# Pani Simulator

Rishabh Sharma

2024-03-24

# 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.

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.

## 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`.

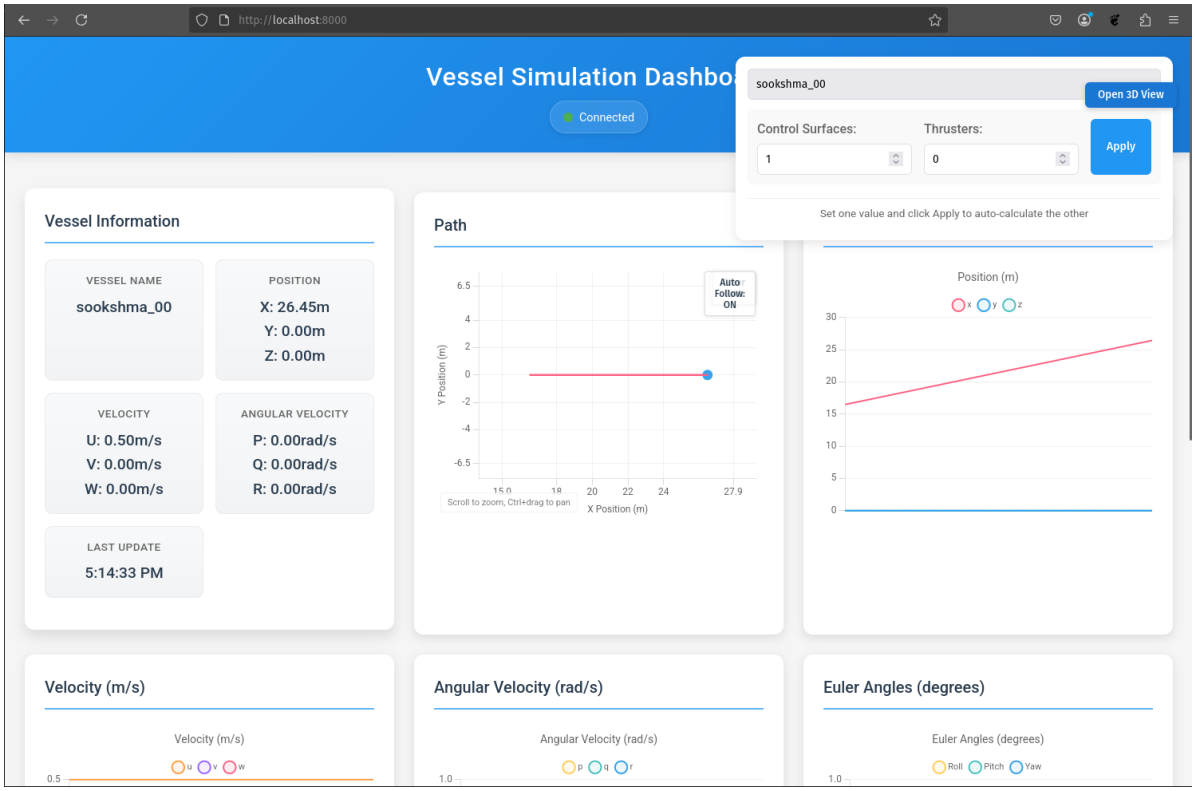



Figure 1: Web-based visualization GUI

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.

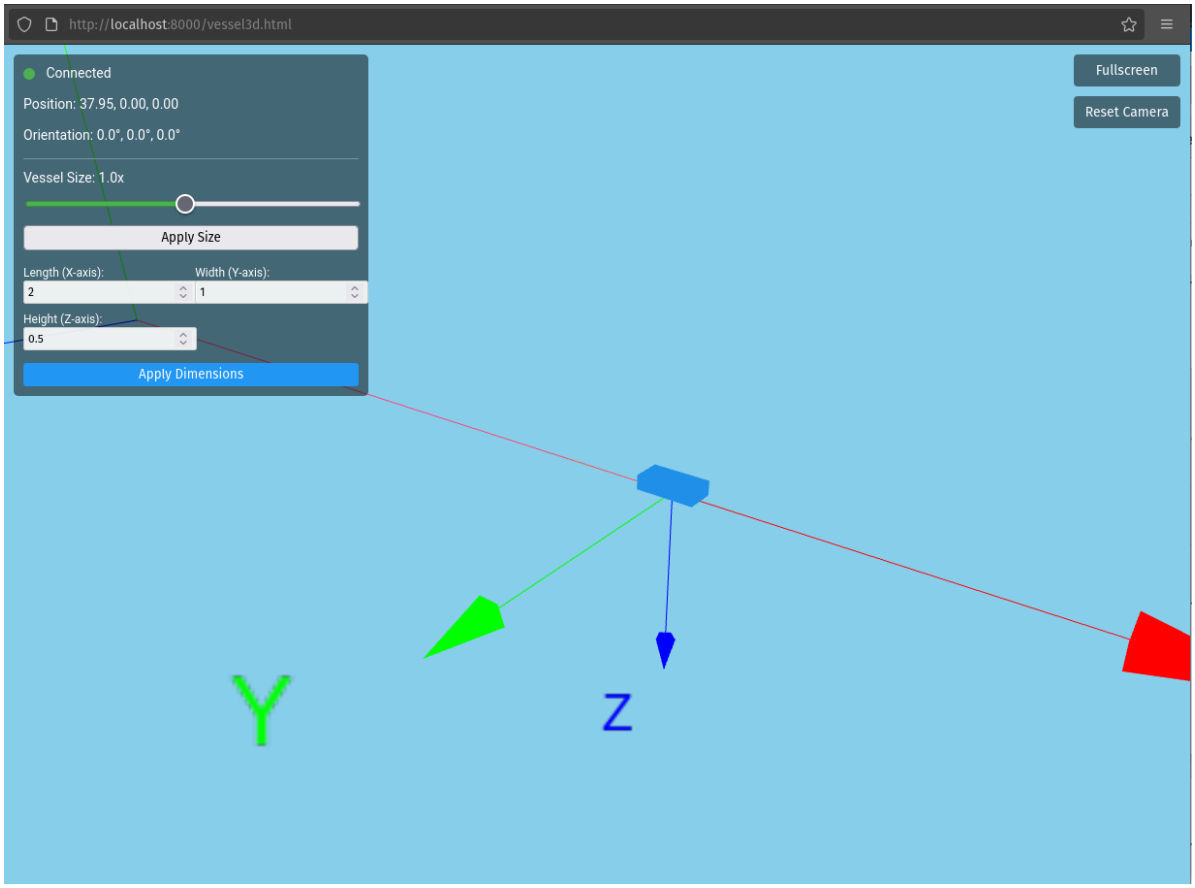



Figure 2: 3D View

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.

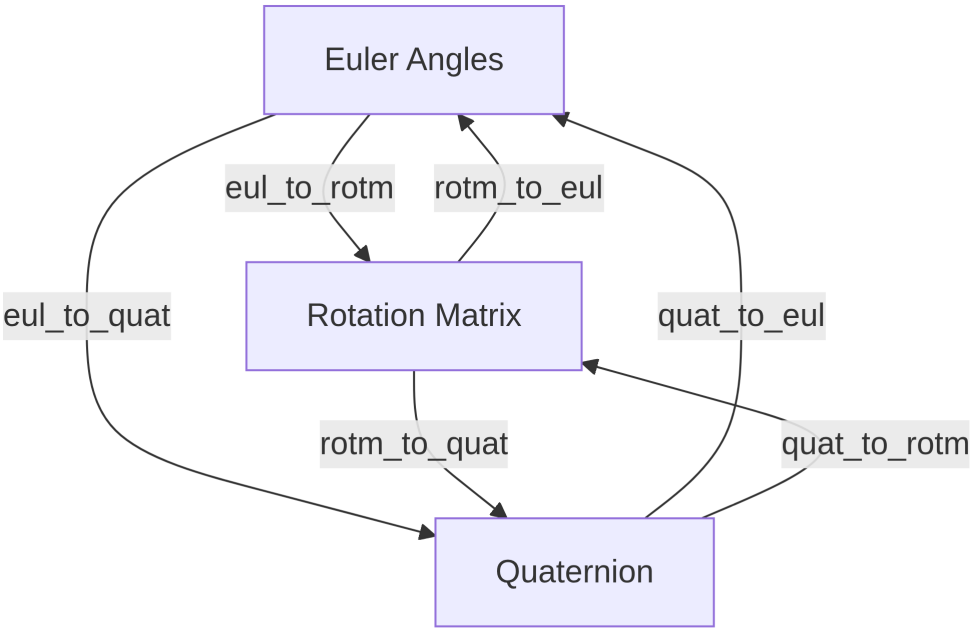
## Coordinate Frames

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 $\phi$, pitch $\theta$, yaw $\psi$) 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 $\phi$, pitch $\theta$, yaw $\psi$). 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 $\phi$, pitch $\theta$, yaw $\psi$) to a unit quaternion $[q_w, q_x, q_y, q_z]$. Assumes ZYX rotation order. Angles can be in radians or degrees. |
| `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 $\phi$, pitch $\theta$, yaw $\psi$). Handles potential ambiguities using `prev_quat` if provided. Can output in radians or degrees. |

| Function | Description |
|---|---|
| `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 $\mathbf{T}(\eta)$ that relates body-frame angular velocity $\omega^b$ to Euler angle rates $\dot{\eta}$ via $\dot{\eta} = \mathbf{T}(\eta)\omega^b$. Assumes ZYX order. |
| `quat_rate_matrix(quat)` | Computes the 4x3 transformation matrix $\mathbf{E}(\mathbf{q})$ that relates body-frame angular velocity $\omega^b$ to quaternion rates $\dot{\mathbf{q}}$ via $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{E}(\mathbf{q})\omega^b$. |
| `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`. |
| `deul_dquat(quat)` | Computes the 3x4 Jacobian matrix $\frac{\partial \eta}{\partial \mathbf{q}}$, representing the partial derivatives of Euler angles with respect to quaternion components. |

| Function | Description |
|---|---|
| `dquat_deul(quat)` | Computes the 4x3 Jacobian matrix $\frac{\partial \mathbf{q}}{\partial \eta}$, 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
- $\tau$ 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

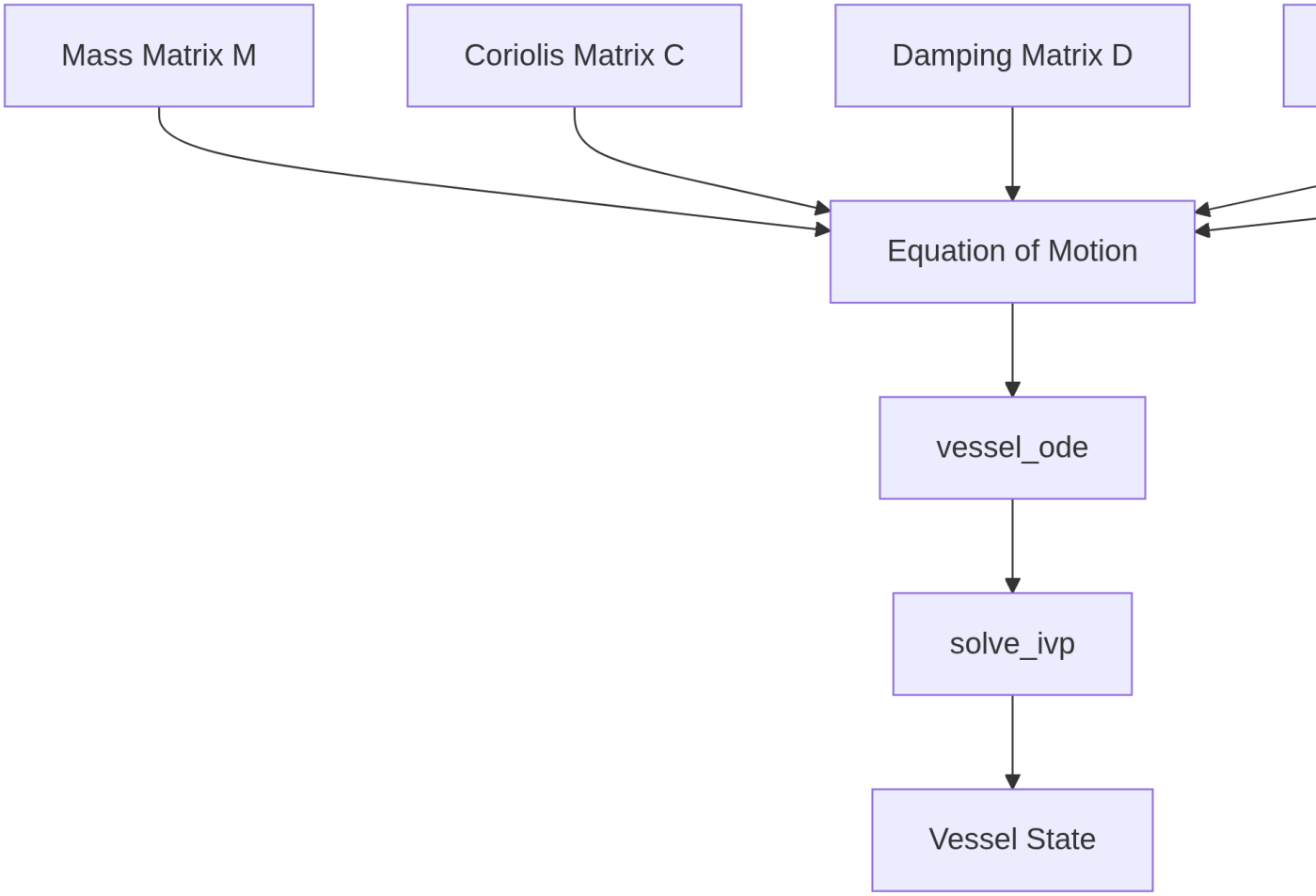
The vessel state is represented as a vector combining velocities, position, orientation, control surface angles, and thruster states:

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 - $\eta_{rot}$ is the orientation, either as Euler angles

$[\phi, \theta, \psi]^T$ or quaternion $[q_0, q_1, q_2, q_3]^T$ - $\delta$ 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 $\nu$.

| 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 $\nabla$ 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 $\tau$ represents the combined effect of control surfaces and thrusters.#### Control Surface Forces

| Function | Description |
|---|---|
| `control_forces(delta)` | Calculates forces and moments generated by control surfaces (rudders, fins, etc.) based on their deflection angles $\delta$. 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)
- $\rho$ 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 $\rho$, length $L$, and characteristic velocity $U$ (mentioned in the `simulation_input.yml` file) |

### Hydrodynamic Coefficient Calculation

| Function | Description |
|---|---|
| `calculate_added_mass_from_hydra(hydra_files)` | 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

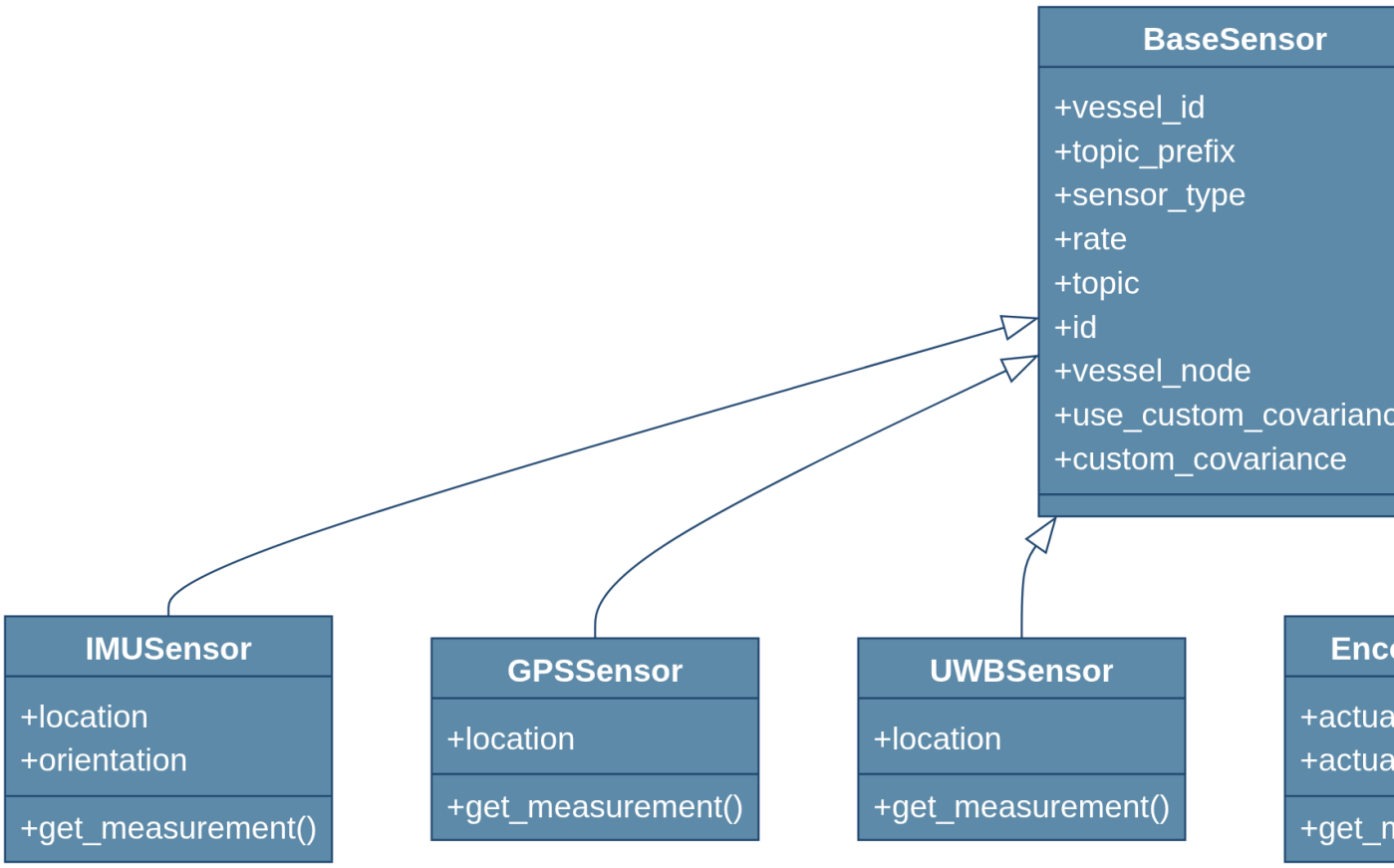
This document explains how sensors are simulated, configured, and how you can create your own custom sensors.

## 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.z

    # 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
    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_noc
    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_covaria
```

```
    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.cust

        # 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

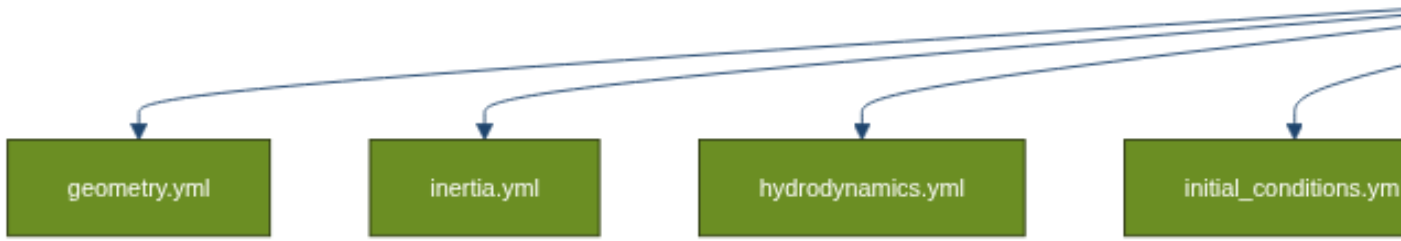
## Overview

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 |
| `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:

```yaml
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 |
| `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-0
      linear_acceleration_covariance: [0.01973958, -0.01976063, 0.02346221, -0.01976063, 0.02
      angular_velocity_covariance: [5.28022053e-05, 4.08840955e-05, -1.15368805e-05, 4.088409

  -
    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,

  -
    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:

```yaml
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 |
| `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 |

| Parameter | Description | Units |
|---|---|---|
| `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**:

```
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
```

```yaml
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

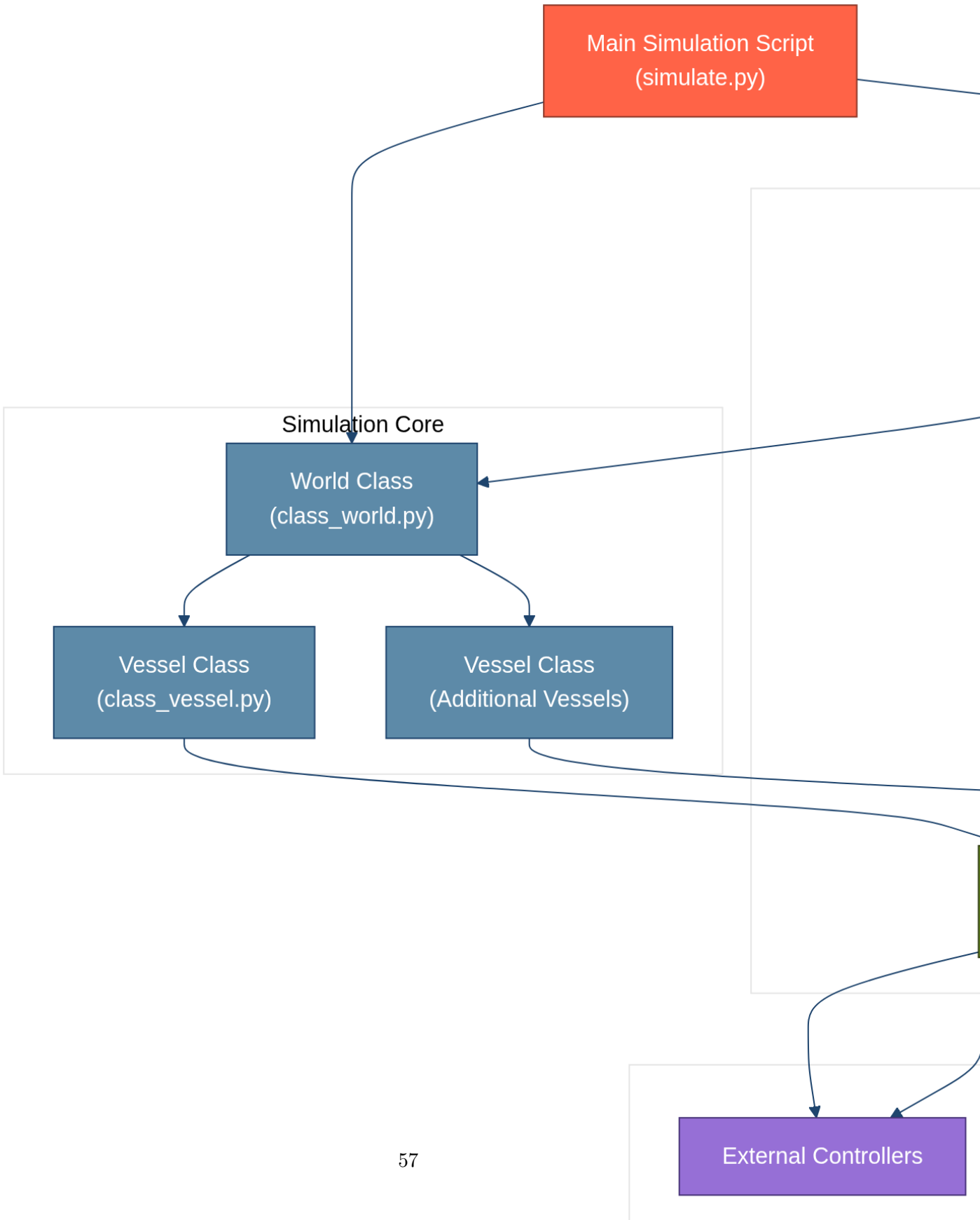
## Overview

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:


Main Simulation Script
(simulate.py)
Simulation Core
World Class
(class_world.py)
Vessel Class
(class_vessel.py)
Vessel Class
(Additional Vessels)
External Controllers

**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:

```
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:

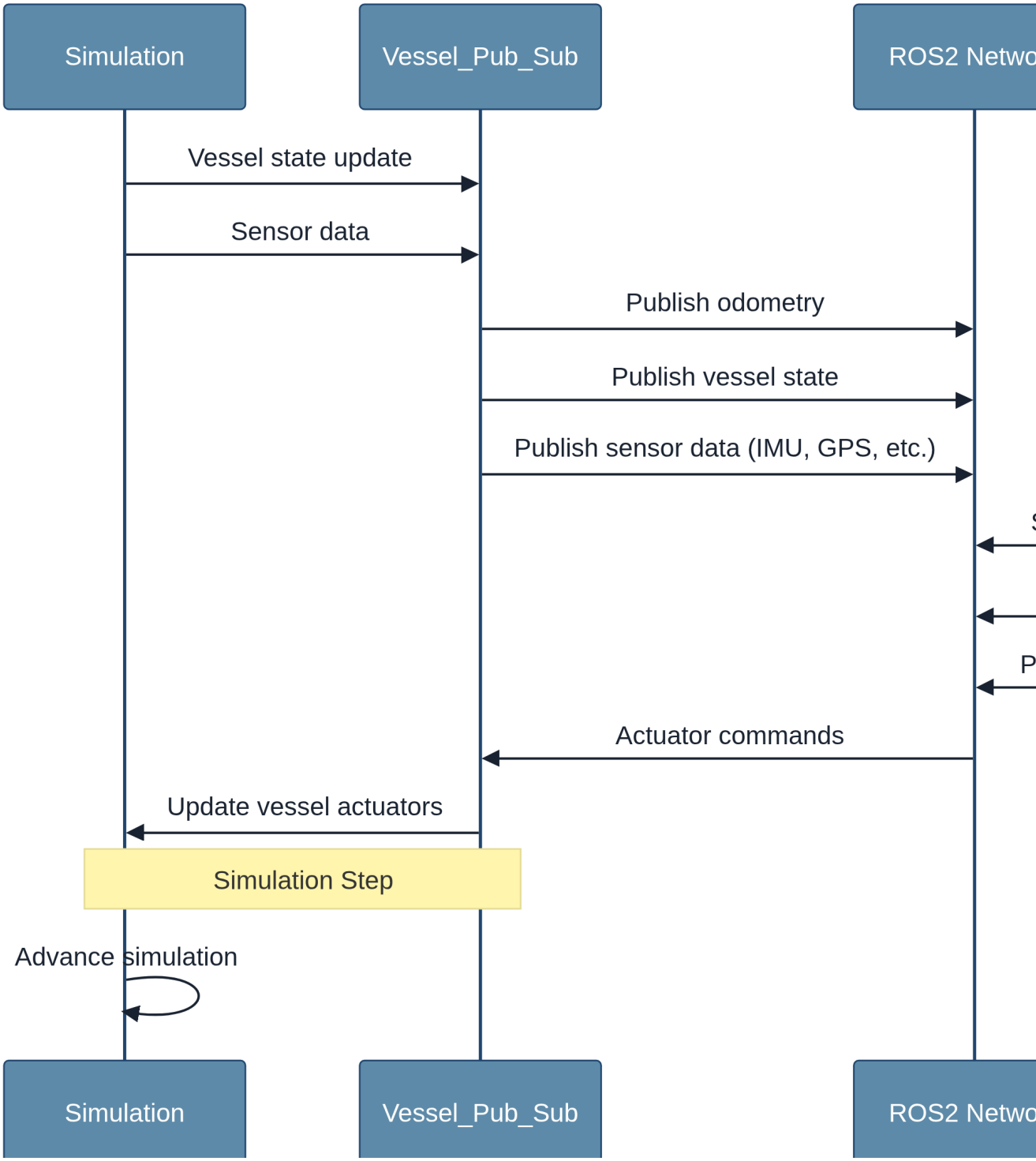
- Creates and manages ROS2 publishers for vessel state and sensor data
- Sets up subscribers for actuator commands
- Handles message conversion between simulator data and ROS2 messages
- Manages sensor data publication at appropriate rates

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` | `nav_msgs/Odometry` | Vessel position, orientation, and velocities |

| Topic | Message Type | Description |
|---|---|---|
| /<vessel_name>_<id>/vessel_state | std_msgs/Float64MultiArray | Complete vessel state including actuators |
| /<vessel_name>_<id>/actuators/cmd | interfaces/Actuator | Commands for control surfaces and thrusters |

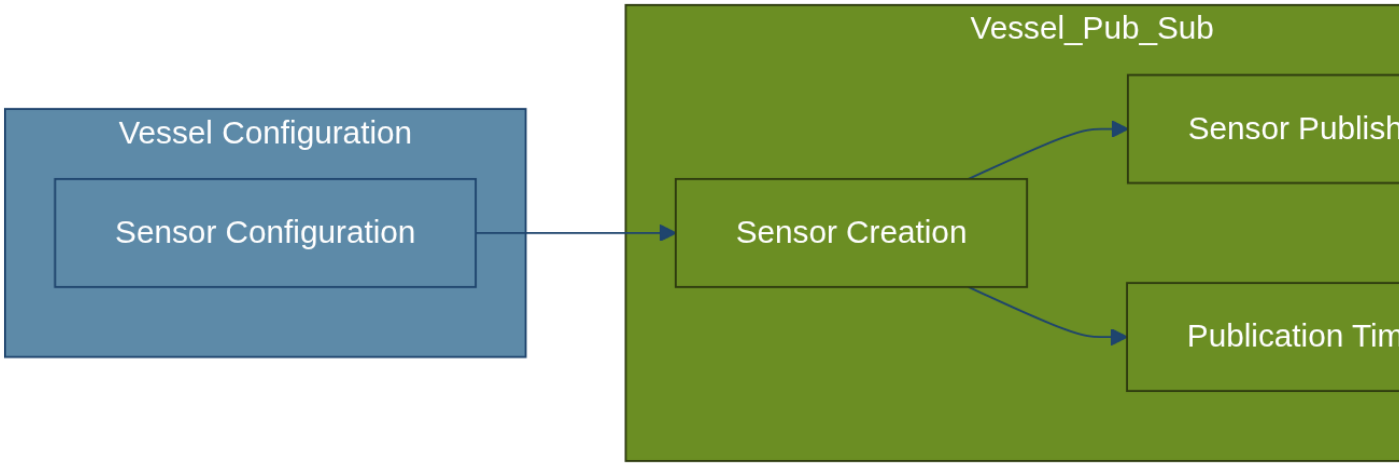
### Sensor Topics

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>/pwb | 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

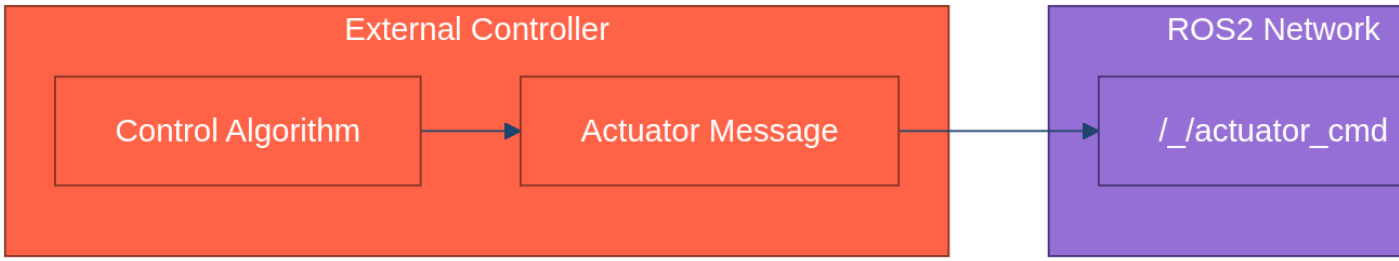
Key methods in the sensor integration:

```
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`:

```
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

            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}
        except (ValueError, IndexError):
            self.world_node.get_logger().warn(f'Unknown actuator ID: {actuator_id}')
```

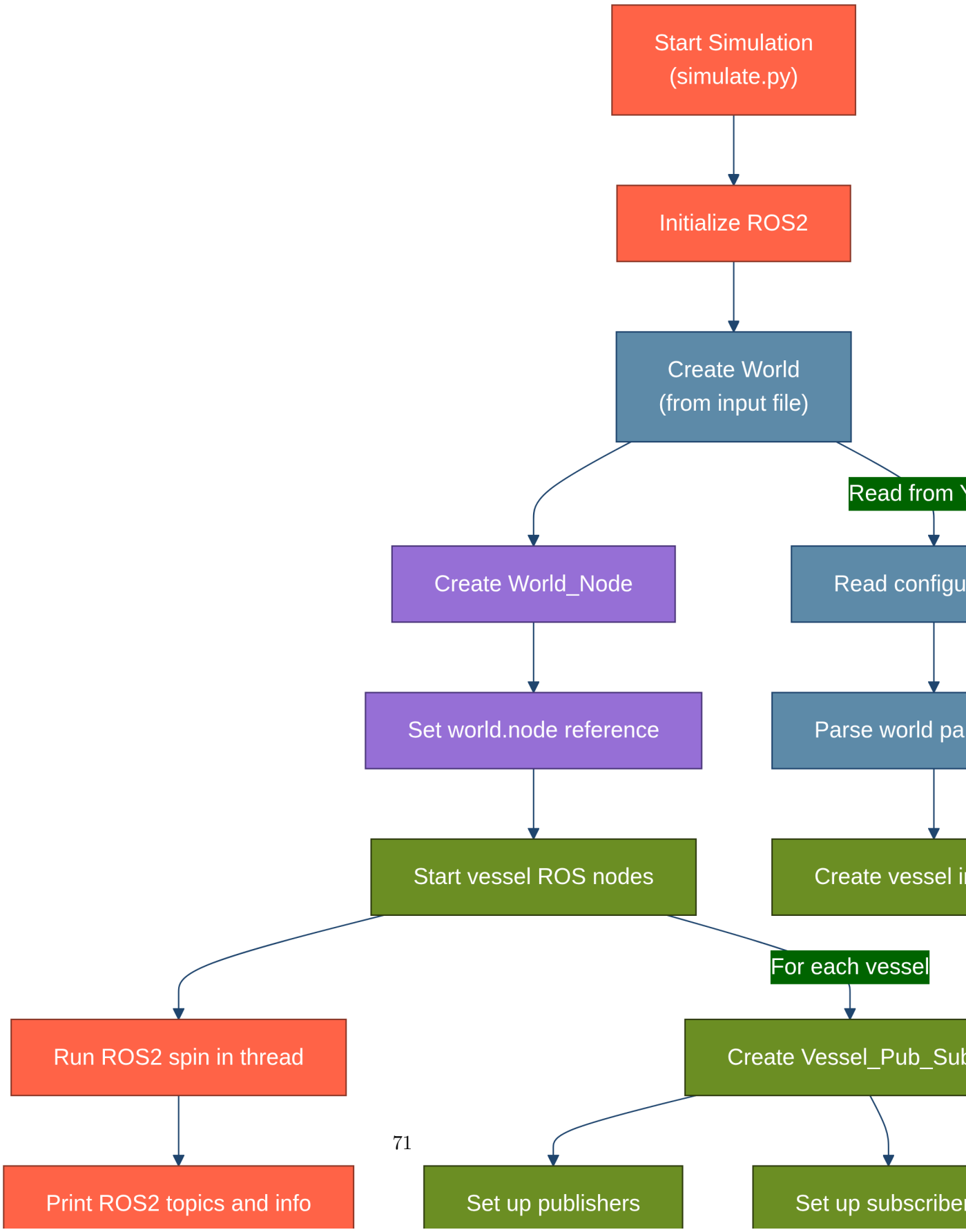
During initialization, `Vessel_Pub_Sub` builds mappings between actuator IDs (as defined in `control_surfaces.yml` and `thrusters.yml`) and their indices:

```
# 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:


Start Simulation
(simulate.py)
Initialize ROS2
Create World
(from input file)
Read from Y
Create World_Node
Read configu
Set world.node reference
Parse world pa
Start vessel ROS nodes
Create vessel i
For each vessel
Run ROS2 spin in thread
Create Vessel_Pub_Sub
Print ROS2 topics and info
Set up publishers
Set up subscriber

**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
    'timer': self.world_node.create_timer(self.vessel.vessel_config['time_step'], se
}
```

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._publ
})
```

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.

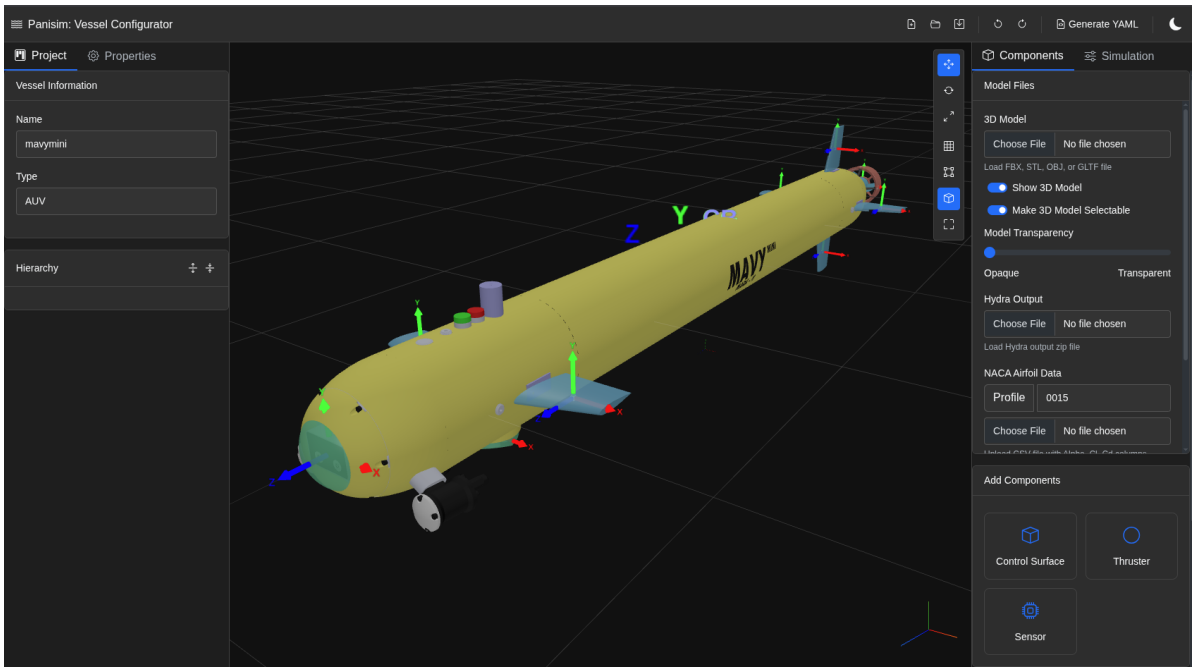



Figure 3: An example configuration in the vessel configurator

> **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.

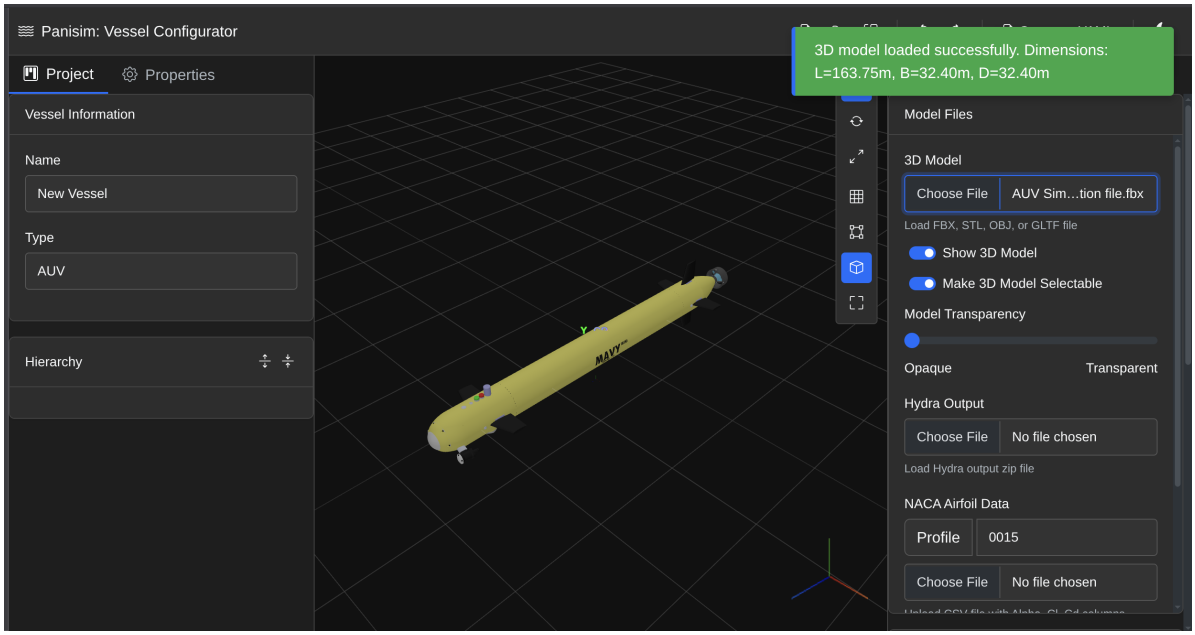



Figure 4: Loaded Vessel

> **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.

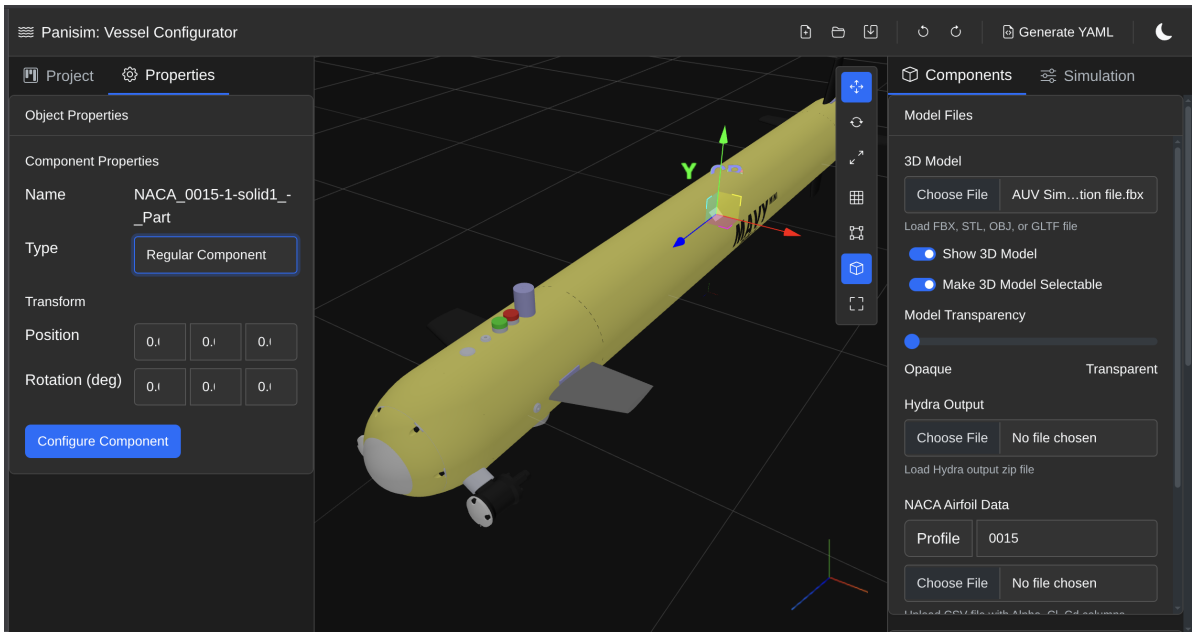



Figure 5: Selected Component

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.

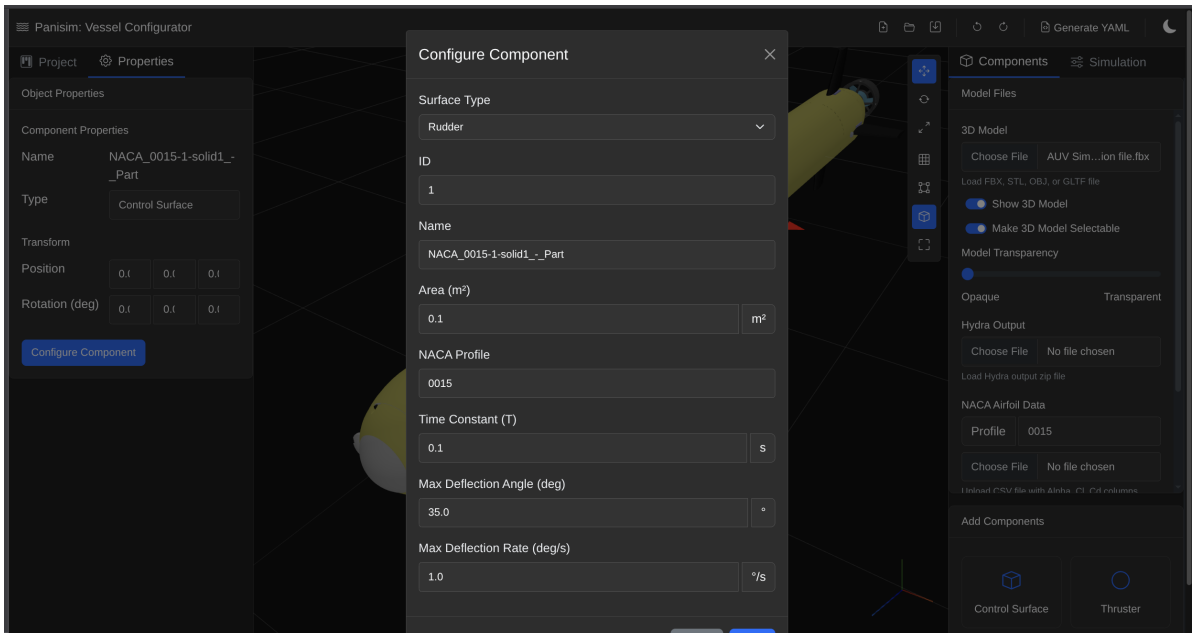



Figure 6: Component Configuration Modal

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.

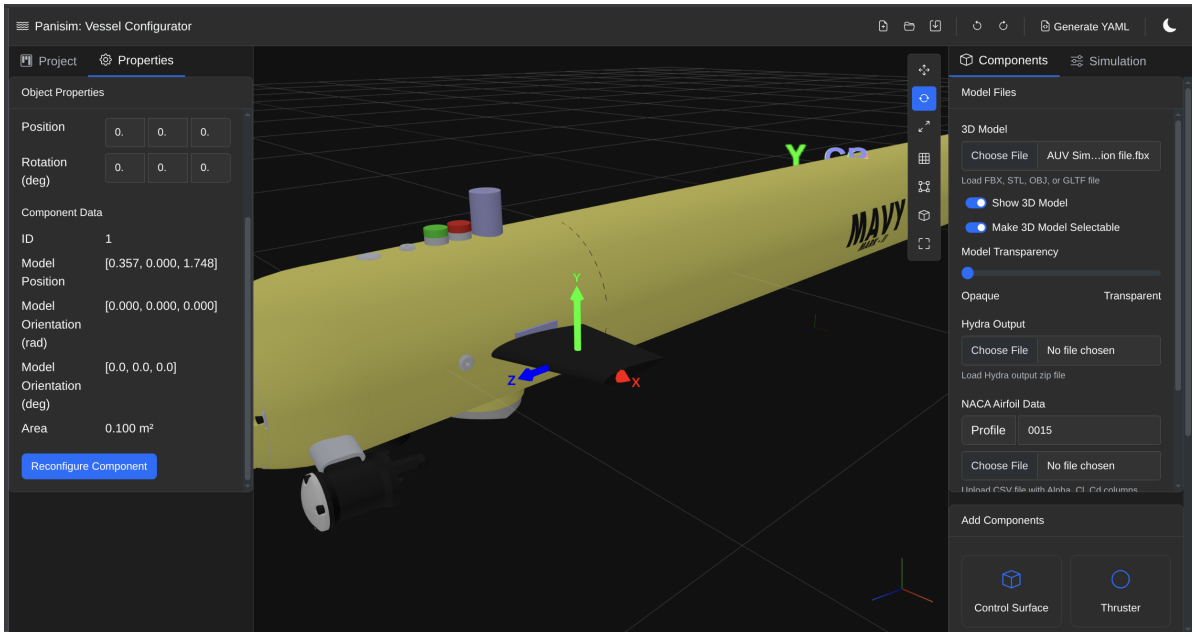



Figure 7: Component Configured

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.

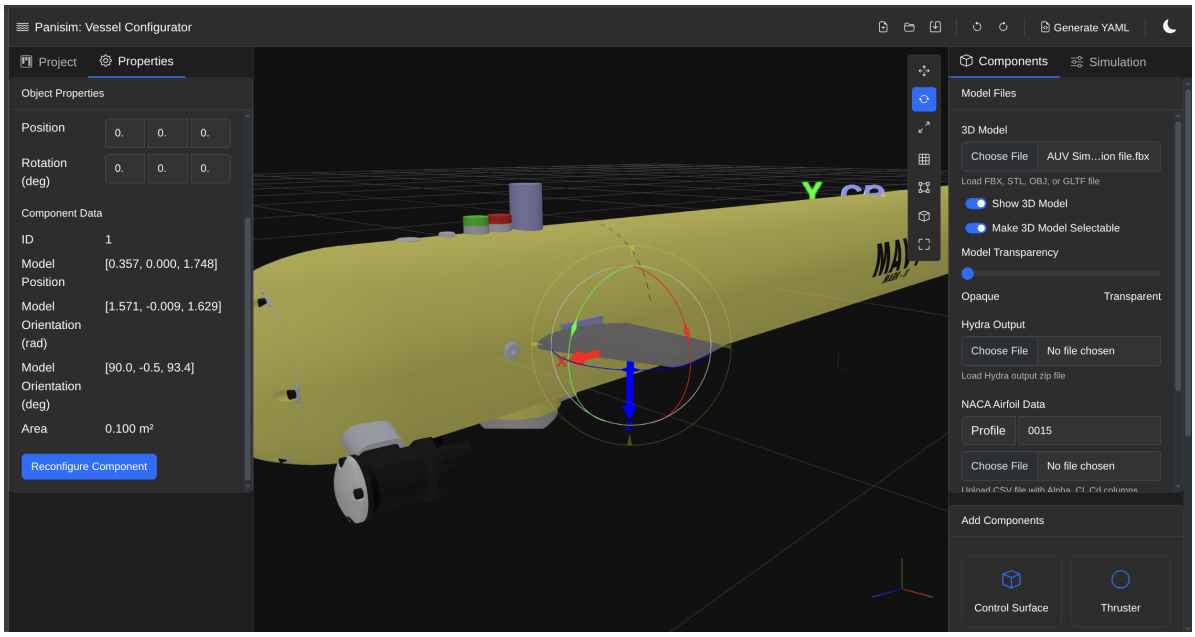



Figure 8: Component Configured

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.

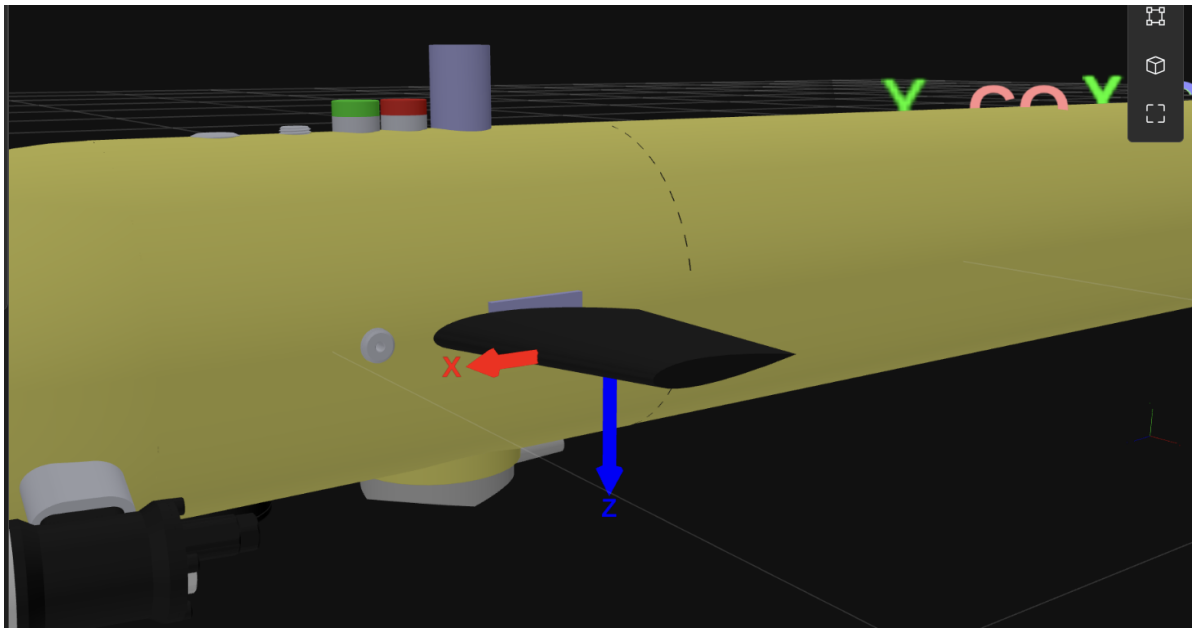



Figure 9: Control Surface

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.

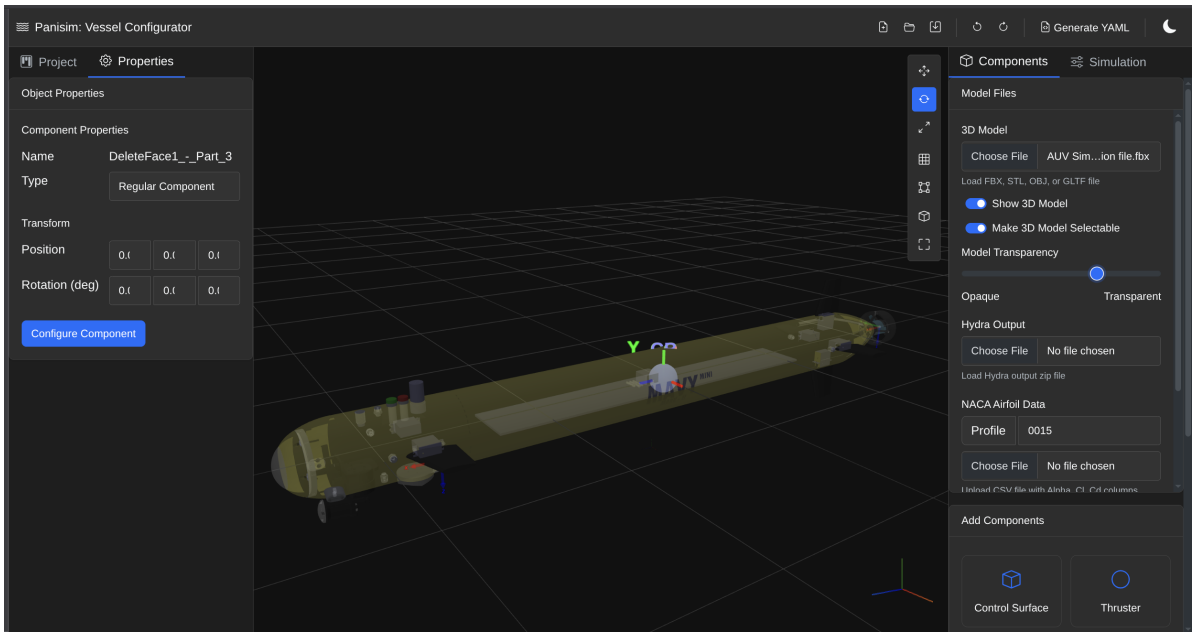



Figure 10: Transparent Vessel

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.

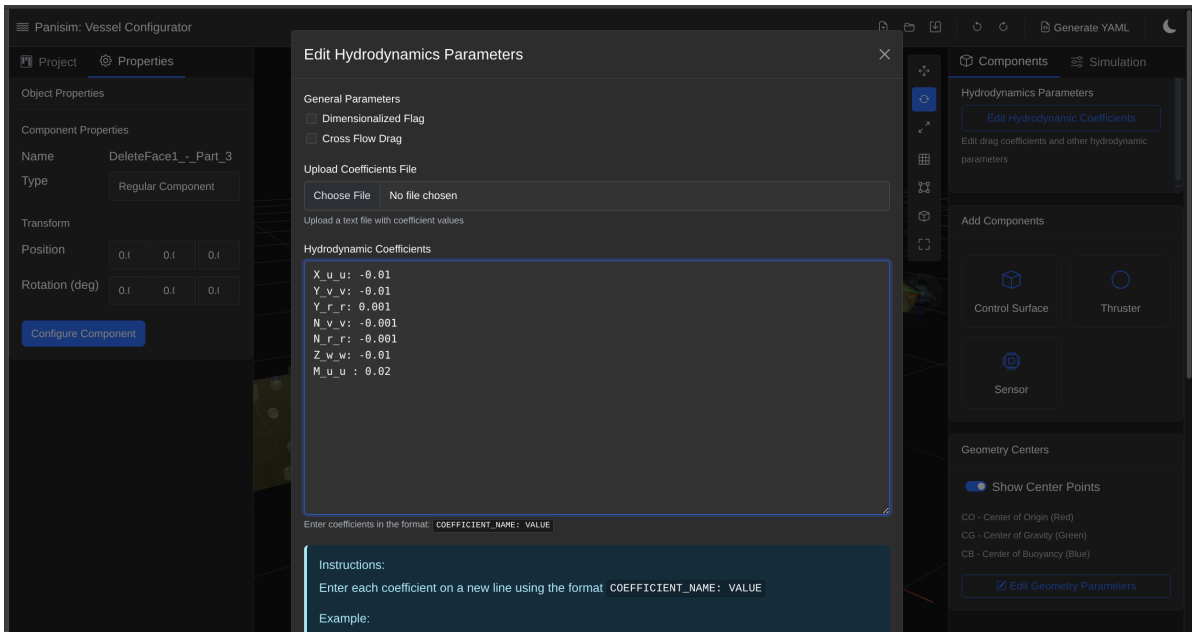



Figure 11: Hydrodynamic Coefficients

## 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.

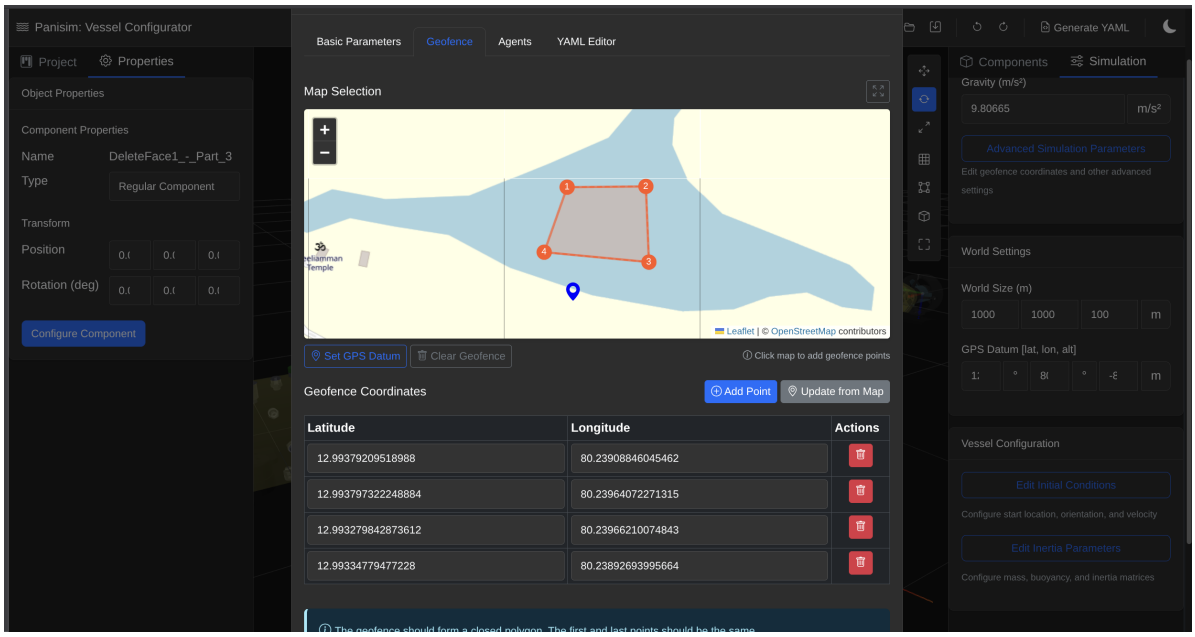



Figure 12: Geofence

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

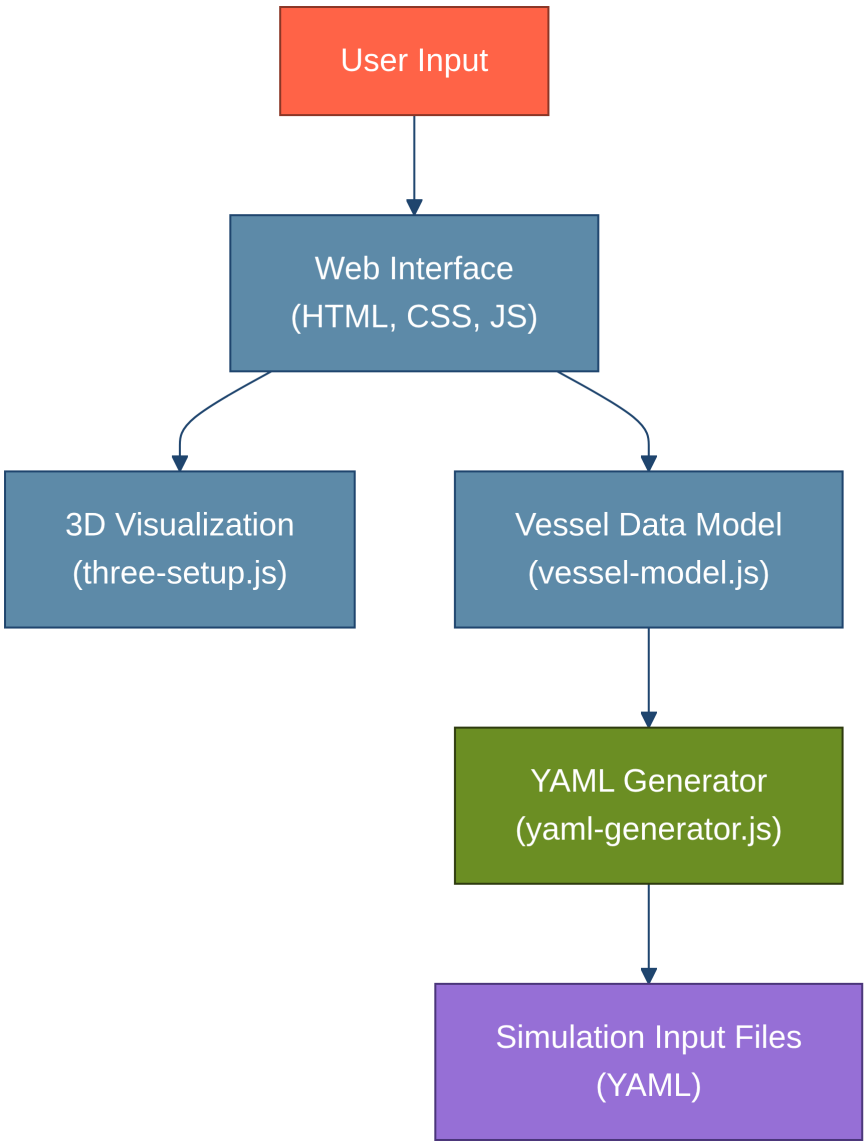
You can get your Hydra output file from the HydRA website. And upload the zip folder in the web gui. The generated input file will then contain your Hydra file with correct path.

## 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