', 0)
    success = success and self.comms.register('SONAR_RANGE', 0)
    success = success and self.comms.register('SONAR_PING_DATA', 0)
    success = success and self.comms.register('SONAR_TRANSMIT_ENABLE', 0)
    success = success and self.comms.register('SONAR_SPEED_OF_SOUND', 0)
    success = success and self.comms.register('NAV_X', 0)
    success = success and self.comms.register('NAV_Y', 0)
    success = success and self.comms.register('NAV_HEADING', 0)
    success = success and self.comms.register('LAT_ORIGIN', 0)
    success = success and self.comms.register('LONG_ORIGIN', 0)
    success = success and self.comms.register('PLUME_DETECTOR_CSV_WRITE_CMD', 0)

    if success:
        self.set_state('DB_CONNECTED')
        self.set_status('GOOD')
    else:
        self.set_state('DB_REGISTRATION_ERROR')
        self.set_status('DB_REGISTRATION_ERROR')

    return success

def set_state(self, state_string_local):
    ''' Sets the state_string class var, and updates the PLUME_DETECTOR_STATE_STRING and PLUME_DETECTOR_STATE_NUM
    MOOS DB vars if the DB is connected'''
    self.state_string = state_string_local
    print("State: {0}".format(self.state_string))

    if self.states[self.state_string] > self.states['DB_DISCONNECTED']:
        self.comms.notify('PLUME_DETECTOR_STATE_STRING', self.state_string, pymoos.time())
        self.comms.notify('PLUME_DETECTOR_STATE_NUM', self.states[self.state_string], pymoos.time())

    return

def set_status(self, status_string_local):
    ''' Sets the status_string class var, and updates the PLUME_DETECTOR_STATUS_STRING and PLUME_DETECTOR_STATUS_NUM
    MOOS DB vars if the DB is connected'''

    if self.status_string != status_string_local:
        print("Status: {0}".format(self.status_string))

    self.status_string = status_string_local

    if self.states[self.state_string] > self.states['DB_DISCONNECTED']:
        self.comms.notify('PLUME_DETECTOR_STATUS_STRING', self.status_string, pymoos.time())
        self.comms.notify('PLUME_DETECTOR_STATUS_NUM', self.statuses[self.status_string], pymoos.time())

    return

def on_mail(self):
    '''Handles incoming messages - calls functions to configure the class and process the ping data. The num steps,
    start angle, stop angle and number of samples can only be set on startup. Once the class is configured, changes
    to these vars in the MOOS DB do not have any effect.'''
```

```
        # Save all new input data
        msg_list = self.comms.fetch()

        for msg in msg_list:

            self.save_input_var(msg)

            # Evaluate state machine, call function to process any new ping data
            if self.state_string in ['DB_DISCONNECTED', 'DB_REGISTRATION_ERROR']:
                # Do nothing if all vars not registered with the DB
                # Also, should not get here if state is 'DB_DISCONNECTED'
                pass

            elif self.state_string == 'DB_CONNECTED':
                if self.configure():
                    if self.transmit_enable:
                        self.set_state('ACTIVE')
                    else:
                        self.set_state('STANDBY')

            elif self.state_string == 'STANDBY':
                if self.transmit_enable:
                    self.set_state('ACTIVE')

            elif self.state_string == 'ACTIVE':
                if self.binary_device_data_msg is not None:
                    if self.process_ping_data():
                        self.set_status('GOOD')
                    else:
                        self.set_status('PROCESSING_ERROR')

        return True

    def save_input_var(self, msg):
        '''Saves message data in correct class var.'''

        # Save message data
        name = msg.name()
        if name == 'SONAR_PING_DATA':
            self.binary_device_data_msg = msg.binary_data()
        else: # Numeric data type
            val = msg.double()
            #print("Received {0}: {1}".format(name, val))

            if name == 'SONAR_NUMBER_OF_SAMPLES':
                self.num_samples = int(val)
            elif name == 'SONAR_NUM_STEPS':
                self.num_steps = int(val)
            elif name == 'SONAR_START_ANGLE_GRADS':
                self.start_angle_grads = int(val)
            elif name == 'SONAR_STOP_ANGLE_GRADS':
                self.stop_angle_grads = int(val)
            elif name == 'SONAR_RANGE':
                self.range_m = val
            elif name == 'SONAR_TRANSMIT_ENABLE':
                self.transmit_enable = int(val)
            elif name == 'SONAR_SPEED_OF_SOUND':
                self.speed_of_sound = int(val)
            elif name == 'NAV_X':
                self.current_nav_x = val
            elif name == 'NAV_Y':
                self.current_nav_y = val
            elif name == 'NAV_HEADING':
                self.current_nav_heading = val
            elif name == 'LAT_ORIGIN':
                self.lat_origin = val
            elif name == 'LONG_ORIGIN':
                self.long_origin = val
            elif name == 'PLUME_DETECTOR_CSV_WRITE_CMD':
                if val == 1:
                    self.write_cluster_centers_csv()

        return

    def configure(self):
        ''' Initialize class data storage arrays if all config variable have been set'''

        required_vars = [self.num_samples, self.range_m, self.num_steps, self.start_angle_grads, self.stop_angle_grads,
                         self.speed_of_sound, self.lat_origin, self.long_origin]

        # Class can be configured if all the config vars have been set
```

```
    if all(item is not None for item in required_vars):

        self.scan_intensities = np.zeros((self.num_samples, self.max_angle_grads), dtype=np.uint8)
        self.scan_intensities_denoised = np.zeros((self.num_samples, self.max_angle_grads), dtype=np.uint8)
        self.seg_scan = np.zeros((self.num_samples, self.max_angle_grads), dtype=np.uint8)

        self.scan_processing_angles = [self.start_angle_grads, self.stop_angle_grads]

        print("Config vars:samples: {0}, range: {1}, steps: {2}, start: {3}, stop: {4}, speed of sound: {5}, "
              "lat origin: {6}, long origin: {7}".format(self.num_samples, self.range_m, self.num_steps,
              self.start_angle_grads, self.stop_angle_grads, self.speed_of_sound, self.lat_origin, self.long_origin))

        return True

    else:
        return False


def process_ping_data(self):
    '''Calls functions to decode the binary ping data and save nav data. If the transducer head is at the start/stop
    angle, it creates a copy of the sector scan data and sets a flag to indicate that the clustering can be run'''

    if not self.decode_device_data_msg():
        return False

    if not self.update_scan_intensities():
        return False

    self.save_nav_data()

    #print(str(self.device_data_msg.angle))
    # Process the data when at the start/stop angles
    if self.device_data_msg.angle in self.scan_processing_angles:
        self.num_scans = self.num_scans + 1
        self.comms.notify('PLUME_DETECTOR_NUM_SCANS', self.num_scans, pymoos.time())

        # Copy data and set flag for clustering to be completed in the run thread
        self.seg_scan_snapshot = copy.deepcopy(self.seg_scan)
        self.clustering_pending = True

    return True


def decode_device_data_msg(self):
    '''Decodes the Ping360 device data message stored in self.binary_device_data_msg, and stores the decoded
      message in self.device_data_msg '''

    ping_parser = PingParser()

    # Decode binary device data message
    for byte in self.binary_device_data_msg:
        # If the byte fed completes a valid message, PingParser.NEW_MESSAGE is returned
        if ping_parser.parse_byte(byte) is PingParser.NEW_MESSAGE:
            self.device_data_msg = ping_parser.rx_msg

    # Set to None as an indicator that the data has been processed
    self.binary_device_data_msg = None

    if self.device_data_msg is None:
        print("Failed to parse message")
        return False

    if self.device_data_msg.message_id != definitions.PING360_DEVICE_DATA:
        print("Received {0} message instead of {1} message".format(self.device_data_msg.name,'device_data'))
        return False

    return True

def update_scan_intensities(self):
    ''' Stores the intensity data, removes the noise close to the transducer and segments the data'''

    # Ensure that dataset is the correct size
    intensities = np.frombuffer(self.device_data_msg.data, dtype=np.uint8) # Convert intensity data bytearray to numpy
                                                                           array
    if intensities.size != self.num_samples:
        print("Intensities array length ({0}) does not match number of samples ({1}). Data not
               stored".format(intensities.size, self.num_samples))
        return False

    # Save the intensity data
    scanned_angle = self.device_data_msg.angle
```

```
        self.scan_intensities[:, scanned_angle] = intensities

        # Remove noise data close to the head
        noise_range_samples = int(self.noise_range_m / self.range_m * self.num_samples)
        self.scan_intensities_denoised[:,scanned_angle] = intensities
        self.scan_intensities_denoised[0:noise_range_samples, scanned_angle] = np.zeros((noise_range_samples),
                                                                                        dtype=np.uint8)

        # Apply a threshold to segment the data
        self.seg_scan[:,scanned_angle] = (self.scan_intensities_denoised[:,scanned_angle]  >
                                          self.threshold).astype(np.uint8)


        return True

    def save_nav_data(self):
        '''Stores the current nav data in the nav data history arrays, at the index location defined by the scanned angle'''

        scanned_angle = self.device_data_msg.angle

        self.nav_x_history[scanned_angle] = self.current_nav_x
        self.nav_y_history[scanned_angle] = self.current_nav_y
        self.nav_heading_history[scanned_angle] = self.current_nav_heading

        return

    def create_sonar_image(self, sector_intensities):
        '''First rearrages sector intensities matrix to match OpenCV reference - includes reference frame conversion as
        Ping 360 reference uses 0 towards aft while OpenCV uses 0 towards right. Then re-grids to cartesian co-ordinates
        using the OpenCV warpPolar function'''

        # Transpose sector intensities to match matrix format required for warping
        sector_intensities_t = copy.deepcopy(sector_intensities)
        sector_intensities_t = sector_intensities_t.transpose()

        # Rearrange sector_intensities matrix to match warp co-ordinates (0 is towards right)
        sector_intensities_mod = copy.deepcopy(sector_intensities_t)
        sector_intensities_mod[0:100] = sector_intensities_t[300:400]
        sector_intensities_mod[100:400] = sector_intensities_t[0:300]

        # Warp intensities matrix into circular image
        radius = int(self.image_width_pixels/2)
        warp_flags = cv.WARP_INVERSE_MAP + cv.WARP_POLAR_LINEAR + cv.WARP_FILL_OUTLIERS + cv.INTER_LINEAR
        warped_image = cv.warpPolar(sector_intensities_mod, center=(radius, radius), maxRadius=radius,
                                    dsize=(2 * radius, 2 * radius),
                                    flags=warp_flags)

        return warped_image

    def cluster(self):
        '''Applies a square window clustering method to self.seg_img, and stores the image with the labelled clustered
        pixels as labelled_clustered_img'''

        # Reset class vars
        self.num_clusters = 0
        self.clusters = []

        # Convert window width from meters to pixels
        image_width_pixels = self.seg_img.shape[1] # Assumes square image
        self.window_width_pixels = self.window_width_m * image_width_pixels / (2 * self.range_m)
        self.window_width_pixels = 2*math.floor(self.window_width_pixels/2) + 1 # Window size should be an odd number

        # Ensure widow width is at least 3 pixels
        if self.window_width_pixels < 3:
            print('Clipping clustering block with to 3 pixels (minimum)')
            self.window_width_pixels = 3
        self.comms.notify('PLUME_DETECTOR_CLUSTERING_BLOCK_WIDTH', self.window_width_pixels, pymoos.time())

        # Convert minimum fill from percentage to pixels
        window_rows = self.window_width_pixels
        window_cols = self.window_width_pixels
        area = window_rows*window_cols
        self.cluster_min_pixels = self.cluster_min_fill_percent*area/100

        # Add border padding to image
        row_padding = math.floor(window_rows / 2)
        col_padding = math.floor(window_cols / 2)
        self.seg_img = cv.copyMakeBorder(self.seg_img, row_padding, row_padding, col_padding, col_padding,
                                         cv.BORDER_CONSTANT, None, value=0)
        #self.seg_img = self.seg_img.astype(np.uint8)
```

```
        start = time.time()

        # Call block clustering function
        self.clustered_cores_img, self.cluster_regions_img, self.labelled_regions_img, self.labelled_clustered_img,
        self.num_clusters = block_cluster_fn(self.seg_img, self.window_width_pixels, row_padding, col_padding,
                                             self.cluster_min_pixels)

        # Save clustering time
        end = time.time()
        self.clustering_time_secs = end - start
        print("Plume Detector Clustering time is ", self.clustering_time_secs)
        self.comms.notify('PLUME_DETECTOR_CLUSTERING_TIME_SECS', self.clustering_time_secs, pymoos.time())

        # Compute number of cluster pixels
        binary_clusters = np.zeros_like(self.labelled_clustered_img)
        binary_clusters[self.labelled_clustered_img > 0] = 1 # Pixel values > 0 are set to 1 in the binary image
        num_cluster_pixels = binary_clusters.sum()

        num_clusters = self.labelled_clustered_img.max()
        num_seg_pixels = self.seg_img.sum()
        num_noise_pixels = num_seg_pixels - num_cluster_pixels

        print("Plume Detector block width pixels, min pixels, clusters, num points, cluster points, noise points, clustering
               time: %.2f, %.1f, %d, %d, %d, %d, %.3f "
              % (self.window_width_pixels, self.cluster_min_pixels, num_clusters, num_seg_pixels, num_cluster_pixels,
                 num_noise_pixels, self.clustering_time_secs))

        return

    def calc_cluster_centers(self):
        '''Calculates the cluster centers and radii, and draws them on the output image'''

        self.clusters = [Cluster() for i in range(self.num_clusters)]

        # Calculate cluster centers and radii
        for cluster_idx in np.arange(self.num_clusters):

            # Get indices of cluster pixels.
            indices = np.nonzero(self.labelled_clustered_img == (cluster_idx+1))

            # Calculate center coordinates
            center_row = indices[0].mean()
            center_col = indices[1].mean()

            # Find furthest point & calculate cluster radius
            radius = 0
            center = np.array([center_row, center_col])
            for i in range(len(indices[0])):
                point = np.array([indices[0][i], indices[1][i]])
                dist = np.linalg.norm(point - center)
                if dist > radius:
                    radius = dist

            # Save values in clusters data structure
            self.clusters[cluster_idx].center_row = center_row
            self.clusters[cluster_idx].center_col = center_col
            self.clusters[cluster_idx].radius_pixels = radius

        return

    def get_cluster_center_nav(self):
        '''Gets the vehicle nav data at time that the sonar pinged the cluster center'''

        num_rows = num_cols = self.labelled_clustered_img.shape[0]  # Assumes square image

        for i in np.arange(self.num_clusters):
            # Cluster center coordinates relative to the top left corner of the image
            row = self.clusters[i].center_row
            col = self.clusters[i].center_col

            # Cluster center coordinates relative to the center of the image
            x = row - (num_rows - 1)/2
            y = -1 * (col - (num_cols - 1)/2)

            # Calculate sonar transmit angle at the time that the cluster center was scanned
            theta_rads = math.atan2(y,x)
            theta_grads = round(theta_rads * self.rads_to_grads)

            # Retrieve nav data at the time that the cluster center was scanned, using the sonar transmit angle
            self.clusters[i].nav_x = self.nav_x_history[theta_grads]
            self.clusters[i].nav_y = self.nav_y_history[theta_grads]
```

```
            self.clusters[i].nav_heading = self.nav_heading_history[theta_grads]

        return

    def georeference_clusters(self):
        '''For each cluster center, transforms from image coordinates to instrument, body, local and earth coordinates.
         Also converts each cluster radius from pixels to meters. Coordinates are stored in self.clusters. '''

        num_rows = num_cols = self.labelled_clustered_img.shape[0] # Assumes square image
        meters_per_pixel = (2 * self.range_m) / self.image_width_pixels

        for i in np.arange(self.num_clusters):

            # Retrieve cluster center and radius in image coordinates
            center_row = self.clusters[i].center_row
            center_col = self.clusters[i].center_col
            radius_pixels = self.clusters[i].radius_pixels

            # Calculate cluster radius in meteres
            self.clusters[i].radius_m = radius_pixels * meters_per_pixel

            # Calculate cluster center in instrument coordinates
            instrument_x = (num_rows - 1)/2 - center_row
            instrument_x = instrument_x * meters_per_pixel
            instrument_y = (num_cols - 1)/2 - center_col
            instrument_y = instrument_y * meters_per_pixel

            # Calculate cluster center in body coordinates
            body_x = instrument_x + self.instrument_offset_x_m
            body_y = instrument_y

            # Calculate cluster center in local coordinates
            theta = (self.clusters[i].nav_heading - 90) * np.pi/180
            local_x = self.clusters[i].nav_x + (body_x * math.cos(theta) + body_y * math.sin(theta))
            local_y = self.clusters[i].nav_y + (body_x * -math.sin(theta) + body_y * math.cos(theta))

            # Calculate cluster center in earth (lat,long) coordinates
            dist = math.hypot(local_x, local_y)
            fwd_azimuth = math.degrees(math.atan2(local_x, local_y)) # Use x/y because azimuth is wrt y axis
            long, lat, back_azimuth = (pyproj.Geod(ellps='WGS84').fwd(self.lat_origin, self.long_origin,
                                                                      fwd_azimuth, dist))

            # Save calculations
            self.clusters[i].instrument_x = instrument_x
            self.clusters[i].instrument_y = instrument_y
            self.clusters[i].body_x = body_x
            self.clusters[i].body_y = body_y
            self.clusters[i].local_x = local_x
            self.clusters[i].local_y = local_y
            self.clusters[i].lat = lat
            self.clusters[i].long = long

            self.cluster_centers.append((local_x,local_y,self.clusters[i].nav_x,self.clusters[i].nav_y))

        return

    def output_sorted_cluster_centers(self):
        '''Sorts the clusters in order of radius (largest to smallest) and sets the cluster outputs for the app'''

        self.cluster_centers_string = ""

        # Sort list of clusters based on radius (largest to smallest)
        self.sorted_clusters = sorted(self.clusters, key=lambda cluster_i: cluster_i.radius_m, reverse=True)

        # Assemble string with cluster centers list
        # Sample format: <cluster 1 x>,<cluster 1 y>,<cluster 1 radius>:<cluster 2 x>,<cluster 2 y>,<cluster 2 radius>
        centers = ""
        for i in np.arange(self.num_clusters):
            cluster = self.sorted_clusters[i]
            # Add cluster <local_x,local_y,radius_m> to string
            centers = centers + "{:.2f}".format(cluster.local_x) + ","
            centers = centers + "{:.2f}".format(cluster.local_y) + ","
            centers = centers + "{:.2f}".format(cluster.radius_m)

            # Add colon delimiter between info for each cluster
            if i < (self.num_clusters-1):
                centers = centers + ":"
        self.cluster_centers_string = centers
        #print(self.cluster_centers_string)

        # Output number of clusters and cluster centers list
```

```
        self.comms.notify('PLUME_DETECTOR_NUM_CLUSTERS', self.num_clusters, pymoos.time())
        self.comms.notify('PLUME_DETECTOR_CLUSTER_CENTERS_LIST', self.cluster_centers_string, pymoos.time())

        # Output the center coordinates  and radius of each cluster. If there are more outputs than detected clusters,
        # those outputs are set to 0.
        for i in range(self.num_cluster_outputs):

            # Construct variable name using cluster number
            cluster_num = i+1 #Numbering of cluster outputs starts at 1
            cluster_x_name = 'PLUME_DETECTOR_CLUSTER_'+ str(cluster_num)+ '_X_M'
            cluster_y_name = 'PLUME_DETECTOR_CLUSTER_' + str(cluster_num) + '_Y_M'
            cluster_radius_name = 'PLUME_DETECTOR_CLUSTER_' + str(cluster_num) + '_RADIUS_M'

            if i < self.num_clusters:
                # Output the center coordinates  and radius of each cluster.
                cluster = self.sorted_clusters[i]
                self.comms.notify(cluster_x_name, cluster.local_x, pymoos.time())
                self.comms.notify(cluster_y_name, cluster.local_y, pymoos.time())
                self.comms.notify(cluster_radius_name, cluster.radius_m, pymoos.time())

            else:
                # Outputs which number more than the detected clusters are set to 0
                self.comms.notify(cluster_x_name, 0, pymoos.time())
                self.comms.notify(cluster_y_name, 0, pymoos.time())
                self.comms.notify(cluster_radius_name, 0, pymoos.time())

    def save_images(self):
        '''Saves a set of original clustering images, as well as modified high-contrast viewable images'''

        ## Save original images
        dir = self.orig_images_path

        filename = "Scan_" + str(self.num_scans) + "_Im1_Segmented_Unwarped.png"
        cv.imwrite(os.path.join(dir, filename), self.seg_scan_snapshot)

        filename = "Scan_" + str(self.num_scans) + "_Im2_Segmented_Warped.png"
        cv.imwrite(os.path.join(dir, filename), self.seg_img)

        filename = "Scan_" + str(self.num_scans) + "_Img3_ClusteredCores.png"
        cv.imwrite(os.path.join(dir, filename), self.clustered_cores_img)

        filename = "Scan_" + str(self.num_scans) + "_Img4_LabelledRegions.png"
        cv.imwrite(os.path.join(dir, filename), self.labelled_regions_img)

        filename = "Scan_" + str(self.num_scans) + "_Img5_Clustered.png"
        cv.imwrite(os.path.join(dir, filename), self.labelled_clustered_img)

        ## Increase image contrast and save
        dir = self.viewable_images_path

        warped = self.create_sonar_image(self.scan_intensities)
        filename = "Scan_" + str(self.num_scans) + "_Img1_Input.png"
        cv.imwrite(os.path.join(dir, filename), warped)

        filename = "Scan_" + str(self.num_scans) + "_Img2_Segmented.png"
        cv.imwrite(os.path.join(dir, filename), 255*self.seg_img)

        filename = "Scan_" + str(self.num_scans) + "_Img3_ClusteredCores.png"
        cv.imwrite(os.path.join(dir, filename), 255*self.clustered_cores_img)

        filename = "Scan_" + str(self.num_scans) + "_Img4_ClusterRegions.png"
        cv.imwrite(os.path.join(dir, filename), 255*self.cluster_regions_img)

        # Convert labelled clustered image to a black and white image
        filename = "Scan_" + str(self.num_scans) + "_Img5_Clustered.png"
        self.clustered_img = np.zeros_like(self.labelled_clustered_img)
        self.clustered_img[self.labelled_clustered_img > 0] = 255 # Pixel values > 0 are set to 255
        cv.imwrite(os.path.join(dir, filename), self.clustered_img)

    def save_text_data(self):
        ''' Create test file with summary info for the scan, as well as detailed info for each cluster '''

        # Save text file with the viewable images
        filename = "Scan_" + str(self.num_scans) + "_Data.txt"
        file_path = os.path.join(self.viewable_images_path, filename)

        with open(file_path, "w") as file:

            # Write summary data
            date_time = datetime.strftime(datetime.now(), '%Y-%m-%d %H:%M:%S')
            file.write("Timestamp: " + date_time + "\n")
```

```
            file.write("Num Clusters: " + str(self.num_clusters) + "\n")
            file.write("Total processing time: " + str(self.total_processing_time_secs) + " secs\n")
            file.write("Clustering time: " + str(self.clustering_time_secs) + " secs\n\n")

            # Write data for each cluster
            for i in np.arange(self.num_clusters):
                data = self.sorted_clusters[i].serialize()
                file.write(str(data) + "\n")


if __name__ == "__main__":

    plume_detector = PPlumeDetector()
    plume_detector.run()
```

Appendix B

Block Clustering Code

The following is the Cython implementation of the block clustering algorithm presented in this thesis. It is available online in the pPlumeDetector git repository [60].

```
import numpy as np
from skimage import measure
cimport numpy as cnp

cnp.import_array()

DTYPE = np.uint8

ctypedef cnp.uint8_t DTYPE_t

def block_cluster_fn(cnp.ndarray[DTYPE_t, ndim=2] seg_img, int window_width_pixels, int row_padding, int col_padding,
cluster_min_pixels):
    window_rows = window_width_pixels
    window_cols = window_width_pixels

   # Initialize image matrices
    cdef int rows = seg_img.shape[0]
    cdef int cols = seg_img.shape[1]
    cdef cnp.ndarray clustered_cores_img = np.zeros((rows, cols), dtype=DTYPE)
    cdef cnp.ndarray cluster_regions_img = np.zeros((rows, cols), dtype=DTYPE)
    cdef cnp.ndarray labelled_clustered_img = np.zeros((rows, cols), dtype=DTYPE)
    cdef cnp.ndarray window_ones = np.ones((int(window_rows), int(window_cols)), dtype=DTYPE)

    cdef int start_row, end_row, start_col, end_col

    # Create clustered_cores_img, identifying pixels at the center of high density windows.
    # Also create cluster_regions_img, identifying high density regions. Note: output matrices are zero padded
    for row in range(row_padding, rows-row_padding, 1):
        for col in range(col_padding, cols-col_padding, 1):
            if seg_img[row, col]:
                start_row = row - row_padding
                end_row   = row + row_padding + 1
                start_col = col - col_padding
                end_col   = col + col_padding + 1
                filled = (seg_img[start_row:end_row, start_col:end_col]).sum()
                if filled > cluster_min_pixels:
                    clustered_cores_img[row,col] = 1
                    cluster_regions_img[start_row:end_row, start_col:end_col] = window_ones

    # Identify and label separate regions
    labelled_regions_img, num_clusters = measure.label(cluster_regions_img, return_num=True, connectivity=2)

    # Mask input image with the labelled regions image to create the labelled clustered image
    labelled_clustered_img = labelled_regions_img * seg_img

    return clustered_cores_img, cluster_regions_img, labelled_regions_img, labelled_clustered_img, num_clusters
```

Appendix C

Scott Inlet Project Software

C.1 Introduction

This appendix contains the documentation for the Scott Inlet Project's software developed for Memorial University's Explorer AUV. It is not intended to be a stand-alone document, and should be read along with [13], which contains the higher-level system and mission descriptions.

The Explorer AUV was equipped with a Ping360 Sonar, and two UviLux fluorometers measuring PAH and CDOM fluorescence. The comprehensive mission used data from these sensors to determine the optimal sampling location, and consisted of three main phases: Search, Survey, and Sample. During the search phase, a coarse pre-planned search of the mission area was conducted. Data collected by the Ping360 sonar during this phase was used to adaptively identify areas of interest where detailed lawnmower surveys were required. Similarly, data collected by the UviLux PAH fluorometer during the search and survey phases was used to adaptively identify the optimal sampling waypoint. The final sample phase involved collection of the water samples at this location.

C.2 Overview

This section contains brief descriptions of each application in the Scott Inlet Project's software suite. It assumes that the reader is familiar with the MOOS-IvP software suite [79], and how

the backseat driver system is realized on Memorial University's Explorer AUV [82]. Fig. C.1 contains a simplified diagram of the applications and how they interact with each other.

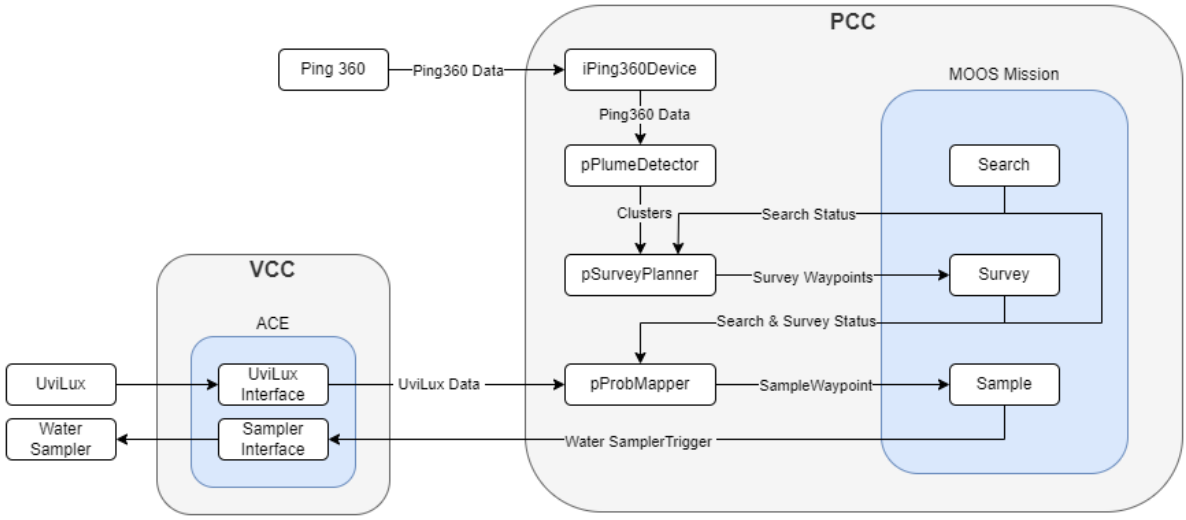


Fig. C.1: Simplified diagram of the software applications and how the interact with each other

The following modules were added to the manufacturer's Automated Control Engine (ACE) software on the Vehicle Control Computer (VCC):

- UviLux Interface: Interface for the UviLux CDOM and PAH fluorometers. It extracts and outputs the information in each sensor's data message.
- Water Sampler Interface: Controls the water sampler. When the water sampler trigger is received, it sequentially actuates the eight sampling syringes to collect water samples.

The following MOOS apps run on the PCC and work together to adaptively control the mission. All the apps except for pMoosCrossing and pBSDOverride are newly developed software modules for the Scott Inlet project:

- pMoosCrossing: The communication bridge between the VCC and PCC. It allows MOOS applications on the PCC to communicate with the VCC through the MOOSDB.
- pBSDOverride: Sets flags to activate the MOOS mission once the VCC indicates that the PCC is allowed to take control.

- iPing360Device [80]: The Ping360 sonar interface. It controls the sonar's pinging, receives the acoustic data, and publishes it to the MOOSDB.
- pPlumeDetector [60]: An implementation of the plume detection algorithm presented in this thesis. It identifies the plume as high-density clusters in the Ping360 sonar data and publishes the geo-referenced cluster centers.
- pSurveyPlanner: The survey waypoint generator. It organizes surveys that target areas in which the pPlumeDetector has identified clusters of interest. It generates the survey waypoints at the end of the search phase, where each survey waypoint corresponds to the center of a survey area.

  pProbMapper: The sampling waypoint generator, and backup survey waypoint generator. Once the survey phase is complete, it constructs a probability map based on the measured PAH fluorescence, and generates a sampling waypoint corresponding to the location where oil is most likely to be found. It also generates a survey waypoint at the end of search phase if the pPlumeDetector did not detect any clusters; this waypoint waypoint corresponds to the location where the highest PAH fluorescence was measured.

C.3 Mission Modes

The built-in MOOS-Ivp application, pHelmIvp, is configured to execute the custom MOOS mission. The comprehensive mission has five modes: Search, Survey, Sample, Loiter, and Return. Each mode activates a different set of behaviours to achieve the goal:

- Search: Performs a pre-planned search of the mission area. During this phase, the AUV visits a series of waypoints and performs a bowtie pattern at each waypoint.
- Survey: Performs surveys at locations which are determined adaptively based on data collected in the search phase. During this phase, the AUV visits the adaptively generated waypoints and performs a lawnmower survey pattern at each waypoint.
- Sample: Transits to the adaptively generated sampling waypoint and triggers the water sampler upon arrival. Conducts a bowtie maneuvering pattern while the water samples are being collected.

- Loiter: Circles around the current location, allowing time for the data to be processed after the search and survey phases. It also ensures that the mission always ends correctly; if an app fails to publish the required variables, the loiter times-out and causes a switch to return mode.
- Return: Returns to a known safe location and relinquishes control to end the mission.

Fig. C.2 below shows the mode transitions for the comprehensive mission; details of the transitions are outlined in the following section.

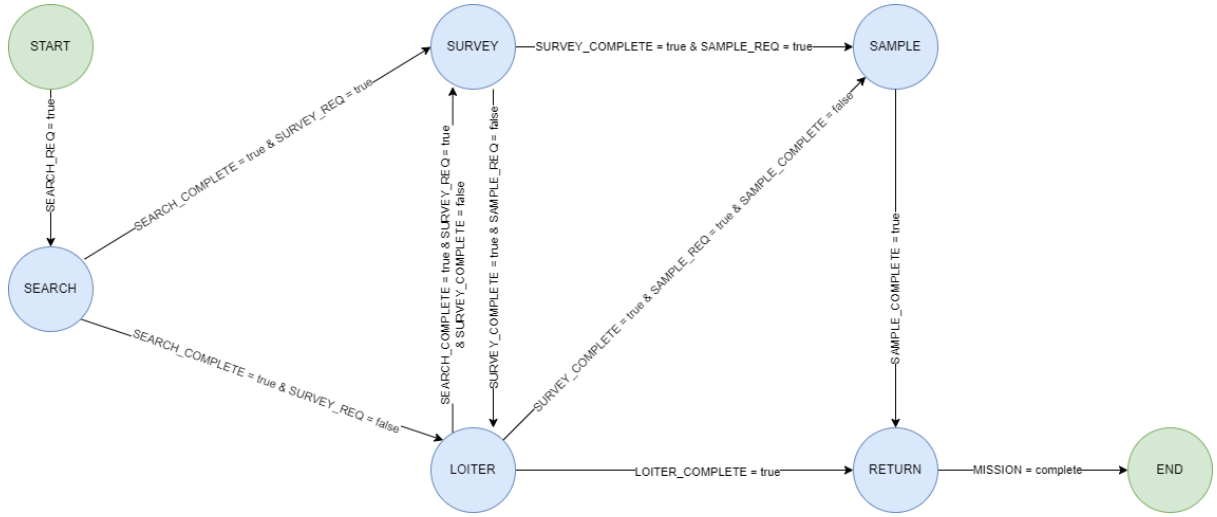


Fig. C.2: Mode transitions for the comprehensive mission

C.4 Application Interactions

Since there are quite a few software elements interacting with each other, the following procedure does not show all possible interactions, but is a starting point to follow along and understand the code.

1) Starting the MOOS mission
   a) The VCC indicates that the PCC can take control by setting AUV_OVERRIDE_ALLOW = true
   b) The pBSDOverride app sets the following variables, which activate the MOOS mission:
      i) AUV_BSD_STATE = 1, which indicates to the VCC that the PCC has taken control

ii) MOOS_MANUAL_OVERRIDE = true, which changes the Helm state from 'Park' to 'Drive'

iii) DEPLOY = true, which causes the Helm to set the mode to ACTIVE

2) Search mode

a) By default, SEARCH_REQ is set to true. The Helm processes this request for search mode, and sets the mode to ACTIVE: SEARCH:SEARCH_WAYPOINT

b) While the mode is ACTIVE: SEARCH, the search_active behaviour sets the SEARCH_ACTIVE flag to true. This causes:

i) pSurveyPlanner to record clusters detected by pPlumeDetector

ii) pProbMapper to record the fluorescence readings

c) When the mode is set to ACTIVE: SEARCH:SEARCH_WAYPOINT the search_waypoint behaviour is activated

d) Upon reaching the waypoint, the search_waypoint behaviour sets SEARCH_BOWTIE_REQ = true, which places a request for a search bowtie. It also updates SEARCH_BOWTIE_UPDATE to specify a bowtie around the current position.

e) The Helm sets the mode to ACTIVE: SEARCH: SEARCH_BOWTIE, which activates the search_bowtie behaviour. The search_bowtie behaviour uses the SEARCH_BOWTIE_UPDATE.

f) Upon completion of the bowtie, the search_bowtie behaviour sets SEARCH_BOWTIE_REQ=false, which places a request to end the search bowtie

g) The Helm sets the mode to ACTIVE: SEARCH:SEARCH_WAYPOINT

h) Steps c to g are repeated, alternating between waypoint and bowtie behaviours, until the last waypoint is reached

i) When the last waypoint is reached, the search waypoint behaviour sets SEARCH_WAYPOINT_COMPLETE to true

j) When the subsequent search_bowtie sets SEARCH_BOWTIE_COMPLETE to true, the search_complete behaviour sets SEARCH_COMPLETE to true

3) Survey Waypoint Generation during LOITER mode

a) In the absence of a request to activate survey or sample mode, the Helm sets the mode to LOITER, which activates the waypt_loiter behaviour, causing the vehicle to circle around its current position.

b) While the vehicle is loitering, the pSurveyPlanner computes the survey waypoints. This computation is triggered when the SEARCH_COMPLETE flag is set to true.

c) If the pSurveyPlanner has recorded at least one cluster detection during the search phase, it computes the survey waypoints and updates:

   i) The SURVEY_UPDATES variable with the survey waypoints

   ii) The NUM_SURVEY_WAYPOINTS variable with the number of survey waypoints

   iii) The SURVEY_REQ variable to true, to request survey mode

d) If the pSurveyPlanner has not recorded any clusters during the search phase, it sets GEN_FLUOR_SURVEY_WAYPOINTS_CMD to true.

   i) This triggers the pProbMapper to generate one survey waypoint, and update the same variables as in step c.

4) Survey Mode

a) The Helm processes the survey mode request (SURVEY_REQ=true), and sets the mode to ACTIVE: SURVEY:SURVEY_WAYPOINT

b) While the mode is ACTIVE: SURVEY, the survey_active behaviour sets the SURVEY_ACTIVE flag to true.

   i) This causes pProbMapper to record the fluorescence readings

c) When the mode is set to ACTIVE: SURVEY:SURVEY_WAYPOINT the search_waypoint behaviour is activated

d) Upon reaching the waypoint, the survey_waypoint behaviour sets SURVEY_LAWNMOWER_REQ = true, which places a request for a search bowtie. It also updates SURVEY_LAWNMOWER_UPDATE to specify a lawnmower pattern around the current position.

e) The Helm sets the mode to ACTIVE: SURVEY: SURVEY_LAWNMOWER, which activates the survey_lawnmower behaviour. They survey_lawnmower behaviour uses the SURVEY_LAWNMOWER_UPDATE.

f) Upon completion of the lawnmower, the survey_lawnmower behaviour sets SURVEY_LAWNMOWER_REQ to false, which places a request to end the survey lawnmower.

g) The Helm sets the mode to ACTIVE: SURVEY:SURVEY_WAYPOINT

h) Steps c to g are repeated, alternating between waypoint and bowtie behaviours, until the last waypoint is reached

i) When the last waypoint is reached, the search waypoint behaviour sets SURVEY_WAYPOINT_COMPLETE to true

j) When the subsequent survey_lawnmower sets SURVEY_LAWNMOWER_COMPLETE to true, the survey_complete behaviour sets SURVEY_COMPLETE to true

5) Sample Waypoint Generation during LOITER MODE

a) In the absence of a sample The Helm sets the mode to LOITER, which activates the waypt_loiter behaviour, causing the vehicle to circle around its current position.

b) While the vehicle is loitering, the pProbMapper computes the sample waypoints. This computation is triggered when the SURVEY_COMPLETE flag is set to true.

c) If the pProbMapper successfully computes the sampling waypoint, it updates:

i) The FLUORESCENCE_WAYPOINTS variable with the sampling waypoint

ii) The SAMPLE_REQ variable to true, to request sample mode

d) If the pProbMapper fails to compute the sampling waypoint, it sets RETURN_REQ to true, to request return mode (Go to Step 7).

6) Sample Mode

a) The Helm processes the sample mode request (SAMPLE_REQ=true), and sets the mode to ACTIVE: SAMPLE:SAMPLE_WAYPOINT. This activates the sample_waypoint behaviour.

b) Upon reaching the waypoint, the sample_waypoint behaviour sets SAMPLE_BOWTIE_REQ = true, which places a request for a sample bowtie. It also updates SAMPLE_BOWTIE_UPDATE to specify a bowtie pattern around the current position.

c) When the first waypoint is reached, the sample_waypoint_complete behaviour sets SAMPLE_WAYPOINT_COMPLETE to true

d) The Helm processes the sample bowtie request and sets the mode to ACTIVE: SAMPLE: SAMPLE_BOWTIE, which activates the sample_bowtie behaviour. The sample_bowtie behaviour uses the SAMPLE_BOWTIE_UPDATE.

e) While the sample_bowtie behaviour is active, it sets WATER_SAMPLER_TRIGGER to 1. This trigger is sent to the VCC to initiate water sampling.

f) Upon completion of the bowties, the sample_bowtie behaviour sets SAMPLE_BOWTIE_COMPLETE to true.

g) The sample_complete behaviour then sets SAMPLE_COMPLETE to true, as SAMPLE_WAYPOINT_COMPLETE is true

h) The set_return_req behaviour sets RETURN_REQ to true when SAMPLE_COMPLETE is set to true

7) Return Mode

a) The Helm processes the return mode request (RETURN_REQ=true), and sets the mode to ACTIVE:RETURN

b) The return behaviour, which transits to the return waypoint, is activated in return mode.

c) When the return waypoint is reached, the behaviour ends the MOOS mission by updating:

i) DEPLOY to false, which changes to mode to INACTIVE

ii) MOOS_MANUAL_OVERRIDE to true, which changes the Helm state from 'Drive' to 'Park'

iii) MISSION to complete, which indicates to the VCC that the PCC has relinquished control.

C.5 Application Interfaces

The following sections lists the relevant publication and subscription variables for the PCC's MOOS applications.

C.5.1 pMoosCrossing

Variables published by pMoosCrossing:

Variable Name	Type	Unit	Description
AUV_OVERRIDE_ALLOW	Integer	-	Set to '1' when MOOS is allowed to take control. Otherwise '0'.
AUV_BSD_CTRL	Integer	-	Set to '1' when a MOOS mission is running and MOOS has control of the vehicle's trajectory. Otherwise '0'.
NAV_LAT	Double	Degrees	The vehicle's current latitude
NAV_LONG	Double	Degrees	The vehicle's current longitude
NAV_SPEED	Double	m/s	The vehicle's current speed
NAV_HEADING	Double	Degrees	The vehicle's current heading
NAV_DEPTH	Double	Meters	The vehicle's current depth
NAV_ALTITUDE	Double	Meters	The vehicle's current altitude
UVILUX_CDOM_FLUORESCENCE, UVILUX_PAH_FLUORESCENCE	Double	QSU	Fluorescence measured by the CDOM and PAH UviLux sensors. Fluorescence is reported in Quinine Sulphate Units (QSU), where 1 QSU is equivalent to the

			fluorescence intensity recorded from 1 ppb quinine sulphate at an excitation wavelength of 347.5 nm and an emission wavelength of 450 nm
UVILUX_CDOM_EHT, UVILUX_PAH_EHT	Double	-	The UviLux handbook [83] does not provide any description of what this is.
UVILUX_CDOM_VIN, UVILUX_PAH_VIN	Double	Volts	Internal 15V rail Voltage of the CDOM and PAH UviLux sensors
UVILUX_CDOM_VREF, UVILUX_PAH_VREF	Double	Volts	Sensor reference voltage
UVILUX_CDOM_TEMP, UVILUX_PAH_TEMP	Double	℃	Onboard Temperature of the CDOM and PAH UviLux sensors
UVILUX_CDOM_QUALITY_FLAG UVILUX_PAH_QUALITY_FLAG	Integer	-	Bit-packed data quality flags. The UviLux handbook [83] does not provide a detailed breakdown of what each of the bits mean. However, it appears that if any of the bits are set to '1', it indicates that the sensor is operating outside of 'normal' conditions. Bit 0 = ADC Clipping Bit 1 = EHT Changed Bit 2 = Low Signal Bit 3 = High Signal, Bit 5 = Negative Fluorescence, Bit 6 = Sampling Error

C.5.2 pNodeReporter

Variables published by pNodeReporter:

Variable Name	Type	Unit	Description
NAV_X	Double	Meters	Vehicle's position measured in eastings (NAV_X) and northings (NAV_Y) from the local frame origin. The origin is defined by the LatOrigin & LongOrigin parameters in the *.moos file. The pNodeReporter app subscribes to the NAV_LAT and NAV_LONG variables, and converts them from the global frame to the local frame.
NAV_Y	Double	Meters	

C.5.3 iPing360Device

The iPing360Device app is configured using the Ping360.ini configuration file, which must be in the same directory as the iPing360Device.py file.

The Ping360.ini file parameters are as follows:

Parameter	Description
port_type	(required) The port type to connect to, must be one of 'udp' or 'serial'
serial_port	(optional, default: /dev/ttyS0) The serial port to connect to, if the PORT_TYPE is set to 'serial'
baudrate	(optional, default: 115200) Serial communications baud rate, must be one of 2400, 4800, 9600, 19200, 38400, 57600, or 115200.
sonar_ip	(optional, default: 192.168.0.100) IP address of the sonar for connecting via a UDP link
udp_port	(optional, default: 12345) Port number for connecting via a UDP link
prefix	(optional, default = '') Message names subscribed for published by iPing360Device will be prefixed by this string. If not included, an underscore will inserted as the last prefix character. If a blank value is provided, no prefix or underscore will precede variable name in the publication.
log_file_dir	(optional, default = './') Directory for saving log files

The iPing360Device application subscribes to the MOOS messages listed in the table below and responds to run-time changes. The Ping360.ini file parameter prefix defines the [prefix] text in the MOOS messages. For example, if prefix=PING360, the app subscribes for the PING360_TRANSMIT_ENABLE variable.

Input Variable	**Type**	**Range**	**Description**
[PREFIX_]DEVICE_COMMS_ENABLE	Integer	0-1	Upon enable, initializes communication with the sonar & configures it. The default is 0.
[PREFIX_]TRANSMIT_ENABLE	Integer	0-1	Enables pinging when set to 1. The default is 0.
[PREFIX_]START_ANGLE_GRADS	Integer	0 to 400 gradians	Scan sector start angle (inclusive). The scan sector is defined by a clockwise rotation from the start angle to the stop angle. The default is 350 gradians (i.e. 315 degrees)
[PREFIX_]STOP_ANGLE_GRADS	Integer	0 to 400 gradians	Scan sector stop angle (inclusive). The scan sector is defined by a clockwise rotation from the start angle to the stop angle. The default is 50 gradians (i.e 45 degrees)
[PREFIX_]NUM_STEPS	Integer	1 to 10 gradians	Number of 0.9 degree motor steps between pings for auto scan (1 to 10 gradians is 0.9 to 9.0 degrees). The default is 1.
[PREFIX_]GAIN	Integer	0: low, 1: normal, 2: high	Analog gain setting.The default is 'normal'
[PREFIX_]RANGE	Float	0m to 50m	Distance from the sonar to scan signals. Smaller ranges will scan faster as the receiver does not have to wait as long to receive a response. The default is 15m.

Input Variable	Type	Range	Description
[PREFIX_]SPEED_OF_SOUND	Float	1450m/s to 1550 m/s	The speed of sound to be used for distance calculations. This should be 1500 m/s in salt water, 1450 m/s in fresh water. The default is 1500m/s.
[PREFIX_]TRANSMIT_FREQUENCY	Integer	500kHz to 1000kHz	Acoustic operating frequency. Although the frequency range is 500kHz to 1000kHz, however it is only practical to use say 650kHz to 850kHz due to the narrow bandwidth of the acoustic receiver. The default is 750kHz
[PREFIX_]NUMBER_OF_SAMPLES	Integer	1 - 1200	Number of samples per reflected signal. The default is 600
[PREFIX_]LOG_ENABLE	Integer	0-1	When set to 1, creates a new log file in LOG_FILE_DIR and logs ping data to it. The file can be replayed with PingViewer. The file name format is ping360_YYYMMDD_HHMMSS.bin
[PREFIX_]DEBUG_ENABLE	Integer	0-1	Enables printing of verbose debug information, such as the complete ping data message

The table below lists the output variables which the iPing360Device app publishes to the MOOSDB.

Output Variable	**Type**	**Description**
[PREFIX_]PING_DATA	Binary	'2301 auto_device_data' message containing the most recent ping intensity data.
[PREFIX_]STATE	String	Indicates the state of the app: DB Disconnected, DB Connected, Ready to Transmit or Transmitting
[PREFIX_]LAST_ERROR	String	Indicates the most recent error that occured
[PREFIX_]LOG_STATUS	String	Indicates logging status. For example: Disabled, Logging, Error (failed to open file)
[PREFIX_]TRANSMIT_ANGLE_GRADS	Float	The current transmit angle, in gradians
[PREFIX_]TRANSMIT_ANGLE_DEGS	Float	The current transmit angle, in degrees

C.5.4 pPlumeDetector

Variables published by pPlumeDetector:

Variable Name	Type	Unit	Description
PLUME_DETECTOR_NUM_SCANS	Integer	-	Number of Ping360 sector scans completed. Incremented every time the start or stop angle is reached.
PLUME_DETECTOR_CLUSTERING_BLOCK_WIDTH	Integer	Pixels	Width of the clustering block
PLUME_DETECTOR_CLUSTER_<NUM>_X_M	Double	Meters	Center coordinates and radius of the 5 largest clusters. The clusters are sorted by radius, with the largest clusters first. If less than 5 clusters are detected, the extra outputs are all set to 0. The center coordinates are measured in the local frame.
PLUME_DETECTOR_CLUSTER_<NUM>_Y_M			
PLUME_DETECTOR_CLUSTER_<NUM>_RADIUS_M	Double	Meters	
PLUME_DETECTOR_NUM_CLUSTERS	Integer	-	Number of detected clusters in the sector scan
PLUME_DETECTOR_CLUSTER_CENTERS_LIST	String	-	String with a list of all the cluster centers (local frame) and radii. The information for each cluster is separated by a colon. Format for three clusters: X1,Y1,Radius1:X2,Y2,Radius2:X3,Y3,Radius3 The clusters are sorted by radius, with the largest clusters first.
PLUME_DETECTOR_STATE_STRING	String	-	Indicates what the app is currently doing. It is set to:

			DB_CONNECTED once the app connects to the MOOS DB. Indicates that the app is waiting for all the configuration variables to be set. STANDBY once the app is configured, but transmit is not enabled on the Ping360. Indicates that the app is ready, but not processing data. ACTIVE once the app is configured and transmit is enabled on the Ping360. Indicates that the app is processing data.
PLUME_DETECTOR_STATE_NUM	Integer	-	Numeric state, set to: '1' if the state is DB_CONNECTED '2' if the state is STANDBY '3' if the state is ACTIVE
PLUME_DETECTOR_STATUS_STRING	String	-	Indicates any errors, otherwise set to 'GOOD'
PLUME_DETECTOR_STATUS_NUM	Integer	-	Numeric status, set to: '1' if the status is GOOD '-1' if the status is DB_REGISTRATION_ERROR '-2' if the status is TIMEOUT '-3' if the status is PROCESSING_ERROR

C.5.5 pHelmIvp

Variables published by pHelmIvP, as per the mission's behaviour file configuration:

Variable Name	Type	Unit	Description
MODE	String	-	o ROOT |--o INACTIVE |--o ACTIVE |--o SEARCH |--o SEARCH_WAYPOINT |--o SEARCH_BOWTIE |--o SURVEY |--o SURVEY_WAYPOINT |--o SURVEY_LAWNMOWER |--o SAMPLE |--o SAMPLE_WAYPOINT |--o SAMPLE_BOWTIE |--o LOITER |--o RETURN
SEARCH_ACTIVE	String	-	Set to 'true' while the search phase is in progress
SURVEY_ACTIVE	String	-	Set to 'true' while the survey phase is in progress
SAMPLE_ACTIVE	String	-	Set to 'true' while the sample collection phase is in progress
SEARCH_COMPLETE	String	-	Set to 'true' once the search phase is complete

SURVEY_COMPLETE	String	-	Set to ‘true’ once the survey phase is complete
SAMPLE_COMPLETE	String	-	Set to ‘true’ once the sample collection phase is complete

Variables subscribed for by pHelmIvP, as per the behaviour file configuration:

Variable Name	**Type**	**Unit**	**Description**
SURVEY_REQ	String	-	Once the search phase is completed, this flag is set to ‘true’ by an app (pSurveyPlanner) if at least one survey waypoint is identified. Otherwise set to ‘false’.
NUM_SURVEY_WAYPOINTS	Integer	-	Number of computed survey waypoints.
SURVEY_UPDATES	String	-	The survey waypoints for the waypoint behaviour, determined using data collected in the search phase. The format should be as follows: “polygon = X1,Y1 : X2,Y2 : X3,Y3” For example: “polygon = 50,150 : 60,160 : 70,170”
SAMPLE_REQ	String	-	Once the survey phase is completed, this flag is set to ‘true’ by an app if at least one water sample collection waypoint has been identified. Otherwise set to ‘false’.
FLUORESCENCE_WAYPOINTS	String	-	The updated sample waypoints for the waypoint behaviour. The format should be as follows: “polygon = X1,Y1 : X2,Y2 : X3,Y3”

			For example: “polygon = 50,150 : 60,160 : 70,170”
RETURN_REQ	String	-	If this flag is set to ‘true’ the vehicle returns to a preset waypoint and the mission completes.

C.5.6 pSurveyPlanner

The pSurveyPlanner organizes surveys that target areas in which the pPlumeDetector has identified clusters of interest; each survey area contains one or more detected clusters. The survey planner minimizes the number of surveys by using agglomerative clustering to group together pPlumeDetector clusters which are near each other.

It generates the survey waypoints at the end of the search phase, where each survey waypoint corresponds to the center of a survey area. The mission's survey module then generates a lawnmower pattern around each survey waypoint to execute the survey.

Variables subscribed for by pSurveyPlanner:

Variable Name	Type	Unit	Description
PLUME_DETECTOR_CLUSTER_CENTERS_LIST	String	-	A list containing the center coordinates and radius of the clusters detected in the most recent sector scan. If the detections occur during the search phase, the information is added to an internal list of clusters.
SEARCH_ACTIVE	String	-	While this flag is set to to 'true', all detected clusters are recorded. Ensures that clusters detected while the vehicle is diving or surfacing are not recorded.
SEARCH_COMPLETE	String	-	When this mission status flag is set to 'true', the app is triggered to compute and output the survey waypoints.
SURVEY_AREA_WIDTH	Double	Meters	Width of the area covered by the survey pattern.

			Used as the cut-off for merging groups in the agglomerative clustering algorithm. If the distance between two points/groups is more than the survey area width, merging is not performed.
NAV_X	Double	Meters	Vehicle's current position measured in eastings (NAV_X) and northings (NAV_Y) from the local frame origin.
NAV_Y	Double	Meters	

Variables published by pSurveyPlanner:

Variable Name	**Type**	**Unit**	**Description**
GEN_FLUOR_SURVEY_WAYPOINTS_CMD	String	-	If no clusters were detected by the pPlumeDetector during the search phase, this variable is set to 'true' when the search phase is complete. It indicates to the pProbMapper that the survey waypoint should be generated based on the fluorescence data.
NUM_SURVEY_WAYPOINTS	Integer	-	Number of computed survey waypoints.
SURVEY_UPDATES	String	-	A list of the computed survey waypoints. The format is as follows: "polygon = X1,Y1 : X2,Y2 : X3,Y3" For example: "polygon = 50,150 : 60,160 : 70,170"

			A greedy algorithm is used to determine the waypoint order. The first waypoint is the one closest to the vehicle's current position. Each subsequent waypoint is then the waypoint which is closest to the last waypoint. If only one survey waypoint is identified, two waypoints are output in the SURVEY_UPDATES string. This is to work-around the bug in the waypoint behavior where it does not update the waypoint flags correctly if there is only one waypoint.
SURVEY_REQ	String	-	Set to 'true' when a survey waypoint is generated.
SURVEY_PLANNER_STATE_STRING	String	-	Indicates what the app is currently doing. It is set to: DB_CONNECTED once the app connects to the MOOS DB. Indicates that the app is waiting for all the configuration variables to be set. STANDBY once the app is configured, but the search phase is not active. ACTIVE once the search phase is active. In this state, the app saves all cluster data.
SURVEY_PLANNER_STATE_NUM	Integer	-	Numeric state, set to: '1' if the state is DB_CONNECTED '2' if the state is STANDBY '3' if the state is ACTIVE
SURVEY_PLANNER_STATUS_STRING	String	-	Indicates any errors, otherwise set to 'GOOD'
SURVEY_PLANNER_STATUS_NUM	Integer	-	Numeric status, set to: '1' if the status is GOOD '-1' if the status is DB_REGISTRATION_ERROR '-2' if the status is PROCESSING_ERROR

C.5.7 pProbMapper

Variables subscribed for by pProbMapper:

Variable Name	Type	Unit	Description
UVILUX_CDOM_FLUORESCENCE, UVILUX_PAH_FLUORESCENCE	Double	QSU	Fluorescence measured by the CDOM and PAH UviLux sensors.
SEARCH_ACTIVE, SURVEY_ACTIVE	String	-	When either of these flags are set to 'true', the fluorescence data is recorded in the map. Ensures that data measured while the vehicle is diving or surfacing is not recorded.
GEN_FLUOR_SURVEY_WAYPOINTS_CMD	String	-	If this flag is set to true, the app generates a survey waypoint corresponding to the location where the highest fluorescence was detected during the search phase.
SURVEY_COMPLETE	String	-	When this mission status flag is set to 'true', the app is triggered to compute and output the sample collection waypoint(s) based on the probability map.
NAV_X	Double	Meters	Vehicle's current position measured in eastings (NAV_X) and northings (NAV_Y) from the local frame origin.
NAV_Y	Double	Meters	

Variables published by pProbMapper:

Variable Name	**Type**	**Unit**	**Description**
SURVEY_REQ	String	-	Set to 'true' if a survey waypoint is generated.
SAMPLE_REQ	String	-	Set to 'true' if at least one sample waypoint is identified. Otherwise set to 'false''.
RETURN_REQ	String	-	Set to 'true' if no survey or sample waypoints are identified.
SURVEY_UPDATES	String	-	The survey waypoint corresponding to the location where the highest fluorescence was detected during the search phase.
NUM_SURVEY_WAYPOINTS	String	-	Set to '1' when the SURVEY_UPDATES variable is published.
FLUORESCENCE_WAYPOINTS	String	-	The updated sample waypoints for the waypoint behaviour. See below for the waypoint string format.

The format of the waypoint strings is as follows:

"polygon = X1,Y1 : X2,Y2 : X3,Y3"

For example:

"polygon = 50,150 : 60,160 : 70,170"

The app outputs two waypoints. However, the expectation is that only the first one is used. This is to work-around the bug in the waypoint behavior where it does not update the waypoint flags correctly if there is only one waypoint.

References

[1] J. Beyer, H. C. Trannum, T. Bakke, P. V. Hodson, and T. K. Collier, "Environmental effects of the Deepwater Horizon oil spill: A review," *Marine Pollution Bulletin*, vol. 110, no. 1, pp. 28–51, Sep. 2016, doi: 10.1016/j.marpolbul.2016.06.027.

[2] Y. Zhang *et al.*, "A peak-capture algorithm used on an autonomous underwater vehicle in the 2010 Gulf of Mexico oil spill response scientific survey," *Journal of Field Robotics*, vol. 22, no. 4, pp. 484–496, 2011, doi: 10.1002/rob.20399.

[3] Joint Analysis Group, "Deepwater Horizon Oil Spill. Review of Subsurface Dispersed Oil and Oxygen Levels Associated with the Deepwater Horizon MC252 Spill of National Significance," Silver Spring, MD, USA, NOAA Technical Report NOS OR&R 27, Aug. 2012.

[4] R. N. Conmy *et al.*, "Advances in Underwater Oil Plume Detection Capabilities," presented at the International Oil Spill Conference, 2021, p. 1141330.

[5] D. Gomez-Ibanez, A. L. Kukulya, A. Belani, R. N. Conmy, D. Sundaravadivelu, and L. DiPinto, "Autonomous Water Sampler for Oil Spill Response," *Journal of Marine Science and Engineering*, vol. 10, no. 4, p. 526, 2022, doi: 10.3390/jmse10040526.

[6] C. Ji, C. J. Beegle-Krause, and J. D. Englehardt, "Formation, Detection, and Modeling of Submerged Oil: A Review," *Journal of Marine Science and Engineering*, vol. 8, no. 9, p. 642, 2020, doi: 10.3390/jmse8090642.

[7] J. C. Kinsey, D. R. Yoerger, M. V. Jakuba, R. Camilli, C. R. Fisher, and C. R. German, "Assessing the Deepwater Horizon oil spill with the sentry autonomous underwater vehicle," presented at the IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2011, pp. 261–267. doi: 10.1109/IROS.2011.6095008.

[8] M. V. Jakuba *et al.*, "Toward automatic classification of chemical sensor data from autonomous underwater vehicles," presented at the IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2011, pp. 4722–4727. doi: 10.1109/IROS.2011.6095158.

[9] P. Winsor, H. Simmons, and R. Chant, "Arctic Tracer Release Experiment (ARCTREX): Applications for Mapping Spilled Oil in Arctic Waters," University of Alaska Fairbanks, Fairbanks, AK, Final Report to Bureau of Ocean Energy Management,M13AC00008, OCS Study BOEM 2017- 062.

[10] Y. Wang, W. Thanyamanta, C. Bulger, and N. Bose, "Experimental study to make gas bubbles as proxies for oil droplets to test AUV detection of oil plumes," *Applied Ocean Research*, vol. 121, 2022, doi: 10.1016/j.apor.2022.103080.

[11] L. DiPinto, H. Forth, J. Holmes, A. Kukulya, R. N. Conmy, and O. Garcia, "Three-Dimensional Mapping of Dissolved Hydrocarbons and Oil Droplets Using a REMUS-600 AUV," Bureau of Safety and Environmental Enforcement (BSEE), 2020. [Online]. Available: https://www.bsee.gov/sites/bsee.gov/files/research-reports//1100aa.pdf

[12] "Investigating Scott Inlet seeps with autonomous underwater vehicles," Nunavut Impact Review Board (NIRB). Accessed: Feb. 11, 2024. [Online]. Available: https://www.nirb.ca/portal/pdash.php?appid=125689

[13] J. Hwang, N. Bose, G. Millar, C. Bulger, G. Nazareth, and X. Chen, "Adaptive AUV Mission Control System Tested in the Waters of Baffin Bay," *Drones*, vol. 8, no. 2, p. 45, Feb. 2024, doi: 10.3390/drones8020045.

[14] J. Kallmeyer, Ed., *Life at vents and seeps*, vol. 5. Boston: Walter de Gruyter GmbH & Co, 2017.

[15] R. G. Gillespie and D. A. Clague, *Encyclopedia of islands*. Berkeley: University of California Press, 2009.

[16] H. K. White, R. N. Conmy, I. R. MacDonald, and C. M. Reddy, "Methods of Oil Detection in Response to the Deepwater Horizon Oil Spill," *Oceanography*, vol. 29, no. 3, pp. 76–87, 2016.

[17] S. B. Joye, "The geology and biogeochemistry of hydrocarbon seeps," *Annual Review of Earth and Planetary Sciences*, vol. 48, pp. 205–231, 2020, doi: 10.1146/annurev-earth-063016-020052.

[18] "What is an oil seep?," National Oceanic and Atmospheric Administration. Accessed: Dec. 16, 2022. [Online]. Available: https://oceanservice.noaa.gov/facts/oilseep.html

[19] L. M. Russell-Cargill, B. S. Craddock, R. B. Dinsdale, J. G. Doran, B. N. Hunt, and B. Hollings, "Using autonomous underwater gliders for geochemical exploration surveys," *The APPEA Journal*, vol. 58, no. 1, p. 367, 2018, doi: 10.1071/AJ17079.

[20] J. R. Boles, J. F. Clark, I. Leifer, and L. Washburn, "Temporal variation in natural methane seep rate due to tides, Coal Oil Point area, California," *J. Geophys. Res.*, vol. 106, no. C11, pp. 27077–27086, Nov. 2001, doi: 10.1029/2000JC000774.

[21] R. G. Jenkins, "Cold Seeps," *Encyclopedia of geobiology*. Springer, Dordrecht, 2011.

[22] M. K. Rogener, A. Bracco, K. S. Hunter, M. A. Saxton, and S. B. Joye, "Long-term impact of the Deepwater Horizon oil well blowout on methane oxidation dynamics in the northern Gulf of Mexico," *Elementa: Science of the Anthropocene*, vol. 6, p. 73, Jan. 2018, doi: 10.1525/elementa.332.

[23] A. Bracco *et al.*, "Transport, Fate and Impacts of the Deep Plume of Petroleum Hydrocarbons Formed During the Macondo Blowout," *Front. Mar. Sci.*, vol. 7, p. 542147, Sep. 2020, doi: 10.3389/fmars.2020.542147.

[24] R. Camilli *et al.*, "Tracking Hydrocarbon Plume Transport and Biodegradation at Deepwater Horizon," *Science*, vol. 330, no. 6001, pp. 201–204, Oct. 2010, doi: 10.1126/science.1195223.

[25] S. Mau, D. L. Valentine, J. F. Clark, J. Reed, R. Camilli, and L. Washburn, "Dissolved methane distributions and air-sea flux in the plume of a massive seep field, Coal Oil

Point, California," *Geophysical Research Letters*, vol. 34, no. 22, Nov. 2007, doi: 10.1029/2007GL031344.

[26] "Natural Oil Seeps : Oil in the Ocean," Woods Hole Oceanographic Institution. Accessed: Oct. 30, 2023. [Online]. Available: https://www.whoi.edu/oilinocean/page.do?pid=51880

[27] S. M. Petillo and H. Schmidt, "Autonomous and Adaptive Underwater Plume Detection and Tracking with AUVs: Concepts, Methods, and Available Technology," *IFAC Proceedings Volumes*, vol. 45, no. 27, pp. 232–237, Jan. 2012, doi: 10.3182/20120919-3-IT-2046.00040.

[28] Ocean Studies Board and Marine Board, National Research Council (U.S.), *Oil in the sea III: Inputs, Fates, and Effects*. Washington, D.C: National Academy Press, 2003.

[29] A. Seward, "Hydrocarbon Sensors for Oil Spill Prevention and Response," *Alliance for Coastal Technologies: Solomons, MD, USA*, 2008.

[30] J. Hwang, N. Bose, B. Robinson, and H. Nguyen, "Assessing hydrocarbon presence in the waters of Port au Port Bay, Newfoundland and Labrador, for AUV oil spill delineation tests," *Journal of Ocean Technology*, vol. 15, no. 3, pp. 100–112, 2020.

[31] IPIECA-IOGP, "In-Water Surveillance of Oil Spills at Sea," London, UK, 2016. [Online]. Available: https://www.ipieca.org/resources/in-water-surveillance-of-oil-spills-at-sea

[32] Y. C. Agrawal and H. C. Pottsmith, "Instruments for particle size and settling velocity observations in sediment transport," *Marine Geology*, vol. 168, no. 1–4, pp. 89–114, Aug. 2000, doi: 10.1016/S0025-3227(00)00044-X.

[33] J. Hwang, N. Bose, H. D. Nguyen, and G. Williams, "Acoustic search and detection of oil plumes using an autonomous underwater vehicle," *Journal of Marine Science and Engineering*, vol. 8, no. 8, p. 618, 2020, doi: 10.3390/JMSE8080618.

[34] A. Balsley, K. Hansen, and M. Fitzpatrick, "Detection of oil within the water column," presented at the International Oil Spill Conference Proceedings, American Petroleum Institute, 2014, pp. 2206–2217.

[35] R. Kubilius, "Multi-frequency acoustic discrimination between gas bubble plumes and biological targets in the ocean," Ph.D dissertation, University of Bergen, Bergen, Norway, 2015.

[36] M. V. Jakuba *et al.*, "Exploration of the gulf of mexico oil spill with the sentry autonomous underwater vehicle," presented at the Proceedings of the International Conference on Intelligent Robots and Systems (IROS) Workshop on Robotics for Environmental Monitoring (WREM), San Francisco, CA, USA, 2011, pp. 25–30.

[37] A. L. Kukulya *et al.*, "Autonomous Chemical Plume Detection and Mapping Demonstration Results with a COTS AUV and Sensor Package," in *OCEANS 2018 MTS/IEEE Charleston*, Oct. 2018, pp. 1–6. doi: 10.1109/OCEANS.2018.8604524.

[38] J. Hwang, N. Bose, H. D. Nguyen, and G. Williams, "Oil Plume Mapping: Adaptive Tracking and Adaptive Sampling From an Autonomous Underwater Vehicle," *IEEE access*, vol. 8, pp. 198021–198034, 2020, doi: 10.1109/ACCESS.2020.3032161.

[39] A. Jayasiri, R. G. Gosine, G. K. I. Mann, and P. McGuire, "AUV-Based Plume Tracking: A Simulation Study," *Journal of Control Science and Engineering*, vol. 2016, pp. 1–15, 2016, doi: 10.1155/2016/1764527.

[40] D. J. Wilson, "AGSO Marine Survey 176 Direct Hydrocarbon Detection North-West Australia: Yampi Shelf, Southern Vulcan Sub-Basin and Sahul Platform (July/September

1996) - Operational Report & Data Compendium," Australian Geological Survey Organisation, Canberra, Australia, AGSO Record 2000/42, 2000.

[41] J. Hwang, N. Bose, G. Millar, C. Bulger, and G. Nazareth, "Bubble Plume Tracking Using a Backseat Driver on an Autonomous Underwater Vehicle," *Drones*, vol. 7, no. 10, p. 635, Oct. 2023, doi: 10.3390/drones7100635.

[42] J. Zhao, D. Mai, H. Zhang, and S. Wang, "Automatic Detection and Segmentation on Gas Plumes from Multibeam Water Column Images," *Remote Sensing*, vol. 12, no. 18, p. 3085, Sep. 2020, doi: 10.3390/rs12183085.

[43] H. Zhang *et al.*, "Subsea pipeline leak inspection by autonomous underwater vehicle," *Applied Ocean Research*, vol. 107, p. 102321, Feb. 2021, doi: 10.1016/j.apor.2020.102321.

[44] J. Zhao, J. Meng, H. Zhang, and S. Wang, "Comprehensive Detection of Gas Plumes from Multibeam Water Column Images with Minimisation of Noise Interferences," *Sensors*, vol. 17, no. 12, p. 2755, Nov. 2017, doi: 10.3390/s17122755.

[45] A. W. Nau, B. Scoulding, R. J. Kloser, Y. Ladroit, and V. Lucieer, "Extended Detection of Shallow Water Gas Seeps From Multibeam Echosounder Water Column Data," *Frontiers in Remote Sensing*, vol. 3, p. 839417, Jul. 2022, doi: 10.3389/frsen.2022.839417.

[46] S. Wang, B. Gong, Y. Liu, and W. Li, "Automatic gas leak detection system," *IOP Conference Series: Earth and Environmental Science*, vol. 514, no. 2, p. 022020, May 2020, doi: 10.1088/1755-1315/514/2/022020.

[47] W. Zhang, T. Zhou, W. Du, S. Xu, M. Liu, and Y. Wang, "A method for undersea gas bubbles detection from acoustic image," presented at the 178th Meeting of the Acoustical Society of America, San Diego, California, 2019, p. 070003. doi: 10.1121/2.0001263.

[48] A. E. A. Blomberg, T. O. Saebo, R. E. Hansen, R. B. Pedersen, and A. Austeng, "Automatic Detection of Marine Gas Seeps Using an Interferometric Sidescan Sonar," *IEEE Journal of Oceanic Engineering*, vol. 42, no. 3, pp. 590–602, Jul. 2017, doi: 10.1109/JOE.2016.2592559.

[49] J. Schneider von Deimling and C. Papenberg, "Technical Note: Detection of gas bubble leakage via correlation of water column multibeam images," *Ocean Sci.*, vol. 8, no. 2, pp. 175–181, Mar. 2012, doi: 10.5194/os-8-175-2012.

[50] S. A. Socolofsky, E. E. Adams, and C. R. Sherwood, "Formation dynamics of subsurface hydrocarbon intrusions following the Deepwater Horizon blowout," *Geophysical Research Letters*, vol. 38, no. 9, p. 2011GL047174, May 2011, doi: 10.1029/2011GL047174.

[51] A. Speck *et al.*, "Supervised Autonomy for Advanced Perception and Hydrocarbon Leak Detection," in *Global Oceans 2020: Singapore – U.S. Gulf Coast*, Biloxi, MS, USA: IEEE, Oct. 2020, pp. 1–6. doi: 10.1109/IEEECONF38699.2020.9389434.

[52] Y. Chen, H. Liang, and S. Pang, "Study on Small Samples Active Sonar Target Recognition Based on Deep Learning," *Journal of Marine Science and Engineering*, vol. 10, no. 8, p. 18, 2022, doi: 10.3390/jmse10081144.

[53] P. Tueller, R. Kastner, and R. Diamant, "Target detection using features for sonar images," *IET Radar, Sonar & Navigation*, vol. 14, no. 12, pp. 1940–1949, Dec. 2020, doi: 10.1049/iet-rsn.2020.0224.

[54] H. Johannsson, M. Kaess, B. Englot, F. Hover, and J. Leonard, "Imaging sonar-aided navigation for autonomous underwater harbor surveillance," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei: IEEE, Oct. 2010, pp. 4396–4403. doi: 10.1109/IROS.2010.5650831.

[55] A. Nikolovska, H. Sahling, and G. Bohrmann, "Hydroacoustic methodology for detection, localization, and quantification of gas bubbles rising from the seafloor at gas seeps from the eastern Black Sea," *Geochemistry, Geophysics, Geosystems*, vol. 9, no. 10, Oct. 2008, doi: 10.1029/2008GC002118.

[56] K. A. Blasco, S. M. Blasco, R. Bennett, B. MacLean, W. A. Rainey, and E. H. Davies, "Seabed geologic features and processes and their relationship with fluid seeps and the benthic environment in the Northwest Passage," Geological Survey of Canada, Open File 6438, 2010. doi: 10.4095/287316.

[57] R. K. Falconer and B. D. Loncarevic, "An Oil Slick Occurrence Off Baffin Island," Geological Survey of Canada, Darthmouth, Canada, Paper 77–1A, 1977. doi: 10.4095/102743.

[58] M. A. Cramm *et al.*, "Characterization of marine microbial communities around an Arctic seabed hydrocarbon seep at Scott Inlet, Baffin Bay," *Science of The Total Environment*, vol. 762, pp. 143961–143961, 2021, doi: 10.1016/j.scitotenv.2020.143961.

[59] "Clyde Inlet to Cape Jameson," Canadian Hydrographic Service, Chart 7565, Oct. 04, 1996.

[60] G. Nazareth, "pPlumeDetector." Feb. 08, 2024. [Online]. Available: https://github.com/GinelleNazareth/pPlumeDetector

[61] K. Klemens, "Understanding and Using Scanning Sonars," BlueRobotics. Accessed: Feb. 18, 2024. [Online]. Available: https://bluerobotics.com/learn/understanding-and-using-scanning-sonars/

[62] J. Hwang, "Adaptive Sampling of a Discrete Underwater Plume Using an Autonomous Underwater Vehicle," Ph.D. dissertation, University of Tasmania, Tasmania, Australia, 2021.

[63] "Access Ping360 data for post processing, Python," Blue Robotics Community Forums. Accessed: Sep. 07, 2022. [Online]. Available: https://discuss.bluerobotics.com/t/access-ping360-data-for-post-processing-python/10416

[64] "Ping360 Scanning Imaging Sonar," Blueye Robotics. Accessed: Jan. 31, 2024. [Online]. Available: https://www.blueyerobotics.com/products/ping360-scanning-sonar

[65] "Ping-Viewer." Blue Robotics, Jul. 08, 2022. Accessed: Sep. 06, 2022. [Online]. Available: https://github.com/bluerobotics/ping-viewer

[66] "Geometric Image Transformations," OpenCV. Accessed: Apr. 14, 2023. [Online]. Available: https://docs.opencv.org/3.4/da/d54/group___imgproc___transform.html#ga49481ab24fdaa0ffa4d3e63d14c0d5e4

[67] D. Han, "Comparison of Commonly Used Image Interpolation Methods," presented at the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013), Paris, France: Atlantis Press, 2013, pp. 1556–1559.

[68] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[69] "Clustering," scikit-learn. Accessed: Jan. 23, 2024. [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html

[70] M. Ester, H.-P. Kriegel, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," *kdd*, vol. 96, no. 34, pp. 226–231, 1996.

[71] P. Cichosz, *Data mining algorithms: explained using R*. Chichester, England: Wiley, 2015.

[72] T. Boonchoo, X. Ao, Y. Liu, W. Zhao, F. Zhuang, and Q. He, "Grid-based DBSCAN: Indexing and inference," *Pattern Recognition*, vol. 90, pp. 271–284, Jun. 2019, doi: 10.1016/j.patcog.2019.01.034.

[73] A. Gunawan, "A Faster Algorithm for DBSCAN," Master's Thesis, Eindhoven University of Technology, Eindhoven, Netherlands, 2013.

[74] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, pp. 1–21, Sep. 2017, doi: 10.1145/3068335.

[75] R. Szeliski, *Computer Vision: Algorithms and Applications*. London: Springer London, 2011. doi: 10.1007/978-1-84882-935-0.

[76] "skimage.measure," scikit-image. Accessed: Sep. 16, 2022. [Online]. Available: https://scikit-image.org/docs/stable/api/skimage.measure.html#skimage.measure.label

[77] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling," *IEEE Trans. on Image Process.*, vol. 29, pp. 1999–2012, 2020, doi: 10.1109/TIP.2019.2946979.

[78] M. L. Seto, Ed., *Marine robot autonomy*. New York, NY: Springer New York, 2013.

[79] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard, "Nested autonomy for unmanned marine vehicles with MOOS-IvP," *Journal of Field Robotics*, vol. 27, no. 6, pp. 834–875, Nov. 2010, doi: 10.1002/rob.20370.

[80] G. Nazareth, "iPing360Device." Jul. 21, 2023. [Online]. Available: https://github.com/GinelleNazareth/iPing360Device

[81] "A Smooth Operator's Guide to Underwater Sonars and Acoustic Devices," BlueRobotics. Accessed: Jan. 12, 2024. [Online]. Available: https://bluerobotics.com/learn/a-smooth-operators-guide-to-underwater-sonars-and-acoustic-devices/

[82] N. Ehrenholz, "MUN Explorer AUV: MOOS Backseat Driver Manual." International Submarine Enginering Ltd., Aug. 21, 2019.

[83] "UviLux Handbook." Chelsea Technologies Ltd, Jul. 07, 2016.