Path Planning of an Autonomous Underwater Vehicle using ROS/Gazebo/UUV_Simulator

MAE 494/598: Introduction to Autonomous Vehicles Engineering

Path Planning Of Autonomous Underwater Vehicle Using ROS Gazebo Simulation

Abstract:
The integration of advanced robotic frameworks such as the Robot Operating System (ROS) with high-fidelity simulation tools like Gazebo has significantly enhanced the design and testing of path-planning algorithms for Autonomous Underwater Vehicles (AUVs). This study explores the implementation and evaluation of real-time collision avoidance and path planning using the A* algorithm within a ROS-Gazebo simulation environment. The system architecture incorporates underwater-specific sensor models, including Doppler Velocity Logs (DVL), Lidar, and inertial measurement units (IMUs), to emulate perception in a realistic submerged environment. The A* algorithm is employed for global path planning, generating optimal trajectories in a grid-based environment, while reactive obstacle avoidance is handled through proximity sensor feedback. This work emphasizes the practical challenges of simulating underwater conditions, such as limited visibility, sensor noise, and hydrodynamic constraints, and contrasts these with anticipated real-world deployment issues. The results validate the robustness of classical algorithms like A* in simulation while highlighting the performance gap when transitioning toward real-world applications. By leveraging ROS tools, the system supports modular testing, visualization, and logging, offering a replicable framework for future research. The study concludes with recommendations for integrating adaptive and AI-based methods into classical pipelines for improved autonomy and robustness in unpredictable underwater environments.

1. Introduction

This report details the design, implementation, and evaluation of a LiDAR-based path planning system for Autonomous Underwater Vehicles (AUVs). The project leverages the Robot Operating System (ROS), Gazebo simulation, and the A* algorithm to enable safe navigation in GPS-denied, complex underwater environments with small obstacles[1].

2. Why Autonomous Underwater Vehicles?

- AUVs are crucial for:
- Oceanographic surveys
- Pipeline inspection
- Search and rescue missions
- Military reconnaissance

They excel in environments inaccessible to humans and where GPS is unavailable, making robust navigation solutions essential.

3. Challenges in Underwater Navigation

AUV navigation faces several unique challenges:

- **No GPS Access:** GPS signals do not penetrate water, requiring alternative localization methods.
- **Dynamic Environments:** Currents, moving obstacles, and changing terrain complicate path planning.
- **Limited Visibility:** Turbidity and low light hinder traditional vision sensors.

4. Objective

The objective of this project is to implement efficient and reliable path planning for Autonomous Underwater Vehicles (AUVs) using LiDAR-based perception. Underwater environments pose unique challenges such as limited visibility, dynamic obstacles, and the absence of GPS. LiDAR sensors offer millimeter-level resolution and fast scanning capabilities, making them ideal for detecting nearby obstacles with high accuracy, even in turbid water when equipped with suitable wavelengths. By integrating LiDAR with A* path planning, we can generate real-time, collision-free routes for AUVs navigating complex underwater terrains. This approach allows the AUV to autonomously adapt its trajectory in response to environmental changes, ensuring safer operations during missions like coral reef monitoring, pipeline inspection, and search and rescue.

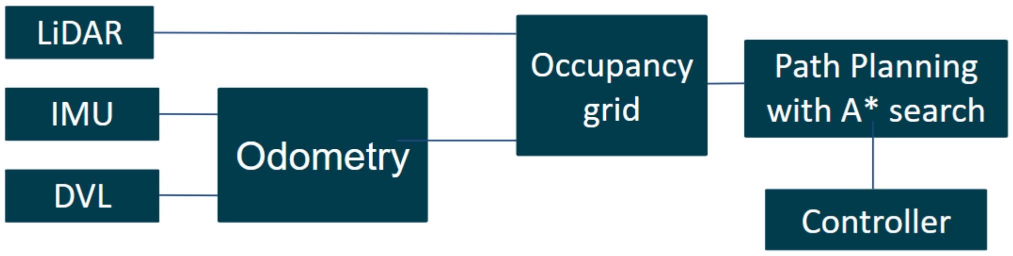
5. System Overview

5.1. How Does an AUV Work in ROS-Gazebo-UUV Simulator?

To implement and test autonomous underwater path planning, we used the Robot Operating System (ROS) framework integrated with Gazebo and the UUV Simulator plugin package. This architecture allowed us to simulate realistic underwater physics, sensor behavior, and autonomous control strategies in a controlled environment.

ROS provides the middleware layer for communication between sensor data, control logic, and simulation interfaces.
Gazebo serves as the 3D physics simulator where the underwater environment and robot dynamics are visualized and calculated.
The **UUV Simulator** is a specialized ROS-Gazebo extension developed specifically for underwater vehicles. It enables accurate simulation of:

- Underwater physics (e.g., buoyancy, drag, added mass),
- Sensor inputs (LiDAR, IMU, DVL),
- Actuator models (thrusters),
- Controllers for autonomous operation.



Three Key Components in the UUV Simulator Setup

1. **Underwater Physics:** Implemented using hydrodynamic models accounting for buoyancy, added mass, and drag.
2. **Sensors:** Simulated LiDAR, IMU, and DVL provide realistic data streams.
3. **Controller:** Sends thrust commands based on planner output.

ROS handles the data flow between sensors, planning algorithms, and controllers, allowing for seamless real-time simulation.

5.3. Physics Behind Underwater Vehicle Motion in the UUV Simulator

From Rigid Body to Fossen's Hydrodynamic Model

Gazebo inherently supports **Newton-Euler rigid body dynamics** through the equation:

$$M_{RB}\dot{\nu} + C_{RB}(\nu)\nu + g_0 = \tau_g$$

Where:

- M_{RB} is the rigid-body mass matrix,
- $C_{RB}(\nu)$ represents Coriolis and centripetal forces,
- g_0 includes gravitational forces,
- τ_g is the sum of applied control and environmental forces.

However, for underwater simulation, **rigid body physics is insufficient** because it ignores key underwater phenomena like added mass, water drag, and hydrodynamic damping. Therefore, the UUV Simulator replaces this with **Fossen's equations of motion**, tailored for marine vehicles [2][3]:

$$(M_{RB} + M_A)\dot{\nu}_r + (C_{RB} + C_A)\nu_r + D(\nu_r)\nu_r + g(\eta) = \tau$$

Where:

- M_A is the added mass matrix (representing the inertia of displaced water),
- $C_A(\nu_r)$ adds hydrodynamic Coriolis terms,
- $D(\nu_r)$ accounts for damping due to drag,
- $g(\eta)$ represents gravity and buoyancy,
- ν_r is the relative velocity between the vehicle and water currents,
- τ is the net control input (thruster forces).

How This Affects Simulation

In the UUV Simulator, this equation is implemented through hydrodynamic plugins that compute **external forces (τg)** each time step[2][3]:

$$\tau_g = -M_A\dot{\nu}_r - C_A(\nu_r)\nu_r - D(\nu_r)\nu_r - g(\eta)$$

These forces are applied to the simulated AUV, resulting in realistic underwater behaviors like:

- Gradual acceleration/deceleration due to added mass,
- Slowdowns from drag,
- Automatic buoyant ascent or descent,
- Reduced maneuverability due to water resistance.

This makes the AUV's simulated behavior **physically consistent**, ensuring that planned paths must be feasible under real-world underwater constraints [2][3].

Sensor Integration

We simulated and integrated three primary sensors:

1. LiDAR

- Emits laser pulses and receives returns to detect obstacles.
- Data used to generate the occupancy grid.

2. IMU (Inertial Measurement Unit)

- Provides angular velocity and acceleration.
- Essential for orientation estimation.

3. DVL (Doppler Velocity Log)

- Uses sonar beams to measure velocity relative to the seabed.
- Crucial for odometry and localization in GPS-denied environments.

Together, these sensors provide full situational awareness and feed data into the path planner.

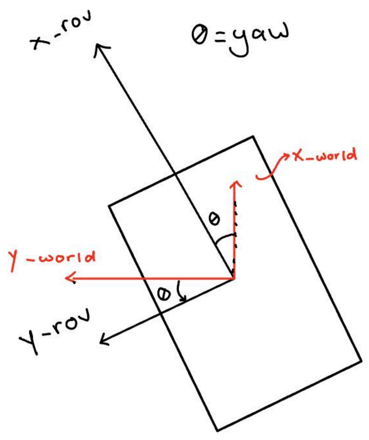
6. Sensor Fusion and Localization - Odometry Message

The majority of the sensor fusion and localization done in the solution are done using an Odometry script, which is shown in full in Appendix I. Odometry refers to a robot's ability to use sensors to estimate its current position with respect to the world's reference frame, as opposed to from its own reference frame [5]. Without considering odometry, the robot would experience significant accumulated error in determining its own true position, and, by extension, determining the position of the next logical step in its path.

For this project, because Python was the primary language used, the rospy library was used to effectively process and perform computations based on ROS message information. The odometry script begins by subscribing to the IMU and DVL messages. Both of these messages are easy to implement within Gazebo because the two sensors are included on the RexRov2: the vehicle used for testing [8]. The odometry message type was also classified as the ROS node which this data would be published to. The odometry message type, similarly to all ROS messages, has a specific way of storing data which is important to understand if that data is to be published to it and accessed from it [6]. nav_msgs/Odometry stores two key pieces of information regarding the robot: a position matrix, and a velocity matrix. sensor_msgs/IMU and the UUV_Simulator DVL message both have their own characteristics which can be used to populate these matrices in the Odometry message. After the robot's current position, orientation/tilt, and velocities in each 2D direction are initialized, their variables are populated with information from the IMU and DVL messages.

An important element is its orientation storage. It is stored as a quaternion. For general geometric purposes this is less useful than the standard orientation angles roll, pitch, and yaw. For this reason, the transformation function euler_from_quaternion was used to extract these angles from what the IMU message provided [7].

The next function shown in Appendix I, defined as rov_to_world_frame, is the most important of them all. It is where the standard transformation matrix is implemented to convert the linear velocities of the robot in the x and y direction and undergo a coordinate transformation to place them in the world's reference frame, by accounting for the angle to which the robot is tilted and course correct. It utilizes the geometry and trigonometry laid out in the following diagram:



$$\vec{V}_{wrld} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \vec{V}_{rov}$$

$$V_{world,x} = (V_{rov,x} \cdot \cos\theta) - (V_{rov,y} \cdot \sin\theta)$$
$$V_{world,y} = (V_{rov,x} \cdot \sin\theta) + (V_{rov,y} \cdot \cos\theta)$$

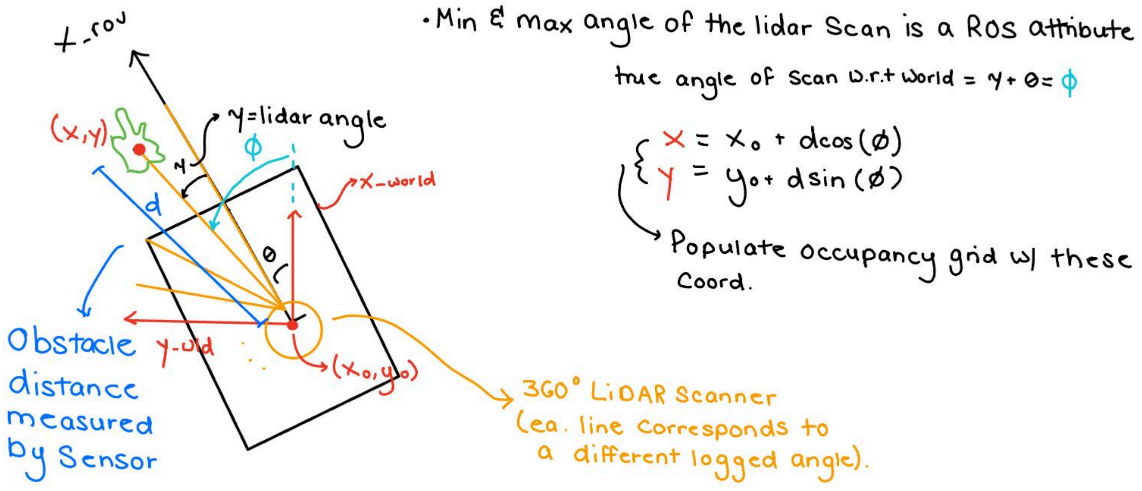
The way in which the x and y world-axes are defined above is the ROS standard [4]. The angle of yaw is used to generate a transformation matrix to convert the velocity values into the x and y coordinate frames of the world [5].

The reason velocity values are needed is so that they can be integrated with respect to time, in order to generate an accurate position with reference to the world reference frame. This is used to update the odometry position message. Finally, the IMU orientation quaternion information is automatically passed to the odometry message's orientation information. All of this is done within the update_odometry function shown in Appendix I. The syntax of each piece of data corresponds with the notation referenced in nav_msgs/Odometry, which can be accessed through the ROS documentation [10].

7. Environmental Mapping - Occupancy Grid Message

The next script for ROS message interfacing and progress through the flowchart is the Occupancy Grid Script, shown in full in Appendix II. An occupancy grid serves as a, in-this-case, 2D array of grid points of a user-specified size which contain values of either 0(free), 100(filled with an obstacle), or -1(unknown). This one subscribes to both the aforementioned odometry message, as well as a sensor_msgs/LaserScan message, which is the 2D Lidar message in ROS. After the map characteristics and positions are initialized and the subscribing/publishing nodes are identified, the odometry data for position and orientation explained in the previous section are read, as is a crucial set of LiDAR data which will allow the AUV to recognize obstacles. In the lidar_input function, the minimum and maximum angles of the scan are read. For the LiDAR sensor used by the team, this was a range of 0 to 2π rad.

The most crucial information read from the LaserScan message is the ranges value, which is an array of distance values returned for each angle within the scan. These represent the positions relative to the robot's forward reference frame of obstacles: their distance vector, and their angle [11]. These distances also need to be converted into the world reference frame if they are to be used to plot obstacles on a grid. This adds an additional layer of difficulty to the previous coordinate transformation as the angle of the scan also needed to be taken into account, requiring the implementation of a for loop. This coordinate transformation is shown in the figure below:


x-rov
(x,y)
γ=lidar angle
ϕ
γ
d
x-world
θ
Obstacle distance measured by Sensor
y-wld
(x₀,y₀)
360° LiDAR Scanner (ea. line corresponds to a different logged angle).
• Min & max angle of the lidar Scan is a ROS attribute
true angle of Scan w.r.t World = γ + θ = ϕ
x = x₀ + dcos(ϕ)
y = y₀ + dsin(ϕ)
Populate occupancy grid w/ these Coord.

This generates a set of x and y coordinates in the world reference frame to be marked as featuring obstacles within the occupancy grid. Based on the characteristics of the Occupancy Grid message as described in the ROS message notation, assigning a space with a value of 100 indicates that an obstacle is present there. This is done for every distance within the LiDAR's reliable range, which is 3 m in this use-case (this is the if loop shown in the Appendix II lidar_input). The inspiration for this block of code was derived from a solution on a ROS forum [16]. A function is then defined to create the occupancy grid in such a format so that it can be published to the ROS Occupancy Grid message and read by the path planning algorithm discussed further below. The final line of this section of code, rospy.spin(), keeps the script running until it is manually stopped, allowing for continuous message transmission. This line was also present in the aforementioned Odometry script.

One element of this script which is very important to note is that it is only capable of publishing an occupancy grid within the LiDAR sensor's capable range. This requires iteration in the next message to ensure that longer goal ranges can be achieved.

8. Path Planning Algorithm - Simple Straight-Line A* Algorithm Implementation

Once an occupancy grid has been generated, it can be fed directly into a path-planning algorithm which outputs the shortest, obstacle free path of movement for the AUV. This code is featured in Appendix III in full. This is where machine learning and artificial intelligence become most relevant to the project. This script subscribes to the Occupancy Grid message populated in the previous section. A* functions by searching grids for efficient paths to a designated goal position.

In basic terms, the A* algorithm functions off of a very simple cost function:

$$f = g + h$$

In this equation, f represents the cost function value, g represents the distance from a starting point within the grid which is fed to the algorithm. h represents the distance to a goal point which is also specified, and is calculated using the Euclidean distance function. There are alternatives however they were not fully explored. For each space within the grid, the algorithm follows the path which minimizes the cost function.

Within the A* class, there are two functions defined as check_neighbors and check_for_obstacles. These two functions add each of the 8 grids surrounding the AUV's current position to a 2D array (pairs of x, y coordinates), and mark them with obstacles if they are marked as such within the Occupancy Grid map [17]. The positions with obstacles are not considered in the calculations of each f value: only those in the open list are considered. The current position of the AUV also needs to be updated with each iteration.

Two very important functions present within this code are the heappop and heappush functions. These are used to establish a priority queue system within the code, which will easily return the lowest cost function value at each iteration. The heuristic distance from start to goal is used as the initial value of f, and for each iteration of the code, the grid coordinates of the point with the lowest tentative cost function are added to a list which contains each path coordinate[18]. Within the code in Appendix III, all of the occupancy grid information is processed into something which this algorithm can work with, and a path publisher message is established, which will enable the AUV's control.

Before moving into ROS, the viability of this algorithm at moving across small distances after receiving low range LiDAR updates was simulated in a regular Python environment. One such result, using MatPlotLib to plot, is shown below, This used simulated, rapidly updating obstacle values as the distance covered by the LiDAR sensor increased. As shown, the algorithm was effective at avoiding randomly generated obstacles. The script for this full setup is featured in Appendix IV.

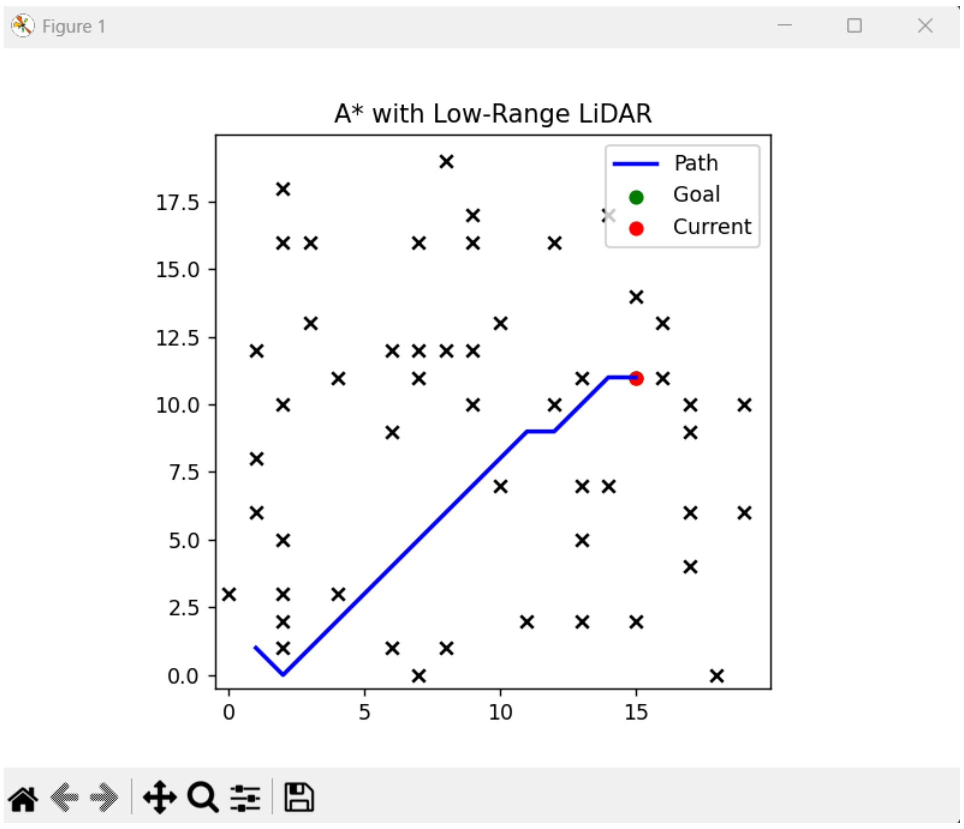
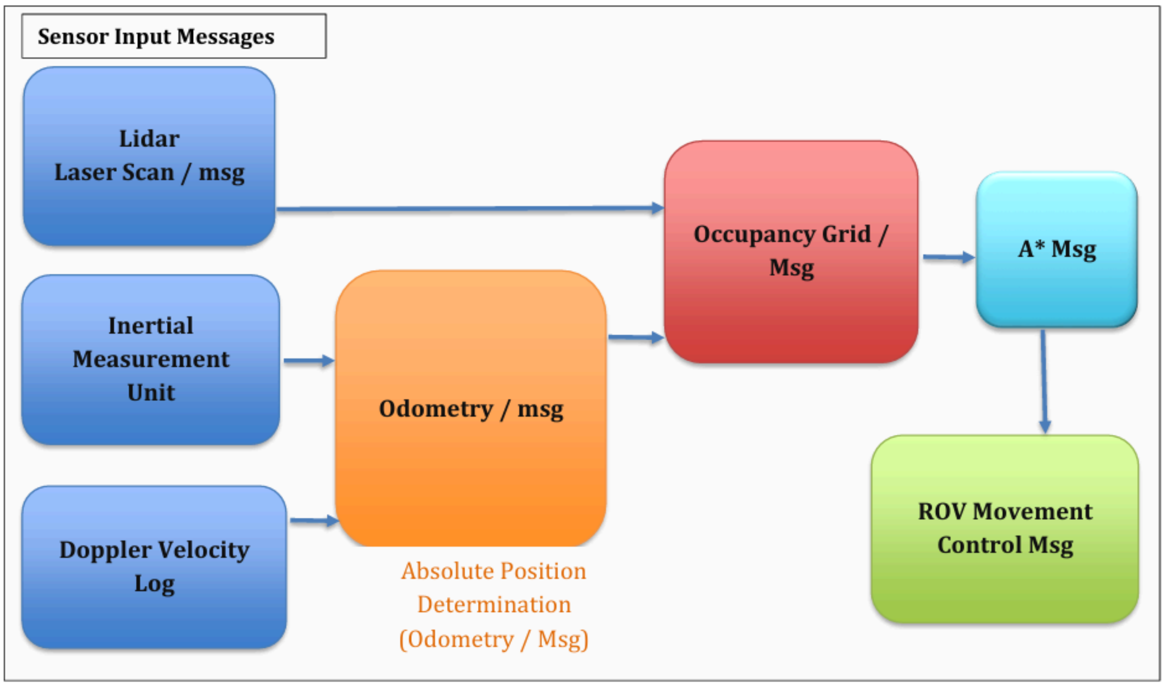


Figure 1
A* with Low-Range LiDAR
Path
Goal
Current
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15

9. Flowchart:

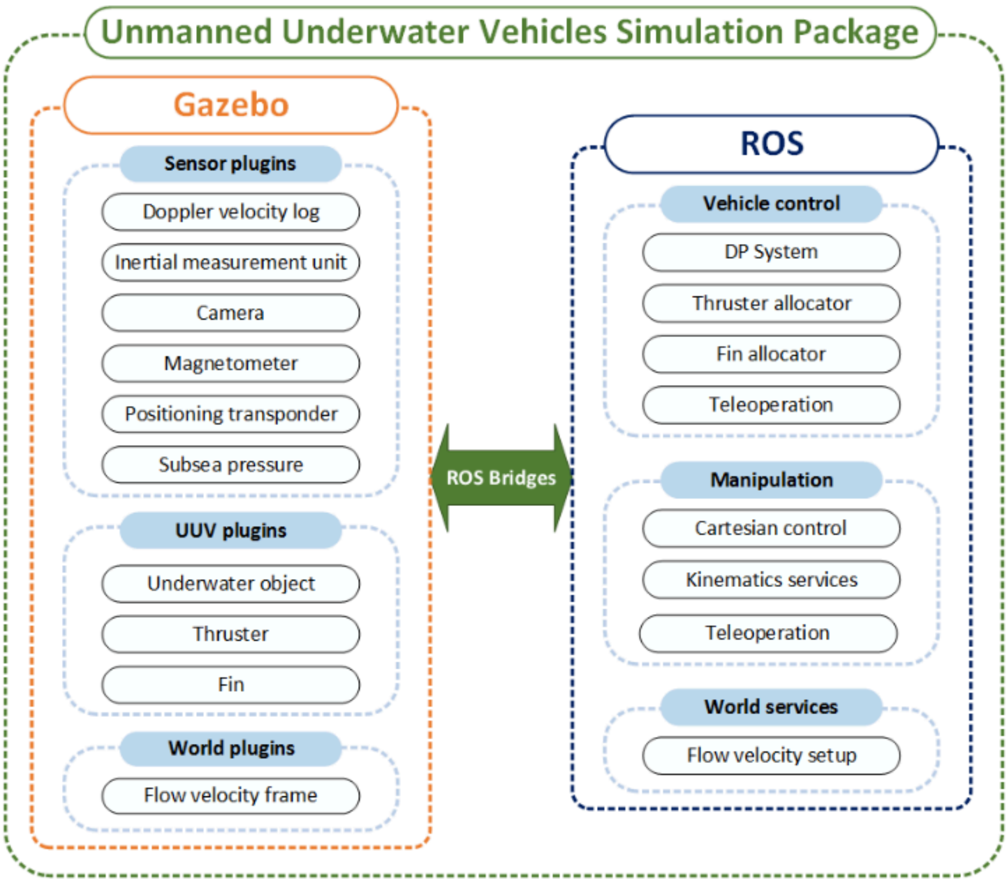

Sensor Input Messages
Lidar Laser Scan / msg
Inertial Measurement Unit
Doppler Velocity Log
Odometry / msg
Absolute Position Determination (Odometry / Msg)
Occupancy Grid / Msg
A* Msg
ROV Movement Control Msg

10. Simulation Environment:

The simulation was executed in Gazebo[13] using the Robot Operating System (ROS)[12], an open source robotics middleware. The UUV Simulator project that was developed under the EU SWARM [1] initiative was used to import an underwater environment that could accurately simulate the hydrodynamics. Gazebo simulator was evaluated as the best option since it provides a robust framework for simulation and visualization of robotics mechanisms, and can be extended for new dynamics, sensors and world models through modular plugins. These plugins can easily be interfaced with ROS which can help communicate between different modules such as actuators, sensors, planner, etc.

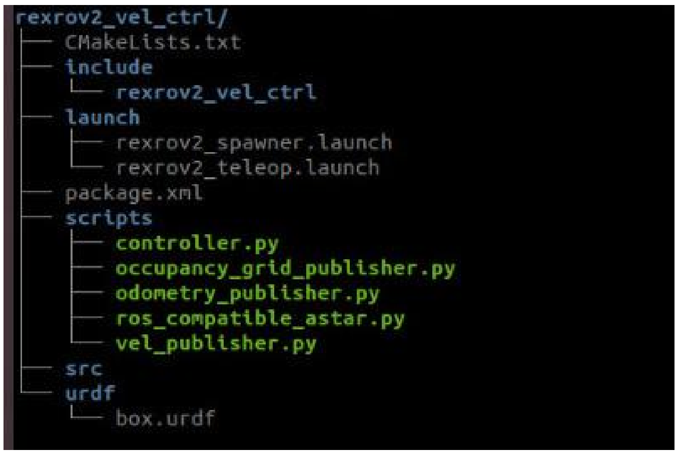
In ROS, each module runs independently as a ROS Node. They communicate with each other via ROS messages. These can be commonly used standard messages or can be custom made. UUV Simulator has defined custom messages that are suited to record sensor data. The nodes send and receive data via ROS Topics. Topics are essentially public message boards where messages of a certain type appear. A node publishes data to a Topic where another node can 'subscribe' to a topic to read the data. For instance in the UUV Simulator, sensors publish data to a topic and the Planning module or Odometry module can subscribe to that topic to get that information for further processing. ROS Services are used when a property of a node is required to be modified from outside or a response to a request is required. All this communication is facilitated by a ROS Master that keeps track of nodes and topics and establishes communication between them like a record keeper. Finally there are launch files[14] which is an XML file that tells ROS which nodes need to be launched and provides arguments for them. Launch files can run other launch files in a recursive manner.

UUV simulator was launched as a single package which was a combination of Gazebo and its plugins which were connected to ROS packages for planning and control via ROS Bridges, which are information pathways between ROS and non-ROS packages. Below figure[2] illustrates this architecture. More details can be found on the UUV Simulator documentation page[15] which are beyond the scope of this project.



UUV Simulator structure combining ROS and Gazebo

On top of UUV Simulator, we built a custom ROS package specifically for our objective and we divide the modules in their respective nodes, where each node works by using Modules and Topics created by the UUV Simulator package. Each node is restricted to performing a single function and only allowed to communicate via topics. Finally the launch file runs relevant nodes. Our package does not employ any Rosservices. Below is the file structure of our package.



```
rexrov2_vel_ctrl/
├── CMakeLists.txt
├── include
│   └── rexrov2_vel_ctrl
├── launch
│   ├── rexrov2_spawner.launch
│   └── rexrov2_teleop.launch
├── package.xml
├── scripts
│   ├── controller.py
│   ├── occupancy_grid_publisher.py
│   ├── odometry_publisher.py
│   ├── ros_compatible_astar.py
│   └── vel_publisher.py
├── src
└── urdf
    └── box.urdf
```

Custom Package file structure

The custom launch file imports other launch files which run nodes relevant to initializing the underwater environment, spawning the AUV, i.e. Rexrov2 in the environment. These two launch files run a number of nodes and activate a number of topics that are relevant to activating sensors, thrusters, corresponding physics engine, etc. While relevant to the working of the project, the following nodes that we created are mainly the scope of our project.

Overview of nodes:

- vel_publisher.py - Runs AUV in Teleop mode, receives keystrokes from users as commands, converts them into velocity input for manual control. Mainly for testing.
- odometry_publisher.py - Receives sensor data and converts that into odometry data, i.e. robot's movement with respect to its initial position.
- occupancy_grid_publisher.py - Runs the node that creates an Occupancy grid map to store obstacle locations by subscribing and processing data from the lidar scan topic.
- ros_compatible_astar_star.py - Run the nodes that plan a trajectory based on the current occupancy grid map state.
- controller.py - Runs a node that receives a planned trajectory, translates that into acceleration commands by means of a Cascaded PID controller.

Nodes are generally defined as a Class to avoid global variables that affect the scope of other functions and retain the ability to have multiple instances in case of a multi robot system.

odometry_Publisher.py: The function of this node is to receive data from the two sensors and convert it to odometry information by performing the coordinate transformation from robot frame to world frame as detailed in section <insert odometry section info>. The velocity information from the DVL sensor is subscribed from the 'rexrov2/dvl_twist' topic and the orientation information from the IMU is subscribed from the 'rexrov2/imu' topic. The update_odometry function updates the position of the robot with respect to the world frame at a 10Hz rate and publishes to a '/rexrov2/odom' topic for other nodes to use. Refer to Appendix I for the python script.

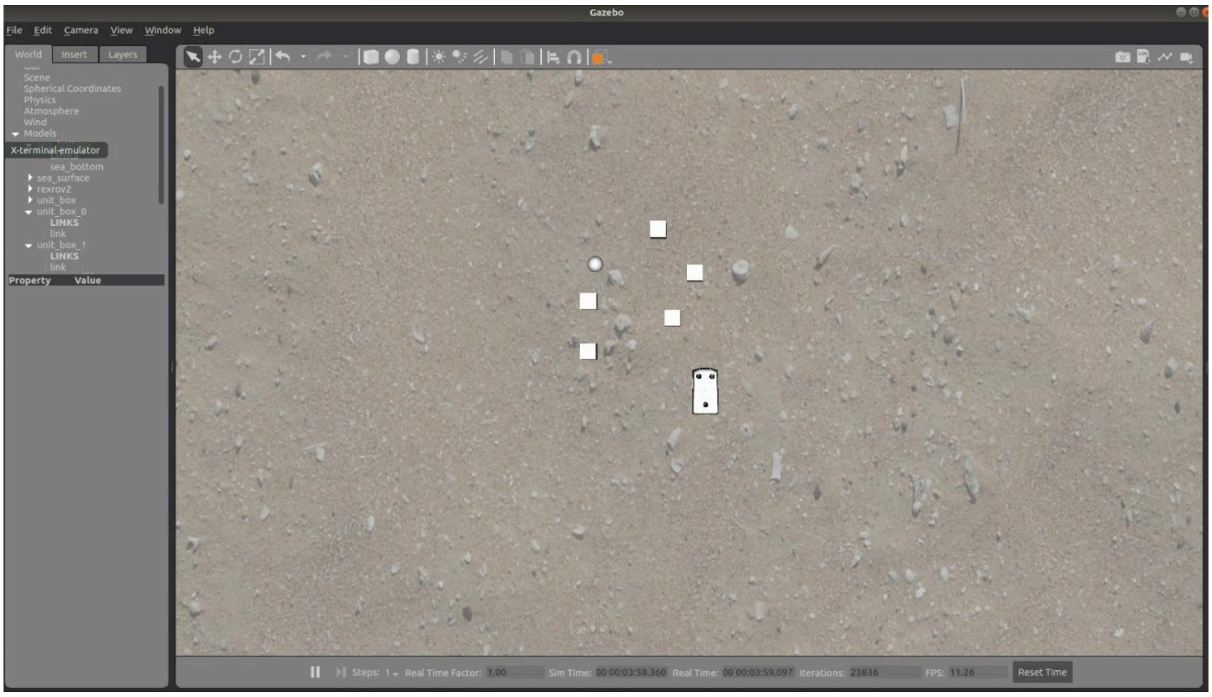
occupancy_grid_publisher.py: This node is responsible for storing the obstacles information in an Occupancy Grid based on the robots current pose estimate and the sensor data from the Lidar sensor. The lidar data arrives on the '/rexrov2/lidar_scan' topic and the robots current pose from '/rexrov2_odom'. The subscribed data is then processed as described in section <insert section> to update the corresponding cell of the grid with obstacle position. An Occupancy grid ROS msg is created and populated with the grid values. Then this msg is published to a '/map' topic for the planner node to use. The topic does not have a form '/{namespace}/map' to enable a single shared map in case of a multi-robot system. Refer to Appendix II for python script.

ros_compatible_astar_star.py: This is the planner node, that plans using the A* search algorithm. The node subscribes to 'rexrov2/odom' and '/map' to know its current pose estimate and current known world state respectively. This is passed as argument to the A* which outputs the shortest path from current position to target position. This path is of nested array form: [[x0, y0], [x1,y1], … , [xn,yn]]. The array is published to '/rexrov2/path' topic in a Float64MultiArray msg type.

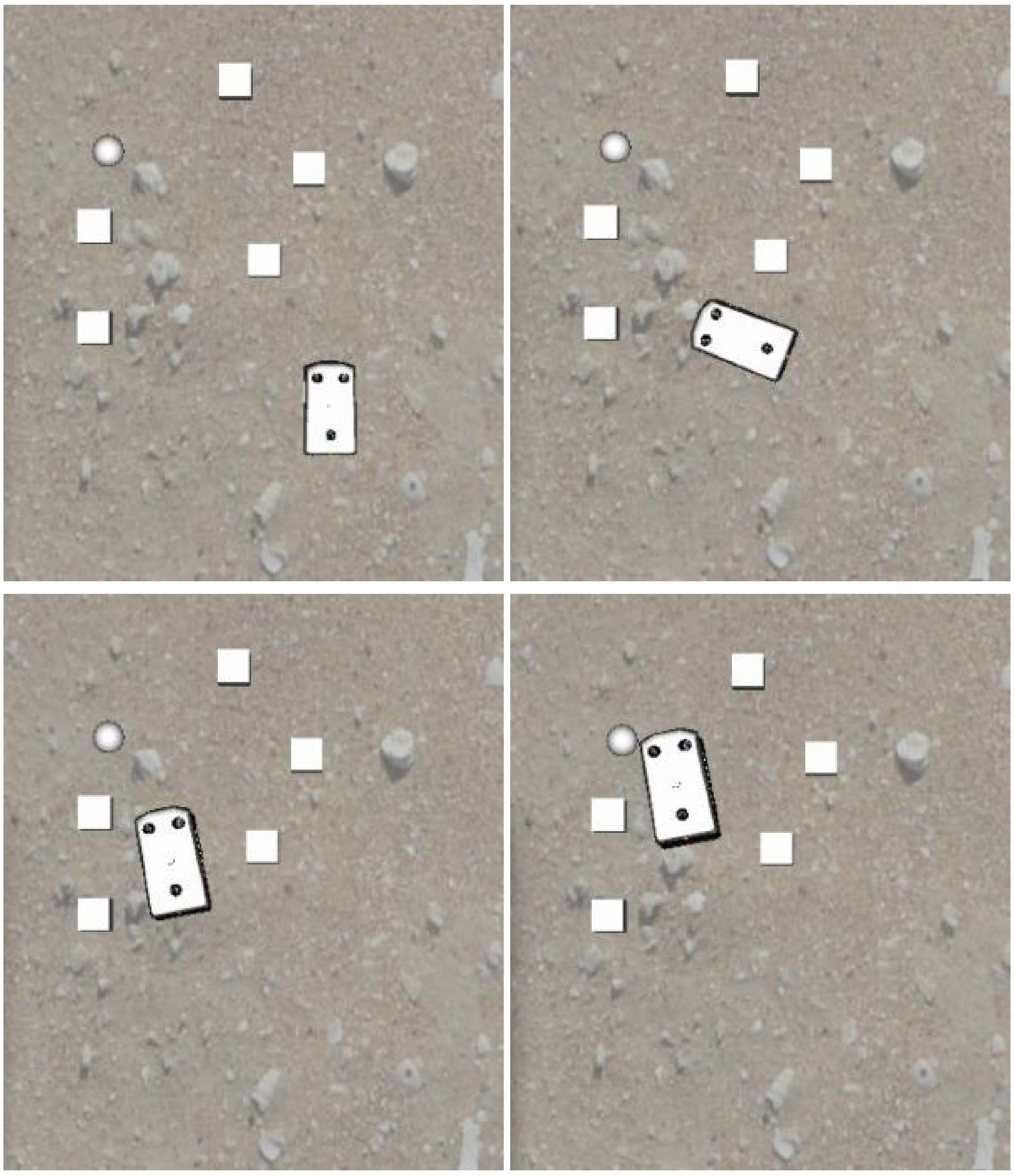
controller.py: The controller node subscribes the path from 'rexrov2/path' and smoothen the trajectory by CubicSpline interpolation. This smooth trajectory is then passed to a Cascaded PID controller, which is a PID controller nested within a PID controller. The position delta, i.e. (desired position - current position) is passed as input to the outer controller, which outputs a desired velocity. The inner PID controller takes velocity delta, i.e. (desired velocity - current velocity) as input and output desired acceleration. This value is then passed implicitly through the Thruster plugin module (part of the core UUV Simulator functions) which integrates the Fossen's underwater vehicle dynamics into the control input to give resultant thrust.

11. Results:

The simulation demonstrates the robots ability to navigate simple column obstacles to get to desired position. The PID controller manages to keep the AUV on a calculated path by countering lateral drift using control in the y-axis of the robot frame while maintaining the heading tangential to desired trajectory.



Initialized Gazebo Workspace


Iteration demonstrating successful traversal of the planned trajectory at consecutive timesteps

Successfully reaches target position autonomously

12. Future Work:

- The algorithm, however sound in theory, is harder to execute robustly due to hydrodynamic effects since it is a general purpose algorithm. The PID controller needs better tuning or a new controller needs to be designed from scratch that inherently models underwater dynamics to generate thrust.
- The system was tested in a very simplified environment due to the time constraints of the project and no real world cases were modelled such as corals or ocean currents as a function of time. Now that the environment has been initialized, the groundwork has been set for further testing of these scenarios in the future using PC hardware more suited for intense Gazebo simulation.
- Due to only using DVL and IMU for odometry, the absolute position of the AUV drifts over long missions and needs to surface to acquire GPS and course correction. This could be a possible feature or another sensor such as Multi-Beam sonar could be used to localize with respect to seabed.
- Imaging Sonar could also be used for effective obstacle avoidance over longer ranges. Multi Beam Sonar or scanning sonar might be a more effective system.
- The A* algorithm, although effective, works in a discretized space. There is potential for a modified A* or a different algorithm altogether which is better suited to continuous space such as Potential Field based algorithms.
- Multi-Robot SLAM: Initializing multiple robots, each running mapping and planning nodes could significantly converge on optimum plans due to distributed decision making.

Works Cited:

[1] SWARMs Project Website. http://www.swarms.eu/
[2] M. M. M. Manhães, S. A. Scherer, M. Voss, L. R. Douat, and T. Rauschenbach, “UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation,” *in Proc. IEEE/OES OCEANS Conference*, 2016, pp. 1–8.
[3] Handbook of Marine Craft Hydrodynamics and Motion Control. https://github.com/cybergalactic/FossenHandbook
[4] A. Rahman. “Understanding ROS Coordinate Frame (Part 1).” Accessed April 15, 2025 https://www.theconstruct.ai/ros-5-mins-023-understanding-ros-coordinate-frame-part-1/.
[5] J. Cooper. “Odometry.” Area 53 Robotics. Accessed May 9, 2025 https://area-53-robotics.github.io/Intro-To-Robotics/software/advanced-concepts/odometry/
[6]“Messages.” *ROS Wiki*. Accessed May 9, 2025. https://wiki.ros.org/Messages.
[7]“transformations — tf 1.11.9 Documentation.” *ROS Jade API Documentation*. Accessed May 9, 2025. https://docs.ros.org/en/jade/api/tf/html/python/transformations.html.
[8]uuvsimulator "*rexrov2_sensors.xacro*." GitHub. Accessed May 9, 2025. https://github.com/uuvsimulator/rexrov2/blob/master/rexrov2_description/urdf/rexrov2_sensors.xacro.
[9] Balasubramanian, S., Rajput, A., Hascaryo, R. W., Rastogi, C., & Norris, W. R. (n.d.). Comparison of dynamic and kinematic model-driven extended Kalman filters (EKF) for the localization of autonomous underwater vehicles. Department of Industrial and Enterprise Systems Engineering & Department of Aerospace Engineering, University of Illinois at Urbana-Champaign.
[10] “nav_msgs/Odometry Message.” *ROS Noetic API Documentation*. Accessed May 9, 2025. https://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html.
[11] “sensor_msgs/LaserScan Message.” *ROS Noetic API Documentation*. Accessed May 9, 2025. https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html.
[12] Quigley, Morgan & Conley, Ken & Gerkey, Brian & Faust, Josh & Foote, Tully & Leibs, Jeremy & Wheeler, Rob & Ng, Andrew. (2009). ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software.
[13] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), Sendai, Japan, 2004, pp. 2149-2154 vol.3, doi: 10.1109/IROS.2004.1389727.
[14] ROS Launch/XML format guide http://wiki.ros.org/roslaunch/XML
[15] UUV Simulator official documentation - https://uuvsimulator.github.io/
[16] terminxd. “How to convert laserscan measurements of ROS into values of Occupancy grid?” *ROS Answers (Q&A Forum)*. Posted April 24, 2023. https://answers.ros.org/question/414727/
[17] N. Swift. “Easy A*(star) Pathfinding”. Accessed April 20, 2025. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2\
[18] GeeksforGeeks. "Heap Queue (heapq) in Python." *GeeksforGeeks*, September 4, 2023. https://www.geeksforgeeks.org/heap-queue-or-heapq-in-python/.