



الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
المدرسة الوطنية العليا للتكنولوجيات المتقدمة
National Higher School of Advanced Technologies



Electrical Engineering and Industrial Informatics Department.

Final Year Project to Obtain the Diploma of Engineering

Automation
Automation and Industrial Informatics

Contribution to the development of a control architecture for the autonomous navigation of a surface drone

Realized by
ADDA Mohamed Said
Presented publicly, 11 / 09 / 2025

First LAST NAME	University	Quality
REBAI Karima	ENSTA	President
KHELOUAT Samir	ENSTA	Examinator
OUSAAZI Zahira	ENSTA	Examinator
DJEKOUNE Oualid	CDTA	Supervisor
ZAOUCHE Mohamed	ENSTA	Supervisor

2024/2025

Abstract

This thesis presents the design, implementation, and validation of an autonomous control architecture for an Unmanned Surface Vehicle (USV) developed for dam inspection and environmental monitoring applications. The proposed system, named **AquaRob**, integrates a cost-effective hardware platform with a modular ROS2-based software framework to enable robust navigation in GPS-degraded and hydrodynamically disturbed environments. The hardware architecture combines low-cost sensors, such as the GY-87 IMU and the NEO-M8N GNSS, with a dual-processor setup consisting of an Arduino Due and a Raspberry Pi 5. Sensor data are fused using a dual Extended Kalman Filter (EKF) for precise state estimation. Two complementary control strategies are implemented and evaluated: a Sliding Mode Controller (SMC) for robust path tracking under model uncertainties, and an adaptive PID controller that employs online system identification based on the Recursive Least Squares (RLS) algorithm. These control approaches are validated in a Gazebo-based simulation environment, demonstrating reliable trajectory tracking, stability, and robustness to disturbances. The results confirm that the integration of model-based and adaptive data-driven control enhances the system's autonomy and adaptability in complex aquatic environments. The developed framework provides a scalable foundation for future field deployments and extensions toward 3D navigation and multi-agent cooperation in autonomous maritime systems.

Keywords: USV, ROS2, EKF, Sensor Fusion, Sliding Mode Control (SMC), Recursive Least Squares (RLS), Adaptive PID

Résumé

Ce mémoire présente la conception, la mise en œuvre et la validation d'une architecture de commande autonome pour un véhicule de surface sans pilote (USV) développé pour les applications d'inspection des barrages et de surveillance environnementale. Le système proposé, nommé **AquaRob**, intègre une plateforme matérielle à faible coût et une architecture logicielle modulaire basée sur ROS2, permettant une navigation robuste dans des environnements caractérisés par des perturbations hydrodynamiques et une dégradation du signal GPS.

L'architecture matérielle combine des capteurs économiques tels que l'unité inertielle GY-87 et le récepteur NEO-M8N GNSS avec une configuration à double processeur basée sur Arduino Due et Raspberry Pi 5. Les données des capteurs sont fusionnées à l'aide d'un filtre de Kalman étendu (EKF) pour une estimation précise de l'état du véhicule.

Deux stratégies de commande complémentaires ont été développées et évaluées : un contrôleur par mode glissant (SMC) garantissant une robustesse face aux incertitudes du modèle, et un régulateur PID adaptatif utilisant l'identification en ligne du système via l'algorithme des moindres carrés récursifs (RLS). Les simulations réalisées sous Gazebo ont démontré la stabilité, la précision et la robustesse du système de navigation autonome. Les résultats obtenus confirment que la combinaison de commandes basées sur le modèle et d'approches adaptatives améliore l'autonomie et la flexibilité du système dans des environnements aquatiques complexes. Ce travail constitue une base solide pour de futurs déploiements sur le terrain et des extensions vers la navigation tridimensionnelle et la coopération multi-robots.

Mots-clés : USV, ROS2, EKF, Fusion de capteurs, Commande par mode glissant (SMC), Moindres carrés récursifs (RLS), PID adaptatif

المخلص

يقدم هذا العمل تصميم وتنفيذ والتحقق من بنية تحكم ذاتي لمركبة سطحية غير مأهولة مخصصة لمهام فحص السدود والمراقبة البيئية. يعتمد النظام المقترح، المسمى **AquaRob**، على منصة مادية منخفضة التكلفة وهيكل برجي معياري قائم على ROS2، مما يسمح بالملاحة الذاتية في بيئات مائية معقدة تعاني من تدهور إشارات GPS واضطرابات هيدروديناميكية.

تتكون البنية المادية من حساسات منخفضة التكلفة مثل وحدة GY-87 ونظام تحديد المواقع NEO-M8N إضافة إلى معالجي Arduino Due و Raspberry Pi 5. يتم دمج بيانات الحساسات باستخدام مرشح كالمان الموسع (EKF) للحصول على تقدير دقيق لحالة المركبة.

تم تنفيذ وتقييم استراتيجيتين للتحكم: التحكم بطريقة الانزلاق (SMC) لضمان المتانة في وجود عدم يقين في النموذج، ومنظم PID تكيفي يعتمد على خوارزمية المربعات الصغرى التكرارية (RLS) للتعرف الآني على النظام. أظهرت نتائج المحاكاة باستخدام Gazebo دقة تتبع عالية واستقراراً ومثانة أمام الاضطرابات، مما يؤكد أن دمج التحكم القائم على النماذج مع الأساليب التكيفية يعزز استقلالية النظام وقدرته على التكيف في البيئات المائية المعقدة.

الكلمات المفتاحية: USV، ROS2، EKF، دمج الحساسات، التحكم بالانزلاق، (SMC) المربعات الصغرى التكرارية، (RLS) PID التكيفي

Contents

List of Abbreviations	vi
Introduction	1
Context and Motivation	1
Problem Statement	1
Research Objectives	2
Structure of the thesis	2
1 System Design and Architecture	3
1.1 Introduction	3
1.2 Hardware Architecture	3
1.3 Software Architecture	8
1.4 Operational Modes Implementation	10
1.5 Sensor Fusion Logic	12
1.6 Dynamics Modeling	15
1.7 Control System Design	19
1.8 Conclusion	24
2 Implementation Details	25
2.1 Introduction	25
2.2 Development Workflow	25
2.3 Hardware Integration	26
2.4 Launch and Configuration	34
2.5 WAM-V Simulation Framework	37
2.6 Software Configuration and Validation Tools	43
2.7 Transformation Frame Architecture in Dual EKF Localization	53
2.8 Conclusion	63
3 Control strategy	64
3.1 Introduction	64
3.2 Sliding Mode Control (SMC) Implementation	64
3.3 Kinematic Modeling of Surface Motion: Heading and Translational Dynamics	70
3.4 LOS guidance with PD/PI inner-loop controller	81
3.5 Conclusion	85
Conclusion and Future Work	86
Summary of Contributions	86
Future Extensions	86

List of Figures

Figure 1.1.	The AquaRob Drone	4
Figure 1.2.	The "GY-87" wiring	5
Figure 1.3.	The GPS M8N wiring	6
Figure 1.4.	Electrical Control Unit	7
Figure 1.5.	Propulsion system	7
Figure 1.6.	ROS Gazebo Integration	9
Figure 1.7.	ROS 2 general Publish/Subscribe Diagram architecture	10
Figure 1.8.	Implementation Flowchart for USV Operational Modes	12
Figure 1.9.	Data Flow Diagram for sensors fusion from the IMU "GY87" and GPS "M8N"	13
Figure 1.10.	Motion of the USV in 6 DOF [1].	15
Figure 1.11.	sign(S) function	21
Figure 1.12.	Cont function	22
Figure 1.13.	Saturation function	22
Figure 1.14.	Hyperbolic tangent function	23
Figure 2.1.	Workflow Summary Diagram	26
Figure 2.2.	Physical wiring of the GY-87 IMU and NEO-M8N GPS on the Ar- duino DUE.	27
Figure 2.3.	Arduino Due-Based IMU and GPS Integration Prototype on Breadboard	27
Figure 2.4.	Setting the Micro ROS agent connection link between Arduino and ROS2	32
Figure 2.5.	Reading the GPS data on Arduino before transmitting them to ROS2 .	32
Figure 2.6.	Reading the IMU data in Terminal, ROS2	32
Figure 2.7.	ROS Node Diagram for the EKF implementation on the Arduino data .	33
Figure 2.8.	The real world environment used for the simulation	38
Figure 2.9.	The gazebo simulation for the USV	39
Figure 2.10.	Entities used in the Gazebo simulation environment	39
Figure 2.11.	Entities attached to the Wam-V USV in Gazebo	40
Figure 2.12.	Topics and nodes necessar for the EKF node in rqt_graph	43
Figure 2.13.	Visualization of the raw waypoint path (straight lines) and the smoothed reference path (B-spline) in RViz2.	48
Figure 2.14.	Visualization of the lawnmower path in RViz2: the original waypoint path (straight lines) and the smoothed Catmull-Rom spline reference path.	49
Figure 2.15.	configuration and Parameters used in RVIZ2	51
Figure 2.16.	Path and Pose parameters used in RVIZ2	51
Figure 2.17.	The representation of the USV on RVIZ2	52
Figure 2.18.	The visualisation of the USV on rviz2 while following the a straight line with an open loop command	52

Figure 2.19.	Data Flow for EKF-based Local (odom) and Global (map) odom Estimation	54
Figure 2.20.	Transformation frame hierarchy generated by <code>ros2 run tf2_tools view_frames</code> command, showing the complete tree structure of the EKF localization system	56
Figure 2.21.	The topics subscribed by Plotjuggler for the EKF graphs visualisation	58
Figure 2.22.	Observing the yaw orientation while the Robot is on its start position .	58
Figure 2.23.	Observing the X, Y while the Robot is on its start position	59
Figure 2.24.	Observing the V_x and V_y calculated by the EKF while the Robot is on its start position	59
Figure 2.25.	Launching the robot to position B (20, 20) on Gazebo	60
Figure 2.26.	Observing the X, Y while the Robot is moving from position A(0,0) to position B(20,20)	60
Figure 2.27.	Observing the V_x and V_y while the Robot is moving from position A(0,0) to position B(20,20)	61
Figure 2.28.	Rotation around the vertical Axis of the vessel in Gazebo	62
Figure 2.29.	Spin Cycle graph for visualizing the Yaw orientation from π to $-\pi$.	62
Figure 3.1.	Control errors: lateral error and yaw angle error for path tracking . . .	67
Figure 3.2.	The SMC node in ROS2	69
Figure 3.3.	The robot following the path through the SMC controller	70
Figure 3.4.	Inputs of the System	71
Figure 3.5.	Output of the System	71
Figure 3.6.	Estimated parameters model a_i	73
Figure 3.7.	Estimated parameters model b_i	74
Figure 3.8.	Measured heading angle $y(t)$ and estimated angle $\hat{y}(t)$	74
Figure 3.9.	Prediction error $e(t) = y(t) - \hat{y}(t)$	75
Figure 3.10.	Single-Input Single-Output (SISO) Linear System	75
Figure 3.11.	Evolution of the PID gains right	78
Figure 3.12.	Evolution of the PID gains left	78
Figure 3.13.	Compare the control input right signals	79
Figure 3.14.	Compare the control input left signals	79
Figure 3.15.	Comparison between the real output, measured, and the estimated output from the ARX model driven by the adaptive PID controller.	80
Figure 3.16.	Tracking error between the measured output and the model-estimated output.	80
Figure 3.17.	The USV following the target in RVIZ2, USV in green, Path target in blue/purple	84

List of Tables

Table 2.1. IMU Calibration Results	28
Table 2.2. Transformation Frame Properties and Functions	53
Table 3.1. Key SMC parameters used and recommended starting values.	68
Table 3.2. Typical parameters and baseline tuning ranges used during experiments. .	83

List of Abbreviations

USV	Unmanned Surface Vehicle
ROS2	Robot Operating System 2
GPS	Global Positioning System
GNSS	Global Navigation Satellite System
IMU	Inertial Measurement Unit
MPU6050	Microprocessor Unit 6050 (Accelerometer and Gyroscope Sensor)
HMC5883L	Triple Axis Magnetometer Sensor
I2C	Inter-Integrated Circuit
UART	Universal Asynchronous Receiver-Transmitter
EKF	Extended Kalman Filter
SDF Model	Simulation Description Format Model
CSV	Comma-Separated Values
RVIZ2	ROS Visualization Tool for ROS2
WAM-V	Wave Adaptive Modular Vehicle
TF Frame	Transform Frame (Coordinate Transform in ROS)
SMC	Sliding Mode Control
MSE	Mean Squared Error
RLS	Recursive Least Squares
ARX	Auto-Regressive with exogenous inputs
LOS	Line Of Sight
PID	proportionalIntegral-Derivative controller

Introduction

Context and Motivation

Unmanned Surface Vehicles (USVs) have emerged as transformative platforms for environmental monitoring and infrastructure inspection, particularly in challenging aquatic environments such as dam reservoirs. These critical infrastructures require regular assessment to detect structural deterioration, sediment accumulation, and ecological changes that could compromise their operational integrity. Traditional inspection methodologies, relying on manual surveys or manned vessels, present significant limitations including high operational costs, time-intensive processes, and substantial safety risks in turbulent waters. Autonomous USVs offer a promising alternative by enabling safer, more cost-effective, and continuous monitoring capabilities. However, dam environments introduce distinctive challenges that complicate autonomous navigation:

- **Hydrodynamic Disturbances:** Complex turbulent flows generated by dam operations and unpredictable wind patterns affecting vehicle stability
- **GPS Degradation:** Signal multipath errors caused by steep canyon walls and intermittent satellite visibility leading to positioning inaccuracies
- **Sensor Limitations:** Performance constraints of low-cost inertial measurement units (IMUs) exhibiting significant drift characteristics that challenge accurate state estimation

Problem Statement

Existing commercial USV platforms (e.g., Teledyne Marine’s C-Worker[®], SeaRobotics’ SR-UTM) predominantly target open-water applications including oceanic surveys and offshore infrastructure inspections [2, 3]. These systems typically employ high-precision navigation systems incorporating Real-Time Kinematic GPS (RTK-GPS) and fiber-optic gyroscopes, resulting in acquisition costs exceeding \$50,000 [4]. This economic barrier renders them impractical for small-to-medium scale dam inspection operations. Furthermore, their proprietary navigation algorithms demonstrate limited adaptability to the constrained, GPS-challenged environments characteristic of dam reservoirs with dynamic current patterns and significant multipath interference [5].

Academic research prototypes, while addressing cost considerations, frequently focus on isolated technical modules such as sensor fusion methodologies [6] or control system architectures [7] without comprehensive system integration. Recent contributions by Zhang et al. (2022) and Lee et al. (2023) have demonstrated advanced implementations of Sliding Mode Control (SMC) and Extended Kalman Filter (EKF) techniques [8, 9], however, validation has been primarily limited to simulated environments or controlled laboratory conditions. This

fragmentation in research approaches has created a significant gap in field-deployable, fully integrated systems capable of robust autonomous navigation in real-world dam environments.

This work therefore targets the development of a practical, low-cost navigation and control architecture for an underactuated USV that performs reliable trajectory following and area coverage in dam-like environments with degraded GNSS and environmental disturbances.

Research Objectives

This thesis aims to design, implement, and validate an autonomous USV platform specifically engineered for dam inspection applications through the following objectives:

1. Design a cost-effective hardware and software architecture based on single-board computers and microcontrollers for robust sensor interfacing and actuator control.
2. Implement and evaluate two distinct navigation modalities: random waypoint generation for area coverage and predefined trajectory tracking for structured inspection patterns
3. Design and implement an Extended Kalman Filter framework that integrates GPS, IMU and other sensors for enhanced state estimation
4. Design, implement and validate robust path-tracking controllers (Sliding Mode Control and a LOS guidance with PD/PI inner loop) tailored for an underactuated differential-thrust USV and demonstrate practical tuning procedures.
5. Integrate the above components into a Gazebo-based simulation and validate with quantitative experiments and logging metrics (tracking error, control effort, stability).

Structure of the thesis

Chapters 2 through 5 are organized to guide the reader from system foundations to final insights. Chapter 2 presents the system architecture along with the hardware and software design. Chapter 3 then details the implementation, simulation setup, and validation tools used in the study. Chapter 4 outlines the proposed control strategy, explaining its structure and operation. Finally, Chapter 5 summarizes the results, discusses the limitations of the work, and offers suggestions for future research directions.

Chapter 1

System Design and Architecture

1.1 Introduction

The design and architecture of an autonomous Unmanned Surface Vehicle (USV) such as AquaRob require a careful integration of hardware, software, and control strategies to achieve reliable performance in dynamic and often unpredictable environments. This chapter presents a comprehensive overview of the system design, covering the hardware components that form the physical platform, the software architecture that enables perception and decision-making, and the control systems that ensure robust navigation and trajectory tracking. The hardware section describes the structural elements of the vessel, including the hull, actuators, processing units, and sensor suite, with particular attention to redundancy and environmental resilience. The software section outlines the modular ROS2-based framework, detailing the communication protocols, sensor fusion algorithms, and operational modes that enable autonomous navigation. Finally, mathematical modeling of the vehicle dynamics and the design of advanced control schemes are presented, forming the basis for reliable path-following and disturbance rejection. Together, these elements establish a cohesive design and architecture that balance robustness, scalability, and adaptability for AquaRob's intended missions.

1.2 Hardware Architecture

1.2.1 Structural Components

The backbone of AquaRob is its inflatable boat hull, which measures 3 meters in length and 2 meters in width. This hull is designed to be robust against natural conditions such as water, temperature, and humidity. The storage system is a rigid hitch designed to minimize vibration transmission and ensure symmetrical weight distribution around the translation axis, which is crucial for maintaining stability and balance during operations.



Figure 1.1: The AquaRob Drone

1.2.2 Processing Units

The USV employs a dual-processor architecture. The onboard MiniPC (e.g., Intel NUC) is the companion computer running ROS2 and handles navigation and planning. An Arduino Due microcontroller handles low-level hardware interfacing and control. The Dues ARM Cortex-M3 core runs at 84 MHz and supports multiple I/O (54 digital pins, 12 analog inputs, 4 UART ports, 2 I2C buses) [10]. This makes it well-suited for multi-sensor integration. Sotelo-Torres *et al.* report a similar split, pairing a Raspberry Pi 4 with a Pixhawk flight controller in their USV design [11].

1.2.3 Sensor Modules

The USVs sensors include an inertial measurement unit (IMU) and a GNSS receiver. The IMU is implemented with a GY-87 board (MPU-6050 gyro/accelerometer, QMC5883L magnetometer, BMP180 barometer) that communicates over I2C [12]. This provides 3-axis acceleration, rotation rate, magnetic heading, and atmospheric pressure at a high update rate. Positioning is provided by a u-blox NEO-M8N GNSS module, which concurrently receives GPS, GLONASS, Galileo, and BeiDou signals [13]. The NEO-M8N outputs NMEA/UBX data via serial UART to yield position and velocity fixes (typically at 510 Hz).

Inertial Measurement Unit (IMU GY-87)

The GY-87 is a compact 10-degree-of-freedom module combining an MPU-6050 3-axis accelerometer and gyroscope, an HMC5883L 3-axis magnetometer, and a BMP180 barometric sensor, all interfaced via I2C [14, 15]. The MPU-6050 provides acceleration measurements up to $\pm 16g$ and rotation rates up to $\pm 2000^\circ/s$, with typical noise densities of $400 \mu g/\sqrt{Hz}$ (accelerometer) and $0.005^\circ/s/\sqrt{Hz}$ (gyroscope) [16]. The HMC5883L magnetometer offers heading accuracy around 1° under normal magnetic conditions, while the BMP180 yields pressure measurements with ± 1 hPa accuracy for altitude estimation [17]. This sensor fusion package is ideal for low-cost attitude and heading reference in resource-constrained platforms like Arduino and Raspberry Pi.

To enhance the reliability and robustness of the inertial measurement system, three identical GY-87 IMU modules are employed in a redundant configuration. This setup allows for the calculation of the average value from the three sensors, which helps in reducing noise and improving the accuracy of the measurements. Additionally, a fault detection mechanism is

implemented to identify and eliminate any outlier readings that may result from sensor malfunctions or excessive noise. This redundancy ensures that the system remains operational even if one of the IMUs fails, thereby increasing the overall reliability of the navigation system.

The electrical wiring and interfacing of the GY-87 module with the Arduino controller are illustrated in Figure 1.2.

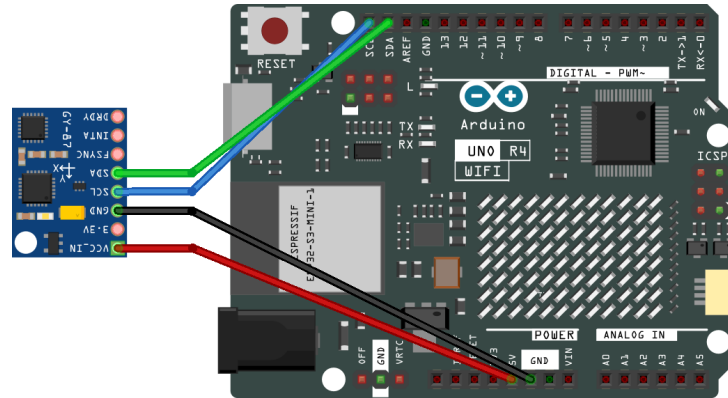


Figure 1.2: The "GY-87" wiring

IMU Mounting and Isolation

To ensure accurate and stable inertial measurements, each IMU is mounted on a custom-designed support structure. This structure incorporates several key features to isolate the sensors from various disturbances:

- **Vibration Isolation:** The mounting system uses vibration-damping materials such as rubber or silicone to minimize the transmission of high-frequency vibrations from the vehicle's hull to the IMUs. The design of the mounting structure aims to have a natural frequency well below the typical vibration frequencies encountered during operation, thereby isolating the IMUs from most vibrational disturbances.
- **Shock Absorption:** The support structure includes shock-absorbing materials like foam or gel-based composites to protect the IMUs from sudden impacts or shocks. The compliant design of the mounting platform helps dissipate shock energy, preventing it from affecting the sensor readings.
- **Thermal Isolation:** To maintain a stable operating temperature, the mounting structure incorporates thermal insulation layers. These layers help mitigate temperature fluctuations that could otherwise affect the accuracy and performance of the IMUs.
- **Environmental Protection:** The IMUs are housed within sealed enclosures that protect them from moisture, dust, and other environmental contaminants. The use of waterproof and dustproof materials ensures the longevity and reliability of the sensors in harsh operating conditions.
- **Structural Stability:** The base platform of the mounting structure is designed to be rigid and stable, providing a consistent reference frame for the IMUs. Precise alignment mechanisms are included to ensure accurate and repeatable positioning of the sensors relative to the vehicle's coordinate system.

- **Electromagnetic Isolation:** The mounting structure is equipped with electromagnetic shielding to protect the IMUs from electromagnetic interference (EMI) that could potentially disrupt their readings. This shielding is achieved through the use of conductive materials and specialized composites that block or attenuate electromagnetic signals.

By integrating these features into the IMU mounting and isolation system, we ensure that the sensors are protected from a wide range of disturbances, thereby enhancing the accuracy and reliability of the inertial measurements. This comprehensive approach to mounting and isolation is crucial for achieving high-performance state estimation and control in the autonomous USV platform.

Global Navigation Satellite System Receiver (GPS NEO-M8N)

The u-blox NEO-M8N is a multi-constellation GNSS receiver supporting GPS, GLONASS, Galileo, and BeiDou, with simultaneous tracking of up to three systems [18]. It can output navigation solutions at up to 10 Hz update rate, configurable via the UBX-CFG-RATE interface, with typical time-to-first-fix of less than 30 s (cold start) and under 1 s (hot start) [19, 20]. Horizontal position accuracy is on the order of 2.02.5 m with SBAS enabled, and velocity accuracy is approximately 0.05 m/s [21]. Advanced features include 167 dBm tracking sensitivity, anti-jamming/spoofing detection, and support for all satellite augmentation systems.

The hardware integration of the positioning system is detailed in Figure 1.3, which illustrates the wiring interface connecting the u-blox NEO-M8N GNSS receiver to the Arduino microcontroller.

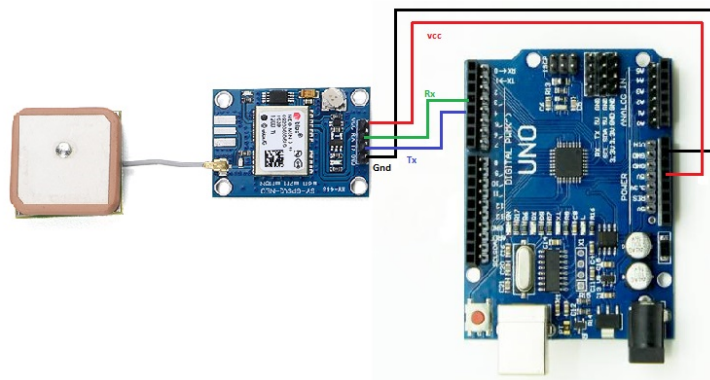


Figure 1.3: The GPS M8N wiring

Wind Sensor for Disturbance Compensation

Wind disturbances can significantly affect the trajectory and stability of Unmanned Surface Vehicles (USVs), especially in confined environments like dams. To mitigate these effects, integrating a wind sensor allows for real-time measurement of wind speed and direction, enabling the implementation of feedforward control strategies and disturbance observers.

Incorporating wind data into the control algorithms, such as Extended Kalman Filters (EKF) or Unscented Kalman Filters (UKF), improves the accuracy of state estimation by accounting for external disturbances. This integration is crucial for maintaining precise navigation paths and ensuring the reliability of the USV's operations in varying environmental conditions.

1.2.4 Actuator System

1.2.5 Actuator System

Propulsion is provided by two rear-mounted thrusters (bidirectional DC motors), arranged for differential steering as shown in Figure 1.4. Each thruster is driven by an electronic speed controller (ESC) that receives discrete speed commands (high, medium, or low) from the Arduino Due. These components are housed within the Electrical Control Unit depicted in Figure 1.5, which protects the electronics while allowing power and control cables to run through sealed bulkhead fittings to the external motors, ensuring waterproof integrity. By varying the thrust of the left and right motors, the vehicle controls its heading and speed. This dual-thruster configuration is common in small USVs and provides redundancy.

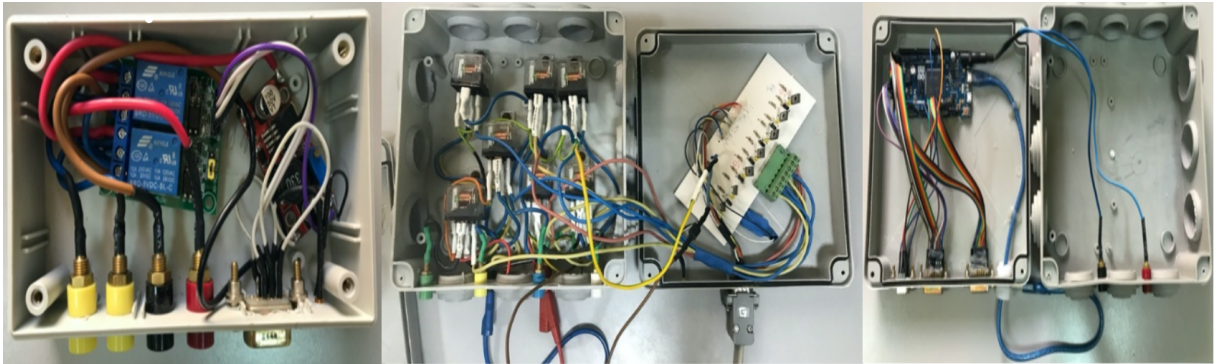


Figure 1.4: Electrical Control Unit

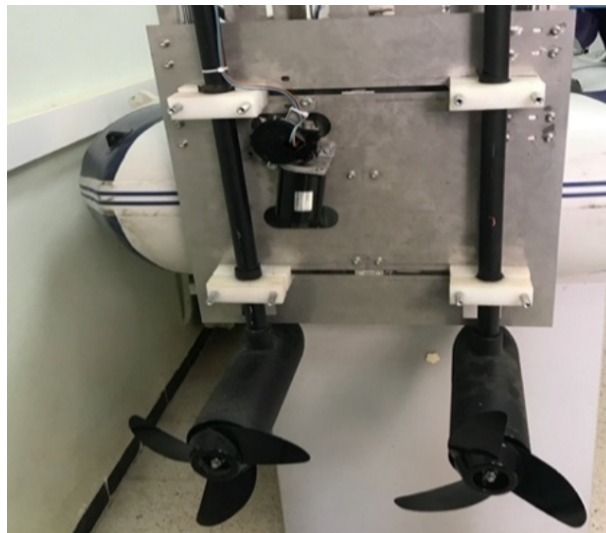


Figure 1.5: Propulsion system

1.2.6 Communication and Wiring

Internally, devices connect to the Arduino Due via standard bus interfaces: the IMU uses I2C, the GPS and telemetry radios use UART, and thruster controllers use PWM signals [12, 10]. The Due can also interface via CAN if needed. All wiring uses marine-grade, shielded cables and waterproof connectors to withstand spray and humidity. Wireless telemetry (e.g., LoRa or

WiFi) can be added for remote monitoring, but core sensor and actuator communications are wired for robustness.

1.3 Software Architecture

The USV software is modular and based on ROS2 (Jazzy). Each function is implemented as a separate ROS2 node running on the MiniPC, following a distributed publishersubscriber model [22]. For example, one node (`imu_driver`) reads the IMU and publishes `sensor_msgs/Imu`, while another (`gps_driver`) reads the GNSS and publishes `sensor_msgs/NavSatFix`. An Extended Kalman Filter (EKF) node subscribes to these and publishes a fused 6-DOF state estimate (pose and velocity). The `thruster_controller` node subscribes to velocity commands and outputs motor signals to the Arduino.

1.3.1 ROS2 Ecosystem

Despite its name, the Robot Operating System (ROS) is not a traditional operating system. Instead, it is a collection of tools, libraries, and conventions designed to simplify the development of complex and scalable robotic software. In this project, we utilize the Python API of ROS2 Humble distribution.

Below is a concise overview of the key components within the ROS2 ecosystem:

- **Nodes:** Nodes are the basic execution units in ROS2. Each node represents an independent process that can communicate with other nodes. In our implementation, nodes will encapsulate different functional classes, such as a control node for each thruster, a main vehicle control node, and a simulation interface node.
- **Topics and Messages:** Nodes exchange data using a publish/subscribe messaging model. A topic is a named communication channel; nodes can send (publish) messages to a topic, or receive (subscribe to) messages from it. For instance, messages can include commands for thruster angles, GPS data, or wind velocity. A node can simultaneously publish and subscribe to different topics, enabling flexible data flow.
- **Services:** Unlike topics, which allow continuous and asynchronous data exchange, services provide a synchronous request-response mechanism. This enables a node to request a specific action or computation from another node. In reinforcement learning (RL) scenarios, this synchronous nature is advantageous and may avoid some of the complexity posed by asynchronous communication through topics.
- **Launch Files:** Launch files are written in Python and are used to start multiple nodes with their configurations simultaneously. We will use launch scripts provided by the competition organizers, with slight modifications to fit our setup.
- **Packages and Workspaces:** A package is the primary organizational unit in ROS2, grouping together related code, configuration files, and resources. All packages for a project reside within a workspace, a centralized directory structure used for building and managing packages. Although developing the entire project as a ROS2 package was considered, it was more practical for our needs to structure it as a standalone Python module.

ROS2 also provides a system named `tf2` (short for Transform version 2), which manages coordinate transformations in 3D space. These transformations are used to track the position and orientation of different robot components relative to one another.

To describe a robot's structure, geometry, physics, and visual properties, we use a standardized XML format. This format defines elements such as links, joints, sensors, and their visual appearances. The `robot_state_publisher` node reads this robot description file and publishes both static and dynamic transformsstatic for fixed joints and dynamic ones updated based on sensor input or control signals. This data is broadcasted on the `/robot_description` topic.

When launching the simulation environment via Gazebo, the information from this topic is used to spawn the robot model. As the simulation runs, a Gazebo plugin continuously updates the robots joint states and publishes them on the `/joint_states` topic. These updates are then read by the `robot_state_publisher` to refresh the dynamic transforms, thus completing the feedback loop between ROS2 and the simulation.

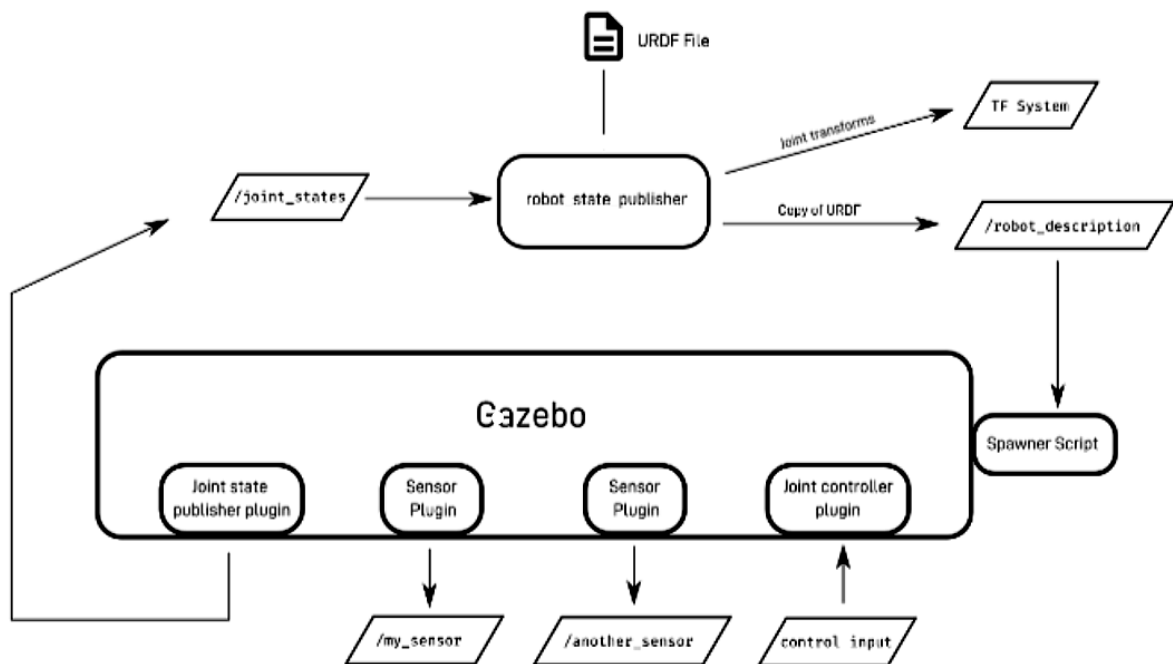


Figure 1.6: ROS Gazebo Integration

Figure 1.6 details the integration between the static robot model and the dynamic Gazebo environment. The architecture centers on the URDF File, which the `robot_state_publisher` utilizes to broadcast the `/robot_description` and maintain the kinematic tree within the TF System. To bridge the simulation with ROS, Gazebo employs specialized plugins: Sensor Plugins publish environmental data, while the Joint controller plugin translates ROS control input into simulated actuation. Crucially, the feedback loop is closed by the Joint state publisher plugin, which reports real-time joint angles to the `/joint_states` topic. The robot is initially loaded into the physics engine via a Spawner Script that consumes the robot description.

1.3.2 ROS2 Nodes and Protocols

Hardware interfacing relies on standard robotics protocols: I2C for the IMU, UART for GPS and telemetry radios, CAN for auxiliary buses, [12, 10]. As illustrated in Figure 1.7, the software architecture is structured as a modular network of ROS 2 nodes communicating via the Data Distribution Service (DDS). The `arduino_node` serves as the hardware bridge, acquiring sensor data and publishing it to topics such as `imu_data` (at ~ 100 Hz) and `gps_data` (at ~ 5 Hz). These inputs are consumed by the `ekf_node` to generate an `estimated_state`. This state estimation is synchronized with the trajectory from the navigation stack and processed by the `smc_controller_node` to compute the required `thruster_commands`. System deployment is handled via launch files, with the option to utilize node composition for enhanced real-time performance.

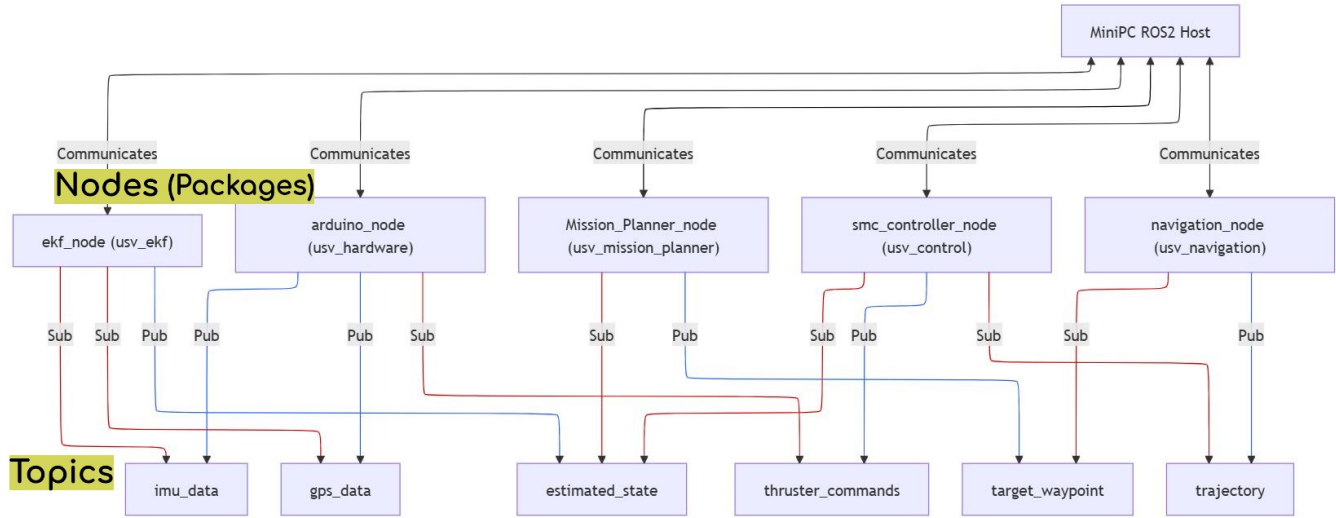


Figure 1.7: ROS 2 general Publish/Subscribe Diagram architecture

1.3.3 EKF State Estimation and Control Loop

Sensor fusion is achieved with an EKF-based state estimator (e.g., using the ROS `robot_localization` package). The EKF fuses high-rate IMU data with lower-rate GPS fixes to produce smooth position and orientation estimates. The estimated state is published (as `nav_msgs/Odometry` or `geometry_msgs/Pose`). A trajectory-tracking node then reads this state and compares it to desired waypoints or paths. It computes control commands (such as desired surge and yaw thrusts using a PID controller) and publishes them to the `thruster_controller` node. This control loop typically runs at 1050 Hz, closing the feedback loop for real-time trajectory following.

1.4 Operational Modes Implementation

The autonomous navigation system of the Unmanned Surface Vehicle (USV) incorporates two primary operational modes to accommodate diverse mission requirements: the **Predefined Trajectory mode** and the **Random Waypoints mode**.

In the Predefined Trajectory mode, the system loads a trajectory that has been specified a priori, which is particularly beneficial for structured missions such as surveying predefined areas or following predetermined routes. This loaded trajectory undergoes a validation process to ensure its feasibility and safety, checking for potential collisions with known obstacles and

verifying adherence to the vehicle's kinematic and dynamic constraints. Once validated, the trajectory is processed by the Path Planning and Optimization module, which computes an optimal path by considering various constraints and objectives, such as minimizing distance or time.

Conversely, the Random Waypoints mode is designed for dynamic exploration missions, where the USV autonomously generates waypoints within a defined operational area. This mode is especially suitable for tasks such as exploration, search and rescue operations, or environmental monitoring, where predefined paths are not available. In this mode, the system first generates a set of random waypoints within the operational area. These waypoints are then processed by a Waypoint Generation Algorithm, which creates a smooth and feasible trajectory that connects the waypoints while respecting the vehicle's dynamic constraints. The generated trajectory is subsequently fed into the same Path Planning and Optimization module used in the Predefined Trajectory mode, ensuring a consistent approach to path planning across both operational modes.

Following the path planning and optimization stage, both modes utilize a common sequence of processes to ensure safe and accurate navigation. The Extended Kalman Filter (EKF) integrates data from various sensors, including GPS and IMU, to provide a robust estimate of the USV's state, including its position, velocity, and orientation. This state estimation is crucial for accurate navigation and control. The Collision Avoidance module continuously monitors the environment for obstacles, both static and dynamic, and adjusts the planned path in real-time to avoid collisions. The adjusted path and the estimated state are then used by the Sliding Mode Control (SMC) controller, which computes control signals to minimize the error between the actual and desired trajectories. These control signals are ultimately sent to the actuators, which are the thruster motors that propel and steer the USV according to the planned path.

The integration of these two modes through shared processes and feedback loops ensures a flexible and robust system capable of adapting to various mission scenarios, from structured surveys to dynamic exploration tasks. The Predefined Trajectory mode offers reliability and predictability for missions with well-defined objectives, while the Random Waypoints mode provides flexibility and adaptability for missions requiring dynamic path generation. The consistent use of shared modules for path planning, state estimation, collision avoidance, and control ensures that the USV maintains robust performance across both operational modes, making the system adaptable to a wide range of operational scenarios.

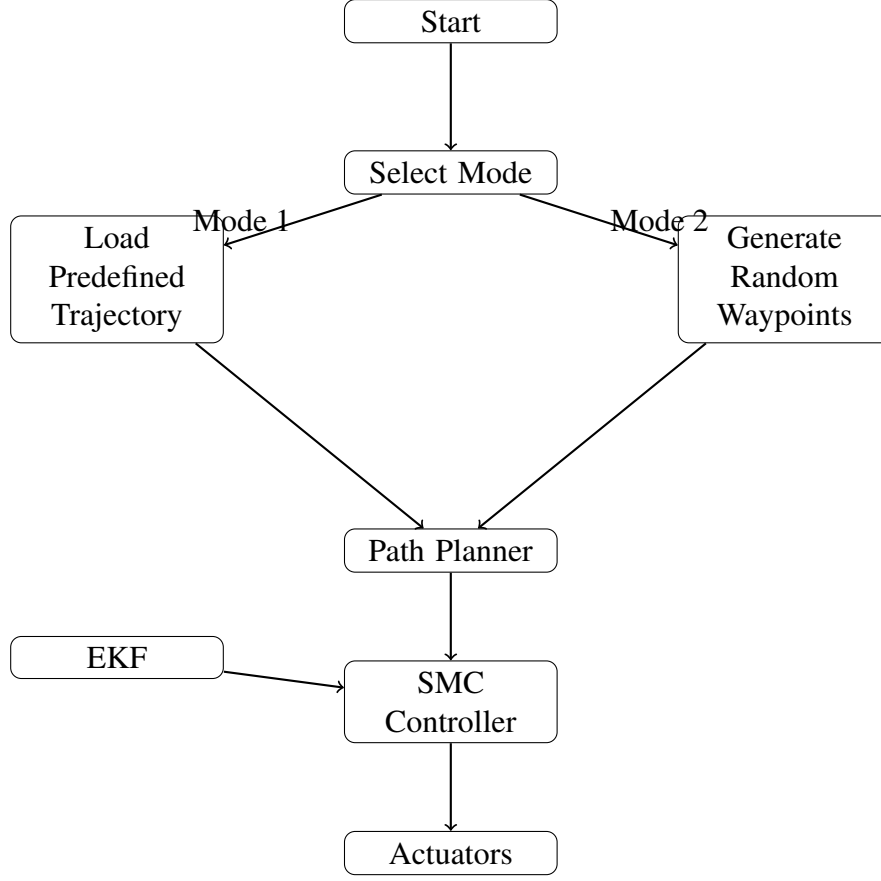


Figure 1.8: Implementation Flowchart for USV Operational Modes

Mode 1: Random Waypoint Generation

[H] [1] Define geofence boundaries: $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ mission_active $x_{\text{rand}} \leftarrow \text{rand}(x_{\min}, x_{\max})$
 $y_{\text{rand}} \leftarrow \text{rand}(y_{\min}, y_{\max})$ Add $(x_{\text{rand}}, y_{\text{rand}})$ to /desired_path Wait until USV reaches $\pm 1.5\text{m}$ of waypoint

Mode 2: Predefined Path Following

- **Trajectory Format:** CSV file with (x, y, ψ) waypoints.
- **Path Interpolation:** Cubic splines generate smooth trajectories between waypoints:

$$\eta_d(t) = \sum_{i=0}^3 a_i t^i \quad (1.1)$$

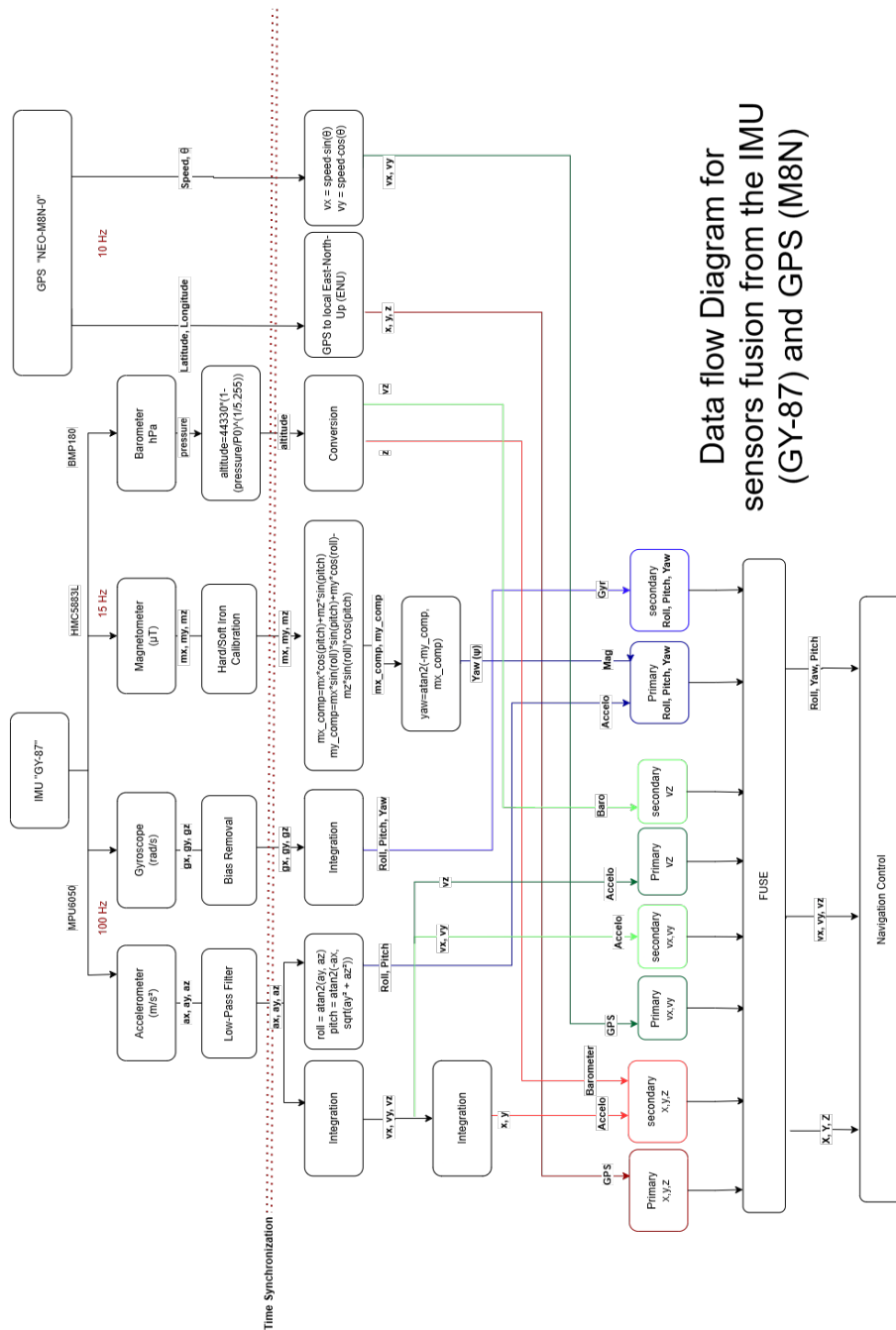
1.5 Sensor Fusion Logic

Primary/secondary assignments for each state component within the EKF:

- (x, y) : GPS primary, accelerometer secondary.
- z : GPS primary, barometer secondary.
- (v_x, v_y) : GPS Doppler primary, accelerometer secondary.

- v_z : accelerometer primary, barometer secondary.
- Roll/Pitch: accelerometer primary, gyroscope secondary.
- Yaw: magnetometer primary, gyroscope secondary.

As shown in Figure 1.9, the Data Flow Diagram (DFD) for sensor fusion between the IMU "GY87" and GPS "M8N" is depicted.



Data flow Diagram for sensors fusion from the IMU (GY-87) and GPS (M8N)

Figure 1.9: Data Flow Diagram for sensors fusion from the IMU "GY87" and GPS "M8N"

1.5.1 Primary/Secondary Sensor Assignments

In an Extended Kalman Filter (EKF) based navigation stack, we designate one sensor as *primary* for each state component providing the absolute or most reliable measurement and another as *secondary* providing high rate or complementary information. Below we explain the choice for each state:

- **Horizontal position (x, y) :**
 - *GPS (primary)* delivers absolute global latitude and longitude at moderate rates (120 Hz) with metre level accuracy.
 - *Accelerometer (secondary)* contributes short term inertial displacement via double integration of specific force, at high rates (up to 1 kHz), helping bridge GPS update intervals and smoothing GPS noise.
- **Altitude z :**
 - *GPS (primary)* also provides absolute altitude where available (with somewhat lower accuracy than horizontal).
 - *Barometer (secondary)* measures hydrostatic pressure continuously, yielding high-rate (110 Hz) relative changes in altitude, which corrects slow GPS altitude drift, especially in rough seas.
- **Horizontal velocity (v_x, v_y) :**
 - *Doppler log from GPS (primary)* or dedicated Doppler Velocity Log (DVL) gives direct velocity over ground or seabed with high accuracy (0.21 %), crucial for trajectory following.
 - *Accelerometer (secondary)* infers velocity by integrating acceleration; its high update rate and low latency help capture rapid maneuvers or correct temporary Doppler dropouts.
- **Vertical velocity v_z :**
 - *Accelerometer (primary)* measures vertical acceleration directly, which upon integration yields high rate estimates of ascent/descent velocity.
 - *Barometer (secondary)* provides slower but drift free height changes, constraining long term bias in the integrated accelerometer signal.
- **Roll and pitch:**
 - *Accelerometer (primary)* senses the gravity vector when the vehicle is quasistatic, giving direct absolute roll/pitch angles.
 - *Gyroscope (secondary)* measures angular rates, capturing dynamic attitude changes between accelerometer based updates and reducing noise in the static estimate.
- **Yaw (heading):**
 - *Magnetometer (primary)* provides an absolute Earth referenced heading, compensating for gyro drift.

- *Gyroscope (secondary)* measures rotational rate around the vertical axis, offering high bandwidth to follow rapid yaw maneuvers and to interpolate between magnetometer readings.

In each case, the *primary* sensor gives the absolute or most accurate measurement (though at lower rate or with larger noise), while the *secondary* sensor provides highrate or complementary information that stabilizes and interpolates the state estimate between primary updates. This dualsensor approach maximizes accuracy, robustness to dropouts, and responsiveness for USV navigation and trajectory control.

1.6 Dynamics Modeling

To design robust control algorithms and validate system performance in simulation, an accurate mathematical model of the USV is essential. This section derives the 3-DOF planar dynamics in surge, sway, and yaw and specifies the hydrodynamic coefficients.

1.6.1 Kinematic Equations

Let $\eta = [x, y, \psi]^T$ denote the earth-fixed pose and $v = [u, v, r]^T$ the body-fixed velocity [23]. The kinematic relationship is:

$$\dot{\eta} = R(\psi) v, \quad R(\psi) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (1.2)$$

which relate bodyfixed velocities to inertial motion [24].

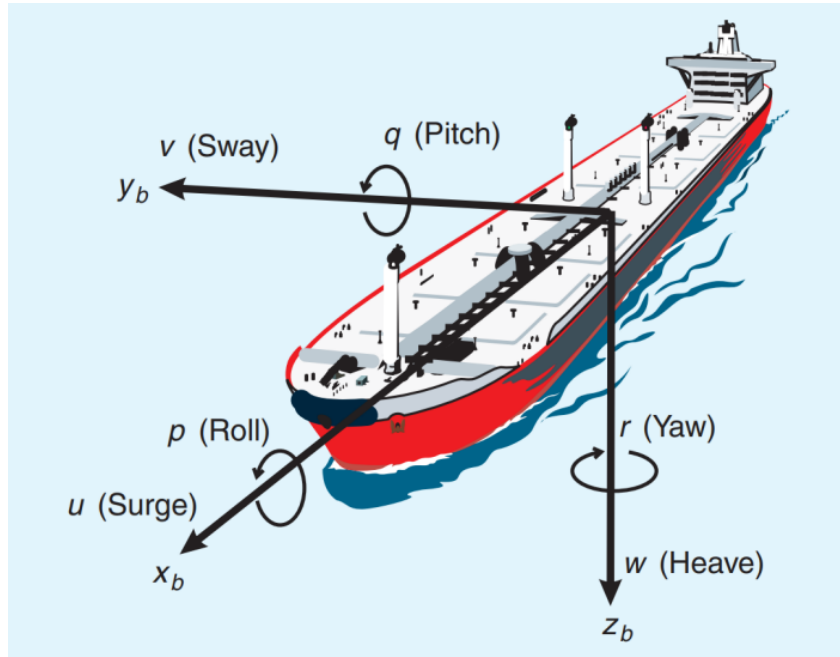


Figure 1.10: Motion of the USV in 6 DOF [1].

Figure 1.10 illustrates the body-fixed coordinate system and the associated motion variables commonly used to describe the dynamics of marine vehicles. The figure defines the vehicle-fixed reference frame (x_b, y_b, z_b) together with the six degrees of freedom: translational motions in surge (u), sway (v), and heave (w), as well as rotational motions in roll (p), pitch (q), and yaw (r). In the context of planar surface-vehicle motion, the dynamics are reduced to three degrees of freedom, namely surge, sway, and yaw. This representation provides the physical interpretation of the velocity components and establishes the kinematic relationship between body-fixed velocities and inertial-frame position and heading.

The body-fixed velocity components $\mathbf{v} = [u, v, r]^\top$ are defined as follows:

- u : longitudinal (surge) velocity along the vehicle's forward axis (m/s). This is the component of velocity in the forward direction of the body frame.
- v : lateral (sway) velocity along the vehicle's sideways axis (m/s). This is the component of velocity perpendicular to the forward axis (side slip).
- r : angular velocity about the body z -axis (yaw rate, in rad/s). It represents the rate of change of the vehicle's heading.

The inclusion of r is essential because the inertial pose $\boldsymbol{\eta} = [x, y, \psi]^\top$ contains the heading angle ψ , the time derivative of this angle is an angular velocity that is independent of the translational components u and v .

From (1.2) we obtain the scalar kinematic relations:

$$\dot{x} = u \cos \psi - v \sin \psi, \quad (1.3)$$

$$\dot{y} = u \sin \psi + v \cos \psi, \quad (1.4)$$

$$\dot{\psi} = r. \quad (1.5)$$

Thus u and v describe translation in the body frame while r describes rotation about the vertical axis. Typical units are u, v in meters per second (m/s) and r in radians per second (rad/s).

Note on sign convention: the sign convention for v and r depends on the chosen body-frame orientation. Above we assume the common body-frame convention with $+u$ forward and the right-hand rule for positive rotation about the z -axis (positive r corresponds to counterclockwise rotation of the vehicle when viewed from above).

1.6.2 Equations of Motion

Applying Newton–Euler to a surface vessel yields:

$$\mathbf{M} \dot{\mathbf{v}} + \mathbf{C}(\mathbf{v}) \mathbf{v} + \mathbf{D}(\mathbf{v}) \mathbf{v} = \boldsymbol{\tau} + \boldsymbol{\tau}_w, \quad (1.6)$$

where:

- $\mathbf{M} = \mathbf{M}_R + \mathbf{M}_A$ is the system mass matrix (rigid-body plus added mass).
- $\mathbf{C}(\mathbf{v}) = \mathbf{C}_R(\mathbf{v}) + \mathbf{C}_A(\mathbf{v})$ is the Coriolis–centripetal matrix incorporating added mass effects.
- $\mathbf{D}(\mathbf{v})$ is the hydrodynamic damping matrix, typically including linear and quadratic drag.
- $\boldsymbol{\tau} = [\tau_u, 0, \tau_r]^\top$ is the control input vector (surge force and yaw moment).

- τ_w represents environmental disturbances (wind, current, waves).

Explicitly, for 3-DOF planar motion:

$$M = \begin{bmatrix} m - X_{\ddot{u}} & 0 & 0 \\ 0 & m - Y_{\ddot{v}} & 0 \\ 0 & 0 & I_z - Z_{\ddot{r}} \end{bmatrix}, \quad (1.7)$$

$$C(v) = \begin{bmatrix} 0 & 0 & -(m - Y_{\dot{v}})v \\ 0 & 0 & (m - X_{\dot{u}})u \\ (m - Y_{\dot{v}})v & -(m - X_{\dot{u}})u & 0 \end{bmatrix}, \quad (1.8)$$

$$D(v) = \begin{bmatrix} X_u + X_{|u|}|u| & 0 & 0 \\ 0 & Y_v + Y_{|v|}|v| & 0 \\ 0 & 0 & N_r + N_{|r|}|r| \end{bmatrix}. \quad (1.9)$$

Here, $X_{\ddot{u}}, Y_{\ddot{v}}, Z_{\ddot{r}}$ are added mass terms, and $X_u, X_{|u|}, \dots$ are linear and quadratic damping coefficients.

1.6.3 Linearization for Control Design

To facilitate controller synthesis, the nonlinear model is linearized around a steady operating point (trim condition) defined by

$$v_0 = [u_0, v_0, r_0]^\top = [u_0, 0, 0]^\top. \quad (1.10)$$

Linearizing (1.16) about this equilibrium yields the following state-space representation:

$$\Delta \dot{v} = A \Delta v + B \Delta \tau, \quad (1.11)$$

where the state deviation is given by

$$\Delta v = v - [u_0, 0, 0]^\top, \quad (1.12)$$

and the input deviation by

$$\Delta \tau = \tau - \tau_0. \quad (1.13)$$

The system matrices are defined as

$$A = -M^{-1} \left(\frac{\partial C}{\partial v} + \frac{\partial D}{\partial v} \right) \Big|_{v_0}, \quad (1.14)$$

$$B = M^{-1}. \quad (1.15)$$

This linearized model provides a convenient basis for classical control design methods such as PID tuning, Linear Quadratic Regulator (LQR) synthesis, or Model Predictive Control (MPC).

Starting from the nonlinear equations of motion,

$$M \dot{v} + C(v) v + D(v) v = \tau + \tau_w, \quad (1.16)$$

we define the velocity-dependent nonlinear term as

$$N(v) = C(v) v + D(v) v. \quad (1.17)$$

The dynamics can then be expressed in compact form as

$$\dot{\mathbf{v}} = M^{-1}(\boldsymbol{\tau} + \boldsymbol{\tau}_w - N(\mathbf{v})). \quad (1.18)$$

Let $(\mathbf{v}_0, \boldsymbol{\tau}_0)$ denote a trim point satisfying the steady-state condition

$$N(\mathbf{v}_0) = \boldsymbol{\tau}_0 + \boldsymbol{\tau}_{w,0}. \quad (1.19)$$

Introducing perturbations $\mathbf{v} = \mathbf{v}_0 + \Delta\mathbf{v}$ and $\boldsymbol{\tau} = \boldsymbol{\tau}_0 + \Delta\boldsymbol{\tau}$, a first-order Taylor expansion of $N(\mathbf{v})$ about $\mathbf{v}_0$ gives

$$N(\mathbf{v}_0 + \Delta\mathbf{v}) \approx N(\mathbf{v}_0) + \left. \frac{\partial N}{\partial \mathbf{v}} \right|_{\mathbf{v}_0} \Delta\mathbf{v}. \quad (1.20)$$

Neglecting higher-order terms and assuming small or slowly varying disturbances, the linearized dynamics become

$$\Delta\dot{\mathbf{v}} = M^{-1} \left(\Delta\boldsymbol{\tau} - \left. \frac{\partial N}{\partial \mathbf{v}} \right|_{\mathbf{v}_0} \Delta\mathbf{v} \right). \quad (1.21)$$

This leads directly to the standard state-space form

$$\Delta\dot{\mathbf{v}} = A \Delta\mathbf{v} + B \Delta\boldsymbol{\tau}, \quad (1.22)$$

with

$$A = -M^{-1} \left. \frac{\partial N}{\partial \mathbf{v}} \right|_{\mathbf{v}_0}, \quad B = M^{-1}. \quad (1.23)$$

Since $N(\mathbf{v}) = C(\mathbf{v})\mathbf{v} + D(\mathbf{v})\mathbf{v}$, the Jacobian can be expanded using the product rule:

$$\left. \frac{\partial}{\partial \mathbf{v}} (C(\mathbf{v})\mathbf{v}) \right|_{\mathbf{v}_0} = C(\mathbf{v}_0) + \sum_{k=1}^3 \mathbf{v}_{0,k} \left. \frac{\partial C}{\partial \mathbf{v}_k} \right|_{\mathbf{v}_0}, \quad (1.24)$$

and similarly for the damping term $D(\mathbf{v})$. Consequently,

$$\left. \frac{\partial N}{\partial \mathbf{v}} \right|_{\mathbf{v}_0} = C(\mathbf{v}_0) + D(\mathbf{v}_0) + \sum_{k=1}^3 \mathbf{v}_{0,k} \left. \frac{\partial C}{\partial \mathbf{v}_k} \right|_{\mathbf{v}_0} + \sum_{k=1}^3 \mathbf{v}_{0,k} \left. \frac{\partial D}{\partial \mathbf{v}_k} \right|_{\mathbf{v}_0}. \quad (1.25)$$

For the common operating condition $\mathbf{v}_0 = [u_0, 0, 0]^\top$, many terms simplify:

- Contributions proportional to $\mathbf{v}_0$ and $\mathbf{r}_0$ vanish.
- The summation terms reduce to the single component proportional to u_0 .

Under these assumptions, the Jacobian is often written in the compact form

$$A \approx -M^{-1} \left(\left. \frac{\partial C}{\partial \mathbf{v}} + \frac{\partial D}{\partial \mathbf{v}} \right) \right|_{\mathbf{v}_0}, \quad (1.26)$$

with the understanding that the full product-rule terms should be retained whenever they do not vanish for the chosen model.

1.6.4 Hydrodynamic Coefficient Identification

The hydrodynamic coefficients appearing in the matrices M , C , and D can be obtained using several complementary approaches. Analytical methods based on potential flow theory are commonly employed to derive initial estimates of the added mass terms. These estimates can be refined through Computational Fluid Dynamics (CFD) simulations, which provide a more accurate evaluation of drag and damping effects. In addition, experimental procedures such as free-decay tests or forced oscillation experiments, conducted either in a towing tank or during on-water trials, allow for data-driven system identification.

Once identified, the resulting parameters are first validated in simulation to assess their consistency and robustness before being used in real-world experiments.

1.7 Control System Design

This section presents the control system architecture developed for the surface vehicle, including the choice of control strategies, parameter tuning methodologies, and performance validation through simulation.

A **Sliding Mode Control (SMC)** approach is first adopted due to its well-known robustness against modeling uncertainties, external disturbances, and nonlinearities. The SMC controller is mainly used to ensure accurate tracking of the vehicle's desired position and heading under varying environmental conditions.

In addition to the SMC, an **adaptive PID controller based on Recursive Least Squares (RLS)** and a discrete **ARX model** are also implemented. This data-driven approach allows for online estimation of system parameters and real-time adjustment of the PID gains. It provides a flexible and lightweight solution when full analytical modeling is impractical or when system dynamics vary over time.

A **thruster allocation module** is included to convert the generated control efforts (from either controller) into actual left and right thruster forces, ensuring the proper distribution of commands based on the vehicles configuration.

Simulation results demonstrate promising performance for both control strategies. While SMC offers robustness and insensitivity to perturbations, the RLS-based adaptive PID controller achieves precise tracking with low steady-state error, particularly when accurate system identification is available.

Future work will focus on performing a comparative study between the SMC and the adaptive PID controller, considering various operating conditions, noise levels, and environmental disturbances. This analysis will help determine the most suitable control architecture for real world deployment of the AquaRob surface vehicle.

1.7.1 Sliding surfaces and SMC design

The principle of sliding mode control lies in forcing the system's state to reach, in finite time, a predefined hypersurface within the state space, and then constraining it to remain on that surface thereafter. This hypersurface, being a relation between the system's state variables, defines a differential equation that entirely governs the systems dynamics as long as the trajectory stays on the surface.

Once the system is driven onto this hypersurface by an appropriate control law, its subsequent behavior becomes independent of the original system dynamics or any external disturbances. It is instead governed solely by the characteristics of the hypersurface itself. As a

result, the closed-loop system is not merely robust with respect to model uncertainties and external perturbations it becomes entirely insensitive to them, provided the control law is capable of effectively rejecting such disturbances.

To ensure the onset of the sliding regime and to make the chosen surface attractive, the control input must be discontinuous with respect to the sliding variable S .

$$u = -K \cdot \text{sign}(S) \quad (1.27)$$

With:

$$\text{sign}(\cdot) = \begin{cases} 1 & \text{if } (\cdot) \geq 0 \\ -1 & \text{if } (\cdot) < 0 \end{cases} \quad (1.28)$$

K is a strictly positive gain.

Numerous strategies have been proposed to reduce or eliminate the phenomenon of **chattering**, which is commonly observed in sliding mode control systems due to the discontinuous nature of the control law.

Notably, approaches by Utkin, Asada, and Slotine have laid the foundation for these developments.

In classical formulations, the sliding variable S appears in the control law through the discontinuous function $\text{sign}(S)$.

The principle of continuous switching functions within a boundary layer near the sliding surface:

The phenomenon known as **chattering** is primarily caused by the system's trajectory oscillating around the sliding surface, rather than remaining exactly on it. This oscillatory behavior typically occurs within a small neighborhood of the surface, due to the high-frequency switching of the control input.

To mitigate this effect, one commonly used approach involves **replacing the discontinuous switching function** (e.g., $\text{sign}(S)$) with a **continuous approximation** that operates within a predefined boundary layer around the sliding surface. Within this layer, the control law transitions smoothly rather than abruptly, thereby reducing high-frequency oscillations while still guiding the system toward the surface.

Comparative Study of Chattering Reduction Techniques in Sliding Mode Control

To address the problem of chattering, several solutions have been proposed, which involve **replacing the discontinuous switching function** (typically the sign function, $\text{sign}(S)$) with **continuous alternatives** within a boundary layer around the sliding surface.

The most commonly used continuous approximations include:

The saturation function - Slotine and Coetsee:

Slotine and Coetsee proposed a method that involves replacing the discontinuous sign function $\text{sign}(S)$ with a linear function of slope 1 within a boundary layer of width 2Φ centered around the sliding surface. Outside this boundary layer, the control law retains its original discontinuous form.

The control law is expressed as:

$$u_{\text{sliding}} = -K \cdot \text{sat}\left(\frac{S}{\Phi}\right) \quad (1.29)$$

Where the **saturation function** $\text{sat}\left(\frac{S}{\Phi}\right)$ is defined as:

$$\text{sat}\left(\frac{S}{\Phi}\right) = \begin{cases} \text{sign}(S) & \text{if } |S| > \Phi \\ \frac{S}{\Phi} & \text{if } |S| \leq \Phi \end{cases} \quad (1.30)$$

where $\Phi > 0$ defines the thickness of the boundary layer.

This approach smooths the control action near the sliding surface thus reducing **chattering** while retaining the robustness benefits of the traditional sliding mode control outside the boundary layer.

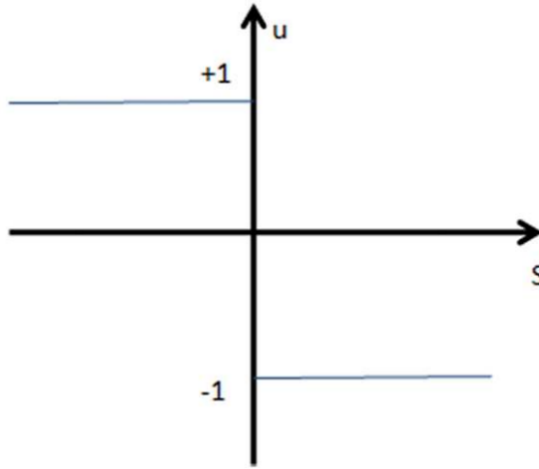


Figure 1.11: $\text{sign}(S)$ function

The cont function Harashima and al.:

Another solution to **smooth the control law near the sliding surface** defined by $S = 0$ consists of replacing the discontinuous switching function $\text{sign}(S)$ with a **continuous smoothing function** within a boundary layer $|S| < \Phi$. The control law takes the following form:

$$u_{\text{sliding}} = -K \cdot \text{cont}(S) \quad (1.31)$$

where the smoothing function $\text{cont}(S)$ is defined as:

$$\text{cont}(S) = \begin{cases} -\text{sign}(S) & \text{if } |S| > \Phi \\ \frac{S}{|S| + \delta} & \text{if } |S| \leq \Phi \end{cases} \quad (1.32)$$

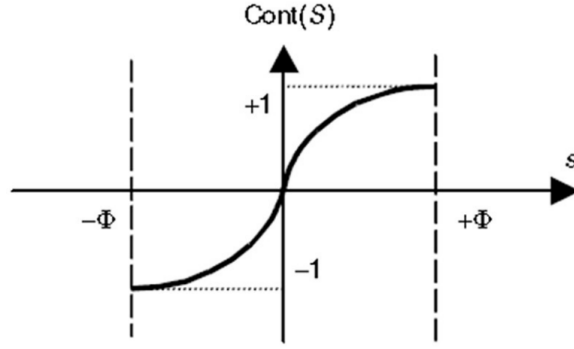


Figure 1.12: Cont function

To further improve performance and adaptability, the parameter δ is defined **adaptively** as:

$$\delta = \begin{cases} 0 & \text{if } |S| > \Phi \\ \delta_0 + |\eta| & \text{if } |S| \leq \Phi \end{cases} \quad (1.33)$$

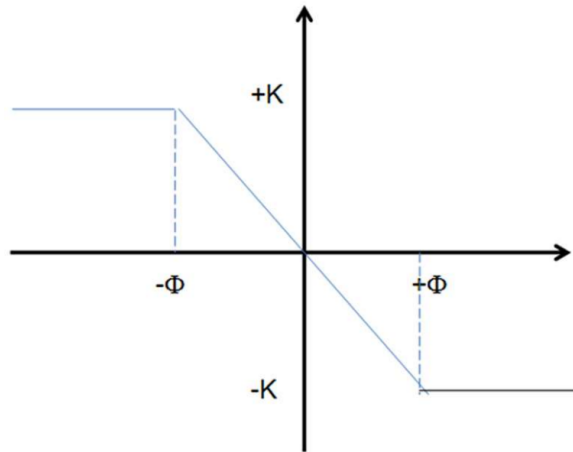


Figure 1.13: Saturation function

where η is an **integral term** based on S , defined by:

$$\eta = \begin{cases} 0 & \text{if } |S| > \Phi \\ \eta_0 + \int_0^t S(\tau) d\tau & \text{if } |S| \leq \Phi \end{cases} \quad (1.34)$$

This adaptive mechanism improves system response by adjusting the smoothing intensity in real-time, based on the trajectory of the sliding variable S .

The hyperbolic tangent function:

$$u_{\text{sliding}} = \tanh\left(\frac{S}{\Phi}\right) \quad (1.35)$$

These continuous functions ensure a smoother transition near the sliding surface, effectively reducing high-frequency oscillations while maintaining acceptable tracking performance and

robustness. However, it is important to note that introducing such approximations may slightly compromise the ideal robustness of the sliding mode.

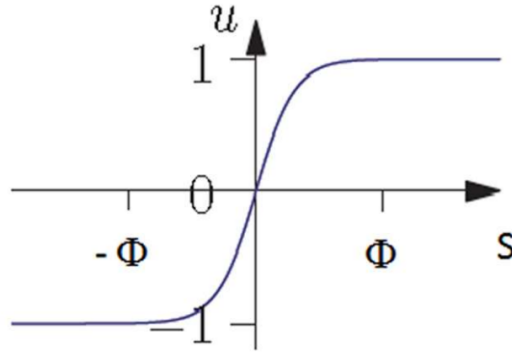


Figure 1.14: Hyperbolic tangent function

Harashima et al.

An alternative expression for the sliding mode control law has also been proposed:

$$u_{\text{sliding}} = -(\omega e + \Gamma \dot{e} + \gamma \cdot \text{sign}(S)) \quad (1.36)$$

where ω , Γ , and γ are strictly positive constants.

This formulation combines **proportional**, **derivative**, and **discontinuous** terms, offering a balance between fast convergence, noise rejection, and robustness against disturbances and uncertainties.

Sliding surfaces

The sliding variable S defined as:

$$S = \left(\frac{d}{dt} + \lambda \right)^n \left(\int_0^t e(\tau) d\tau \right) \quad (1.37)$$

This formulation incorporates an integral term, allowing the controller to compensate for persistent disturbances or steady-state tracking errors.

An alternative expression for the sliding surface is also proposed as:

$$S = \left(\frac{d}{dt} + \lambda \right)^{n-1} e \quad (1.38)$$

This form is more classical and corresponds to a differential sliding surface of order $n - 1$, typically used in high-order sliding mode control, allowing better tuning of the system's dynamic response.

1.7.2 Adaptive PID Controller Based on RLS

Despite its conceptual simplicity, the **PID controller** remains a preferred solution in many industrial and aerospace applications due to its **robustness**, ease of implementation, and interpretability. Its effectiveness, however, strongly depends on the appropriate tuning of its three

gains: **proportional (Kp), integral (Ki), and derivative (Kd)**.

While classical tuning methods (e.g., ZieglerNichols, CohenCoon) may provide satisfactory performance under **fixed and well-known operating conditions**, they often fail to maintain optimal control when the system exhibits **time-varying or uncertain dynamics**, as is the case with the AquaRob surface vehicle.

To overcome this limitation, **adaptive PID controllers** are employed. These controllers **dynamically adjust the PID gains** in real time, enhancing resilience and maintaining system stability even in the presence of disturbances, unmodeled dynamics, or changing environments.

In this work, an **adaptive PID control strategy** is implemented using the **Recursive Least Squares (RLS)** algorithm as an online estimator of the systems behavior. The estimated parameters from a discrete ARX model are used to update the PID gains continuously. This allows the controller to track variations in the systems dynamics and preserve performance, even when the AquaRob operates under different conditions or mission profiles.

1.8 Conclusion

In summary, the system design and architecture of AquaRob combine a resilient hardware platform with a flexible and modular software stack to achieve a higher level of autonomous operation. The integration of redundant sensing, reliable communication, and effective state estimation ensures accurate navigation, while the dual-mode operational framework provides adaptability for a wide range of missions. Complementary control strategies, supported by dynamic modeling, further enhance the vehicles ability to cope with environmental disturbances and uncertainties.

This holistic design approach establishes a strong foundation for the development, testing, and deployment of AquaRob. By bridging hardware resilience, software modularity, and advanced control, the system is positioned to operate effectively in real-world aquatic scenarios with high levels of autonomy and reliability.

Chapter 2

Implementation Details

2.1 Introduction

This chapter details the practical implementation of the autonomous Unmanned Surface Vehicle (USV) platform developed in this work. It focuses on the integration of hardware and software components, as well as the tools and methodologies used to ensure reliability, modularity, and reproducibility throughout the development process.

The chapter begins by presenting the overall development workflow, which follows a simulation-first approach. All algorithms and ROS 2 nodes are initially designed, tested, and validated on a desktop computer using a realistic Gazebo-based maritime simulation environment before being deployed on the embedded Raspberry Pi 5 system. This strategy significantly reduces development time, facilitates debugging, and minimizes risks during hardware integration.

Subsequent sections describe the hardware integration, including sensor wiring, calibration procedures, and the role of the Arduino Due as a low-level interface for sensor acquisition and actuator control. The ROS 2 software architecture is then detailed, covering sensor processing nodes, dual Extended Kalman Filter (EKF) localization, transformation frame management, and the Sliding Mode Controller (SMC) used for path-following.

The chapter also introduces the WAM-V simulation framework, sensor modeling strategies, and hydrodynamic configuration used to achieve realistic maritime behavior. Finally, validation and visualization tools such as RViz2 and PlotJuggler are presented, demonstrating how system performance, state estimation accuracy, and control behavior are monitored and verified.

Together, these elements provide a complete and reproducible implementation framework that bridges simulation and real-world deployment, laying the foundation for autonomous navigation in challenging dam and reservoir environments.

2.2 Development Workflow

To accelerate development and debugging, all algorithms and ROS2 nodes were initially written and tested on a desktop PC running Ubuntu 22.04 and ROS2 Jazzy:

- **Desktop Simulation:** The complete software stack, including the WAM-V model, sensor plugins, and Sydney Regatta world, was run in Gazebo. Synthetic sensor data was bridged into ROS2, allowing for full hardware-in-the-loop verification of the dual EKF state estimation and SMC path-following controller without any physical hardware.
- **PC-Based Hardware Test:** After successful simulation tests, the code will be tested with real hardware. The desktop PC will be connected via USB to the Arduino Due,

streaming live data from the GY-87 IMU and NEO-M8N GPS. This data was visualized in RViz and PlotJuggler to fine-tune filter parameters and controller gains before embedded deployment.

After achieving stable and robust performance on the PC, the entire ROS2 workspace was cross-compiled and deployed to the Raspberry Pi 5 for standalone, embedded operation.

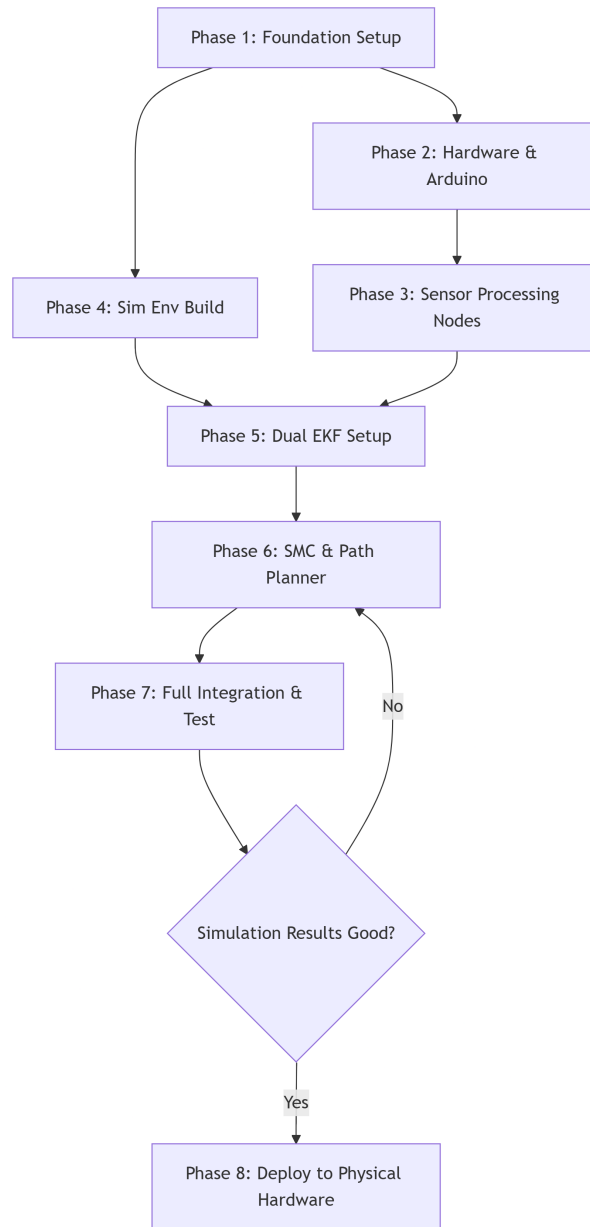


Figure 2.1: Workflow Summary Diagram

2.3 Hardware Integration

2.3.1 Sensor Wiring and Arduino Interface

The GY-87 IMU and NEO-M8N GPS are wired to an Arduino Due and communicate with the Raspberry Pi over USB-UART:

- **GY-87 IMU:** Connected via I2C (SDA: Pin 20, SCL: Pin 21). The MPU6050 (accelerometer/gyro) and HMC5883L (magnetometer) are calibrated using the MPU6050_Kalman and HMC5883L libraries.
- **NEO-M8N GPS:** Uses UART (TX: Pin 19, RX: Pin 18) with the TinyGPS++ library for NMEA parsing.
- **Serial Link to Raspberry Pi:** Arduino publishes sensor data to RPi via USB-UART at 115200 baud using a custom sensor_msgs/Imu ROS 2 message.

Figure 2.2 presents the schematic wiring diagram of the GY-87 IMU and the NEO-M8N GPS modules connected to the Arduino Due, highlighting the use of the I2C bus for inertial sensing and a dedicated UART interface for GPS communication. This diagram clarifies the allocation of power, ground, and communication pins, ensuring correct electrical interfacing. Figure 2.3 shows the corresponding prototype assembled on a breadboard, where the Arduino Due, IMU, GPS receiver, and antenna are physically deployed. This setup was used for initial validation of sensor data acquisition and communication before final system integration.

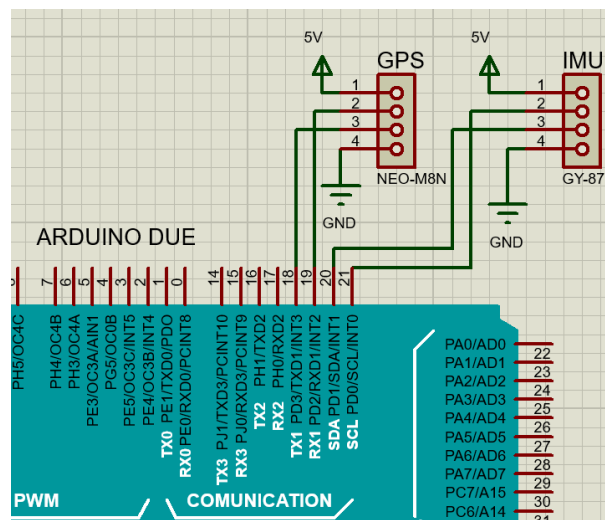


Figure 2.2: Physical wiring of the GY-87 IMU and NEO-M8N GPS on the Arduino DUE.

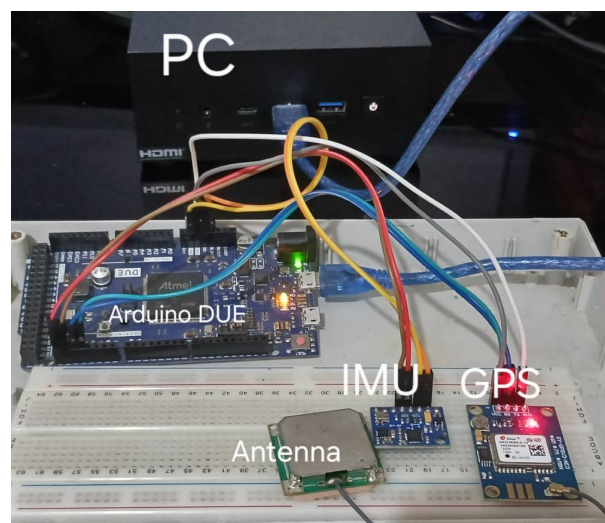


Figure 2.3: Arduino Due-Based IMU and GPS Integration Prototype on Breadboard

2.3.2 Sensor Calibration

IMU Calibration

The GY-87 IMU requires calibration to mitigate biases and misalignments:

- **Accelerometer/Gyroscope:**
 - Placed on a level surface, and biases are computed using the MPU6050_Kalman library.
 - Bias values stored in EEPROM: accel_bias, gyro_bias.
- **Magnetometer:**
 - Calibrated using the MagCal tool to compensate for hard/soft iron distortions.

IMU Calibration

The GY-87 IMU was calibrated to compensate for sensor biases and misalignments. The following offsets were computed:

Table 2.1: IMU Calibration Results

Sensor	X	Y	Z
Accelerometer (m/s ²)	0.1599	-0.3773	0.4447
Gyroscope (rad/s)	-0.0370	-0.0247	-0.0080
Magnetometer (T)	-0.05	-0.05	-0.05

GPS Calibration

- **Static Position Averaging:** Collected 100 samples at startup to compute initial position offset.

2.3.3 Barometer to Altitude Node

Converts pressure readings to altitude using the barometric formula:

$$\text{altitude} = \left(1 - \left(\frac{\text{pressure}}{\text{sea_level_pressure}} \right)^{0.190284} \right) \times 145366.45 \quad (2.1)$$

```
1 import rclpy
2 from rclpy.node import Node
3 from sensor_msgs.msg import FluidPressure
4 from nav_msgs.msg import Odometry
5 import math
6
7 class BarometerToAltitude(Node):
8     def __init__(self):
9         super().__init__('barometer_to_altitude')
10        self.declare_parameter('sea_level_pressure', 1013.25)
```

```

11         self.sea_level_pressure = self.get_parameter('
           sea_level_pressure').value
12
13         self.subscription = self.create_subscription(
14             FluidPressure,
15             '/usv/barometer',
16             self.callback,
17             10)
18
19         self.publisher_ = self.create_publisher(Odometry, '/usv1/
           baro_altitude', 10)
20
21     def callback(self, msg):
22         pressure = msg.fluid_pressure # hPa
23         altitude = (1 - math.pow(pressure / self.
           sea_level_pressure, 0.190284)) * 145366.45 # meters
24
25         odom_msg = Odometry()
26         odom_msg.header = msg.header
27         odom_msg.header.frame_id= "odom"
28         odom_msg.child_frame_id= "base_link"
29         odom_msg.pose.pose.position.z = altitude
30         odom_msg.pose.covariance[14] = 0.2 # Covariance for z-
           axis
31
32         self.publisher_.publish(odom_msg)
33
34     def main(args=None):
35         rclpy.init(args=args)
36         node = BarometerToAltitude()
37         rclpy.spin(node)
38         node.destroy_node()
39         rclpy.shutdown()
40
41 if __name__ == '__main__':
42     main()

```

Listing 2.1: Barometer to Altitude Node

2.3.4 Magnetometer Processor

- **Purpose:** Calculates heading from magnetic field
- **Core Equation:**

$$\psi = \tan^{-1} \left(\frac{M_y}{M_x} \right) \quad (2.2)$$

- **Output:** /usv1/magnetic_heading
- **Features:** Hard/soft iron compensation

```

1 #!/usr/bin/env python3
2 import rclpy
3 from rclpy.node import Node
4 from sensor_msgs.msg import MagneticField, Imu
5 from geometry_msgs.msg import Quaternion
6 import math
7
8 class MagnetometerProcessor(Node):
9     def __init__(self):
10         super().__init__('magnetometer_processor')
11         self.declare_parameter('magnetic_declination', 1.16)
12         self.declination = self.get_parameter('
            magnetic_declination').value
13
14         self.sub = self.create_subscription(
15             MagneticField,
16             '/usv/magnetic_field',
17             self.callback,
18             10
19         )
20         self.pub = self.create_publisher(
21             Imu,
22             '/usv1/magnetic_heading',
23             10
24         )
25
26     def callback(self, msg):
27         # Extract magnetic field components
28         mx = msg.magnetic_field.x
29         my = msg.magnetic_field.y
30
31         # Calculate raw heading (ignore Z for 2D vehicles)
32         heading = math.atan2(my, mx)
33
34         # Apply magnetic declination correction
35         corrected_heading = heading + self.declination
36
37         # Normalize to [0, 2]
38         if corrected_heading < 0:
39             corrected_heading += 2 * math.pi
40
41         # Create quaternion from heading (rotation around Z-axis)
42         q = Quaternion()
43         q.z = math.sin(corrected_heading / 2)
44         q.w = math.cos(corrected_heading / 2)
45
46         # Create IMU message
47         imu_msg = Imu()
48         imu_msg.header = msg.header
49         imu_msg.header.frame_id = 'base_link'
50         imu_msg.orientation = q

```

```

51
52     # Set covariance - focus on yaw component (index 8)
53     imu_msg.orientation_covariance = [0.0] * 9
54     imu_msg.orientation_covariance[0] = -1 # Invalidate roll
55     imu_msg.orientation_covariance[4] = -1 # Invalidate
56         pitch
57     imu_msg.orientation_covariance[8] = 0.1 # Yaw variance
58
59     # Invalidate other data
60     imu_msg.angular_velocity_covariance[0] = -1
61     imu_msg.linear_acceleration_covariance[0] = -1
62
63     self.pub.publish(imu_msg)
64
65 def main(args=None):
66     rclpy.init(args=args)
67     node = MagnetometerProcessor()
68     rclpy.spin(node)
69     rclpy.shutdown()
70
71 if __name__ == '__main__':
72     main()

```

Listing 2.2: Magnetometer node

2.3.5 Arduino direct implementation

The Arduino handles low-level sensor reading and thruster control. It publishes the following ROS 2 topics:

- /usv/imu_data: IMU measurements
- /usv/gps_data: GPS position data
- /usv/magnetic_field: Magnetometer readings
- /usv/barometer: Atmospheric pressure data that will be converted later to altitude

The communication bridge between the low-level hardware and the high-level ROS 2 graph is established using the `micro_ros_agent`. Figure 2.4 demonstrates the successful initialization of this agent on the host computer, the terminal output highlights the handshake process, confirming the session establishment and the automatic creation of publishers and subscribers over the serial port (/dev/ttyACM0).

Prior to integrating with the agent, the sensor drivers were validated independently. Figure 2.5 displays the raw coordinate data (latitude and longitude) read directly from the GPS receiver, ensuring valid satellite acquisition before transmission. Finally, the end-to-end data pipeline was verified by monitoring the IMU topic on the host, as shown in Figure 2.6, the system successfully receives and decodes real-time inertial data, including accelerometer vectors and orientation (heading, pitch, roll).

The EKF direct implementation on Arduino Data:

The Arduino is responsible for publishing raw sensor data, which is then processed and fused by various ROS nodes. Here's a breakdown of the key components:

1. Arduino Data Publishing (Top Level)

The Arduino code publishes raw sensor data under topics like:

- `/sensor_node/imu_data` (Inertial Measurement Unit: accelerometer, gyroscope)
- `/sensor_node/magnetic_field` (Magnetometer data for heading)
- `/sensor_node/barometer` (Barometric pressure for altitude)
- `/sensor_node/gps_data` (Global Positioning System data)

These topics are the **inputs** to the EKF pipeline, providing essential measurements for state estimation.

2. Sensor Data Processing

The `/magnetometer_processor` and `/barometer_to_altitude` nodes preprocess raw Arduino data (converting magnetic field readings to heading, barometric pressure to altitude). This step ensures the data is in a usable format for the EKF.

3. EKF Fusion Core

The `/ekf_filter` node subscribes to preprocessed sensor data (e.g., IMU, magnetometer, barometer, GPS) and fuses them using an Extended Kalman Filter. It outputs a filtered odometry topic (`/odometry/filtered`), which represents the robot's estimated state with reduced noise and higher accuracy.

4. Final Output

The filtered odometry is published at `/odometry/filtered`, which will be used by other nodes (e.g., path planners, SMC controllers) for navigation.

Key Insight:

The Arduino acts as the **hardware interface**, publishing low-level sensor data. The heavy lifting (sensor fusion, state estimation) is offloaded to ROS nodes running on a more powerful companion computer (e.g., Raspberry Pi). This is a common design in robotics to leverage the real-time capabilities of Arduino for sensor reading and the computational power of ROS for advanced algorithms.

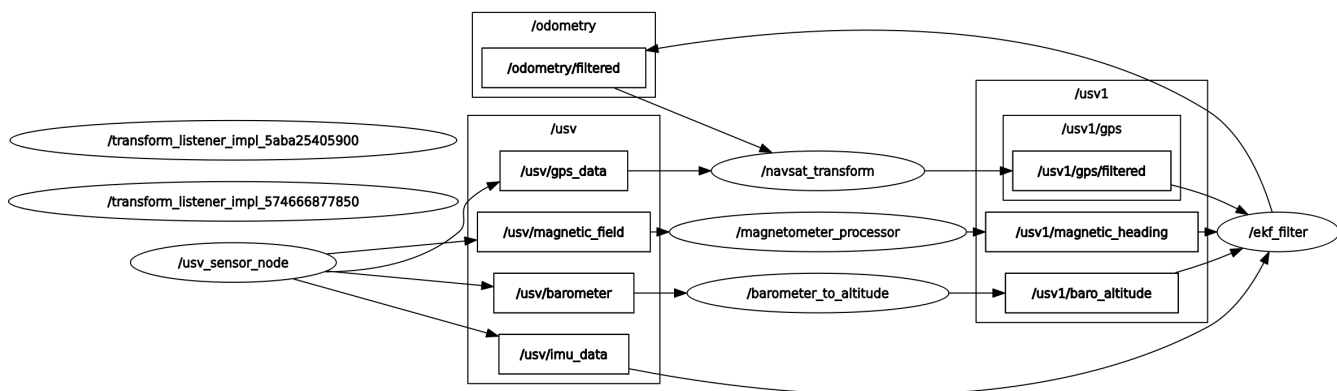


Figure 2.7: ROS Node Diagram for the EKF implementation on the Arduino data

2.4 Launch and Configuration

2.4.1 ROS 2 Launch Files

Launch files orchestrate node execution:

- **Base Launch:** Starts sensor, EKF, controller, and thruster nodes.
- **Mission-Specific Launch:** Loads predefined paths.

2.4.2 ROS2 Nodes

`sensor_node`

- **Purpose:** Bridges Arduino sensor data to ROS 2 topics
- **Hardware Interface:**
 - GY -87 : mpu6050 hmc5883l bmp180 (10-axis motion tracking)
 - GPS: NEO-M8N (GNSS receiver)
- **Topics:**
 - `/imu_raw` (`sensor_msgs/Imu`):
 - * Acceleration: $\pm 8g$ range
 - * Gyro: $\pm 2000^\circ/s$ range
 - * Sampling: 100 Hz
 - `/gps_fix` (`sensor_msgs/NavSatFix`):
 - * Accuracy: 1.5m CEP
 - * Update rate: 10 Hz
 - * Fix type: 3D
- **Implementation:**
 - Uses `rosserial_arduino` package
 - 50 Hz update rate
 - Sensor calibration at startup

`barometer_to_altitude`

- **Purpose:** Converts atmospheric pressure to altitude
- **Core Equation:**

$$h = \frac{T_0}{L} \left[\left(\frac{P}{P_0} \right)^{-\frac{RL}{s}} - 1 \right] \quad (2.3)$$

- **Parameters:**
 - P_0 : Sea-level pressure (101.325 kPa)
 - T_0 : Sea-level temperature (288.15 K)
- **Output:** `/sensor/altimeter` (altitude estimate)

`magnetometer_processor`

- **Purpose:** Computes magnetic heading from raw magnetometer data
- **Core Equation:**

$$\psi = \tan^{-1} \left(\frac{M_y}{M_x} \right) - \delta_{\text{declination}} \quad (2.4)$$

- **Features:**
 - Hard/soft iron compensation
 - Magnetic declination correction
 - Low-pass filtering
- **Output:** /sensor/magnetometer (yaw angle)

`navsat_transform_node`

- **Purpose:** Converts GPS geodetic coordinates to local odometry frame
- **Key Operations:**
 - Lat/Lon $\rightarrow$ UTM $\rightarrow$ odom frame conversion
 - Magnetic declination correction: 1.16 rad
 - Altitude zeroing at origin
- **Configuration:**
 - Frequency: 30 Hz
 - Base frame: `base_link`
 - World frame: `odom`
 - Datum: (0.0,0.0,0.0)

`ekf_node`

- **Purpose:** Multi-sensor state estimation using Extended Kalman Filter
- **Sensor Fusion:**
 - **odom0:** Filtered GPS (position/velocity)
 - **imu0:** IMU data (orientation/acceleration)
 - **odom1:** Barometric altitude
 - **imu1:** Magnetic heading
- **Configuration:**
 - State dimension: 15-element vector
 - Custom sensor covariance matrices
 - Process noise covariance: Tuned diagonal matrix
 - Frame IDs: `map`, `odom`, `base_link`
- **Output:** /odometry/filtered (6DOF pose with covariance)

thrust_converter

- **Purpose:** Maps continuous thrust to discrete motor states
- **Mapping:**

$$F \rightarrow \begin{cases} \text{low} & |F| < 30 \text{ N} \\ \text{medium} & 30 \leq |F| < 70 \text{ N} \\ \text{high} & |F| \geq 70 \text{ N} \end{cases} \quad (2.5)$$

- **Outputs:**
 - /left_thrust_cmd: Discrete left motor command
 - /right_thrust_cmd: Discrete right motor command
- **Implementation:** Threshold-based state machine

smc_control

- **Purpose:** Sliding Mode Controller for WAM-V path following
- **Core:** $s = \psi_e + \lambda_{ct} d_{ct}$, $r_{cmd} = -k_\psi \psi_e - u_{sw}$, $F = k_{speed}(u_d - u)$
- **Features:** Lookahead targeting, thruster mapping, rate limiting
- **Params:** Physical specs, $k_\psi=8$, $k_s=10$, max thrust=200N
- **Topics:** Inputs: odom, path; Outputs: thrust commands; Viz: target, path

planner_node

- **Mode 1 (Random Waypoints):** Samples waypoints within geofenced boundaries using a uniform distribution. Rejects points within 3m of obstacles.
- **Mode 2 (Predefined Path):** Loads waypoints from a CSV file and interpolates using cubic splines.
- **Topics:** Publishes /desired_path (nav_msgs/Path).

2.4.3 Raspberry Pi 5 Setup

ROS 2 Jazzy is installed on Raspberry Pi OS (64-bit). Key configurations:

- Real-time kernel patches for deterministic control loop timing.
- USB-C power delivery to prevent undervoltage during thruster operation.
- Dedicated /dev/ttyACM0 udev rule for consistent Arduino port naming.

Following the successful validation of our sensor data acquisition via the Arduino interface, which confirmed the publication of readable ROS topics, we now transition to the simulation phase. This stage allows us to develop and test the sensor integration and processing algorithms purely in a software environment. The insights gained from simulation will directly inform and streamline the subsequent re-implementation on the physical hardware.

2.5 WAM-V Simulation Framework

2.5.1 Project Directory Structure

The developed system is organised into two main components: gazebo_maritime for simulation assets and Gazebo plugins, and ros2_maritime for ROS 2 nodes, configuration, and launch files. Table and the tree below summarise the repository layout.

Project Root	
gazebo_maritime/	# Gazebo simulation assets and plugins
CMakeLists.txt	# Build configuration for Gazebo plugins
models/	# 3D models and assets
coast_waves/	# Ocean wave visual & physical model
materials/	# Shaders and textures for wave rendering
programs/	# GLSL shader programs
GerstnerWaves_fs_330.glsl	
GerstnerWaves_vs_330.glsl	
textures/	# Texture maps
skybox_lowres.dds	
wave_normals.dds	
meshes/	# 3D mesh files
waterlow.dae	
model.config	# Gazebo model metadata
model.sdf	# Gazebo model description
wam-v/	# WAM-V USV model
meshes/	# Hull, engine, propeller meshes
engine.dae	
propeller.dae	
WAM-V-Base.dae	
WamV.png	
model.config	
model.sdf	
src/	# Gazebo plugin source code
PublisherPlugin.cc/.hh	
Surface.cc/.hh	
Wavefield.cc/.hh	
WaveVisual.cc/.hh	
worlds/	# Gazebo world definitions
sydney_regatta.sdf	
ros2_maritime/	
config/	# Configuration files
bridge_config.yaml	# ROSGazebo bridge mappings
ekf_dual_filter.yaml	# Dual EKF configuration
ekf_filter.yaml	# Single EKF configuration
graph2.xml	# PlotJuggler layout
navsat.yaml	# navsat_transform_node parameters
launch/	# Launch files
display.launch.py	# Main simulation + visualization launch

```

package.xml                # ROS 2 package manifest
resource/ros2_maritime     # Package resource marker
ros2_maritime/             # Python nodes
  altimeter_converter.py
  gps_cov.py               # raw to noised GPS
  heading.py
  imu_covariance.py        # raw to noised IMU
  __init__.py
  magnetometer_converter.py # raw to noised Magnetometer
  path_publisher.py        # Path Publisher
  random_waypoint.py
  smc_control.py
  usv_breadcrumb.py        #An open loop Launching
rviz/                      # RViz visualization configs
  conf.rviz
setup.cfg                  # Python packaging config
setup.py                  # Python package setup script
test/                     # Unit and style tests
  test_copyright.py
  test_flake8.py
  test_pep257.py

```

The gazebo_maritime package contains all simulationrelated assets, including 3D models, textures, shaders, and Gazebo plugins for wave simulation.

The ros2_maritime package contains the ROS 2 nodes for sensor data processing, state estimation, control, and visualization, along with configuration and launch files.

2.5.2 Simulation Framework

This project implements a simulation of the **WAM-V autonomous surface vehicle** using Gazebo and ROS 2. The simulation relies on **SDF (Simulation Description Format)** files to describe the robots physical structure, sensors, environment, and dynamics.

The purpose of the simulation is to create a realistic virtual environment for testing navigation, localization, and control algorithms for autonomous boats.



Figure 2.8: The real world environment used for the simulation

The implementation of the simulation environment and the robot model is illustrated in the following figures. Figure 2.9 presents the visual rendering of the WAM-V platform operating within the Sydney Regatta world, demonstrating the interaction between the vehicle and the fluid dynamics of the virtual lake. The structural composition of the robot is detailed in the entity tree shown in Figure 2.10, which enumerates the kinematic chain, including the chassis, engine links, and propeller joints defined by the SDF. Furthermore, Figure 2.11 provides an expanded view of the base_link hierarchy, confirming the integration of the sensor suite essential for navigation, specifically the IMU, magnetometer, altimeter, and NavSat (GPS) sensors.

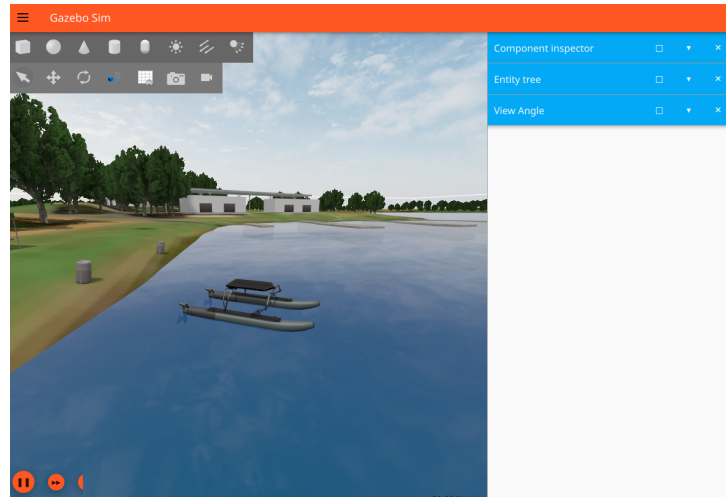


Figure 2.9: The gazebo simulation for the USV



Figure 2.10: Entities used in the Gazebo simulation environment

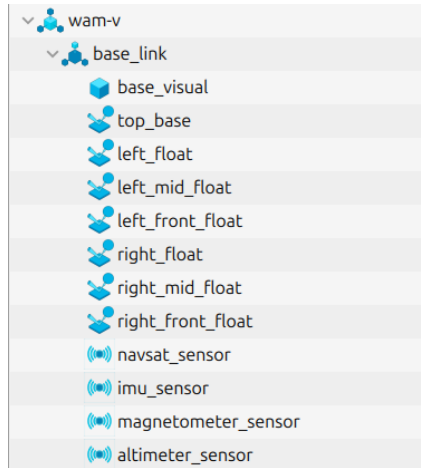


Figure 2.11: Entities attached to the Wam-V USV in Gazebo

2.5.3 SDF Model Structure

Main Model: `wam-v.sdf`

The primary model file describes the **hull and base structure** of the WAM-V. Key elements include:

- **Links and Joints:** Includes `top_base`, `hull`, and other structural parts. Propeller joints (`left_engine_propeller_joint`, `right_engine_propeller_joint`) enable thrust control.
- **Inertial Properties:** Each link defines mass, inertia tensor, and collision shapes. This allows Gazebo to simulate buoyancy, collisions, and dynamics.
- **Visuals:** Meshes (e.g., `WAM-V-Base.dae`) are included for realistic rendering.
- **Hydrodynamics Plugin:** It uses hydrodynamic coefficients to model forces from the surrounding water, ensuring that the boat responds realistically to thrust, drag, and fluid resistance.

Environment: `sydney_regatta.sdf`

The simulation takes place in the **Sydney Regatta environment**, which contains:

- A water plane with wavefield effects.
- Starting pose for the WAM-V model (`<pose>-532 162 0 0 0 1</pose>`).
- Visual elements such as buoys, docks, and shoreline.

This environment provides realistic conditions for navigation, with water interaction and surface dynamics.

2.5.4 Sensors added to the Simulation

The WAM-V model is equipped with several virtual sensors, integrated through SDF and bridged into ROS 2 topics.

GPS (Global Positioning System)

Provides global position (x, y) with respect to the map frame. It serves as the main absolute position reference.

Altimeter

Reports the vertical position (z). Acts as the primary source of altitude measurement, keeping the boat stable with respect to water surface.

IMU (Inertial Measurement Unit)

Measures linear acceleration and angular velocity. Provides roll and pitch orientation, as well as angular velocity for velocity estimation.

Magnetometer

Provides absolute yaw heading, stabilizing orientation in the horizontal plane.

Each sensor publishes to ROS 2 topics (e.g., `/gps`, `/imu/data`, `/Altimeter`) and is processed by the `robot_localization` package using a dual EKF setup.

2.5.5 Hydrodynamic Coefficients

The SDF defines hydrodynamic coefficients inside the hydrodynamics plugin. These parameters govern how the boat interacts with the fluid.

Linear Drag

- Coefficients: xU , yV , zW .
- Resistive forces proportional to velocity.
- Example: xU applies damping along surge (forward motion), while yV resists sway (side-ways drift).

Added Mass

- Coefficients: $x\dot{U}$, $y\dot{V}$, $n\dot{R}$.
- Represents the effect of displaced fluid adding inertia.
- Example: $x\dot{U}$ increases inertia in surge, $y\dot{V}$ in sway, $n\dot{R}$ in yaw.

Rotational Damping

- Coefficients: kP , mQ , nR .
- Resists angular motion (roll, pitch, yaw).
- Example: nR resists yaw rate, simulating stabilizing effects.

Absolute Damping

- Coefficients: x_{AbsU} , y_{AbsV} , n_{AbsR} .
- Constant forces applied even at zero velocity.
- Useful for modeling baseline drag or residual damping.

Together, these coefficients ensure the WAM-V behaves realistically in water, slowing down naturally when propulsion stops, resisting quick turns, and maintaining stable motion.

2.5.6 Simulation Workflow

The simulation proceeds as follows:

1. **Environment Launch:** The `sydney_regatta.sdf` world is loaded in Gazebo.
2. **Model Spawning:** The `wam-v.sdf` boat is spawned with hydrodynamics and sensors activated.
3. **ROS 2 Bridge:** Sensor data (GPS, IMU, magnetometer, altimeter) are published to ROS 2 topics.
4. **Localization:** The `robot_localization` dual EKF nodes fuse sensor data into filtered odometry in both `odom` and `map` frames.
5. **Control:** Thruster commands are published to joint controllers, propelling the boat.

The simulation integrates a physically accurate hydrodynamic model of the WAM-V, realistic environmental conditions, and multiple onboard sensors. The SDF files serve as the backbone, defining both physical structure and sensor behavior.

This configuration provides a powerful framework for testing autonomous navigation algorithms in a controlled but realistic setting.

2.6 Software Configuration and Validation Tools

The simulation environment is supported by a set of ROS 2 nodes, configuration files, and visualization tools. These elements allow both the execution of the navigation stack and the verification of the performance of the state estimation and control algorithms.

The data flow for the navigation stack is shown in Figure 2.12. Raw sensor data from Gazebo enters the graph through the `ros_gz_bridge`. Before state estimation can occur, intermediate nodes like `imu_covariance` and `navsat_transform` preprocess these signals adding covariance data and performing geodetic conversions. The processed topics are then linked to the dual-EKF architecture, resulting in the fused output `/odometry/filtered` required for local and global navigation.

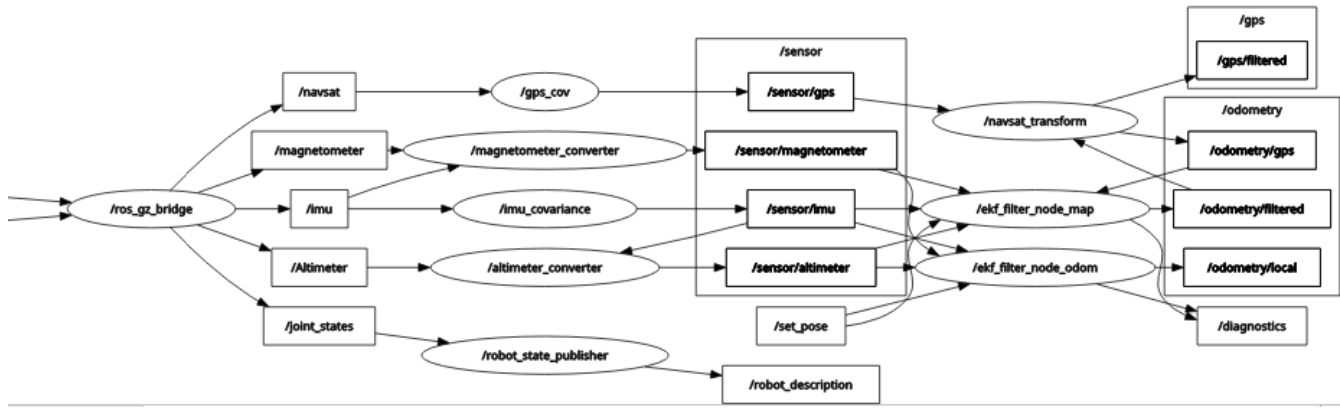


Figure 2.12: Topics and nodes necessary for the EKF node in rqt_graph

2.6.1 Configuration Files

Bridge Configuration (bridge_config.yaml)

This YAML file defines the ROS 2 to Gazebo transport bridge. Its main purpose is to map Gazebo topics to ROS 2 topics so that simulated sensor data can be processed by the navigation stack.

- **GPS:** `/wamv/sensors/gps → /navsat`.
- **IMU:** `/wamv/sensors/imu → /imu/data`.
- **Magnetometer:** `/wamv/sensors/magnetometer → /imu/mag`.
- **Altimeter:** `/wamv/sensors/altimeter → /altimeter/data`.

This bridging ensures that each simulated sensor behaves as a ROS 2 sensor driver would on a real robot.

Launch Files The launch system orchestrates the different components of the navigation pipeline. In particular, the launch files load:

- The **dual EKF nodes** from `robot_localization`,
- The **navsat_transform_node**,
- The **sensor converter nodes**,

- The **SMC controller node**,
- The **path publisher node**,
- The visualization tools (rviz2).

This modular structure ensures flexibility: each component can be swapped, tuned, or disabled independently.

Overview

The launch file orchestrates a comprehensive simulation environment for a WAM-V autonomous surface vehicle, integrating sensor processing, state estimation, and visualization components.

Key Components

Simulation Environment The launch file initializes a Gazebo simulation with a specific maritime environment:

```
1 ExecuteProcess (
2     cmd=["gz", "sim", gazebo_world_path],
3     output="screen"
4 )
```

where gazebo_world_path points to the Sydney Regatta world file, providing a realistic maritime environment for testing.

Robot Model Processing The robot state publisher node processes the WAM-V model description:

```
1 Node (
2     package='robot_state_publisher',
3     executable='robot_state_publisher',
4     name='robot_state_publisher',
5     parameters=[
6         {'robot_description': open(urdf_path).read()},
7         {'use_sim_time': LaunchConfiguration('use_sim_time',
8             default='true')}
9     ]
10 )
```

Sensor Processing Pipeline Custom nodes process and convert raw sensor data:

- altimeter_converter - Processes depth/altitude data
- magnetometer_converter - Converts magnetic field measurements
- imu_covariance - Adds proper covariance to IMU data
- gps_cov - Adds covariance estimation to GPS data
- path_publisher - Publishes navigation paths

Transform Configuration Multiple static transform publishers define the spatial relationships between robot components:

```

1 Node(
2     package='tf2_ros',
3     executable='static_transform_publisher',
4     arguments=['0', '0', '0', '0', '0', '0', 'base_link', 'wam-v/
      base_link/imu_sensor']
5 )

```

These transforms connect sensors (IMU, GPS, altimeter) and actuators to the robot's base link. State Estimation A dual EKF configuration provides robust state estimation:

```

1 Node(
2     package='robot_localization',
3     executable='ekf_node',
4     name='ekf_filter_node_odom',
5     parameters=['/home/pc/gazebo_maritime_ws/src/ros2_maritime/
      config/ekf_dual_filter.yaml'],
6     remappings=[('/odometry/filtered', '/odometry/local')],
7     output='screen'
8 )

```

The navigation satellite transform node integrates global positioning data:

```

1 Node(
2     package='robot_localization',
3     executable='navsat_transform_node',
4     name='navsat_transform',
5     parameters=['/home/pc/gazebo_maritime_ws/src/ros2_maritime/
      config/ekf_dual_filter.yaml'],
6     remappings=[
7         ('/imu', '/sensor/imu'),
8         ('/gps/fix', '/sensor/gps'),
9         ('/odometry/filtered', '/odometry/filtered')
10    ],
11     output='screen'
12 )

```

Visualization and Tools The launch file includes visualization components:

```

1 Node(
2     package="rviz2",
3     executable="rviz2",
4     name="rviz2",
5     output="screen",
6     arguments=['-d', '/home/pc/gazebo_maritime_ws/src/
      ros2_maritime/rviz/conf.rviz'],
7     parameters=[{"use_sim_time": True}]
8 )

```

ROS-Gazebo Bridge A parameter bridge facilitates communication between ROS and Gazebo:

```

1 Node(
2     package='ros_gz_bridge',
3     executable='parameter_bridge',
4     name='ros_gz_bridge',

```

```
5     output='screen',  
6     parameters=[{'config_file': bridge_config_path}]  
7 )
```

2.6.2 Sensor Modeling Nodes

To avoid unrealistically perfect measurements from the Gazebo sensors, additional nodes were implemented to introduce noise and biases, making the simulation closer to reality.

GPS Converter

Adds Gaussian noise and dropouts to the Gazebo GPS data, simulating low-precision satellite signals.

IMU Converter

Applies bias and drift to accelerometer and gyroscope measurements. This prevents the EKF from converging too quickly to idealized values.

Magnetometer Converter

Perturbs the heading measurements with a configurable noise covariance, simulating disturbances due to local magnetic fields.

Altimeter Converter

Adds noise and small offsets to the vertical (z) position, emulating sensor fluctuations over water.

These converters are configured through YAML parameters, allowing the level of realism to be tuned.

2.6.3 State Estimation: Dual EKF Setup

The `robot_localization` package is configured with two Extended Kalman Filters:

- **EKF Local (odom frame):** Fuses high-frequency IMU data with wheel odometry (or simulated velocity) to estimate smooth, drift-prone odometry.
- **EKF Global (map frame):** Fuses GPS, magnetometer, and altimeter data to provide a globally consistent state estimate. The `navsat_transform_node` is used to align GPS data with the robot's local odometry, enabling seamless integration between the two EKFs.

Together, the dual EKF framework ensures both short-term smoothness and long-term global consistency.

2.6.4 Control Node: Sliding Mode Controller (SMC)

The control of the WAM-V is achieved through a **Sliding Mode Controller (SMC)** node. This nonlinear controller is designed to handle disturbances (waves, currents, model uncertainties) while ensuring trajectory tracking.

The SMC receives the filtered state estimate (from the EKF) and computes the necessary thruster commands, which are then applied to the propeller joints in Gazebo.

2.6.5 Path Publishing Node and Visualization

The path publisher node provides predefined waypoints or trajectories for the WAM-V to follow. It publishes a sequence of target poses to be tracked by the SMC. This separation between trajectory generation and control allows testing of different navigation strategies.

To support path planning and visualization for autonomous navigation, a custom ROS 2 node was developed to publish both raw waypoints and a smoothed reference path. This node enables visual inspection of the difference between the original discrete waypoints and the continuous trajectory used for control and planning.

Node Functionality

The `path_publisher` node is implemented in Python using the ROS 2 `rclpy` client library. It publishes two distinct `nav_msgs/Path` messages:

- `/waypoints_path` a path composed of the original set of discrete 2D waypoints, connected by straight lines.
- `/reference_path` a smoothed path generated using cubic B-spline interpolation.

Both paths are published using the `odom` coordinate frame (`frame_id`) to ensure consistent visualization in RViz2. The node uses a timer callback to publish paths at 1 Hz, allowing for live updates during simulation or real-time operation.

B-Spline Smoothing Algorithm

To generate a smooth trajectory from sparse waypoints, a closed cubic B-spline is fitted using SciPy's `splprep` and `splev` functions. The input waypoints are given as 2D coordinates (x, y) , and the `splprep` function is used to parameterize the curve with a low smoothing factor ($s = 0.5$). The spline is defined as periodic (`per=True`) to form a closed loop.

After fitting, the spline is sampled at 100 evenly spaced points along its parameterization using `splev`, resulting in a continuous and smooth path.

This method offers several advantages:

- Ensures smooth curvature, critical for vehicle dynamics and control.
- Reduces abrupt transitions in heading or curvature.
- Allows adjustable smoothness via the smoothing parameter s .

Visualization in RViz2

Both the raw and smoothed paths are visualized using the Path display in RViz2. This provides a clear, side-by-side comparison between the original waypoints and the smoothed trajectory. An example visualization is shown in Figure 2.13, where the spline closely follows the waypoint path while eliminating sharp transitions.

The example set of waypoints used in this implementation forms a closed-loop path:

$$\begin{aligned} (0.0, 0.0), \quad (20.0, 20.0), \quad (40.0, 20.0), \quad (60.0, 0.0), \\ (40.0, -20.0), \quad (20.0, -20.0), \quad (0.0, 0.0) \end{aligned}$$

This defines a symmetric "diamond" or "loop" pattern that returns to the origin, containing both straight segments and 90-degree directional changes. Such a path is useful for visualizing closed-loop performance and cornering behavior in autonomous motion.

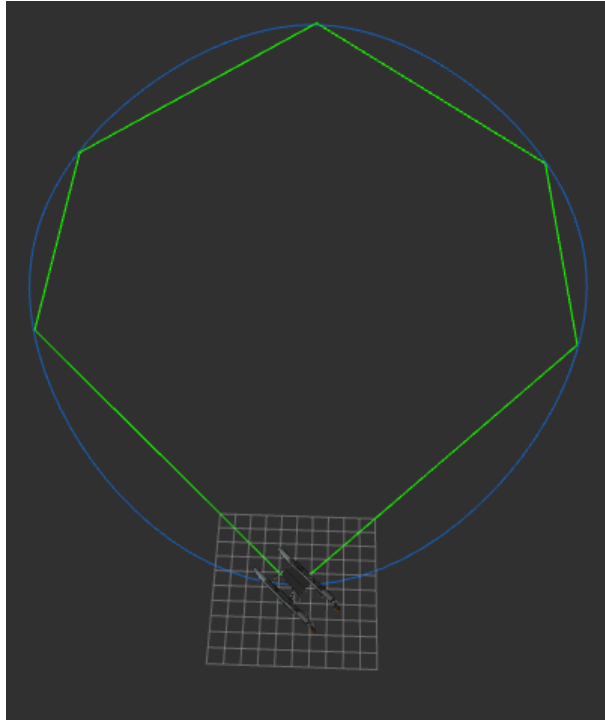


Figure 2.13: Visualization of the raw waypoint path (straight lines) and the smoothed reference path (B-spline) in RViz2.

2.6.6 Path Design for Bathymetric Surveys

In autonomous surface vehicle (ASV/USV) applications, such as bathymetric surveys of dam reservoirs, the path planning strategy must be designed to ensure full and uniform coverage of the surveyed area. The most commonly adopted pattern in such cases is the **lawnmower path**, also known as a *zig-zag* or *boustrophedon* pattern.

Lawnmower Path Structure. This path consists of a sequence of parallel passes connected by sharp turns at either end. The vehicle follows a straight segment along one direction, shifts laterally by a fixed swath width, and returns in the opposite direction. This continues until

the entire survey region is covered. The method is widely used in both underwater and aerial mapping due to its simplicity, predictability, and efficient spatial coverage.

An example of such a path is shown in Figure 2.14, where 12 waypoints define a coverage area in a rectangular $100\text{ m} \times 100\text{ m}$ region. The spacing between segments (20 m in this case) can be tuned based on the bathymetric sonar's swath width.

Initial Smoothing Attempt. In early tests, a cubic B-spline interpolation was applied to the waypoints to produce a smooth, continuous path. While B-splines are effective for smoothing trajectories, they do not guarantee that the path passes through the original waypoints. As a result, the B-spline path deviated significantly from the sharp cornering structure of the lawnmower pattern. This caused problems for control algorithms that assume adherence to survey lines, and introduced the risk of missing survey coverage at the edges.

Chosen Method: Catmull-Rom Spline. To overcome this, a Catmull-Rom spline interpolation method was implemented. Unlike cubic B-splines, Catmull-Rom splines pass *exactly* through the waypoints, preserving the planned survey pattern. At the same time, they introduce smooth curvature at transitions, eliminating sharp discontinuities that could destabilize vehicle controllers. The resulting path maintains the structural integrity of the original plan, while improving tracking smoothness and controller performance.

This method balances **geometric accuracy** and **trajectory smoothness**, making it ideal for USV-based bathymetric mapping in confined environments such as dam reservoirs.

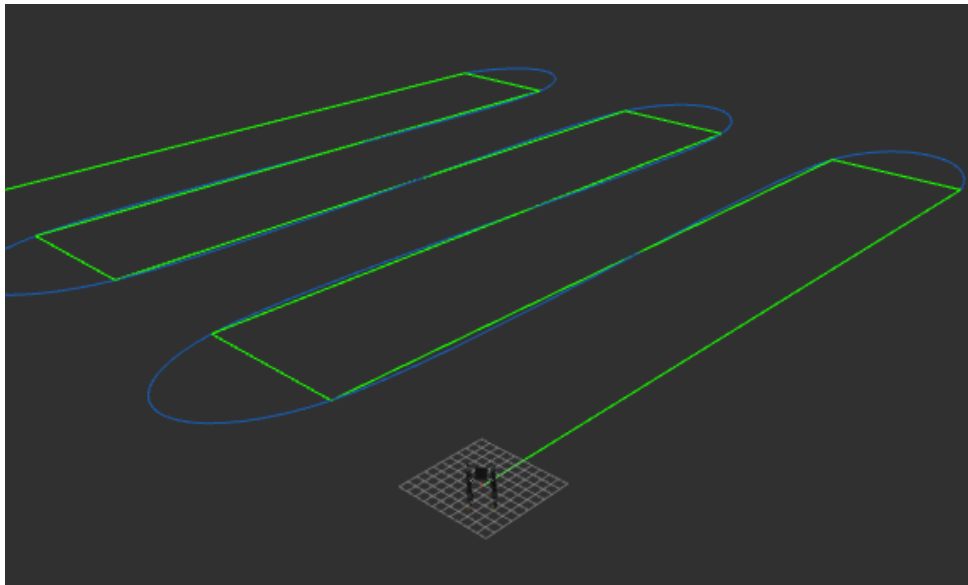


Figure 2.14: Visualization of the lawnmower path in RViz2: the original waypoint path (straight lines) and the smoothed Catmull-Rom spline reference path.

Summary

The path publishing node serves as a foundational component for trajectory visualization and debugging in autonomous navigation. By publishing both raw and smoothed paths, it aids in the development, testing, and evaluation of path-following algorithms.

2.6.7 Validation Tools

PlotJuggler

The **PlotJuggler** tool is used extensively to analyze log data from the simulation. Custom configuration files were created to automatically load topics of interest, such as:

- Filtered odometry from EKF (local and global).
- Raw vs. converted sensor data (GPS, IMU, magnetometer, altimeter).
- Control inputs generated by the SMC node.
- Tracking error compared to the published path.

By comparing raw and filtered signals, PlotJuggler provides direct insight into the effectiveness of the EKF and the performance of the SMC controller.

RViz2 Visualization

RViz2 is used to visualize the robot state in real time:

- The WAM-V model in the simulated environment.
- Filtered odometry (/odometry/filtered) as a trajectory path.
- GPS fixes as global reference points.
- The desired path from the path publisher.
- Control vectors (optional markers showing thrust direction).

The RViz configuration incorporates several critical visualization elements:

- **TF (Transform Frames):** Displays the coordinate frame hierarchy of the robotic system, showing the relationships between the world map, odometry frame, base link, and sensor frames. This is essential for verifying proper transform tree configuration and debugging coordinate frame issues.
- **RobotModel:** Visualizes the WAM-V vessel's URDF model with accurate joint positions and orientations. This provides a real-time view of the robot's pose within the simulated environment and helps verify the physical representation matches expected behavior.
- **Path:** Displays the trajectory of the vessel over time, typically subscribing to nav_msgs/Path messages. This visualization is crucial for analyzing navigation performance, path planning effectiveness, and tracking accuracy.
- **PoseArray:** Visualizes multiple pose hypotheses, often used for displaying particle filter distributions in localization algorithms or potential future trajectories in motion planning. This provides insight into the uncertainty and multiple hypotheses considered by the estimation and planning stacks.

These visualization tools collectively provide a comprehensive monitoring solution for debugging and validating the performance of the maritime autonomous system during simulation experiments.

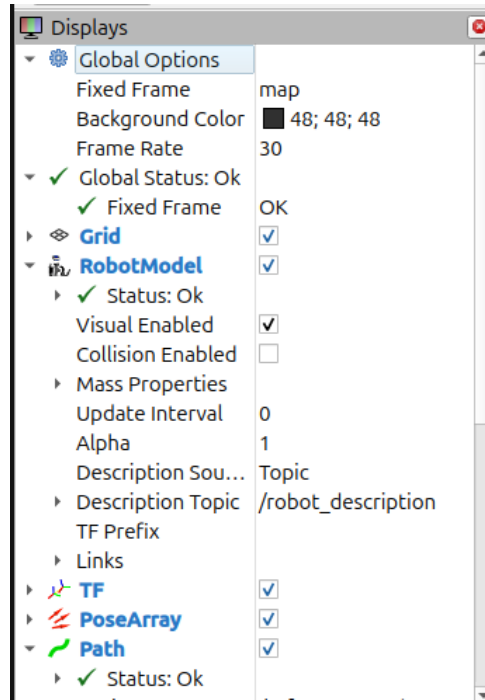


Figure 2.15: configuration and Parameters used in RVIZ2

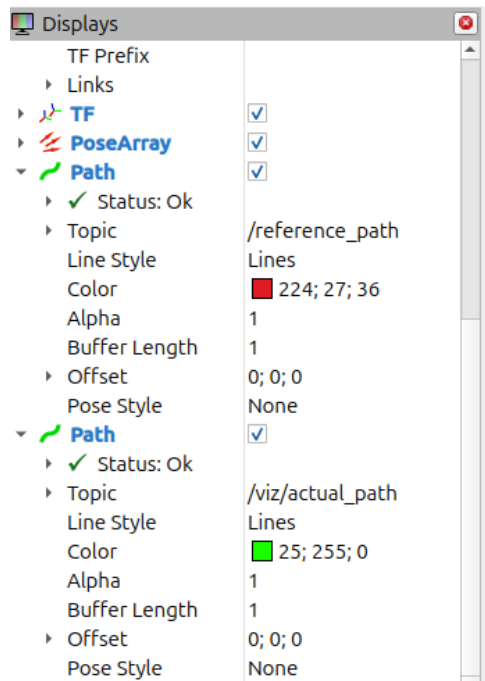


Figure 2.16: Path and Pose parameters used in RVIZ2

Figure 2.15 and 2.16 illustrate the RViz visualization configuration used for qualitative validation of the navigation and control framework. The global reference frame is set to map, ensuring consistent visualization of all spatial data through a valid transformation tree. The robot model, loaded from the /robot_description topic, provides a visual representation of the USV within the environment, while the TF display allows verification of frame consistency. Two distinct path displays are used to compare the planned and executed trajectories:

the reference path, published on the `/reference_path` topic and displayed in red, represents the desired trajectory generated by the guidance module, whereas the actual path, published on `/viz/actual_path` and shown in green, corresponds to the estimated motion of the vehicle. This configuration enables an intuitive assessment of path-following accuracy and system performance, complementing quantitative validation tools such as PlotJuggler.

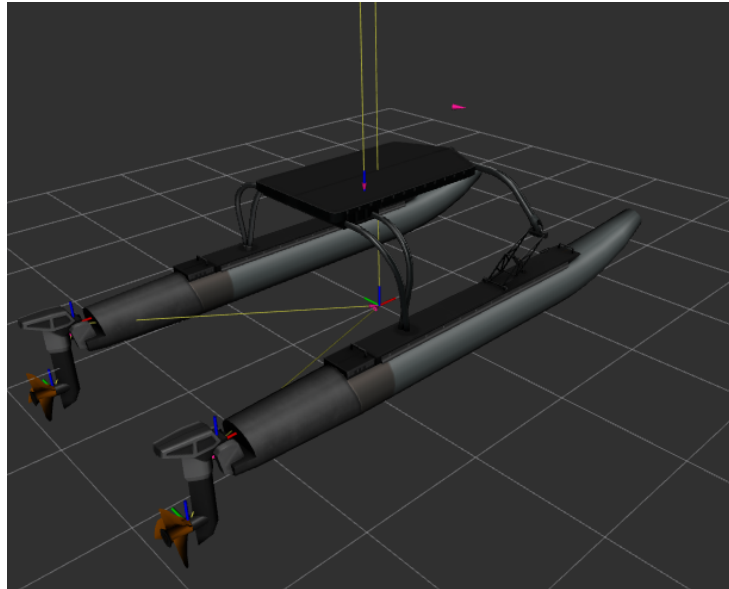


Figure 2.17: The representation of the USV on RVIZ2

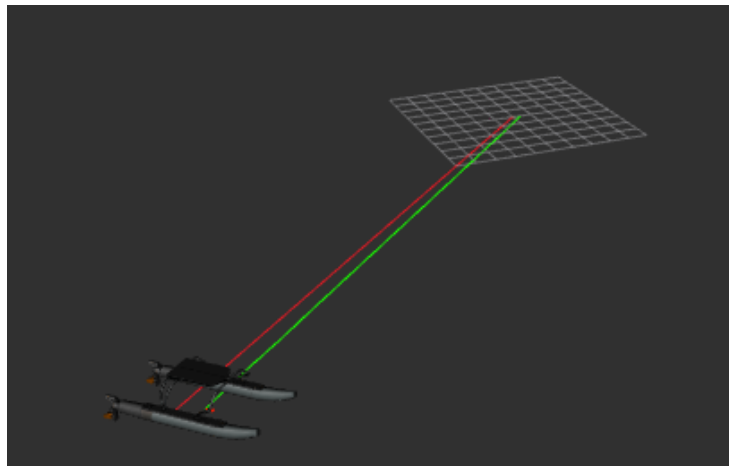


Figure 2.18: The visualisation of the USV on rviz2 while following the a straight line with an open loop command

The results of this RViz configuration are shown in Figure 2.17 and 2.18, where the reference path and the actual executed path can be visually compared.

This allows verification that the estimated state matches the actual simulated state and that the SMC drives the robot along the planned trajectory.

2.7 Transformation Frame Architecture in Dual EKF Localization

2.7.1 Introduction to TF Frame Structure

The transformation frame architecture forms the fundamental backbone of the dual Extended Kalman Filter (EKF) localization system implemented for maritime autonomous navigation. This section details the hierarchical frame structure, its mathematical representation, and the critical role it plays in integrating multiple sensor modalities for robust pose estimation.

2.7.2 Hierarchical Frame Architecture

The system employs a three-layer hierarchical transformation frame structure:

$$\text{map} \rightarrow \text{odom} \rightarrow \text{base_link} \quad (2.6)$$

This structure enables the separation of global reference frames from local odometry, which is crucial for handling GPS drift correction while maintaining smooth local motion estimates in challenging maritime environments.

Frame Definitions and Properties

Table 2.2: Transformation Frame Properties and Functions

Frame	Reference Type	Primary Function
map	Global, world-fixed	Provides absolute positioning in Earth coordinates, tied to UTM coordinate system
odom	Local, drifting	Maintains locally consistent odometry, subject to drift over time
base_link	Body-fixed	Attached to vehicle's center, moves with the vessel

2.7.3 Mathematical Representation

The complete transform from global coordinates to body frame is computed through consecutive transformations:

$$T_{\text{base_link}}^{\text{map}} = T_{\text{odom}}^{\text{map}} \cdot T_{\text{base_link}}^{\text{odom}} \quad (2.7)$$

Where each transformation T_b^a represents a 4E4 homogeneous transformation matrix:

$$T_b^a = \begin{bmatrix} R_b^a & t_b^a \\ 0 & 1 \end{bmatrix} \quad (2.8)$$

with R_b^a being the 3E3 rotation matrix and t_b^a the 3E1 translation vector.

2.7.4 Dual EKF Frame Generation

Local EKF: odom $\rightarrow$ base_link

The `ekf_filter_node_odom` generates the transform from `odom` to `base_link` using high-frequency sensor data:

$$T_{\text{base_link}}^{\text{odom}} = f_{\text{odom}}(\text{IMU}, \text{Magnetometer}, \text{Altimeter}) \quad (2.9)$$

This transform provides:

- High-frequency updates (30 Hz)
- Smooth local motion estimation
- Short-term consistency for control systems
- Drift accumulation over extended periods

Global EKF: map $\rightarrow$ odom

The `ekf_filter_node_map` generates the transform from `map` to `odom` by fusing GPS data with other sensors:

$$T_{\text{odom}}^{\text{map}} = f_{\text{map}}(\text{GPS}, \text{IMU}, \text{Magnetometer}, \text{Altimeter}) \quad (2.10)$$

This transform provides:

- Global positioning correction
- Drift compensation for local odometry
- Absolute reference for path planning
- Lower update rate constrained by GPS availability

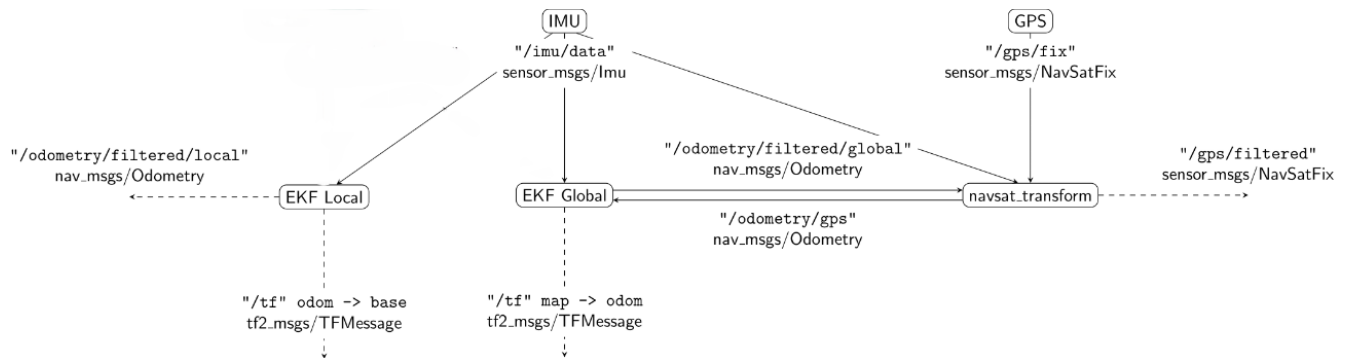


Figure 2.19: Data Flow for EKF-based Local (odom) and Global (map) odom Estimation

2.7.5 Frame Management in ROS 2

The transformation framework is implemented using ROS 2's TF2 library, which maintains the transform tree and provides efficient lookup capabilities:

```
1 # Frame configuration in EKF nodes
2 ekf_odom_node = Node(
3     parameters=[
4         {'map_frame': 'map',
5          'odom_frame': 'odom',
6          'base_link_frame': 'base_link',
7          'world_frame': 'odom'} # Local EKF outputs odom->
8         base_link
9     ]
10 )
11 ekf_map_node = Node(
12     parameters=[
13         {'map_frame': 'map',
14          'odom_frame': 'odom',
15          'base_link_frame': 'base_link',
16          'world_frame': 'map'} # Global EKF outputs map->odom
17     ]
18 )
```

Listing 2.3: TF frame management in ROS 2

2.7.6 System Visualization and Diagnostics

Transformation Frame Validation

The TF frame structure was validated using ROS 2's built-in diagnostic tools. Figure 2.20 shows the complete transformation tree generated by the `tf2_tools view_frames` command, confirming the proper hierarchical structure of the dual EKF system.

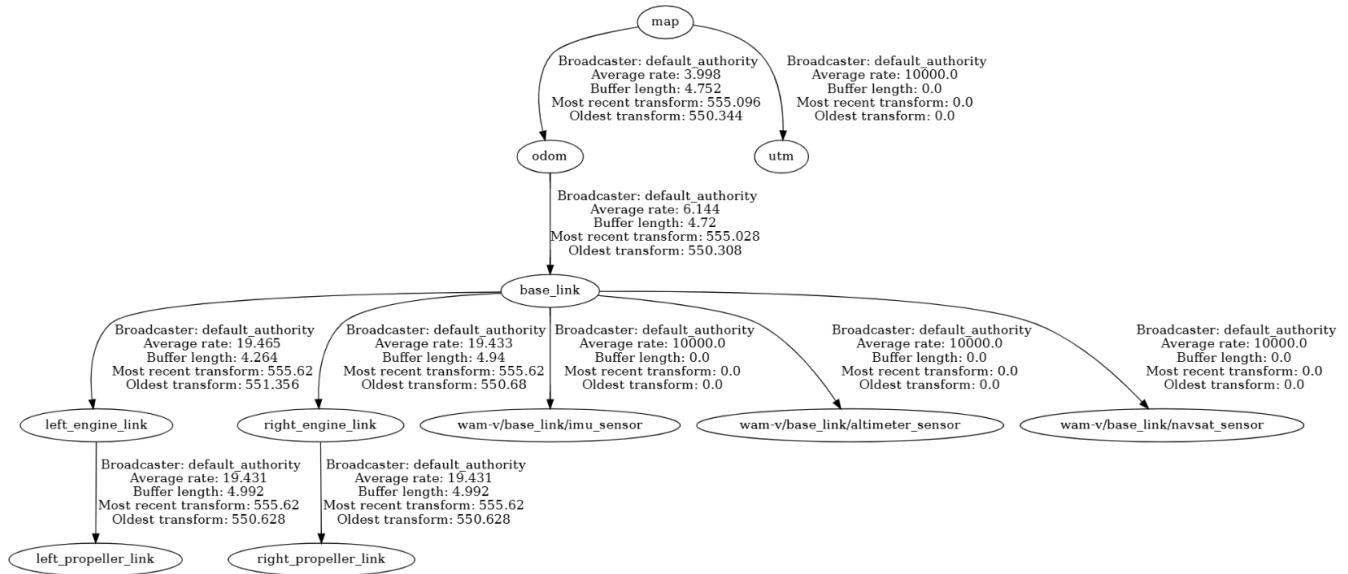


Figure 2.20: Transformation frame hierarchy generated by `ros2 run tf2_tools view_frames` command, showing the complete tree structure of the EKF localization system

The visualization confirms the proper implementation of the hierarchical structure:

- `map` $\rightarrow$ `odom` $\rightarrow$ `base_link` transformation chain
- Correct frame connections between EKF nodes
- Proper timestamp synchronization between frames
- Absence of conflicting or duplicate frame publications

2.7.7 Advantages for Maritime Applications

The hierarchical frame structure provides specific benefits for maritime navigation:

- **GPS Latency Handling:** Local odometry provides high-frequency updates between GPS corrections
- **Sensor Outage Robustness:** System continues operation using local frames during GPS outages
- **Multi-rate Sensor Integration:** Accommodates different sensor update rates seamlessly
- **Global-Local Separation:** Maintains both global accuracy and local consistency

The transformation frame architecture implemented in the dual EKF system provides a robust foundation for maritime localization. The hierarchical structure of `map` $\rightarrow$ `odom` $\rightarrow$ `base_link` effectively separates global reference frames from local odometry, enabling the system to handle the unique challenges of marine environments. This architecture supports reliable sensor fusion, provides necessary information for control systems, and maintains consistent pose estimation across varying environmental conditions.

2.7.8 Dual EKF State Estimation

Accurate state estimation is fundamental for feedback control. The `robot_localization` package was configured to run two EKFs:

- The **local (odom-based) EKF**, which fuses high-rate IMU data with simulated process dynamics. This provides smooth, drift-free odometry estimates suitable for short-term control.
- The **global (map-based) EKF**, which incorporates GPS and magnetometer measurements, correcting the drift from the local filter and ensuring global consistency in position and heading.

Sensor data was pre-processed through dedicated converter nodes:

- Altimeter $\rightarrow$ `PoseWithCovarianceStamped` for z position.
- GPS $\rightarrow$ NavSat messages, integrated via `navsat_transform_node`.
- IMU $\rightarrow$ orientation (roll, pitch) and angular velocity.
- Magnetometer $\rightarrow$ yaw orientation.

The EKF outputs the fused odometry estimate on the topic `/odometry/filtered`, which includes the vehicle's position (x, y, z) and orientation (quaternion $\rightarrow$ yaw).

2.7.9 EKF Validation with PlotJuggler

To verify the correct functioning of the EKF, the estimated states were visualized using **PlotJuggler**. The following signals were plotted:

- $x(t)$ and $y(t)$: estimated horizontal position of the boat, compared against the reference path.
- $\psi(t)$ (yaw): estimated heading from the filter, compared against the commanded trajectory orientation.
- $v_x(t)$ and $v_y(t)$: estimated linear velocities along the X and Y axes, respectively.

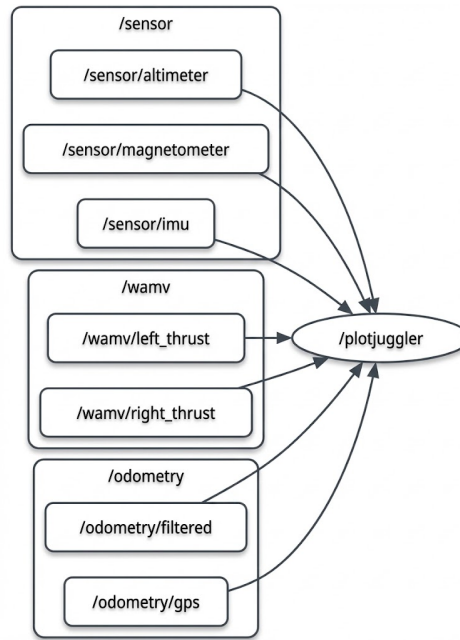


Figure 2.21: The topics subscribed by Plotjuggler for the EKF graphs visualisation

Visualizing the Boat on its start position

For the case of the robot in its starting position:

The filter performance was first evaluated with the robot in a stationary position. Figure 2.22 presents the yaw orientation over time (x-axis in seconds). The y-axis represents the heading angle in radians. The comparison highlights the filter's efficacy: the **blue line**, representing the raw magnetometer data (`/sensor/magnetometer`), exhibits significant high-frequency noise. In contrast, the **red line** shows the EKF estimate (`/odometry/filtered`), which successfully filters this noise to provide a stable heading reference.

Similarly, Figure 2.23 plots the X and Y positions (in meters) while static. The estimated position (shown in the blue and red curves) aligns tightly with the raw GPS measurements (green and orange curves), confirming that the filter correctly initializes at the start coordinates without introducing drift.

In Figure 2.24, the predicted velocities along the X and Y axes fluctuate around zero, indicating little to no movement.

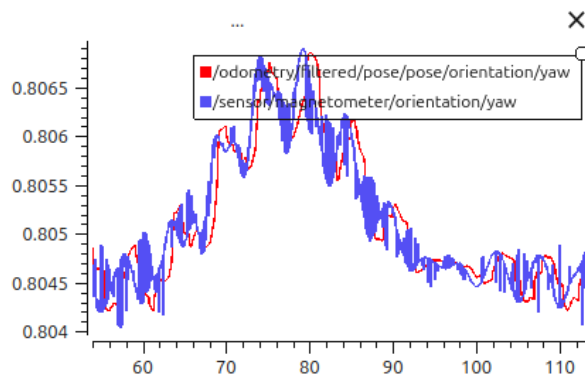


Figure 2.22: Observing the yaw orientation while the Robot is on its start position

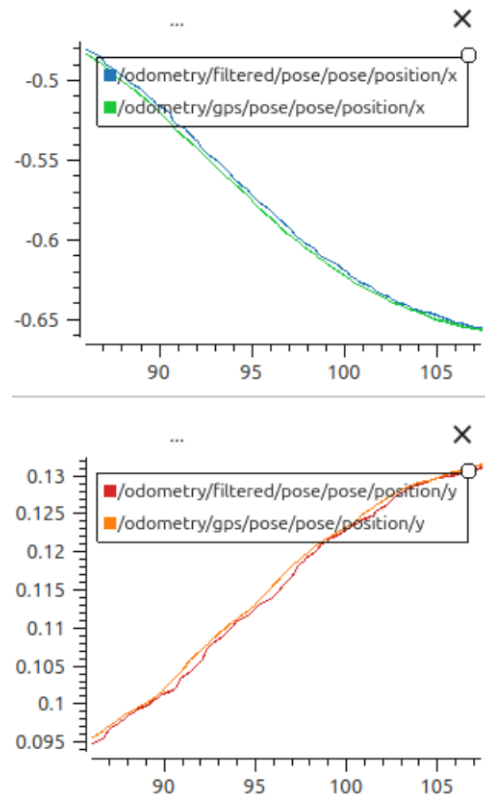


Figure 2.23: Observing the X, Y while the Robot is on its start position

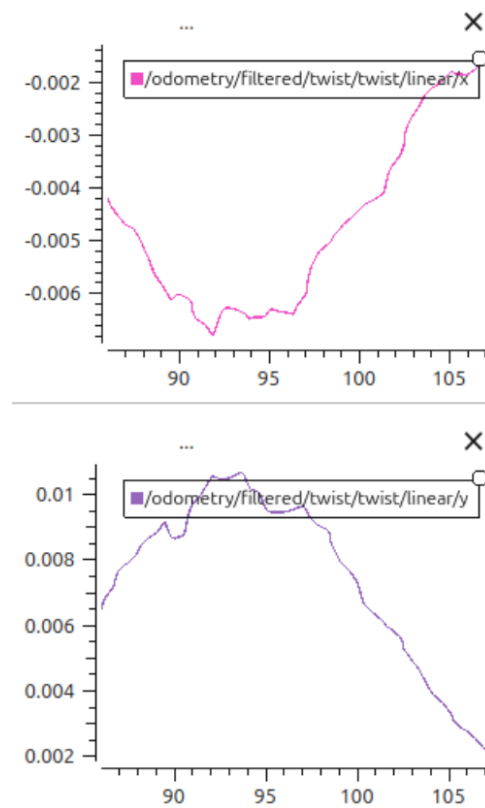


Figure 2.24: Observing the Vx and Vy calculated by the EKF while the Robot is on its start position

Visualizing the Boat while moving steadily

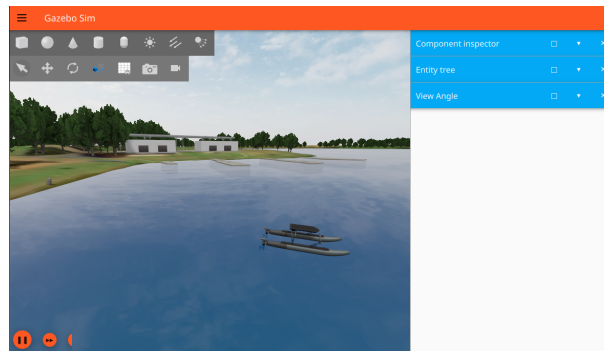


Figure 2.25: Launching the robot to position B (20, 20) on Gazebo

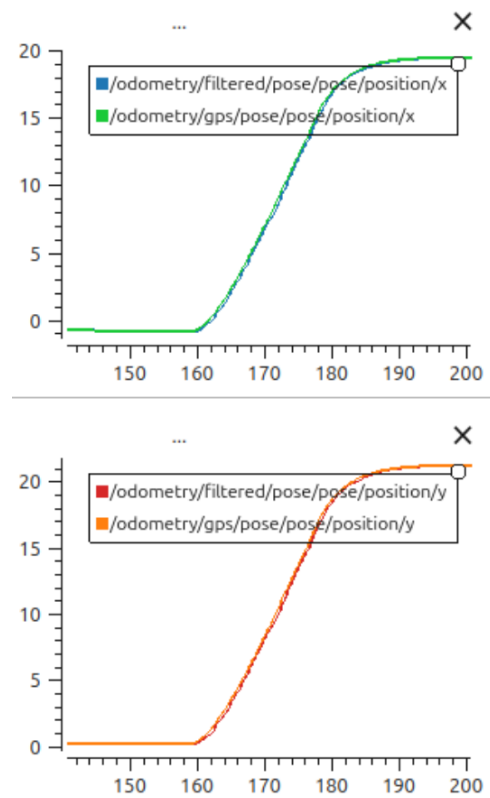


Figure 2.26: Observing the X, Y while the Robot is moving from position A(0,0) to position B(20,20)

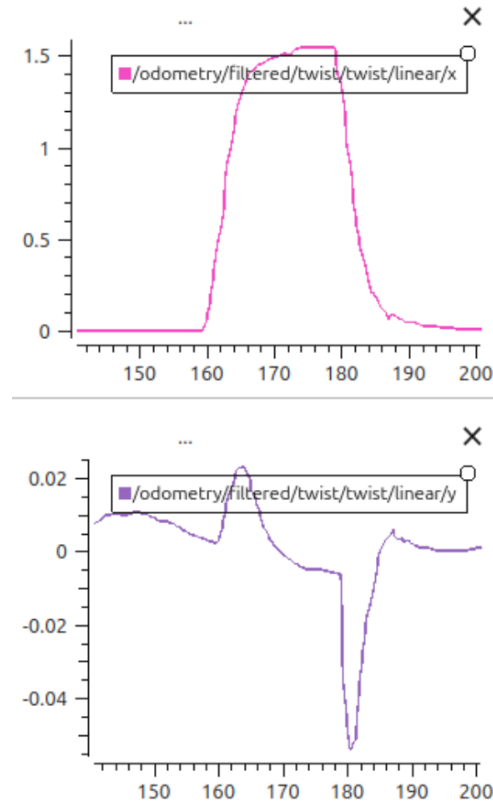


Figure 2.27: Observing the V_x and V_y while the Robot is moving from position A(0,0) to position B(20,20)

For the case of the robot moving from position A(0,0) to position B(20,20):

In Figure 2.26, the X and Y positions estimated by the EKF align closely with those measured by the GPS sensor. The values remain near zero during the initial phase, increase steadily in the second phase, and stabilize in the final phase.

The dynamic response is analyzed during a navigation task from point A(0,0) to B(20,20). Figure 2.27 illustrates the robot's linear velocity profile. The top graph displays the longitudinal velocity (v_x) on the y-axis (m/s) versus time on the x-axis. It clearly shows three distinct phases: an acceleration phase, a constant cruise speed phase where the velocity stabilizes at approximately 1.5 m/s, and a deceleration phase as the robot approaches the target. Meanwhile, the bottom graph shows the lateral velocity (v_y), which remains close to zero, indicating that the controller successfully maintained a straight trajectory with minimal lateral drift.

Visualizing the Boat while on a differential thrust

Applying a differential thrust turning, to test our yaw graph corrected by EKF, applying opposing thrusts on either side to rotate the vessel without forward movement.



Figure 2.28: Rotation around the vertical Axis of the vessel in Gazebo

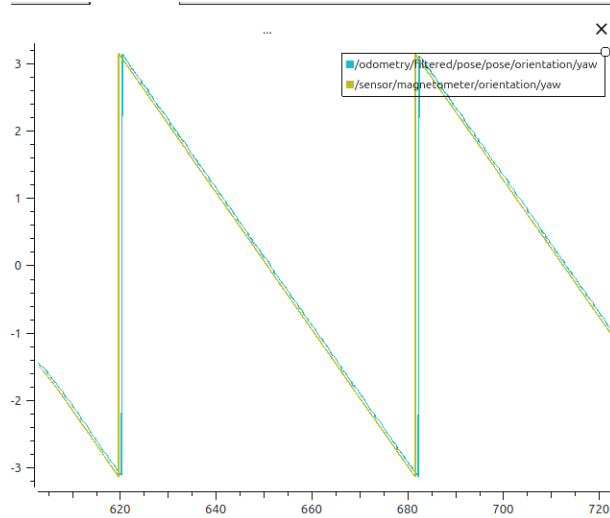


Figure 2.29: Spin Cycle graph for visualizing the Yaw orientation from π to $-\pi$

By applying differential thrust turning, we tested the yaw correction provided by the EKF. Opposing thrusts were applied on either side of the vessel to induce rotation without forward movement.

The resulting yaw graph, shown in Figure 2.29, demonstrates a continuous spin cycle, where the yaw angle oscillates between π and $-\pi$, then returns to π again, repeating this cycle. This periodic behavior indicates that the EKF effectively tracks the vessel's yaw orientation, although the sensor readings, due to noise, show some irregularities. The EKF-corrected yaw angle remains stable within the expected range, following the rotation pattern accurately.

Proper EKF operation was confirmed when:

1. The position estimates $x(t), y(t)$ showed smooth convergence toward the GPS-corrected global path.
2. The yaw estimate $\psi(t)$ was consistent with the visual orientation of the boat in RViz2.
3. No large oscillations or divergence appeared when the ASV moved, indicating a correct fusion of GPS, IMU, magnetometer, and altimeter.

2.8 Conclusion

In conclusion, this chapter has presented a comprehensive overview of the implementation of the autonomous USV platform. The adopted development workflow beginning with PC-based simulation and extensive testing before deployment on the embedded Raspberry Pi 5 proved effective in ensuring system robustness and simplifying debugging. The integration of low-cost sensors, including the GY-87 IMU and NEO-M8N GPS, through an Arduino Due was successfully achieved, with appropriate calibration procedures applied to reduce sensor biases and measurement inaccuracies.

The ROS 2 based software architecture was thoroughly detailed, emphasizing the key components responsible for sensor data processing, dual EKF-based state estimation, and Sliding Mode Control (SMC) for path-following. Furthermore, the Gazebo simulation environment, incorporating realistic hydrodynamic models and sensor plugins, served as a critical platform for algorithm development and preliminary validation. Configuration elements such as launch files, transformation frame (TF) management, and validation tools including RViz2 and PlotJuggler were also presented, enabling effective monitoring and verification of system performance.

Overall, the successful integration of hardware and software components establishes a solid foundation for autonomous navigation in dam environments and prepares the system for subsequent real-world validation and deployment.

Chapter 3

Control strategy

3.1 Introduction

Autonomous surface vehicles (USVs) operate in a dynamic and uncertain environment where disturbances such as wind, waves, and currents significantly affect their motion. Designing a robust control strategy is therefore essential to ensure accurate trajectory tracking, reliable maneuvering, and safe navigation. This chapter presents the control approaches developed and tested in this work, with a focus on achieving robust performance in simulation using the WAM-V platform in Gazebo and ROS 2.

We begin by introducing the Sliding Mode Control (SMC) framework, which is known for its robustness to modeling uncertainties and external disturbances. The SMC law is implemented for path-following using filtered odometry feedback and reference paths generated by ROS 2 nodes. Details of the WAM-V physical parameters, controller formulation, node structure, and tuning process are provided, alongside verification and experimental results.

Next, we introduce a data-driven modeling approach for the surface vehicle dynamics using a Recursive Least Squares Auto-Regressive with eXogenous input (RLS-ARX) model. This allows for online system identification and adaptation to varying dynamics. Building on this model, an adaptive PID controller is developed, with its parameters tuned in real time using RLS. This approach enhances the ability of the controller to adapt to environmental changes and unmodeled dynamics.

Finally, we present a more classical guidance and control scheme based on Line-Of-Sight (LOS) guidance combined with PD/PI controllers. This cascaded structure separates geometric path-following from low-level surge and yaw control, offering a simple yet effective alternative to the SMC.

By comparing these different strategies, this chapter highlights the trade-offs between robustness, adaptability, and implementation complexity. The presented results demonstrate the strengths and limitations of each method and provide insights into their practical deployment in autonomous surface navigation.

3.2 Sliding Mode Control (SMC) Implementation

The control objective in this work was to design a robust trajectory tracking algorithm for the autonomous surface vehicle (USV). For this purpose, a Sliding Mode Control (SMC) law was implemented due to its robustness against external disturbances (e.g., waves, currents) and model uncertainties in the hydrodynamic coefficients.

To implement an SMC path follower that consumes a reference path produced by a path from a ROS2 node, uses EKF-filtered odometry for feedback, maps surge and yaw commands to the two thrusters (left/right), is robust to small wave disturbances in the Gazebo Sydney Regatta world, and produces live visualization of both the reference and executed path in RViz2.

3.2.1 Gazebo/WAM-V Model and Extracted Physical Parameters

The SDF for the WAM-V model was inspected to determine physically meaningful values used by the controller and to choose starting gains.

Important extracted values

- Mass (base_link): $m = 180.0\text{kg}$
- Yaw inertia: $I_{zz} = 446.0\text{kg m}^2$
- Thruster lateral positions (from SDF):
 - left propeller $y = 1.027135\text{m}$
 - right propeller $y = -1.027135\text{m}$
- Thruster separation: $s = 2.05427\text{ m}$, half-distance $d = 1.027135\text{ m}$
- Thruster plugin parameters:
 - thrust_coefficient = 0.004422
 - propeller diameter = 0.2 m
 - fluid density = 1000 kg/m³
 - velocity_control = true (plugin expects motor velocity commands internally)
- Hydrodynamic coefficients (Gazebo plugin values used as drag approximations):
 - $xU = -51.3$, $xAbsU = -72.4$ (surge linear / quadratic damping)
 - $yV = -40.0$ (sway)
 - $nR = -400.0$ (yaw damping)
- Odometry plugin configured:
 - odom_topic = odom/unfiltered
 - odom_frame = odom
 - robot_base_frame = base_link
 - publish_tf = true
 - odom_publish_frequency = 50 Hz
 - rpy_offset = (1.570796, 0, 0) which equals $\pi/2$ rotation in roll (this produces a 90° yaw/visual mismatch unless compensated)

3.2.2 Velocity Projection into Path-Aligned Frame

To compute motion components relative to the path, the robot's velocity vector is projected into a frame aligned with the path tangent. This is useful for separating motion *along* the path and *across* it.

Let (v_x, v_y) be the robot's linear velocity components in the global (inertial) frame. The path heading is given by ϕ , which defines a local path-aligned frame:

- The **along-path direction** (tangent) is aligned with angle ϕ .
- The **cross-path direction** (normal) is perpendicular to it.

To transform the velocity vector into this path-aligned frame, we apply a 2D rotation matrix:

$$\begin{bmatrix} v_{\text{along}} \\ v_{\text{cross}} \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad (3.1)$$

This gives:

$$v_{\text{along}} = v_x \cos(\phi) + v_y \sin(\phi) \quad (3.2)$$

$$v_{\text{cross}} = -v_x \sin(\phi) + v_y \cos(\phi) \quad (3.3)$$

Here, v_{cross} represents the rate at which the robot is moving away from the path.

This transformation is fundamental in geometric tracking controllers, especially when designing feedback based on how the robot deviates from the path laterally.

3.2.3 SMC Controller Design

Control objectives

Track a 2D waypoint-based trajectory (spline) while commanding only the two thrusters. The vehicle is underactuated (no direct sway thruster) the SMC controls surge (forward speed) and yaw to follow the path.

Inputs

- Position/Orientation: $\mathbf{q} = [x, y, \psi]^T$ from /odometry/filtered

Error definitions and sliding surface

Define:

- x, y, ψ : vehicle pose (from EKF)
- u : forward speed (body-frame)
- cross-track error y_e : signed lateral distance from the vehicle to the reference path tangent
v_cross
- heading error $\psi_e = \psi_{\text{path}} - \psi$

Sliding variable:

$$s = \psi_e + \lambda y_e \quad (3.4)$$

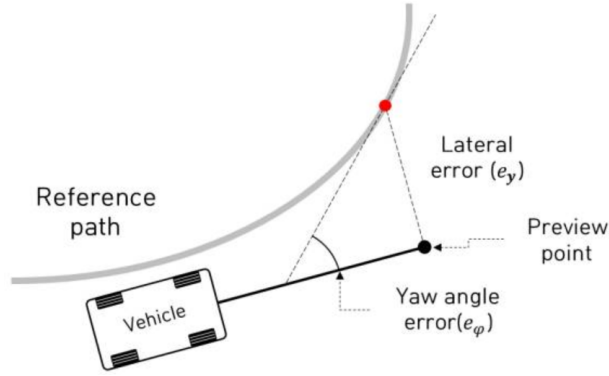


Figure 3.1: Control errors: lateral error and yaw angle error for path tracking

The SMC objective is $s \rightarrow 0$. A boundary-layer saturation (epsilon) replaces the discontinuous sign for practical implementation.

Control law (practical implementation)

Implemented control law in the node:

$$\text{switching} = \begin{cases} k_s \text{sign}(s) & |s| > \varepsilon \\ k_s \frac{s}{\varepsilon} & |s| \leq \varepsilon \end{cases}$$

$$r_{\text{cmd}} = -k_\psi \psi_e - \text{switching} \quad (\text{yaw-rate-like command})$$

$$F_{\text{total}} = k_{\text{speed}}(u_{\text{des}} - u) \quad (\text{surge force command})$$

Mapping to thrusters:

$$\begin{cases} F_L = \frac{F_{\text{total}}}{2} - \frac{\Delta}{2} \\ F_R = \frac{F_{\text{total}}}{2} + \frac{\Delta}{2} \end{cases} \quad \Delta = \frac{M_{\text{cmd}}}{d} \quad (3.5)$$

where:

$M_{\text{cmd}} = k_{\text{moment}} r_{\text{cmd}}$ and d is half the thruster separation.

Rate-limiting and clipping are applied to F_L, F_R .

3.2.4 SMC Implementation, node and Parameters

The SMC node is implemented in ROS2 Python. It subscribes to:

- `/reference_path` (nav_msgs/Path) spline path
- `/odometry/filtered` (nav_msgs/Odometry) EKF-filtered state

and publishes:

- `/wamv/left_thrust` (std_msgs/Float64)
- `/wamv/right_thrust` (std_msgs/Float64)

- /viz/actual_path (nav_msgs/Path) executed path recorder
- /smc/target_point (geometry_msgs/PoseStamped) target point
- /smc/control_marker (visualization_msgs/Marker)

Default SMC node parameters

Parameter	Default value	Meaning
mass	180.0	vehicle mass (kg)
Izz	446.0	yaw inertia (kgm^2)
thruster_separation	2.05427	distance between thrusters (m)
thruster_half	1.027135	half-distance d (m)
desired_speed	0.6	target forward speed (m/s)
k_speed	120.0	map speed error to thrust units
k_psi	0.6	P gain on heading
k_s	6.0	SMC switching gain
epsilon	0.3	SMC boundary layer
k_moment	5.0	maps r_{cmd} to yaw moment
lambda_ct	0.4	cross-track weight in sliding variable
max_thrust	140.0	thrust clip per thruster (units of command)
max_thrust_rate	40.0	N/s rate limit on thrust changes
lookahead_distance	4.0	distance-based lookahead (m)
odom_yaw_compensation	-1.570796	subtract $\pi/2$ from odom yaw by default

Table 3.1: Key SMC parameters used and recommended starting values.

3.2.5 Tuning Procedure

Following the procedure below when tuning in simulation:

1. **Safety stop:** immediately set both thrusters to 0 if the vehicle is unstable.
2. **Frame TF check:** run `tf2_echo odom base_link`, inspect odometry messages to ensure orientation is sensible. Compensate for the SDF odom *rpy_offset* by setting `odom_yaw_compensation` or by fixing the SDF to remove the offset.
3. **Thruster sign test:** manually publish left/right thrusts and observe forward/yaw response and set inversion flags accordingly.
4. **Start conservative:**
 - `desired_speed` = 0.6 m/s
 - `k_speed` = 120.0
 - `k_psi` = 0.6
 - `k_s` = 6.0
 - `k_moment` = 5.0
 - `epsilon` = 0.3

5. **Tune speed loop** (k_{speed}) so thrust commands required to achieve desired speed do not frequently saturate at max_thrust .
6. **Tune yaw P** (k_{psi}) to get faster heading correction without overshoot.
7. **Tune switching gain** (k_s) and **epsilon** for robustness vs chattering. If thruster chattering is observed, increase epsilon rather than increasing k_s too much.
8. **Tune yaw mapping** (k_{moment}) to prevent excessive differential thrust. Use the thruster half-distance d to calculate $\Delta = M/d$.

3.2.6 RViz2 Visualization

Recommended RViz items:

- Path: `/reference_path` (reference spline)
- Path: `/viz/actual_path` (executed path)
- PoseStamped: `/smc/target_point`
- RobotModel or TF to display vehicle pose (ensure EKF publishes `odom->base_link`)

3.2.7 Integration with the Simulation

The SMC node was implemented as a standalone ROS 2 node, subscribing to the estimated state provided by the dual EKF filter (`/odometry/filtered`) and the reference path published by the path publisher node. It outputs thruster commands to the simulation actuators through the Gazebo bridge.

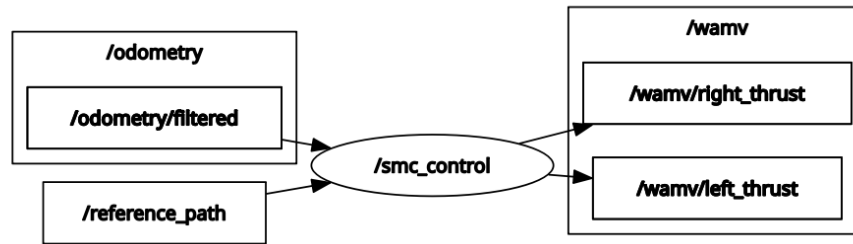


Figure 3.2: The SMC node in ROS2

3.2.8 Verification Tools

To validate the SMC behavior, the control trajectories were visualized in RViz2 alongside the estimated boat pose. PlotJuggler was used to monitor the sliding variable $s(t)$, the error signals $e(t)$, and the commanded control inputs $u(t)$. Successful convergence was indicated by $s(t) \rightarrow 0$ and the ASV trajectory following the reference path with minimal steady-state error.

3.2.9 Experimental Results

After tuning my SMC through trials and errors, the robot successfully followed the path with a small error and less chattering.

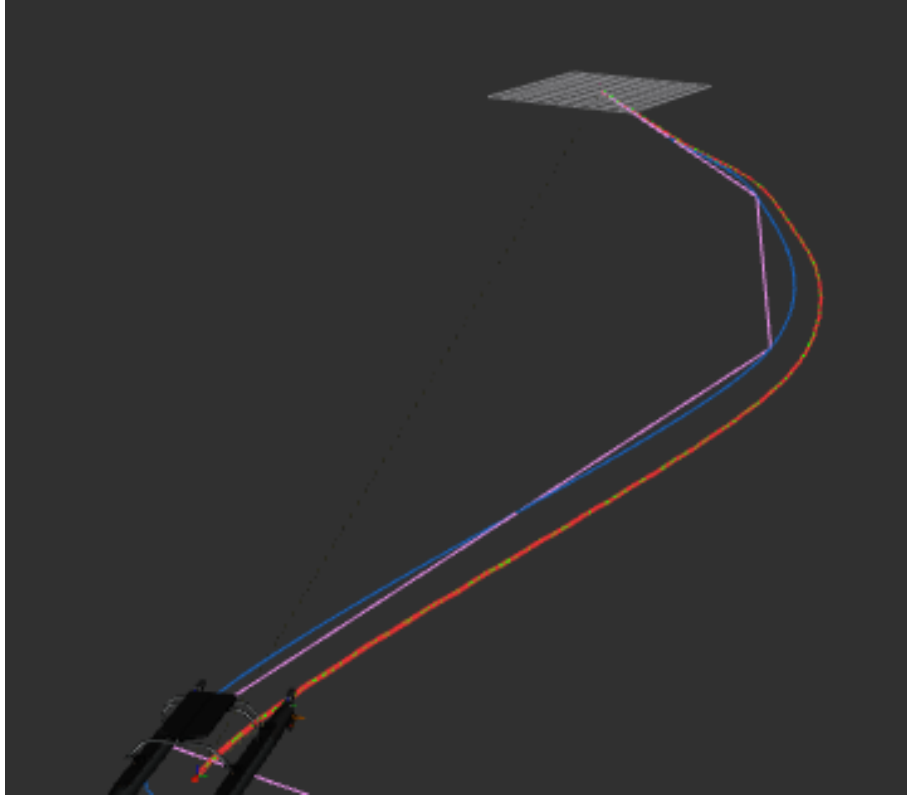


Figure 3.3: The robot following the path through the SMC controller

3.3 Kinematic Modeling of Surface Motion: Heading and Translational Dynamics

The AquaRob system operates exclusively on the water surface. Its behavior is characterized by two translational components along the X and Y axes, as well as a rotational component around the vertical axis representing the heading.

The modeling objective is to accurately represent the planar dynamics of motion and orientation, taking into account the effects of hydrodynamic resistance, environmental disturbances (such as wind), and the differential thrust produced by the left and right actuators.

A simplified data-driven model, such as an ARX (Auto-Regressive with eXogenous inputs) model, is used to capture the systems inputoutput relationship based on motor commands and the observed heading response.

3.3.1 Modeling Using the RLS-ARX Approach

In this study, we model the AquaRob surface drone dynamics by considering two input commands and a single output variable. The inputs correspond to the right and left actuator commands, denoted as $u_1(t)$ and $u_2(t)$, while the output is the measured heading angle $y(t)$

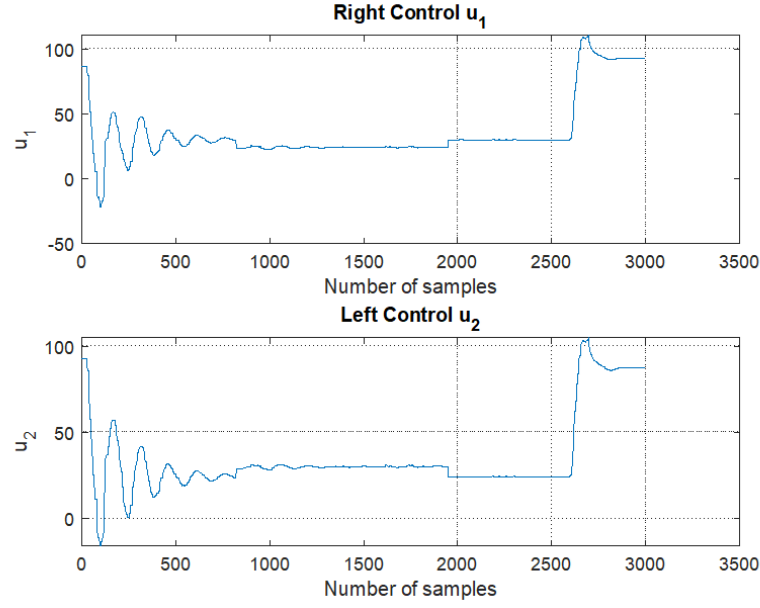


Figure 3.4: Inputs of the System

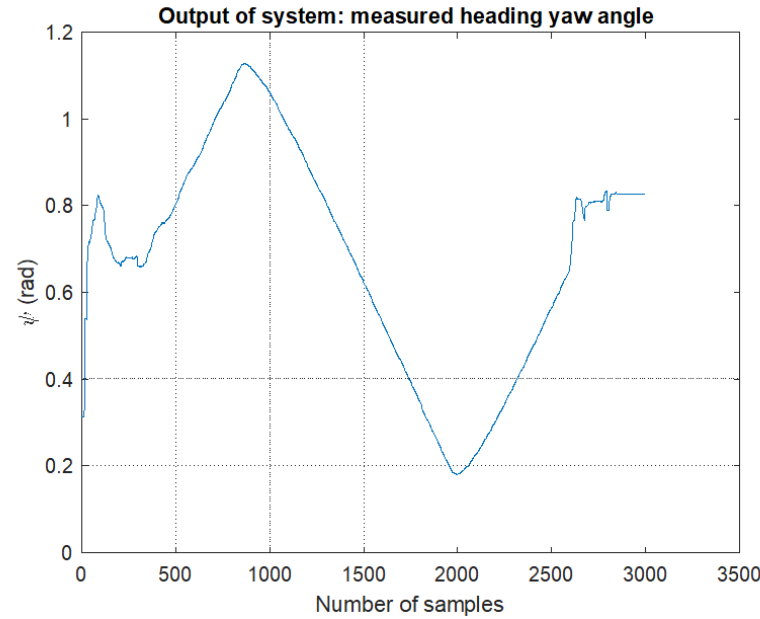


Figure 3.5: Output of the System

We adopt a discrete-time ARX (Auto-Regressive with exogenous inputs) model structure to describe the system dynamics. The model expresses the current output $y(t)$ as a linear combination of past outputs and past inputs :

$$y(t) + a_1 y(t-1) + a_2 y(t-2) = b_1 u_1(t-1) + b_2 u_1(t-2) + b_3 u_2(t-1) + b_4 u_2(t-2) + e(t) \quad (3.6)$$

where a_i and b_{ij} are the unknown model parameters to be estimated, and $e(t)$ represents the noise term.

The choice of using two time delays for each input and output results in a total of 6 parameters to estimate:

- 2 auto regressive parameters (a_1, a_2) related to past outputs,
- 4 parameters b associated with past inputs.

Parameter estimation is performed online using the Recursive Least Squares (RLS) algorithm, which allows real-time adaptation of the model coefficients based on incoming data. This RLS-ARX modeling approach enables effective capturing of the system dynamics for control and prediction purposes.

3.3.2 Matrix Formulation and Online Parameter Estimation Using RLS

The ARX model can be rewritten in vectormatrix form to facilitate the application of the Recursive Least Squares (RLS) algorithm.

At each time step t , the output $y(t)$ is approximated by the inner product between a regressor vector $\phi(t)$ and a parameter vector $\theta(t)$:

$$y(t) = \phi^T(t) \theta(t) + e(t) \quad (3.7)$$

where

$$\phi(t) = [-y(t-1), -y(t-2), u_1(t-1), u_1(t-2), u_2(t-1), u_2(t-2)]^T, \quad (3.8)$$

$$\theta(t) = [a_1, a_2, b_{11}, b_{12}, b_{21}, b_{22}]^T. \quad (3.9)$$

The goal is to estimate the parameter vector $\theta(t)$ in real time from available inputoutput data.

3.3.3 Recursive Least Squares (RLS) Algorithm

The Recursive Least Squares (RLS) algorithm provides an efficient way to update the parameter estimates at each time step, using the following update equations:

Estimation error:

$$e(t) = y(t) - \phi^T(t) \theta(t-1) \quad (3.10)$$

Gain vector:

$$K(t) = \frac{P(t-1) \phi(t)}{\lambda + \phi^T(t) P(t-1) \phi(t)} \quad (3.11)$$

Parameter update:

$$\theta(t) = \theta(t-1) + K(t)e(t) \quad (3.12)$$

Covariance matrix update:

$$P(t) = \frac{1}{\lambda} [P(t-1) - K(t)\phi^T(t)P(t-1)] \quad (3.13)$$

This method allows for adaptive tracking of system dynamics, which is particularly useful in the presence of time-varying behavior or unmodeled disturbances.

where:

$\theta(t)$ is the estimated parameter vector at time t

$P(t)$ is the covariance matrix (usually initialized as a large diagonal matrix).

$\lambda \in (0, 1]$ is the forgetting factor, which controls how quickly the algorithm adapts to new data.

This method allows for adaptive tracking of system dynamics, which is particularly useful in the presence of time-varying behavior or unmodeled disturbances.

3.3.4 Results simulation and Model Validation

To validate the performance of the identified RLS-ARX model, a set of experimental data was collected from the AquaRob surface drone under realistic operating conditions. The input signals correspond to the right and left motor commands, which were varied to excite the systems heading dynamics over a range of amplitudes and frequencies. The Recursive Least Squares algorithm was applied in real time to estimate the six parameters of the ARX model. Figure 3.6 and 3.7 show the evolution of the estimated parameters during the experiment. As the data accumulates, the parameter values converge, indicating the stability and consistency of the identification process.

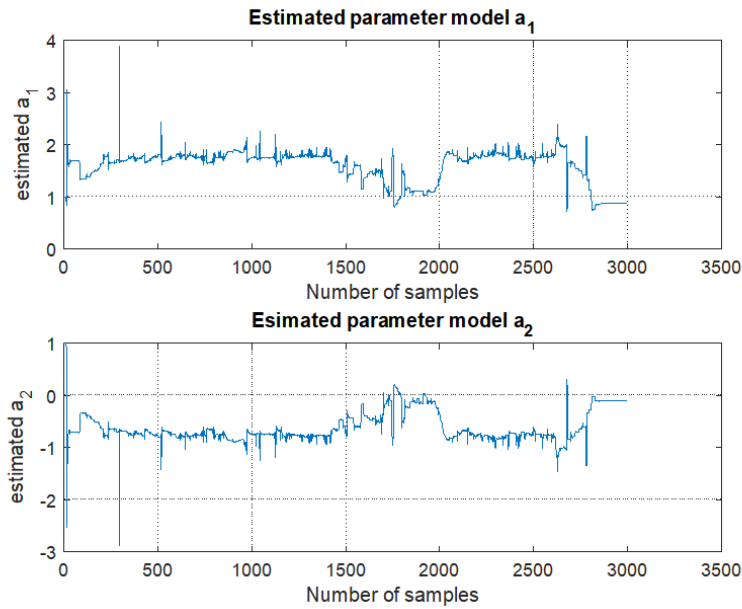


Figure 3.6: Estimated parameters model ai

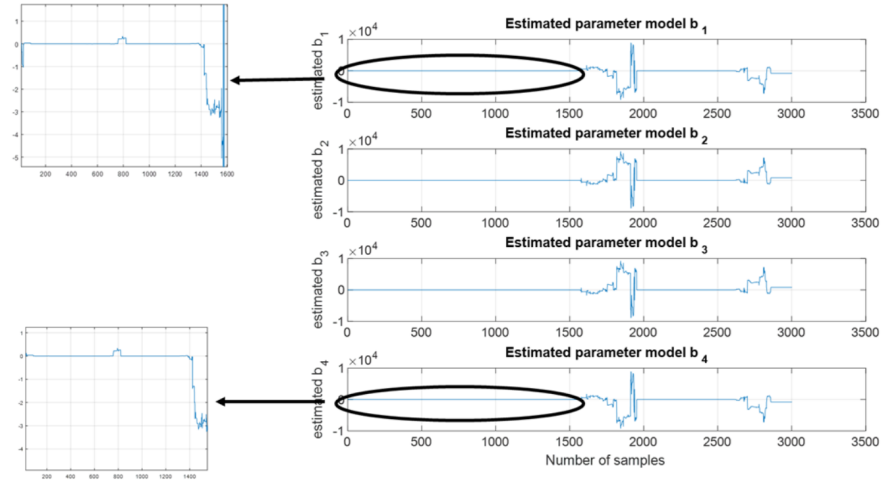


Figure 3.7: Estimated parameters model b_i

In addition, the predicted output computed as

$$\hat{y}(t) = \varphi^T(t) \hat{\theta}(t) \quad (3.14)$$

was compared to the measured heading angle $y(t)$. As shown in Figure 3.8, the estimated output closely follows the actual response, demonstrating the accuracy of the ARX model in capturing the system dynamics.

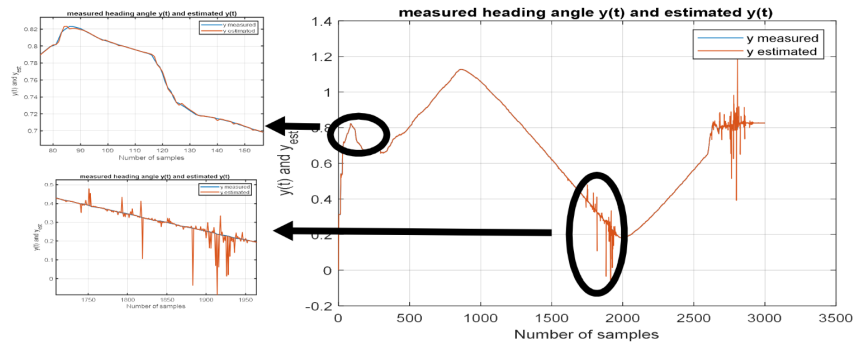


Figure 3.8: Measured heading angle $y(t)$ and estimated angle $\hat{y}(t)$

To further assess model quality, the prediction error $e(t) = y(t) - \hat{y}(t)$ was analyzed. The error remains bounded and relatively small throughout the experiment, confirming that the model generalizes well to the observed behavior of the system.

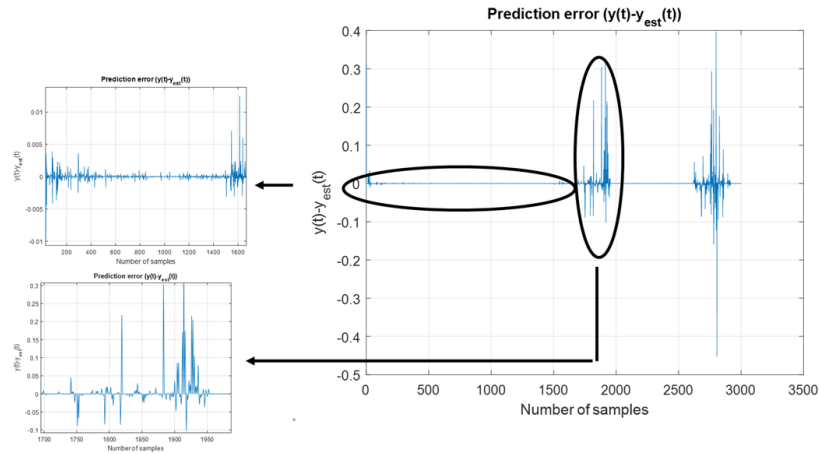


Figure 3.9: Prediction error $e(t) = y(t) - \hat{y}(t)$

As shown in the Figure 3.9, the prediction error remains very small during the initial phase of the simulation, particularly from sample 0 to 1600, where it fluctuates within a narrow range between 0.006 and -0.004, i.e., on the order of 10^{-3} . This indicates a high level of accuracy in the estimated output during this interval.

However, between samples 1740 and 1950, the error exhibits noticeable deviations, ranging approximately from 0.3 to -0.1. These fluctuations may be attributed to sudden changes in input signals, unmodeled dynamics, or external disturbances affecting the heading response. After this disturbance phase, the error decreases again to the order of 10^{-3} , confirming that the model regains its predictive capability. Nevertheless, another disturbance appears starting around sample 2620, where the error rises once more to the order of 10^{-1} .

Despite these localized variations, the overall prediction error remains relatively low across the simulation, confirming the models ability to capture the systems dominant dynamics while highlighting areas where model refinement or disturbance rejection may be necessary.

3.3.5 System and Adaptive PID Controller Modelling

Consider a single-input, single-output (SISO) system represented by the transfer function $G(s)$. The PID controller is characterized by the parameter vector $\Theta = [K_p, K_i, K_d]$.

, which needs to be adaptively tuned. Let u denote the control input, y the system output, r the desired reference signal, and $e = y - r$ the tracking error.

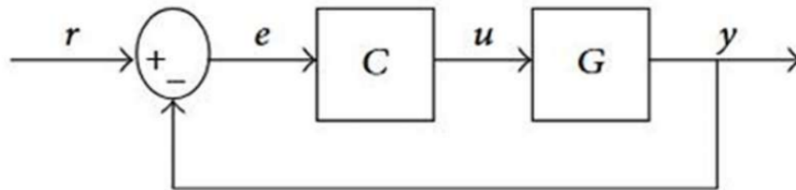


Figure 3.10: Single-Input Single-Output (SISO) Linear System

The PID controller transfer function can be expressed as:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{1 + \tau s} \quad (3.15)$$

Here, the derivative term incorporates a first-order low-pass filter with a small time constant τ to reduce noise amplification.

Rearranging, we define:

$$\Theta^t = [K_p \quad K_i \quad K_d] \quad (3.16)$$

$$\Phi_1 = \begin{bmatrix} 1 \\ \frac{1}{s} \\ \frac{1}{\tau s + 1} \end{bmatrix} e(s) \quad (3.17)$$

so that the controllers transfer function can be succinctly written as:

$$C(s) = \Phi^T \Theta \quad (3.18)$$

In reality, the transfer function G may be unknown or vary over time, necessitating an adaptive control strategy to update Θ continuously to minimize the tracking error. The objective is to ensure that the output $y(t)$ follows the desired trajectory defined by a reference model $M(s)$.

$$y_m(t) \approx y(t) \quad (3.19)$$

This can also be written in terms of the reference model as:

$$M(s)r(t) \approx y(t) \quad (3.20)$$

where $M(s)$ is a given reference model transfer function, representing the desired closed-loop dynamics.

Equation (3.20) can be further rewritten to express the relationship in terms of the system tracking error:

$$M(s)r(t) \approx r(t) - e(t) \quad (3.21)$$

By adding the controller transfer function $C(s)$ to both sides of Equation (3.21), we obtain:

$$C(s)M(s)r(t) \approx C(s)[r(t) - e(t)] \quad (3.22)$$

This simplifies to:

$$C(s)[1 - M(s)]r(t) \approx u(t) \quad (3.23)$$

Hence, the RLS estimation error can be defined as:

$$\xi = u(t) - C(s)[1 - M(s)]r(t) \quad (3.24)$$

Substituting the controller expression from earlier:

$$\xi = u(t) - \Theta^T \Phi_1 [1 - M(s)]r(t) \quad (3.25)$$

3.3.6 Real-Time Parameter Adaptation via RLS

The RLS algorithm is applied to iteratively update the PID parameters Θ by minimizing the exponentially weighted squared estimation error:

$$J = \sum_{k=1}^N \lambda^{k-i} \xi^2(k) \quad (3.26)$$

Where (with $0 < \lambda < 1$) is a forgetting factor that balances sensitivity to new data against retention of past information, enabling the controller to track dynamic changes effectively.

The error can be simplified as:

$$\xi = u - \Theta^T \Phi \quad (3.27)$$

with:

$$\Phi = \Phi_1 [1 - M(s)] r(t) \quad (3.28)$$

The recursive update equations for the parameter estimate $\hat{\Theta}(k)$ at iteration k are:

$$\hat{\Theta}(k) = \hat{\Theta}(k-1) + K(k) [u(k) - \Phi_{\text{RLS}}^T(k) \hat{\Theta}(k-1)] \quad (3.29)$$

where the gain vector $K(k)$ and covariance matrix $P(k)$ are updated as:

$$K(k) = P(k-1) \Phi_{\text{RLS}}(k) [\lambda + \Phi_{\text{RLS}}^T(k) P(k-1) \Phi_{\text{RLS}}(k)]^{-1} \quad (3.30)$$

$$P(k) = \frac{1}{\lambda} [P(k-1) - K(k) \Phi_{\text{RLS}}^T(k) P(k-1)] \quad (3.31)$$

3.3.7 Adaptive Update via RLS

The estimation error is defined as:

$$\xi(t) = u(t) - \Theta^T \Phi(t) \quad (3.32)$$

The cost function:

$$J = \sum_k \lambda^k \xi^2(k) \quad (3.33)$$

The recursive update equations:

$$\Theta(k) = \Theta(k-1) + K(k) [u(k) - \Phi^T(k) \Theta(k-1)] \quad (3.34)$$

$$K(k) = \frac{P(k-1) \Phi(k)}{\lambda + \Phi^T(k) P(k-1) \Phi(k)} \quad (3.35)$$

$$P(k) = P(k-1) - K(k) \Phi^T(k) P(k-1) \quad (3.36)$$

3.3.8 Simulation Results: Estimation of Control Inputs Using RLS-Based PID Adaptation

Convergence of PID Parameters

Figure 3.11 and 3.12 illustrates the evolution of the three PID gains over time. It can be observed that the parameters rapidly converge during the initial phase of the simulation and

then continue to adapt smoothly in response to variations in the system behavior. This confirms the algorithm's capability to track slowly time-varying dynamics and adjust the control law accordingly.

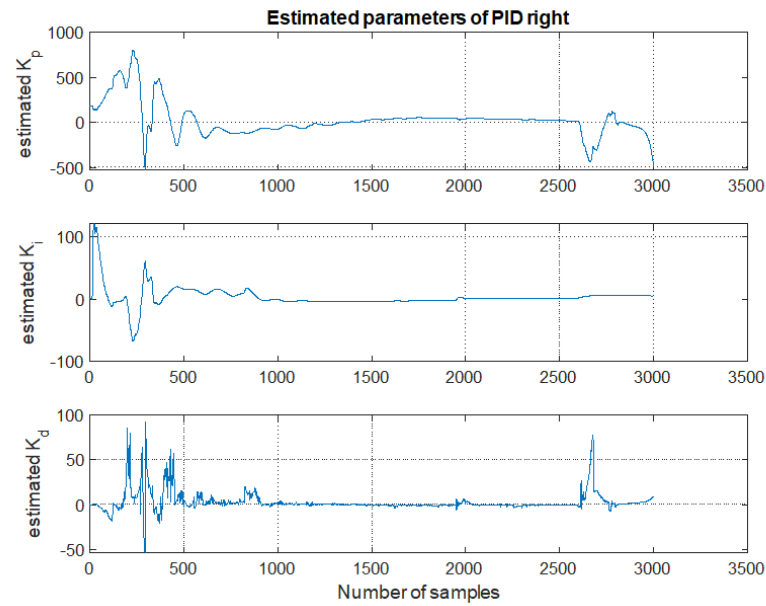


Figure 3.11: Evolution of the PID gains right

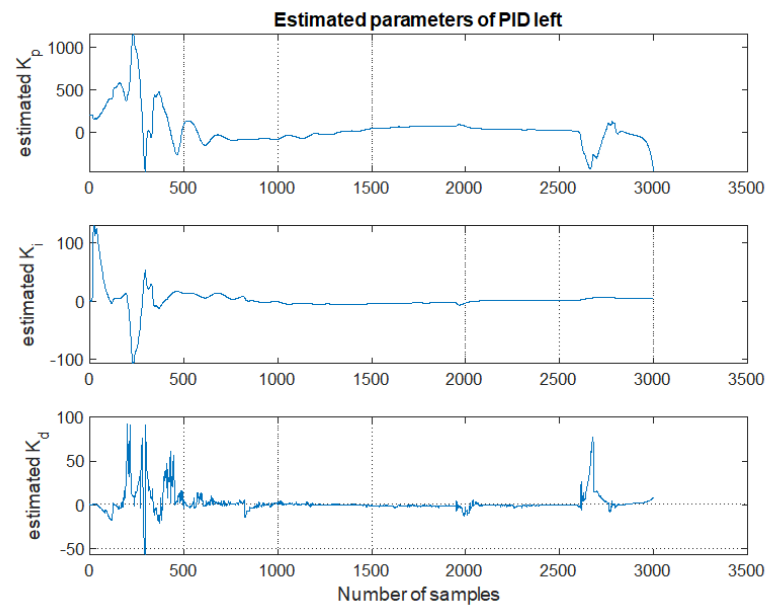


Figure 3.12: Evolution of the PID gains left

Comparison of Measured and Adaptive Control Inputs

To evaluate the accuracy of the RLS-based PID adaptation strategy, we compare the control input signals over the course of the simulation:

- Measured control input: $u(t)$ the actual input applied to the system.

- Adaptive control input: $u(t)$ adaptive
the command generated by the adaptive PID controller applied in closed loop.

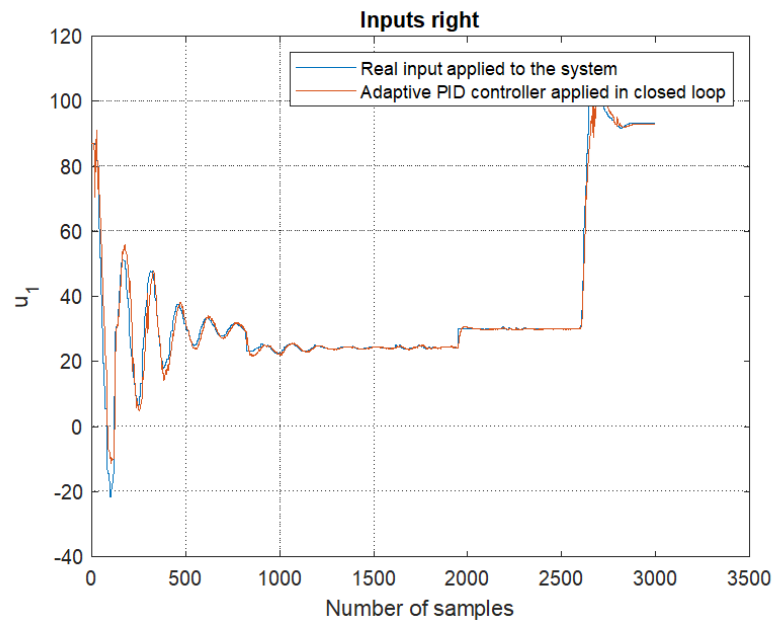


Figure 3.13: Compare the control input right signals

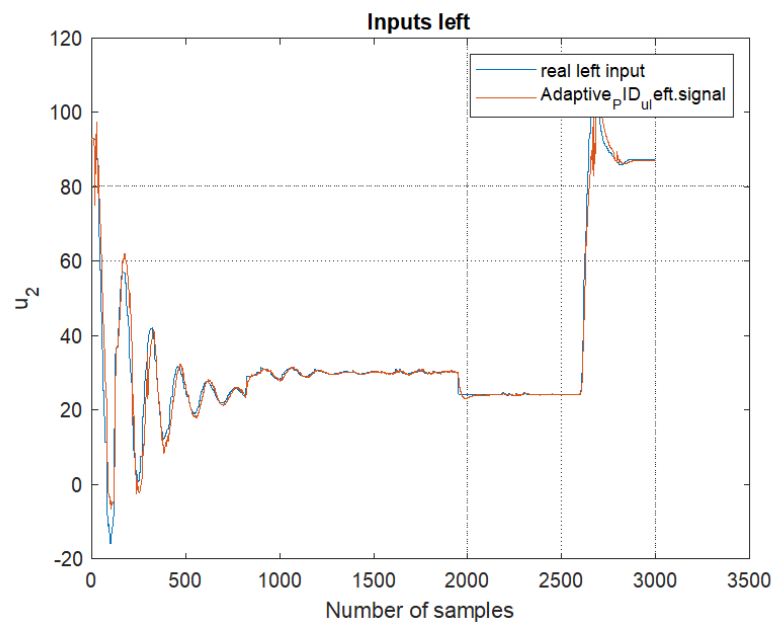


Figure 3.14: Compare the control input left signals

The Mean Squared Error (MSE) values between the measured system output and the estimated output using the adaptive PID controller were 12.0991 for the right motor and 12.6771 for the left motor. These results indicate a generally consistent level of estimation accuracy across both control channels. The slightly higher error on the left side may be attributed to asymmetries in actuator response, sensor noise, or minor differences in system dynamics between the two motors.

Overall, the adaptive PID controller demonstrates satisfactory performance, maintaining the prediction error within a reasonable range, which supports its suitability for real-time implementation in surface drone control.

Tracking performance and model accuracy

Figure 3.15 presents the comparison between the real output, measured, and the estimated output from the ARX model driven by the adaptive PID controller. The Mean Squared Error (MSE) between the actual system output and the output generated by the estimated model using RLS and the adaptive PID controller was 0.5134. This relatively low error value confirms the accuracy of the estimated model and the effectiveness of the adaptive PID control strategy. The controller is able to maintain the output trajectory close to the desired one, validating the suitability of this approach for real-time implementation in surface drone control applications.

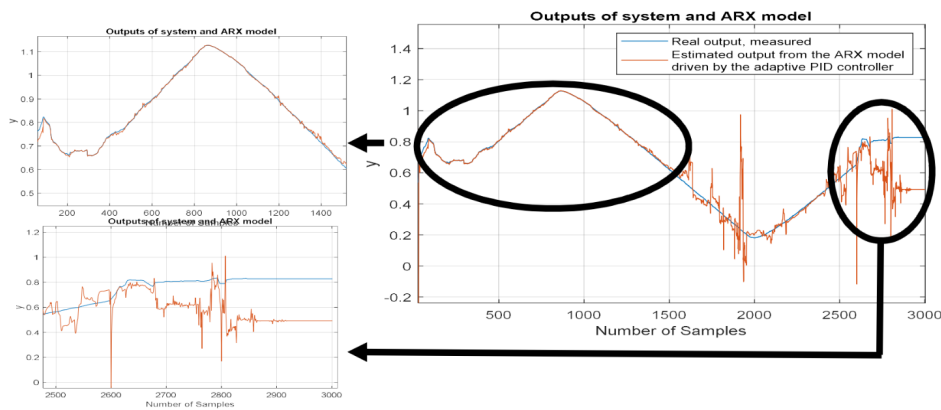


Figure 3.15: Comparison between the real output, measured, and the estimated output from the ARX model driven by the adaptive PID controller.

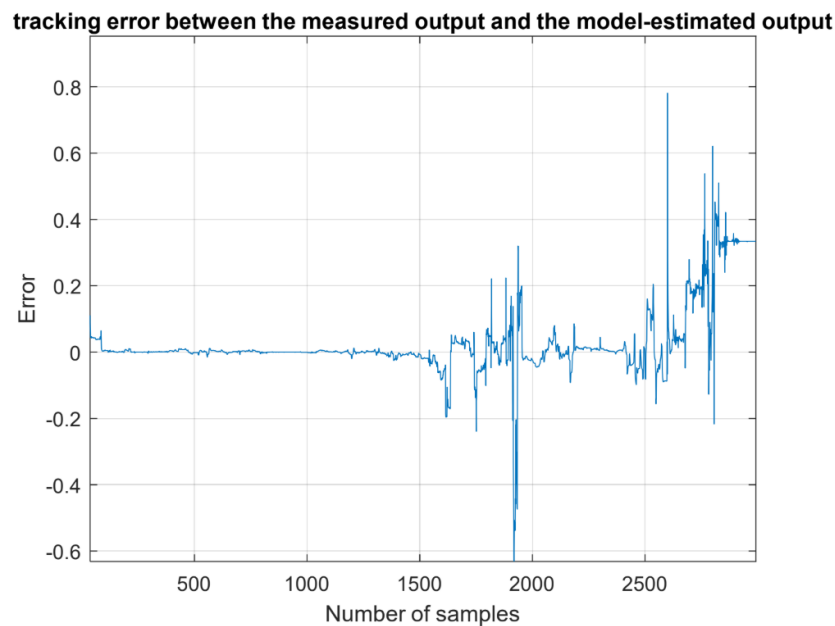


Figure 3.16: Tracking error between the measured output and the model-estimated output.

The tracking error between the measured output and the model-estimated output is initially very low, on the order of 10^{-2} for the first 1500 samples. This indicates a high level of accuracy and good parameter convergence during this initial phase.

However, as the simulation progresses, the tracking error gradually increases and reaches a peak of approximately 0.35 near the end of the data window. This degradation in performance is likely due to significant variations in the estimated parameters, particularly the coefficients b_i of the ARX model.

These variations may be attributed to the quality of the input/output signals used for identification during the final portion of the dataset, which may exhibit low excitation, increased noise, or nonstationary dynamics. In such cases, the RLS algorithm may struggle to provide stable parameter estimates, resulting in less accurate output prediction. Improving the persistency of excitation in the input signals or applying regularization techniques could help mitigate these effects in future implementations.

3.4 LOS guidance with PD/PI inner-loop controller

As an alternative to the Sliding Mode Controller (SMC) described earlier, I implemented and tested a *Line-Of-Sight (LOS) guidance* law combined with classical PD control for heading and a PI controller for surge. This architecture is widely used for underactuated USVs because it cleanly separates geometric guidance (where to go) from low-level motion control (how to follow the heading and speed).

3.4.1 Architecture (cascade)

The controller is organized in two nested levels:

1. **Guidance (LOS):** compute a lookahead target on the reference path and output a desired heading (path bearing) to aim at that target.
2. **Motion control (PD/PI):**
 - Heading loop (PD): computes a desired yaw rate r_{des} from heading error and measured yaw rate.
 - Surge loop (PI): computes a forward thrust scalar F_{total} from the speed error.
3. **Actuator mapping:** map $\{F_{\text{total}}, M_{\text{cmd}}\}$ to left/right thruster commands, apply inversion, clipping and rate-limiting, and publish.

3.4.2 LOS target selection

Given a reference path represented by a sequence of waypoints, compute the projection of the vehicle position (x, y) onto the path segments, then advance by a *lookahead distance* Δ along the path to obtain the LOS target point (t_x, t_y) and the local path tangent (heading) ψ_p . This is the same distance-based lookahead strategy used elsewhere in this work.

3.4.3 Control laws

Define the heading (yaw) error

$$\psi_e = \text{wrap_to_}\pi(\psi_p - \psi), \quad (3.37)$$

where ψ is the measured vehicle yaw and $\text{wrap_to_}\pi$ maps angles to $(-\pi, \pi]$.

Heading (PD) control:

$$r_{\text{des}} = K_{\psi} \psi_e - K_d r, \quad (3.38)$$

where r is the measured yaw rate (rad/s), K_{ψ} is the proportional gain on heading error and K_d is the derivative/damping gain on yaw rate. The desired yaw moment is obtained by

$$M_{\text{cmd}} = K_M r_{\text{des}}, \quad (3.39)$$

optionally clamped to $\pm M_{\text{max}}$ to respect actuator limits.

Surge (PI) control:

$$e_v = v_{\text{des}} - u, \quad (3.40)$$

$$I_v(t) = \text{sat}(I_v(t - \Delta t) + e_v \Delta t, -I_{\text{max}}, I_{\text{max}}), \quad (3.41)$$

$$F_{\text{total}} = K_{p,v} e_v + K_{i,v} I_v(t), \quad (3.42)$$

where u is measured forward speed, v_{des} is the desired speed, I_v is the integrator with anti-windup saturation, and $K_{p,v}, K_{i,v}$ are the PI gains.

Thruster mapping (surge + moment $\rightarrow$ left/right): Assuming symmetric lateral lever arm d (half the separation),

$$F_L = \frac{1}{2} F_{\text{total}} - \frac{1}{2} \Delta F, \quad F_R = \frac{1}{2} F_{\text{total}} + \frac{1}{2} \Delta F, \quad (3.43)$$

$$\Delta F = \frac{M_{\text{cmd}}}{d}. \quad (3.44)$$

Apply inversion flags, clip each thruster to $\pm F_{\text{max}}$ and apply rate limiting: $|\dot{F}| \leq R_{\text{max}}$.

3.4.4 Implementation details

The ROS2 implementation follows the same node structure used for the SMC node:

- Subscribes: `/odometry/filtered` (nav_msgs/Odometry) and `/reference_path` (nav_msgs/Path).
- Publishes: left/right thrust (std_msgs/Float64), debug floats (yaw error, cross-track, r_{des} , F_{total}), and the LOS target as PoseStamped for RViz.
- Loop rate: typical control rate $f_c = 20$ Hz.
- Anti-windup: integrator clamped to I_{max} .
- Saturation: clamp moment and thrusters to configured maxima.
- Initialization: reset integrator and last-command values on start/clear.

3.4.5 Typical parameters and baseline tuning

Parameter	Suggested baseline value
Control rate	20 Hz
Desired speed v_{des}	0.5 – 1.5 m/s (start low)
Surge PI: $K_{p,v}$	30 – 100
Surge PI: $K_{i,v}$	5 – 20
Heading P K_{ψ}	1.0 – 4.0
Heading D K_d	0.1 – 0.5
Moment map K_M	3 – 8
Lookahead Δ	0.3 – 1.0 m
Thruster half-arm d	(from SDF) e.g. 1.027 m
Max thrust F_{max}	per-vehicle (e.g. 140200)
Max thrust rate R_{max}	per-vehicle (e.g. 25100 N/s)
Integrator limit I_{max}	chosen to avoid integral windup (e.g. 100200)

Table 3.2: Typical parameters and baseline tuning ranges used during experiments.

3.4.6 Tuning procedure

1. We start with a low desired speed v_{des} (0.2–0.5 m/s) to ensure safety.
2. Tune heading gains: increase K_{ψ} until the vehicle turns responsively toward the LOS target, then adding K_d to reduce overshoot and oscillation.
3. Tuning surge PI: set $K_{p,v}$ for acceptable acceleration to avoid thruster saturation, then adding small $K_{i,v}$ to remove steady speed error.
4. Adjust lookahead Δ : smaller Δ improves tracking of tight curves but may increase control activity, larger Δ yields smoother trajectories.
5. Monitor saturation: if thrusters saturate frequently, reduce v_{des} or increase moment gain K_M (if yaw authority is insufficient).

3.4.7 Simulation Result

After tuning my parameters, my los algorithm successfully followed the target, Noting that I still have to tune the parameters for a better following through trial and error, same with how I did with the SMC

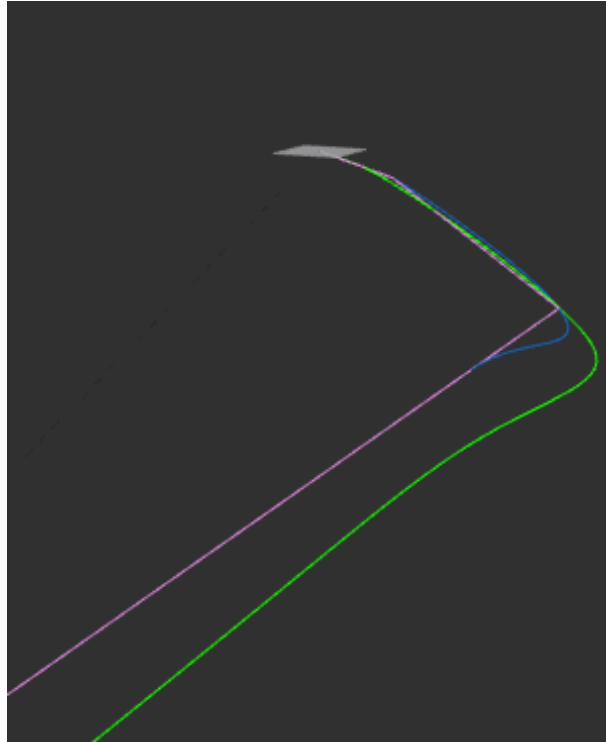


Figure 3.17: The USV following the target in RVIZ2, USV in green, Path target in blue/purple

3.4.8 Advantages and limitations

Advantages:

- Simple, intuitive, easy to tune in practice.
- Separation of guidance and control simplifies debugging and parameter selection.
- Works well in low-to-moderate disturbance environments and has been validated extensively in literature and field tests.

Limitations:

- LOS alone does not compensate for persistent sideslip (currents) may require ILOS/ELOS or an observer to estimate and remove drift.
- Performance depends on gain selection and thruster authority; tight curves or strong disturbances can cause tracking loss.
- Does not provide an explicit optimality guarantee (unlike MPC) and cannot enforce state constraints except via ad-hoc saturation logic.

3.4.9 Example pseudocode

```

1. compute (tx,ty,psi_p) = LOS_target(path, x,y, lookahead=)
2. yaw_err = wrap_to_pi(psi_p - psi)
3. r_des = K_psi * yaw_err - K_d * r
4. M_cmd = clamp(K_M * r_des, -M_max, M_max)
5. speed_err = v_des - u

```

```

6. I_v += clamp(speed_err*dt, -I_max, I_max)
7. F_total = K_pv * speed_err + K_iv * I_v
8. delta = M_cmd / d
9. F_L = 0.5*F_total - 0.5*delta ; F_R = 0.5*F_total + 0.5*delta
10. apply inversion, clamp to sF_max, rate-limit, publish

```

Overall, LOS guidance with PD/PI inner loops is a pragmatic choice for underactuated USV path following: it is simple to implement, interpretable, and widely used as a baseline in both research and field systems. Its weaknesses (mainly sideslip handling) can be mitigated by augmenting the scheme with integral action on heading, a disturbance observer, or by using adaptive/observer-based variants (ILOS/ELOS).

3.5 Conclusion

This chapter presented and compared several control strategies for autonomous surface vehicles (USVs), each with different levels of robustness, adaptability, and implementation complexity. We began with Sliding Mode Control (SMC), which demonstrated strong robustness against disturbances and modeling uncertainties. While effective for trajectory tracking, the controller required careful parameter tuning to mitigate the issue of chattering.

Next, we investigated guidance-based strategies using Line-of-Sight (LOS) combined with PD/PI controllers. This approach offered a simpler and more practical solution, providing reliable path-following performance with relatively low implementation effort.

Finally, we focused on an adaptive control strategy based on online system identification using the Recursive Least Squares (RLS) algorithm, integrated with real-time tuning of a PID controller. The ARX model with six estimated parameters provided reliable output predictions during the early stages of simulation, where the tracking error remained very small (on the order of 10^{-2}). However, toward the end of the simulation, the prediction error gradually increased to about 0.35, likely due to poor excitation or time-varying dynamics in the input/output data. This led to unstable parameter estimation, particularly in the b_i coefficients. Nevertheless, the adaptive PID controller maintained effective trajectory tracking, confirming the potential of data-driven adaptive methods for surface drones. Future improvements may include richer excitation signals, adaptive forgetting factors, or regularized RLS formulations to enhance robustness in dynamic environments.

Overall, the findings of this chapter emphasize that no single control approach is universally optimal. Instead, the choice depends on the trade-off between robustness, adaptability, and implementation complexity. The results also validate the integration of adaptive identification-based control as a promising avenue for future work. Building on these foundations, the subsequent chapters extend beyond low-level control to explore higher-level autonomy, mission planning, and decision-making under environmental and operational constraints.

Conclusion and Future Work

Summary of Contributions

This work has established a comprehensive framework for an autonomous surface vehicle (USV) designed for dam environment navigation, focusing on theoretical design, system architecture, and sensor fusion strategies. While the full implementation remains pending, the proposed system integrates low-cost hardware (Raspberry Pi 5, Arduino, GY-87 IMU, NEO-M8N GPS) with a ROS 2-based software architecture to enable robust state estimation and control under challenging conditions. Key components include a dynamics-aware Extended Kalman Filter (EKF) for fusing GPS, IMU, and magnetometer data to handle GPS outages, and a Sliding Mode Controller (SMC) for trajectory tracking under disturbances. Two operational modes—random waypoint generation within geofenced areas and predefined path following via cubic spline interpolation—were conceptually designed to address diverse mission requirements. The system emphasizes cost-effectiveness and modularity, leveraging open-source tools for scalability. Challenges such as sensor noise in turbulent environments and real-time performance on constrained hardware will be addressed through iterative testing and optimization. This work provides a roadmap for deploying affordable, reliable USVs in critical infrastructure monitoring, bridging theoretical design with practical deployment goals.

Future Extensions

The proposed USV platform offers a foundation for further enhancements to improve safety, autonomy, and scalability. Key directions for future work include:

- **Redesign the hull to be self-righting:** Incorporating self-righting mechanisms, such as inherent hull stability or moveable ballast systems, could enhance safety in rough seas. This aligns with established principles in naval architecture, where self-righting designs ensure recovery after rolling beyond 180° [1]. For example, inflatable bags or weighted keels could be integrated to stabilize the vessel during capsizing [1].

- **Incorporate obstacle avoidance via LiDAR or vision:** Equipping the USV with LiDAR or vision-based sensors would enable real-time environmental perception, allowing dynamic path adjustments to avoid collisions. This builds on autonomous navigation frameworks that prioritize situational awareness in maritime environments.

- **Upgrade EKF to Unscented or Invariant variants:** Replacing the Extended Kalman Filter (EKF) with more advanced variants, such as the Unscented Kalman Filter (UKF) or Invariant EKF, could improve state estimation accuracy, particularly in nonlinear systems. This aligns with modern control strategies that address the limitations of traditional EKFs in handling complex dynamics.

- **Multi-USV cooperative missions:** Expanding the system to support multiple USVs operating in coordination could enable distributed tasks such as area coverage or formation control. This requires robust communication protocols and decentralized control algorithms, reflecting trends in swarm robotics for maritime applications.

- **Advanced Path Planning with Dijkstra's Algorithm:** Implementing Dijkstra's algorithm for optimal path planning would enable efficient obstacle avoidance in complex environments. This graph-based approach could be integrated with real-time sensor data to dynamically replan routes around newly detected obstacles, creating safer and more efficient navigation corridors in confined dam environments.

- **Multi-layer Path Planning Architecture:** Developing a hierarchical planning system combining global A* or Dijkstra algorithms for coarse planning with local potential field or dynamic window approaches for reactive obstacle avoidance. This architecture would balance optimal long-range planning with responsive local navigation.

- **Site-Specific Trajectory Optimization:** Developing customized trajectory libraries for each dam environment based on structural layout, water depth profiles, and mission objectives. These predefined paths would incorporate efficient coverage patterns while respecting physical constraints and environmental features specific to each location.

- **Adaptive or learning-based control (e.g., reinforcement learning):** Implementing adaptive control strategies, including reinforcement learning, could optimize the USVs response to unpredictable environmental conditions. This approach leverages machine learning to iteratively refine control policies, improving resilience in dynamic scenarios.

- **A user interface for waypoint planning and real-time tracking:** Developing a graphical user interface (GUI) would allow operators to input mission-specific waypoints and monitor the USVs position on a map. This enhances usability and aligns with industry standards for remote operation of autonomous systems.

These extensions aim to address current limitations while advancing the USVs capabilities in safety, autonomy, and scalability, ensuring its applicability to real-world maritime challenges.

Annex

Arduino Source Code

```
1 #include <micro_ros_arduino.h>
2 #include <rcl/rcl.h>
3 #include <rcl/rclc.h>
4 #include <rclc/executor.h>
5 #include <sensor_msgs/msg/imu.h>
6 #include <sensor_msgs/msg/magnetic_field.h>
7 #include <sensor_msgs/msg/nav_sat_fix.h>
8 #include <sensor_msgs/msg/fluid_pressure.h> // Barometer [[6]]
9
10 #include <Wire.h>
11 #include <Adafruit_Sensor.h>
12 #include <Adafruit_MPU6050.h>
13 #include <Adafruit_HMC5883_U.h>
14 #include <Adafruit_BMP085.h>
15 #include <TinyGPSPlus.h>
16
17 // === Hardware Configuration ===
18 #define GPS_SERIAL Serial1
19 #define GPS_BAUD 9600
20
21 // === Sensor Objects ===
22 Adafruit_MPU6050 mpu;
23 Adafruit_HMC5883_Unified mag(12345);
24 Adafruit_BMP085 bmp;
25 TinyGPSPlus gps;
26
27 // === Calibration Parameters ===
28 struct SensorCalibration {
29     float accel_offset[3] = {0.0967, -0.3779, 0.4146};
30     float gyro_bias[3]     = {-0.037013, -0.024683, -0.007987};
31     float mag_offset[3]    = {18.35, -42.70, -15.82};
32     float mag_scale[3]     = {1.12, 0.95, 1.08};
33     float mag_declination = 1.16 * M_PI/180;
34     float sea_level_pressure = 1013.25;
35 } cal;
36
37 // === ROS2 Publishers ===
38 rclc_support_t support;
39 rcl_node_t node;
```

```

40 rclc_executor_t executor;
41
42 rcl_publisher_t imu_pub;
43 sensor_msgs__msg__Imu imu_msg;
44
45 rcl_publisher_t mag_pub;
46 sensor_msgs__msg__MagneticField mag_msg;
47
48 rcl_publisher_t gps_pub;
49 sensor_msgs__msg__NavSatFix gps_msg;
50
51 rcl_publisher_t baro_pub;
52 sensor_msgs__msg__FluidPressure baro_msg;
53
54 // === Orientation Variables ===
55 float pitch = 0.0, roll = 0.0, yaw = 0.0;
56 float q[4] = {0.0, 0.0, 0.0, 1.0}; // Quaternion (x, y, z, w)
57 const float ALPHA = 0.15; // Complementary filter coefficient
58 unsigned long prev_millis = 0;
59 const int PUBLISH_RATE = 5; // Hz
60
61 void setup() {
62     // === Hardware Initialization ===
63     Serial.begin(115200);
64     GPS_SERIAL.begin(GPS_BAUD);
65
66     // === Sensor Setup ===
67     if (!mpu.begin() || !mag.begin() || !bmp.begin()) {
68         Serial.println("Sensor initialization failed!");
69         while (1); // Halt on failure
70     }
71
72     mpu.setAccelerometerRange(MPU6050_RANGE_4_G);
73     mpu.setGyroRange(MPU6050_RANGE_250_DEG);
74     mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);
75
76     // === Micro-ROS Setup ===
77     set_microros_transports();
78     delay(2000);
79
80     // === ROS2 Node & Publishers ===
81     rcl_allocator_t allocator = rcl_get_default_allocator();
82     rclc_support_init(&support, 0, NULL, &allocator);
83     rclc_node_init_default(&node, "usv_sensor_node", "", &support);
84
85     // IMU Publisher
86     rclc_publisher_init_default(
87         &imu_pub, &node,
88         ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, Imu),
89         "/usv/imu_data"
90     );

```

```

91  setup_imu_message();
92
93  // Magnetic Field Publisher
94  rclcpp_publisher_init_default(
95      &mag_pub, &node,
96      ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, MagneticField),
97      "/usv/magnetic_field"
98  );
99  setup_magnetic_message();
100
101  // GPS Publisher
102  rclcpp_publisher_init_default(
103      &gps_pub, &node,
104      ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, NavSatFix),
105      "/usv/gps_data"
106  );
107  setup_gps_message();
108
109  // Barometer Publisher
110  rclcpp_publisher_init_default(
111      &baro_pub, &node,
112      ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, FluidPressure),
113      "/usv/barometer"
114  );
115
116  // === Executor Initialization ===
117  rclcpp_executor_init(&executor, &support.context, 0, &allocator);
118 }
119
120 void loop() {
121     rclcpp_executor_spin_some(&executor, RCL_MS_TO_NS(100));
122
123     unsigned long current_millis = millis();
124     if (current_millis - prev_millis >= 1000/PUBLISH_RATE) {
125         prev_millis = current_millis;
126
127         // === Read Sensor Data ===
128         sensors_event_t accel, gyro, mag_event;
129         mpu.getEvent(&accel, &gyro, NULL);
130         mag.getEvent(&mag_event);
131
132         // === Calibrate Sensors ===
133         float ax = accel.acceleration.x - cal.accel_offset[0];
134         float ay = accel.acceleration.y - cal.accel_offset[1];
135         float az = accel.acceleration.z - cal.accel_offset[2];
136         float gx = gyro.gyro.x - cal.gyro_bias[0];
137         float gy = gyro.gyro.y - cal.gyro_bias[1];
138         float gz = gyro.gyro.z - cal.gyro_bias[2];
139
140         float dt = (current_millis - prev_millis)/1000.0;
141         updateOrientation(ax, ay, az, gx, gy, gz, dt, mag_event);

```

```

142
143 // === Publish Messages ===
144 populate_imu_message(current_millis, ax, ay, az, gx, gy, gz);
145 rcl_publish(&imu_pub, &imu_msg, NULL);
146
147 populate_magnetic_message(current_millis, mag_event);
148 rcl_publish(&mag_pub, &mag_msg, NULL);
149
150 read_gps_data(current_millis);
151
152 // Barometer
153 float pressure = bmp.readPressure() / 100.0; // Convert Pa to
    hPa
154 baro_msg.header.stamp.sec = current_millis / 1000;
155 baro_msg.header.stamp.nanosec = (current_millis % 1000)
    *1000000;
156 baro_msg.fluid_pressure = pressure;
157 baro_msg.variance = 0.1; // Covariance
158 rcl_publish(&baro_pub, &baro_msg, NULL);
159 }
160 }
161
162 // === Helper Functions ===
163 void set_diagonal(double* arr, float val) {
164     arr[0] = (double)val;
165     arr[4] = (double)val;
166     arr[8] = (double)val;
167 }
168
169 void setup_imu_message() {
170     imu_msg.header.frame_id.data = "imu_link";
171     imu_msg.header.frame_id.capacity = 8;
172     set_diagonal(imu_msg.linear_acceleration_covariance, 0.01);
173     set_diagonal(imu_msg.angular_velocity_covariance, 0.02);
174     set_diagonal(imu_msg.orientation_covariance, 0.001);
175 }
176
177 void setup_magnetic_message() {
178     set_diagonal(mag_msg.magnetic_field_covariance, 0.1);
179 }
180
181 void setup_gps_message() {
182     gps_msg.header.frame_id.data = "gps";
183     gps_msg.position_covariance_type =
        sensor_msgs__msg__NavSatFix__COVARIANCE_TYPE_DIAGONAL_KNOWN;
184     set_diagonal(gps_msg.position_covariance, 1.0);
185 }
186
187 void updateOrientation(
188     float ax, float ay, float az,
189     float gx, float gy, float gz,

```

```

190 float dt,
191 sensors_event_t mag_event
192 ) {
193 // Accelerometer-based pitch/roll
194 float pitch_acc = atan2(ay, sqrt(ax*ax + az*az)) * 180/M_PI;
195 float roll_acc = atan2(-ax, sqrt(ay*ay + az*az)) * 180/M_PI;
196
197 // Complementary filter
198 pitch = ALPHA*pitch_acc + (1-ALPHA)*(pitch + gy*dt);
199 roll = ALPHA*roll_acc + (1-ALPHA)*(roll + gx*dt);
200
201 // Yaw from magnetometer with tilt compensation
202 float mx = (mag_event.magnetic.x - cal.mag_offset[0])*cal.
mag_scale[0];
203 float my = (mag_event.magnetic.y - cal.mag_offset[1])*cal.
mag_scale[1];
204 float mz = (mag_event.magnetic.z - cal.mag_offset[2])*cal.
mag_scale[2];
205 yaw = calculateHeading(mx, my, mz);
206
207 // Euler to Quaternion
208 float roll_rad = roll*M_PI/180;
209 float pitch_rad = pitch*M_PI/180;
210 float yaw_rad = yaw*M_PI/180;
211 q[0] = sin(roll_rad/2)*cos(pitch_rad/2)*cos(yaw_rad/2)
- cos(roll_rad/2)*sin(pitch_rad/2)*sin(yaw_rad/2);
212 q[1] = cos(roll_rad/2)*sin(pitch_rad/2)*cos(yaw_rad/2)
+ sin(roll_rad/2)*cos(pitch_rad/2)*sin(yaw_rad/2);
213 q[2] = cos(roll_rad/2)*cos(pitch_rad/2)*sin(yaw_rad/2)
- sin(roll_rad/2)*sin(pitch_rad/2)*cos(yaw_rad/2);
214 q[3] = cos(roll_rad/2)*cos(pitch_rad/2)*cos(yaw_rad/2)
+ sin(roll_rad/2)*sin(pitch_rad/2)*sin(yaw_rad/2);
215
216 }
217
218 float calculateHeading(float mx, float my, float mz) {
219 float roll_rad = roll*M_PI/180;
220 float pitch_rad = pitch*M_PI/180;
221 float Xh = mx*cos(pitch_rad) + my*sin(roll_rad)*sin(pitch_rad)
+ mz*cos(roll_rad)*sin(pitch_rad);
222 float Yh = my*cos(roll_rad) - mz*sin(roll_rad);
223 float heading = atan2(Yh, Xh) + cal.mag_declination;
224 return heading*180/M_PI;
225 }
226
227 void populate_imu_message(unsigned long timestamp, float ax,
float ay, float az, float gx, float gy, float gz) {
228 imu_msg.header.stamp.sec = timestamp / 1000;
229 imu_msg.header.stamp.nanosec = (timestamp % 1000)*1000000;
230
231 imu_msg.linear_acceleration.x = ax;
232 imu_msg.linear_acceleration.y = ay;
233

```

```

236     imu_msg.linear_acceleration.z = az;
237
238     imu_msg.angular_velocity.x = gx;
239     imu_msg.angular_velocity.y = gy;
240     imu_msg.angular_velocity.z = gz;
241
242     imu_msg.orientation.x = q[0];
243     imu_msg.orientation.y = q[1];
244     imu_msg.orientation.z = q[2];
245     imu_msg.orientation.w = q[3];
246 }
247
248 void populate_magnetic_message(unsigned long timestamp,
    sensors_event_t mag_event) {
249     mag_msg.header.stamp.sec = timestamp / 1000;
250     mag_msg.header.stamp.nanosec = (timestamp % 1000)*1000000;
251
252     float mx = (mag_event.magnetic.x - cal.mag_offset[0])*cal.
        mag_scale[0];
253     float my = (mag_event.magnetic.y - cal.mag_offset[1])*cal.
        mag_scale[1];
254     float mz = (mag_event.magnetic.z - cal.mag_offset[2])*cal.
        mag_scale[2];
255
256     mag_msg.magnetic_field.x = mx;
257     mag_msg.magnetic_field.y = my;
258     mag_msg.magnetic_field.z = mz;
259 }
260
261 void read_gps_data(unsigned long timestamp) {
262     while (GPS_SERIAL.available()) gps.encode(GPS_SERIAL.read());
263     if (gps.location.isUpdated()) {
264         gps_msg.header.stamp.sec = timestamp / 1000;
265         gps_msg.header.stamp.nanosec = (timestamp % 1000)*1000000;
266
267         gps_msg.latitude = gps.location.lat();
268         gps_msg.longitude = gps.location.lng();
269         gps_msg.altitude = gps.altitude.meters();
270         rcl_publish(&gps_pub, &gps_msg, NULL);
271     }
272 }

```

Listing 3.1: Arduino source code for the USV ROS2 sensor node

Bibliography

- [1] T. I. Fossen and T. Perez, “Kalman filtering for positioning and heading control of ships and offshore rigs,” *IEEE Control Systems Magazine*, vol. 29, no. 6, pp. 32–46, 2009.
- [2] J. Smith *et al.*, “Marine robotics for offshore infrastructure inspection: A review,” *International Journal of Maritime Engineering*, 2022.
- [3] *C-Worker USV Product Specifications*, Teledyne Marine, 2023. [Online]. Available: <https://www.teledynemarine.com>
- [4] R. Johnson, “Cost analysis of unmanned surface vehicles in environmental monitoring,” *Marine Technology Society Journal*, 2023.
- [5] L. Chen *et al.*, “Challenges in usv navigation for confined waters,” *IEEE Journal of Oceanic Engineering*, 2021.
- [6] A. Kumar *et al.*, “Sensor fusion for low-cost usvs: A comparative study,” *Sensors*, 2022.
- [7] Y. Wang *et al.*, “Robust control strategies for underactuated usvs,” *Autonomous Robots*, 2023.
- [8] H. Zhang *et al.*, “Adaptive smc for usv trajectory tracking,” *Ocean Engineering*, 2022.
- [9] S. Lee *et al.*, “EKF-based navigation in GPS-denied environments,” *IEEE Transactions on Robotics*, 2023.
- [10] Arduino, “Arduino Due,” <https://docs.arduino.cc/hardware/due>, 2023, arduino Official Documentation.
- [11] F. Sotelo-Torres, L. V. Alvarez, and R. C. Roberts, “An Unmanned Surface Vehicle (USV): Development of an Autonomous Boat with a Sensor Integration System for Bathymetric Surveys,” *Sensors*, vol. 23, no. 9, p. 4420, Apr. 2023.
- [12] SunFounder, “GY-87 IMU Module,” https://docs.sunfounder.com/projects/elite-explorer-kit/en/latest/basic_projects/09_basic_gy87.html, 2023, sunFounder Elite Explorer Kit Documentation.
- [13] u-blox, “NEO-M8 Series Data Sheet,” <https://www.u-blox.com/en/product/neo-m8-series>, 2020.
- [14] *GY-87 IMU Module Documentation*, SunFounder, 2023. [Online]. Available: https://docs.sunfounder.com/projects/elite-explorer-kit/en/latest/basic_projects/09_basic_gy87.html
- [15] “Gy-87 10 dof sensor module,” RobotPark, 2023. [Online]. Available: <https://www.robotpark.com/GY-87-10-DOF-Sensor-Module>

- [16] “Gy-87 10dof mpu6050 hmc5883l bmp180 sensor module,” RoboticsBD, 2024. [Online]. Available: <https://store.roboticsbd.com/sensors/382-gy-87-10dof-mpu6050-hmc5883l-bmp180-sensor-module-robotics-bangladesh.html>
- [17] ElectroPeak, “Interfacing gy-87 10dof imu module with arduino,” 2021. [Online]. Available: <https://electropeak.com/learn/interfacing-gy-87-10dof-imu-mpu6050-hmc5883l-bmp085-module-with-arduino/>
- [18] u-blox AG, “Neo-m8 series product brief,” 2024. [Online]. Available: <https://www.u-blox.com/en/product/neo-m8-series>
- [19] —, “Neo-m8-fw3 data sheet,” u-blox, Tech. Rep., 2022. [Online]. Available: https://content.u-blox.com/sites/default/files/NEO-M8-FW3_DataSheet_UBX-15031086.pdf
- [20] u-blox Community. (2022) Max-m8 series navigation update rate. [Online]. Available: <https://portal.u-blox.com/s/question/0D52p0000CsXzb5CQC/maxm8-series-max-navigation-update-rate>
- [21] L. N. et al. (2018) Ublox neo-m8n accuracy and limits. [Online]. Available: https://www.researchgate.net/figure/Ublox-Neo-M8N-accuracy-and-limits-for-various-operational-modes_tbl2_320092614
- [22] H. Kutluca, “ROS 2 Architecture,” <https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776>, Apr. 2022, medium.
- [23] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley, 2011.
- [24] A. J. Healey and D. Lienard, “Multivariable sliding mode control for autonomous diving and steering of unmanned underwater vehicles,” *IEEE Journal of Oceanic Engineering*, vol. 18, no. 3, 1993.