Dynamics assisted Dead Reckoning for Marine Vessels in a high-fidelity Simulator using a low-cost IMU

Dissertation submitted in partial fulfillment of the requirements
for the award of the degree of

Dual Degree (B.Tech + M.Tech.)

by

Rishabh Sharma

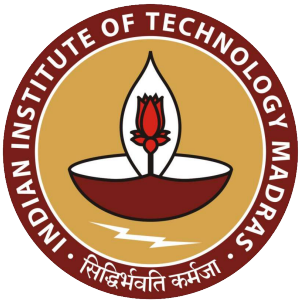
(Roll No. NA20B051)

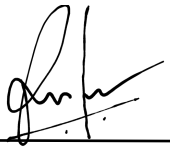
Under the Supervision of

Dr. Abhilash Somayajula

SIGNED VIA ILOVEPDF
1A52B03E-609F-4508-AED7-01B0ABF8AEE5

Dr. Abhilash Somayajula





Rishabh Sharma

Department of Ocean Engineering

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

Chennai - 600036, India

Jan, 2025

Declaration

I declare that this written submission represents my ideas in my own words. Where others' ideas and words have been included, I have adequately cited and referenced the original source. I declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated, or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will cause disciplinary action by the Institute and can also evoke penal action from the source which has thus not been properly cited or from whom proper permission has not been taken when needed.

.....

Rishabh Sharma
Roll No.: NA20B051
Date: 9/05/2025
Place: IIT Madras

Abstract

About 95% of the ocean still remains unexplored with a vast wealth of natural resources and new life waiting to be tapped in. To practically cover this vast volume, a large fleet of Autonomous Marine Vehicles (AUVs and USVs) will be required. The fundamental piece of the puzzle precursor to any autonomous robotics task is Navigation, i.e knowing where the robot is. Traditional navigation sensors used in marine vessels are prohibitively expensive making them an unexpendable asset. Our proposed method is expected to bring down the cost of navigation considerably making marine vessels expendable for large operations. In this paper, Dead Reckoning using a low-cost IMU sensor and the AUV dynamics is explored. Testing on an actual vessel is costly and time consuming, thus a simulator which mimics the real world dynamics is required. For this, an in-house high-fidelity and light weight simulator is developed and modularized to scale its use with any kind of marine vessel.

Index Terms — Deep Learning, Dead Reckoning (DR), Autonomous Underwater Vehicles(AUVs), GNSS-denied Navigation, Inertial Measurement Unit (IMU), Unmanned Surface Vehicles (USVs), Marine Simulator

Contents

List of Figures

Abbreviations

DR	**D**ead **R**eckoning
IMU	**I**nertial **M**easurment **U**nit
AUV	**A**utonomous **U**nderwater **V**ehicles
DVL	**D**oppler **V**elocity **L**og
VO	**V**isual **O**dometry
EKF	**E**xtended **K**almam **F**ilter
RNN	**R**ecurrent **N**eural **N**etworks
LSTM	**L**ong **S**hort **T**erm **Memory**
AHRS	**A**ttitude and **H**eading **R**eference **S**ystem
GPS	**G**lobal **P**ositioning **S**sytem
GNSS	**G**lobal **N**avigation **S**atellite **S**ystems
MEMS	**M**icro **E**lectro **M**echanical **S**ystems
UAV	**U**nmanned **A**erial **V**ehicles
USV	**U**nmanned **S**urface **V**ehicle
ROS	**R**obot **O**perating **S**ystem
FNN	**F**eed-forward **N**eural **N**etwork

Chapter 1

Introduction

With the advancement of technology and the need for deep ocean exploration, the demand for Autonomous Marine Vessels (AUVs and USVs) has significantly increased. They are being used for a spectrum of reasons such as deep ocean exploration, survey missions, search and rescue operations, oil and gas exploration and for defense and military purposes. One key factor in making these missions successful is to have robots with robust autonomous capabilities. Out of the three major sections of autonomy, i.e. Guidance, Navigation and Control, Navigation is a precursor to the other two since it is essential for a robot to know where it is. This knowledge then helps the robot make decisions on its next goal position (Guidance) and actuate accordingly (Control). In contrast to land robots that generally get their location via GNSS and other sensor fusion techniques, underwater robots face a major challenge in localization since the GNSS signals do not propagate underwater. To tackle this, an underwater GPS system using an array of acoustic transducers in the ocean or using highly accurate motion capture systems in lab-based environments is widely used. However, these methods are very expensive and time-consuming to deploy and maintain. Moreover, localization techniques that depend on such external hardware systems are prone to spoofing, jamming, and electronic warfare. Thus, there is a need to develop robust localization

methods using sensors which do not rely on external electronic signals. Dead Reckoning is a method that predicts a robot's position and attitude using on-board sensors to calculate the relative motion from a known starting position. Widely used techniques for underwater dead-reckoning include using highly accurate Inertial Measurement Units (IMUs), Doppler Velocity Log (DVL) speed sensors and fusing them using classical state estimation algorithms. Recently robust visual odometry (VO) algorithms have also been developed [4]. A standalone VO mobile application was developed for my BTech Project (*Standalone Vision Based Localization of ships for Wavebasin(GPS-denied environment)*) which showed promising results compared to GPS as shown in Fig 1.1. However DVL and VO methods are susceptible to various environmental factors and disturbances. DVL dead-reckoning works best when it is used in bottom lock mode. The bottom lock mode is when the AUV is operating near the seabed, and the DVL calculates the speed relative to the seabed (stationary). VO algorithms rely on extracting same features from two consecutive images to calculate pose. Unless the robot is near the sea bed or near some debris, there aren't many features to extract and hence VO algorithms would result in drifting. Traditionally when an AUV is deployed for deep sea exploration, it uses dead reckoning technique relying on IMU measurements and DVL in water lock mode till the AUV reaches the seabed and then uses DVL in bottom lock mode to get a more accurate speed measurement. Due to inherent sensor bias and noise in IMU sensors, and the unreliable speed estimates of DVL in water lock mode, the calculated pose could be off by a lot than the ground truth. There exists IMU sensors with very less bias and noise characteristics but they are expensive and bulky. This makes the AUV an expensive asset and thus it would be uneconomical if the AUVs are to be used as expendables in a particular mission or are not retrieved due to system or communication failure.

Before deploying any GNC application, it has to be extensively tested and tuned. Doing this in a real world scenario is often costly, time consuming and often unsafe in the case of marine vessels. Thus need of a high fidelity simulator which mimics the real world

FIGURE 1.1: VO mobile app result (BTech project). Green line is position from Visual Odometry and Blue dots are position from GPS

dynamics and condition is required. Based on our findings there are a few open source marine vessel simulators out of which *HoloOcean* stands out [5]. However, *HoloOcean* requires *Unreal Engine* to configure and add new vessels along with the caveat that they do not provide the 3D world rendering and thus one has to build their own from scratch. This is extremely time consuming and resource intensive (*Unreal Engine* is a huge software (60GB+) and requires good compute). Thus we have made a light weight ROS2 based modular high-fidelity simulator, *Panisim*. The simulator is fully opensource and the documentation is available at [6]

1.0.1 Literature Review

1.0.1.1 AUV Dead Reckoning

A lot of recent research is going on in developing algorithms to use low-cost IMU sensors for dead reckoning. Sabet et al [7] have developed a DR navigation system for an AUV

using a low-cost IMU and the surge dynamic model of the AUV. They use an Extended Kalman Filter (EKF) to perform fusion of the dynamic model and sensor measurement values. They are able to achieve a DR system with 8% error in position during an 18min GPS outage using the AUV velocity model and IMU. Recently due to the advancement of data driven approaches, people have exploited the temporal learning capabilities of deep learning models like Recurrent Neural Netowrks (RNNs) and Long Short Term Memory Networks (LSTMs) to learn the AUV dynamics and sensor characteristics to perform DR. Edoardo et al [8] have developed a DR system using LSTMs networks. This uses generalized forces and previous velocities to estimate the surge and sway body-fixed velocities without relying on a DVL. However the DVL is still used to estimate the depth. Also their trained network gives good prediction as long as lawnmower paths are performed. Mu et al [9] have developed a navigation system using Hybrid Recurrent Neural Networks (RNN). Their algorithm relies on a range of sensors including DVL, Pressure sensor, Attitude and Heading Reference System (AHRS) and GPS. However, in certain situations the method is unstable. Song et al [10] have developed a neural network based navigation for AUVs in fast changing environments. They introduce a six phase method to train and use a model for dead reckoning using a single IMU and a depth sensor. However the dead reckoning carried out is only in the surge and heave direction. Saksvik et al [11] have used a RNN to predict the relative horizontal velocities of an AUV using data from IMU, pressure sensor and control inputs. However, IMU sensor noise was not considered in the simulation results.

1.0.1.2 IMU Noise

An Inertial Measurement Unit, is a sensor which measures the linear acceleration (accelerometer) and angular velocity (gyroscope) of a body. Modern day IMU units also come with a magnetometer which measures the strength of the magnetic fields in different directions. The IMU's which are generally used for small Robotics applications

like AUVs, UAVs are MEMS IMU. Fig 1.2 shows the construction of an accelerometer. There are moving and stationary electrodes which together form a capacitor. As the body moves, the electrodes move closer or father away from each other which changes the capacitance. The capacitance can be related to the force acting on the electrodes and thus one can measure the acceleration acting on the body. A gyroscope also has a similar construction as shown in Fig 1.3 except that it works on the principle of Coriolis effect. There is an oscillating mass at the center which moves under an application of angular velocity changing the capacitance which can be measured. Although the MEMS IMU sensors are widely found today in robotics and smartphones, they all have a drawback of inherent sensor noise characteristics. The noise can be divided into two components viz White noise and Sensor Bias. White noise can be thought of an irregular, unpredictable oscillating signal. Sensor Bias is an offset in the measurement which is also observed to follow a slowly varying random walk. Thus, an acceleration measurement from the sensor can be written as

$$\tilde{a}(t) = a(t) + b(t) + n(t) \tag{1.1}$$

where, $\tilde{a}(t)$ is the acceleration measurement, $b(t)$ is the bias and $n(t)$ is the white noise and $a(t)$ is the true acceleration.

The sources for the IMU noise can be thermal noise, electronic interference, mechanical noise, quantization noise (converting from analog signal to digital signal), calibration errors and manufacturing defects. The Bias in IMUs is also observed to be temperature dependent. The consequences of the inherent sensor noise are critical in the sense that an IMU can alone not be used for Dead Reckoning without the aid of a more accurate sensor such as GPS or DVL. Double Integation of acceleration to get the position also amplifies the noise and thus the integration starts to drift. This can be seen in Fig 1.4. Even though the IMU is stationary, the double integration starts to drift as time progresses and common filtering techniques such as low pass filtering does not help. Now, that we know

that the IMU has noise, it is necessary to devise a method to compare different IMUs and their noise characteristics.

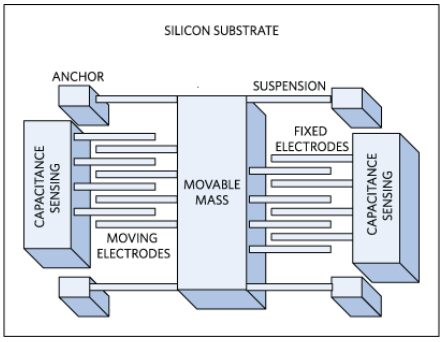


FIGURE 1.2: Accelerometer working (source: analog.com)

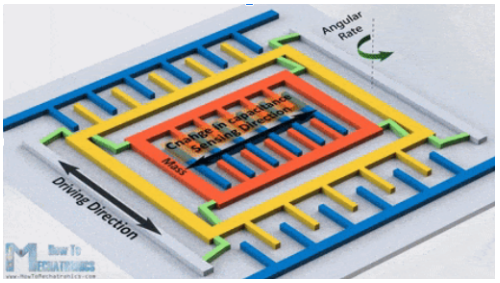


FIGURE 1.3: Gyroscope working (source: howtomechatronics.com)

Given a history of noisy sensor measurements, one quick way to get a value close to the true measurement would be take the average of the measurement over a period of time. As we increase the data points and the time averaging period, the noise variance must continuously decrease. However, this fails in our case of IMU measurements since the Bias term drifts (random walk). This can be clearly seen in the Fig 1.5. We observe that in the case of a signal with drift, the noise variance tends to become constant as number of

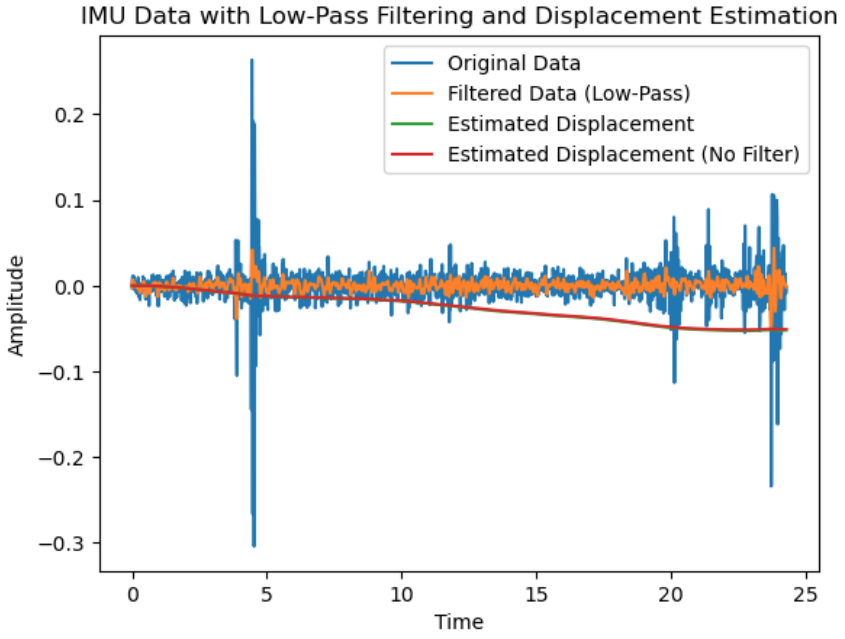


FIGURE 1.4: Accelerometer Double Integration

averaged samples increases. To overcome this drawback of the signal averaging technique, we use the *Allan variance* method.

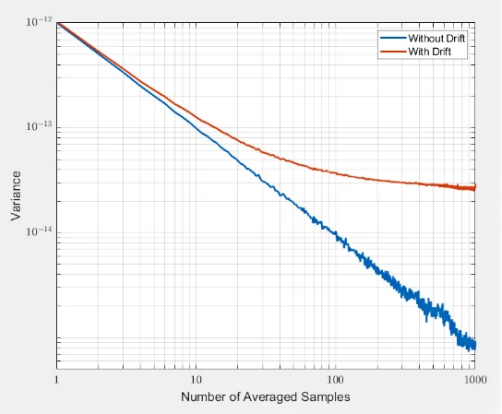


FIGURE 1.5: Signal Averaging (source: [1])

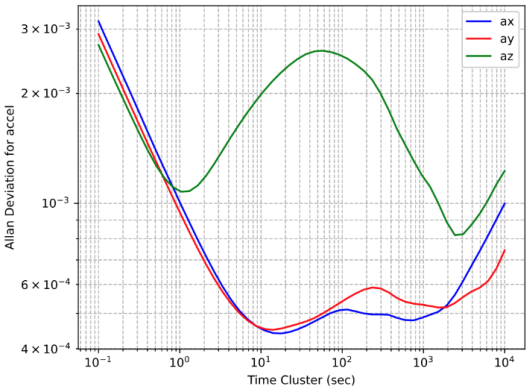


FIGURE 1.6: Allan Deviation for LSM6DSR (source: [2])

Allan variance has two methods viz Non-overlapping Allan Variance and Overlapping Allan Variance. Here we will discuss Non-overlapping Allan variance. Consider the Gyroscope measurements against Time as shown in Fig 1.7. There are total N samples from the gyroscope output $\Omega(t)$ at multiples of the sampling time t_0. These N samples are divided into K disjoint groups of equal length where each group (or cluster) has n samples. If the sampling time is t_0, the total time duration of each group is $T = nt_0$.The average of the kth cluster is denoted by $\bar{\Omega}_k(t)$. Then, the Allan Variance is given as:

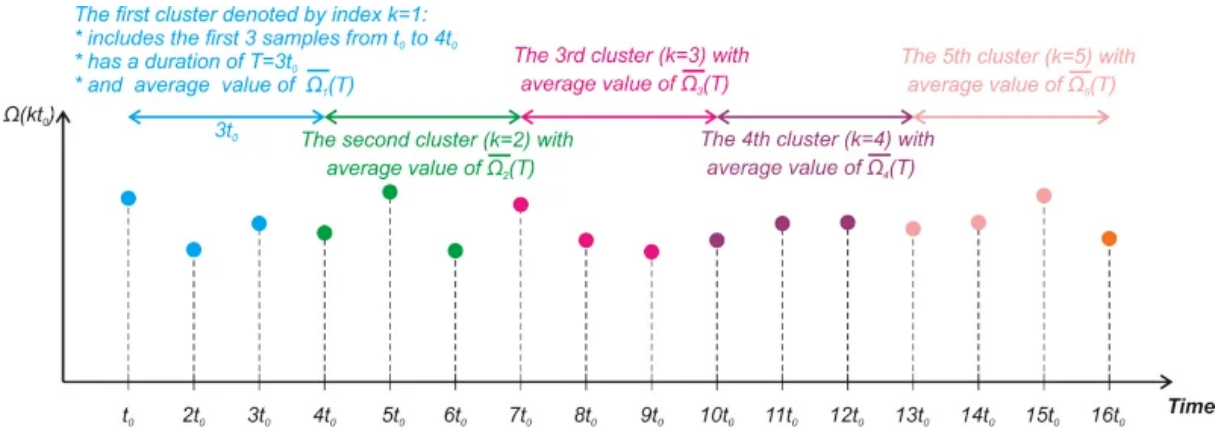


FIGURE 1.7: Gyroscope Output (source:[1])

$$\sigma^2(T) = \frac{1}{2(k-1)} \sum_{k=1}^{k-1} (\bar{\Omega}_{k+1}(T) - \bar{\Omega}_k(T))^2 \tag{1.2}$$

A typical Allan Deviation curve against the cluster time is shown as in Fig 1.8. The different slopes in the plot give a quantified idea about the different noise in the the sensor. A typical plot for the allan deviation of a real sensors (STM LSM6DSR IMU), common in most smartphones is as shown in Fig 1.6. A comprehensive study on this is done by Suvorkin et al [2].

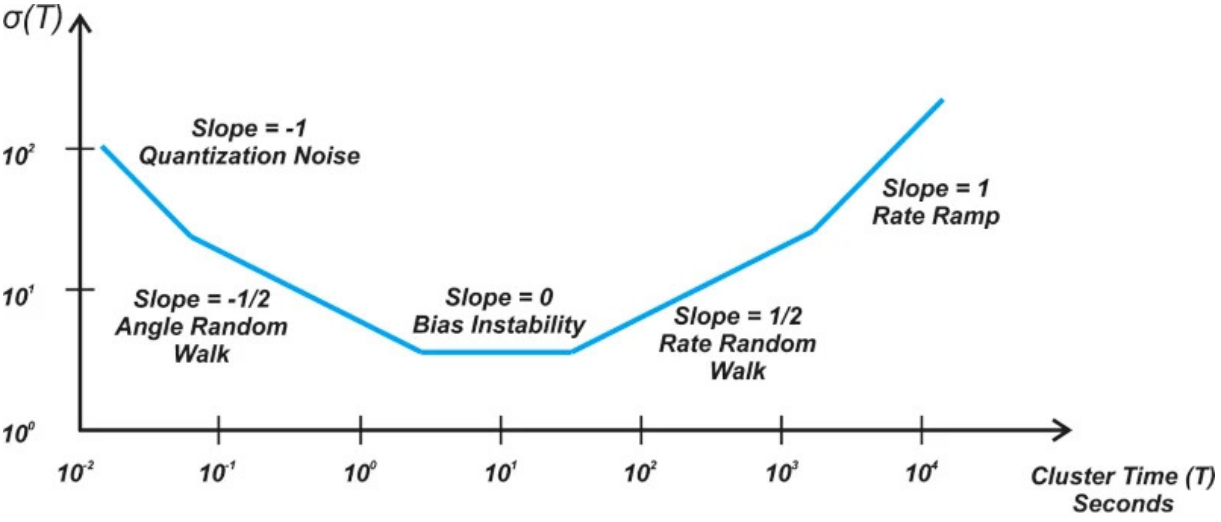


FIGURE 1.8: Allan deviation curve (source:[1])

1.0.2 Simulation

Field testing of AUVs is very expensive and risky. For the development of a data-driven Dead Reckoning model or the AUV system identification, a huge amount of sensor data will be required. Thus an effective simulation environment becomes necessary. This simulation environment needs to replicate the sensors accurately along with the sensor noise. *HoloOcean* is an open source high-fidelity Underwater Robotics Simulator developed by the Field Robotic System Lab at Brigham Young University. It is based on Unreal Engine 4, a widely used 3D game development engine. The simulator offers a wide variety of physics based sensors which can be customized to add noise. The simulator also allows one to write the dynamics of robot from scratch giving one full control over the simulation. A simulation still from the simulator on our MayvMini AUV (developed by MAV Lab, IIT Madras) is as shown in Fig 1.10.

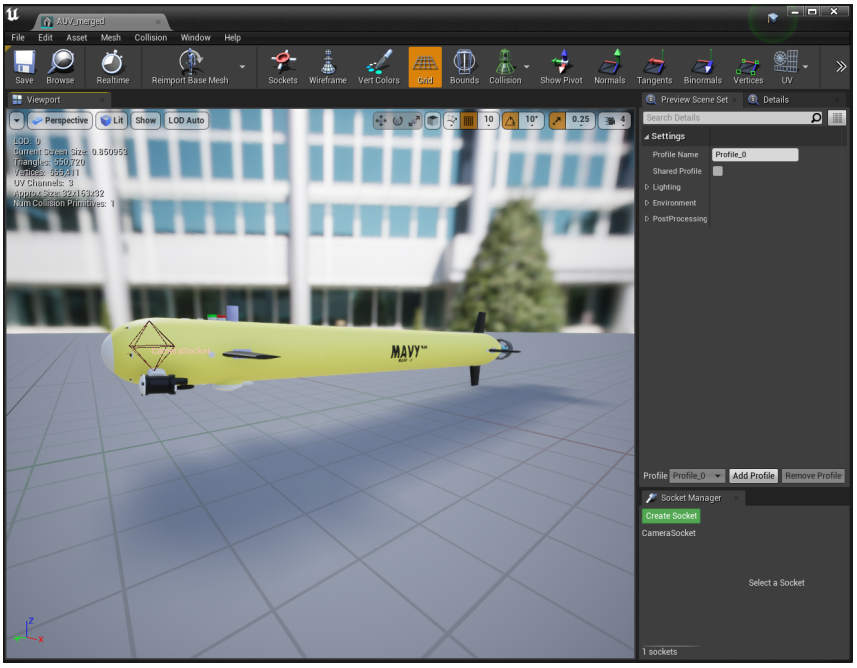


FIGURE 1.9: Mavymini (from MAV LAB) AUV in HoloOcean

However for loading custom AUVs as required in our case, one needs Unreal Engine which may be a very heavy software to run on some systems. HoloOcean does not provide the source code for the worlds shipped with the package, thus if one wants to just add their own custom AUV, one would also have to design and make their own world. Thus, an in-house light weight and high-fidelity marine simulator based on Python and ROS2 is developed, Panisim [6]. It is a lightweight simulator which is packaged in docker and does not require extensive system setup to run. The GUI is lightweight and built for the web.

1.0.3 Objective

A low-cost data-driven DR for marine vessels, which just relies on a single low-cost IMU sensor, is proposed. The method will be compared against an Extended Kalman Filter DR technique using IMU measurements and AUV dynamic model. The setup will be tested in a simulated environment with simulated AUV dynamics and IMU measurements with

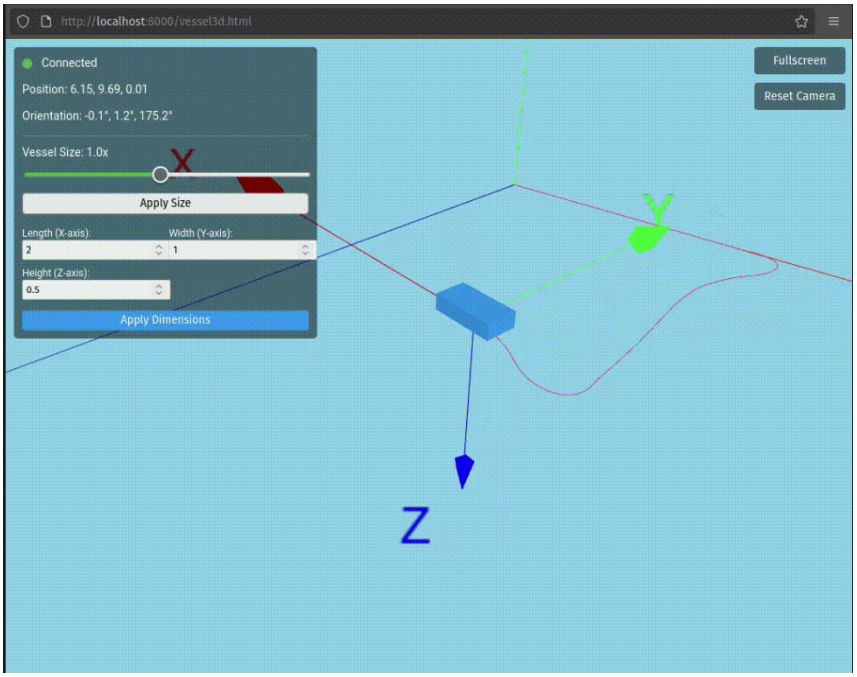


FIGURE 1.10: Visualization from Panisim simualtor

noise and bias. An inhouse highly modular simulator is developed based on Python3, ROS2.

1.0.4 Scope

This research will be bounded to the simulations and experiments on the *Sookshma vessel* built by the Marine Autonomous Vehicle Laboratory (MAVLAB) in the department of Ocean Engineering and Naval Architecture, IIT Madras. The AUV dynamics derived will be via simulations and analytical results rather than experimentation. The simulation software developed in-house will be used for simulations and hardware in the loop integrations. The new simulator will be documented.

Chapter 2

Equations of Motion

2.1 Introduction

The motion of any rigid body can be described by kinematics and dynamics. Kinematics describes how the pose (position+orientation) and velocity of a body change with time without considering the forces and moments (which will together be mentioned as generalized forces) acting on the body. Dynamics, on the other hand, describes the motion of the body when an external generalized force is applied. Conventionally, the dynamics equation describes the motion of the center of gravity of the body. The chapter is arranged as follows, in section 2.2, we will set some mathematical preliminaries required to decipher the equation of motions, then in section 2.3 we will see the kinematics equation of motion along with the reference frames used. Finally in section 2.4 we will see the equation which defines how the velocity of the body changes when a generalized force is applied on the marine vessel.

2.2 Mathematical Preliminaries

The sets of real numbers is denoted by $\mathbb{R}$. We denote by $\mathbb{R}^n$ the n-dimensional Euclidean space. The Euclidean norm of a vector $x \in \mathbb{R}^n$ is denoted by $\|x\|$. The *n-by-n* identity matrix is denoted by I_n and *n-by-m* zeros matrix is denoted by $0_{n\times m}$.

Let $\{I\}$ be an inertial frame and $\{B\}$ be a frame attached to a rigid body. The matrix $R_B^I \in \mathrm{SO}(3)$ denotes the rotation of frame $\{B\}$ with respect to frame $\{I\}$.

$$\{I\} = R_B^I\{B\} \tag{2.1}$$

In general, the rotation of any frame {A} with respect to frame {B} is denoted by R_A^B.

For each $x = [x_1, x_2, x_3]^T \in \mathbb{R}^3$, we define $x^\times$ as a skew-symmetric matrix given by

$$x^\times = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \tag{2.2}$$

The rotation matrix R for a ZYX Euler angle sequence, with the angles ϕ (roll), θ (pitch), and ψ (yaw), is given by:

$$R = R_z(\psi)R_y(\theta)R_x(\phi) \tag{2.3}$$

where $R_z(\psi)$, $R_y(\theta)$, and $R_x(\phi)$ are the rotation matrices about the z-, y-, and x-axes, respectively:

$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix},\ R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix},\ R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}. \tag{2.4}$$

Substituting these into the equation, the complete rotation matrix R is:

$$R = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}. \quad (2.5)$$

2.3 Kinematics

2.3.1 Coordinate Systems

When calculating the velocity of a body in motion, different observers will have different answers depending on their *frame of reference*. For example, for a person sitting inside a car, the velocity of the car is zero, and for a person standing on the road, the velocity of the moving car matches the reading of the speedometer in the car. Thus, the frame of reference or more formally, defining the coordinate system while writing the equation of motion is crucial. There are various frames of reference which are used in Robotics and Marine Navigation, viz, *Earth Centered Inertial (ECI)*, *Earth Centered Earth Fixed (ECEF)*, *North East Down (NED)*, *East North Up* etc. Below we will define the coordinate systems used in this paper.

2.3.1.1 The Global Coordinate System

Global Coordinate System (GCS) (a.k.a the world frame), is a fixed coordinate system (inertial frame) with respect to which the motion of a moving body is defined. In our case the GCS which we will be using is the North East Down (NED) frame. The origin of the NED frame is defined as the intersection point of a tangential plane to the surface of the Earth as shown in fig 2.1. This planar approximation of the surface of the Earth is valid within a square of length 10 km with the center as the tangential point. In the NED

frame, the x axis points North, y axis points East and the z axis points Down, towards the center of the Earth.

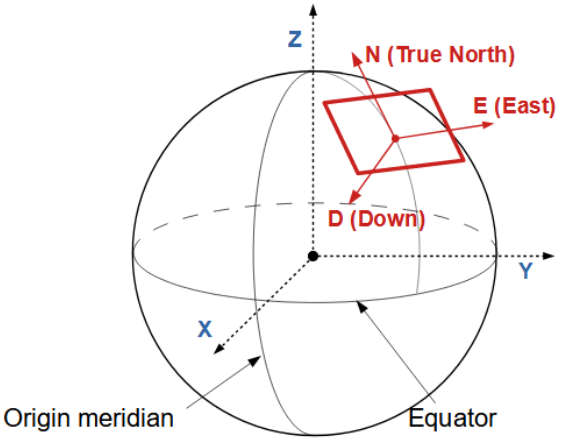


FIGURE 2.1: NED Frame of Reference

2.3.1.2 The Body Coordinate System

Body Coordinate System (BCS) (a.k.a the body frame), is a coordinate system fixed to the origin of the moving body, but moves with respect to the GCS. Since the BCS is a moving frame, it is a non-inertial frame of reference. As we will see in the dynamics equation, the non-inertial frame gives rise to Coriolis forces. Conventionally for AUVs, the body frame x axis points to the forward of the AUV, the y axis point to the starboard (right side when looked from behind) of the AUV and the z axis points down as shown in fig 2.2. The origin of the Body frame is denoted as [CO].

2.3.2 Euler Angles

The 3D position of an AUV can be represented by the x, y and z coordinates of [CO] in the GCS frame. But the AUV exhibits six degrees of freedom since it can rotate about its individual axes. As shown in Fig 2.2, rotation about positive x axis (counter-clockwise) is called roll (ϕ), about positive y axis is called pitch (θ) and about positive z axis is called

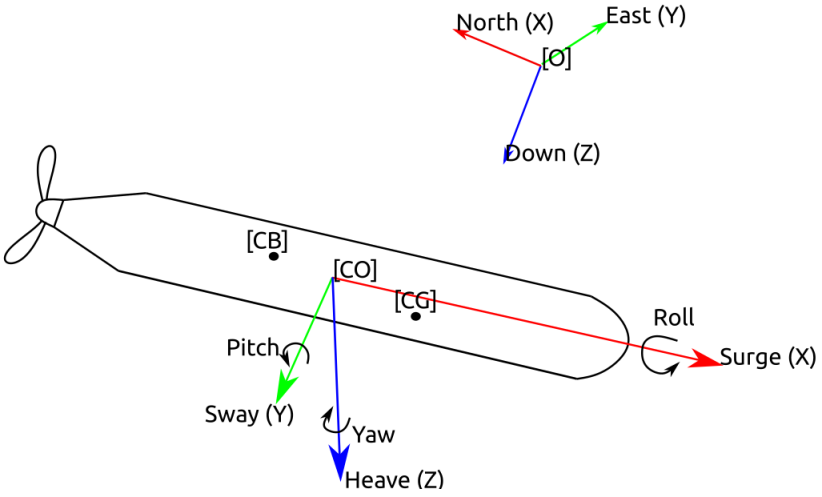


FIGURE 2.2: Body Frame and Euler Angles

yaw (ψ). These axes are the axes of the Body-Fixed Coordinate frame and the angles are called as Euler Angles. From these Euler Angles, one can derive a linear transformation, more formally a rotation matrix to transform a coordinate from one frame of reference to another as described in section 2.2.

Euler angles suffer from a flaw called a Gimbal Lock which occurs when during rotation, two axes align with each other leading to a loss of degree of freedom. A good visualization of the Gimbal lock can be seen at [12]. Also, the inverse problem of getting the Euler angles from the Rotation Matrix results in two different solutions. To avoid these problems, a widely used mathematical representation of rotations in space is of Quaternions. A Quaternion is a four dimensional vector which uniquely defines a rotation in space and does not suffer the shortcomings of Euler Angles. A good introduction to Quaternions can be found in the article by Shuster [13]. However Quaternions introduce an extra variable to be taken care of and the mathematics becomes complex for practical implementation in Kalman Filters (Multiplicative Kalmans Filters are used in that case). Since our AUV is unlikely to pitch by 90 degrees and thus avoid a gimbal lock situation, in this paper, we will be following rotation representation with Euler Angles. However, if this was being done for a space craft, such as a rocket, Gimbal Lock is inevitable and thus Quaternions must be used.

2.3.3 Kinematic Equations

Building on the preliminaries above, we will now ensemble and write the Kinematic Equations of Motion. Let the position vector of the AUV from the origin of GCS (NED) to the origin of the BCS expressed in GCS be defined as shown in Eq (2.6)

$$\{r_0\} = \{\eta_1\} = \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^T \tag{2.6}$$

Let the Euler angles be defined as a vector as shown in Eq (2.7) and explained in section 2.3.2. Each of the angles sequentially denotes the rotations from GCS through the intermediate frames to the BCS. (ZYX rotation as defined in 2.2)

$$\{\eta_2\} = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^T \tag{2.7}$$

The total pose vector $\{\eta\}$ is defined as the concatenation of $\{\eta_1\}$ and $\{\eta_2\}$ as shown in Eq (2.8)

$$\{\eta\} = \begin{bmatrix} \{\eta_1\}^T & \{\eta_2\}^T \end{bmatrix}^T = \begin{bmatrix} x_0 & y_0 & z_0 & \phi & \theta & \psi \end{bmatrix}^T \tag{2.8}$$

Let the translational velocity of the vehicle expressed in BCS be defined as in Eq (2.9)

$$\{\vec{v}\} = \{\nu_1\} = \begin{bmatrix} u & v & w \end{bmatrix}^T \tag{2.9}$$

Similarly, the angular velocity of the vehicle in BCS is defined as in Eq (2.10)

$$\{\vec{\omega}\} = \{\nu_2\} = \begin{bmatrix} p & q & r \end{bmatrix}^T \tag{2.10}$$

The total velocity vector $\{\nu\}$ is defined as the concatenation of $\{\nu_1\}$ and $\{\nu_2\}$ as shown in (19).

$$\{\nu\} = \begin{bmatrix} \{\nu_1\}^T & \{\nu_2\}^T \end{bmatrix}^T = \begin{bmatrix} u & v & w & p & q & r \end{bmatrix}^T \tag{2.11}$$

Now $\frac{d}{dt}\{\eta\} = \{\dot{\eta}\}$ represents the time derivative of the position vector of BCS origin expressed in the GCS. Since GCS is an inertial frame of reference, $\{\dot{\eta}_1\}$ is the translational velocity of the BCS origin with respect to GCS. The translational velocity of BCS origin expressed in BCS is given by $\{\nu_1\}$. These two vectors are related through Eq (2.12)

$$\{\dot{\eta}_1\} = R\{\nu_1\} \tag{2.12}$$

The angular velocity and the derivative of the Euler angles can also be related by Eq (2.13). Note that J is not the same rotation matrix as used in Eq (2.12). This matrix is called as the Euler Rate Matrix.

$$\{\dot{\eta}_2\} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \frac{s_1 s_2}{c_2} & \frac{c_1 s_2}{c_2} \\ 0 & c_1 & -s_1 \\ 0 & \frac{s_1}{c_2} & \frac{c_1}{c_2} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} = J\{\nu_2\} \tag{2.13}$$

Combining Eq (2.12) and Eq (2.13), we get the Kinematics Equation of Motion as (2.14).

$$\{\dot{\eta}\} = \begin{bmatrix} \dot{\eta}_1 \\ \dot{\eta}_2 \end{bmatrix} = \begin{bmatrix} R(\eta_2) & 0 \\ 0 & J(\eta_2) \end{bmatrix} \{\nu\} \tag{2.14}$$

This concludes the Kinematics Equation of Motion which describe the motion of the body without considering the generalized forces acting on the body.

2.4 Dynamics

From the previous section 2.3, we now have a relation between the state variables (position and angles) and their velocities. Now we need an equation which governs how the AUV velocity changes with time. This is the equation which the Dynamics of the AUV will give. The change in velocity with time is essentially the acceleration and acceleration is a result of a force acting on a body. For an AUV, the elements which apply a generalized force (Force and Moment) on the body are as listed

- **Inertial:** Inertial forces arise from an object's mass and its acceleration. The moving mass of the AUV, also moves a certain amount of water with it and this contributes to an additional mass felt by the system called as *Added Mass*.
- **Coriolis or Centripetal Forces:** The Coriolis force arises due to rotation frame of reference, in our case due to the rotation between the GCS and the BCS frame.
- **Gravity and Buoyancy:** The forces due to gravity and buoyancy (Archimedes Principle) are crucial as their combination determines whether the AUV will be stable underwater and not capsize under small disturbances.
- **Fluid Forces (Damping):** When the AUV moves through water, it observes a resistance force due to the viscous effects of water. This force is directly proportional to the velocity squared of the AUV. This force is called as the Damping force and we will consider these forces through hydrodynamic coefficients.
- **Due to Control Surface (Fins):** Control surfaces, such as fins and rudder are used to stabilize and maneuver vehicles. They influence the flow of fluid around the body and help in controlling direction and attitude. The AUV considered in this paper consists of four fins, two each on the forward and aft and two rudders on the aft.

- **Propulsion Force:** Propulsion force is the force due to the thrusters which overcomes the resistance due to drag and accelerates the AUV forward.
- **Environmental Forces:** External environmental factors such as wind, waves, currents, can significantly impact the motion of marine vessels. However these forces are not considered in the scope of the present work.

2.4.1 The Dynamic Equation

In robotics, the matrix form of the dynamic equation as in Eq (2.15) is widely used.

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = \tau \tag{2.15}$$

Drawing inspiration from Eq (2.15), Fossen (1991) introduced a similar matrix notation for Dynamics of Marine Vehicles as shown in Eq (2.16)

$$M\dot{v} + C(v)v + D(v)v + g(\eta) = \tau + \tau_{wind} + \tau_{wave} \tag{2.16}$$

Now, we will expand on each term in Eq (2.16) and complete the dynamics for our marine vessel/

2.4.1.1 The Mass Matrix

The Mass matrix (M) of a marine vessel consists of two components viz the Rigid Body Mass Matrix (M_{RB}) and the Added Mass Matrix (M_A). The Rigid Body Mass matrix about the Center of Gravity is given as in Eq (2.17)

$$M_{RB} = \begin{bmatrix} mI_3 & 0_{3\times3} \\ 0_{3\times3} & I_g \end{bmatrix} \tag{2.17}$$

where, I_g represents the inertia due to rotation motion about CG. However, we will be writing the equation of motion about the BCS (CO). Thus using parallel axis theorem, we get the Rigid Body Mass Matrix about CO as

$$M_{RB} = \begin{bmatrix} mI_3 & -mr_{bg}^{b\times} \\ mr_{bg}^{b\times} & I_b^b \end{bmatrix} \tag{2.18}$$

where r_{bg}^b represents the position vector from CO to CG expressed in {b}. The Added Mass matrix is calculated through Hydrodynamic Code using the in-house Hydrodynamic Response Analysis software, HydRA [14].

2.4.1.2 The Coriolis Matrix

Just like the Mass Matrix, the Coriolis-Centripetal Matrix also has an added mass component. The Coriolis Matrix arises due to the fact that our equation is written in a rotating non-inertial frame of reference, i.e BCS. The matrix is given as shown

$$C_{RB}(\nu) = \begin{bmatrix} m\nu_2^{\times} & -m\nu_2^{\times} r_{bg}^{b\times} \\ mr_{bg}^{b\times}\nu_2^{\times} & -(I_b^b\nu_2)^{\times} \end{bmatrix} \tag{2.19}$$

where, $\nu_2 = [p, q, r]^T$.

2.4.1.3 Hydrostatics

The combined generalized forces due to gravity and buoyancy are given as

$$g(\eta) = \begin{bmatrix} (W-B)\sin(\theta) \\ -(W-B)\cos(\theta)\sin(\phi) \\ -(W-B)\cos(\theta)\cos(\phi) \\ -(y_gW - y_bB)\cos(\theta)\cos(\phi) + (z_gW - z_bB)\cos(\theta)\sin(\phi) \\ (z_gW - z_bB)\sin(\theta) + (x_gW - x_bB)\cos(\theta)\cos(\phi) \\ -(x_gW - x_bB)\cos(\theta)\sin(\phi) - (y_gW - y_bB)\sin(\theta) \end{bmatrix} \tag{2.20}$$

where W is the Weight of the Body (mass times gravity), B is the Buoyancy Force which can be calculated by Archimedes Principle. In our case, the Weight of the AUV remains constant with the fore and aft nose cones flooded with water. However, some underwater vehicles may have a variable ballast to dynamically change Weight of the body. x_g, y_g, z_g denote the distance of the Center of Gravity from CO in respective directions. Similarly, x_b, y_b, z_b denote the distance of the Center of Buoyancy from CO.

2.4.1.4 Linear Damping

In Eq (2.21), $D(v)$ denotes the Linear Damping on the body. Damping could be Non-Linear and Coupled for an AUV moving at high speed, but we are going to assume a non coupled motion and Linear Damping. The D terms will be as given as

$$-diag\{X_u, Y_v, Z_w, K_p, M_q, N_r\} \tag{2.21}$$

These coefficients can be derived using experimental analysis a method known as *system identification*. For ships, these coefficients can also be found using Hydrodynamic Response Analysis software. However since we do not have any waves being generated,

we cannot use this approach. For our simulation, we can put an approximation or even ignore this damping.

2.4.1.5 Non Linear Damping

To account for the Non Linear Damping, we will be using the Cross Flow Drag principle as explained in Fossen [3]. This exploits strip theory, in which the hull is cut into circular 2D strips and the 2D Drag is calculated and accumulated to get the total drag acting on the hull. Hoerner [15] curve is used to calculate the 2D drag. The total moment in yaw and force in sway direction due to cross flow drag is given by

$$Y = -\frac{1}{2}\rho \int_{-L_{pp}/2}^{L_{pp}/2} T(x) C_d^{2D}(x) |v_r + xr|(v_r + xr) dx \tag{2.22}$$

$$N = -\frac{1}{2}\rho \int_{-L_{pp}/2}^{L_{pp}/2} T(x) C_d^{2D}(x) x |v_r + xr|(v_r + xr) dx \tag{2.23}$$

Here, C_d^{2D} is the constant 2-D drag coefficient, T is the draft, L_{pp} is the length between the perpendiculars i.e, the length of the AUV between the forward and aft nose cones. However it will become computationally expensive to calculate the integrals every time during simulation. Thus, a curve fitting excercise is performed offline which gives us the non-linear coefficients as below

$$Y = Y_{|v_r|v_r}|v_r|v_r + Y_{|v_r|r}|v_r|r + Y_{v_r|r|}v_r|r| + Y_{|r|r}|r|r \tag{2.24}$$

$$N = N_{|v_r|v_r}|v_r|v_r + N_{|v_r|r}|v_r|r + N_{v_r|r|}v_r|r| + N_{|r|r}|r|r \tag{2.25}$$

A similar exercise is performed for Heave and Pitch motion to get the coefficients in the Z (heave) and M (pitch) direction.

Another important thing to note is that here we are assuming Independence Principle between cross-flow drag and on-flow drag as explained in Zdrakovich [16]. Essentially this

means that for an oblique flow, we consider the components of the flow on the parallel and perpendicular direction to calculate the cross flow and on-flow drag respectively instead of considering any coupling effects.

2.4.1.6 Thrusters

For our marine vessel (Sookshma), we are going to consider the velocity to be constant surge. Generally, thrust provided is essentially a function of the thruster RPM, whose profile is defined by a K_T vs J_a where J_a stands for open-water advanced coefficient and is given by

$$J_a = \frac{u_a}{nD} \tag{2.26}$$

where u_a is the advance speed, n is the propeller revolution and D is the propeller diameter. K_T is also a function of J_a. The generalized force and $K_T(J_a)$ relation is given by

$$T = \rho D^4 K_T(J_a)|n|n \tag{2.27}$$

Due to lack of experimental results yet, we will assume the K_T vs J_a to be linear. We assume that at $J_a = 1$, K_T dies down to 0 and at $J_a = 0$, K_T corresponds to the thrust at zero velocity as mentioned in the manufacturers datasheet.

2.4.1.7 Control Surfaces

The Control Surface exerts forces via Lift and Drag on the aerofoil sections. In our simulator, there is a provision to include the hydrodynamic coefficients for the control surfaces obtained from system identification. Thus the generalized forces are derived by simply multiplying the actuator angle with the respective hydrodynamic coefficient. For example if the surge coefficient is provided i.e X_δ, the surge force is given simply by:

$$F_x = X_\delta * \delta \tag{2.28}$$

The algorithm can handle any orientation and position of the control surface and calculate the generalized force at the decided body center of the vessel. If the hydrodynamic coefficient is not provided, the force is calculated via the lift and drag coefficients in the provided NACA file

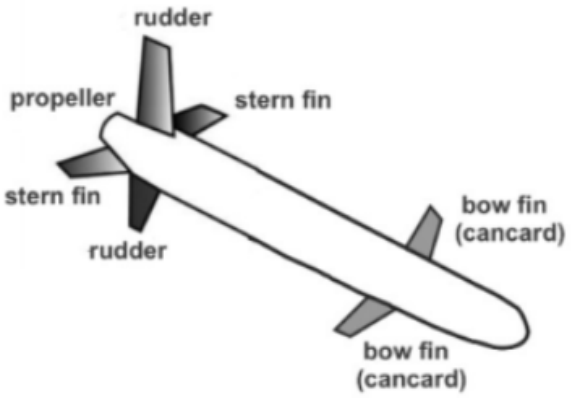


FIGURE 2.3: AUV with control surfaces
source: Fossen [3]

Chapter 3

Navigation

3.1 Introduction

Navigation tells a robot where it is with respect to its global position which is a crucial requirement and precursor for the robot to plan its next step. Navigation sensors such as GPS, UWB, and Sound multilateration (underwater GPS) provide the global position to the robot, whereas sensors like IMU and DVL provide raw data (acceleration and velocity) of the robot in its local frame and one needs to perform dead reckoning in order to find the global position. Dead Reckoning is composed of two words, *Dead* which means *stationary* and *Reckoning* which means *the action or process of calculating or estimating something*, so together Dead Reckoning means *Estimating the position of something with respect to a stationary point.* In our case the stationary point will be the starting point of the robot. All the sensors which we discussed are not accurate and have some inherent noise involved in their measurements. This noise introduces errors in dead reckoning and this error on accumulation becomes drift as shown in Fig 1.4. Thus estimation algorithms are employed to handle this noise. Traditionally the proven and tested state estimation algorithm is the *Extended Kalman Filter*.

The Kalman filter is an algorithm which uses the system dynamics and sensor measurements to predict and correct the state of a system especially in-case of noisy measurements. It can be thought of like a frequency band filter except that it has the ability to adjust its gains based on certain probability equations.

The Kalman filter has majorly two steps. The first one is the prediction step which uses the systems dynamic measurements such as velocity or acceleration to estimate the state of the system in the next instance. Then is the updation step which adjusts the co-variance matrices defined for the system and spits out the Kalman Gain matrix which is used to update the prediction based on sensor measurement.

For example consider we have system model for the position of the car undergoing acceleration from time t = t_{k-1} to t = t_k as

$$\dot{x} = u + at \tag{3.1}$$

where $\dot{x}$ is the velocity of the car, x being the position, u is the velocity of the car at the time t_{k-1} and a is the acceleration of the car between time t_{k-1} and time t_k. Suppose our car is equipped with an accelerometer and a GPS. The accelerometer give the acceleration of the car and the GPS gives us the position of the car. Our task is to estimate the position of the car. First, using the accelerometer measurements, we can find a prediction of the position of the car using 3.1. Let us say that the predicted position of the car is 10m from the staring point. However as per the GPS readings our car has moved only 9m. Now, we know that there is an uncertainity related to the GPS measurements and the accelerometer measurements. Let us say that based on our personal observations we find the GPS to be more accurate than the accelerometer. Thus during the updation step, we'll give more weightage to the GPS reading than the accelerometer prediction. Let us say we have 60% confidence on the GPS readings and 40% confidence on the accelerometer

readings, thus we have the estimated position of the car as

$$x = 0.4 \cdot 10 + 0.6 \cdot 9 = 9.4 \tag{3.2}$$

In this example we manually gave some constant confidences on the measurements. In case of a Kalman filter these are calculated dynamically using Process noise and sensor co-variance matrices.

Let us now establish the five equations used in the Kalman filer algorithm

$$x_k^- = A\hat{x_{k-1}} + Bu_k \tag{3.3a}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{3.3b}$$

$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R} \tag{3.4a}$$

$$\hat{x_k} = \hat{x_k^-} + K_k(y_k - Cx_k^-) \tag{3.4b}$$

$$P_k = (I - K_k C)P_k^- \tag{3.4c}$$

Equations 3.3 are called the prediction equations while equations 3.4 are called the update equations. Specifically, 3.3a is called the state extrapolation equation, as we'll see it describes how the state evolves with time. 3.3b is called the Covariance extrapolation equation which updates the covariance matrices which directly affects the Kalman Gain (or the confidence constants as discussed earlier). 3.4a is the Kalman gain equation, 3.4b is the state update equation and 3.4c is the Covariance update equation. The Kalman Filter becomes Extended Kalman Filter when we deal with non-linear equations of motions. In this case the non-linear equations are linearized using Taylor expansion.

3.1.1 EKF - Experimental Results

We will now see some of the Experimental Results of the performance of the Extended Kalman Filter done on the *Sookshma* vessel in the Department of Ocean Engineering Wave Basin at IIT Madras. The algorithm leverages the Panisim Simulator Architecture with hardware in the loop. The goal of the experiment is to do waypoint tracking across four corner points in the wavebasin.

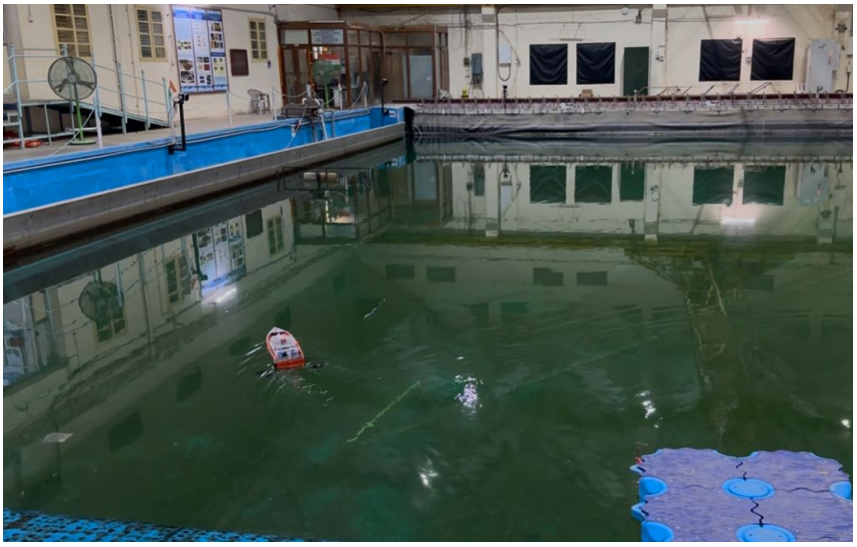


FIGURE 3.1: Way-point Tracking Experiment in Wavebasin at IIT Madras

All the run data has been collected using *ROS2* bags. The navigation sensors used in this experiment are, four UWB sensors kept at each of the corner of the wavebasin, and an onboard IMU (by SBG Systems). The EKF is used to fuse the sensors and get the state estimates. In total, 15 states are estimated, three position (x,y,z), three orientation (Euler Angles), three linear velocity, three angular velocity and three linear acceleration. Some of the EKF experiments are shown in Fig 3.4.

As we can see in the Figures, the EKF estimation is very jerky and not smooth. In this scenario, the sensor noise (IMU) was considered to be higher and thus the UWB position estimates had more weightage over the IMU. There are also some sudden jumps seen in between in the position estimate plots. These jumps are due to incorrect UWB

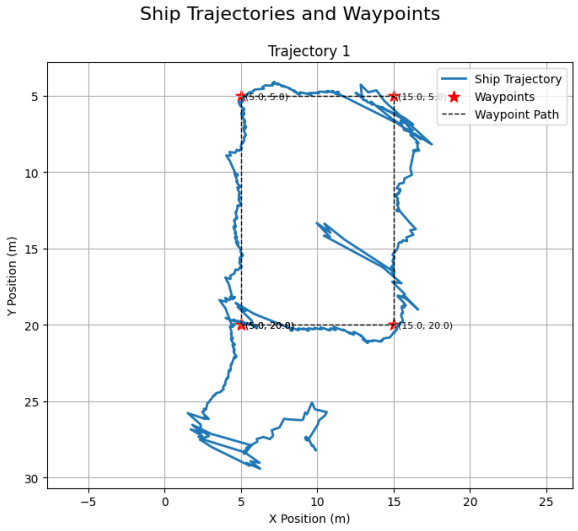


FIGURE 3.2: Waypoint Tracking example 1

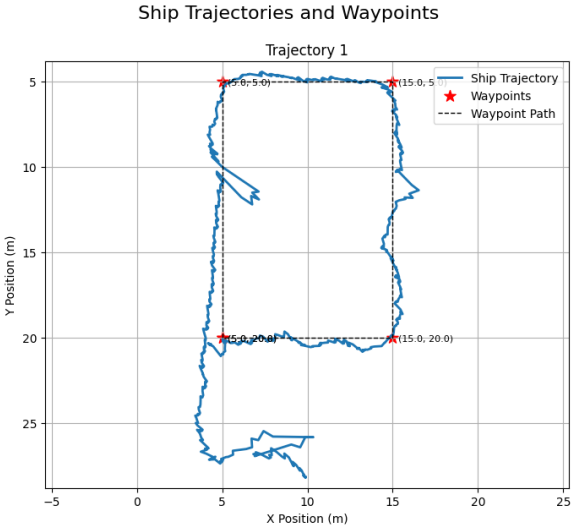


FIGURE 3.3: Waypoint Tracking example 2

FIGURE 3.4: Waypoint Tracking examples

measurements near that area. These incorrect measurements are caused due to a physical obstruction in the wavebasin which causes the UWB signals to reflect off.

This is one of the disadvantage of using navigation sensors which depend on external signals. They often get obstructed (like GPS on a cloudy day) or even jammed or spoofed (during warfare). This calls for more accurate navigation sensors which do not rely on any external signals. Now we will explore one such proposed method.

3.1.2 DR via Machine Learning

With advancements in machine learning algorithms, data driven approaches for dead reckoning have started to emerge. We will now explore our attempt at the data-driven approach.

To start with, we attempted at two dof position estimate (x and y) for the USV *Sookshma*. The goal was to develop a model capable of predicting the vessel's next 2D position (x, y) based on a set of inputs: current acceleration (from IMU: a_x, a_y), angular velocity

(w_z), the current estimated position (x,y), and the rudder angle. We have only chosen the measurements which have relevance in the two degree planar motion. The core idea was for the model to implicitly learn and compensate for IMU sensor noise and biases, learn the USV dynamics with the control input (rudder) and effectively performing dead reckoning over time. The data collected from the experiments was too noisy and jerky and thus the data for training the models was created via simulation in Panisim. Two sets of data were collected, one with the IMU sensor simulated without noise and the other with IMU sensor simulated with covariance with added noise. The IMU data was simulated at 50Hz whereas the Position and rudder data was being published at 10Hz. A data extraction was carried out from the ROS2 bags and the data was synchronized at 10Hz for training the model.

Since Dead Reckoning is a Markov process, i.e the next states only depend on the value of the current states and not previous history, it was decided to first test with a simple Feedforward Neural Network.

It is to be noted the prediction happens in an auto-regressive manner, i.e the position predicted by the model in the current time step is used as the input to the model to find the position in the next time step.

3.1.2.1 Feedforward Neural Network (FNN) Development

Several attempts were made with Feed Forward Neural Network. The below summarizes of the attempts which did not work and some which performed a bit decent. The below graphs are for the data with no IMU noise.

- **Absolute Position Prediction:** Initially the FNN model was designed to directly predict the next absolute (x,y) position. A simple Feed forward architecture was used. The inputs were current planar acceleration, body velocity, angular velocity in the normal direction, current position and rudder angle. The output was the next

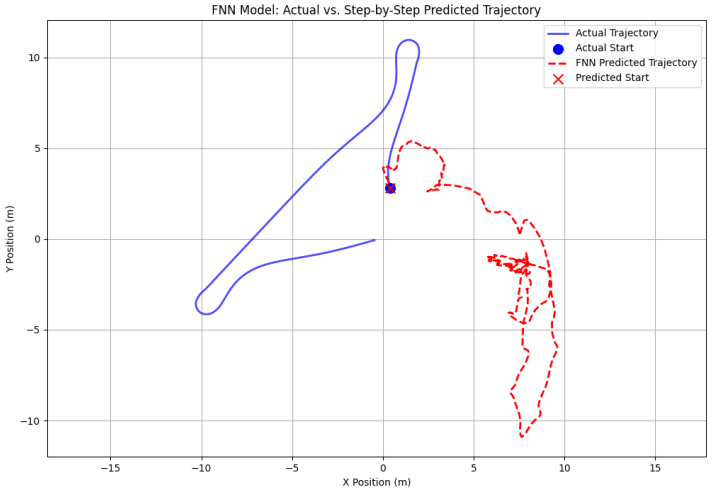


FIGURE 3.5: Simple FNN performance

time step's position and velocity. However, this approach led to poor performance, with the predicted trajectory quickly accumulating errors and drifting significantly from the actual path. The predictions also started diminishing in change. This was attributed to the auto-regressive nature of the prediction, where small errors at each step compound over time.

- **Delta Position Prediction:** To mitigate error accumulation and the diminishing prediction, the FNN was modified to predict the *change* in position $(\Delta x, \Delta y)$ from the current step to the next. The absolute position was then calculated by adding this predicted delta to the current position. This showed some improvement, but drift remained a considerable issue.

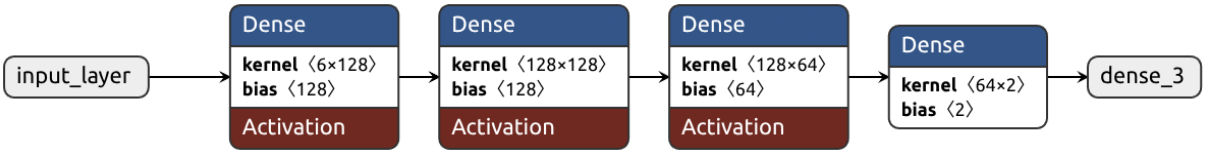


FIGURE 3.6: Architecture of the *Difference in Position Prediction* model

Further attempts to improve the delta-predicting FNN involved adding 'Dropout' layers for regularization and an additional 'Dense' layer to increase model capacity.

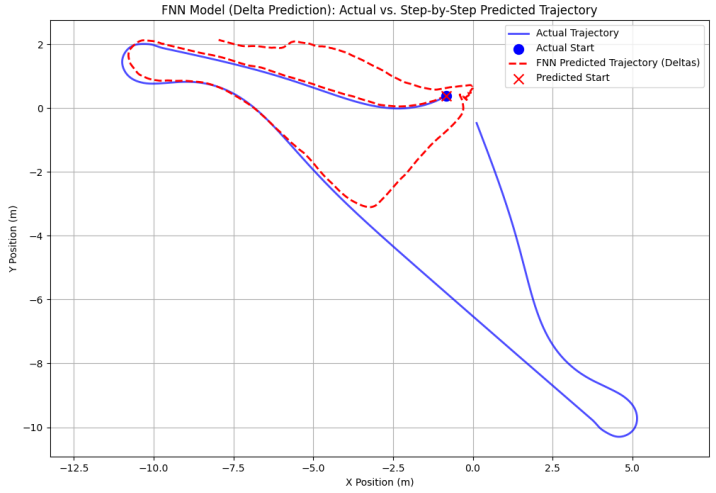


FIGURE 3.7: Prediction Result from delta predictions

These changes offered only marginal benefits, the FNN's performance was still not satisfactory for reliable dead reckoning.

- **Decoupled FNN:** Inspired by successes with the LSTM model in [8], a decoupled approach was adopted for the FNN. Two separate FNN models were trained: one exclusively for predicting Δx and the other for Δy. Each model had its own output scaler. Also velocity was not used this time. This model performed the best for some paths. But still struggled with others.

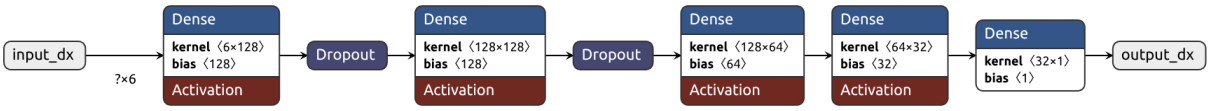


FIGURE 3.8: Decoupled Model Architecture

- **Velocity Prediction and Integration**: It was noted the the acceleration and angular velocity measurements, are in the body frame of the vessel whereas the position is in the global frame. In the planar 2D motion, the yaw angle is what relates both of them together. Thus it was decided to attempt at a simple model which just takes in the planar accelerations, normal angular velocity, current linear velocity, rudder angle and this time the current yaw angle as input.The model then predicts

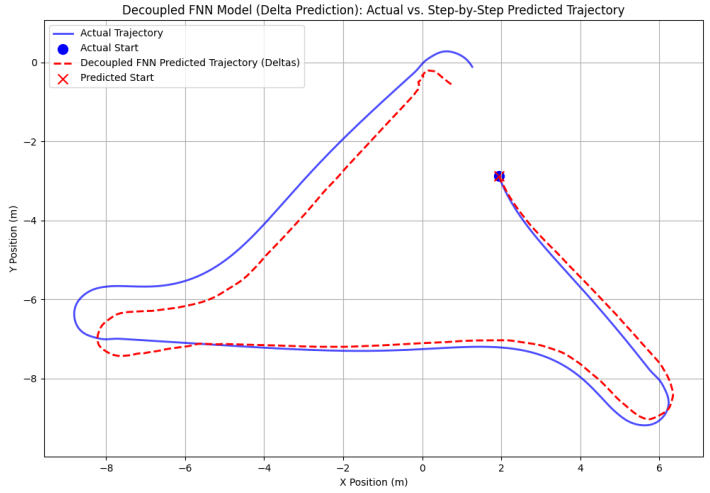


FIGURE 3.9: Best Prediction performed with Decoupled FNN Architecture

the next velocity. The predicted velocities are auto-regressed for next predictions and integrated with timestep 0.1s to obtain the global position. Surprisingly this approach is very robust and gave the best results.

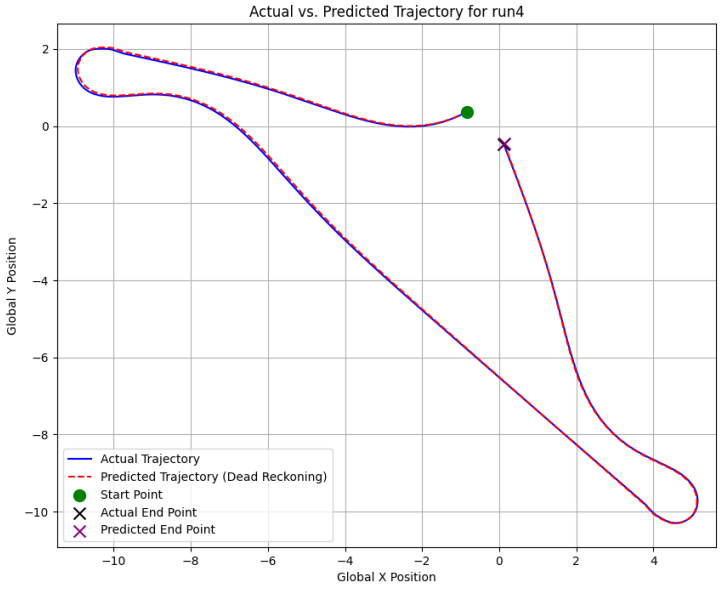


FIGURE 3.10: Ground truth vs Trajectory with velocity based dead reckoning

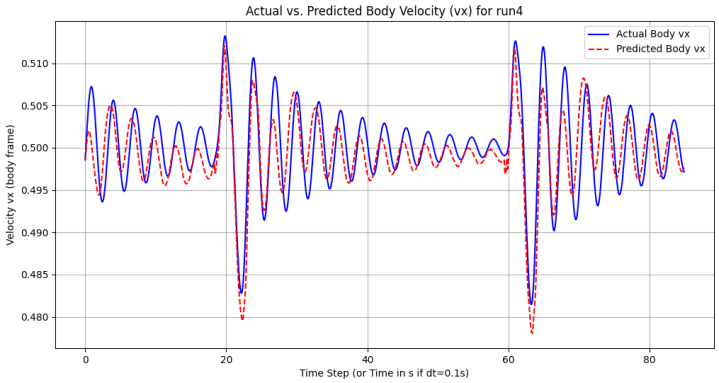


FIGURE 3.11: Velocity Predictions

This model uses the predicted velocities to calculate the position, however it relies on accurate yaw information from the data. It was suspected that the model performs really good since the speed of the body in the simulation is more or less constant and thus it becomes a fairly simple problem, given accurate yaw information. So next, we try by modifying our model to also predict the yaw angle and auto-regress the predicted yaw angles during testing.

- **Velocity and Yaw Prediction**: This time, the model also predicted the yaw output. So it was ensured that we don't have to rely on yaw data (although a lot of IMU sensors provide accurate yaw information). It was expected that the model would learn to calculate the yaw angle using the angular velocity input. This provided some realistic results and drift was now seen in the predictions as a result of error accumulation.

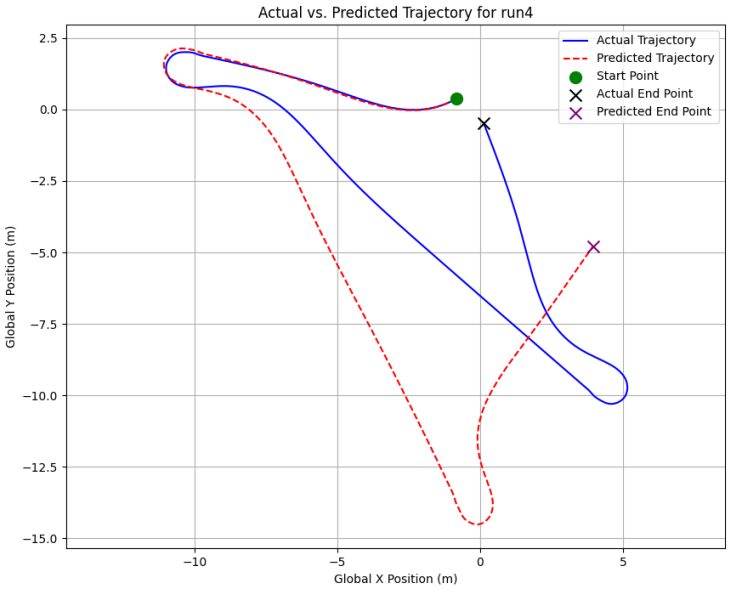


FIGURE 3.12: Trajectory based on Velocity and Yaw Predictions

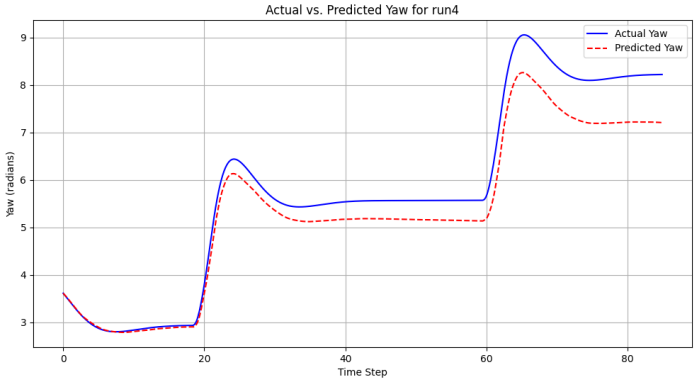


FIGURE 3.13: Yaw Predictions

- **Velocity and Yaw Delta Prediction**: Now taking inspiration from previous delta predictions, it was decided to try a model which predicts the change in velocity and yaw. The inputs to the model were a_x, a_y, w_z, v_x, v_y, $sin(yaw)$, $cos(yaw)$, and rudder angle. During prediction, the yaw and velocity values were auto regressed. Instead of raw yaw angle, the sin and cos of the yaw angle were used for the model to learn the cyclic nature of angles (0 to 2π) since previously in some cases the yaw angle would explode. This model perform even better.

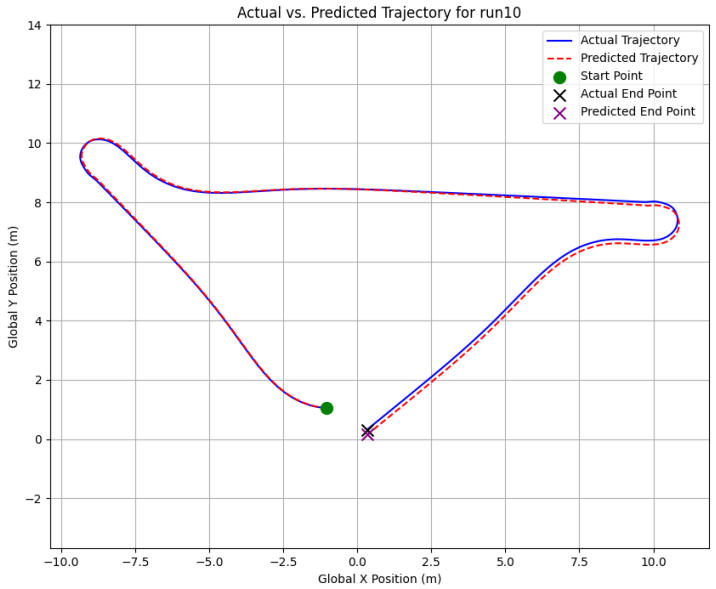


FIGURE 3.14: Trajectory via Yaw and Velocity delta predictions

Now that we have our model working well on no noise IMU dataset, it is time to test our model with IMU sensor noise involved. Here the model did had some error in prediction with almost a meter of error. The model seems to have generalized good as we can see a decent performance in the unseen data as shown in Fig 3.16. Also the training values were scaled before training using standardization.

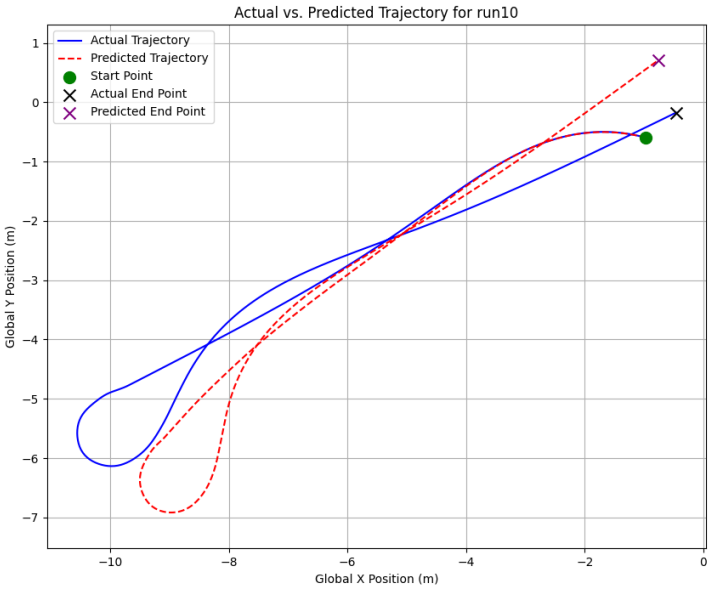


FIGURE 3.15: Trajectory Prediction with sensor Noise

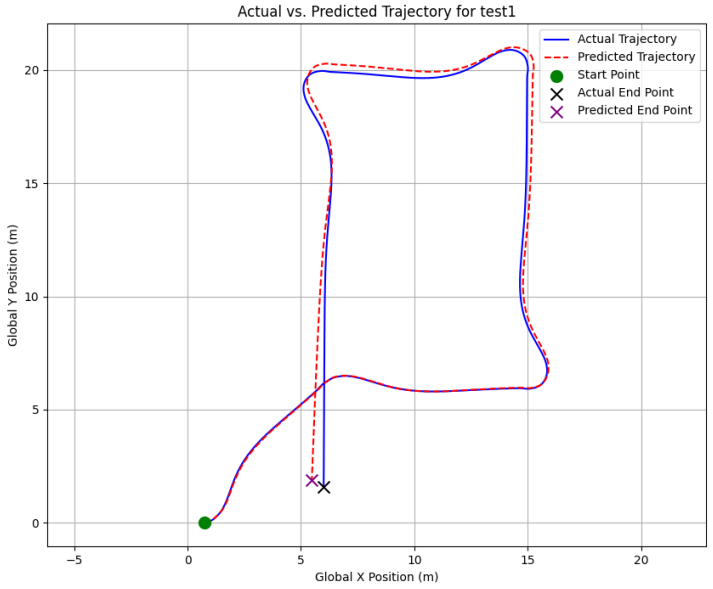


FIGURE 3.16: Trajectory Prediction with sensor Noise on **unseen data**

3.1.2.2 Long Short-Term Memory (LSTM) Network Development

Although Dead Reckoning is a Markov Process, it was proposed that the model may perform better by learning the trajectory and constraints between consecutive time steps, allowing it to capture temporal dependencies in the vessel's motion patterns. Given the sequential nature of navigation data, Long Short-Term Memory (LSTM) networks were explored as an alternative to the FNN approach. Also the IMU sensor noise and bias may have temporal dependency which LSTM may be able to capture well.

- **Absolute Position Prediction:** Initially, similar to the FNN approach, an LSTM model was designed to predict absolute positions directly. The model took sequences of IMU data, current position, and rudder angle as inputs. However, this approach suffered from the same limitations as the FNN, with significant drift and diminishing changes in predictions over time due to the auto-regressive prediction method.

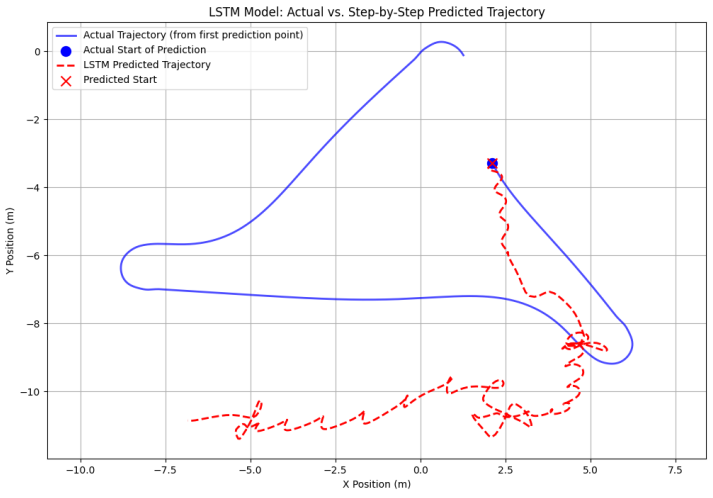


FIGURE 3.17: Initial LSTM with absolute position prediction

- **Delta Position Prediction:** Learning from the FNN experience, the LSTM models were modified to predict position deltas $(\Delta x, \Delta y)$ instead of absolute positions. Sequence-to-vector prediction was used, where a sequence of past inputs

predicts the delta at the end of the sequence. However this modification had no improvement.

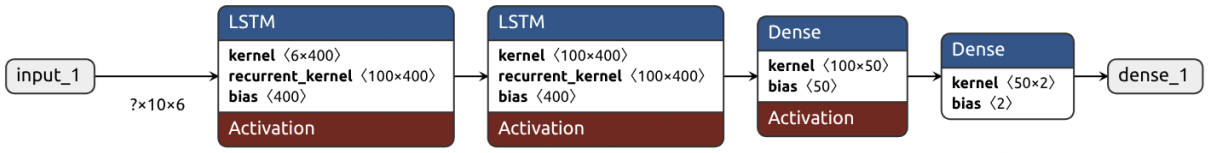


FIGURE 3.18: Architecture of the LSTM Delta Prediction Model

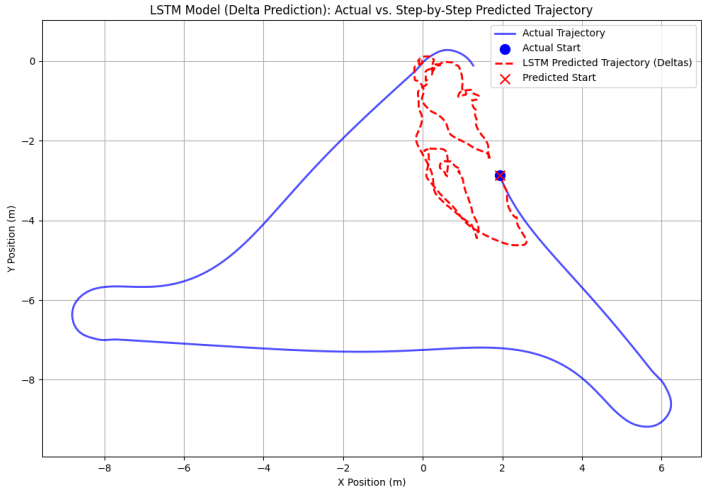


FIGURE 3.19: LSTM Delta Prediction Performance

- **Decoupled LSTM (Based on Research):** Inspired by research in [8] we tried decoupling the LSTM layers for x and y direction. This approach involved:

  1. Creating separate LSTM models for predicting Δx and Δy, allowing each model to specialize in a single component of motion.
  2. Using 128 LSTM units with recurrent dropout to prevent overfitting within the recurrent connections.
  3. Applying a higher dropout rate (50%) after dense layers to further enhance regularization.
  4. Creating dedicated scalers for each model to normalize inputs and outputs independently.

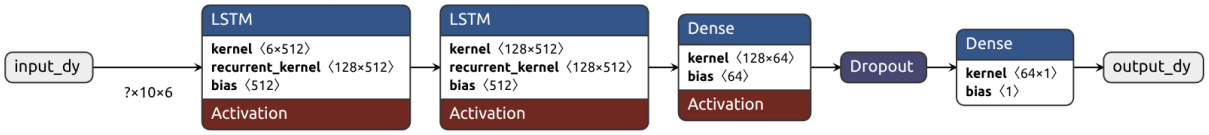


FIGURE 3.20: Decoupled LSTM Model Architecture

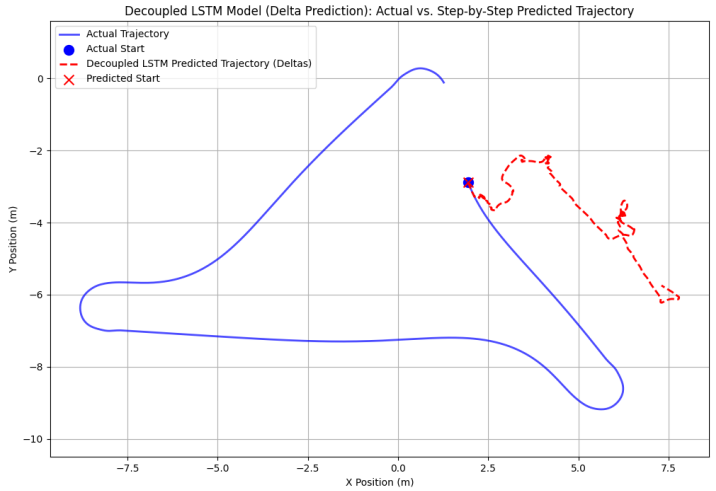


FIGURE 3.21: Performance of the Decoupled LSTM Approach

Although the model converged well in training, the auto-regressive in prediction still caused a large drift. Unlike the reference paper [8], we are passing the acceleration and position in the input whereas the reference used a highly accurate DVL to predict velocities.

Chapter 4

Conclusion

In conclusion, a high-fidelity, light-weight and modular Marine Vessel Simulator (Panisim) was developed with proper dynamics and kinematics. Extensive documentation for the simulator was made and is available at [6]. The Simulator documentation is also attached at the end of this report. Waypoint tracking experiment in wavebasin was conducted with an Extended Kalman Filter and data for sixty runs were collected. This data can be used to further process and train machine learning models for system identification or navigation models. Machine Learning Based approach for marine vessel navigation was explored. Best result was found with a simple Feed Forward Neural Network which predicts the change in body frame velocity and yaw angle. More data and physics based loss approach (dynamics in the loss function) may improve the model prediction capability.

References

[1] All About Circuits, "Intro to allan variance: Analysis, non-overlapping, and overlapping allan variance," 2023. Accessed: 2024-09-20.

[2] V. Suvorkin, M. Garcia-Fernandez, G. González-Casado, M. Li, and A. Rovira-Garcia, "Assessment of noise of mems imu sensors of different grades for gnss/imu navigation," *Sensors*, vol. 24, no. 6, 2024.

[3] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. Chichester, UK: John Wiley & Sons, 2 ed., 2021. 736 pages.

[4] J. Zhang, V. Ila, and L. Kneip, "Robust visual odometry in underwater environment," in *2018 OCEANS - MTS/IEEE Kobe Techno-Oceans (OTO)*, (Kobe, Japan), pp. 1–9, 2018.

[5] E. Potokar, S. Ashford, M. Kaess, and J. G. Mangelson, "Holoocean: An underwater robotics simulator," in *2022 International Conference on Robotics and Automation (ICRA)*, (Philadelphia, PA, USA), pp. 3040–3046, 2022.

[6] Rishabh Sharma, "Panisim simulator documentation." https://goldenrishabh.github.io/PanisimDocs/, 2025. Accessed: 2025-05-08.

[7] M. T. Sabet, H. M. Daniali, A. Fathi, and E. Alizadeh, "A low-cost dead reckoning navigation system for an auv using a robust ahrs: Design and experimental analysis," *IEEE Journal of Oceanic Engineering*, vol. 43, no. 4, pp. 927–939, 2018.

[8] E. T. et al., "Lstm-based dead reckoning navigation for autonomous underwater vehicles," in *Global Oceans 2020: Singapore – U.S. Gulf Coast*, (Biloxi, MS, USA), pp. 1–7, 2020.

[9] X. Mu, B. He, X. Zhang, Y. Song, Y. Shen, and C. Feng, "End-to-end navigation for autonomous underwater vehicle with hybrid recurrent neural networks," *Ocean Engineering*, vol. 194, p. 106602, 2019.

[10] S. Song, J. Liu, J. Guo, J. Wang, Y. Xie, and J. H. Cui, "Neural-network-based auv navigation for fast-changing environments," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9773–9783, 2020.

[11] I. Saksvik, A. Alcocer, and V. Hassani, "A deep learning approach to dead-reckoning navigation for autonomous underwater vehicles with limited sensor payloads," *Preprint*, 2021.

[12] GuerrillaCG, "Euler (gimbal lock) explained." YouTube, 2009.

[13] M. D. Shuster, "A survey of attitude representations," *Journal of Guidance, Control, and Dynamics*, vol. 16, no. 5, pp. 727–734, 1993.

[14] A. Somayajula, "Hydra." https://hydra.hydratech.in/, 2021.

[15] S. F. Hoerner, *Fluid-Dynamic Drag: Practical Information on Aerodynamic Drag and Hydrodynamic Resistance*. Midland Park, NJ: Hoerner Fluid Dynamics, 2 ed., 1965. Self-published.

[16] M. M. Zdravkovich, *Flow Around Circular Cylinders: Volume 2: Applications*, vol. 2 of *Oxford Science Publications*. Oxford: Oxford University Press, 1997.

Pani Simulator

Rishabh Sharma

2025-05-07

Table of contents

Pani Simulator

Pani (which means *water* in Hindi) is a high fidelity marine vessel simulator. It is built in a manner such that minimum backend edits are required by the user and the simulator can be both configured and run in an intuitive lightweight GUI application which runs on the web. An attempt to make it as "Microsoft's Flight Simualtor" but for marine vessels has been made.

The simulator can be used to configure a new marine vessel, simulate sensors such as DVL, IMU, LiDAR and Camera, and control the vessel either via teleop input or custom files for autonomous tasks. The simulator supports multiple agents which is useful for swarm autonomy tasks or to develop better collision avoidance algorithms.

The project has been made open source to allow for further development of this simulator and add additional features to make it more versatile.

Overview

The Simulator is majorly divided into two parts:

- Vessel Configurator
- Vessel Simulation

The Vessel Configurator is what allows you to configure a new marine vessel. You can load a FBX Geometry file, select individual components such as control surfaces, thrusters, sensors and configure their parameters via the interactive GUI. If you like a more geeky experience, you can edit the input files manually as well. There are also options to enter the non-linear hydrodynamics coefficients for the vessel (which we plan to integrate with our in-house system identification algorithm, in future), set vessel geometry parameters and specify initial conditions. This is also where you will setup the simulation parameters such as time, time-step, geofence, GPS datum and physical constants (gravity, density etc). It also supports the output from our in-house hydrodynamics program *HyDRA*, which gives the added mass coefficients and wave hydrodynamics of the vessel. Once you are done with the vessel configuration, the program auto generates the input files required for the simulation to run.

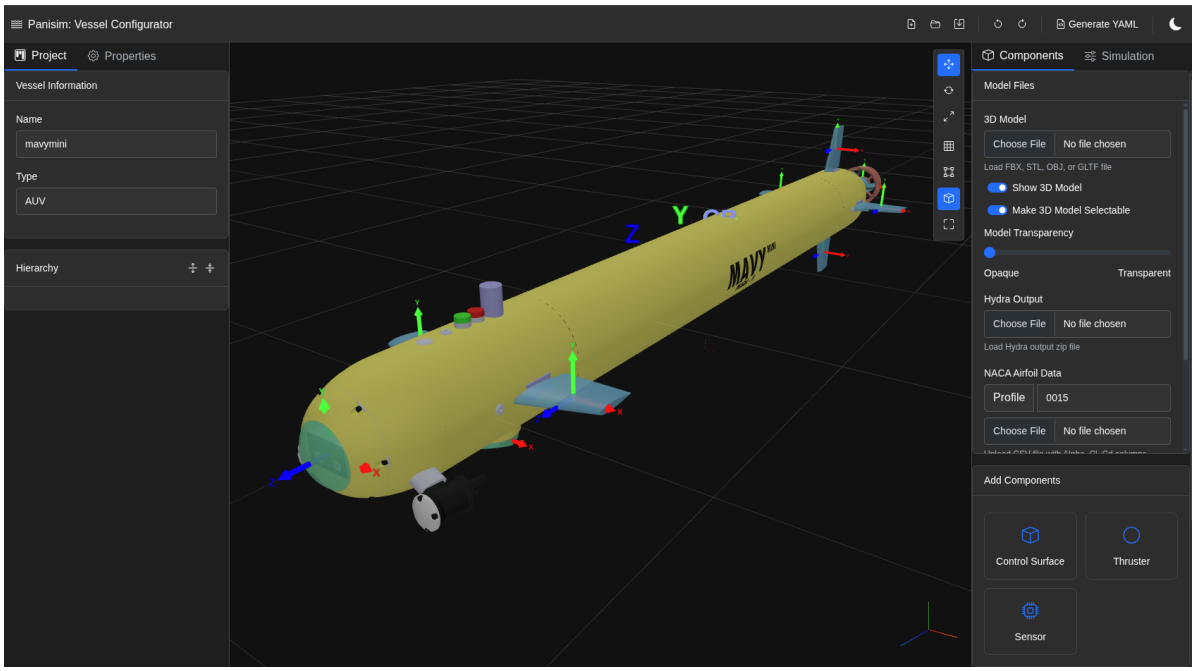


Figure 1: Vessel Configurator

The Vessel Simulation is where the actual simulation takes place. It follows the dynamics as per the parameters you set in the Vessel Configurator. This is where you can also custom code your control and guidance algorithms which will publish the required data in ROS2 to make the vessel run. Currently the sensors are simulated with added noise whose outputs you can see in the ROS2 terminal, or from the GUI. We soon plan to add modelling for the simulator with real world physics.

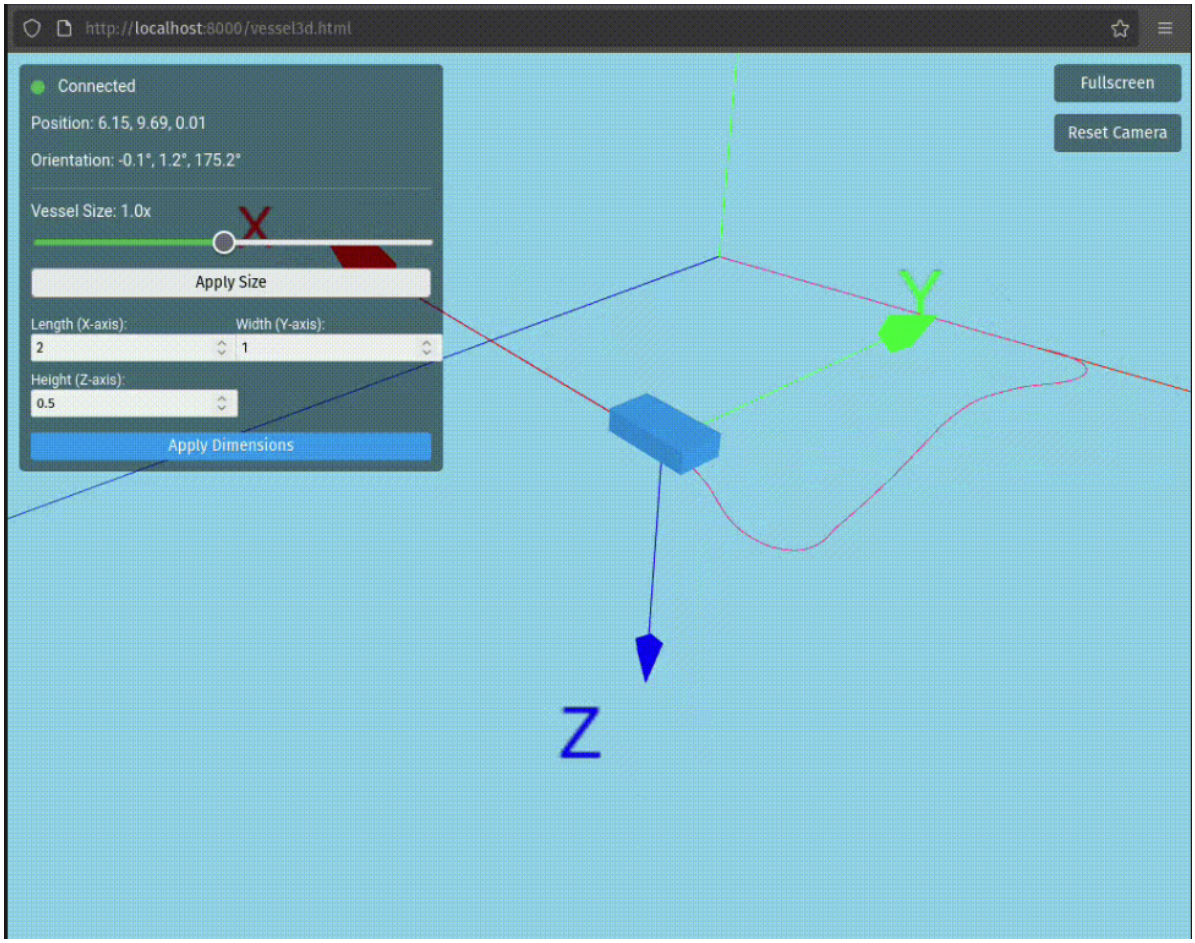


Figure 2: Way-point trackingSimulation Visualization

Key Features

- **Realistic Physics**: Follows dyanmics modelling as defined by Fossen's *Handbook of Marine Craft Hydrodynamics and Motion Control*
- **Modular Architecture**: Easy integration of custom vessel models, sensors, and controllers
- **ROS2 Integration**: Native support for Robot Operating System 2 (ROS2) for distributed robotics applications
- **Web Application**: Lightweight web application for configuring and running the simulation
- **Multiple Agents**: Simulate multiple vessels in the same environment

How to Use This Documentation

Navigate through the sections using the sidebar. The documentation discusses the Setup, theory on the Kinematics and Dynamics used in the simulator, the coordinate systems used for various components (control surfaces, thrusters, sensors and Body centre), the python simulator backend and how to use the Panisim vessel configurator and the Panisim vessel simulator.

Setup

Prerequisites

- Git installed on your system
- Docker installed on your system
- Basic understanding of terminal commands
- Python version 3.10 or higher

Cloning the Repository

1. Open a terminal window
2. Clone the repository using the following command:

```
git clone https://github.com/MarineAutonomy/makara.git
```

3. Navigate to the cloned repository:

```
cd makara
```

4. Switch to the mavymini branch:

```
git checkout mavymini
```

Building the Docker Image

Before running the simulator, you need to build the Docker image:

1. Ensure you're in the root directory of the cloned repository
2. Build the Docker image by executing:

```
./ros2_devdocker.sh
```

3. Wait for the build process to complete (this may take several minutes depending on your internet connection and system performance)

Troubleshooting

- **Docker issues**: Ensure Docker is installed and running on your system
- **Permission issues**: If you encounter permission errors when running scripts, try prefixing the commands with `sudo`

Quickstart

After building the Docker image, you can start the simulator:

1. From the root directory of the repository, run the following script in your terminal:

```
./ros2_simulator.sh
```

2. Once the simulator starts, it will display the ROS2 topics available in the terminal. You should see an output similar to this:

```
   ==================================================
   ROS2 Simulation Started Successfully!
   ==================================================

Available ROS2 Topics:
------------------------------
/parameter_events
/rosout
/sookshma_00/Rudder_1/encoder
/sookshma_00/actuator_cmd
/sookshma_00/imu/data
/sookshma_00/odometry
/sookshma_00/odometry_sim
/sookshma_00/uwb
/sookshma_00/vessel_state
/sookshma_00/vessel_states_ekf
/sookshma_00/waypoints

 View Topic Data:
------------------------------
1. Open a new terminal:
    docker exec -it panisim bash

2. Subscribe to a topic:
    ros2 topic echo <topic_name>

 Example:
```

```
    ros2 topic echo /vessel_0/odometry_sim

==================================================
```

To inspect the data published on these topics, follow these steps:

a. Open a new terminal window.

b. Enter the Docker container by executing the following command in a new terminal:

```
docker exec -it panisim bash
```

c. Subscribe to a specific topic using the `ros2 topic echo` command. Replace `<topic_name>` with the actual topic you want to inspect.

```
ros2 topic echo <topic_name>
```

For example, to view the odometry data for `mavymini_00`, you would use:

```
ros2 topic echo /sookshma_00/odometry_sim
```

> **Note**
>
> The specific topic names may vary depending on the agents activated in your `/inputs/simulation_input.yml` configuration file.

Currently, the example vessel is configure with an initial velocity of 0.5m/s. (You can change this by editing the `/inputs/simulation_input.yml` file.)

A Web-based visualization GUI would have started automatically. You can access it by opening a new browser tab and navigating to `http://localhost:8000`.

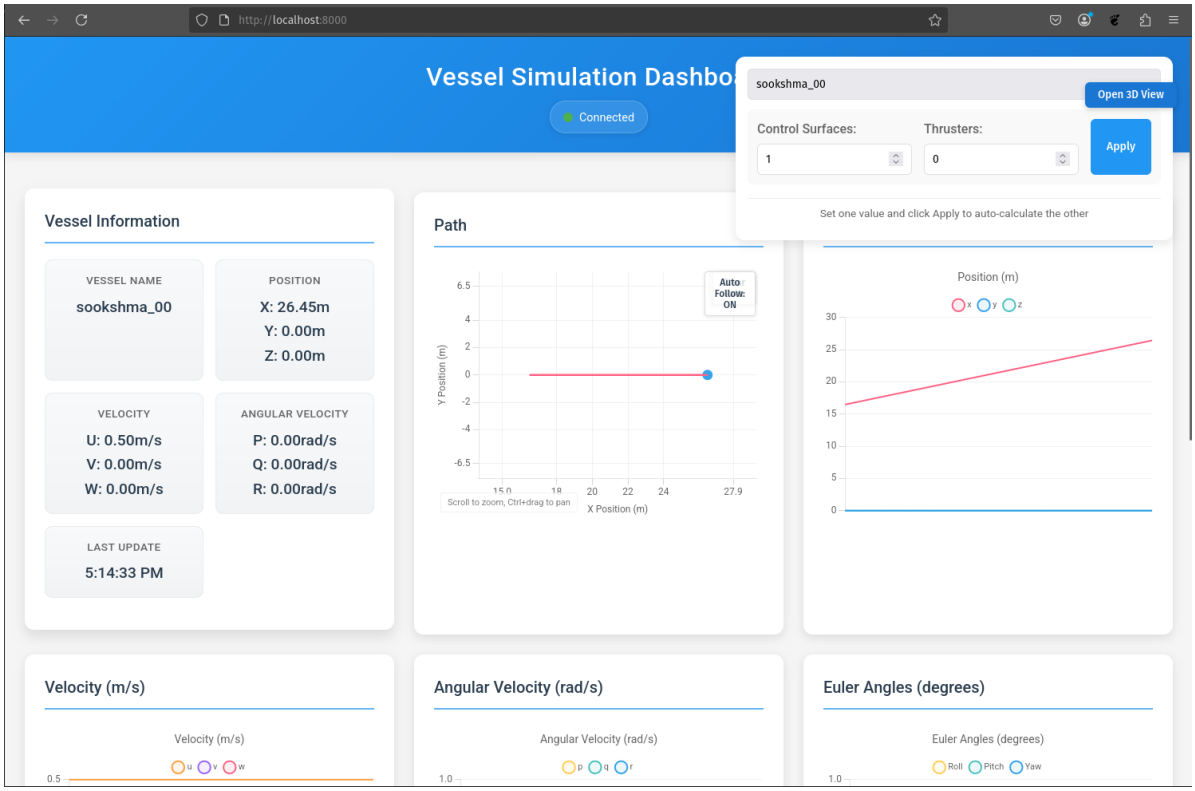


Figure 3: Web-based visualization GUI

This visualization GUI shows you the current state of the vessel, including its position, velocity, orientation and actuator data.

You can also see the 6 DOF of the vessel via the “Open 3D View” button. A new window will appear showing you a box shaped vessel with its position and orientation in the simulation.

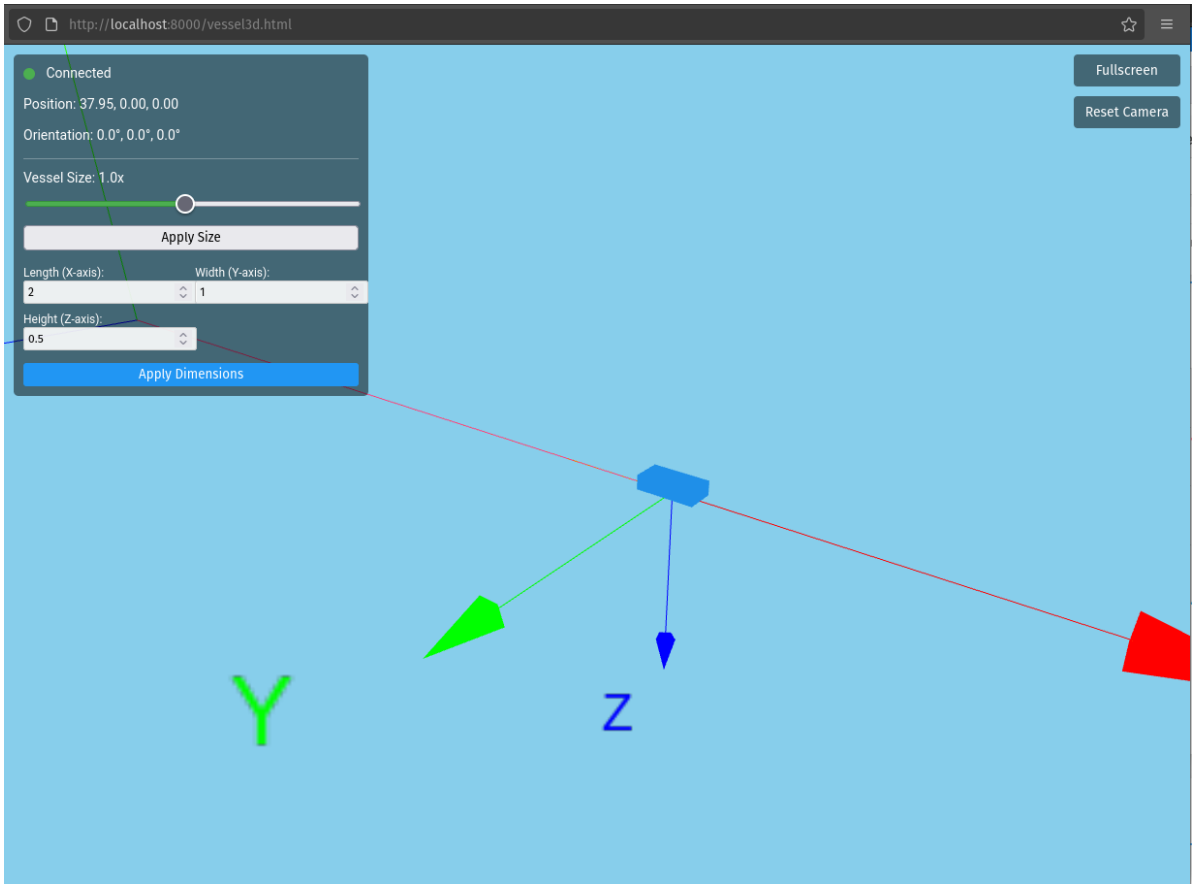


Figure 4: 3D View

And that's it! The simulation is up and running and now you can start any other ROS nodes to command the vessel actuators, perform Extended Kalman Filtering, or anything else you want. We will be covering a waypoint tracking example to show you how you can write your own ROS2 nodes to control your vessel.

Kinematics

Overview

The Kinematics module (`module_kinematics.py`) provides a comprehensive set of mathematical functions for handling coordinate transformations, rotation representations, and kinematic calculations essential for marine vehicle simulation.

> Tip
>
> This module serves as the mathematical foundation for all spatial transformations in the simulator.

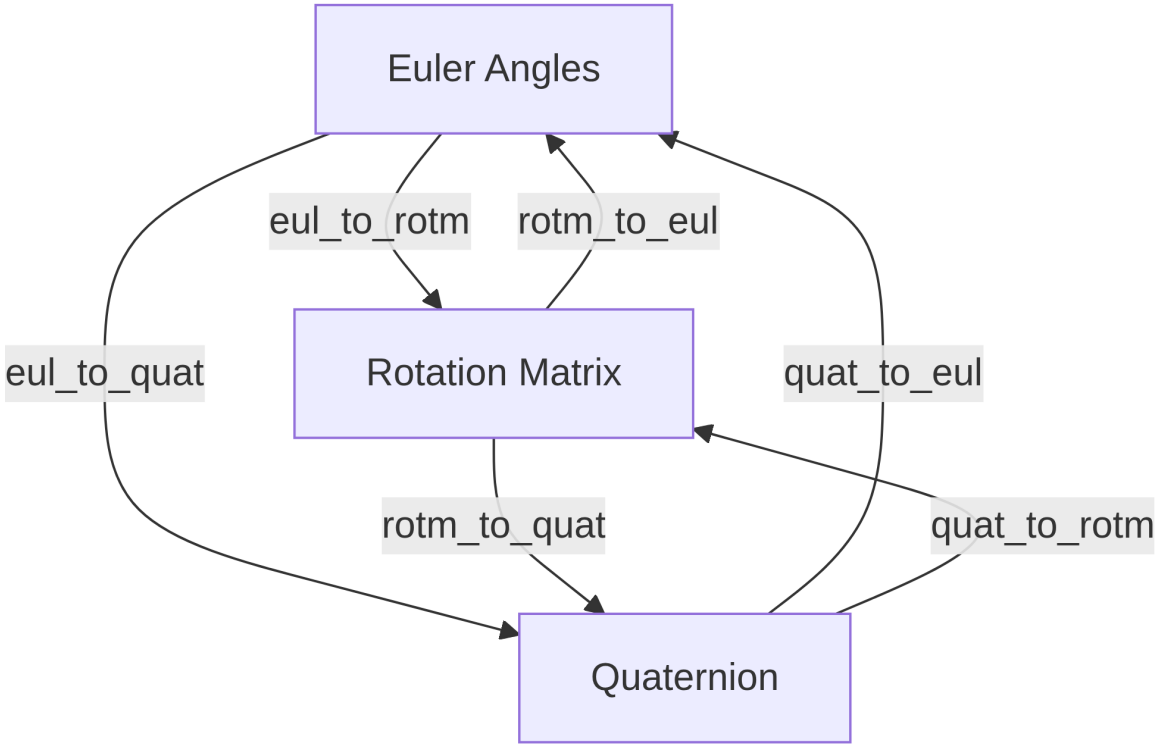
Coordinate Frames

The simulator utilizes several key coordinate frames:

1. **Earth Centered Inertial (ECI) frame {i}**: An inertial frame with its origin at the Earth's center. It does not rotate with the Earth's rotation.
2. **Earth Centered Earth Fixed (ECEF) frame {e}**: Rotates with the Earth. Its origin is at the Earth's center. The x-axis passes through the intersection of the prime meridian and the equator, the z-axis aligns with the Earth's rotation axis, and the y-axis completes the right-handed system.
3. **North-East-Down (NED) frame {n}**: A local tangent frame fixed to a point on the Earth's surface (or a reference point). The x-axis points North, the y-axis points East, and the z-axis points Down, perpendicular to the Earth's ellipsoid surface. This is the primary navigation frame.
4. **Body frame {b}**: An orthogonal frame fixed to the vehicle. The x-axis points forward, the y-axis points right (starboard), and the z-axis points down.

Function Categories

Rotation Representation Conversions



Function	Description
`eul_to_rotm(eul, order='ZYX', deg=False)`	Converts Euler angles (roll ϕ, pitch θ, yaw ψ) to a 3x3 rotation matrix that transforms vectors from the Body frame {b} to the NED frame {n}. Assumes ZYX rotation order. Angles can be in radians or degrees.
`rotm_to_eul(rotm, order='ZYX', prev_eul=None, deg=False, silent=True)`	Converts a 3x3 rotation matrix back to Euler angles (roll ϕ, pitch θ, yaw ψ). Handles potential ambiguities and gimbal lock using previous Euler angles (`prev_eul`) if provided. Can output in radians or degrees.
`eul_to_quat(eul, order='ZYX', deg=False)`	Converts Euler angles (roll ϕ, pitch θ, yaw ψ) to a unit quaternion $[q_w, q_x, q_y, q_z]$. Assumes ZYX rotation order. Angles can be in radians or degrees.

Function	Description
`quat_to_eul(quat, order='ZYX', deg=False, prev_quat=None, silent=True)`	Converts a unit quaternion $[q_w, q_x, q_y, q_z]$ to Euler angles (roll ϕ, pitch θ, yaw ψ). Handles potential ambiguities using `prev_quat` if provided. Can output in radians or degrees.
`quat_to_rotm(quat)`	Converts a unit quaternion $[q_w, q_x, q_y, q_z]$ to a 3x3 rotation matrix that transforms vectors from the Body frame {b} to the NED frame {n}.
`rotm_to_quat(rotm)`	Converts a 3x3 rotation matrix (Body to NED) to a unit quaternion $[q_w, q_x, q_y, q_z]$.

Quaternion Operations

Function	Description
`quat_multiply(q1, q2)`	Multiplies two quaternions $\mathbf{q}_1 \otimes \mathbf{q}_2$. Order matters: represents rotation $\mathbf{q}_2$ followed by rotation $\mathbf{q}_1$.
`quat_conjugate(quat)`	Computes the conjugate of a quaternion $\mathbf{q}^* = [q_w, -q_x, -q_y, -q_z]^T$.
`rotate_vec_by_quat(vec_a, q_a_b)`	Rotates a 3D vector `vec_a` using the quaternion rotation `q_a_b` (representing rotation from frame A to frame B) to get the vector in frame B. Computes $\mathbf{v}'_b = \mathbf{q}_{a\to b} \otimes \mathbf{v}'_a \otimes \mathbf{q}^*_{a\to b}$.

Rate Calculations

Function	Description
`eul_rate_matrix(eul, order='ZYX', deg=False)`	Computes the 3x3 transformation matrix $\mathbf{T}(\boldsymbol{\eta})$ that relates body-frame angular velocity $\boldsymbol{\omega}^b$ to Euler angle rates $\dot{\boldsymbol{\eta}}$ via $\dot{\boldsymbol{\eta}} = \mathbf{T}(\boldsymbol{\eta})\boldsymbol{\omega}^b$. Assumes ZYX order.
`quat_rate_matrix(quat)`	Computes the 4x3 transformation matrix $\mathbf{E}(\mathbf{q})$ that relates body-frame angular velocity $\boldsymbol{\omega}^b$ to quaternion rates $\dot{\mathbf{q}}$ via $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{E}(\mathbf{q})\boldsymbol{\omega}^b$.
`eul_rate(eul, w, order='ZYX')`	Calculates Euler angle rates $[\dot{\phi}, \dot{\theta}, \dot{\psi}]$ given current Euler angles `eul` and body-frame angular velocity `w` $= [p, q, r]^T$. Uses `eul_rate_matrix`.
`quat_rate(quat, w)`	Calculates quaternion rates $[\dot{q}_w, \dot{q}_x, \dot{q}_y, \dot{q}_z]$ given the current quaternion `quat` and body-frame angular velocity `w` $= [p, q, r]^T$. Uses `quat_rate_matrix`.

Function	Description
`deul_dquat(quat)`	Computes the 3x4 Jacobian matrix $\frac{\partial \boldsymbol{\eta}}{\partial \mathbf{q}}$, representing the partial derivatives of Euler angles with respect to quaternion components.
`dquat_deul(quat)`	Computes the 4x3 Jacobian matrix $\frac{\partial \mathbf{q}}{\partial \boldsymbol{\eta}}$, representing the partial derivatives of quaternion components with respect to Euler angles. Requires converting `quat` to `eul` internally first.

Navigation Functions

Function	Description
`ssa(ang, deg=False)`	Converts an angle (in radians or degrees) to its smallest signed angle representation within the range $(-\pi, \pi]$ radians or $(-180, 180]$ degrees.
`clip(value, threshold)`	Limits the input `value` to the range `[-threshold, threshold]`.
`ned_to_llh(ned, llh0)`	Converts a position vector `ned` = [North, East, Down] relative to a reference point `llh0` = [lat0, lon0, h0] into absolute geodetic coordinates `llh` = [latitude, longitude, height]. Uses WGS84 ellipsoid model.
`llh_to_ned(llh, llh0)`	Converts an absolute geodetic position `llh` = [lat, lon, h] into a local NED position vector `ned` = [North, East, Down] relative to a reference point `llh0` = [lat0, lon0, h0]. Uses WGS84 ellipsoid model.
`generate_waypoints()`	Generates a predefined sequence of waypoints as LLH coordinates for a rectangular survey pattern relative to a specific datum location (IITM lake). Returns a numpy array of [lat, lon, height] waypoints.
`rotm_ned_to_ecef(llh)`	Computes the 3x3 rotation matrix $\mathbf{R}_n^e$ that transforms vectors from the local NED frame (defined at `llh`) to the ECEF frame.

Utility Functions

Function	Description
`Smat(vec)`	Creates the 3x3 skew-symmetric matrix $\mathbf{S}(\mathbf{v})$ corresponding to the input 3D vector `vec`. Used for representing cross products as matrix multiplications ($\mathbf{a} \times \mathbf{b} = \mathbf{S}(\mathbf{a})\mathbf{b}$).

Usage Examples

Coordinate Transformations

```
import numpy as np
from mav_simulator.module_kinematics import eul_to_rotm, llh_to_ned

# Convert Euler angles to rotation matrix
euler_angles = np.array([0.1, 0.2, 0.3])  # [roll, pitch, yaw] in radians
R = eul_to_rotm(euler_angles)  # Rotation matrix

# Convert latitude, longitude, height to NED coordinates
reference_point = np.array([60.0, 10.0, 0.0])  # [lat, lon, height]
target_point = np.array([60.001, 10.001, 10.0])  # [lat, lon, height]
ned_coords = llh_to_ned(target_point, reference_point)
```

Rotation Representations

```
from mav_simulator.module_kinematics import eul_to_quat, quat_to_rotm

# Convert from Euler angles to quaternion
euler_angles = np.array([0.1, 0.2, 0.3])  # [roll, pitch, yaw] in radians
quaternion = eul_to_quat(euler_angles)

# Convert from quaternion to rotation matrix
R = quat_to_rotm(quaternion)
```

Best Practices

- Use the `ssa()` function to normalize angles when working with Euler angles

- Be consistent with the rotation order convention throughout your code (the default is 'ZYX')
- When transforming between frames, always keep track of the reference frames involved

Dynamics

Overview

The Dynamics modules (`class_vessel.py` and `calculate_hydrodynamics.py`) implement the mathematical foundation for simulating the motion of marine vehicles through water. The simulator uses Fossen's nonlinear 6-DOF equation of motion:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) = \tau$$

where,

- $M = M_{RB} + M_A$ is the mass matrix, combining rigid body mass M_{RB} and added mass M_A
- $C(\nu) = C_{RB}(\nu) + C_A(\nu)$ is the Coriolis and centripetal matrix
- $D(\nu)$ is the hydrodynamic damping matrix
- $g(\eta)$ is the gravitational and buoyancy force vector
- τ is the control force vector from thrusters and control surfaces
- $\nu = [u, v, w, p, q, r]^T$ is the velocity vector in the body frame
- $\eta = [x, y, z, \phi, \theta, \psi]^T$ is the position and orientation vector

> **Tip**
>
> The dynamics modules serve as the core of the simulation, determining how forces and moments translate into vessel motion through water.

State Representation

The vessel state is represented as a vector combining velocities, position, orientation, control surface angles, and thruster states:

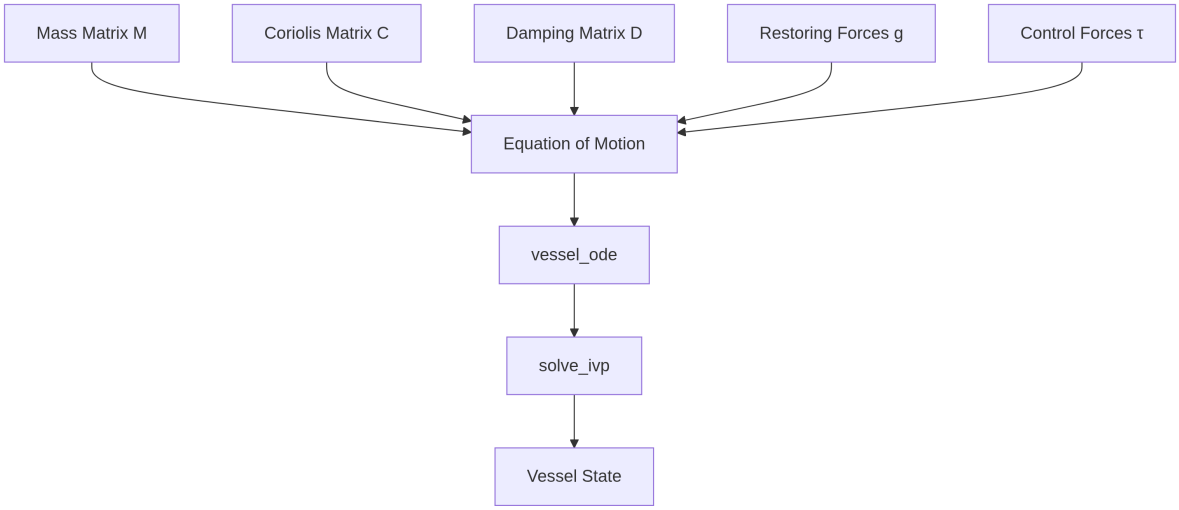
state $= [\nu^T, \eta_{pos}^T, \eta_{rot}^T, \delta^T, n^T]^T$

Where:

- $\nu = [u, v, w, p, q, r]^T$ are the linear and angular velocities in the body frame

- $\eta_{pos} = [x, y, z]^T$ is the position in the NED frame
- η_{rot} is the orientation, either as Euler angles $[\phi, \theta, \psi]^T$ or quaternion $[q_0, q_1, q_2, q_3]^T$
- δ is the vector of control surface angles
- n is the vector of thruster rotational speeds (RPM)

Components of the Dynamic Equation



Mass Matrix (M)

The mass matrix M combines the rigid-body inertia matrix M_{RB} and the added mass matrix M_A:

$$M = M_{RB} + M_A$$

Rigid-body Mass Matrix (M_{RB})

The rigid-body mass matrix is generated in the `_generate_mass_matrix` method in `calculate_hydrodynamics.py`:

$$M_{RB} = \begin{bmatrix} mI_{3\times3} & -mS(r_G) \\ mS(r_G) & I_G \end{bmatrix}$$

Where:

- m is the vessel mass
- $I_{3\times3}$ is the 3×3 identity matrix
- r_G is the center of gravity vector relative to the origin of the body frame
- $S(r_G)$ is the skew-symmetric matrix of r_G (implemented by `Smat()` in `module_kinematics.py`)
- I_G is the inertia tensor calculated from the radii of gyration

```
def _generate_mass_matrix(self, CG, mass, gyration):
    # Calculate gyration tensor about vessel frame
    xprdct2 = np.diag(gyration)**2 - Smat(CG)@Smat(CG)

    # Generate the inertia matrix using radii of gyration
    inertia_matrix = xprdct2 * mass

    # Generate the mass matrix
    mass_matrix = np.zeros((6,6))
    mass_matrix[0:3][:, 0:3] = mass * np.eye(3)
    mass_matrix[3:6][:, 3:6] = inertia_matrix
    mass_matrix[0:3][:, 3:6] = -Smat(CG) * mass
    mass_matrix[3:6][:, 0:3] = Smat(CG) * mass

    return mass_matrix
```

Added Mass Matrix (M_A)

The added mass matrix represents the added inertia due to accelerating the surrounding fluid. It's calculated from hydrodynamic data obtained from HydRA (You can also enter custom hydrodynamics added mass coefficients (example Y_vd) in the `hydrodynamics.yml` input file instead of using HydRA, as discussed in the Inputs section).

$$M_A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Function	Description
`calculate_added_mass_from_hydra(hydra_file)`	Computes the added mass matrix from HydRA data file containing frequency-dependent hydrodynamic coefficients. Extracts zero-frequency added mass for most terms, but computes heave, roll, and pitch terms at their natural frequencies for better accuracy.

```python
def calculate_added_mass_from_hydra(self, hydra_file):
    # Load hydrodynamic data
    with open(hydra_file, 'r') as file:
        mdict = json.load(file)

    # Extract arrays from data
    omg = np.array(mdict['w'])      # Frequency array
    AM = np.array(mdict['AM'])      # Added mass array

    # Extract zero-frequency added mass
    A_zero = AM[1, :, :, 0, 0]

    # Calculate natural frequencies and
    # interpolate added mass at those frequencies
    # ...

    # Return properly transformed added mass matrix
    return A
```

> **Note**
>
> The added mass calculated from HydRA is in ENU frame and so it is converted to NED frame in the `calculate_added_mass_from_hydra` function.

Coriolis and Centripetal Matrix

The Coriolis matrix ($C(\nu)$) accounts for forces arising from motion in a rotating reference frame (Body frame is rotating with respect to NED frame):

$$C(\nu) = C_{RB}(\nu) + C_A(\nu)$$

These matrices are calculated using:

$$C_{RB}(\nu) = \begin{bmatrix} 0_{3\times3} & -S(M_{11}v_1 + M_{12}v_2) \\ -S(M_{11}v_1 + M_{12}v_2) & -S(M_{21}v_1 + M_{22}v_2) \end{bmatrix}$$

$$C_A(\nu) = \begin{bmatrix} 0_{3\times3} & -S(A_{11}v_1 + A_{12}v_2) \\ -S(A_{11}v_1 + A_{12}v_2) & -S(A_{21}v_1 + A_{22}v_2) \end{bmatrix}$$

Where $v_1 = [u, v, w]^T$ and $v_2 = [p, q, r]^T$ are the linear and angular velocity components of ν.

Function	Description
`calculate_coriolis_matrices(vel)`	Computes the rigid body and added mass Coriolis-centripetal matrices based on the current velocity state. Returns a tuple of (C_{RB}, C_A) matrices.

Damping Matrix

The damping matrix ($D(\nu)$) represents hydrodynamic resistance forces and is typically modeled as:

$$D(\nu) = D_{linear} + D_{nonlinear}(\nu)$$

In the implementation, damping is handled through hydrodynamic coefficients (entered in the `hydrodynamics.yml` input file) rather than an explicit matrix:

Function	Description
`hydrodynamic_forces(vel)`	Computes hydrodynamic damping forces by applying coefficients to velocity components. Supports linear, quadratic, and cross-terms.
`cross_flow_drag()`	Estimates sway-yaw damping coefficients for surface vessels using strip theory and Hoerner's cross-flow drag.
`cross_flow_drag_AUV()`	Similar to `cross_flow_drag()` but includes heave-pitch calculations for AUVs.
`hoerner()`	Implements Hoerner's method for computing 2D drag coefficients based on section beam-to-draft ratio.

The coefficients follow a naming convention that indicates which force/moment they affect and which velocity components are involved (you can enter any custom hydrodynamic coefficients in the `hydrodynamics.yml` input file and it will be handled by the function for force calculations):

- Linear damping: `X_u`, `Y_v`, `Z_w`, `K_p`, `M_q`, `N_r`
- Quadratic damping: `X_u_au`, `Y_v_av`, `Z_w_aw`, etc. (the `a` indicates absolute value)
- Cross-coupling: `X_vr`, `Y_ur`, `N_uv`, etc.

```python
def hydrodynamic_forces(self, vel):
    # Extract velocities
    u, v, w, p, q, r = vel

    # Initialize forces vector
    F = np.zeros(6)

    # Map velocity components to values
    vel_map = {'u': u, 'v': v, 'w': w, 'p': p, 'q': q, 'r': r}

    # Process each hydrodynamic coefficient
    for coeff_name, coeff_value in self.hydrodynamics.items():
        if coeff_value == 0: continue

        parts = coeff_name.split('_')
        force_dir = parts[0]
        if force_dir not in self.force_indices: continue

        # Calculate force contribution
        force = coeff_value
        for component in parts[1:]:
            if component.startswith('a') and len(component) > 1:
                # Absolute value term (e.g., u|u|)
                v_char = component[1:]
                force *= abs(vel_map[v_char])
            elif component in vel_map:
                # Regular term
                force *= vel_map[component]

        # Add to force vector
        F[self.force_indices[force_dir]] += force

    return F
```

Restoring Forces

The restoring forces ($g(\eta)$) include gravitational and buoyancy effects:

$$g(\eta) = \begin{bmatrix} (W-B)\sin\theta \\ -(W-B)\cos\theta\sin\phi \\ -(W-B)\cos\theta\cos\phi \\ -(y_G W - y_B B)\cos\theta\cos\phi + (z_G W - z_B B)\cos\theta\sin\phi \\ (z_G W - z_B B)\sin\theta + (x_G W - x_B B)\cos\theta\cos\phi \\ -(x_G W - x_B B)\cos\theta\sin\phi - (y_G W - y_B B)\sin\theta \end{bmatrix}$$

Where:

- $W = mg$ is the weight
- $B = \rho g \nabla$ is the buoyancy force (with ∇ being the displaced volume)
- $r_G = [x_G, y_G, z_G]^T$ is the center of gravity mentioned in the `geometry.yml` input file
- $r_B = [x_B, y_B, z_B]^T$ is the center of buoyancy mentioned in the `geometry.yml` input file

Function	Description
`gravitational_forces(phi, theta)`	Computes the gravitational and buoyancy forces and moments as a function of roll and pitch angles.

Control Forces

The control forces vector τ represents the combined effect of control surfaces and thrusters.

Function	Description
`control_forces(delta)`	Calculates forces and moments generated by control surfaces (rudders, fins, etc.) based on their deflection angles δ. Supports both hydrodynamic coefficient-based and aerofoil-based calculation methods.

For hydrodynamic coefficient-based calculation: - Forces are calculated as $\tau_i = C_{i\delta}\delta$, where $C_{i\delta}$ is the control coefficient mentioned in the `control_surface_hydrodynamics` in the `control_surfaces.yml` input file

For aerofoil-based calculation:

- Local velocity at the control surface is calculated considering vessel motion (automatically done)

- Angle of attack is determined from flow direction and surface deflection
- Lift and drag forces are calculated using NACA airfoil data (path to the NACA file is mentioned in the `control_surfaces.yml` input file)
- Forces are transformed to the body frame and a generalized force vector is returned

Thruster Forces

Function	Description
`thruster_forces(n_prop)`	Calculates forces and moments from propellers/thrusters based on their rotational speeds n in RPM. Models thrust using propeller theory with advance ratio and thrust coefficients.

The thrust calculation follows the form:

$$X_{prop} = K_T \rho D^4 |n| n$$

Where:

- K_T is the thrust coefficient (approximated as $K_{T,J=0}(1 - J)$ with J being the advance ratio)
- ρ is the water density
- D is the propeller diameter
- n is the propeller speed in revolutions per second

Vessel ODE and Simulation

The vessel dynamics are implemented as an ordinary differential equation (ODE) in the `vessel_ode(t, state)` method, which computes the state derivatives:

```
def vessel_ode(self, t, state):
    # Extract state components
    vel = state[0:6]  # [u, v, w, p, q, r]
    pos = state[6:9]  # [x, y, z]
    angles = state[9:attitude_end]  # [phi, theta, psi] or quaternion

    # Calculate forces and moments
    F_hyd = self.hydrodynamic_forces(vel)
    F_control = self.control_forces(state[control_start:thruster_start])
```

```
    F_thrust = self.thruster_forces(state[thruster_start:])
    F_g = self.gravitational_forces(angles[0], angles[1])

    if self.coriolis_flag:
        C_RB, C_A = self.calculate_coriolis_matrices(vel)
        F_C = (C_RB + C_A) @ vel
    else:
        F_C = np.zeros(6)

    # Total force
    F = F_hyd + F_control + F_thrust - F_g - F_C

    # Calculate velocity derivatives
    M = self.mass_matrix + self.added_mass_matrix  # Total mass matrix
    state_dot[0:6] = np.linalg.inv(M) @ F

    # Calculate position and attitude derivatives
    # ...

    return state_dot
```

Function	Description
`step()`	Advances the simulation by one time step by solving the ODE using SciPy's `solve_ivp`.
`simulate()`	Runs the full simulation by repeatedly calling `step()` until the final time.
`reset()`	Resets the vessel to its initial state.

Helper Methods and Utilities

Dimensionalization

Hydrodynamic coefficients can be provided in non-dimensional form and converted to dimensional form:

Function	Description
`_dimensionalize_coefficients(rho, L, U)`	Converts non-dimensional hydrodynamic coefficients to dimensional form based on density ρ, length L, and characteristic velocity U (mentioned in the `simulation_input.yml` file)

Hydrodynamic Coefficient Calculation

Function	Description
`calculate_added_mass_from_hydra(hydra_file)`	Calculates the added mass matrix from HydRA data files.
`cross_flow_drag()`	Estimates sway-yaw hydrodynamic coefficients using strip theory.
`cross_flow_drag_AUV()`	Estimates both sway-yaw and heave-pitch coefficients for underwater vehicles.
`hoerner()`	Implements Hoerner's method for 2D section drag coefficient calculation.

> **Note**
>
> Understanding these dynamics components is essential for correctly configuring vessel parameters and interpreting simulation results. The implementation allows for flexible modeling of different vessel types by adjusting the parameters and coefficients.

Sensors Simulation

Overview

The Panisim simulator provides realistic sensor simulations to enable the development and testing of perception, estimation, and control algorithms. The sensor system is designed to be:

- **Modular**: Each sensor is implemented as a separate class
- **Configurable**: Sensors can be customized through YAML configuration files
- **Realistic**: Includes appropriate noise models and physical placement effects
- **Extensible**: New sensor types can be easily added

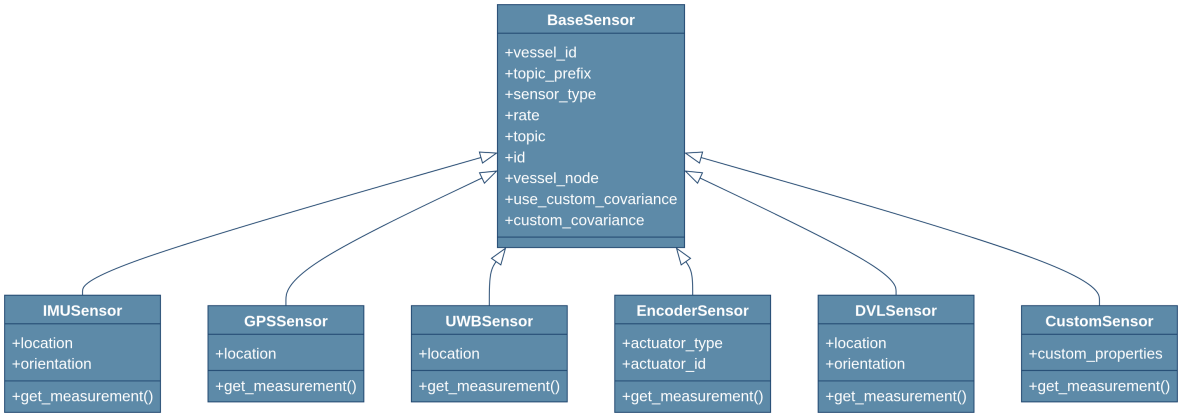
This document explains how sensors are simulated, configured, and how you can create your own custom sensors.

Sensor Architecture

All sensors in the simulator inherit from a common `BaseSensor` class that handles basic functionality such as:

- Sensor identification and topic naming
- Update rate management
- Access to vessel state
- Noise and covariance management

The sensor architecture follows this class hierarchy:



Sensor Types and Simulation

The simulator currently supports the following sensor types:

IMU (Inertial Measurement Unit)

IMU sensors measure:

- Orientation (quaternion)
- Angular velocity (rad/s)
- Linear acceleration (m/s^2)

Simulation details:

The IMU simulation accounts for:

1. Sensor placement relative to the vessel's center of mass
2. Sensor orientation relative to the vessel's body frame
3. Gravitational acceleration effects
4. Centripetal and tangential accelerations due to vessel rotation
5. Realistic noise models for all measurements

Noise model:

- Orientation: Multivariate normal distribution applied to Euler angles
- Angular velocity: Multivariate normal distribution
- Linear acceleration: Multivariate normal distribution

The covariances are defined in the `sensors.yml` file.

```
# Example of how IMU measurements are generated
def get_measurement(self, quat=False):
    state = self.vessel_node.vessel.current_state
    state_der = self.vessel_node.vessel.current_state_der

    # Convert orientation and add noise
    q_sensor = kin.rotm_to_quat(kin.quat_to_rotm(quat)
@ kin.quat_to_rotm(orientation_quat))
    q_sensor = kin.eul_to_quat(kin.quat_to_eul(q_sensor)
+ np.random.multivariate_normal(np.zeros(3), self.eul_cov))

    # Calculate acceleration including all effects
    acc_bcs = state_der[0:3] + np.cross(omg_bcs, v_bcs)
    acc_s_bcs = acc_bcs + np.cross(alpha, self.location)
+ np.cross(omg_bcs, np.cross(omg_bcs, self.location))
    acc_sensor = kin.quat_to_rotm(orientation_quat).T @ acc_s_bcs

    # Add gravity and noise
    acc_sensor = acc_sensor + kin.quat_to_rotm(q_sensor).T
@ np.array([0, 0, -self.vessel_node.vessel.g])
    acc_sensor = acc_sensor + np.random.multivariate_normal(np.zeros(3), self.lin_acc_cov)

    # Calculate angular velocity with noise
    omg_sensor = kin.quat_to_rotm(orientation_quat).T @ omg_bcs
    omg_sensor = omg_sensor + np.random.multivariate_normal(np.zeros(3), self.ang_vel_cov)

    return {...}  # Return measurements and covariances
```

GPS (Global Positioning System)

GPS sensors measure:

- Latitude (degrees)
- Longitude (degrees)
- Altitude (meters)

Simulation details:

The GPS simulation accounts for:

1. Sensor placement on the vessel

2. Conversion between NED (North-East-Down) and LLH (Latitude-Longitude-Height) coordinates
3. Realistic position noise

Noise model:

- Position: Multivariate normal distribution applied to NED coordinates before conversion to LLH

```
# Example of how GPS measurements are generated
def get_measurement(self, quat=False):
    state = self.vessel_node.vessel.current_state
    llh0 = self.vessel_node.vessel.gps_datum

    # Get position in NED, add sensor offset and noise
    ned = state[6:9] + kin.quat_to_rotm(orientation) @ self.location
    ned = ned + np.random.multivariate_normal(np.zeros(3), self.gps_cov)

    # Convert to latitude-longitude-height
    llh = kin.ned_to_llh(ned, llh0)

    return {...}  # Return latitude, longitude, altitude and covariance
```

UWB (Ultra-Wideband Positioning)

UWB sensors measure:

- Position in the NED frame (meters)

Simulation details:

The UWB simulation accounts for:

1. Sensor placement on the vessel
2. Position in the NED coordinate frame
3. Realistic position noise (typically higher precision than GPS)

Noise model:

- Position: Multivariate normal distribution applied to NED coordinates

```
# Example of how UWB measurements are generated
def get_measurement(self, quat=False):
    state = self.vessel_node.vessel.current_state
    ned = state[6:9]

    # Get position and add sensor offset and noise
    r_sen = ned + kin.quat_to_rotm(orientation) @ self.location
    r_sen = r_sen + np.random.multivariate_normal(np.zeros(3), self.uwb_cov)

    return {...}  # Return position and covariance
```

Encoder Sensors

Encoder sensors measure:

- Actuator position (degrees for control surfaces, RPM for thrusters)

Simulation details:

The encoder simulation accounts for:

1. Specific actuator type (rudder, thruster, etc.)
2. Unit conversion from state vector to appropriate units
3. Actuator-specific noise characteristics

Noise model:

- Actuator value: Normal distribution with specified RMS noise

```
# Example of how Encoder measurements are generated
def get_measurement(self):
    state = self.vessel_node.vessel.current_state

    # Get the specific actuator value from the state vector
    actuator_value_raw = state[self.state_index]

    # Apply unit conversion (e.g., rad to degrees, rad/s to RPM)
    actuator_value_converted = actuator_value_raw * self.unit_conversion

    # Add noise to the converted measurement
    actuator_value_noisy = actuator_value_converted + np.random.normal(0, self.noise_rms)

    return {...}  # Return actuator value, name, and covariance
```

DVL (Doppler Velocity Log)

DVL sensors measure:

- Linear velocity in the body frame (m/s)

Simulation details:

The DVL simulation accounts for:

1. Sensor placement and orientation
2. Body-frame velocity measurements
3. Realistic velocity noise

Noise model:

- Velocity: Multivariate normal distribution applied to body-frame velocity

```
# Example of how DVL measurements are generated
def get_measurement(self, quat=False):
    state = self.vessel_node.vessel.current_state

    # Get body-frame velocity and add noise
    v_body = state[0:3]
    v_body_noisy = v_body + np.random.multivariate_normal(np.zeros(3), self.vel_cov)

    return {...}  # Return velocity and covariance
```

Sensor Configuration

Sensors are configured through the YAML configuration files. Each sensor definition includes:

- **sensor_type**: The type of sensor (IMU, GPS, UWB, encoder, DVL)
- **sensor_topic**: ROS topic for publishing sensor data
- **sensor_location**: Position of the sensor in the body frame [x, y, z]
- **sensor_orientation**: Orientation of the sensor in the body frame [roll, pitch, yaw]
- **publish_rate**: Update frequency in Hz
- **use_custom_covariance**: Boolean flag to use custom noise parameters
- **custom_covariance**: Custom noise covariance matrices for each measurement

Example configuration for an IMU sensor:

```yaml
sensors:
  -
    sensor_type: IMU
    sensor_topic: /vessel_01/imu/data
    sensor_location: [0.0, 0.0, 0.1]
    sensor_orientation: [0.0, 0.0, 0.0]
    publish_rate: 100
    use_custom_covariance: true
    custom_covariance:
      orientation_covariance: [0.001, 0.0, 0.0, 0.0, 0.001, 0.0, 0.0, 0.0, 0.001]
      angular_velocity_covariance: [0.0001, 0.0, 0.0, 0.0, 0.0001, 0.0, 0.0, 0.0, 0.0001]
      linear_acceleration_covariance: [0.01, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.01]
```

Creating Custom Sensors

To create your own custom sensor, follow these steps:

1. Create a new sensor class

Create a new class that inherits from `BaseSensor` and implements the `get_measurement()` method:

```python
class CustomSensor(BaseSensor):
    def __init__(self, sensor_config, vessel_id, topic_prefix, vessel_node):
        super().__init__(sensor_config, vessel_id, topic_prefix, vessel_node)
        # Initialize sensor-specific properties
        self.location = np.array(sensor_config['sensor_location'])
        self.custom_param = sensor_config.get('custom_param', default_value)

        # Initialize noise parameters
        self.custom_rms = np.array([0.1, 0.1, 0.1])
        self.custom_cov = np.diag(self.custom_rms ** 2)

        # Override with custom covariance if provided
        if self.use_custom_covariance and self.custom_covariance:
            if 'custom_measurement_covariance' in self.custom_covariance:
                self.custom_cov =
                 np.array(self.custom_covariance['custom_measurement_covariance']).reshape(3

    def get_measurement(self, quat=False):
```

```
        # Access vessel state
        state = self.vessel_node.vessel.current_state

        # Generate realistic measurements based on state
        # Apply appropriate transformations, physics models, etc.

        # Add noise to measurements
        measurement = calculated_value +
 np.random.multivariate_normal(np.zeros(3), self.custom_cov)

        # Return measurement as a dictionary
        return {
            'custom_measurement': measurement,
            'covariance': self.custom_cov.flatten()
        }
```

2. Register your sensor in the factory function

Modify the `create_sensor` function to include your new sensor type:

```
def create_sensor(sensor_config, vessel_id, topic_prefix, vessel_node):
    """Factory function to create appropriate sensor object based on sensor type"""
    sensor_type = sensor_config['sensor_type']
    sensor_classes = {
        'IMU': IMUSensor,
        'GPS': GPSSensor,
        'UWB': UWBSensor,
        'encoder': EncoderSensor,
        'DVL': DVLSensor,
        'CustomSensor': CustomSensor  # Add your custom sensor here
    }

    if sensor_type not in sensor_classes:
        raise ValueError(f"Unknown sensor type: {sensor_type}")

    return sensor_classes[sensor_type](sensor_config, vessel_id, topic_prefix, vessel_node)
```

3. Create a ROS publisher for your custom sensor

In your vessel's ROS node, add a publisher for your custom sensor:

```python
# In the vessel ROS node initialization
if sensor.sensor_type == 'CustomSensor':
    # Create a publisher for your custom sensor
    # Choose an appropriate message type or create a custom one
    from your_package.msg import CustomSensorMsg

    publisher = self.create_publisher(
        CustomSensorMsg,
        f'{self.topic_prefix}/{sensor.topic}',
        10
    )

    # Add the publisher to the sensor publishers dictionary
    self.sensor_publishers[sensor] = publisher

# In the vessel ROS node update method
if sensor.sensor_type == 'CustomSensor':
    # Create a message for your custom sensor
    msg = CustomSensorMsg()

    # Fill the message with sensor data
    measurement = sensor.get_measurement()
    msg.data = measurement['custom_measurement']
    msg.covariance = measurement['covariance']

    # Add a timestamp
    msg.header.stamp = self.get_clock().now().to_msg()

    # Publish the message
    self.sensor_publishers[sensor].publish(msg)
```

4. Configure your custom sensor in the YAML file

Add your custom sensor to the sensors configuration file:

```yaml
sensors:
  -
    sensor_type: CustomSensor
    sensor_topic: /vessel_01/custom_sensor
    sensor_location: [0.1, 0.0, 0.2]
    sensor_orientation: [0.0, 0.0, 0.0]
```

```yaml
    publish_rate: 50
    custom_param: 42.0
    use_custom_covariance: true
    custom_covariance:
      custom_measurement_covariance: [0.01, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.01]
```

Best Practices for Sensor Simulation

When working with sensors in the simulator, follow these guidelines:

1. **Physical realism**:
   - Place sensors at realistic positions on the vessel
   - Use realistic noise parameters based on actual sensor specifications
   - Consider environmental effects if appropriate (e.g., GPS degradation underwater)
2. **Computational efficiency**:
   - Set appropriate update rates based on real-world sensors
   - Use optimized noise generation methods for high-frequency sensors
3. **Sensor fusion testing**:
   - Configure multiple sensor types to test fusion algorithms
   - Vary noise parameters to test robustness of estimation algorithms
4. **Custom sensors**:
   - Follow the class structure of existing sensors
   - Document physical models and assumptions clearly
   - Use appropriate coordinate transformations
5. **Topic naming**:
   - Use consistent naming conventions for sensors
   - Include vessel ID in topic names for multi-vessel simulations

Simualtion Inputs

Overview

The Panisim simulator uses a structured set of YAML configuration files to define all aspects of the simulation environment, vessel properties, sensor configurations and guidance. This modular approach allows for easy customization and extension of simulation scenarios without modifying the core simulation code. The input files are auto generated via the interactive Web GUI, but can also be modified manually.

> **Tip**
>
> All input files follow the YAML format, which is human-readable and easy to edit.

Input File Structure

The input file system follows a hierarchical structure (zoom in to see the chart more clearly).

The top-level `simulation_input.yml` file defines global simulation parameters and references vessel-specific configuration files stored in separate folders (one for each vessel type).

Main Configuration File

The `simulation_input.yml` file is the entry point for the simulator configuration. It specifies global parameters and the list of agents (vessels) to include in the simulation.

Example structure:

```yaml
sim_time: 100.0                # Total simulation time in seconds
time_step: 0.01                # Simulation time step in seconds
density: 1025.0                # Water density (kg/m³)
gravity: 9.80665               # Gravitational acceleration (m/s²)
world_size: [1000, 1000, 100] # Simulation world dimensions [x, y, z] in meters
gps_datum: [13.06, 80.28, 0]  # GPS reference point [latitude, longitude, height]
nagents: 1                     # Number of agents in the simulation
geofence: []                   # Optional geofencing boundaries

agents:
  - name: "sookshma"           # Vessel name
    type: "usv"                # Vessel type (usv, auv, etc.)
    active_dof: [1, 1, 0, 0, 0, 1]  # Active degrees of freedom [u, v, w, p, q, r]
    U: 0.0                     # Initial forward speed
    maintain_speed: false      # Whether to maintain constant forward speed

    # Paths to configuration files (relative or absolute)
    # {name} will be replaced with the vessel name
    geometry: "inputs/{name}/geometry.yml"
    inertia: "inputs/{name}/inertia.yml"
    hydrodynamics: "inputs/{name}/hydrodynamics.yml"
    control_surfaces: "inputs/{name}/control_surfaces.yml"
    thrusters: "inputs/{name}/thrusters.yml"
    initial_conditions: "inputs/{name}/initial_conditions.yml"
    sensors: "inputs/{name}/sensors.yml"
    guidance: "inputs/{name}/guidance.yml"
    control: "inputs/{name}/control.yml"
```

Key Parameters

Parameter	Description	Units
`sim_time`	Total simulation duration	seconds
`time_step`	Simulation time step size	seconds
`density`	Water density	kg/m^3
`gravity`	Gravitational acceleration	m/s^2
`world_size`	Simulation world dimensions [x, y, z]	meters
`gps_datum`	Reference point for GPS coordinates [lat, lon, height]	degrees, degrees, meters
`nagents`	Number of agents (vessels) in the simulation	integer

Parameter	Description	Units
`geofence`	Optional boundaries for the simulation area	meters

Agent Configuration

Each agent (vessel) in the `agents` list requires:

- `name`: Identifier for the vessel, used for file path resolution
- `type`: Type of vessel ("usv", "auv", etc.)
- `active_dof`: Array of 6 binary values indicating which degrees of freedom are active [surge, sway, heave, roll, pitch, yaw]
- `U`: Initial/desired forward speed
- `maintain_speed`: Boolean flag to maintain constant forward speed

File paths can be absolute or relative (although we would recommend using the same folder structure as the default). The special placeholder `{name}` will be replaced with the vessel name, allowing for a standardized folder structure.

Vessel-Specific Configuration Files

Each vessel has its own set of configuration files that define its physical properties, sensors, and control elements. This is defined in a folder named after the vessel.

Geometry Configuration (`geometry.yml`)

The geometry file defines the vessel's physical dimensions and coordinate frames.

Example from sookshma:

```yaml
# Dimensions of the box enclosing the vessel
length: 1
breadth: 0.24
depth: 0.5
Fn: 0.16
gyration: [0.096, 0.25, 0.25]
CO:
  position: [0.0, 0.0, 0.0] # Relative to the origin of the 3d software
  orientation: [0.0, 0.0, 0.0]
CG:
```

```
  position: [2.89415958e-02, 0.0, 1.62558889e-01]
CB:
  position: [2.89415958e-02, 0.0, 1.36000000e-01]

geometry_file: /workspaces/mavlab/inputs/sookshma/HydRA/input/sookshma.gdf
```

Key Parameters

Parameter	Description	Units
`length`	Overall length	meters
`breadth`	Overall width	meters
`depth`	Overall depth	meters
`Fn`	Froude number	dimensionless
`gyration`	Gyration radii [x, y, z]	meters
`CO`	Coordinate origin position and orientation	meters, radians
`CG`	Center of gravity position and orientation	meters, radians
`CB`	Center of buoyancy position and orientation	meters, radians
`geometry_file`	Path to 3D geometry file	string

Inertia Configuration (`inertia.yml`)

The inertia file defines the vessel's mass properties.

Example from sookshma:

```
mass: 7.65 # in kg
buoyancy_mass: 7.65 # in kg
inertia_matrix: None # in kg*m^2
added_mass_matrix: None # in kg*m^2/s
```

Key Parameters

Parameter	Description	Units
`mass`	Vessel mass	kg
`buoyancy_mass`	Mass of displaced fluid (for neutral buoyancy, equal to mass)	kg

Parameter	Description	Units
`inertia_matrix`	Optional explicit inertia matrix needs to be 6x6 (example: `[[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]]`)	$kg \cdot m^2$
`added_mass_matrix`	Optional explicit added mass matrix needs to be 6x6	kg, $kg \cdot m^2$, $kg \cdot m$

If the inertia matrix is not provided, it is calculated using the gyration radii and the mass via the `_generate_mass_matrix()` function in the `calculate_hydrodynamics.py` module. If the added mass matrix is not provided, it will be calculated either using the hydrodynamic derivatives or using the hydrodynamic coefficients from the HydRA model.

Hydrodynamics Configuration (`hydrodynamics.yml`)

The hydrodynamics file contains coefficients for damping forces and added mass.

Example from sookshma:

```yaml
hydra_file: /workspaces/mavlab/inputs/sookshma/HydRA/matlab/sookshma_hydra.json
dim_flag: false
cross_flow_drag: false
coriolis_flag: false # For linear dynamic model

# Surge hydrodynamic derivatives
X_u: -0.01

# Sway hydrodynamic derivatives
Y_v: -0.010659050686665396
Y_r: 0.0017799664706713543

# Yaw hydrodynamic derivatives
N_v: -0.0006242757399292063
N_r: -0.001889456681929024
```

Key Parameters

Parameter	Description	Units
`hydra_file`	Path to HydRA data file	string
`dim_flag`	Whether coefficients are dimensional (true) or non-dimensional (false)	boolean
`cross_flow_drag`	Whether to use cross-flow drag calculation	boolean
`coriolis_flag`	Whether to include Coriolis forces	boolean

You can enter any combination of the hydrodynamic coefficients, and it'll be handled by the simulator. They should be separated by underscores. For example, `X_u_u` or `M_r_r`.

Control Surfaces Configuration (`control_surfaces.yml`)

This file defines the vessel's control surfaces such as rudders, fins, or foils.

Example from sookshma:

```yaml
naca_number: &naca None
naca_file: &naca_file None
delta_max: &dmax 35.0
deltad_max: &ddmax 1.0
area: &area 0.1

control_surfaces:
  -
    control_surface_type: Rudder
    control_surface_id: 1
    control_surface_location: [0.0, 0.0, 0.0]    # With respect to BODY frame
    control_surface_orientation: [0.0, 0.0, 0.0]  # With respect to BODY frame
    control_surface_area: *area
    control_surface_NACA: *naca
    control_surface_T: 0.1
    control_surface_delta_max: *dmax
    control_surface_deltad_max: *ddmax
    control_surface_hydrodynamics:
      Y_delta: 0.02100751285923063
      N_delta: -0.006813248905845932
      X_delta: 0.0
```

Key Parameters

Parameter	Description	Units
`naca_number`	NACA airfoil designation	string
`naca_file`	Path to NACA airfoil data file	string
`delta_max`	Maximum deflection angle	degrees
`deltad_max`	Maximum deflection rate	degrees/s
`area`	Surface area	m^2
`control_surface_type`	Type of control surface (e.g., Rudder, Fin)	string
`control_surface_id`	Unique identifier	integer
`control_surface_location`	[x, y, z] position	meters
`control_surface_orientation`	[roll, pitch, yaw] orientation	radians
`control_surface_T`	Actuation time constant	seconds
`control_surface_hydrodynamics`	Force coefficients for the control surface	various

If `control_surface_hydrodynamics` is provided, the control surface location, orientation and naca file will be ignored. And the generalized force vector due to control surface will be calculated using the hydrodynamic coefficients.

Thruster Configuration (`thrusters.yml`)

This file defines the vessel's propulsion systems.

Example from sookshma (commented out):

```yaml
thrusters:
  -
    thruster_name: back
    thruster_id: 1
    thruster_location: [0.0, 0.0, 0.0]    # With respect to BODY frame
    thruster_orientation: [0.0, 0.0, 0.0]  # With respect to BODY frame
    T_prop: 1.0
    D_prop: 0.1
    tp: 0.1
    KT_at_J0: 0.0
    n_max: 2668 # in RPM
    J_vs_KT: None # File path to J_vs_KT.csv, Not being used currently
```

Key Parameters

Parameter	Description	Units
`thruster_name`	Descriptive name	string
`thruster_id`	Unique identifier	integer
`thruster_location`	[x, y, z] position	meters
`thruster_orientation`	[roll, pitch, yaw] orientation	radians
`T_prop`	Propeller thrust coefficient	dimensionless
`D_prop`	Propeller diameter	meters
`tp`	Thrust deduction coefficient	dimensionless
`KT_at_J0`	Thrust coefficient at zero advance ratio	dimensionless
`n_max`	Maximum RPM	RPM
`J_vs_KT`	Path to file with advance ratio vs thrust coefficient data	string

Initial Conditions Configuration (`initial_conditions.yml`)

This file defines the vessel's initial state for the simulation.

Example from sookshma:

```yaml
start_location: [0, 0, 0]
start_orientation: [0, 0, 0]
start_velocity: [0.5, 0, 0, 0, 0, 0]
use_quaternion: false
```

Key Parameters

Parameter	Description	Units
`start_location`	Initial [x, y, z] position	meters
`start_orientation`	Initial [roll, pitch, yaw] orientation	radians
`start_velocity`	Initial [u, v, w, p, q, r] velocity	m/s, rad/s
`use_quaternion`	Whether to use quaternions for orientation	boolean

Sensors Configuration (`sensors.yml`)

This file defines the sensors attached to the vessel.

Example from sookshma:

```yaml
# Custom covariance flag is applicable only when using GNC

sensors:
  -
    sensor_type: IMU
    sensor_topic: /sookshma_00/imu/data
    sensor_location: [0.0, 0.0, 0.0]
    sensor_orientation: [0.0, 0.0, 0.0]
    publish_rate: 50
    use_custom_covariance: true
    custom_covariance:
      orientation_covariance: [4.97116638e-07, 1.92100383e-07, -5.37921803e-06,
 1.92100383e-07, 4.19220441e-07, -2.48717925e-06
, -5.37921803e-06, -2.48717925e-06, 1.15176790e-04]
      linear_acceleration_covariance: [0.01973958, -0.01976063, 0.02346221,
 -0.01976063, 0.0211394, -0.02188356,
 0.02346221, -0.02188356, 0.03132967]
      angular_velocity_covariance: [5.28022053e-05, 4.08840955e-05, -1.15368805e-05,
 4.08840955e-05, 3.58062060e-05, -8.83069166e-06,
 -1.15368805e-05, -8.83069166e-06, 5.01080310e-06]

  -
    sensor_type: UWB
    sensor_topic: /sookshma_00/uwb
    sensor_location: [0.0, 0.0, 0.0]
    sensor_orientation: [0.0, 0.0, 0.0]
    publish_rate: 1
    use_custom_covariance: true
    custom_covariance:
      position_covariance: [0.04533883, -0.05014115, 0.0,
 -0.05014115, 0.05869406, 0.0,
 0.0, 0.0, 0.00000001]

  -
    sensor_type: encoder
    actuator_type: Rudder
    actuator_id: 1
```

```
    sensor_topic: /sookshma_00/Rudder_1/encoder
    publish_rate: 10
```

Key Parameters

Parameter	Description	Units
`sensor_type`	Type of sensor (IMU, GPS, UWB, DVL, encoder, etc.)	string
`sensor_topic`	ROS topic for publishing sensor data	string
`sensor_location`	[x, y, z] position	meters
`sensor_orientation`	[roll, pitch, yaw] orientation	radians
`publish_rate`	Data publication frequency	Hz
`use_custom_covariance`	Whether to use custom noise covariance	boolean
`custom_covariance`	Sensor-specific covariance matrices	various
`actuator_type`	For encoders, the type of actuator	string
`actuator_id`	For encoders, the ID of the actuator	integer

Guidance Configuration (`guidance.yml`)

This file defines waypoints and guidance parameters for autonomous navigation as used in the PID Waypoint Tracking Example.

Example from sookshma:

```
waypoints:
- [0, 0, 0]
- [-4, -10, 0]
- [8, -5, 0]
- [9, 9, 0]
- [0, 0, 0]
waypoints_type: XYZ
```

Key Parameters

Parameter	Description	Units
`waypoints`	List of waypoint coordinates	meters

Parameter	Description	Units
`waypoints_type`	Coordinate system for waypoints (XYZ, LLA, NED)	string
`lookahead_distance`	Optional: Distance for lookahead-based guidance	meters
`acceptance_radius`	Optional: Radius to consider waypoint reached	meters
`guidance_type`	Optional: Type of guidance algorithm	string

NACA Airfoil Data (example: `NACA0015.csv`)

This file contains aerodynamic/hydrodynamic coefficients for the NACA0015 airfoil profile, used for modeling control surfaces like rudders. The file includes:

- `Alpha`: Angle of attack in degrees
- `Cl`: Lift coefficient
- `Cd`: Drag coefficient
- `Cdp`: Pressure drag coefficient
- `Cm`: Pitching moment coefficient
- `Top_Xtr`: Top surface transition location
- `Bot_Xtr`: Bottom surface transition location

Sample excerpt:

```
Alpha,Cl,Cd,Cdp,Cm,Top_Xtr,Bot_Xtr
-19.750,-1.2970,0.09858,0.09490,-0.0142,1.0000,0.0253
-19.500,-1.3072,0.09342,0.08964,-0.0166,1.0000,0.0255
...
0.000,0.0000,0.00632,0.00147,0.0000,0.6107,0.6107
...
19.750,1.2970,0.09858,0.09490,0.0142,0.0253,1.0000
```

This data is used to calculate the forces and moments generated by control surfaces at different angles of attack. This file is obtained from Airfoil Tools website.

Input File Processing

The `read_input.py` module is responsible for loading and processing all input files. The main function is `read_input()`, which:

1. Loads the main simulation configuration file
2. Extracts global simulation parameters
3. Processes each vessel's configuration files
4. Transforms component positions and orientations to be relative to the vessel's coordinate origin (CO)
5. Returns the processed simulation parameters and agent configurations

Key Functions

`read_input(input_file)`

The main entry point that reads the entire configuration:

```
def read_input(input_file: str = None) -> Tuple[Dict, List[Dict]]:
    """Read vessel parameters and configuration from input files.

    Args:
        input_file: Path to the main simulation input file. If None, uses default.

    Returns:
        Tuple containing:
            sim_params (Dict): Global simulation parameters
            agents (List[Dict]): List of agents, each containing vessel config and hydrodynar
    """
```

`load_yaml(file_path)`

Loads a YAML file and returns its contents:

```
def load_yaml(file_path: str) -> Dict:
    """Load a YAML file and return its contents."""
    with open(file_path, 'r') as file:
        return yaml.safe_load(file)
```

`resolve_path(base_path, file_path, name)`

Resolves file paths, replacing `{name}` with the actual vessel name:

```
def resolve_path(base_path: str, file_path: str, name: str) -> str:
    """Resolve a file path, replacing {name} with the actual vessel name."""
    resolved = file_path.replace('{name}', name)
    if not os.path.isabs(resolved):
        resolved = os.path.join(base_path, resolved)
    return resolved
```

transform_to_co_frame(vessel_config)

Transforms all component positions and orientations to be relative to the vessel's coordinate origin (CO):

```
def transform_to_co_frame(vessel_config: Dict) -> Dict:
    """Transform all component positions and orientations to be relative to the CO frame."""
    # Implementation details...
```

Cross-Flow Drag Generation

The `read_input` function also initializes a `CrossFlowGenerator` to calculate cross-flow drag coefficients:

```
cross_flow_generator = CrossFlowGenerator(gdf_file, hydra_file, hydrodynamics_file,
                                          initial_conditions_file, agent['type'])
cross_flow_generator.update_yaml_file()
```

This automatically computes and updates the hydrodynamic coefficients for sway-yaw (and heave-pitch for AUVs) using strip theory and Hoerner's cross-flow drag formulation. This happens if the `cross_flow_drag` flag is set to `true` in the `hydrodynamics.yml` file.

Common Customizations

Here are some common customizations you might want to make to the input files:

Changing Vessel Dynamics

To adjust a vessel's dynamic behavior:

1. Modify hydrodynamic coefficients in `hydrodynamics.yml`
2. Adjust mass and inertia properties in `inertia.yml`
3. Change the center of gravity or buoyancy in `geometry.yml`

Adding or Modifying Sensors

To add a new sensor:

1. Add a new sensor configuration to `sensors.yml` with appropriate parameters
2. Set a realistic update rate and noise characteristics
3. Position the sensor at an appropriate location on the vessel

Creating a New Vessel

To create a new vessel type:

1. Create a new folder with the vessel name
2. Copy and modify the configuration files from an existing vessel
3. Adjust all physical parameters to match the new vessel's characteristics
4. Add the new vessel to the `agents` list in `simulation_input.yml`

Example: Complete Vessel Configuration

Here's an example of a complete vessel configuration for a surface vessel:

1. **simulation_input.yml**:

```yaml
sim_time: 100.0
time_step: 0.01
density: 1025.0
gravity: 9.80665
world_size: [1000, 1000, 100]
gps_datum: [13.06, 80.28, 0]
nagents: 1

agents:
  - name: "example_vessel"
```

```
    type: "usv"
    active_dof: [1, 1, 0, 0, 0, 1]
    U: 2.0
    maintain_speed: false
    geometry: "inputs/{name}/geometry.yml"
    inertia: "inputs/{name}/inertia.yml"
    hydrodynamics: "inputs/{name}/hydrodynamics.yml"
    control_surfaces: "inputs/{name}/control_surfaces.yml"
    thrusters: "inputs/{name}/thrusters.yml"
    initial_conditions: "inputs/{name}/initial_conditions.yml"
    sensors: "inputs/{name}/sensors.yml"
    guidance: "inputs/{name}/guidance.yml"
```

2. **geometry.yml**:

```
geometry_file: "inputs/example_vessel/geometry.gdf"
length: 5.0
breadth: 1.5
depth: 1.0

CO:
  position: [0.0, 0.0, 0.0]
  orientation: [0.0, 0.0, 0.0]

CG:
  position: [0.0, 0.0, 0.2]
  orientation: [0.0, 0.0, 0.0]

CB:
  position: [0.0, 0.0, -0.3]
  orientation: [0.0, 0.0, 0.0]

gyration: [1.5, 1.5, 0.8]
```

3. **inertia.yml**:

```
mass: 1000.0
buoyancy_mass: 1000.0
inertia_matrix: "None"
added_mass_matrix: "None"
```

4. **hydrodynamics.yml**:

```yaml
dim_flag: false
hydra_file: "inputs/example_vessel/hydrodynamics.json"
coriolis_flag: true

X_u: -50.0
X_u_au: -100.0
Y_v: -200.0
Y_v_av: -400.0
Y_r: -50.0
N_v: -50.0
N_r: -300.0
N_r_ar: -400.0
```

5. **control_surfaces.yml**:

```yaml
naca_file: "inputs/naca0015.csv"

control_surfaces:
  - control_surface_id: 1
    control_surface_name: "Rudder"
    control_surface_location: [2.4, 0.0, 0.0]
    control_surface_orientation: [0, 0, 1.57]
    control_surface_area: 0.5
    control_surface_T: 0.5
    control_surface_delta_max: 35.0
    control_surface_deltad_max: 20.0
    control_surface_hydrodynamics: "None"
```

6. **thrusters.yml**:

```yaml
thrusters:
  - thruster_id: 1
    thruster_name: "Main Propeller"
    thruster_location: [-2.4, 0.0, 0.0]
    thruster_orientation: [0.0, 0.0, 0.0]
    D_prop: 0.4
    KT_at_J0: 0.5
    n_max: 1500
    tp: 0.05
```

7. **initial_conditions.yml**:

```yaml
start_location: [0.0, 0.0, 0.0]
start_orientation: [0.0, 0.0, 0.0]
start_velocity: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
use_quaternion: false
```

8. **sensors.yml**:

```yaml
sensors:
  - sensor_type: "IMU"
    sensor_topic: "/vessel/imu"
    sensor_location: [0.0, 0.0, 0.1]
    sensor_orientation: [0.0, 0.0, 0.0]
    publish_rate: 100
    use_custom_covariance: true
    custom_covariance:
      orientation_covariance: [0.01, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.01]
      angular_velocity_covariance: [0.01, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.01]
      linear_acceleration_covariance: [0.05, 0.0, 0.0, 0.0, 0.05, 0.0, 0.0, 0.0, 0.05]
```

9. **guidance.yml**:

```yaml
waypoints:
  - [0.0, 0.0, 0.0]
  - [50.0, 0.0, 0.0]
  - [50.0, 50.0, 0.0]
  - [0.0, 50.0, 0.0]
  - [0.0, 0.0, 0.0]
waypoints_type: XYZ
```

ROS2 Architecture

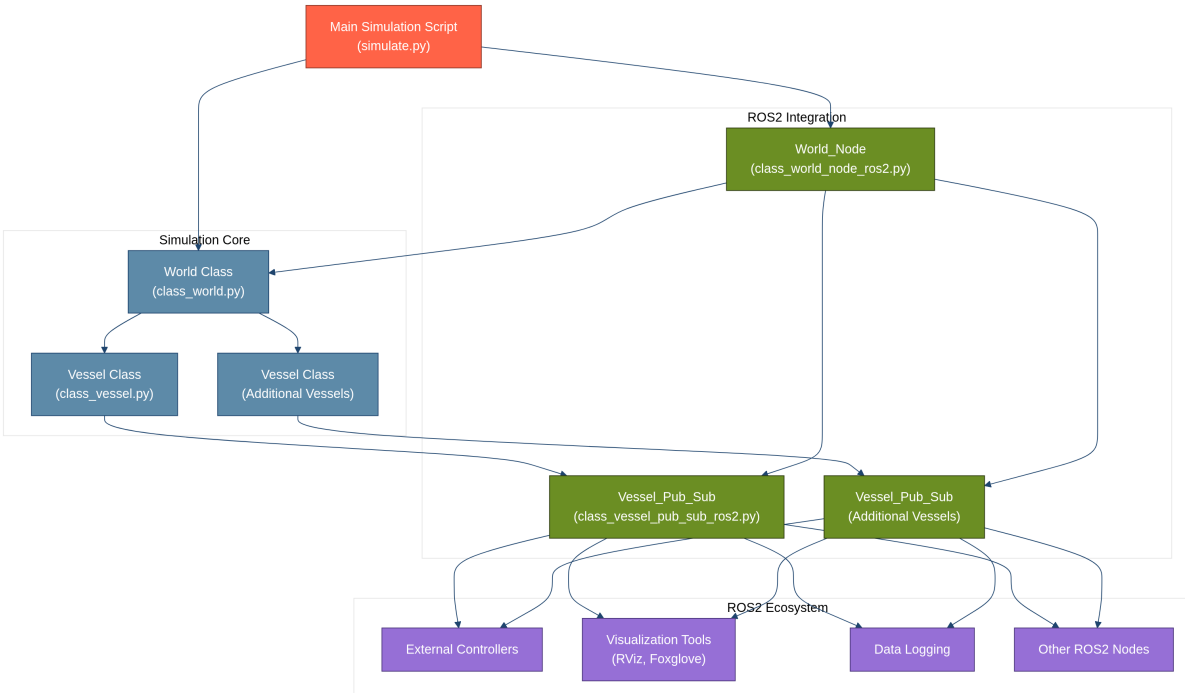
Overview

The Panisim runs on the Robot Operating System 2 (ROS2), enabling distributed simulation, external control, visualization, and data recording. The ROS2 integration architecture follows a modular design pattern that separates the simulation core from the communication layer. The Docker container of the simlator runs ROS2 Humble.

> Tip
>
> The ROS2 integration allows the Panisim to run completely isolated from controllers or guidance systems, thus enabling you to code your own controllers or guidance systems to interact with the simulator.

Architecture

The ROS2 integration consists of several key components that work together to bridge the simulation with the ROS2 ecosystem:



Explanation of the Architecture Flow:

- **Main Simulation Script (`simulate.py`)**: Initiates the simulation by:
  - Calling `rclpy.init()` to initialize the ROS2 middleware
  - Creating a `World` instance by calling `World(config_file_path)`
  - Instantiating a `World_Node` that wraps the simulation in ROS2
  - Calling `world.start_vessel_ros_nodes(world_node)` to connect vessels to ROS2
- **World Class (`class_world.py`)**:
  - Maintains multiple vessel instances using `vessels = []`
  - Processes world input with `process_world_input(world_file)`
  - Creates vessels via `Vessel(vessel_params, hydrodynamic_data, vessel_id)`
  - Updates simulation state with `step()` method that calls `vessel.step()` for each vessel
- **World_Node Class (`class_world_node_ros2.py`)**:
  - Inherits from ROS2 `Node` class
  - Creates a timer with `create_timer(1/self.rate, self.world.step)` to drive simulation
- **Vessel_Pub_Sub Class (`class_vessel_pub_sub_ros2.py`)**:
  - Creates publishers, subscribers, and timers for each vessel

- Publishes vessel state and sensor data
- Handles actuator commands via subscribers

Key Components

Main Simulation Script (`simulate.py`)

The entry point for running Panisim with ROS2 integration. This script:

1. Initializes the ROS2 environment
2. Creates the World simulation instance
3. Sets up the World_Node for ROS2 communication
4. Establishes connections between simulation and ROS2
5. Runs the ROS2 spin loop in a separate thread

```
def main():
    # Creates an object of class 'World'
    rclpy.init()
    world = World('/workspaces/mavlab/inputs/simulation_input.yml')
    world_node = World_Node(world_rate=1/world.dt)

    world.node = world_node
    world.start_vessel_ros_nodes(world_node)

    # Run ROS on a separate thread
    ros_thread_instance = threading.Thread(target=ros_thread, args=(world_node,))
    ros_thread_instance.start()

    # Prints available ROS2 topics and usage information
    # ...
```

The `ros_thread` function is defined as:

```
def ros_thread(node):
    rclpy.spin(node)
    rclpy.shutdown()
```

This function handles the ROS2 event loop, allowing callbacks to be processed in the background while the main thread can perform other tasks.

World Class (`class_world.py`)

The World class serves as the simulation environment container, managing:

- Multiple vessel instances within a unified simulation
- Initialization from configuration files
- Simulation time stepping across all vessels
- Global parameters like GPS datum and world boundaries

```
class World():
    terminate = False                   # Flag indicating simulation termination
    vessels = []                        # List of Vessel objects
    nvessels = 0                        # Number of vessels
    size = np.zeros(3)                  # Size of the world (X-Y-Z)
    gps_datum = None                    # GPS reference point
    node = None                         # ROS2 node reference
    dt = 0.01                           # Simulation time step

    def __init__(self, world_file=None):
        """Initialize the World object from configuration file"""
        if world_file is not None:
            self.process_world_input(world_file)

    def step(self):
        """Advance the simulation by one time step"""
        for vessel in self.vessels:
            vessel.step()
```

The `process_world_input` method reads the simulation configuration from YAML files:

```
def process_world_input(self, world_file=None):
    try:
        sim_params, agents = read_input(world_file)
        self.size = np.array(sim_params['world_size'])
        self.nvessels = sim_params['nagents']
        self.gps_datum = np.array(sim_params['gps_datum'])
        agent_count = 0
        self.dt = sim_params['time_step']

        for agent in agents[0:self.nvessels]:
            vessel_config = agent['vessel_config']
            hydrodynamic_data = agent['hydrodynamics']
```

```
            self.vessels.append(Vessel(vessel_params=vessel_config,
                                       hydrodynamic_data=hydrodynamic_data,
                                       vessel_id=agent_count))
            agent_count += 1
    except yaml.YAMLError as exc:
        print(exc)
        exit()
```

The World class includes a method to start the ROS2 nodes for all vessels:

```
def start_vessel_ros_nodes(self, world_node):
    """Initialize ROS2 nodes for all vessels in the world"""
    for vessel in self.vessels:
        vessel.vessel_node = Vessel_Pub_Sub(vessel, world_node)
```

World_Node Class (`class_world_node_ros2.py`)

The World_Node class is a ROS2 node wrapper for the World class:

```
class World_Node(Node):
    rate = None
    def __init__(self, world_rate=100, world_file=None):
        super().__init__('world')
        self.rate = world_rate
        self.world = World(world_file)
        self.create_timer(1/self.rate, callback=self.world.step)
```

This class: - Inherits from the ROS2 Node class - Creates a timer that triggers the simulation step function at a specified rate - Provides the ROS2 context for the World class

The `create_timer` method sets up a timer that calls the `World.step()` method at regular intervals determined by `world_rate`. This is what drives the simulation forward in time while maintaining synchronization with the ROS2 ecosystem.

Vessel_Pub_Sub Class (`class_vessel_pub_sub_ros2.py`)

The Vessel_Pub_Sub class implements the ROS2 communication interface for vessel objects:

```python
class Vessel_Pub_Sub():
    def __init__(self, vessel, world_node):
        """Initialize the vessel interface"""
        self.world_node = world_node
        self.vessel = vessel
        self.vessel_id = vessel.vessel_id
        self.vessel_name = vessel.vessel_name
        self.topic_prefix = f'{self.vessel_name}_{self.vessel_id:02d}'

        # Initialize publishers, subscribers, and timers
        # ...
```

This class:

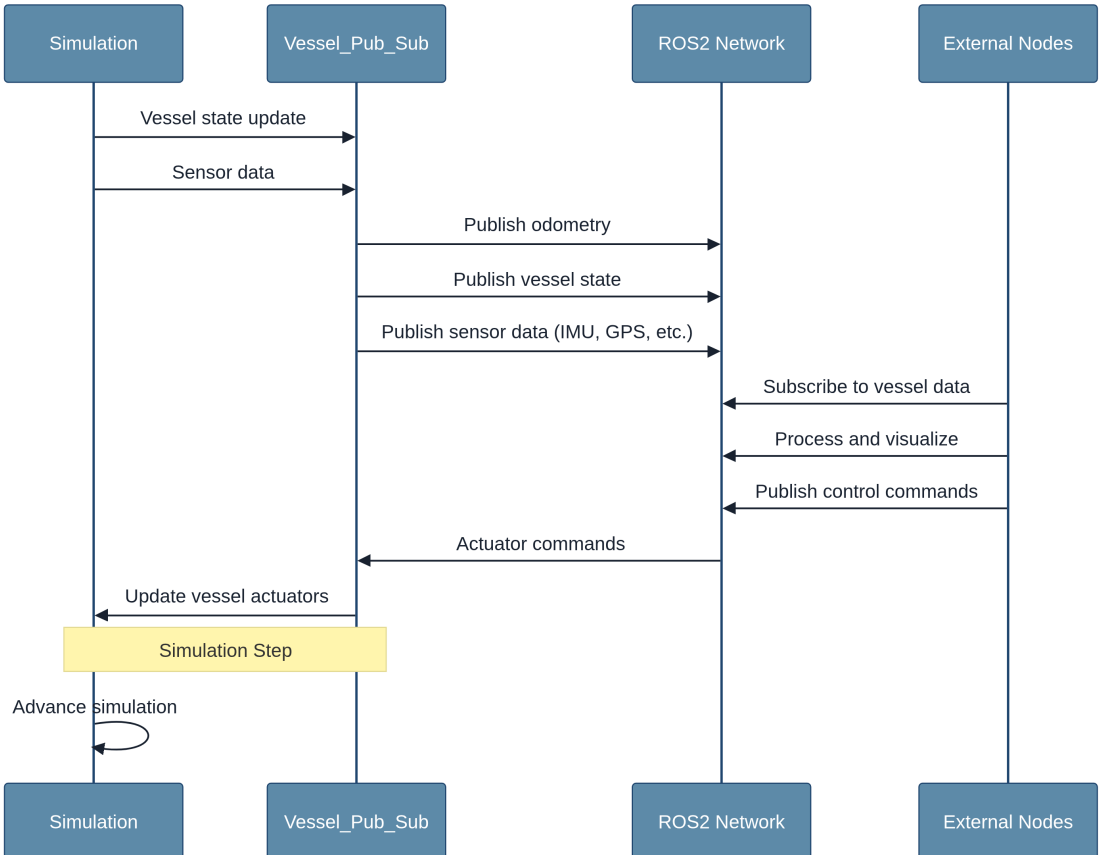
- Creates and manages ROS2 publishers for vessel state and sensor data
- Sets up subscribers for actuator commands
- Handles message conversion between simulator data and ROS2 messages
- Manages sensor data publication at appropriate rates

Key initialization methods include:

- Setting up control surface and thruster mappings
- Creating sensor objects and their publishers
- Initializing odometry and vessel state publishers
- Creating actuator command subscribers

Message Flow

The communication between simulation components and the ROS2 ecosystem follows this message flow:



Detailed Function Call Sequence:

1. **Vessel State Update**:
   - `vessel.step()` in `World.step()` updates the vessel state
   - This updates `vessel.current_state`, which is accessed by publishers
2. **Sensor Data Generation**:
   - Sensor objects call `get_measurement()` to generate simulated sensor readings based on vessel state
3. **Publishing Data to ROS2**:
   - `publish_odometry()` is called by a timer to publish vessel position and orientation
   - `publish_vessel_state()` is called by a timer to publish complete state vector
   - `_publish_sensor()` is called by sensor-specific timers to publish sensor data
4. **External Nodes Subscribing**:
   - External ROS2 nodes subscribe to topics using `ros2 topic echo` or programmatically

5. **Control Commands**:
    - External nodes publish to `/<vessel_name>_<id>/actuator_cmd`
    - `actuator_callback(msg)` processes these commands
6. **Updating Actuator States**:
    - `vessel.delta_c` (control surfaces) and `vessel.n_c` (thrusters) are updated
    - These affect the vessel dynamics in the next simulation step
7. **Simulation Step**:
    - The timer in `World_Node` triggers `world.step()`
    - This advances all vessels' states using their dynamics models

Topics and Messages

Each vessel in the simulation publishes data on several ROS2 topics:

Standard Topics

Topic	Message Type	Description
`/<vessel_name>_<id>/odometry_sim`	`nav_msgs/Odometry`	Vessel position, orientation, and velocities
`/<vessel_name>_<id>/vessel_state`	`std_msgs/Float64MultiArray`	Complete vessel state including actuators
`/<vessel_name>_<id>/actuator_cmd`	`interfaces/Actuator`	Commands for control surfaces and thrusters

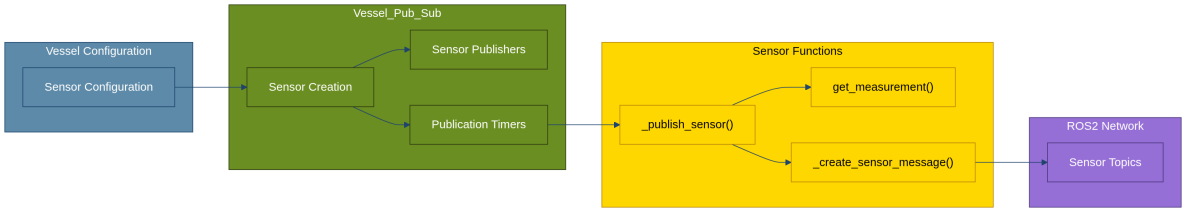
Sensor Topics

Each sensor configured for a vessel publishes on its own topic:

Sensor	Topic	Message Type
IMU	`/<vessel_name>_<id>/imu`	`sensor_msgs/Imu`
GPS	`/<vessel_name>_<id>/gps`	`sensor_msgs/NavSatFix`
DVL	`/<vessel_name>_<id>/dvl`	`interfaces/DVL`
UWB	`/<vessel_name>_<id>/uwb`	`geometry_msgs/PoseWithCovarianceStamped`

Sensor Integration

Sensors are created and managed through the Vessel_Pub_Sub class:



Detailed Function Calls in Sensor Integration:

1. **SensorConfig to SensorCreation**:
   - During `Vessel_Pub_Sub` initialization, it reads sensor configurations from `vessel.vessel_config['sensors']`
   - For each sensor configuration, it calls `create_sensor(sensor_config, vessel_id, topic_prefix, self)`
2. **SensorCreation to Publishers and Timers**:
   - After creating a sensor object, `Vessel_Pub_Sub` creates a publisher using:

     ```
     self.world_node.create_publisher(self._get_msg_type(sensor.sensor_type), sensor_topi
     ```

   - It also creates a timer for each sensor:

     ```
     self.world_node.create_timer(1/sensor.rate, lambda s=sensor: self._publish_sensor(s)
     ```

3. **Timer to PublishSensor**:
   - Each timer periodically calls `_publish_sensor(sensor)` at the sensor's configured rate
4. **PublishSensor to GetMeasurement**:
   - The `_publish_sensor` method calls `sensor.get_measurement()` to obtain the latest sensor reading
   - Each sensor type has its own implementation of `get_measurement()`
5. **PublishSensor to CreateMessage**:
   - After getting the measurement, `_publish_sensor` calls `_create_sensor_message(sensor_type, measurement)`
   - This converts the measurement to the appropriate ROS2 message type
6. **CreateMessage to Topics**:

- Finally, the message is published to the ROS2 network:

```
s['pub'].publish(msg)
```

The process for sensor integration:

1. Sensor configurations are defined in the vessel parameters (input files `sensors.yml`)
2. The Vessel_Pub_Sub class creates sensor objects using the `create_sensor` factory function
3. For each sensor, a publisher and timer are created
4. The timer periodically calls the sensor's `get_measurement()` method
5. Measurements are converted to ROS2 messages and published

Key methods in the sensor integration:

```
def _publish_sensor(self, sensor):
    """Publish sensor data."""
    measurement = sensor.get_measurement()
    msg = self._create_sensor_message(sensor.sensor_type, measurement)

    # Find the publisher for this sensor
    for s in self.sensors:
        if s['sensor'] == sensor:
            s['pub'].publish(msg)
            break
```

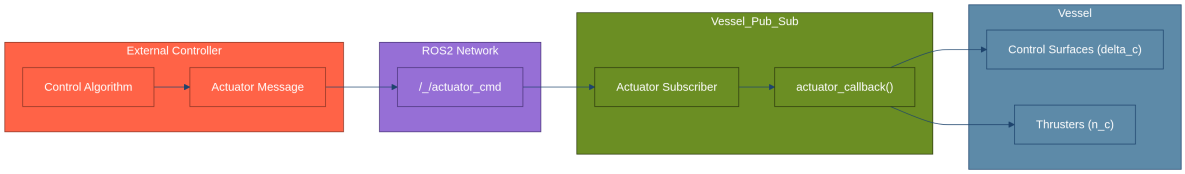
```
def _create_sensor_message(self, sensor_type, measurement):
    """Create a ROS message for sensor data."""
    current_time = self.world_node.get_clock().now().to_msg()

    # Create appropriate message based on sensor type
    if sensor_type == 'IMU':
        msg = Imu()
        # Fill message fields
    elif sensor_type == 'GPS':
        msg = NavSatFix()
        # Fill message fields
    # ... other sensor types

    return msg
```

Actuator Control

Vessels can be controlled through ROS2 by sending actuator commands:



Detailed Function Calls in Actuator Control:

1. **ControlAlgorithm to ActuatorMsg**:
   - External control algorithms compute desired actuator values
   - They create an `Actuator` message with the desired values
2. **ActuatorMsg to ActuatorTopic**:
   - The controller publishes the message to the actuator command topic
   - This is done using ROS2's `publisher.publish(msg)` mechanism
3. **ActuatorTopic to Subscriber**:
   - The subscriber in `Vessel_Pub_Sub` receives the message
   - This was set up during initialization with:

```
self.actuator_sub = self.world_node.create_subscription(
    Actuator,
    f'{self.topic_prefix}/actuator_cmd',
    self.actuator_callback,
    1
)
```

4. **Subscriber to Callback**:
   - The subscriber triggers `actuator_callback(msg)` in `Vessel_Pub_Sub`
5. **Callback to ControlSurfaces and Thrusters**:
   - The callback parses the message and updates vessel control values:

```
# For control surfaces
self.vessel.delta_c[idx] = value * np.pi / 180.0

# For thrusters
self.vessel.n_c[idx] = value
```

Actuator Message Structure (`Actuator.msg`)

The `Actuator.msg` is a custom message type used for controlling vessel actuators. Its structure is:

```
std_msgs/Header header       # Standard header with timestamp
string[] actuator_names      # Array of actuator identifiers
float64[] actuator_values    # Array of corresponding values
```

The actuator names use a prefixing convention: - `cs_N` for control surfaces (e.g., `cs_1` for the first control surface) - `th_N` for thrusters (e.g., `th_1` for the first thruster)

Values for control surfaces are specified in **degrees** (converted to radians internally) and values for thrusters are in **RPM** (Revolutions Per Minute).

Example usage in `class_vessel_pub_sub_ros2.py`:

```python
def actuator_callback(self, msg):
    """Handle actuator command messages."""
    if len(msg.actuator_names) != len(msg.actuator_values):
        self.world_node.get_logger().warn('Mismatch between actuator IDs and values length')
        return

    for actuator_id, value in zip(msg.actuator_names, msg.actuator_values):
        try:
            if actuator_id.startswith('cs_'):
                # Handle control surface (convert degrees to radians)
                if actuator_id in self.control_surface_ids:
                    idx = self.control_surface_ids[actuator_id]
                    self.vessel.delta_c[idx] = value * np.pi / 180.0
                else:
                    self.world_node.get_logger().warn(f'Unknown control surface ID:
                    {actuator_id}')

            elif actuator_id.startswith('th_'):
                # Handle thruster (RPM value)
                if actuator_id in self.thruster_ids:
                    idx = self.thruster_ids[actuator_id]
                    self.vessel.n_c[idx] = value
                else:
                    self.world_node.get_logger().warn(f'Unknown thruster ID: {actuator_id}')

            else:
```

```
                self.world_node.get_logger().warn(f'Invalid actuator ID format:
                {actuator_id}. Must start with cs_ or th_')
        except (ValueError, IndexError):
            self.world_node.get_logger().warn(f'Unknown actuator ID: {actuator_id}')
```

During initialization, `Vessel_Pub_Sub` builds mappings between actuator IDs (as defined in `control_surfaces.yml` and `thrusters.yml`) and their indices:

```
# Store actuator IDs with type prefixes
self.control_surface_ids = {}  # Maps 'cs_id' to index
self.thruster_ids = {}         # Maps 'th_id' to index

for idx, cs in enumerate(self.control_surfaces):
    cs_id = cs.get('control_surface_id', idx+1)
    self.control_surface_ids[f'cs_{cs_id}'] = idx

for idx, th in enumerate(self.thrusters):
    th_id = th.get('thruster_id', idx+1)
    self.thruster_ids[f'th_{th_id}'] = idx
```

These mappings ensure that actuator commands are properly routed to the correct control surfaces and thrusters in the vessel's internal state.

World and Vessel Population

The process of populating the world with vessels follows these steps:

Detailed Function Calls in World and Vessel Population:

1. **Start to Init**:
   - `simulate.py` starts and calls `rclpy.init()` to initialize the ROS2 middleware
   - This initializes the ROS2 context for the application
2. **Init to CreateWorld**:
   - `World('/workspaces/mavlab/inputs/simulation_input.yml')` is called
   - This creates the World instance and loads configuration
3. **CreateWorld to ReadConfig**:

- `World.__init__` calls `self.process_world_input(world_file)`
- `process_world_input` calls `read_input(world_file)` to parse YAML

4. **ReadConfig to ParseWorld**:
   - `process_world_input` extracts world parameters like:
     - `self.size = np.array(sim_params['world_size'])`
     - `self.nvessels = sim_params['nagents']`
     - `self.gps_datum = np.array(sim_params['gps_datum'])`
5. **ParseWorld to CreateVessels**:
   - `process_world_input` creates vessel instances:

```
self.vessels.append(Vessel(vessel_params=vessel_config,
                           hydrodynamic_data=hydrodynamic_data,
                           vessel_id=agent_count))
```

6. **CreateWorld to CreateNode**:
   - `World_Node(world_rate=1/world.dt)` is called
   - This creates a ROS2 node wrapper for the World
7. **SetNode to StartVessels**:
   - `world.start_vessel_ros_nodes(world_node)` is called
   - This iterates through vessels and creates ROS2 interfaces
8. **StartVessels to CreateVesselPubSub**:
   - `Vessel_Pub_Sub(vessel, world_node)` is called for each vessel
   - Each vessel gets its own ROS2 communication interface
9. **CreateVesselPubSub to SetupPublishers**:
   - Publishers are created for each vessel's odometry and state:

```
self.odometry = {
    'pub': self.world_node.create_publisher(Odometry,
    f'{self.topic_prefix}/odometry_sim', 10),
    'timer': self.world_node.create_timer(self.vessel.vessel_config[
    'time_step'], self.publish_odometry)
}
```

10. **CreateVesselPubSub to SetupSubscribers**:
    - Subscribers are created for actuator commands:

```
self.actuator_sub = self.world_node.create_subscription(
    Actuator,
    f'{self.topic_prefix}/actuator_cmd',
    self.actuator_callback,
    1
)
```

11. **CreateVesselPubSub to SetupSensors**:
    - Sensors are created and configured based on vessel configuration:

```
sensor = create_sensor(sensor_config, self.vessel_id, self.topic_prefix, self)
self.sensors.append({
    'sensor': sensor,
    'pub': self.world_node.create_publisher(self._get_msg_type(sensor.sensor_type),
    'timer': self.world_node.create_timer(
    1/sensor.rate, lambda
     s=sensor: self._publish_sensor(s))
})
```

12. **StartVessels to SpinThread**:
    - `threading.Thread(target=ros_thread, args=(world_node,))` is created and started
    - This runs `rclpy.spin(node)` in a separate thread
13. **SpinThread to PrintInfo**:
    - Available ROS2 topics are listed and helpful usage information is printed

Key steps in this process:

1. The simulation begins in `simulate.py`
2. ROS2 is initialized
3. A World object is created from the input configuration file
4. A World_Node is created and linked to the World object
5. Vessel ROS nodes are started for each vessel in the world
6. For each vessel, a Vessel_Pub_Sub object is created, which:
    - Sets up publishers for odometry and vessel state
    - Sets up subscribers for actuator commands
    - Creates sensor objects based on the vessel configuration
    - Sets up timers for publishing sensor data

Running the Simulation

To run the MAV simulator with ROS2 integration:

```
# Navigate to the workspace
cd /path/to/makara

# Build ROS2 packages
colcon build

# Source the workspace
source install/setup.bash

# Run the simulator
ros2 run mav_simulator simulate
```

After starting the simulation, you can interact with it using standard ROS2 commands:

```
# List all available topics
ros2 topic list

# Subscribe to vessel odometry
ros2 topic echo /<vessel_name>_<id>/odometry_sim

# Publish control commands
ros2 topic pub /<vessel_name>_<id>/actuator_cmd interfaces/Actuator \
"{header: {stamp: {sec: 0}}, actuator_names: ['cs_1', 'th_1'],actuator_values: [15.0, 1200.0]
```

Best Practices

- **Launch Files**: Create launch files for different simulation scenarios. There are some example launch files in the `/workspace/mavlab/ros2_ws/src/mav_simulator/mav_simulator/launch` folder.

Vessel Configurator

Overview

The Vessel Configurator is a light weight web-based application that provides an intuitive interface for configuring, and generating input files for the Panisim simulator. This tool eliminates the need to manually create YAML configuration files, offering instead a visual approach to visualize a vessel in 3D and configure its components.

Previosly, without the web GUI, one would have to manually enter entries into the YAML files, which was very time consuming and error prone. The major parameters entered in the YAML files are the position and orientation of the various components such as thrusters, control surfaces, sensors etc. With the Web GUI, you simply load a FBX file into the configurator, select your components and the software logs the position and orientation of the components automatically still allowing you to make modifications. You can select the Centre of Buoyancy, Centre of Gravity and Body Centre visually and Panisim takes care of the frame of reference conversion (to the body frame) automatically.

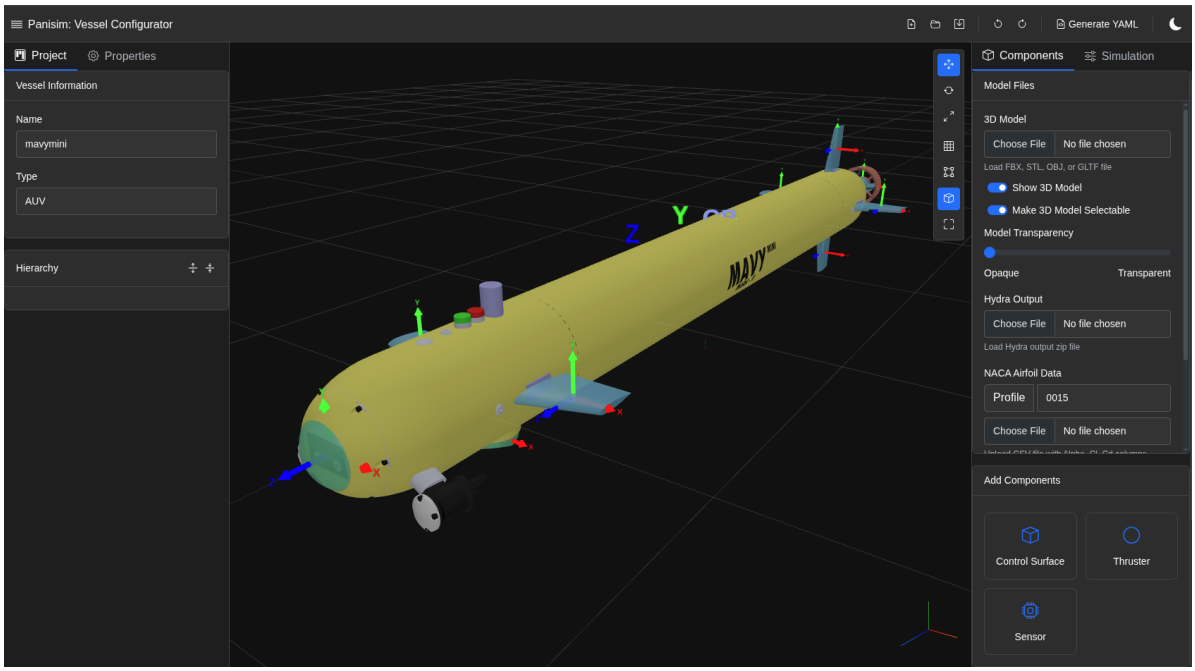


Figure 5: An example configuration in the vessel configurator

> **Tip**
>
> The vessel configurator was largely built using AI!

> **Note**
>
> The configurator is still needs development and some features may be unstable. The `guidance.yml` and `control.yml` files will still need to be manually edited.

Tutorial

To Quickstart and playaround, you can load the pre-configured `mavymini_config.json` vessel model in the `/inputs/example configurations` folder. Or follow the step below to configure your own vessel.

Step 1: Load a Vessel Model

1. Open the vessel configurator in your web browser

2. Click on the “Choose Vessel” button
3. Select the FBX file of the vessel model you want to configure

Now you should see the vessel model in the 3D view.

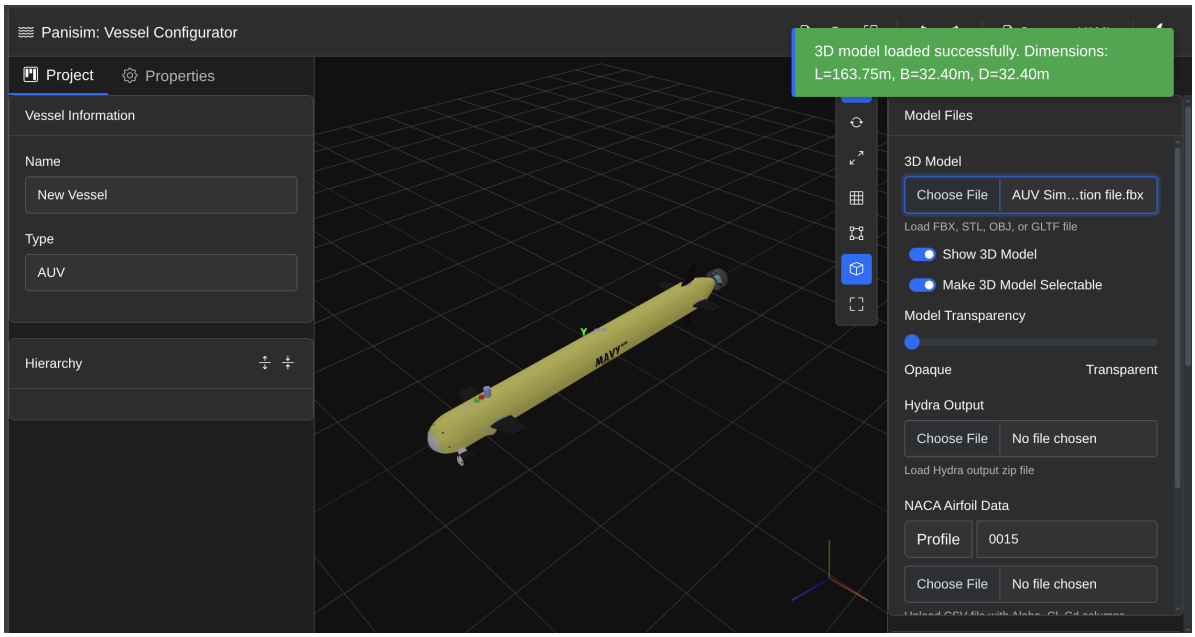


Figure 6: Loaded Vessel

> **Tip**
>
> The software automatically calculates the length, breadth and depth of the vessel from the FBX file via a bounding box. if possible, you should have your vessel in the configuration used in marine vessels, i.e X axis along the length of the vessel pointing forward, Y axis along the breadth pointing to the starboard and Z axis along the depth pointing downwards.

Step 2: Configure the Vessel

1. Select the component (Control Surfaces, Thrusters, or Sensors) you want to configure. It will be slightly highlighted on selection.

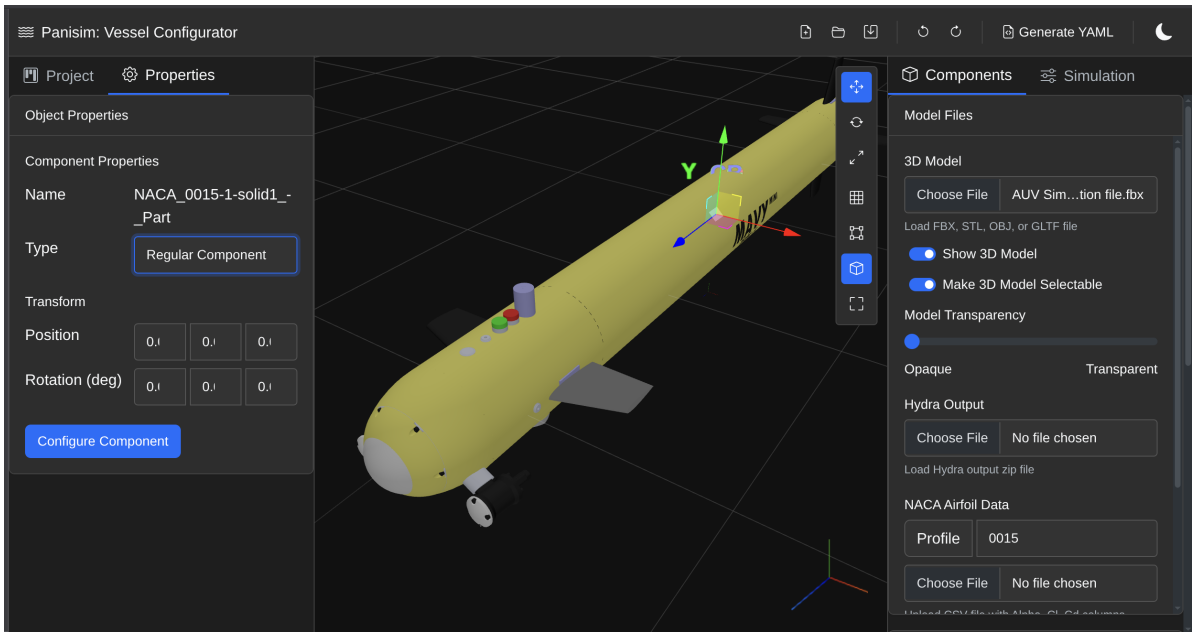


Figure 7: Selected Component

2. From the left sidebar, switch to the Properties tab and select the component type.
3. Configuration modal would appear on selecting the component type.

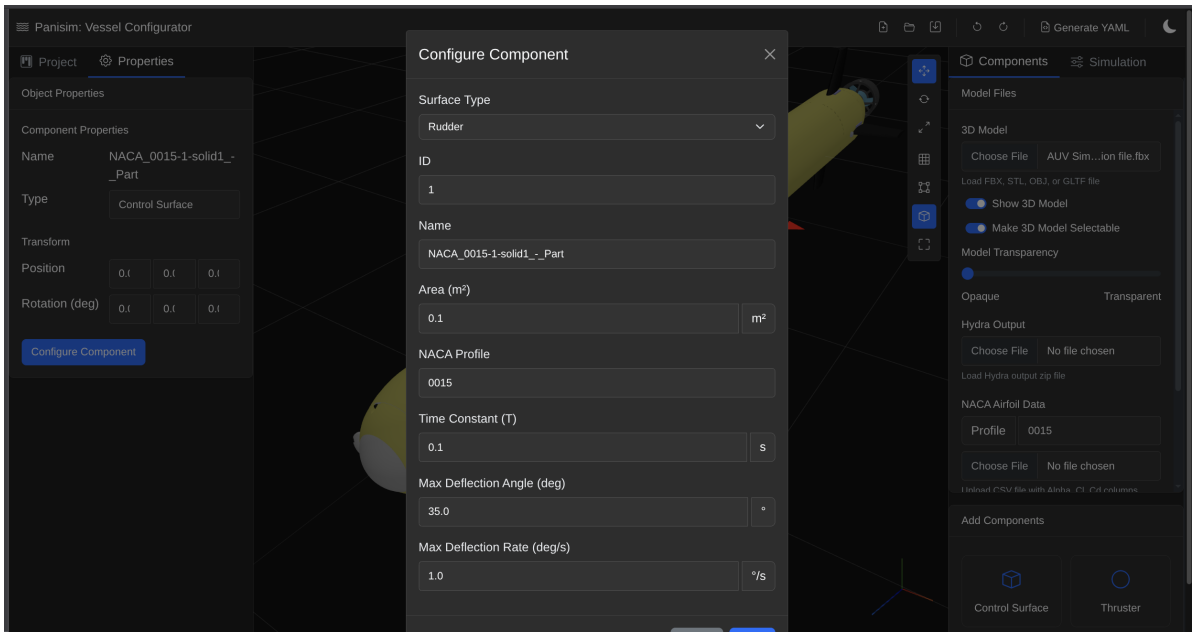


Figure 8: Component Configuration Modal

4. Enter the required parameters and click on the “Apply” button.
5. The component will then be configured. This can be confirmed if you see a coordinate system assigned to your vessel.

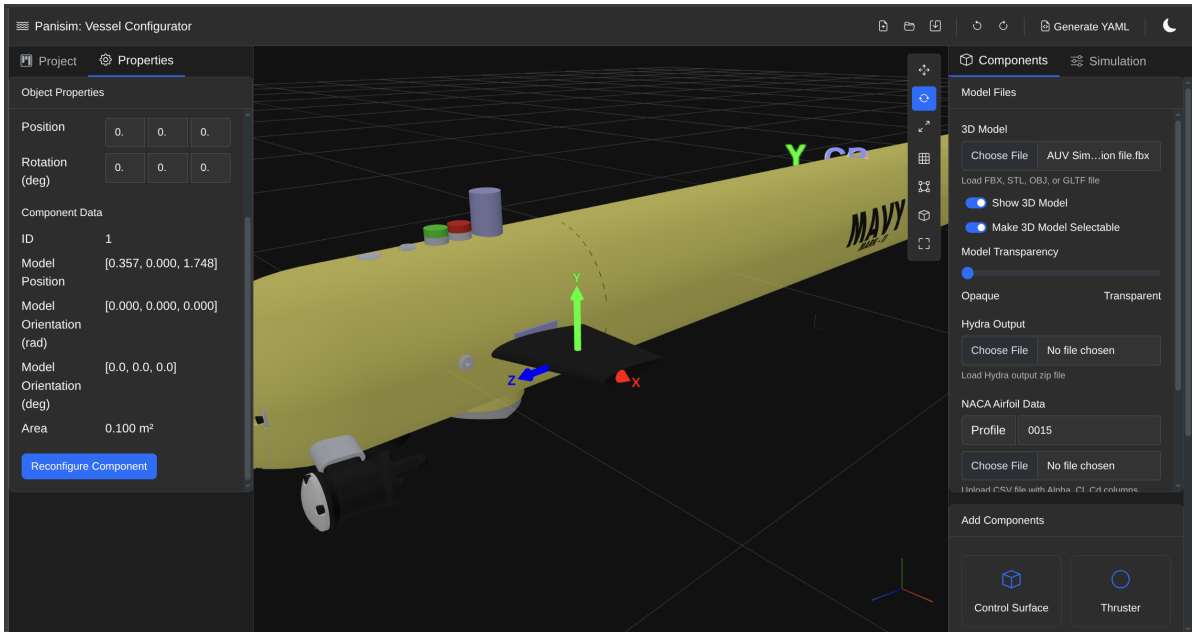


Figure 9: Component Configured

6. You should see the position and orientation in the properties tab.
7. You can select the coordinate and transform it in the 3D view based on where you want to place it.

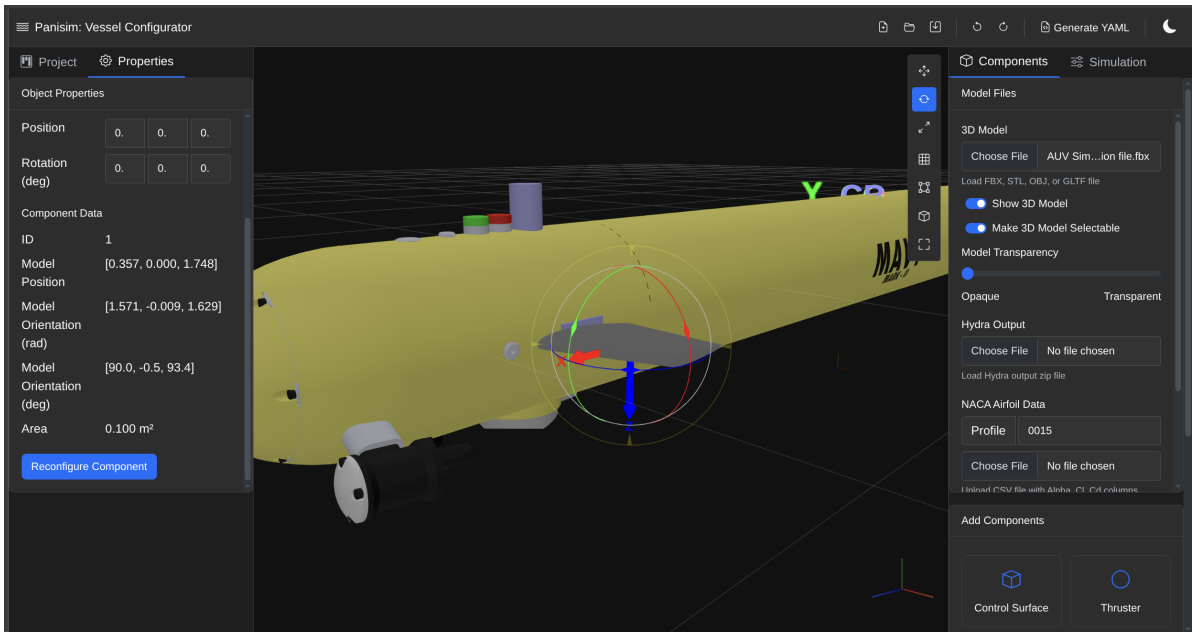


Figure 10: Component Configured

For sensors, the orientation have to be how you have mounted it on the vessel.

For control surfaces, the X axis needs to be along the chord, Y axis pointing along the span inwards to the vessel and Z axis completing the right hand coordinate system.

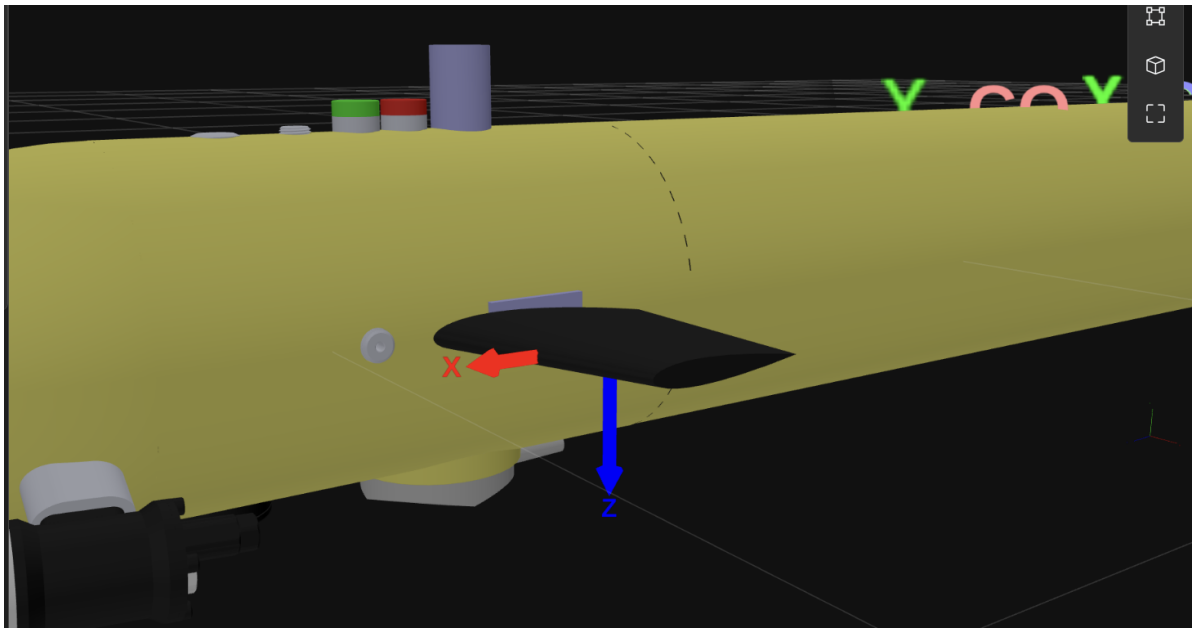


Figure 11: Control Surface

For thrusters, the X axis needs to be along the axis of the thruster, Y axis pointing towards the starboard and Z axis completing the right hand coordinate system.

Everything follows the NED (North, East, Down) coordinate system.

Step 3: Placing the vessel center points

1. Make the vessel transparent to see the inside. And untoggle the "Make 3D Model Selectable" option. This will allow you to select the centre points and transform them.

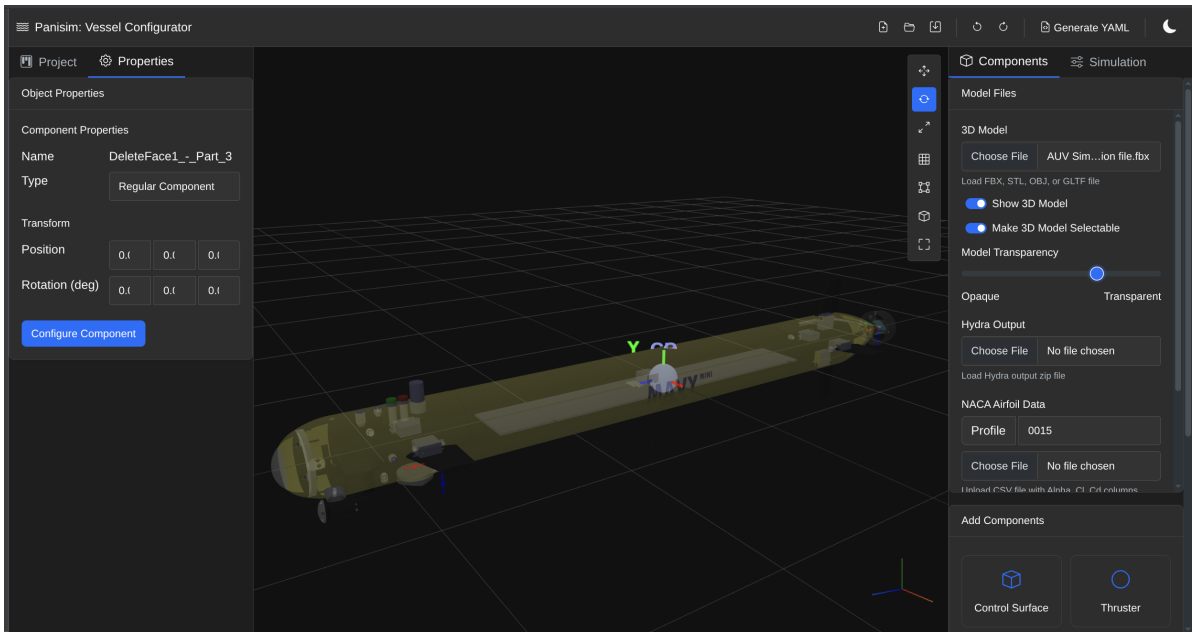


Figure 12: Transparent Vessel

2. You can select each of the center points and transform them to the desired location.
3. You can also manually enter the coordinates in the `Edit Geometry Parameters` in the right sidebar components tab.
4. The Body Centre [CO], is the point with respect to which all other coordinates are defined. And therefore it must be placed in the NED configuration.

Step 4: Edit Hydrodynamic Coefficients

1. Edit the hydrodynamic coefficients in the `Edit Hydrodynamic Coefficients` in the right sidebar components tab. Follow the instructions in the modal to enter the hydrodynamic coefficients.

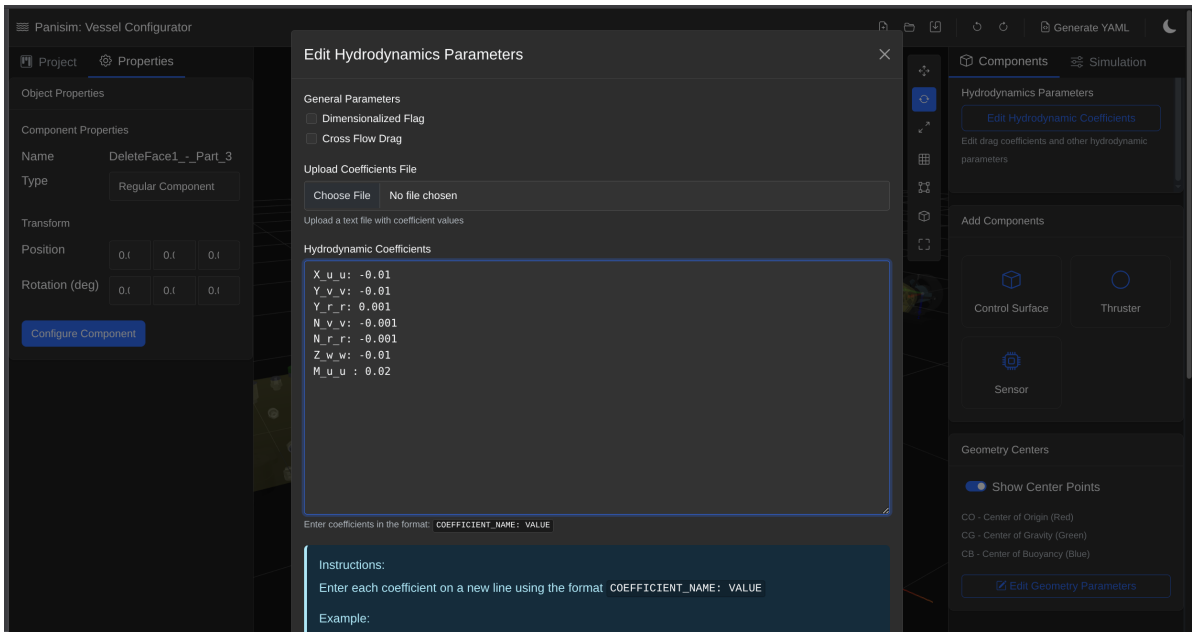


Figure 13: Hydrodynamic Coefficients

Step 5: Simulation settings

1. Edit all the parameters in the `Simulation` tab in the right sidebar.

> **Note**
>
> In the `Advanced Simualtion Parameters`, you can visually select the geofence boundaries. This also can be further extended to select waypoints in a future release.

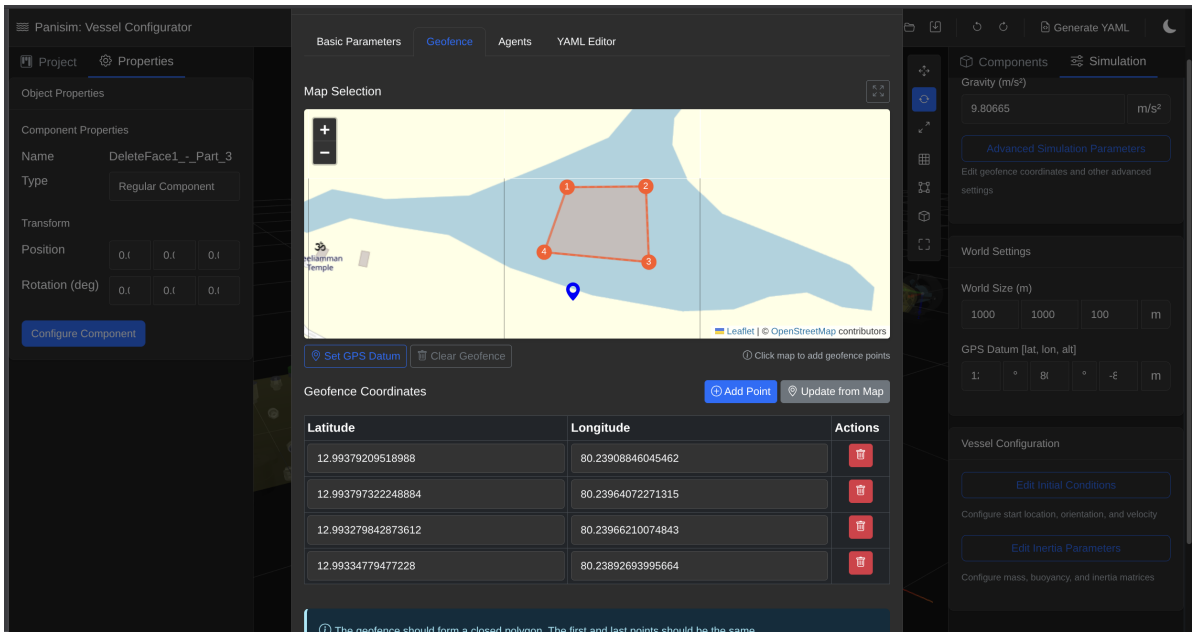


Figure 14: Geofence

Step 6: Upload the Hydra output file (optional, if you are not using cross-flow drag)

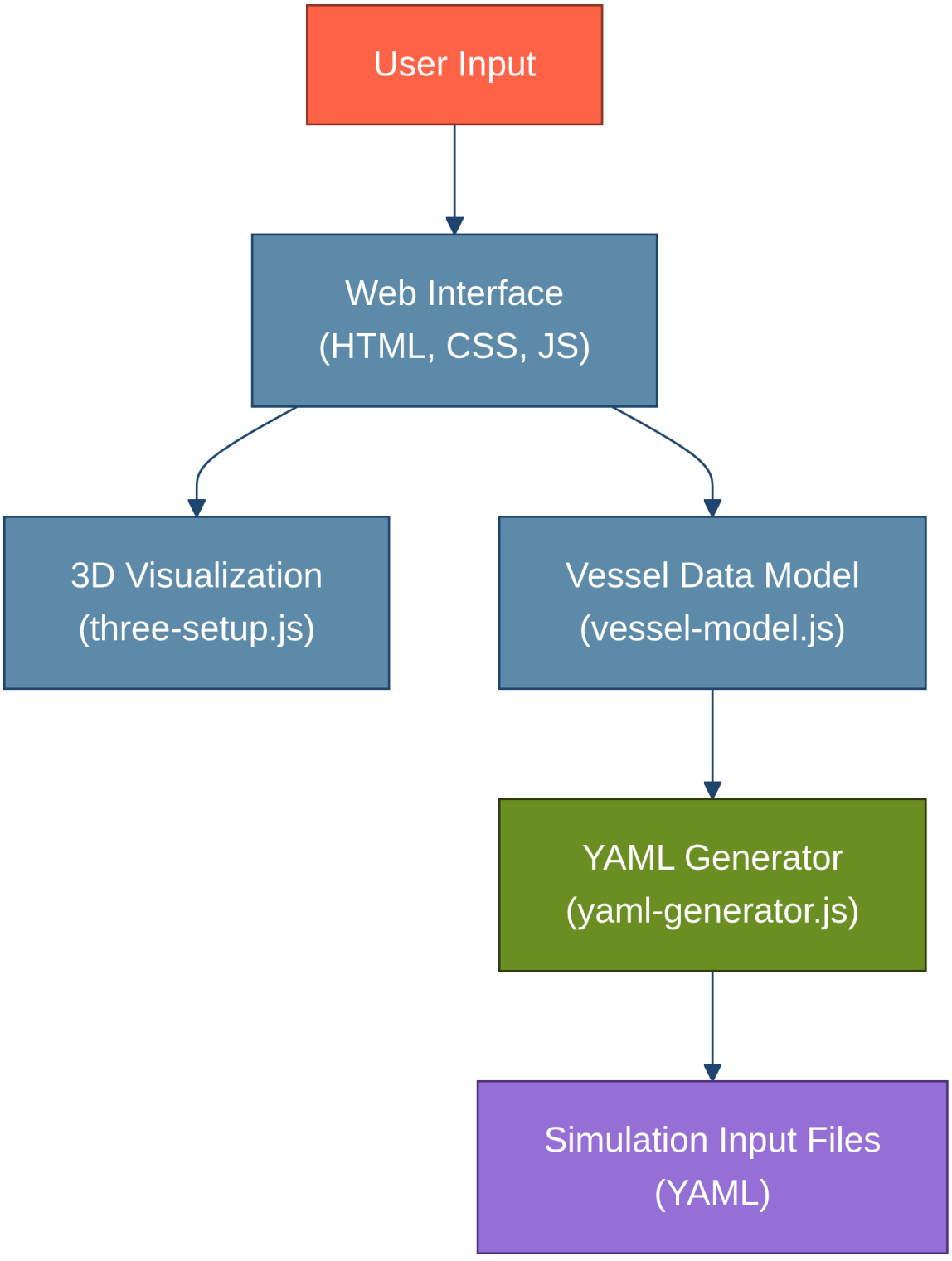
You can get your Hydra output file from the HydRA website. And upload the zip folder in the web gui. The generated input file will then contain your Hydra file with correct path.

Step 7: Generate the YAML files

1. Click on the "Generate YAML" button in the toolbar.
2. The YAML files will be generated and downloaded as a zip file.
3. Unzip the file and place it in the `inputs` directory.

And there you have it! You have successfully configured your vessel and generated the required input files for the Panisim simulator. Some configuration file may require manual edits. Feel free to play around with the Web GUI and extend its functionality.

Software Architecture



The Vessel Configurator provides a complete workflow for creating and configuring marine vessels with the following key features:

- **Parameter Configuration**: Intuitive interfaces for setting physical properties and simulation parameters, GPS waypoints.
- **Component Management**: Add, edit, and position thrusters, control surfaces, and sensors
- **YAML Generation**: Automatic generation of properly formatted configuration files for the simulator
- **Validation**: Built-in checks to ensure valid component ID configuration

Core Components

1. **User Interface (`index.html`, `styles.css`, `themes.css`)**
    - Implements a responsive, CAD-style interface with Bootstrap 5.3.2 framework integration
    - Features a component hierarchy with:
        - Primary toolbar containing application controls (new/load/save/undo/redo)
        - Split-pane workspace with resizable panels using Split.js
        - Left sidebar with tabbed project explorer and properties inspector
        - 3D viewport with transform controls and visualization options
        - Right sidebar with component addition and configuration panels
        - Modal dialogs for advanced configuration options and YAML preview
    - Supports context-sensitive property editing with dynamic form generation
    - Implements theme switching functionality with CSS variables for light/dark modes
    - Maintains responsive layout through Bootstrap grid system and custom flex containers
2. **3D Visualization Engine (`three-setup.js`)**
    - Implements a Three.js-based rendering system with WebGL acceleration
    - Configures high-performance renderer with antialiasing, physically correct lighting, and shadow mapping
    - Manages scene graph with hierarchical component structure and parent-child relationships
    - Features multiple coordinate systems (world, vessel-local, component-local)
    - Implements interactive controls with:
        - OrbitControls for camera navigation (pan, rotate, zoom)
        - TransformControls for direct manipulation (translate, rotate, scale)
        - Raycaster-based object selection with visual feedback
    - Provides real-time synchronization between 3D objects and data model properties
    - Implements custom visual feedback systems including:

  - Color-coded component highlighting for selection state
  - Local axes visualization for component orientation
  - Text sprite labeling for identification and measurements
  - Dynamic material updates for selection and hover states
- Optimizes rendering performance using:
  - Request animation frame with conditional rendering
  - Efficient mesh creation with geometry instancing
  - Adaptive resolution scaling based on device capabilities
  - Conditional shadow casting based on object importance

3. **Vessel Data Model (`vessel-model.js`)**
   - Implements a comprehensive object-oriented data structure with 1156+ lines of structured code
   - Maintains vessel configuration as a deeply nested JavaScript object with strong typing conventions
   - Organizes data in specialized subsystems:
     - Physical properties (dimensions, mass, centers)
     - Hydrodynamic coefficients (added mass, damping, restoring forces)
     - Component collections (control surfaces, thrusters, sensors)
     - Simulation parameters (time step, environment, boundary conditions)
   - Provides transaction-based modification methods with validation:
     - Addition, update, and removal operations for all component types
     - Auto-incrementing ID allocation for component tracking
     - Reference integrity management between components
     - Unit conversion and normalization for consistent data representation
   - Implements bidirectional mapping between model data and 3D objects using UUID tracking
   - Maintains persistence through JSON serialization/deserialization with state version control
   - Supports incremental updates through partial property modification methods

4. **YAML Generator (`yaml-generator.js`)**
   - Implements specialized transformation engine for converting vessel model to simulation-compatible YAML
   - Features robust numerical formatting with:
     - Configurable precision control (4-6 decimal places based on parameter type)
     - Unit annotation through comments
     - Scientific notation for small coefficient values
     - Array formatting with dimension-appropriate representation
   - Organizes output into standard simulation file hierarchy:
     - `geometry.yml` with vessel dimensions and center points

  - `hydrodynamics.yml` with coefficient grouping by motion direction (X/Y/Z/K/M/N)
  - `inertia.yml` with mass matrix and added mass coefficients
  - `control_surfaces.yml` with NACA profile properties and dynamics
  - `thrusters.yml` with propulsion characteristics and placement
  - `initial_conditions.yml` with startup position and velocity
  - `sensors.yml` with instrumentation configuration and noise models
- Generates Docker-compatible file paths for simulator integration (`/workspaces/mavlab/inputs/...`)
- Performs semantic validation with appropriate warning generation
- Uses JSZip library for creating multi-file archives with proper directory structure

5. **Utility Modules**

- `gdf-loader.js`: Implements parser for Geometric Data Files (GDF) with:
  - Triangulated mesh conversion
  - Hydrodynamic coefficient extraction
  - Center of buoyancy calculation from mesh geometry
- `controls.js`: Extends Three.js transform controls with:
  - Custom snapping behavior for precise positioning
  - Event handling for synchronized model updates
  - Specialized axis constraints for different component types
- External library integration:
  - Three.js (r128) for 3D visualization
  - JSZip (3.10.1) for configuration packaging
  - js-yaml (4.1.0) for YAML parsing/generation
  - Bootstrap (5.3.2) for UI components
  - Leaflet (1.9.4) for geofencing and waypoint mapping
  - FileSaver (2.0.5) for client-side file downloads

Simulation Visualization Dashboard

Overview

Panisim provides a web-based visualization dashboard to monitor vessel simulation in real-time. The dashboard offers a comprehensive interface for monitoring vessel state, visualizing trajectories, viewing sensor data, and interacting with the simulation.

The visualization system consists of two main components:

1. **2D Dashboard** - A data-rich interface showing vessel state, charts, and control inputs
2. **3D Visualization** - A three-dimensional view of the vessel in its environment with position and orientation

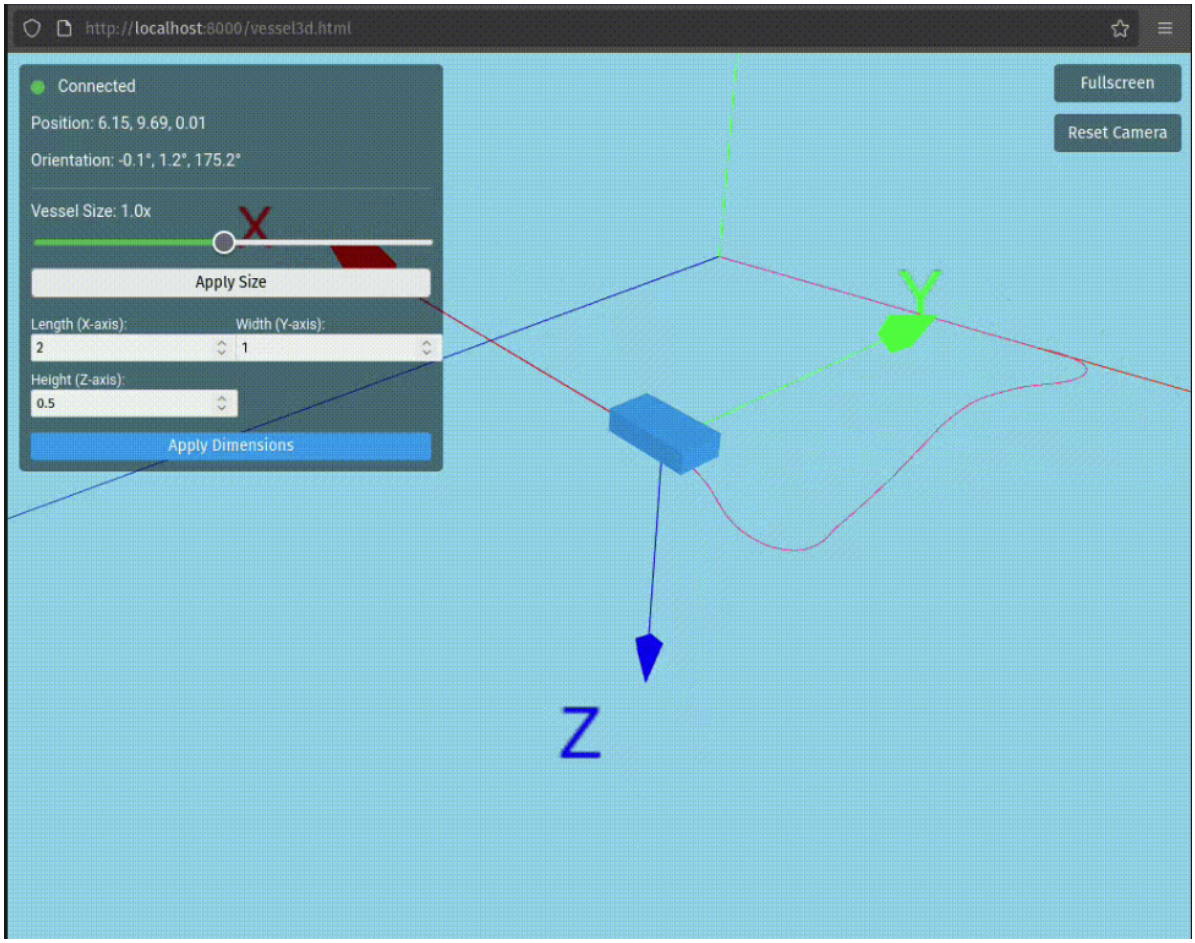
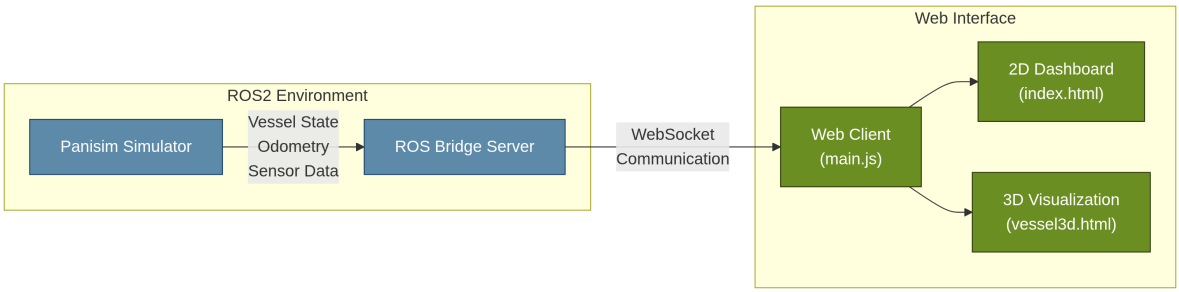


Figure 15: Way-point trackingSimulation Visualization



Starting the Visualization

The web interface is automatically started when launching the Panisim simulator. It uses:

1. A lightweight HTTP server to host the web files
2. The ROS Bridge server for WebSocket communication with ROS2

The visualization launches when running the simulation:

```
ros2 launch gnc gnc_launch.py
```

Once started, you can access the dashboard by opening your web browser:

- **Main Dashboard**: `http://localhost:8000/`
- **3D Visualization**: `http://localhost:8000/vessel3d.html`

Main Dashboard

The main dashboard (index.html) provides a comprehensive overview of the vessel's state, sensor data, and control inputs.

Key Features

1. **Vessel Selection**: Switch between different vessels in the simulation
2. **Connection Status**: Real-time display of connectivity to the ROS2 environment
3. **Vessel Information**: Display of key vessel parameters (position, velocity, orientation)
4. **Data Visualizations**: Real-time charts for:
    - Vessel path and trajectory
    - Velocities (linear and angular)
    - Orientation (roll, pitch, yaw)
    - Control surface positions
    - Thruster states

Dashboard Layout

The dashboard is organized in a responsive grid layout with cards for different data categories:

Section	Description
Vessel Info	Basic vessel parameters including position, velocities, and orientation
Path Visualization	2D trajectory visualization showing vessel movement in X-Y plane
Vessel Velocities	Charts for surge (u), sway (v), and heave (w) velocities

Section	Description
Vessel Angular Rates	Charts for roll rate (p), pitch rate (q), and yaw rate (r)
Orientation	Charts for roll (), pitch (), and yaw () angles
Control Surfaces	Visual representation of control surface positions
Thrusters	Visual representation of thruster RPMs

Real-time Data Updates

The dashboard updates in real-time with configurable update rates. Data collection begins automatically when connecting to a vessel and includes:

- Subscribe to `/[vessel_name]/odometry_sim` for position and velocity data
- Subscribe to `/[vessel_name]/vessel_state` for complete vessel state including actuators
- Optional subscription to additional sensor topics

3D Visualization

The 3D visualization (vessel3d.html) provides an interactive three-dimensional view of the vessel in its environment.

Key Features

1. **Real-time Movement**: The vessel model moves and rotates according to simulation data
2. **Trajectory Path**: A colored trail showing the vessel's path through the environment
3. **Customizable View**: Camera controls for rotating, panning, and zooming
4. **Vessel Dimensions**: Adjustable vessel size and proportions to match different vessel types
5. **Full-screen Mode**: Expand the visualization to utilize the entire screen

Controls

The 3D view offers several interactive controls:

- **Orbit**: Click and drag to rotate the camera around the vessel
- **Pan**: Right-click and drag (or middle-click and drag) to move the camera
- **Zoom**: Scroll wheel to zoom in and out

- **Reset Camera**: Button to restore the default camera position
- **Fullscreen**: Button to toggle fullscreen mode

Vessel Model Customization

You can adjust the vessel's appearance through the control panel:

- **Vessel Size**: Overall scaling of the vessel model
- **Dimensions**: Individual control over length, width, and height proportions
- **Apply Changes**: Button to update the model with the new dimensions

Integration with ROS2

The visualization dashboard communicates with ROS2 using the ROS Bridge Server, which provides WebSocket connectivity between the ROS2 environment and web clients.

Communication Flow

1. **ROS2 Topics to WebSockets**: The ROS Bridge Server converts ROS2 topic messages to WebSocket messages
2. **Web Client Processing**: JavaScript code processes the WebSocket messages and updates the visualizations
3. **Bidirectional Communication**: The system supports both subscribing to and publishing ROS2 topics

Required ROS2 Components

To enable the visualization, the launch file includes:

```
# HTTP server for web files
ExecuteProcess(
    cmd=['python3', '-m', 'http.server', '8000', '--directory', web_build_dir],
    name='http_server',
    output='screen'
),

# ROS Bridge server for WebSocket communication
IncludeLaunchDescription(
    AnyLaunchDescriptionSource(rosbridge_launch)
),
```

Topic Subscriptions

The visualization automatically discovers vessel topics by querying available ROS2 topics. It looks for the following patterns:

- `/<vessel_name>/odometry_sim`: For vessel position and velocity
- `/<vessel_name>/vessel_state`: For complete vessel state including actuators

Implementation Details

The visualization dashboard is built using modern web technologies:

- **Chart.js**: For responsive, interactive data visualization
- **three.js**: For 3D rendering of the vessel
- **roslibjs**: For ROS2 communication via WebSockets
- **Responsive Design**: Automatically adapts to different screen sizes

Key JavaScript Components

The system is organized into the following components:

1. **VesselManager**: Handles vessel discovery, selection, and configuration
2. **VesselVisualizer**: Manages data collection, processing, and visualization
3. **3D Rendering**: Handles the three-dimensional visualization of vessels

Vessel State Representation

The dashboard visualizes the complete vessel state model:

- **Position**: x, y, z (in meters)
- **Linear Velocity**: u, v, w (in m/s)
- **Angular Velocity**: p, q, r (in rad/s)
- **Orientation**: , , (in degrees)
- **Actuators**: Control surface angles (in degrees) and thruster RPMs

Customization Options

The visualization can be customized in several ways:

1. **Update Rates**: Chart refresh rates can be adjusted
2. **Data Window**: The amount of historical data displayed in charts
3. **Vessel Configuration**: The number of control surfaces and thrusters displayed
4. **3D Model**: Vessel dimensions and appearance
5. **Chart Scaling**: Auto-rescaling and manual zoom/pan on charts

Troubleshooting

Connection Issues

If the dashboard shows "Disconnected" status:

1. Ensure the ROS Bridge server is running
2. Check that WebSocket port 9090 is accessible
3. Verify the simulator is publishing required topics

Missing Vessel Data

If vessel data isn't displaying:

1. Select a vessel from the dropdown menu
2. Check ROS2 topics are being published:
   ```
   ros2 topic list | grep vessel
   ```
3. Verify message publication with:
   ```
   ros2 topic echo /<vessel_name>/odometry_sim
   ```

Browser Compatibility

The visualization works best with modern browsers:

- Google Chrome
- Mozilla Firefox

- Microsoft Edge
- Brave Browser

Conclusion

The Panisim visualization dashboard provides a comprehensive interface for monitoring and interacting with simulated vessels. With its real-time updates, interactive charts, and 3D visualization, it offers a powerful tool for understanding vessel behavior and validating custom control or guidance algorithms.

Example - Creating a ROS2 Waypoint Tracking Controller

Overview

This example demonstrates how to create a custom ROS2 package to interact with the Panisim simulator. We'll build a Guidance, Navigation, and Control (GNC) package that implements a waypoint tracking PID controller for marine vessels. This example covers:

1. Creating a ROS2 package structure
2. Implementing core controller logic
3. Setting up ROS2 nodes to interface with the simulator
4. Building and launching the complete system

> **Tip**
>
> All the below codes and folders are already implemented in the Repository. This example is only for the reference and to show how you can configure your own ROS2 package to interact with the Panisim simulator.

Creating the ROS2 Package

1. Package Structure

First, let's create a new ROS2 package called `gnc` in the `ros2_ws/src` directory:

```
cd ros2_ws/src
ros2 pkg create --build-type ament_python gnc
--dependencies rclpy nav_msgs geometry_msgs interfaces
```

This creates a basic package structure:

```
gnc/
 gnc/
    __init__.py
 package.xml
 setup.py
 setup.cfg
```

2. Update Package Dependencies

Edit `package.xml` to include all required dependencies:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"schematypens="http://ww
<package format="3">
  <name>gnc</name>
  <version>0.0.1</version>
  <description>Guidance, Navigation and Control package for Panisim</description>
  <maintainer email="example@example.com">user</maintainer>
  <license>Apache License 2.0</license>

  <depend>rclpy</depend>
  <depend>nav_msgs</depend>
  <depend>geometry_msgs</depend>
  <depend>std_msgs</depend>
  <depend>interfaces</depend>
  <depend>mav_simulator</depend>

  <exec_depend>ros2launch</exec_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

3. Configure Setup Scripts

Update `setup.py` to include our entry points and specify dependencies:

```
from setuptools import setup
import os
from glob import glob
```

```
package_name = 'gnc'

setup(
    name=package_name,
    version='0.0.1',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob('launch/*.py')),
        (os.path.join('share', package_name, 'web'), glob('web/**/*', recursive=True)),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='example@example.com',
    description='Guidance, Navigation and Control package for Panisim',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'gc = gnc.guidance_control:main',
            'nav = gnc.navigation:main',
        ],
    },
)
```

4. Create Directory Structure

Set up the complete directory structure:

```
mkdir -p gnc/launch
mkdir -p gnc/web
```

Implementing the Controller Logic

1. Control Module

First, create `module_control.py` in the gnc package to implement the PID control logic:

```
# ros2_ws/src/gnc/gnc/module_control.py
import numpy as np

"""
Rudder control module for vessel simulation.

This module provides rudder control strategies for vessel steering with PID controllers.
"""

def ssa(ang, deg=False):
    """
    Smallest signed angle that lies between -pi and pi.
    If deg is True, the angle is assumed to be in degrees and is converted to radians.
    If deg is False, the angle is assumed to be in radians.
    """
    if deg:
        ang = (ang + 180) % (360.0) - 180.0
    else:
        ang = (ang + np.pi) % (2 * np.pi) - np.pi
    return ang

def clip(val, min_val, max_val):
    """
    Clip a value to a range.
    """
    return max(min_val, min(val, max_val))

def pid_control(t, state, waypoints, waypoint_idx, ye_int=0.0):
    """
    Implement a PID control strategy to follow the waypoints.

    Args:
        t (float): Current simulation time [s]
        state (ndarray): Current vessel state vector
        waypoints (ndarray): Waypoints array
        waypoint_idx (int): Current waypoint index
```

```
        ye_int (float): Integral term of cross-track error

    Returns:
        float: Commanded rudder angle in radians
        float: Cross-track error
        int: Next waypoint index
    """

    # Check if we have reached the last waypoint
    if waypoint_idx == len(waypoints):
        return 0.0, 0.0, waypoint_idx

    # Current state
    u, v, r = state[3], state[4], state[5]
    x, y, psi = state[6], state[7], state[11]

    # Current and previous waypoints
    wp_xn, wp_yn, _ = waypoints[waypoint_idx]
    wp_xn1, wp_yn1, _ = waypoints[waypoint_idx - 1]

    # Calculate path line equation: ax + by + c = 0
    a = -(wp_yn1 - wp_yn)
    b = (wp_xn1 - wp_xn)
    c = -wp_yn1*b-a*wp_xn1

    # Path direction unit vector
    wp_unit_vec = np.array([wp_xn - wp_xn1, wp_yn - wp_yn1, 0.0])
    wp_unit_vec = wp_unit_vec / np.linalg.norm(wp_unit_vec)

    # Calculate cross-track error
    ye = -(a*x + b*y + c)/np.sqrt(a**2 + b**2)

    # Path direction angle
    pi_p = np.angle(wp_unit_vec[0] + 1j * wp_unit_vec[1])

    # Outer loop PID gains
    Kpo = 0.6  # Proportional gain
    Kio = 0.05 # Integral gain

    # Desired heading (outer loop control)
    psid = pi_p + Kpo*(-ye) + Kio*(-ye_int)
```

```
    # Inner loop PID gains
    Kpi = 0.7  # Proportional gain
    Kdi = 0.5  # Derivative gain

    # Rudder command (inner loop control)
    delta_c = Kpi*ssa(psid - psi) + Kdi*(0 - r)

    # Clip the rudder angle
    delta_c = clip(delta_c, -35*np.pi/180, 35*np.pi/180)

    # Distance to waypoint
    wp_dist = np.linalg.norm(np.array([x - wp_xn, y - wp_yn, 0.0]))
    if wp_dist < 0.5:  # 0.5m threshold
        waypoint_idx += 1

    # Return negative delta_c (due to sign convention of rudder angle)
    return -delta_c, ye, waypoint_idx
```

2. Guidance Control Node Class

Next, create `class_guidance_control.py` to implement the ROS2 node class:

```
# ros2_ws/src/gnc/gnc/class_guidance_control.py
from rclpy.node import Node
from nav_msgs.msg import Odometry
from interfaces.msg import Actuator
from geometry_msgs.msg import PoseArray, Pose
from mav_simulator.module_kinematics import quat_to_eul
import gnc.module_control as con
import numpy as np

class GuidanceControl(Node):
    def __init__(self, vessel):
        super().__init__('guidance_controller')

        self.vessel = vessel

        # Configure topics based on vessel name and ID
        self.odom_topic = f'{self.vessel.vessel_name}_
        {self.vessel.vessel_id:02d}/odometry_sim'
        self.actuator_topic = f'{self.vessel.vessel_name}_
```

```
        {self.vessel.vessel_id:02d}/actuator_cmd'
        self.waypoints_topic = f'{self.vessel.vessel_name}_
        {self.vessel.vessel_id:02d}/waypoints'

        # Create subscriber for odometry
        self.odom_sub = self.create_subscription(
            Odometry,
            self.odom_topic,
            self.odom_callback,
            10)

        # Create publisher for rudder command
        self.actuator_pub = self.create_publisher(
            Actuator,
            self.actuator_topic,
            10)

        # Create publisher for waypoints
        self.waypoints_pub = self.create_publisher(
            PoseArray,
            self.waypoints_topic,
            10)

        # Set up timer to publish waypoints
        self.timer = self.create_timer(1.0, self.publish_waypoints)

        # Read waypoints from vessel configuration
        self.waypoints = np.array(self.vessel.vessel_config['guidance']['waypoints'])
        self.waypoints_type = self.vessel.vessel_config['guidance']['waypoints_type']
        self.waypoint_idx = 1

        self.waypoints_data = {
            'waypoints': self.waypoints,
            'waypoints_type': self.waypoints_type
        }

        # Initialize time
        self.start_time = self.get_clock().now()

        # Initialize integrator
        self.ye_int = 0.0
        self.te_int = 0.0
```

```
        # Select control mode
        self.control_mode = con.pid_control

        self.get_logger().info('Guidance Controller initialized')

    def publish_waypoints(self):
        """Publish waypoints for visualization"""
        if not self.waypoints_data:
            return

        pose_array = PoseArray()
        pose_array.header.stamp = self.get_clock().now().to_msg()
        pose_array.header.frame_id = "map"

        for waypoint in self.waypoints_data.get('waypoints', []):
            pose = Pose()
            pose.position.x = float(waypoint[0])
            pose.position.y = float(waypoint[1])
            pose.position.z = float(waypoint[2])
            pose_array.poses.append(pose)

        self.waypoints_pub.publish(pose_array)

    def odom_callback(self, msg):
        """Process odometry data and calculate control commands"""
        # Get current time in seconds
        current_time = self.get_clock().now()
        t = (current_time - self.start_time).nanoseconds / 1e9

        # Extract orientation quaternion
        quat = np.array([
            msg.pose.pose.orientation.w,
            msg.pose.pose.orientation.x,
            msg.pose.pose.orientation.y,
            msg.pose.pose.orientation.z
        ])

        # Convert quaternion to Euler angles
        eul = quat_to_eul(quat, order='ZYX')

        # Create state vector from odometry
        state = np.array([
```

```
            msg.twist.twist.linear.x,
            msg.twist.twist.linear.y,
            msg.twist.twist.linear.z,
            msg.twist.twist.angular.x,
            msg.twist.twist.angular.y,
            msg.twist.twist.angular.z,
            msg.pose.pose.position.x,
            msg.pose.pose.position.y,
            msg.pose.pose.position.z,
            eul[0],
            eul[1],
            eul[2],
            0.0 # rudder angle
        ])

        # Calculate control command
        rudder_cmd, ye, wp_indx = self.control_mode(t, state, self.waypoints, self.waypoint_i

        # Handle waypoint transitions
        if wp_indx != self.waypoint_idx:
            self.get_logger().info(f'Waypoint {
            self.waypoint_idx} reached')
            self.waypoint_idx = wp_indx
            self.ye_int = 0.0  # Reset integral term

            # Loop back to first waypoint when all are visited
            if self.waypoint_idx >= len(self.waypoints):
                self.get_logger().info('***All waypoints reached***')
                self.waypoint_idx = 1

        # Calculate time difference for integration
        dt = t - self.te_int
        if dt > 0:  # Avoid division by zero or negative time
            # Update the integral term
            self.ye_int += ye * dt
            self.te_int = t

        # Publish actuator command
        cmd_msg = Actuator()
        cmd_msg.actuator_values = [float(rudder_cmd * 180 / np.pi)]  # Convert to degrees
        cmd_msg.actuator_names = ['cs_1']  # Control surface 1
        cmd_msg.covariance = [0.0]
```

```
        self.actuator_pub.publish(cmd_msg)
```

3. Main Entry Point

Create `guidance_control.py` to serve as the entry point for the ROS2 node:

```
# ros2_ws/src/gnc/gnc/guidance_control.py
import rclpy
from gnc.class_guidance_control import GuidanceControl
from mav_simulator.class_world import World
from rclpy.executors import MultiThreadedExecutor

def main():
    # Initialize ROS2
    rclpy.init()

    # Load world and vessels from configuration
    world = World('/workspaces/mavlab/inputs/simulation_input.yml')
    vessels = world.vessels

    # Create multi-threaded executor
    executor = MultiThreadedExecutor()

    # Initialize guidance controllers for all vessels
    gcs = []
    for vessel in vessels:
        gcs.append(GuidanceControl(vessel))
        executor.add_node(gcs[-1])

    try:
        # Start spinning the nodes
        executor.spin()

    finally:
        # Clean up on shutdown
        executor.shutdown()

        for gc in gcs:
            gc.destroy_node()

        rclpy.shutdown()
```

```
if __name__ == '__main__':
    main()
```

4. Create a Navigation Node Placeholder

Add a simple placeholder for the navigation node to satisfy the launch file requirements:

```
# ros2_ws/src/gnc/gnc/navigation.py
import rclpy
from rclpy.node import Node

class Navigation(Node):
    def __init__(self):
        super().__init__('navigation')
        self.get_logger().info('Navigation node started')

def main():
    rclpy.init()
    node = Navigation()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Creating the Launch File

Create a launch file to start all components of the system:

```
# ros2_ws/src/gnc/launch/gnc_launch.py
import os
from launch import LaunchDescription
from launch_ros.actions import Node
from launch import LaunchDescription
from launch.actions import ExecuteProcess, IncludeLaunchDescription
from launch.launch_description_sources import AnyLaunchDescriptionSource
from launch_ros.substitutions import FindPackageShare
from launch.substitutions import PathJoinSubstitution
from launch_ros.actions import Node
```

```
from mav_simulator.class_world import World
from launch.substitutions import LaunchConfiguration
from launch.actions import GroupAction
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import PushRosNamespace

def generate_launch_description():

    rosbridge_launch = PathJoinSubstitution([
        FindPackageShare('rosbridge_server'),
        'launch',
        'rosbridge_websocket_launch.xml'
    ])

    # Path to the web directory in gnc package
    web_dir = PathJoinSubstitution([
        FindPackageShare('gnc'),
        'web'
    ])

    web_build_dir = '/workspaces/mavlab/ros2_ws/src/mav_simulator/web'

    return LaunchDescription([
        Node(
            package='mav_simulator',  # Replace with your package name
            executable='simulate',  # Replace with your script name
            name='mavsim',
            output='screen'
        ),
        Node(
            package='gnc',
            executable='nav',
            name='nav',
            output='screen'
        ),
        Node(
            package='gnc',
            executable='gc',
            name='gc',
            output='screen'
        ),
        # Start the HTTP server to serve the web interface
```

```
        ExecuteProcess(
            cmd=['python3', '-m', 'http.server', '8000', '--directory', web_build_dir],
            name='http_server',
            output='screen'
        ),
        # Start the rosbridge server
        IncludeLaunchDescription(
            AnyLaunchDescriptionSource(rosbridge_launch)
        )
    ])
```

Building and Launching the Package

With all the files in place, you can build and launch your custom ROS2 package:

1. Build the Package

```
cd /workspaces/mavlab/ros2_ws
colcon build --packages-select gnc
source install/setup.bash
```

2. Launch the System

```
ros2 launch gnc gnc_launch.py
```

This will start: 1. The Panisim simulator 2. The navigation node 3. The guidance control node with waypoint tracking

Configuring Waypoints

Waypoints for vessel navigation are defined in the `guidance.yaml` input file:

```yaml
waypoints_type: XYZ
waypoints:
- [0.0, 0.0, 0.0]     # Starting point
- [10.0, 0.0, 0.0]    # First waypoint
- [10.0, 10.0, 0.0]   # Second waypoint
- [0.0, 10.0, 0.0]    # Third waypoint
- [0.0, 0.0, 0.0]     # Return to start
```

Monitoring System Performance

Once the system is running, you can monitor various aspects of the waypoint tracking:

```bash
# View odometry data
ros2 topic echo /<vessel_name>_<id>/odometry_sim

# View actuator commands
ros2 topic echo /<vessel_name>_<id>/actuator_cmd

# View waypoints
ros2 topic echo /<vessel_name>_<id>/waypoints
```

You can also visualize the vessel's movement in the web interface at `http://localhost:8000`.

Running the simulator without ROS2

Overview

Panisim provides a standalone mode that allows you to run simulations without requiring the ROS2 middleware. This lightweight mode is ideal for quick testing, development, and scenarios where the full ROS2 ecosystem is not needed.

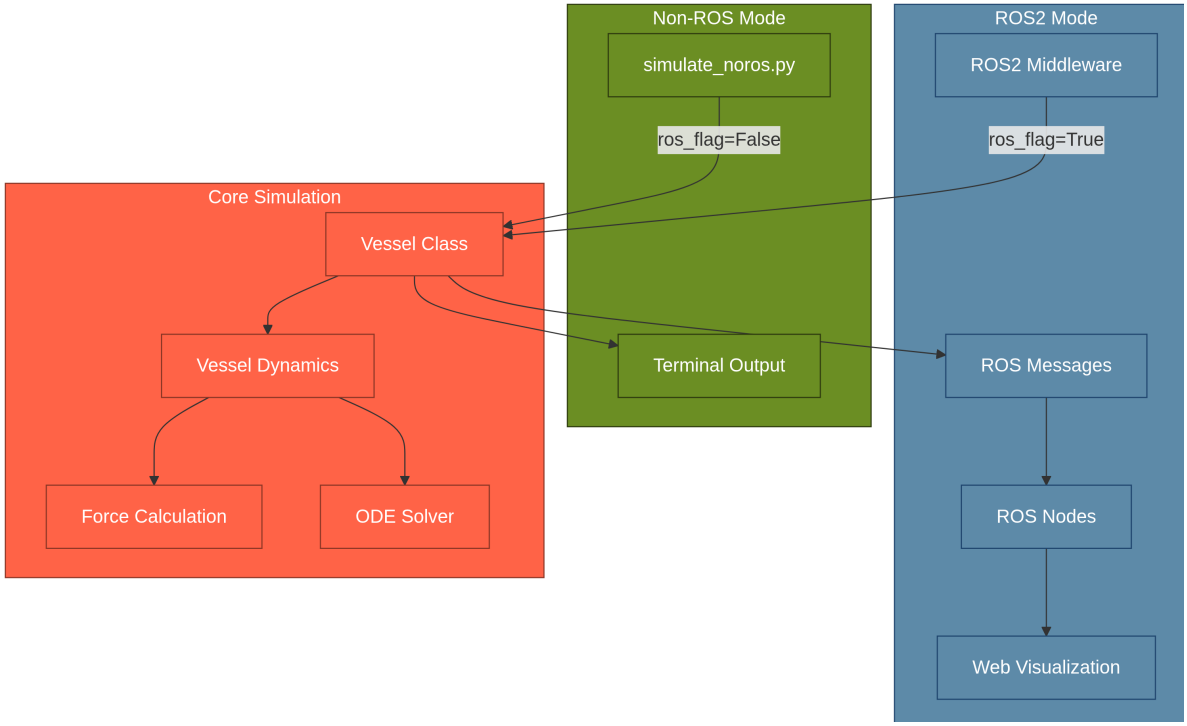
The non-ROS mode:

- Eliminates ROS2 dependencies and overhead
- Provides terminal-based feedback
- Supports the full vessel dynamics model
- Allows custom controller implementation
- Simplifies debugging of core simulation components

Implementation Details

The non-ROS mode is implemented through two key components:

1. `simulate_noros.py`: A standalone script that initializes and runs the simulation
2. `class_vessel.py`: The core vessel class with a `ros_flag` parameter that controls ROS-specific behavior



The `ros_flag` Parameter

The `ros_flag` parameter in the `Vessel` class constructor determines whether the vessel should use ROS2-specific functionality:

```
def __init__(self, vessel_params: Dict, hydrodynamic_data: Dict, vessel_id:
int, ros_flag: bool = True):
    """Initialize vessel with parameters and hydrodynamic data.

    Args:
        vessel_params: Dictionary containing vessel parameters
        hydrodynamic_data: Dictionary containing hydrodynamic coefficients
        vessel_id: Unique identifier for the vessel
        ros_flag: Flag indicating whether to use ROS2 functionality
    """
    self.ros_flag = ros_flag
    # ... rest of initialization
```

When `ros_flag` is set to `False`:

1. No ROS2 node is created for the vessel

2. No publishers or subscribers are established
3. Command inputs must be set directly in code or through custom control logic
4. Simulation data is displayed in the terminal instead of through ROS2 topics

Running the Non-ROS Simulator

To run Panisim without ROS2:

1. Navigate to the makara directory
2. Run the `simulate_noros.py` script:

```
cd /workspaces/mavlab/ros2_ws/src/mav_simulator/mav_simulator
python3 simulate_noros.py
```

The simulation will use the same configuration files as the ROS2 version, loading vessel parameters, hydrodynamics, and initial conditions from the input files.

Simulation Output

The non-ROS simulator provides detailed terminal output showing:

- Simulation time
- Vessel position and orientation
- Linear and angular velocities
- Forces acting on the vessel, including:
  - Hydrodynamic forces
  - Gravitational forces
  - Control surface forces
  - Thruster forces
- Control surface angles
- Thruster RPMs

Example output:

```
Starting simulation...
Time [s] | Position [x, y, z] | Velocity [u, v, w]
------------------------------------------------------------

Time:   0.00
Position [x, y, z]: [  0.00,   0.00,   0.00]
```

```
Velocity [u, v, w]: [  0.50,   0.00,   0.00]
Orientation [phi, theta, psi] (deg): [  0.00,   0.00,   0.00]
Hydrodynamic forces: [ -0.40,   0.00,   0.00,   0.00,   0.00,   0.00]
Gravitational forces: [  0.00,   0.00,   0.00,   0.00,   0.00,   0.00]
Control surface forces: [  0.00,   0.00,   0.00,   0.00,   0.00,   0.00]
Thruster forces: [  0.00,   0.00,   0.00,   0.00,   0.00,   0.00]
Control surface angles [deg]: []
Thruster RPMs: []

Time:   0.01
Position [x, y, z]: [  0.01,   0.00,   0.00]
Velocity [u, v, w]: [  0.50,   0.00,   0.00]
...
```

Implementing Custom Controllers

One of the key advantages of the non-ROS mode is the ability to easily implement custom controllers directly in the code. The `vessel_ode` method in `class_vessel.py` contains a section specifically for non-ROS mode:

```python
if not self.ros_flag:
    ## TODO: Implement your controller logic here to get the actuator commands

    ## example
    # if self.control_surface_control_type == 'fixed_rudder':
    #     self.delta_c = con.fixed_rudder(t, state, n_control_surfaces, 10.0)    #Enter rudde
    # elif self.control_surface_control_type == 'switching_rudder':
    #     self.delta_c = con.switching_rudder(t, state, n_control_surfaces)
    # else:
    #     raise ValueError(f"Invalid control surface control type:    {self.control_surface_

    # # Get thruster commands
    # if self.thruster_control_type == 'fixed_rpm':
    #     self.n_c = con.fixed_thrust(t, state, n_thrusters,1000.0) #Enter RPM here
    # else:
    #     raise ValueError(f"Invalid thruster control type: {self.thruster_control_type}")
    pass
```

To implement your own controller:

1. Create a controller module (e.g., `my_controller.py`) with your control algorithms

2. Import your controller module in `simulate_noros.py`
3. Modify the `vessel_ode` method to use your controller:

```
# In vessel_ode method of class_vessel.py
if not self.ros_flag:
    # Custom controller implementation
    if hasattr(self, 'control_surface_control_type'):
        if self.control_surface_control_type == 'my_controller':
            self.delta_c = my_controller.calculate_rudder(t, state, self.n_control_surfaces)
```

Example: Adding a Simple PID Controller

Here's an example of how to add a simple PID heading controller:

1. Create a controller module:

```
# my_controller.py
import numpy as np

def pid_heading_controller(t, state, n_surfaces, desired_heading=0.0):
    """Simple PID heading controller.

    Args:
        t: Current time
        state: Vessel state vector
        n_surfaces: Number of control surfaces
        desired_heading: Target heading in radians

    Returns:
        numpy.ndarray: Control surface angles in radians
    """
    # Extract current heading
    current_heading = state[11]  # psi is at index 11

    # Calculate heading error
    heading_error = desired_heading - current_heading

    # Normalize to -pi to pi
    heading_error = (heading_error + np.pi) % (2 * np.pi) - np.pi

    # PID gains
    Kp = 0.5
```

```
    Ki = 0.01
    Kd = 0.1

    # Calculate PID terms (simplified - would need integral and derivative state)
    p_term = Kp * heading_error

    # Calculate rudder command (assume first surface is main rudder)
    rudder_angle = np.clip(p_term, -np.pi/4, np.pi/4)

    # Create control surface command array
    delta_c = np.zeros(n_surfaces)
    if n_surfaces > 0:
        delta_c[0] = rudder_angle

    return delta_c
```

2. Modify the `vessel_ode` method in `class_vessel.py`:

```
if not self.ros_flag:
    # Import controller at the top of the file
    import my_controller

    # Use controller for rudder commands
    self.delta_c = my_controller.pid_heading_controller(
        t, state, self.n_control_surfaces, desired_heading=np.pi/4)  # 45 degrees

    # Set thruster commands if needed
    if self.n_thrusters > 0:
        self.n_c = np.ones(self.n_thrusters) * 1000.0  # Fixed RPM
```

Customizing the Non-ROS Simulator

Modifying Simulation Parameters

You can customize the simulation by modifying the `simulate_noros.py` file:

1. **Change input file path**:
   ```
   sim_params, agents = read_input("/path/to/your/custom_input.yml")
   ```

2. **Modify output format**:

```
# Customize the print statements in the simulation loop
print(f"Custom format: time={vessel.t:6.2f}, position=[{pos[0]:6.2f}, {pos[1]:6.2f}]")
```

3. **Add data logging**:

```
# Add at the beginning of the simulate function
import csv
log_file = open('simulation_log.csv', 'w', newline='')
csv_writer = csv.writer(log_file)
csv_writer.writerow(['Time', 'X', 'Y', 'Z', 'U', 'V', 'W', 'P', 'Q', 'R', 'Phi', 'Theta'

# Inside the simulation loop
csv_writer.writerow([vessel.t, pos[0], pos[1], pos[2], vel[0], vel[1], vel[2],
                     vessel.current_state[3], vessel.current_state[4], vessel.current_sta
                     angles[0], angles[1], angles[2]])
```

Extending the Simulator

You can extend the non-ROS simulator for specific use cases:

1. **Batch simulations**:

```
def run_batch_simulations():
    results = []
    for initial_speed in [0.5, 1.0, 1.5, 2.0]:
        sim_params, agents = read_input("/workspaces/mavlab/inputs/simulation_input.yml'
        agents[0]['vessel_config']['initial_conditions']['start_velocity'][0]
= initial_speed
        vessel =
 Vessel(agents[0]['vessel_config'], agents[0]['hydrodynamics'], vessel_id=
0, ros_flag=False)
        vessel.simulate()
        # Extract results
        final_pos = vessel.history[-1, 6:9]
        results.append((initial_speed, final_pos))
    return results
```

2. **Parameter studies**:

```
def parameter_study():
    import matplotlib.pyplot as plt

    turning_radii = []
    rudder_angles = np.arange(-30, 31, 5) * np.pi / 180  # -30 to 30 degrees
```

```
    for rudder_angle in rudder_angles:
        # Set up simulation with fixed rudder angle
        sim_params, agents =
read_input("/workspaces/mavlab/inputs/simulation_input.yml")
        vessel =
Vessel(agents[0]['vessel_config'], agents[0]['hydrodynamics'], vessel_id=0, ros_flag=Fa

        # Custom controller for fixed rudder
        vessel.control_surface_control_type = 'fixed'
        vessel.fixed_rudder_angle = rudder_angle

        # Run simulation
        vessel.simulate()

        # Calculate turning radius from final path
        # (simplified - would need actual path analysis)
        path = vessel.history[:, 6:8]  # x, y positions
        turning_radius = calculate_turning_radius(path)
        turning_radii.append(turning_radius)

    # Plot results
    plt.plot(rudder_angles * 180 / np.pi, turning_radii)
    plt.xlabel('Rudder Angle (degrees)')
    plt.ylabel('Turning Radius (m)')
    plt.savefig('turning_radius_study.png')
```

Advantages of Non-ROS Mode

The non-ROS mode offers several advantages:

1. **Simplicity**: No need to set up ROS2 environment or handle ROS-specific configurations
2. **Performance**: Lower overhead without ROS middleware, especially for rapid prototyping
3. **Debugging**: Easier to debug core simulation components without ROS complexity
4. **Quick Testing**: Fast iteration for testing vessel dynamics or controller concepts
5. **Portability**: Can run anywhere Python is available, without ROS2 installation
6. **Learning**: Excellent starting point for understanding vessel dynamics before moving to ROS2
7. **Batch Processing**: Efficient for running multiple simulations for parameter studies or optimization

Limitations

Compared to the ROS2 mode, the non-ROS mode has some limitations:

1. **No Visualization**: Lacks the web-based visualization dashboard available in ROS2 mode
2. **Limited External Interaction**: No standardized way to interact with external systems
3. **No Distributed Simulation**: Cannot distribute components across multiple processes/machines
4. **No Standardized Messaging**: No access to ROS2's message types and communication patterns
5. **Manual Data Collection**: Requires custom code for data logging and analysis

Conclusion

The non-ROS mode provides a lightweight alternative for running Panisim simulations without the complexity of the ROS2 ecosystem. It's particularly useful for rapid development, testing, and educational purposes. By understanding the implementation through `ros_flag` and custom controller integration, users can leverage this mode for a wide range of applications while still benefiting from Panisim's accurate vessel dynamics modeling.