

**ELEC330 Assignment 3**

**Biomimetic Robot Autonomous Navigation & Object Detection**

**By: Alvaro Mesa Giner (201656665), Gideon Tladi (201426759), Junyang Xiao (201769708), Yanzhang Wang (201608925), Yifan Wang (201677153), Zijin Wu (201769693)**

**Department of Electrical Engineering and Electronics**

**18 May 2025**

# Abstract

In this project, Butterflybot (originally named Nectar Wings), a biomimetic aerial robot, is developed and simulated. It is made to look like a butterfly while still having the stability of a quadcopter. Due to issues with dynamic stability and wing control, the project's original flapping-wing design was abandoned in favour of a propeller-based structure. The current design uses a quadcopter frame covered in a lightweight butterfly-shaped shell and is fully implemented in a Gazebo Harmonic simulation environment. The simulation environment is a carefully planned arena with obstacles and targets. The robot has a 2D LiDAR system, a bounding-box camera, and an Inertial Measurement Unit (IMU). The sensors are attached to the robot's main body (base\_link) and communicate with ROS 2 via the ros\_gz\_bridge, enabling perception, mapping, and navigation. Sensor fusion enables localisation, object detection, and environmental awareness by utilising Google Cartographer for SLAM and OpenCV for color-based recognition. The colour detection capability is implemented using HSV-based colour segmentation. Detection2DArray messages containing the detection results are sent to the /object\_detections topic. The quadcopter model was equipped with four thrusters for autonomous navigation, which were managed by ROS 2 launch files and Python-based nodes. Despite integration and time constraints impeding the realisation of fully autonomous navigation, significant progress was made. After object detection protocols were implemented, the robot was able to successfully identify and recognise colour-coded targets in the simulation. The foundation for later work was established by early attempts at autonomous control using the ArduPilot and Gazebo thruster plugins. The overall structure facilitates modular development and creates a strong basis for further iteration and eventual implementation in practical settings.

**Declaration**

By submitting this work, we confirm that we have read and understood the University’s Academic Integrity Policy and the definitions of plagiarism and collusion as outlined in the Code of Practice on Assessment (Appendix L). We confirm that we have acted honestly, ethically, and professionally in conduct leading to assessment for the programme of study.

We declare that the work we are submitting is entirely our own. We have not committed plagiarism, colluded with any other party, or commissioned any third party to produce this work. We have not copied material from any person or source, nor fabricated, falsified, or embellished data. Where we have quoted or used figures from published or unpublished sources (including web-based content), we have clearly indicated and acknowledged them.

We confirm that we have not used artificial intelligence software in an unacceptable manner to generate this work\*. We understand the consequences of engaging in plagiarism, collusion, or any form of academic misconduct.

\*Software applications include, but are not limited to, ChatGPT, Bing Chat, DALL·E, and Gemini.

Contents

[1 Introduction 4](#_Toc198490797)

[1.1 Background Information 4](#_Toc198490798)

[1.2 Project Objectives 5](#_Toc198490799)

[2 Design and Methodology 5](#_Toc198490800)

[2.1 Overview 5](#_Toc198490801)

[2.2 Sensor Integration and Architecture 8](#_Toc198490802)

[2.3 Object Detection 9](#_Toc198490803)

[2.4 Mapping 11](#_Toc198490804)

[2.5 Autonomous Navigation 12](#_Toc198490805)

[3 Testing Scenarios and Results 13](#_Toc198490806)

[3.1 Object Detection 13](#_Toc198490807)

[3.2 Mapping 15](#_Toc198490808)

[3.3 Autonomous Navigation 18](#_Toc198490809)

[4 Discussion and Reflection 20](#_Toc198490810)

[5 Bill of Materials (BOM) 21](#_Toc198490811)

[6 Conclusions 22](#_Toc198490812)

[References 24](#_Toc198490813)

[Appendices 25](#_Toc198490814)

[Team Roles, Contributions, and Reflections 25](#_Toc198490815)

[Project Files and Code 32](#_Toc198490816)

# 1 Introduction

## 1.1 Background Information

This project constructs a high-fidelity virtual testbed for a butterfly-shaped bio-inspired aerial robot named butterflybot by seamlessly integrating ROS 2 middleware with the Gazebo Harmonic simulator. It aims to achieve inherently silent, ultra-manoeuvrable flight capabilities suitable for indoor inspection, ecological monitoring, greenhouse pollination, and confined-space search-and-rescue while avoiding the cost, fragility, and development delays associated with early hardware prototypes.

Real butterfly flapping delivers exceptional lift and near-silent acoustics through complex vortex shedding, clap-and-fling interactions, and tight body–wing coupling that defy conventional modelling and real-time control. After encountering early instability in attempting to replicate true flapping mechanics, the project team adopted a pragmatic compromise of re-designing the robot by encasing a quadrotor within a lightweight butterfly fuselage with flexible wings whose outward outline preserves biological lift, natural camouflage, and low wing loading, while proven four-rotor dynamics guarantee baseline stability and controllability.

The unified simulation stack couples Gazebo Harmonic’s rigid-body physics engine and a quasi-steady lift-drag plugin for the wings with noise-injected simulated cameras, LiDAR, and IMU sensors to emulate real-world uncertainty, all orchestrated by ROS 2 nodes; the Nav2 stack utilises an occupancy grid map generated by Cartographer for real-time localisation and mapping, initially supporting 2D (and, in future developments, 3D) SLAM through scan-matching and pose graph optimisation; OpenCV pipelines detect coloured fiducials so the robot can “forage” for virtual flowers; and RViz renders live point clouds, cost-maps, planned trajectories, and sensor streams, dramatically accelerating parameter tuning and system debugging.

Despite this comprehensive framework, several formidable challenges remain, including balancing aerodynamic fidelity against the real-time requirements of the simulator—where simplified force models must nonetheless reproduce lift coefficients and dynamic behaviours documented in Delfly [1], RoboBee [2], and KUBeetle [3] studies—ensuring control robustness in the face of altered inertia and airflow induced by protruding wings (necessitating adaptive, sliding-mode, or neural augmentation beyond traditional PID gains), achieving perception reliability under simulated lighting variance and sensor noise (mitigated via domain randomisation and disturbance injection to narrow the sim-to-real gap), enabling true 3D obstacle avoidance in environments with floor, ceiling, and lateral constraints (requiring fast replanning and fail-safe mechanisms), and planning for future real-world deployment on embedded PX4 hardware with limited compute resources, which demands that every algorithm be profiled and optimised for onboard, real-time performance.

Deliverables for this endeavour include an open-source URDF/SDF model of the butterfly quadrotor, ROS 2 launch files and parameter configurations, a Docker-based continuous integration pipeline, and a hardware-in-the-loop bridge to facilitate seamless migration from simulation to real-world testing, thereby providing the robotics community with an accessible, extensible sandbox for rapid iteration that accelerates the day when a silent, agile “mechanical butterfly” will flutter beyond the screen into physical reality.

## 1.2 Project Objectives

This project is continuing the work on an autonomous butterfly robot first looked at in a prior assignment. Originally intended with a biologically inspired flapping-wing design, the robot was first navigated within a simulated environment during the earlier phase using preliminary efforts. Still, the simple control problems at this point were caused by the lightweight, unstable design and the complicated wing actuation. These limitations prompted a complete redesign of the robot model for the present project.   
  
The modified design moves from rigorous biomimicry to a quadrotor-style structure to improve simulation stability and controlability. While keeping decorative butterfly wings to preserve visual continuity, the focus is shifted to the creation of a practical and testable platform. Defined in URDF and SDF format, the new model is deployed in Gazebo Harmonic using ROS 2 Jazzy, therefore allowing integration with control nodes and sensor systems. As described in the project reports for Assignments 1 and 2, this initiative is focused on enhancing the navigation and perception abilities of the robot. It draws on earlier efforts. Object recognition and environmental awareness are made possible by the robot model's bounding box camera, IMU, and 2D LiDAR. A ros\_gz\_bridge node links these sensors to ROS 2, therefore enabling the system to handle real-time data and interact with its surroundings.

The main objective is to confirm the robot's ability to run steadily in a simulated environment. Using sensor input, the robot is expected to find a target item in the surroundings and move towards it. By simplifying the mechanical design and broadening perception and control, this project aims to build a more robust basis for autonomous behaviour in simulation.

# 2 Design and Methodology

## 2.1 Overview

This project sets out to build a “butterfly” UAV with multirotor performance. The initial concept used two servos to flap the fore-wings up-and-down, replicating the thoracic flapping kinematics of real butterflies.

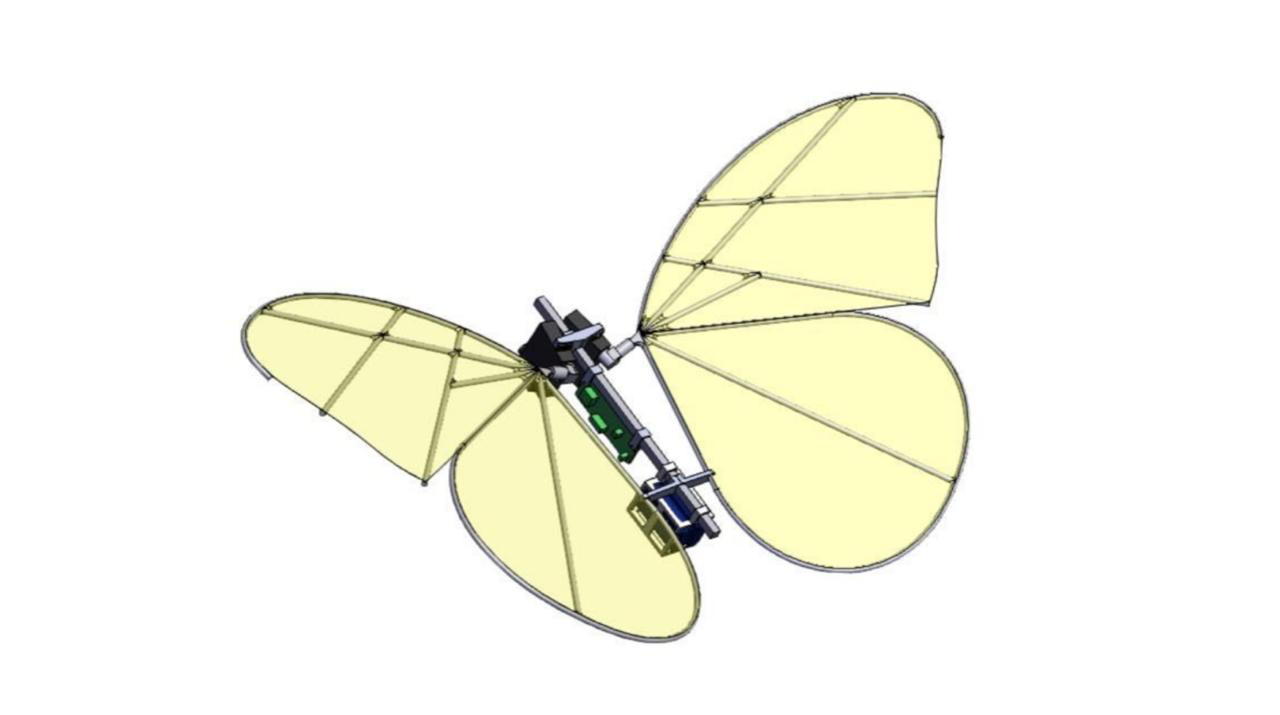


Figure 2.1.1: Previous Design

Studies show that butterflies harvest unsteady leading-edge vortices for transient lift and leverage abdominal motion for attitude regulation [4]. Such coupled biomechanics are hard to emulate mechanically, leading to highly unstable flight and insufficient lift.

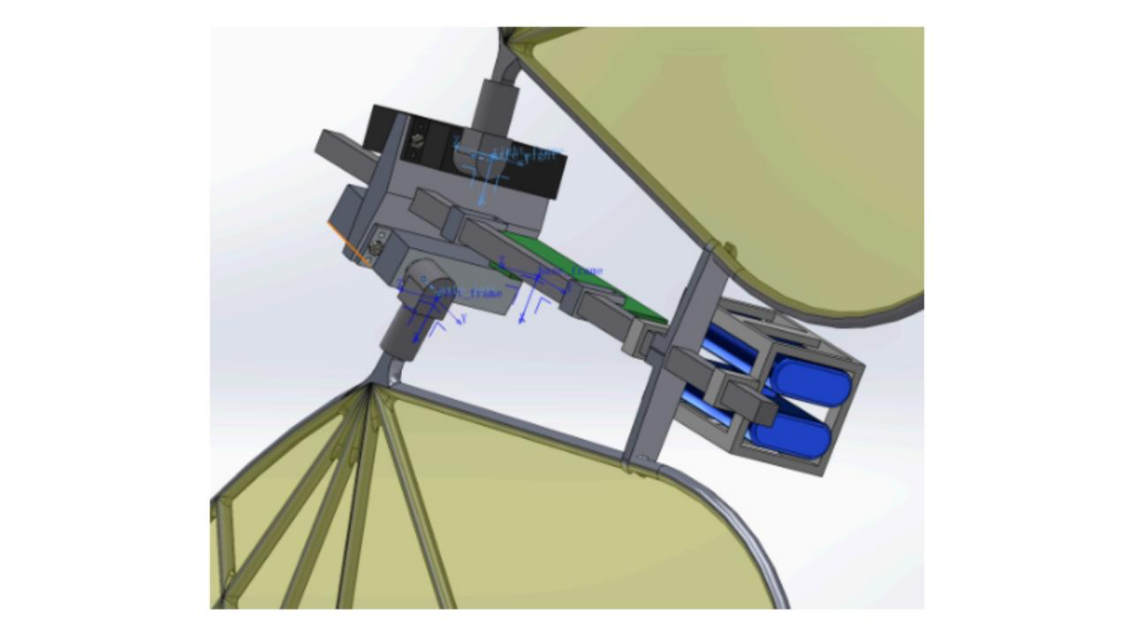


Figure 2.1.2: Two servos that emulate the contraction of a butterfly’s thoracic muscles actuate the wings.

Accordingly, we rebuilt the airframe for this stage: the “∞”-shaped wings are locked horizontally, with a motor-propeller unit in each cell, rendering the craft a symmetric quad-rotor in principle.

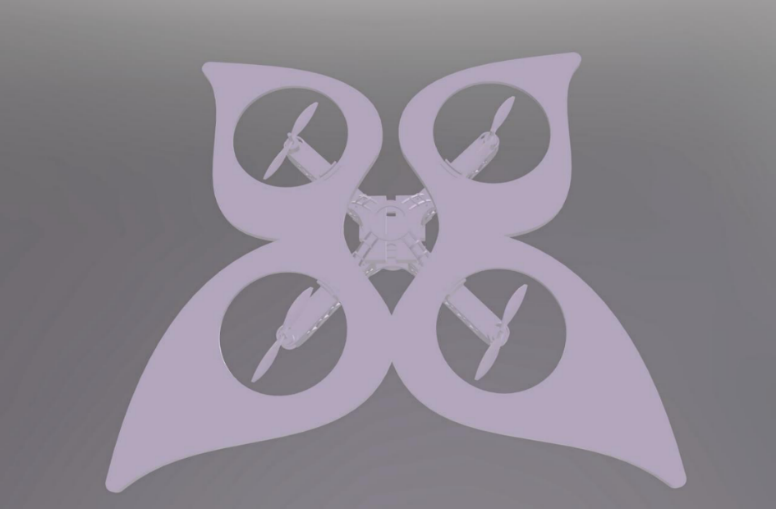


Figure 2.1.3: New design with a motor-propeller

The symmetric layout co-locates the mass and geometric centres. The frame features a perforated honeycomb topology, cutting mass while boosting specific stiffness.

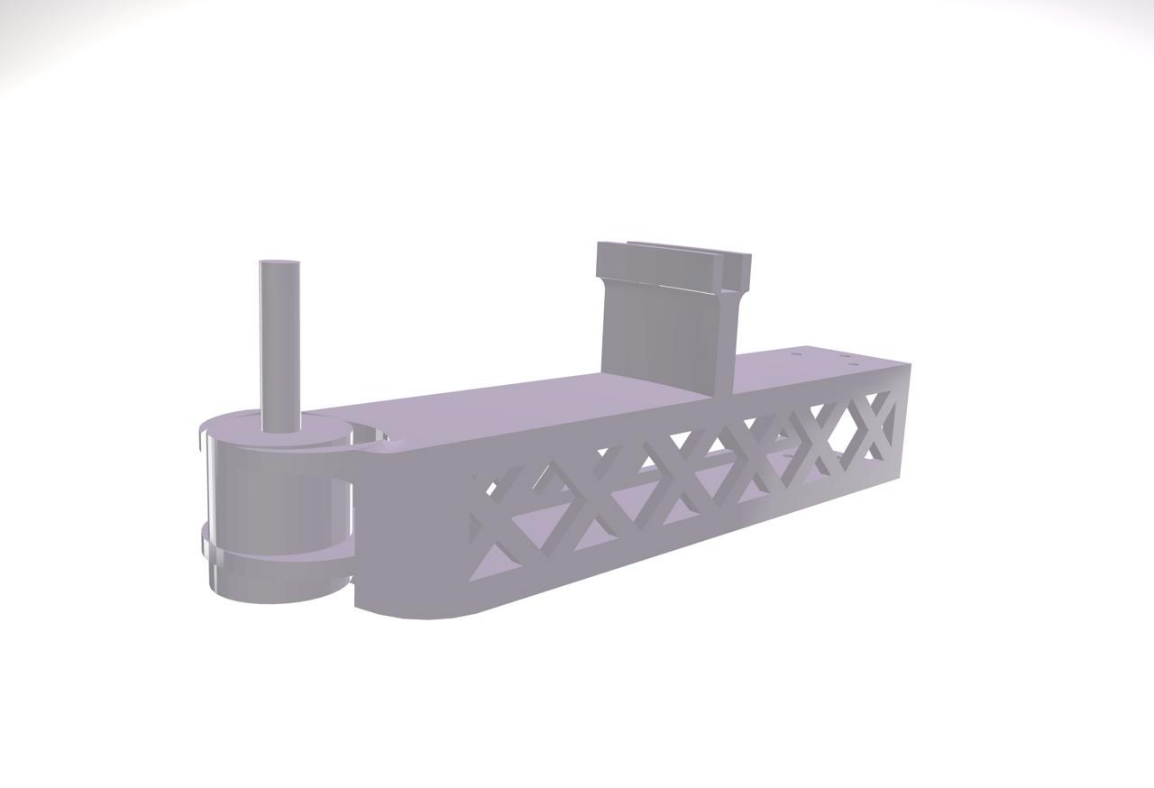


Figure 2.1.4: Perforated (lightweight) propeller mount.

Smooth spherical fillets connect the “∞” wing rims to the fuselage, lowering drag and acting as a built-in crash bumper.

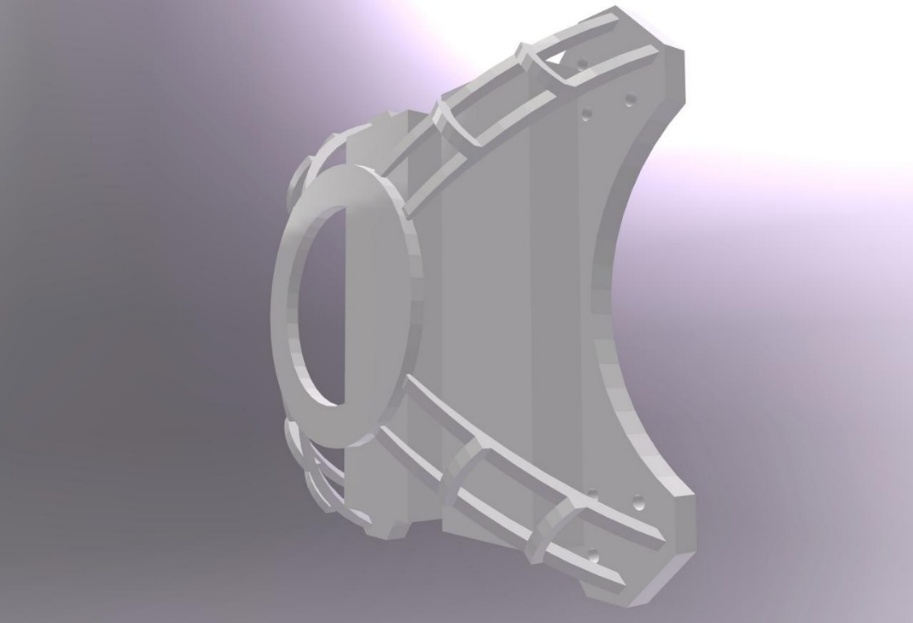


Figure 2.1.5: Hemispherical perforated battery guard/cage.

A hemispherical guard encloses the battery and board; arched ribs on the dome raise impact resistance.

## 2.2 Sensor Integration and Architecture

The sensor system in this project is set up entirely within the robot’s SDF model. It is intended to equip the robot with the ability to perceive its surroundings and support autonomous interaction with the environment. All the sensors are mounted on the main body of the robot, known as the “base\_footprint”. This setup makes it easier to manage each sensor’s position and orientation. Using multiple sensors in combination is generally more effective for SLAM than relying on a single type of sensor, as it improves robustness and compensates for individual limitations [5].

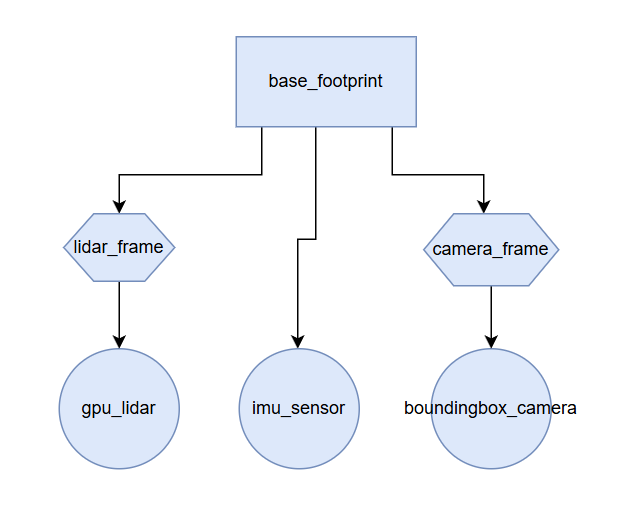


Fig 2.2: The structure of the sensor system

As shown in the diagram above, this project makes use of three types of sensors: a LiDAR, an IMU, and a camera. All of them are mounted directly on the main body link, “base\_link”, with their own frame in the robot’s SDF model. This setup ensures that each sensor maintains a fixed and clearly defined position relative to their own frame [6]. Their own frames are mounted with the main body link. This structure is more efficient for us to modify the direction of the sensor and debug the errors.

The Lidar sensor is mounted on a frame named “lidar\_frame”, which is positioned slightly above the robot and rotated -1.57 radians around the vertical axis to face forward. It simulates horizontal laser scans across a 160-degree field of view with 640 samples per scan. The sensor is configured to operate at 10 Hz and detects obstacles within a range of 0.08 to 10 meters. In the launch file, this sensor’s output is bridged to ROS 2 using the “ros\_gz\_bridge” node via the topic “/lidar\_scan”. Once bridged, the data becomes available to other ROS 2 components, such as navigation modules or mapping frameworks.

The “camera\_frame” is located behind the base and similarly rotated to face forward. It is used to define the position of the camera sensor. The camera simulates object detection output and is configured to produce images at a resolution of 800 by 600 pixels with a horizontal field of view of approximately 60 degrees. It is updated at 10 Hz and provides object bounding boxes rather than raw image frames. In the ROS 2 interface, the output is published on the “/boxes\_image” topic and used by a custom node called “object\_detection\_node”. This node is launched within the same launch file and is expected to process the incoming camera data to identify specific targets placed in the environment.

In addition to LiDAR and camera sensors, an inertial measurement unit (IMU) is embedded directly within the base link. This sensor provides angular velocity and linear acceleration data on all three axes, incorporating Gaussian noise for realism. The IMU operates at 100 Hz and is used to support orientation tracking and potentially motion estimation. Its output is bridged to the ROS 2 topic /imu/data through the same “parameter\_bridge” node. Although not yet integrated with state estimation frameworks in the current implementation, the IMU data could be used in the future for sensor fusion and improved localisation.

All sensor data is transmitted from Gazebo to ROS 2 via the “ros\_gz\_bridge::parameter\_bridge” node, which is explicitly configured in the launch file with a list of topic mappings. These include “/lidar\_scan, /boxes\_image”, and /imu/data, each paired with their corresponding ROS 2 message types. The launch file also initialises a “transform\_broadcaster\_node”, which is responsible for maintaining the transform relationships between sensor frames and the robot base. Although the system currently assumes static transforms defined in the SDF, this node ensures that TF data remains accessible to ROS visualisation tools such as RViz. In addition, the launch file includes a separate node for visualising the system in RViz with a pre-configured “.rviz” layout file, allowing real-time monitoring of sensor data streams and robot position.

## 2.3 Object Detection

图示

AI 生成的内容可能不正确。

Figure 2.3: Object Detection Node Flow Chart

As shown in Fig. 2.3, this algorithm realises an object recognition function based on ROS2 and OpenCV. When it starts, it performs node initialisation and sets up three key communication channels. The first one is creating a new subscriber named “/boxes\_image” which can listen to the “box\_camera” topic to receive the image to be processed. As for the other two publishers respectively send the detection results to “/object\_detections” publisher and the annotated image to “/annotated\_image” publisher. After the initialisation is completed, the node enters a loop waiting state, ready to process the incoming image data.

After the node initialises, the system reaches the “rclpy.spin(node)” component. This function is important to the ROS2 operation which can create the continuous execution loop that keeps the node running. When the experimenters call this node, it will activate the node's event handling system and listen continuously to the incoming messages.

When the image message is received, the system firstly converts the ROS format image into the OpenCV format for processing because the ROS2 uses specific message types (such as “sensor\_msgs/Image”) to represent and transmit image data, but for OpenCV, it uses the Mat file, which is a matrix data structure optimised for computer vision algorithms. Moreover, the core recognition strategy is based on colour segmentation in the HSV (hue, saturation and value) colour space. Compared to the RGB-based recognition, HSV-based recognition technology can separate colour information from brightness, making detection more robust under different lighting. [7] Furthermore, the author specifically sets range for two colours: green (HSV values 40-80, 50-255, 50-255) and blue (HSV values 100-130, 50-255, 50-255).

More specifically, the system will evaluate each pixel on the figures and estimate if its HSV value falls within the specified threshold range set above. For the corresponding mask pixel, they will be set to 255 (white), otherwise pixels will be set to 0 (black) to form a binary mask image. Next, for these flawed pixels, the system will take morphological operations such as dilation and erosion to reduce noise and fill small holes. Finally, the system will use contour detection algorithm to identify the boundaries of enclosed regions on the generated binary mask images and classify the objects based on contour characteristics.

For each identified object, the algorithm will firstly calculate the bounding box of the court and extract the position and size information. Next, the system will draw the bounding box and corresponding category label on the original position. These pieces of information are encapsulated into the standard Detection2D message format to be integrated with other components in the ROS2 workspace. Finally, after processing all the detected objects, the system will package the results as a Detection2DArray type message and publish it to the “/object\_detections” topic. At the same time, it will also publish the image with bounding boxes and labels to the “/annotated\_image” topic.

## 2.4 Mapping

Implementing Simultaneous Localisation and Mapping (SLAM) is vital for autonomous aerial robots operating in GPS-denied or indoor environments. Achieving accurate environmental awareness without compromising flight dynamics is a significant challenge for a butterfly-inspired drone, where manoeuvrability and lightness are prioritised. This section outlines how Google Cartographer [8] and a lightweight 2D LiDAR enabled real-time mapping and localisation for autonomous navigation.

Cartographer is an open-source, real-time SLAM library developed by Google that supports 2D and 3D SLAM with multiple sensor fusion capabilities, including LiDAR, IMU, and optional odometry. It is well-suited for structured indoor environments and some simple outdoor environments, where a butterfly-shaped drone can operate in a controlled manner due to aerodynamic constraints. Like traditional rotor drones, this autonomous robot design requires precise stabilisation, making reliable localisation and mapping a necessity rather than an enhancement. Cartographer's submap-based SLAM approach and loop closure capability minimise positional drift over time. This makes Cartographer ideal for a lightweight aerial platform that lacks GPS and wheel odometry, as it relies primarily on LiDAR and IMU, both of which can be mounted without adding excessive weight.

In simulation, *butterflybot* utilises a miniature 2D LiDAR scanner via the Gazebo gpu\_lidar sensor package [9]. The sensor is fitted underneath the base\_link of the robot’s body, facing forward whilst being parallel to the base\_link and ground\_plane. This position gives the 2D sensor an optimal angle to capture planar scans of the environment. Additionally, an IMU module was integrated on the onboard flight controller to provide orientation and acceleration data.

Sensor data were published on appropriate ROS2 topics:

* *“/lidar\_scan”* (remapped to */scan*) for LiDAR data.
* *“/imu/data”* for orientation and linear acceleration.

Transformations between the LiDAR, IMU, and drone body frame were handled using ROS2 tf2, ensuring consistent spatial alignment of sensor inputs. Static transform nodes were added to the launch file, while dynamic transform nodes were implemented in a custom transform\_broadcaster.py file. Furthermore, Cartographer configuration files (butterflybot\_2d.lua) were tuned for low-latency scan matching, smaller submap resolution (e.g., 0.05 m/cell), and relatively high IMU weight due to the drone’s frequent body oscillations from balancing the thrust forces of the four propellers. Similarly, a 3D version of the .lua file was also coded, for future point-cloud sensing implementation. Finally, a cartographer\_ros node was added to the launch.py file, making use of the 2D .lua file to generate a 2D occupancy grid map in Rviz2.

## 2.5 Autonomous Navigation

This project originally aimed to create a robot that mimicked the flight dynamics and aerodynamics of a butterfly by using intricate articulated wings. However, upon discussing the idea with the supervisor, the team recognised that the original design was beyond the scope of what could be achieved within the allocated time. As a result, a quadcopter-based system was adopted, reducing the overall complexity of the mechanical, electrical and control systems while still achieving the goals of autonomous navigation, object detection and mapping.

Initially, simple manual simulations were implemented, after which professional drone-control systems like ArduPilot were tried. Eventually, the team settled on using Gazebo’s thruster plugins for a simpler propulsion solution. Autonomous navigation could not be fully achieved due to the challenges posed by computational issues, time constraints and integration difficulties.

**Manual Navigation Attempts for Initial Testing**

The first stage of testing consisted of applying simulated forces to a simplified quadcopter model within the Gazebo environment. It verified fundamental concepts in drone physics such as mass distribution, propulsion dynamics and the ability to steer. As a result, there was a need to develop more efficient and reliable methods for testing quadcopter dynamics.

**ArduPilot Integration and Challenges**

A minor breakthrough was achieved by combining ArduPilot, a popular autonomous drone control software, with Gazebo simulation environment. ArduPilot’s proper operation was initially confirmed by running it with a basic drone SDF model in an empty Gazebo environment. This configuration confirmed that the drone model was communicating with ArduPilot and responding to commands such as component arming.

Connecting ArduPilot plugins to the butterfly quadcopter’s detailed SDF model proved to be a complex process. The university computers often overheat which negatively impacted the stability and performance of the simulations. Furthermore, the drone experienced inconsistencies when following ArduPilot’s autonomous commands in Gazebo. Unfortunately, ongoing issues with both hardware and software prevented the team from successfully integrating ArduPilot plugins into the quadcopter’s SDF model.

**Thruster-Based Propulsion Approach**

Given the challenges with ArduPilot, an alternative solution was to use Gazebo’s built-in thruster plugins. Four separate thruster modules were added to the quadcopter model, placing them at the centre of each rotor. This setup provided the necessary infrastructure to develop autonomous flight nodes and path-following functionality in the future.

The idea was to develop ROS2 nodes to automatically manage the thrusters and implement various predefined autonomous flight patterns. Testing predefined flight paths created using ROS2 launch files and Python-based nodes would provide a reliable way to evaluate the quadcopter’s autonomous capabilities. Unfortunately, these ROS2 autonomous flight nodes could not be fully implemented in time for the project’s bench inspection and final report submission. The final version of the code features topics for each propellor joint in the butterflybot.sdf and launch.py files, but the robot has unstable and erratic flight behaviour, likely due to issues with properly bridging and transforming the relevant propellor topic data.

**Intended Autonomous Navigation Strategy**

Had development continued, the full autonomous navigation solution would have included:

* **Path Planning**: Utilising the A\* algorithm (via ROS2 Nav2) for generating optimal collision-free flight paths within predefined maps.
* **Obstacle Avoidance**: Real-time obstacle avoidance using a combination of LiDAR sensor input and IMU-based orientation data, integrated through Nav2’s Dynamic Window Approach (DWA) or Timed Elastic Band (TEB) planners.
* **Flight and Precision Landing**: Employing the color-recognition algorithm developed by teammates (OpenCV-based, utilizing functions such as cv2.cvtColor, cv2.inRange, and cv2.findContours, along with supporting packages such as numpy and cv\_bridge), enabling vision-guided landing. This would allow the drone to autonomously detect and precisely land on clearly defined landing markers, supported by altitude measurements (LiDAR) and orientation stabilisation (IMU).

The simplified thruster-based control approach represented a highly practical and achievable baseline from which further autonomous navigation capabilities could have been readily expanded, integrating seamlessly with the object detection and mapping capabilities developed by team members.

# 3 Testing Scenarios and Results

## 3.1 Object Detection

As shown in Fig, the output result is shown below. The image displays an implemented colour detection application, which is capable of identifying different colour areas in cat sample photo. The output has detected two colour areas, mainly identifying a "red" area (near the nose) and multiple "orange" areas (distributed on the paws and other parts, which were some errors in the HSV parameters, and adjustments were made subsequently). Each detected colour is highlighted with a green bounding box and labelled with the corresponding colour name.

The algorithm determines the colour name by calculating the minimum absolute distance between the HSV value of the selected pixel and the colours in the database. This is an effective colour matching method. When the user double-clicks on the image, the program captures the HSV value of that point, then searches for the closest colour name in the database and displays the result on the interface.

猫躺在电脑旁

AI 生成的内容可能不正确。

Figure 3.1.1: Colour detection

As for the actual application in the ROS2 shown in Fig 3.1.1, the experimental results show that the object detection algorithm based on the Gazebo simulation environment can successfully identify the red cube, blue cube, green sphere, and traffic cone in the environment. As shown in the right figure, the detection results are displayed in a green border format, and the algorithm can effectively distinguish different shapes and colours of objects and accurately locate their positions in the scene.

However, the current system still has several disadvantages. The detection function is mainly limited to simple objects with obvious colour contrast. It may face recognition challenges for complex shapes or objects with similar colours, which means the accuracy of the detection algorithm also needs to be improved, especially for non-rectangular objects such as spheres, where the rectangular border generated by the system cannot closely fit the actual contour of the object. Additionally, the system's performance in handling complex environments, partial object occlusion, or lighting changes conditions has not been fully tested, which are common situations in practical applications.

Future improvement work can focus on several key aspects. For example, introducing advanced deep learning models such as YOLO or CNN (convolutional neural network) can improve detection accuracy and enhance the system's robust in complex environments. Improving the boundary box generation algorithm can enable the system to generate detection results that are closer to the actual contour of the object, rather than just rectangular boxes. Last but not least, adding real-time tracking function is also crucial for handling moving objects in dynamic environment.

图形用户界面, 应用程序

AI 生成的内容可能不正确。Figure 3.1.1: Object detection in an empty world loaded with coloured objects for testing.

## 3.2 Mapping

As physical testing of the butterfly-inspired drone was not conducted in this phase of the project, the performance and validation of the mapping system were carried out entirely within a simulated environment. The team evaluated how a virtual, four-propeller aerial vehicle equipped with LiDAR and IMU sensors would map and localise itself in a structured indoor space. This section presents the methodology, configuration, and observed results from a series of mapping simulations.

**Simulation Setup**

Simulations were conducted in a custom-built Gazebo world (env\_ws.sdf) designed in previous phases of the project (Assignments 1 and 2). The simulated arena measured approximately 15 m (L) × 15 m (W) by 9 m (H, relative to the ground plane) and featured a static wall, a slope, an arch, spheres, rocks, trees, and a flower. The world also has open vertical space to allow for flight, and the transparent boundary walls have collision enabled to prevent the butterfly from flying out of the world.

The objective was to assess how well Cartographer could process virtual sensor data in real time and produce accurate 2D maps suitable for autonomous navigation.

A simplified URDF model of the **butterfly drone** was loaded onto Rviz using the RobotModel and TF displays to feature the robot body and virtual joints to represent propeller rotation. A joint\_state\_publisher\_gui plugin was also utilised. Refer to the video linked below:

<https://drive.google.com/file/d/1ocXCF32CBfPMEmBc18trog47MhSQGAIW/view?usp=sharing>

**Virtual Sensor Configuration**

To simulate the mapping capabilities of a real system, the virtual drone was equipped with:

* A **2D gpu\_lidar plugin,** with a 160° field of view and 0.08 m - 10 m range.
* A simulated **IMU sensor**, providing linear acceleration and orientation data with optional Gaussian noise.
* ROS 2 topics: /lidar\_scan, /imu/data, and TF frames.

Sensor output was published at realistic frequencies (LiDAR at 10 Hz, IMU at 100 Hz), and noise parameters were tuned to simulate the inaccuracies of a lightweight real-world sensor suite.

**Methodology**

The simulation tests were divided into three core stages:

1. **Static Mapping Validation**: The "<static>" variable of the butterflybot.sdf was set to "true" to cause the robot to load in a static state. Upon starting the simulation, the robot was manually translated to different locations within the map while the simulated LiDAR scanned the environment.
2. **Dynamic Mapping Validation**: The "<static>" variable of the butterflybot.sdf was set to "false" to cause the robot to load in a dynamic state. In an ideal scenario, the robot would hover at a fixed position above ground, glide towards each corner of the environment, and ultimately produce a complete map.

In both cases 1 and 2, the aim was to get the drone to revisit previously mapped areas to trigger loop closure mechanisms. All mapping outputs were visualised in **RViz** and saved via ros2 bag for post-processing. The cartographer\_ros configuration files were adjusted for simulation, emphasising scan matching and submap resolution while reducing reliance on odometry (which was excluded to mimic real drone constraints).

Key parameters included:

* **Submap resolution:** 0.05 m
* **Map frame update frequency:** 10 Hz
* **Base-to-lidar and base-to-IMU transforms:** In the launch.py file, "robot1/lidar\_link/gpu\_lidar" and "robot1/imu\_link/imu\_sensor" were the static transforms implemented to enable mapping.
* **Lua configuration file:** Used "robot1/imu\_link/imu\_sensor" as the tracking frame and published to the "base\_link" frame.

**Results and Observations**

The Cartographer SLAM algorithm showed promise under simulation conditions but was ultimately let down by the drone's flight instability.

During **static mapping tests**, the virtual LiDAR produced slightly cleaner range data. However, making the drone a static object ultimately caused submap data to overlap at different points on the map. Figure 3.2.1 shows the map visualised in Rviz after manually translating the robot across the entire world along the same horizontal plane, turning the robot by 90° at each corner. While the robot could initially map out some corners, it ultimately failed to perform loop closure in this stage.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3.2.1: Result of the static mapping test.

The following video shows the static mapping attempt:

<https://drive.google.com/file/d/1dAxJ9PYrZ7ZLPQJ5zp14kh23v09ANr_F/view?usp=drive_link>

During **dynamic mapping tests**, the virtual LiDAR produced noisier range data, as visualised in Figure 3.2.2. The leading cause was the robot's instability during flight, as it kept spinning around rapidly, leading to multiple inconsistencies and overlapping submap outputs. However, once the robot stopped after crashing into an object, the result seemed to improve, as the map accurately showed the transparent boundary, the wall ahead, and the corner of the arch next to it. Refer to Figure 3.2.3.

The following video shows the dynamic mapping attempt:

<https://drive.google.com/file/d/18LCBP274TzHw8v7bzhcZ2ZW48zG51zj6/view?usp=drive_link>

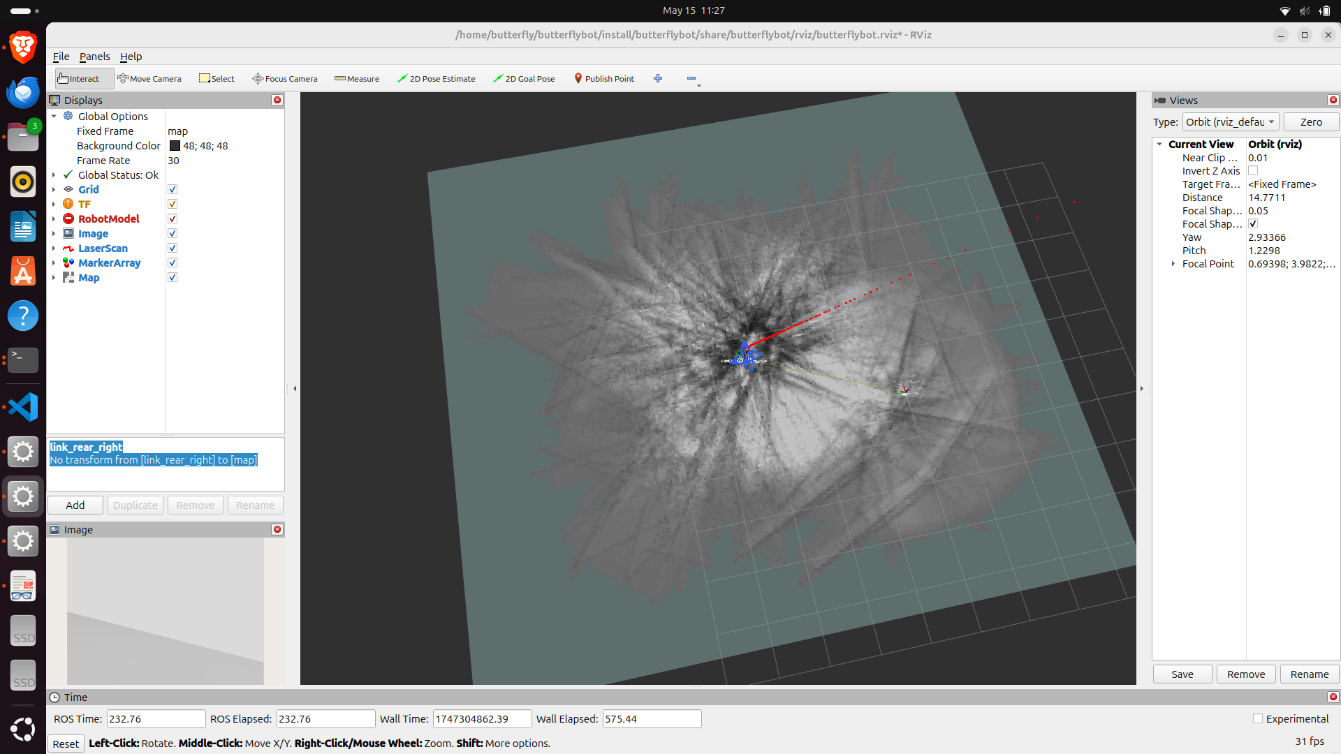


Figure 3.2.2: Noisy result of the dynamic mapping test.

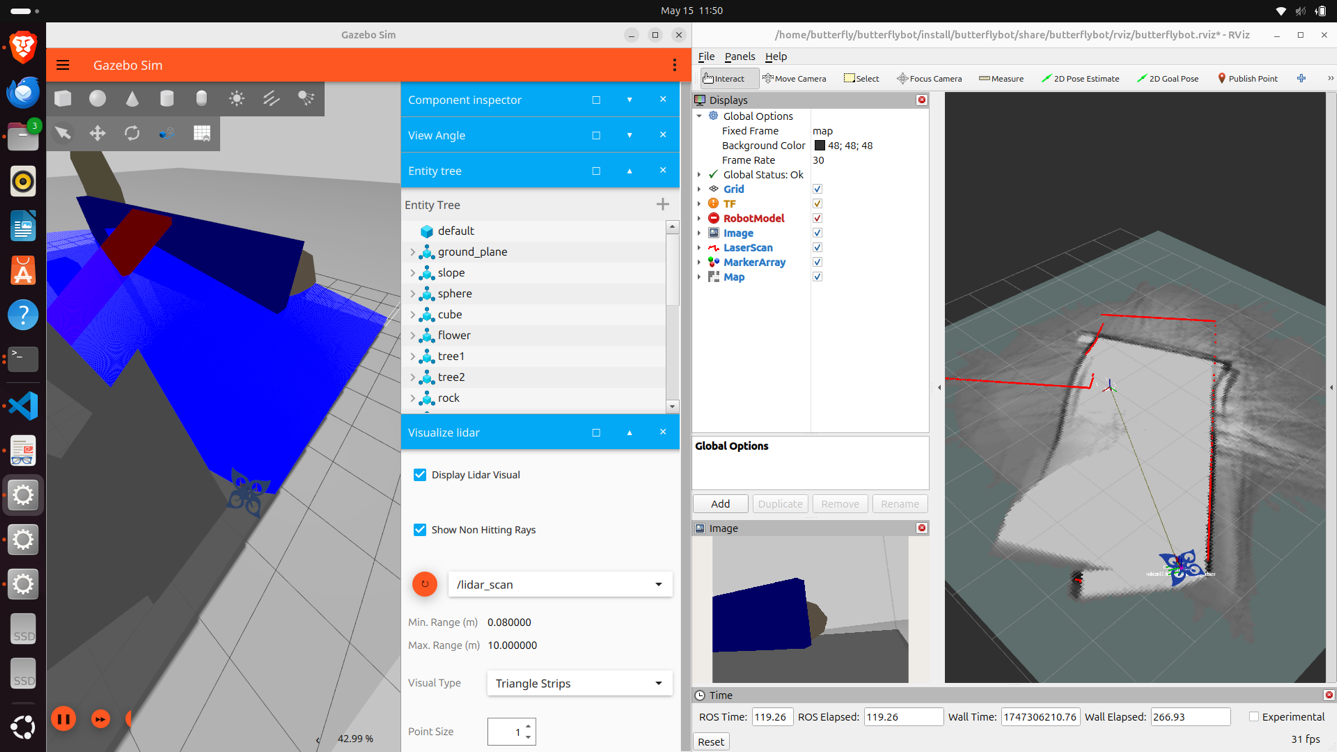


Figure 3.2.3: Result of the dynamic mapping test when the robot stabilised after crashing.

## 3.3 Autonomous Navigation

Unfortunately, extensive autonomous navigation tests were unable to be conducted because of limited time and resources. Evaluation scenarios were established to thoroughly evaluate the quadcopter’s autonomous capabilities, considering its integration with the object detection, mapping and obstacle avoidance systems developed by other team members. The following sections describe the evaluation scenarios that are intended to be use and the criteria that will be employed to assess the results.

**Proposed Integrated Test Scenarios**

**Scenario 1: Obstacle-rich Path Test (Obstacle Avoidance and Mapping)**

**Setup:**

* A simulated environment populated with obstacles and predefined mapping areas, testing the LiDAR-based SLAM mapping capabilities developed by team members using the Cartographer package.

**Execution:**

* Autonomous flight nodes, built upon the thruster control approach and integrated with ROS2 Nav2 navigation stack, would navigate safely through obstacle courses, dynamically avoiding obstacles using LiDAR and IMU sensor data.

**Evaluation Metrics:**

* **Obstacle Avoidance Manoeuvres:** Success rate and efficiency of autonomous avoidance.
* **Mapping Accuracy:** Accuracy and completeness of the environmental map generated using Cartographer.
* **Trajectory Efficiency:** Smoothness and overall flight efficiency compared to manual benchmarks.

**Intended Deliverables:**

* Videos and annotated screenshots demonstrating integrated autonomous navigation and mapping performance.

**Scenario 2: Precision Landing Test (Color-based Landing Integration)**

**Setup:**

* Landing pads identified by distinctive colour markers, proving the OpenCV-based colour recognition functionality developed by teammates.

**Execution:**

* Autonomous navigation routines, combining our thruster-based approach and OpenCV color-recognition system, guiding the quadcopter to perform controlled precision landings.

**Evaluation Metrics:**

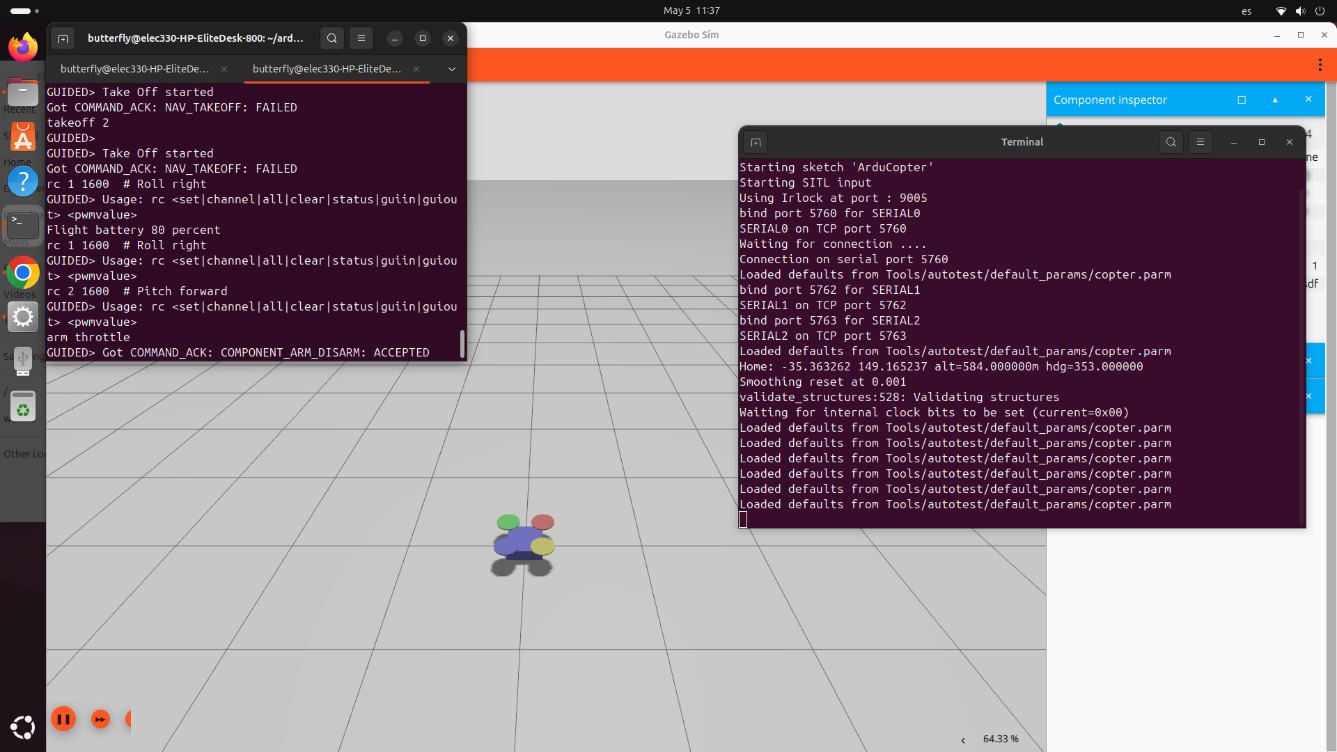
* **Successful Landing Rate:** Proportion of successful autonomous landings.
* **Positional Accuracy:** Precision of landing relative to marker centre.
* **System Integration Performance:** Stability and consistency of interactions between autonomous navigation routines and color-based landing detection.

**Intended Deliverables:**

* Demonstrative video footage and statistical summaries of performance and accuracy.

**Figure 3.3: Initial Integration Testing**

Screenshot demonstrating initial connectivity tests between ArduPilot SITL and a simplified drone model in Gazebo, confirming basic command acceptance and initial communication setup.

Figure 3.3: ArduPilot screenshot showing initial integration testing.

# 4 Discussion and Reflection

The newer robot design proved helpful, as it was more practical to simulate a quadcopter drone than a flapping butterfly given the time constraints. Even so, the team experienced challenges with exporting the CAD model from SolidWorks and converting it into a URDF and SDF that can be manipulated with ROS 2 Jazzy via simulation in Gazebo Harmonic and Rviz. The initial CAD export produced meshes with excessively high detail, resulting in prolonged load times and frame rates in the single digits.

To mitigate these performance issues, the team applied aggressive mesh‑decimation and multi‑level‑of‑detail techniques. Inertia values were computed assuming homogeneous materials; the addition of real wiring, batteries, and fasteners will shift the centre of mass and may introduce unanticipated interactions with the control loops. Also, the constrained schedule for autonomous flight testing left little opportunity for hardware trials, so troubleshooting efforts relied predominantly on simulation and rapid workaround strategies.

Although the current implementation successfully integrates multiple sensors into the robot model, there are several limitations in the way sensor data is processed and utilised. Firstly, while the LiDAR, IMU, and bounding box camera are all individually functional and properly bridged into ROS 2, the system lacks any form of sensor fusion. The data streams remain isolated, without a unified framework such as an Extended Kalman Filter to estimate the robot’s pose from multiple sources. This affects the stability and reliability of navigation.

Secondly, while the LiDAR data (via the */lidar\_scan* topic) and Cartographer can convert laser scans into a 2D occupancy grid map, the robot’s unstable and random flight pattern creates noise, making it difficult to generate a reliable and useful map. Furthermore, mapping vertical obstacles was not possible as 2D LiDAR restricts mapping to the horizontal plane. As a result, the robot operates without a coherent understanding of the surrounding space.

In addition, the use of a boundingbox\_camera introduces compatibility challenges. This camera type is specific to Gazebo and does not output standard RGB image topics. The current configuration does not provide a general image stream, making it difficult to apply common vision-based algorithms for object detection. The TF tree also lacks complete definition. Although a transform broadcaster is included, there is no clear mechanism to ensure correct spatial alignment between the robot base and each sensor frame. This may lead to inconsistencies when visualising data in RViz or when performing coordinate transformations. Finally, the IMU model is basic and only simulates Gaussian noise. Its data is not used beyond simple publishing, and there is no attempt to estimate orientation or integrate motion over time.

To address these issues, several improvements of the future work could be implemented in the following aspects. Replacing or supplementing the camera with a standard RGB sensor would enhance vision compatibility. A 3D scanning and mapping solution can be implemented, such as replacing the 2D LiDAR sensor with a 3D point cloud sensor. The TF tree should be completed using static or dynamic transform publishers, and existing transform publishers should be thoroughly checked and refined if needed. Furthermore, care should be taken to rectify the unstable flight of the robot, by ensuring the propellors apply appropriate thrusting and inertial forces and further refining the implementation of ArduPilot in the robot’s functionality. Lastly, leveraging the IMU data for pose estimation or dead reckoning would further enhance system awareness.

# 5 Bill of Materials (BOM)

Table 1 below gives a list of materials that could be used to build a real-life prototype of the drone:

Table 1: Bill of Materials

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **Component** | **Spec / Link to data sheet** | **Qty** | **Unit price (GBP)** | **Sub-total (GBP)** | **Why this part?** |
| 1 | **Brushless motor** | EMAX MT1806 2280 kV, CW/CCW mix ([vintagemodelcompany.com](https://www.vintagemodelcompany.com/emax-mt1806-2280kv-brushless-motor?utm_source=chatgpt.com)) | 4 | £18.00 | **£72.00** | Proven 18 g racing motor sized for ≤6″ props; 460 g thrust on 3 S gives >2∶1 T/W with safety margin. |
| 2 | **ESC** | T-Motor AT 12 A 2–3 S (with 5 V BEC) ([hobbyrc.co.uk](https://www.hobbyrc.co.uk/escs?utm_source=chatgpt.com)) | 4 | £8.40 | **£33.60** | Matches motor current, BLHeli-S firmware, slim 5 g package keeps arm sections narrow. |
| 3 | **Propeller set** | Gemfan 5030, 2 × CW + 2 × CCW (4 pcs) ([porcupinerc.com](https://www.porcupinerc.com/Gemfan-5030-Multirotor-CRP-Propellers-4pcs-Black_p_189.html?utm_source=chatgpt.com)) | 1 | £1.96\* | **£1.96** | Light 5″ props maximise motor efficiency; inexpensive to replace after crashes. *(US $2.60 → £1.96 @ 0.753 GBP/USD)* |
| 4 | **Flight MCU** | Arduino Pro Mini 328, 3.3 V / 8 MHz ([SK Pang Electronics Ltd](https://www.skpang.co.uk/products/arduino-pro-mini-328-3-3v-8mhz?utm_source=chatgpt.com)) | 1 | £7.20 | **£7.20** | Minimal, low-power board; enough I/O for IMU, ESC signal lines and telemetry bridge. |
| 5 | **IMU** | GY-521 MPU-6050 (6-DoF) ([switchelectronics.co.uk](https://www.switchelectronics.co.uk/products/gy-521-mpu-6050-3-axis-gyroscope-accelerometer-gyro-module?srsltid=AfmBOopH2kpn6bbQP4Me34UO_Z1fAfL9KTeOJFT8kMExWOlens3EQAKU&utm_source=chatgpt.com)) | 1 | £4.78 | **£4.78** | Provides gyro + acc data required for attitude control; widely supported libraries. |
| 6 | **Battery** | 3 S Li-Po 1000 mAh 65 C (Liperior) ([eBay UK](https://www.ebay.co.uk/itm/134878792001?utm_source=chatgpt.com)) | 1 | £15.99 | **£15.99** | Delivers ≈20 A continuous – enough for full-throttle hover with 8 min endurance. |
| 7 | **3-D-printed frame** | 1 kg spool PLA (1.75 mm) ([3D Prima](https://www.3dprima.co.uk/filament-resin/filaments?srsltid=AfmBOoofUZqdS3F7Ko3ijaULUlzuxlyRf_zCFoJIRxRUgpSe1zEga_L2&utm_source=chatgpt.com)) | 1 | £14.95 | **£14.95** | One spool covers two complete frames + spares; PLA easy to print and recycle. |
| 8 | **Wiring & connectors** | 18 AWG silicone lead, XT30 pigtail, bullet sets | — | £5.00 | **£5.00** | High-strand wire minimises resistive loss; XT30 suits 20 A class build. |

**Total Cost：£155.48**

Cost justification & trade-offs

* Motors + ESCs account for ~60 % of airborne power electronics spend. Cheaper 1104–1306 “whoop” motors were rejected because thrust margins fell below 1.3∶1, risking loss of attitude control in gusts.
* Arduino Pro Mini cuts £40-plus compared with a Pixhawk while still supporting complementary‐filter attitude loops (rate-only Nav1 scope).
* Gemfan thermoplastic props are inexpensive consumables; crash-replacement stock adds just £2 to the budget.
* PLA frame is the biggest single non-electronic item; switching to carbon-filled nylon would triple filament cost for only a 14 % stiffness gain—unjustified at this scale.
* Battery size balances endurance and mass; moving up to 1300 mAh adds 24 g and only 1.2 min extra hover, therefore 1000 mAh is the sweet spot.
* A 10 % contingency keeps the project under half the allowed £350 ceiling, leaving room for sensors (e.g. optical flow) in future iterations without breaching budget.

# 6 Conclusions

The *butterflybot* assignment set out to transform an unstable flapping-wing mock-up into a credible testbed for autonomous perception and control. The team’s first major milestone was a complete mechanical redesign: decorative butterfly wings were retained for visual identity, but the lift mechanism was changed to a symmetric quad-rotor enclosed in a lightweight “∞-shaped” fuselage with perforated motor mounts and a hemispherical battery cage, bringing the centre of mass in line with the geometric centre and providing the stiffness-to-weight ratio required for precise attitude control .

On the software side, the model was exported to URDF/SDF and deployed in Gazebo Harmonic under ROS 2 Jazzy. A sensor stack — 2D LiDAR, IMU and bounding-box camera — was bridged to ROS topics, giving the robot real-time range, inertial and visual data feeds. Object-detection nodes based on HSV colour segmentation successfully identified colour-coded targets inside the arena, proving the vision pipeline and message flow. Cartographer-based SLAM generated 2D occupancy grids during hover tests, demonstrating that, with further refinements, the LiDAR and IMU configuration can support mapping in GPS-denied spaces.

Attempts to close the loop with full autonomous navigation revealed the most significant bottlenecks. ArduPilot SITL linked correctly to a simple quadcopter but became unstable once coupled to the detailed butterfly model, largely because university workstations throttled under load and the propeller joints were not yet fully bridged. A fall-back thruster plugin proved air-frame actuation works, but the time budget precluded implementation of Nav2-based trajectory tracking. Consequently, end-to-end autonomy remains future work, although the team has already drafted path-planning and precision-landing strategies that can be plugged in when compute and tuning time are available.

The bill of materials shows that a real-world prototype could be assembled for about £155, which is less than half the £350 ceiling, leaving resources for extra sensors such as optical flow or depth cameras.

#### Key take-aways

* A pragmatic shift from pure biomimicry to a hybrid quad-rotor architecture delivered the stability needed for perception research while preserving the butterfly aesthetic.
* The ROS 2 Jazzy + Gazebo Harmonic stack, combined with open-source SLAM, object-detection and thrust-control plugins, provides a reproducible digital twin that accelerates iteration and de-risks future hardware builds.
* Integration, not component capability, is now the dominant challenge: sensor fusion, propeller-level control loops and real-time computing power must be refined before autonomous flight can move from simulation to a physical prototype.

#### Outlook

With the mechanical model, sensor bridges and perception algorithms already validated in simulation, the next logical steps are (i) to introduce an Extended Kalman Filter for LiDAR-IMU-vision fusion, (ii) to migrate propulsion commands to hardware-in-the-loop PX4 or ArduPilot targets, and (iii) to port the Nav2 local-planner stack for full six-degree-of-freedom path-following. Completing these tasks will close the sim-to-real gap and enable *butterflybot* to progress from a promising digital butterfly to a silent, agile inspection platform in the real world.

# References

[1] G. C. H. E. de Croon, M. Perçin, B. D. W. Remes, R. Ruijsink, and C. De Wagter, *The DelFly*. Dordrecht: Springer Netherlands, 2016. doi: https://doi.org/10.1007/978-94-017-9208-0.

[2] K. Y. Ma, P. Chirarattananon, S. B. Fuller, R. J. Wood, “Controlled flight of a biologically inspired, insect-scale robot” Science, 340(6132), 603–607, 2013. <https://doi.org/10.1126/science.1231806>

[3] H. V. Phan, S. Aurecianus, T. K. L. Au, T. Kang, and H. C. Park, “Towards the Long-Endurance Flight of an Insect-Inspired, Tailless, Two-Winged, Flapping-Wing Flying Robot,” *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 5059–5066, Oct. 2020, doi: https://doi.org/10.1109/lra.2020.3005127.

[4] Y. Zhang et al., “Kinematic and aerodynamic investigation of the butterfly in forward free flight for the butterfly-inspired flapping-wing air vehicle,” Applied Sciences, vol. 11, no. 6, Art. no. 2620, 2021. MDPI. h[ttps://doi.org/10.3390/app11062620](https://doi.org/10.3390/app11062620)

[5] J. Zhu, H. Li, and T. Zhang, “Camera, LiDAR, and IMU based multi-sensor fusion SLAM: A survey,” *Tsinghua Science and Technology*, vol. 29, no. 2, pp. 415–429, Apr. 2024. [Online]. Available: <https://doi.org/10.26599/TST.2023.9010010>

[6] Z. Fan, L. Zhang, X. Wang, Y. Shen, and Y. Liu, “LiDAR, IMU, and camera fusion for simultaneous localization and mapping: a systematic review,” *Artificial Intelligence Review*, vol. 58, no. 6, Mar. 2025. Available: <https://doi.org/10.1007/s10462-025-11187-w>

[7] M. S. Alam, P. Rai, R. K. Tiwari, S. Hussain, B. Kumar, and A. Kumar, “A Novel Deep Learning Approach Utilizing HSV Color Space for Real-Time Early Fire,” *2022 IEEE 11th International Conference on Communication Systems and Network Technologies (CSNT)*, pp. 14–20, Mar. 2025, doi: https://doi.org/10.1109/csnt64827.2025.10968207.

[8] “ROS 2 Cartographer — ROS 2 workshop documentation,” *Readthedocs.io*, 2022. https://ros2-industrial-workshop.readthedocs.io/en/latest/\_source/navigation/ROS2-Cartographer.html

[9] Open Robotics, “Gazebo Sensors: GpuLidarSensor Class Reference,” *Gazebosim.org*, 2025. https://gazebosim.org/api/sensors/9/classgz\_1\_1sensors\_1\_1GpuLidarSensor.html (accessed May 18, 2025).

[10] S. Wang, D. A. Olejnik, C. de Wagter, B. W. van Oudheusden, G. C. H. E. de Croon, and S. Hamaza, "Battle the wind: improving flight stability of a flapping wing micro air vehicle under wind disturbance with onboard thermistor-based airflow sensing," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 9605–9612, Oct. 2022.

[11] N. O. Perez-Arancibia, N. J. Finio, and R. J. Wood, "Roll, pitch and yaw torque control for a robotic bee," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011, pp. 1381–1388.

[12] H. V. Phan, W. T. Lee, and H. C. Park, "Dynamic stability and flight control of biomimetic flapping-wing micro air vehicle," *Aerospace*, vol. 8, no. 12, p. 362, Dec. 2021.

# Appendices

## Team Roles, Contributions, and Reflections

**Alvaro**:

For the entire project, I was mainly tasked with creating autonomous navigation for the biomimetic robot. To begin, I joined forces with Gideon to examine and fix the important robot files (env\_ws.sdf, butterfly.sdf and butterfly.urdf) so that the LiDAR, camera and IMU sensors would be integrated correctly. We ran into several difficulties, mainly with launching ROS2 and setting up strong publisher-subscriber connexions.

I took part in experiments to improve how sensors are integrated, mainly focusing on persistent problems in sensor and ROS2-Gazebo interaction. For example, we improved the performance of LiDAR in simulations by updating links, joints and plugins in robot configuration files.

After our instructors recommended a change in strategy, I played a key role in creating and adjusting our butterfly-based design into a functional quadcopter model. We began by studying the topic and discussing it with the team which resulted in a design that has propellers and thrusters.

I was able to instal ArduPilot and set it up for the first time, as well as get the ArduPilot console to communicate with a simple drone inside Gazebo. While the initial integration and exchange of commands were successful, the problems with the hardware and software meant that further work had to stop.

Then, I suggested and tested a straightforward autonomous flight method using Gazebo’s built-in thruster plugins. The approach needed more optimization due to difficulties in the simulations.

It should be noted that I was given the task of implementing autonomous flight at a very short notice, so I did not have much time. Despite the problem, I was proactive and tried different solutions within the time I had.

Finally, I took responsibility for organizing and uploading various autonomous flight attempt scripts and documentation onto our team's GitHub repository, ensuring transparent and accessible project documentation for future reference and further development.

**Junyang ：**

This project was constructed using part of design from Assignment 2. Upon realising that our initial flapping-wing structure was inherently unstable and could not be effectively controlled, we transitioned to a simplified quadrotor architecture that maintained our initial performance objectives while utilising propeller-driven thrust for both lift and control.

I was tasked with the primary responsibility of reconstructing the new butterfly-inspired model within our updated simulation environment. This involved converting Yifan's refined STL meshes into a URDF representation and subsequently into SDF for seamless Gazebo integration. In collaboration with Zijin, I defined the complete skeleton hierarchy, including links, joints and their limits, collision, and visual geometry. I also established the coordinate frame of each sensor before configuring the ros\_gz\_bridge to channel all relevant topics between Gazebo and ROS 2. Zijin and I concentrated on all sensor-related development after the structural framework was established.

I reworked the initial launch file to consolidate model spawning, world loading, RViz setup, and sensor node startup into a single, cohesive script, which other team members subsequently extended with their own nodes and subscriptions. This was done to ensure that the LiDAR, IMU, and bounding-box camera's data streams would appear correctly on ROS 2 topics. I attempted to stabilise the original flapping-wing model by adjusting environmental settings and adjusting Gazebo plugin parameters, such as air density and mass properties, prior to finalising the design switch. However, these modifications were insufficient, and a review of analogous control strategies convinced us to abandon the flapping approach in favour of the quadrotor.

I collaborated with Zijin in our final simulation environment to resolve model-generation conflicts by remeshing and renaming collision objects. Additionally, I constructed multiple test worlds that were populated with arches, boulders, and pine trees. I also constructed a comprehensive TF tree and RViz configuration to accurately visualise sensor outputs in real time. Lastly, I contributed to the project documentation by outlining our design decisions, simulation setup, control integration, and test results.

**Zijin**:

This project builds on a previous design where we had already implemented basic navigation for a biomimetic butterfly robot. However, the original flapping-wing structure proved to be unstable and difficult to control. As a result, we collectively decided to replace it with a simplified quadrotor-based design that retained the visual identity of a butterfly but used propellers for control and lift.

My main responsibility was rebuilding the robot model for integration into the new simulation setup. I began by reviewing the SDF and launch files from the previous project, identifying reusable components, and migrating them into the new robot design. I worked closely with Junyang to convert the STL files into a URDF model, and then into SDF format for integration in Gazebo. The entire reconstruction of the model—including frame hierarchy, link and joint definitions, sensor frames, and ROS 2 bridge integration—was primarily handled by me.

After completing the structural model, I focused on writing all sensor-related code. This included defining the LiDAR, IMU, and bounding box camera in the SDF file, configuring their positions and update parameters, and ensuring they were properly bridged to ROS 2 topics using ros\_gz\_bridge. I also wrote the initial version of the launch file that brings together the robot model, environment, RViz, and relevant ROS 2 nodes. Although other team members later added their own nodes to the launch script, the base version was developed and tested by me.

Throughout the process, I encountered and resolved multiple technical issues, especially those related to mismatched file paths between the old and new models. I rewrote bridge configurations to match the new directory structure and ensure reliable data flow between Gazebo and ROS 2. I also contributed to configuring the simulation environment. In the early design phase, I experimented with physics-based lift by modifying air density through a Gazebo plugin and tuning the robot’s mass. While this method was not successful in producing controlled flight, it informed our decision to switch to a propeller-based design.

In the finalized environment, I created several models including an arch, rocks, and pine trees. I also helped configure RViz and construct the TF tree to ensure accurate sensor visualization. These two tasks were completed in collaboration with Junyang. Finally, I contributed to writing project documentation.

**Yifan Wang:**

As a team member, I suggested replacing the unstable flapping concept with a four-propeller layout after surveying papers on blending biomimetic looks with multirotor stability. I outlined key ideas—fixed “∞” wings, coincident mass and geometric centres, and honey-comb light-weighting—and shared them with the group.

I produced the full SolidWorks model: frame, prop mounts and hemispherical battery dome, more than 30 features with parametric sketches for quick redesigns. After exporting the assembly to URDF/SDF, wrote custom launch scripts so the craft starts in ROS 2 Jazzy and Gazebo Humble with one command.

I also compared MT1806 motors, 5030 props and a 3 S 1000 mAh pack, turned the analysis into the bill of materials.

Beyond mechanical duties, I teamed up with Alvaro to tackle autonomous navigation tasks. Together, we sifted through a lots of papers and open-source examples on ROS 2 Nav2, optical-flow odometry, 3-D SLAM and obstacle avoidance. Fixed TF trees and remapped topics so LiDAR, IMU and simulated camera data fused correctly.

During testing jointly debugging timing mismatches between the Gazebo physics engine and ArduPilot SITL. At the same time, I kept the CAD model up to date, pushed literature links and presented navigation progress, keeping the team aligned on navigation goals.

Key Achievements：I rebuilt the entire aircraft in 3-D, turning the original flapping-wing sketch into the present “∞-shaped” quad-rotor and modelling every structural detail in SolidWorks. After exporting the assembly with the SolidWorks-to-URDF plug-in, the model loaded correctly in ROS 2 Jazzy / Gazebo Humble. Working with Alvaro, I then integrated the ROS 2 Nav2 navigation stack, configured ArduPilot, and tuned TF trees, planner parameters and PID gains so the virtual vehicle could carry out waypoint missions in simulation. Although the tight schedule meant full autonomous flight was not completely achieved, most of the pipeline—from map build to path following—was implemented. The process gave me practical command of Gazebo’s new plug-in ecosystem and solid hands-on experience with SLAM, localisation and path-tracking modules.

Challenges & Limitations：The first CAD export produced meshes that were far too dense, leading to long load times and single-digit FPS; aggressive mesh decimation and level-of-detail techniques were needed to restore performance. Inertia values still assume homogeneous materials—real wiring, batteries and fasteners will shift the centre of mass and could couple unexpectedly with the control loops. Finally, the limited time available for autonomous-flight tasks left little room for flight-test iterations, so troubleshooting had to focus on simulation and on quick alternative approaches.

Lessons Learnt：Walking the full chain—SolidWorks → URDF → SDF → ROS 2 / Gazebo → Nav2—taught me how to convert detailed CAD into a physically consistent digital twin and anchor an autonomous-flight stack on top of it. The experience sharpened my sense of where to trade geometric fidelity for real-time speed and deepened my understanding of how navigation algorithms depend on an accurate mechanical and inertial model.

**Gideon:**

As the deputy project manager, I led the group’s initial planning discussion at the start of Assignment 3, and wrote a draft plan for the team, covering key milestones we needed to achieve and the division of labour.

From weeks 2 onwards, I worked with Alvaro on the task of mapping the simulated environment. However, we encountered several issues in the process, as we realised there were several bugs in the code. The key difficulty was establishing a suitable publisher-subscriber relationship between relevant topics in the old design, as the transforms were poorly set. This made it difficult to add the relevant RViz display data for mapping, as the robot frames and transforms were misaligned.

I meticulously analysing each line of code in the drone’s .sdf and .urdf file, the world file (env\_ws.sdf), launch files and associated node files, to identify the misaligned parts and correct them. Other workarounds included trying to replace the gpu\_lidar with a point cloud lidar or ultrasound sensor. While I made good improvements to the code, including re-linking the correct paths to each file in the code, I soon realised it would be difficult to proceed with the existing robot design, due to the multiple launch and TF errors encountered at each stage.

I bolstered communication within the group by ensuring to check on other teammates when they did not show up. I relayed important information to them, such as when the instructors informed us that we could modify our design to include propellors.

I conducted project work both during the scheduled labs and at home via an OracleVM, USB-installed virtual machine running the same OS and software packages as the university PC. Due to the instability of the virtual machine, I ultimately resolved to dual-boot Linux on my laptop to allow smoother performance, as it had twice the RAM of the University PCs.

Once the new design was made, I updated the plugins in the world file to allow for quicker visualisation of the LiDAR data and entity tree upon launch.

During the Easter Break, I modelled a plan for the work to be done by each group member before bench inspection day. I also encouraged that we create and regularly update our group GitHub repository. This encouraged productivity and pro-active engagement, along with providing clarity to all members as the key deadlines were approaching.

As bench inspection approached, I drew inspiration from the turtlebot3\_cartographer and turtlebot3\_navigation2 packages to create the initial versions of the mapping and navigation sub-packages. However, it proved challenging to configure the files to work seamlessly by bench inspection day.

To assist with easy access to files, I created a Linktree to house important links we needed for Bench Inspection Day. Post-bench inspection, I continued to work on the project until the deadline. I created templates for the Assignment 3 report and Executive Summary on behalf of the group and uploaded them to the group’s OneDrive for easy access. I also made a backup of the group’s work, which also enabled me to continue working from home using the up-to-date code. From then onwards, I thoroughly cleaned up and commented the code, taking extra care to substitute the full file paths for dynamic ones. This change ensured that the file paths of the STL files referenced within the .sdf and .urdf files can be loaded up properly when launching the program, even if the package has been moved to a different workspace.

Other important work I did post-bench inspection includes the following. I edited the thruster topics in the .sdf file to match those in the launch file, serving as a foundation for flight. I successfully visualised the butterfly drone in Rviz, with a joint\_state\_publisher\_gui plugin for manually checking the movement of the joints. Shortly after, I achieved the breakthrough of getting the mapping working by integrating the cartographer\_ros package. This was achieved by creating custom configuration files and appropriate static and dynamic transforms. I added more Rviz displays, like RobotModel display, ensuring the topics match appropriately. While I couldn’t resolve the erratic flight of the robot, which would have aided in my task of producing a reliable 2D map, I am ultimately grateful for the progress made and the learning and collaboration achieved throughout this project.

**Yanzhang:**

I worked as a team member in the project.

This project is based on our previous version project whose details shown in Assignment 2. Its original intention was to design a flapping-wing model of a bionic butterfly and conduct simulations using Gazebo and ROS2. In the original project, apart from the modelling of the initial model and the addition of simulation environment contents, from creating the environment file, sensor configuration, model motion balance to model dynamic flight, all were completed under my contribution.

However, as further study of Gazebo and ROS2, I discovered that the simulation capabilities of Gazebo in fluid mechanics have exceeded the functional scope it can handle on its own. The buoyancy I initially designed was simply not sufficient to keep our model balanced during movement. This technical limitation compelled us to undertake subsequent remedial projects - transforming the original design into a butterfly-shaped unmanned aerial vehicle to complete a more reliable SLAM task.

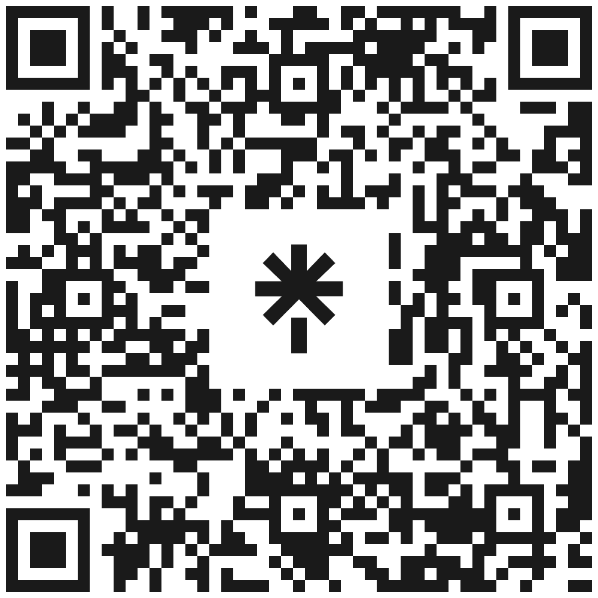
In this remedial project, I once again participated in the same series of tasks as in the initial project, such as generating the ROS2 model and equipping the sensors. As the core of the team's technical support, I was mainly responsible for solving various technical problems that my teammates encountered during the work, ensuring the smooth progress of the project.

Compared to the original project, the significant innovation I made in the new project lies in the design and implementation of the target recognition system. This system receives image data from the topic of “box\_camera” and performs precise colour recognition based on the HSV colour space, significantly enhancing the environmental perception and target capture capabilities of the unmanned aerial vehicle.

Apart from the technical implementation, I was also responsible for creating and managing the GitHub repository, establishing an effective version control system to facilitate collaborative development among the team. Additionally, I undertook the task of writing project documentation to ensure that the development process and technical details were fully documented.

## Project Files and Code

All code and files can be accessed via the QR Code below:



If unable to scan the QR Code, or if it does not function as expected, kindly access the group’s multimedia via the following links:

* Linktree: <https://linktr.ee/NectarWings>
* GitHub Repository: <https://github.com/YanzhangWang/ELEC330_Butterfly>
* Google Drive Folder: <https://drive.google.com/drive/folders/1MfXtSQJq7_PgpVzn3WEl8bAbTliOOvM7>

A ReadMe and the most important aspects of the code are included below:

**ReadMe (Setup Instructions)**

Follow these instructions to launch the simulation:

MAIN METHOD

1. Download/pull the code from GitHub to your Linux PC's home directory (e.g., "*home/butterfly/*" or "*home/user/*").

2. Open a terminal and cd into the butterflybot file.

*cd butterflybot*

3. Build the files

*colcon build*

4. Source the workspace

*source install/setup.bash*

5. Launch the simulation

*ros2 launch butterflybot launch.py*

ALTERNATIVE METHOD

1. Download/pull the code from GitHub to your Linux PC's home directory.

e.g., "*home/butterfly/*" or "*home/user/*"

2. Open a terminal and cd into the butterflybot scripts file.

*cd butterflybot/scripts*

3. Open a Terminal and give the butterflybot.sh script excecute permissions \*:

*chmod +x butterflybot.sh*

\* This step is only useful during the first launch. You can skip it during future launches.

4. Run the script with:

*./butterflybot.sh*

**launch.py**

import os

from ament\_index\_python.packages import get\_package\_share\_directory

from launch import LaunchDescription

from launch.substitutions import LaunchConfiguration, Command

from launch\_ros.substitutions import FindPackageShare

from launch.actions import DeclareLaunchArgument, SetEnvironmentVariable

from launch\_ros.actions import Node

from launch\_ros.parameter\_descriptions import ParameterValue

from launch.actions import IncludeLaunchDescription

from launch.launch\_description\_sources import PythonLaunchDescriptionSource

from ament\_index\_python.packages import get\_package\_share\_directory

from launch.actions import ExecuteProcess

def generate\_launch\_description():

    use\_sim\_time = LaunchConfiguration('use\_sim\_time')

    sdf\_file\_name = 'butterflybot.sdf'

    rviz\_config = 'butterflybot.rviz'

    urdf\_robot\_file = 'butterflybot.urdf'  # or butterflybot.xacro

    path\_to\_urdf\_robot = os.path.join(

    get\_package\_share\_directory('butterflybot'),

    'urdf',

    urdf\_robot\_file)

    path\_to\_sdf = os.path.join(

        get\_package\_share\_directory('butterflybot'),

        'urdf',

        sdf\_file\_name)

    path\_to\_world = os.path.join(

        get\_package\_share\_directory('butterflybot'),

        'urdf',

        'env\_ws.sdf')

    path\_to\_rviz = os.path.join(

        get\_package\_share\_directory('butterflybot'),

        'rviz',

        rviz\_config)

    path\_to\_config = os.path.join(

        get\_package\_share\_directory('butterflybot'),

        'config'

    )

    world\_launch = f"-r -v 4 {path\_to\_world}"

    resource\_path = get\_package\_share\_directory('butterflybot')

    set\_gz\_sim\_resource\_path = SetEnvironmentVariable(

        name='GZ\_SIM\_RESOURCE\_PATH',

        value=resource\_path

    )

    node\_robot\_state\_publisher = Node(

        package='robot\_state\_publisher',

        executable='robot\_state\_publisher',

        name='robot\_state\_publisher',

        output='screen',

        parameters=[{

            'robot\_description': ParameterValue(open(path\_to\_urdf\_robot, 'r').read(), value\_type=str),

            'use\_sim\_time': use\_sim\_time

        }]

    )

    gz\_sim = IncludeLaunchDescription(

        PythonLaunchDescriptionSource(

            os.path.join(

                get\_package\_share\_directory("ros\_gz\_sim"),

                "launch",

                "gz\_sim.launch.py"

            )

        ),

        launch\_arguments={"gz\_args": world\_launch}.items(),

    )

    spawn\_entity = Node(

        package="ros\_gz\_sim",

        executable="create",

        arguments=[

            "-name", "robot1",

            "-file", path\_to\_sdf,

            "-x", "-1", "-y", "0", "-z", "0.5"

        ],

        output="screen",

    )

    bridge = Node(

        package='ros\_gz\_bridge',

        executable='parameter\_bridge',

        name='ros\_gz\_bridge',

        arguments=[

            # Joint States

            '/joint\_states@sensor\_msgs/msg/JointState[gz.msgs.Model',

            '/model/robot1/joint\_state@sensor\_msgs/msg/JointState[gz.msgs.Model',

            # Propellor Joint Data

            '/joint\_front\_left\_topic@std\_msgs/msg/Float64@gz.msgs.Double',

            '/joint\_front\_right\_topic@std\_msgs/msg/Float64@gz.msgs.Double',

            '/joint\_rear\_left\_topic@std\_msgs/msg/Float64@gz.msgs.Double',

            '/joint\_rear\_right\_topic@std\_msgs/msg/Float64@gz.msgs.Double',

            # Propellor Thrust Data

            '/model/robot1/joint/joint\_front\_left/cmd\_thrust@std\_msgs/msg/Float64@gz.msgs.Double',

            '/model/robot1/joint/joint\_front\_right/cmd\_thrust@std\_msgs/msg/Float64@gz.msgs.Double',

            '/model/robot1/joint/joint\_rear\_left/cmd\_thrust@std\_msgs/msg/Float64@gz.msgs.Double',

            '/model/robot1/joint/joint\_rear\_right/cmd\_thrust@std\_msgs/msg/Float64@gz.msgs.Double',

             # Bridge velocity commands (if controlling with velocity) between ROS 2 and Gazebo

            '/cmd\_vel@geometry\_msgs/msg/Twist[gz.msgs.Twist',

            # Sensor Data

            '/boxes\_image@sensor\_msgs/msg/Image[gz.msgs.Image',

            '/lidar\_scan@sensor\_msgs/msg/LaserScan[gz.msgs.LaserScan',

            '/imu/data@sensor\_msgs/msg/Imu[gz.msgs.IMU',

            # Visualization Marker Array Bridge

            '/visualization\_marker\_array@visualization\_msgs/msg/MarkerArray[gz.msgs.Marker',

            # Navigation Topic

            #'/map@nav\_msgs/msg/OccupancyGrid[gz.msgs.OccupancyGrid',

            #'/path@nav\_msgs/msg/Path[gz.msgs.Path',

            #'/odom@nav\_msgs/msg/Odometry[gz.msgs.Odometry',

            # TF Data

            '/tf@tf2\_msgs/msg/TFMessage[gz.msgs.Pose\_V',

            # '/tf\_static@tf2\_msgs/msg/TFMessage[gz.msgs.Pose\_V',

            # Clock

            '/clock@rosgraph\_msgs/msg/Clock[gz.msgs.Clock',

        ],

        output='screen'

    )

    # Add TF broadcaster to send transforms for base\_link and camera\_link

    transform\_broadcaster\_node = Node(

        package='butterflybot',  # Your package name

        executable='transform\_broadcaster',  # The Python script you created

        name='transform\_broadcaster\_node',

        output='screen'

    )

    # Static Transform Publisher for map to world

    static\_transform\_node = Node(

        package='tf2\_ros',

        executable='static\_transform\_publisher',

        name='map\_to\_world\_broadcaster',

        output='screen',

        arguments=[

            '--x', '0',

            '--y', '0',

            '--z', '0',

            '--qx', '0',

            '--qy', '0',

            '--qz', '0',

            '--qw', '1',

            '--frame-id', 'map',

            '--child-frame-id', 'world'

        ]

    )

    # Static transform from base\_link to lidar\_link

    static\_lidar\_transform = Node(

        package='tf2\_ros',

        executable='static\_transform\_publisher',

        name='base\_to\_lidar\_broadcaster',

        output='screen',

        arguments=[

            '--x', '0',

            '--y', '0',

            '--z', '0',

            '--qx', '0',

            '--qy', '0',

            '--qz', '0',

            '--qw', '1',

            '--frame-id', 'base\_link',

            '--child-frame-id', 'robot1/lidar\_link/gpu\_lidar'

        ]

    )

    # Static transform from base\_link to imu\_link

    static\_imu\_transform = Node(

        package='tf2\_ros',

        executable='static\_transform\_publisher',

        name='base\_to\_imu\_broadcaster',

        output='screen',

        arguments=[

            '--x', '0',

            '--y', '0',

            '--z', '0',

            '--qx', '0',

            '--qy', '0',

            '--qz', '0',

            '--qw', '1',

            '--frame-id', 'base\_link',

            '--child-frame-id', 'robot1/imu\_link/imu\_sensor'

        ]

    )

    # Add Object Detection Node

    object\_detection\_node = Node(

        package='butterflybot',

        executable='object\_detection\_node',  # This stays the same

        name='object\_detection\_node',

        parameters=[{'use\_sim\_time': use\_sim\_time}],

        output='screen'

    )

    joint\_state\_publisher\_gui\_node = Node(

        package="joint\_state\_publisher\_gui",

        executable="joint\_state\_publisher\_gui",

        name="joint\_state\_publisher\_gui",

        output='screen'

    )

    # Joint controller in a new terminal window

    joint\_controller\_node = ExecuteProcess(

        cmd=[

            'gnome-terminal', '--', 'bash', '-c',

            'ros2 run butterflybot joint\_controller; exec bash'

        ],

        output='screen'

    )

    rviz\_node = Node(

        package='rviz2',

        executable='rviz2',

        name='rviz2',

        arguments=['-d', path\_to\_rviz],

        parameters=[{'use\_sim\_time': use\_sim\_time}],

        output='screen'

    )

    # Cartographer SLAM Node

    cartographer\_node = Node(

        package='cartographer\_ros',

        executable='cartographer\_node',

        name='cartographer\_node',

        output='screen',

        parameters=[

            {'use\_sim\_time': use\_sim\_time},

            {'tf\_buffer\_qos\_durability': 'transient\_local'}

        ],

        arguments=[

            '-configuration\_directory', path\_to\_config,

            '-configuration\_basename', 'butterflybot\_2d.lua'

        ],

        remappings=[

           # ('odom', '/odom'),

            ('scan', '/lidar\_scan'),

            ('/imu', '/imu/data'),

        ]

    )

    # Occupancy Grid Node

    cartographer\_occupancy\_grid\_node = Node(

        package='cartographer\_ros',

        executable='cartographer\_occupancy\_grid\_node',

        name='cartographer\_occupancy\_grid\_node',

        parameters=[

            {'use\_sim\_time': use\_sim\_time},

            {'resolution': 0.05},

            {'publish\_period\_sec': 1.0}

        ],

        output='screen'

    )

    return LaunchDescription([

        DeclareLaunchArgument(

            'use\_sim\_time',

            default\_value='true',

            description='Use simulation time if true'

        ),

        set\_gz\_sim\_resource\_path,

        node\_robot\_state\_publisher,

        gz\_sim,

        spawn\_entity,

        bridge,

        rviz\_node,

        transform\_broadcaster\_node,

        static\_transform\_node,

        static\_lidar\_transform,

        static\_imu\_transform,

        object\_detection\_node,

        joint\_controller\_node,

        joint\_state\_publisher\_gui\_node,

        cartographer\_node,

        cartographer\_occupancy\_grid\_node

    ])

**butterflybot.sdf**

<?xml version="1.0" ?>

<sdf version='1.11'>

  <model name='Butterflybot'>

    <!-- ROBOT BODY -->

    <!-- Base -->

    <link name='base\_link'>

      <inertial>

        <pose>0.0467837442624986 0.047668828289127 0.0302566124276802 0 0 0</pose>

        <mass>1.81343265737798</mass>

        <inertia>

          <ixx>0.0292434022065882</ixx>

          <ixy>2.3758686826485899e-07</ixy>

          <ixz>-0.00012290016217477099</ixz>

          <iyy>0.037851344971650498</iyy>

          <iyz>1.35402349531979e-06</iyz>

          <izz>0.0087084350366914698</izz>

        </inertia>

      </inertial>

      <collision name='base\_link\_collision'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/base\_link.STL</uri>

          </mesh>

        </geometry>

      </collision>

      <visual name='base\_link\_visual'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/base\_link.STL</uri>

          </mesh>

        </geometry>

        <material>

          <diffuse>0.2 0.4 1.0 1.0</diffuse>

          <ambient>0.2 0.4 1.0 1.0</ambient>

        </material>

      </visual>

    </link>

    <!-- Front Left -->

    <joint name='joint\_front\_left' type='revolute'>

      <pose relative\_to='base\_link'>-0.079189999999999997 0.062068999999999999 0.058430000000000003 1.5708 0 -0.27447999999999984</pose>

      <parent>base\_link</parent>

      <child>link\_front\_left</child>

      <axis>

        <xyz>0 -1 0</xyz>

        <limit>

          <lower>-inf</lower>

          <upper>inf</upper>

        </limit>

        <dynamics>

          <spring\_reference>0</spring\_reference>

          <spring\_stiffness>0</spring\_stiffness>

        </dynamics>

      </axis>

    </joint>

    <link name='link\_front\_left'>

      <pose relative\_to='joint\_front\_left'>0 0 0 0 0 0</pose>

      <inertial>

        <pose>-1.3158665906898301e-08 0.0030876204619819199 3.2896587114084099e-08 0 0 0</pose>

        <mass>0.00097127102853952004</mass>

        <inertia>

          <ixx>1.0768895837296399e-06</ixx>

          <ixy>3.8328099170122897e-14</ixy>

          <ixz>1.9302501471666401e-07</ixz>

          <iyy>1.1185884576620301e-06</iyy>

          <iyz>2.18108405894349e-14</iyz>

          <izz>4.3791642931003e-08</izz>

        </inertia>

      </inertial>

      <collision name='link\_front\_left\_collision'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_front\_left.STL</uri>

          </mesh>

        </geometry>

      </collision>

      <visual name='link\_front\_left\_visual'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_front\_left.STL</uri>

          </mesh>

        </geometry>

        <material>

          <diffuse>1.0 0.2 0.6 1.0</diffuse>

          <ambient>1.0 0.2 0.6 1.0</ambient>

        </material>

      </visual>

    </link>

    <!-- Front Right -->

    <joint name='joint\_front\_right' type='revolute'>

      <pose relative\_to='base\_link'>0.17136999999999999 0.062745999999999996 0.059461 1.5708 0 -0.27447999999999984</pose>

      <parent>base\_link</parent>

      <child>link\_front\_right</child>

      <axis>

        <xyz>0 -1 0</xyz>

        <limit>

          <lower>-inf</lower>

          <upper>inf</upper>

        </limit>

        <dynamics>

          <spring\_reference>0</spring\_reference>

          <spring\_stiffness>0</spring\_stiffness>

        </dynamics>

      </axis>

    </joint>

    <link name='link\_front\_right'>

      <pose relative\_to='joint\_front\_right'>0 0 0 0 0 0</pose>

      <inertial>

        <pose>1.10804414255039e-08 0.0030876220561473302 -7.1078507857258404e-08 0 0 0</pose>

        <mass>0.000971268794454539</mass>

        <inertia>

          <ixx>1.0768847108197401e-06</ixx>

          <ixy>5.65245392107505e-14</ixy>

          <ixz>1.93023878397352e-07</ixz>

          <iyy>1.1185833114050301e-06</iyy>

          <iyz>-5.1246892916307002e-14</iyz>

          <izz>4.3791367120243599e-08</izz>

        </inertia>

      </inertial>

      <collision name='link\_front\_right\_collision'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_front\_right.STL</uri>

          </mesh>

        </geometry>

      </collision>

      <visual name='link\_front\_right\_visual'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_front\_right.STL</uri>

          </mesh>

        </geometry>

        <material>

          <diffuse>1.0 0.2 0.6 1.0</diffuse>

          <ambient>1.0 0.2 0.6 1.0</ambient>

        </material>

      </visual>

    </link>

    <!-- Rear Left -->

    <joint name='joint\_rear\_left' type='revolute'>

      <pose relative\_to='base\_link'>-0.080467999999999998 -0.19675000000000001 0.054993 1.5708 0 -0.27447999999999984</pose>

      <parent>base\_link</parent>

      <child>link\_rear\_left</child>

      <axis>

        <xyz>0 -1 0</xyz>

        <limit>

          <lower>-inf</lower>

          <upper>inf</upper>

        </limit>

        <dynamics>

          <spring\_reference>0</spring\_reference>

          <spring\_stiffness>0</spring\_stiffness>

        </dynamics>

      </axis>

    </joint>

    <link name='link\_rear\_left'>

      <pose relative\_to='joint\_rear\_left'>0 0 0 0 0 0</pose>

      <inertial>

        <pose>-1.3159026139575301e-08 0.0030876236503368901 3.2898901047850897e-08 0 0 0</pose>

        <mass>0.00097127102853952004</mass>

        <inertia>

          <ixx>1.0768798381927601e-06</ixx>

          <ixy>3.8328125808511603e-14</ixy>

          <ixz>1.9302274210829399e-07</ixz>

          <iyy>1.1185781654340401e-06</iyy>

          <iyz>2.1811183096077101e-14</iyz>

          <izz>4.3791091312534198e-08</izz>

        </inertia>

      </inertial>

      <collision name='link\_rear\_left\_collision'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_rear\_left.STL</uri>

          </mesh>

        </geometry>

      </collision>

      <visual name='link\_rear\_left\_visual'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_rear\_left.STL</uri>

          </mesh>

        </geometry>

        <material>

          <diffuse>1.0 1.0 0.3 1.0</diffuse>

          <ambient>1.0 1.0 0.3 1.0</ambient>

        </material>

      </visual>

    </link>

    <!-- Rear Right -->

    <joint name='joint\_rear\_right' type='revolute'>

      <pose relative\_to='base\_link'>0.17607 -0.19605 0.051529999999999999 1.5708 0 -0.27447999999999984</pose>

      <parent>base\_link</parent>

      <child>link\_rear\_right</child>

      <axis>

        <xyz>0 -1 0</xyz>

        <limit>

          <lower>-inf</lower>

          <upper>inf</upper>

        </limit>

        <dynamics>

          <spring\_reference>0</spring\_reference>

          <spring\_stiffness>0</spring\_stiffness>

        </dynamics>

      </axis>

    </joint>

    <link name='link\_rear\_right'>

      <pose relative\_to='joint\_rear\_right'>0 0 0 0 0 0</pose>

      <inertial>

        <pose>1.1080436457255899e-08 0.0030876220561439202 -7.1077704361099894e-08 0 0 0</pose>

        <mass>0.000971268794454539</mass>

        <inertia>

          <ixx>1.0768847111581501e-06</ixx>

          <ixy>5.6524537061138701e-14</ixy>

          <ixz>1.93023878437882e-07</ixz>

          <iyy>1.1185833117482701e-06</iyy>

          <iyz>-5.1247553697434301e-14</iyz>

          <izz>4.3791367125021401e-08</izz>

        </inertia>

      </inertial>

      <collision name='link\_rear\_right\_collision'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_rear\_right.STL</uri>

          </mesh>

        </geometry>

      </collision>

      <visual name='link\_rear\_right\_visual'>

        <pose>0 0 0 0 0 0</pose>

        <geometry>

          <mesh>

            <scale>1 1 1</scale>

            <uri>model://meshes/link\_rear\_right.STL</uri>

          </mesh>

        </geometry>

        <material>

          <diffuse>1.0 1.0 0.3 1.0</diffuse>

          <ambient>1.0 1.0 0.3 1.0</ambient>

        </material>

      </visual>

    </link>

    <!-- SENSORS -->

    <!-- LiDAR-->

    <link name="lidar\_link">

      <pose>0 0 0 0 0 -1.57</pose>

      <sensor name='gpu\_lidar' type='gpu\_lidar'>

        <pose relative\_to='lidar\_link'>0 0 0 0 0 3.14</pose>

        <topic>lidar\_scan</topic>

        <update\_rate>10</update\_rate>

        <ray>

            <scan>

                <horizontal>

                    <samples>640</samples>

                    <resolution>1</resolution>

                    <min\_angle>-1.396263</min\_angle>

                    <max\_angle>1.396263</max\_angle>

                </horizontal>

                <vertical>

                    <samples>1</samples>

                    <resolution>0.01</resolution>

                    <min\_angle>0</min\_angle>

                    <max\_angle>0</max\_angle>

                </vertical>

            </scan>

            <range>

                <min>0.08</min>

                <max>10.0</max>

                <resolution>0.01</resolution>

            </range>

        </ray>

        <always\_on>1</always\_on>

        <visualize>true</visualize>

      </sensor>

    </link>

    <joint name="lidar\_joint" type="fixed">

      <parent>base\_link</parent>

      <child>lidar\_link</child>

    </joint>

    <!-- Camera -->

    <link name="camera\_link">

      <pose>0 0 0 0 0 -1.57</pose>

      <sensor name="boundingbox\_camera" type="boundingbox\_camera">

         <pose relative\_to='camera\_link'>0 0 0.5 0 0 3.14</pose>

         <topic>boxes</topic> <!-- Do not change this -->

         <camera>

           <box\_type>2d</box\_type>

           <horizontal\_fov>1.047</horizontal\_fov>

           <image>

             <width>800</width>

             <height>600</height>

           </image>

           <clip>

             <near>0.1</near>

             <far>10</far>

           </clip>

         </camera>

         <always\_on>1</always\_on>

         <update\_rate>10</update\_rate>

         <visualize>true</visualize>

      </sensor>

    </link>

    <joint name="camera\_joint" type="fixed">

      <parent>base\_link</parent>

      <child>camera\_link</child>

    </joint>

    <!-- IMU -->

    <link name="imu\_link">

      <sensor name="imu\_sensor" type="imu">

        <pose relative\_to='imu\_link'>0 0 0.5 0 0 3.14</pose>

        <topic>imu/data</topic>

        <update\_rate>100</update\_rate>

        <always\_on>true</always\_on>

        <visualize>false</visualize>

        <imu>

          <linear\_acceleration>

            <x>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </x>

            <y>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </y>

            <z>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </z>

          </linear\_acceleration>

          <angular\_velocity>

            <x>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </x>

            <y>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </y>

            <z>

              <noise type="gaussian">

                <mean>0.0</mean>

                <stddev>0.01</stddev>

              </noise>

            </z>

          </angular\_velocity>

        </imu>

      </sensor>

    </link>

    <joint name="imu\_joint" type="fixed">

      <parent>base\_link</parent>

      <child>imu\_link</child>

    </joint>

    <!-- PLUGINS -->

    <!-- Thruster plugins-->

    <plugin

      filename="gz-sim-thruster-system"

      name="gz::sim::systems::Thruster">

      <use\_angvel\_cmd>0</use\_angvel\_cmd>

      <joint\_name>joint\_front\_left</joint\_name>

      <thrust\_coefficient>0.005</thrust\_coefficient>

      <fluid\_density>1.225</fluid\_density>

      <propeller\_diameter>0.2</propeller\_diameter>

    </plugin>

    <plugin

      filename="gz-sim-thruster-system"

      name="gz::sim::systems::Thruster">

      <use\_angvel\_cmd>0</use\_angvel\_cmd>

      <joint\_name>joint\_front\_right</joint\_name>

      <thrust\_coefficient>0.005</thrust\_coefficient>

      <fluid\_density>1.225</fluid\_density>

      <propeller\_diameter>0.2</propeller\_diameter>

    </plugin>

     <plugin

      filename="gz-sim-thruster-system"

      name="gz::sim::systems::Thruster">

      <use\_angvel\_cmd>0</use\_angvel\_cmd>

      <joint\_name>joint\_rear\_left</joint\_name>

      <thrust\_coefficient>0.005</thrust\_coefficient>

      <fluid\_density>1.225</fluid\_density>

      <propeller\_diameter>0.2</propeller\_diameter>

    </plugin>

    <plugin

      filename="gz-sim-thruster-system"

      name="gz::sim::systems::Thruster">

      <use\_angvel\_cmd>0</use\_angvel\_cmd>

      <joint\_name>joint\_rear\_right</joint\_name>

      <thrust\_coefficient>0.005</thrust\_coefficient>

      <fluid\_density>1.225</fluid\_density>

      <propeller\_diameter>0.2</propeller\_diameter>

    </plugin>

    <!-- Joint-state-publisher plugins-->

    <plugin

      filename="gz-sim-joint-state-publisher-system"

      name="gz::sim::systems::JointStatePublisher">

      <joint\_name>joint\_front\_left</joint\_name>

    </plugin>

    <plugin

      filename="gz-sim-joint-state-publisher-system"

      name="gz::sim::systems::JointStatePublisher">

      <joint\_name>joint\_front\_right</joint\_name>

    </plugin>

    <plugin

      filename="gz-sim-joint-state-publisher-system"

      name="gz::sim::systems::JointStatePublisher">

      <joint\_name>joint\_rear\_left</joint\_name>

    </plugin>

    <plugin

      filename="gz-sim-joint-state-publisher-system"

      name="gz::sim::systems::JointStatePublisher">

      <joint\_name>joint\_rear\_right</joint\_name>

    </plugin>

    <!-- Joint-position-controller plugins-->

    <plugin

      filename="gz-sim-joint-position-controller-system"

      name="gz::sim::systems::JointPositionController">

      <joint\_name>joint\_front\_left</joint\_name>

      <topic>joint\_front\_left\_topic</topic>

      <p\_gain>1</p\_gain>

      <i\_gain>0.1</i\_gain>

      <d\_gain>0.01</d\_gain>

      <i\_max>1</i\_max>

      <i\_min>-1</i\_min>

      <cmd\_max>1000</cmd\_max>

      <cmd\_min>-1000</cmd\_min>

      <force>0.0</force>

    </plugin>

    <plugin

      filename="gz-sim-joint-position-controller-system"

      name="gz::sim::systems::JointPositionController">

      <joint\_name>joint\_front\_right</joint\_name>

      <topic>joint\_front\_right\_topic</topic>

      <p\_gain>1</p\_gain>

      <i\_gain>0.1</i\_gain>

      <d\_gain>0.01</d\_gain>

      <i\_max>1</i\_max>

      <i\_min>-1</i\_min>

      <cmd\_max>1000</cmd\_max>

      <cmd\_min>-1000</cmd\_min>

      <force>0.0</force>

    </plugin>

    <plugin

      filename="gz-sim-joint-position-controller-system"

      name="gz::sim::systems::JointPositionController">

      <joint\_name>joint\_rear\_left</joint\_name>

      <topic>joint\_rear\_left\_topic</topic>

      <p\_gain>1</p\_gain>

      <i\_gain>0.1</i\_gain>

      <d\_gain>0.01</d\_gain>

      <i\_max>1</i\_max>

      <i\_min>-1</i\_min>

      <cmd\_max>1000</cmd\_max>

      <cmd\_min>-1000</cmd\_min>

      <force>0.0</force>

    </plugin>

    <plugin

      filename="gz-sim-joint-position-controller-system"

      name="gz::sim::systems::JointPositionController">

      <joint\_name>joint\_rear\_right</joint\_name>

      <topic>joint\_rear\_right\_topic</topic>

      <p\_gain>1</p\_gain>

      <i\_gain>0.1</i\_gain>

      <d\_gain>0.01</d\_gain>

      <i\_max>1</i\_max>

      <i\_min>-1</i\_min>

      <cmd\_max>1000</cmd\_max>

      <cmd\_min>-1000</cmd\_min>

      <force>0.0</force>

    </plugin>

    <!-- Pose, Static model, Self Collide -->

    <pose>0 0 0.0 0 0 0</pose>

      <static>false</static>

      <self\_collide>false</self\_collide>

  </model>

</sdf>