Chapter 3

METHODOLOGY AND DESIGN

3.1 Robot Architecture

The main processing unit for this project is Nvidia Jetson Xavier single board computer. Addition to that there will be a low level control unit, which is a micro controller developer board to run the motors of the differential rive unit and control other actuators and sensor readings. The two main control units will communicate through serial communication. Other peripherals of the robot will be connected to these units as follows in the **Figure 3.1**.

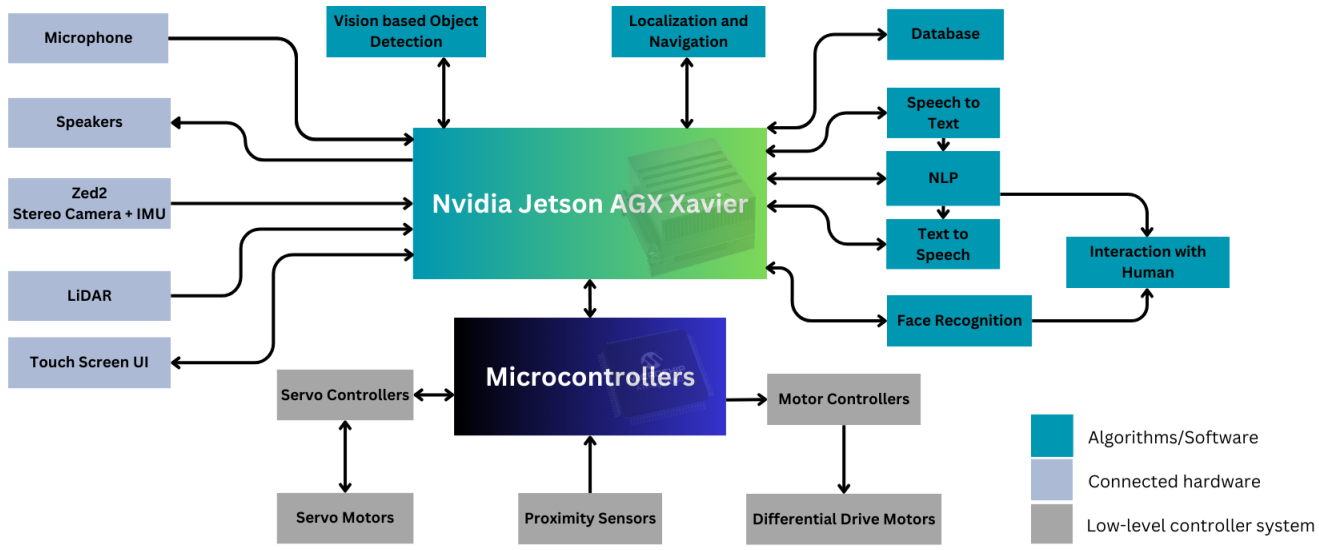


Figure 3.1: Robot architecture

3.2 Physical Design

One of the main part of the robot is the mobile platform. Mobile Platform of the robot refers to the base which carries the robot. As discussed before, the driving method for this platform will be differential wheel driving. The implementation of the robot will be done step-wise as mentioned below.

1. Design the required platform using a designing software. (SOLIDWORKS)
2. Check the stability and performance using simulations.
    - Create a URDF (Unified Robotics Description Format) model using the created Solidworks design.
    - Set the required physics properties to achieve proper simulation of the robot. (Gazebo simulator is the chosen simulation platform with ROS2 Foxy distribution.)
    - Test for various floor conditions such as bumps and cliffs.
3. After verifying the stability of the platform design, start the physical implementation.
    - Implement the bottom part of the platform to start mapping and navigation tasks.
    - Then start building the required upper body part of the robot.
4. Test the full physical implementation and verify results obtained from simulations and real-world design.

Resource usage and availability of material and components are crucial during the physical implementation. Proper planning and execution must be followed before every physical modification to the robot platform.

3.2.1 Solidworks Design

As the first step of this project, a conceptual design was made to get an idea of the robot's components and understanding how those components should be placed. Then with the available hardware resources, a more accurate design was done to do the actual implementation. Conceptual design is shown in **Figure 3.2** and the more practical design is shown in **Figure 3.3**.

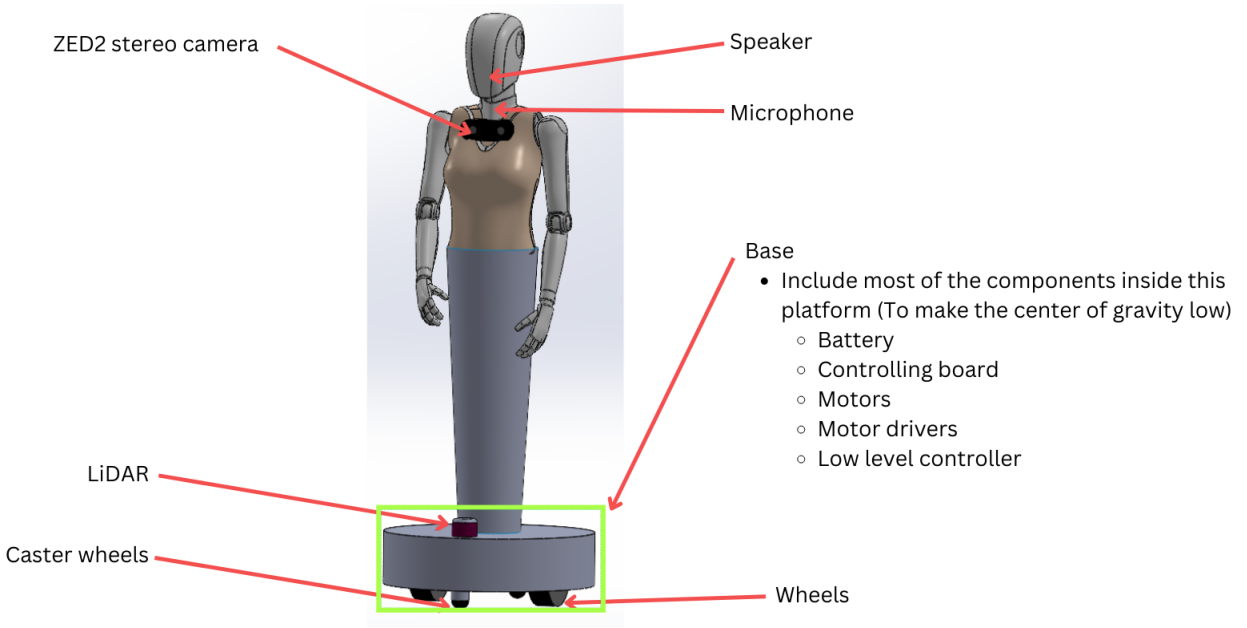


Figure 3.2: Conceptual design

Conceptual design had some conflicts with the resources. Therefore, a better design was done to replicate real world design.

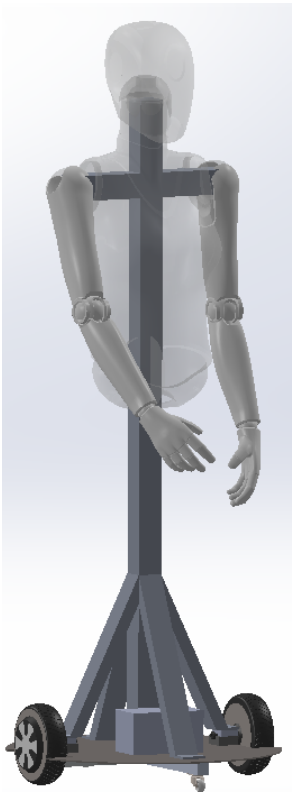


Figure 3.3: More practical design

3.2.2 Simulation

Another major part of this project is to simulate the robot's behaviour before implementing it physically. A simulation platform should be selected to continue the the implementation. Since we decided to do the robot's programs based on ROS2 , Gazebo was selected as the simulation platform.

To carry on a simulation on Gazebo, we need an URDF model of the robot. To create the model, the solidworks design can be used. There is an exporter[11] to export the model we want as an URDF file. There is a minor change that must be done before exporting the robot. To simulate the robot's caster wheels, they should be modeled as fixed spheres to simulate properly.

The axes we need for the joints (wheels) must be given at the exporting time. Also the meshes (shape of the parts) also should be exported to visualize the robot well inside the simulation.

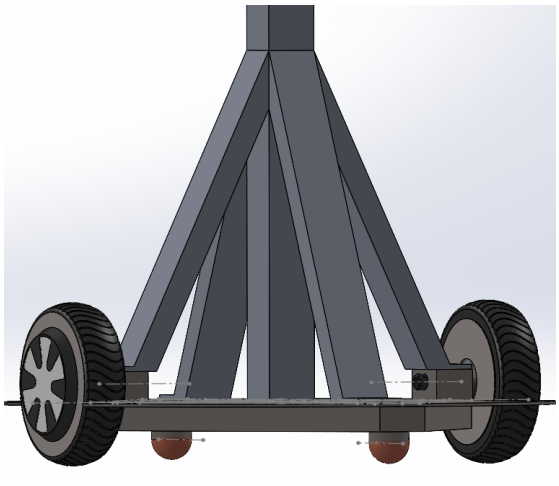


Figure 3.4: Changing caster wheels for the simulation

After exporting the model, the robot's axes can be visualized using rviz2. The axes of the wheel joints can be changed during the export. Therefore, proper setting of the joint parameters is a must for correct simulation. The other physics properties also must be set properly for correct simulation. Eg: friction coefficient, inertia, center of gravity, bounding shapes, gazebo environment settings. Also we must specify the robot controller which is used to operate the robot inside the simulation. The changes done for the caster wheel are shown in **Figure 3.4**. Visualization of the robot with links is shown in **Figure 3.5**.

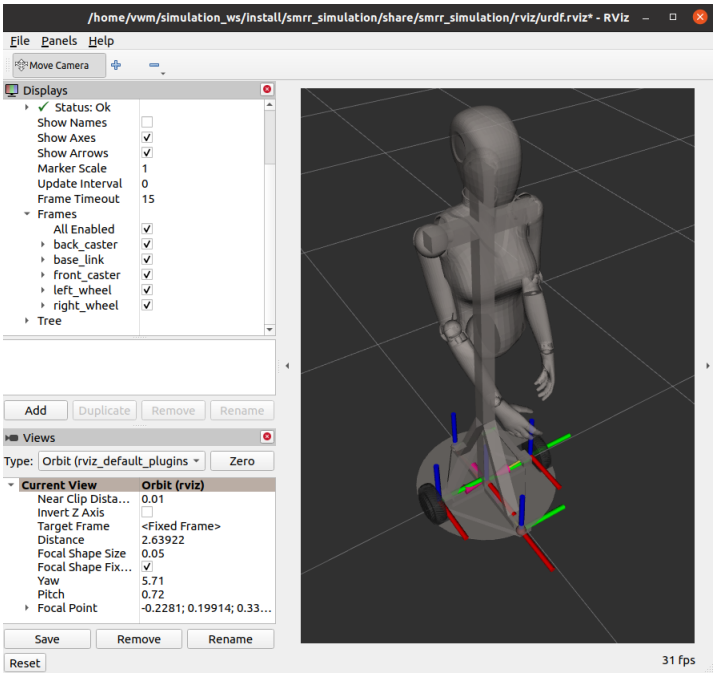
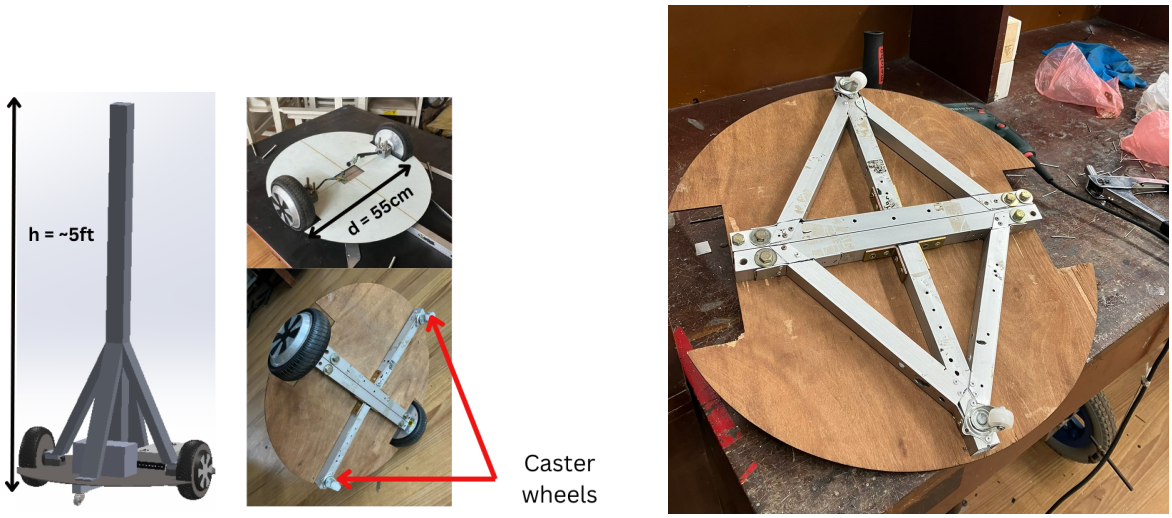


Figure 3.5: Visualizing the robot model

3.2.3 Physical Implementation

The physical implementation was done in steps to avoid conflicts with the conceptual design. The mobility part was a crucial feature to implement. Therefore, as the first step, the lower part of the platform was implemented according to the solidworks design. Steps are shown in **Figure 3.6**.



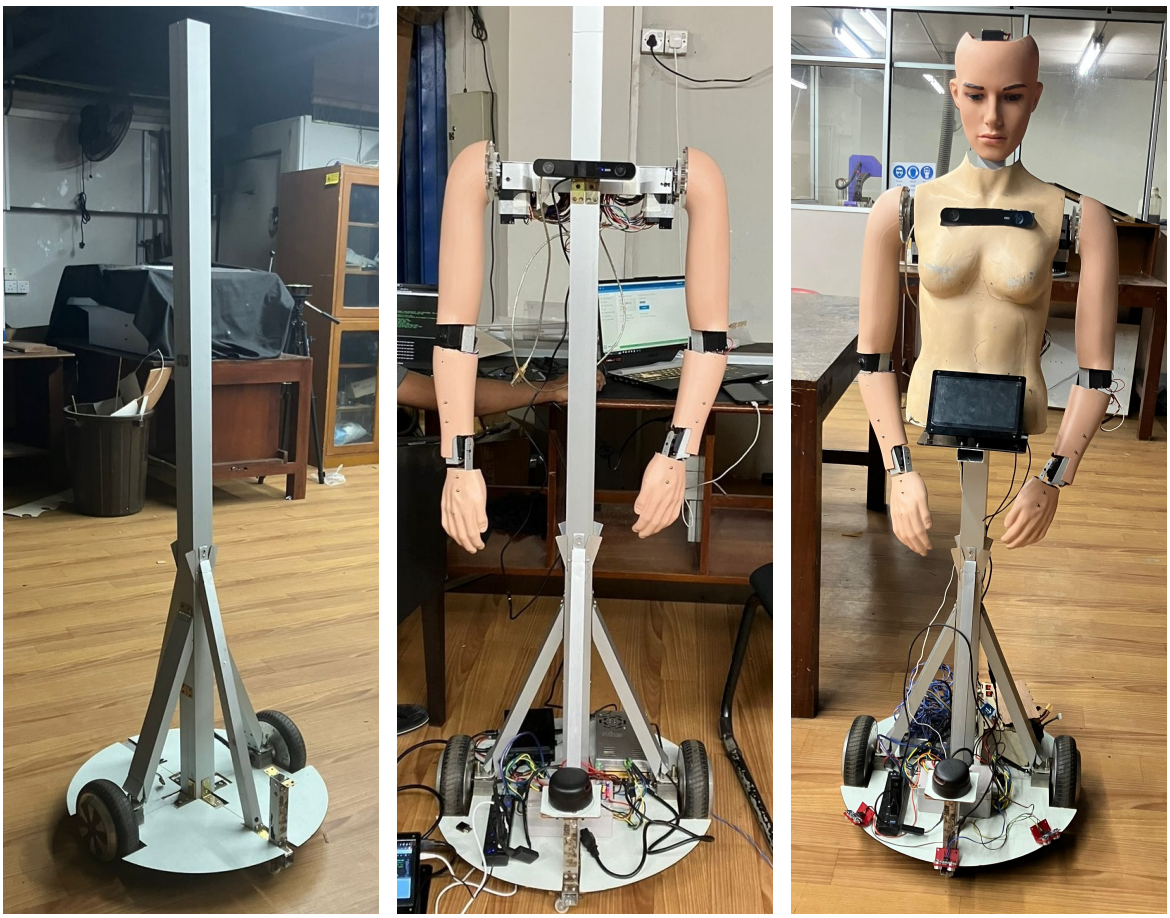
(a) Bottom platform (b) Bottom support

Figure 3.6: Implementing the bottom part of the platform

After implementing the bottom part successfully and the hand gesture testings, the ver-

tical bar of the robot is implemented. Steps are shown in **Figure 3.7**. Remarks about the current design:

- Previous design was totally relying on the mannequin's stability and shape.
- Current design has a strong support from inside and it not relying on the shape of the mannequin at all. The developer have more flexibility to choose the shape of the robot (physical appearance).



(a) Vertical bar (b) Arms (c) Fixing outer body

Figure 3.7: Phase two of the physical implementation

3.2.4 Stability

The robot needs to be stable when it is operating on a flat floor. Usually it is unlikely to break the stability on a flat surface. But for the safe side we should do calculations to make sure the robot is able to move on a surface that is angled at least 10°.

First, find an estimation for the center of gravity of the robot. Then calculate an upper bound for the location of the center of gravity of the robot, starting from the bottom of the robot (**Figure 3.8**). If the estimated center of gravity falls withing the range during a falling situation for a angled surface, the robot is stable enough for execution.

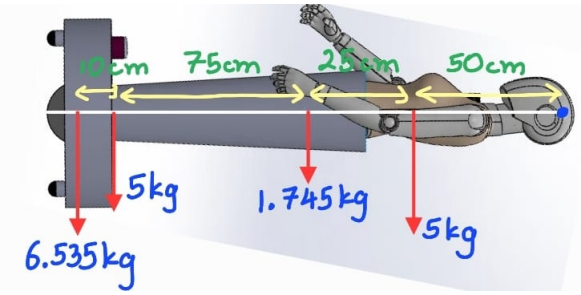
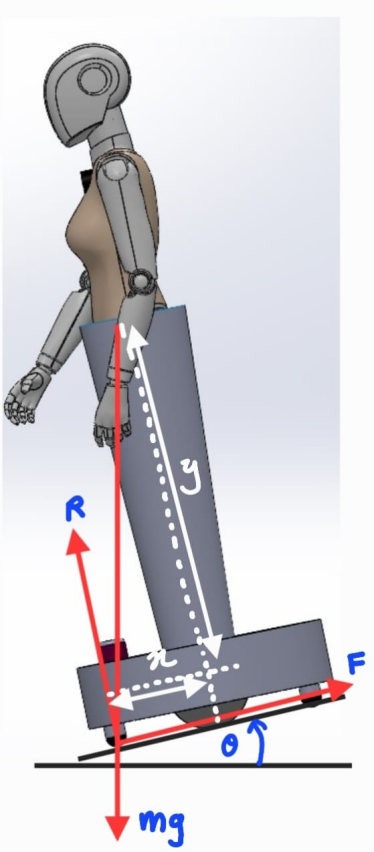
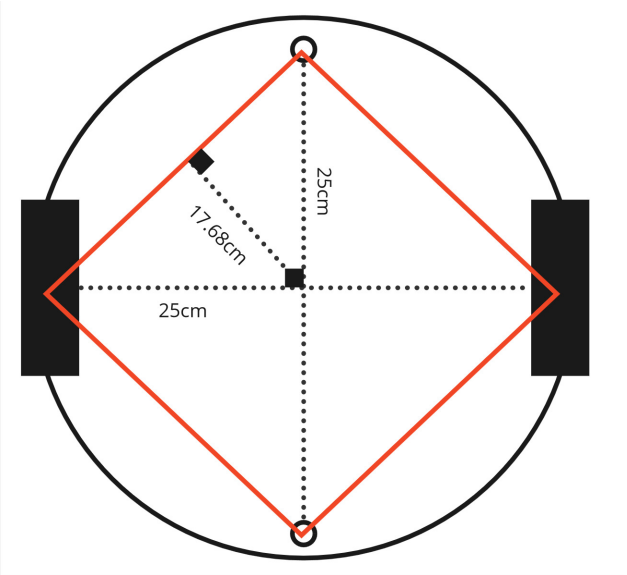


Figure 3.8: Estimating the location of center of gravity



(a) Forces acting on robot



(b) stability checking polygon (Bottom view of the platform)

Figure 3.9: Calculating the maximum height for center of gravity (Without flipping over)

The estimated maximum height for the location of center of gravity is obtained from following calculation. Free body diagram is shown in **Figure 3.9**.

$$\theta = 10^\circ \ , \qquad x = 17.68, \qquad y_{max} = \frac{17.68cm}{\tan 10^\circ} = 100.268cm$$

From the center of gravity estimation we get $y = 30.94cm$ Since $30.94cm < 100.268cm$, the robot will not flip over for 10° slope.

3.2.5 Required Motor Torque

To calculate the required motor torque, First need to consider the total weight of the robot. **Table 3.1** shows the rough estimation on the robot's total weight.

Table 3.1: Total mass of the robot

Component	Mass (kg)
Existing robot body	9.9
Weight of the newly added components	
Battery	
LiDAR	
ZED2 camera	
Secondary power supply	
Nvidia Jetson AGX Xavier board	5
Mobile platform (with motors)	10
Total	24.9

Now calculate the required forces to keep the robot moving. Since the mass is $24.9kg$, let's use a upper bound of $25kg$ for further calculations. The required forces will be, force required to roll F_{roll}, force required to climb up a slope of angle ($\alpha = 10°$) F_{slope} and force required to accelerate the robot to it's maximum velocity F_{acc}.

W_r = Weight of the robot (N)
m_r = Mass of the robot (kg)
v_{max} = Maximum velocity (ms^{-1})
t_d = Time takes to accelerate the robot to v_{max} (s)
r_w = Radius of a wheel (m)
c_{rf} = Coefficient of friction
k_{rf} = Torque multiplying factor considering the wastage for mechanical parts

$$F_{roll} = W_r \times c_{rf} = 25 \times 9.81 \times 0.42 = 103.005N \tag{3.1}$$

$$F_{slope} = W_r \times \sin\alpha = 25 \times 9.81 \times \sin 10 = 45.5872N \tag{3.2}$$

$$F_{acc} = \frac{m_r \times v_{max}}{t_d} = \frac{25 \times 1.42}{4} = 8.875N \tag{3.3}$$

$$F_{total} = F_{roll} + F_{slope} + F_{acc} = 157.4672N \tag{3.4}$$

$$Torque = F_{total} \times r_w \times k_{rf} = 157.4672 \times \frac{0.1651}{2} \times 1.15 = 14.94875Nm = 1.524kgm \tag{3.5}$$

Therefore, required torque for one wheel = 0.762kgm

3.3 Lower Level System

There is a low level processing element for the robot which is a micro controller. Selected micro controller development board for this implementation is Arduino Mega. Following are the required tasks need to be done using the micro controller.

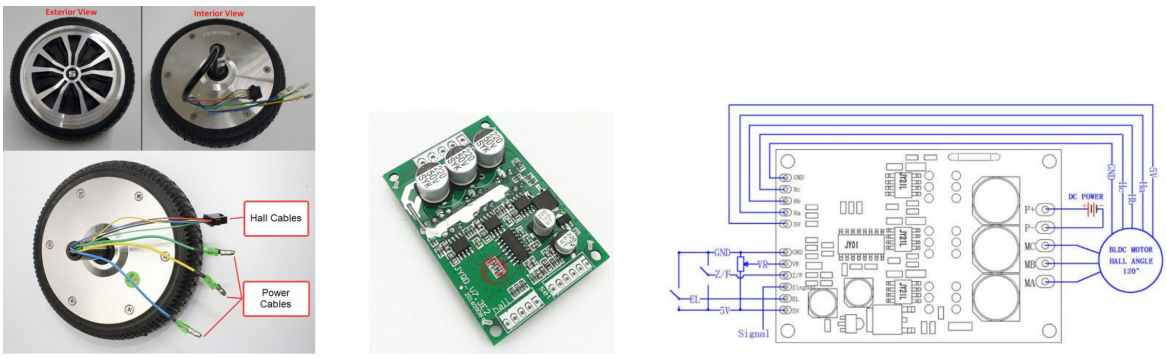
- Execute all the velocity calculations which were done for the robot's mobility. The velocities are for the two driving wheels of the robot.
- Send encoder data to the navigation stack.
- Turn the head of the robot using a servo motor.
- Sense cliffs around the robot and stop executing velocity commands which were received from the higher level controller.
- Do the power switching according the power state published from the higher level controller.

3.3.1 Connected Devices

The main processing unit for this level is selected considering the required number of actuating hardware interfaces. Following are the connected actuators and sensors for the lower level driving system.

- **Motor driver with wheel encoders**: The selected motors are hub wheel motors. They need to be controlled using special motor drivers dedicated for each motor. The hall sensor reading circuitry is also included with the selected motor driver[12] (JYQD-V7.3E). **Figure 3.10** shows used motors and motor driver.

  A custom Arduino library[13] is implemented for the hub motors. The library can be used to drive the hub wheel after configuring the input and output pins for the motor driver.



(a) Hub wheel (b) Motor driver (c) Connection diagram

Figure 3.10: Wheel and motor driver

- **Head servo motor**: The head of the robot is turned according the position of the user which interacts with it. The angle which required for the movement is provided from the higher level system, which is extracted using the camera images.

  A custom library[14] is used to control the robot's head because the existing servo driving library for Arduino is not capable of controlling the speed. Therefore, a new library is implemented to control the head smoothly.

- **Ultrasonic sensors**: These sensors are used to detect cliffs when the robot go near them. This feature adds additional safety for the robot. The detection of cliff must be done before a caster wheel of the robot goes over the cliff. Therefore, From this configuration we can check the maximum velocity that robot can take without falling over a cliff. The calculations are done under this **section**

- **Relays**: A power switching system is designed to change the powering method of the robot. Three relays are used in the design. According to the requirement from the top level, the relays will be switched.

3.3.2 Serial Communication

The communication between Arduino and Jetson is done using serial communication with a baud rate of 115200. The communication link is one way at a time. Therefore, always the first command comes from the higher level controller. After that, other relevant sensor data is sent back from the lower level system. **Figure 3.11** shows the communicating messages in a graphical illustration. Meanings of the terms are as follows.

- String sent from the Jetson side
  - **left_vel**: Velocity of the left wheel as a float
  - **right_vel**: Velocity of the right wheel as a float
  - **head_servo_angle**: Angle for the robot head servo as an integer
  - **power_state**: Power state an integer

- String sent from the Arduino side
  - **left_en_count**: Left wheel encoder count
  - **right_en_count**: Right wheel encoder count
  - **s1_distance**: Front ultrasonic sensor reading
  - **s2_distance**: Left ultrasonic sensor reading
  - **s3_distance**: Right ultrasonic sensor reading

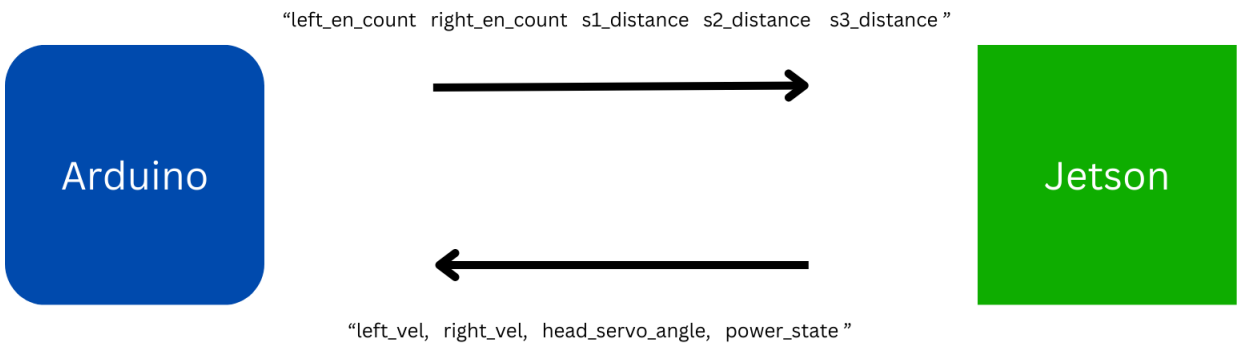


Figure 3.11: Communication with Jetson AGX Xavier

3.4 Framework: Robot Operating System(ROS)

The project has adopted the Robot Operating System (ROS) framework, specifically focusing on ROS2 Foxy Fitzroy[15] (**Figure 3.12**). This decision was influenced by hardware compatibility requirements, particularly with the Nvidia Jetson AGX Xavier, which supports Ubuntu 20. Among the ROS2 distributions compatible with Ubuntu 20, both Foxy Fitzroy[15] and Galactic Geochelone are available; however, both have reached their end of life. Foxy Fitzroy[15] is the Long-Term Support (LTS) version, making it a preferable choice.

While ROS1 Noetic, which also supports Ubuntu 20, remains an active option, we opted for ROS2 Foxy Fitzroy[15] for strategic reasons to ensure the project's continuity. ROS1 Noetic is the final version in the ROS1 series, and transitioning from ROS1 to ROS2 involves significant challenges. Starting with ROS2 Foxy Fitzroy[15] positions the project for a smoother transition to future ROS2 distributions.

Figure 3.12: ROS2 Foxy Fitzroy

3.5 Mobility and Environmental Perception

This section centers on the fundamental capabilities of the mobile robot receptionist to navigate autonomously, localize itself within its environment, and interact with objects and individuals. It encompasses the technologies and algorithms employed for mapping unknown

environments, achieving precise localization, and detecting and avoiding both dynamic and static obstacles. Additionally, it covers the integration of sensor data for real-time environmental perception. **Figure 3.13** shows the overview of the navigation system.

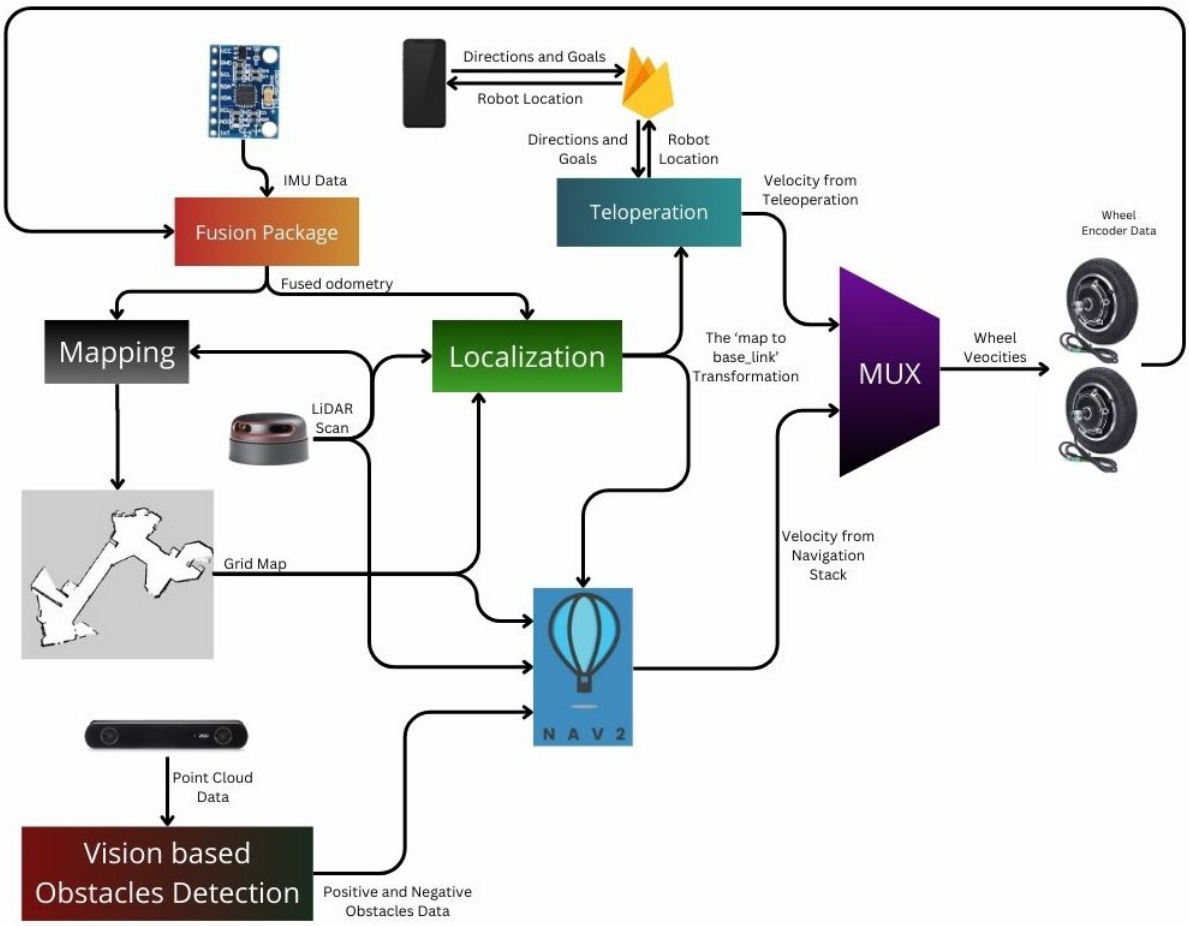


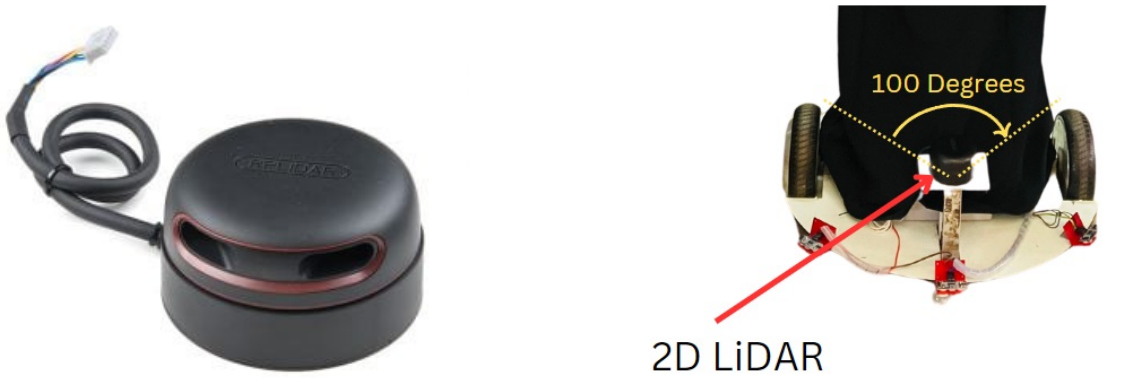
Figure 3.13: Overview of the navigation system

3.5.1 Sensors

Effective localization and mapping are achieved through precise sensory data, facilitated by an integrated suite of sensors that collect various types of environmental information. This setup enables the mobile robot receptionist to navigate accurately and respond to dynamic changes within its environment.

- **2D LiDAR**: The RPLIDAR A2M6 (**Figure 3.14**) is a 360-degree 2D LiDAR sensor developed by SLAMTEC, designed primarily for indoor robotic applications such as autonomous navigation, obstacle avoidance, and environment monitoring. It offers comprehensive coverage with no blind spots due to its ability to perform 360-degree scans. With a range of up to 18 meters and a high resolution of less than 0.5 millimeters. Due to the structure of the robot and the attire of the robot receptionist, some portions

of the LiDAR scan are obstructed, effectively limiting the sensor's field of view. This results in certain areas within the robot's immediate environment being less visible or completely unseen by the LiDAR system. These areas are neglected during mapping and localization processes.



(a) External view (b) Limiting the LiDARS's field of view

Figure 3.14: RPLiDAR A2M6

- **Wheel Encoders**: Complementing the LiDAR data, odometry information is gathered from wheel encoders. These sensors measure the rotation of the robot's wheels, providing data on the distance traveled and speed, which assists in estimating the robot's movement over time.
- **IMU**: A 6-axis MPU-6050 IMU (**Figure 3.15**) is employed to understand the robot's orientation and acceleration dynamics. This sensor helps in correcting any positional drifts estimated from wheel encoders and supports accurate localization by providing additional data on the robot's tilt and acceleration. The accelerometer range and the gyroscope range are ±2g and ±250°/sec respectively.

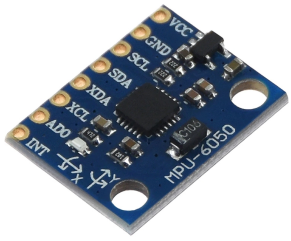


Figure 3.15: MPU-6050

- **Stereo Camera**: The ZED 2 stereo camera (**Figure 3.16**) is an advanced 3D vision sensor developed by Stereolabs, designed for depth sensing, motion tracking, and

spatial understanding. Equipped with dual cameras that mimic human stereoscopic vision, it provides high-resolution 3D images and environmental mapping. It supports a depth range of 0.3 to 20 meters and delivers high resolutions in its output: 2K at 15 FPS with a side-by-side resolution of 2x(2208x1242), and lower resolutions at varying frame rates, ensuring detailed and versatile visual data capture for depth analysis and other computations.The ZED 2 has a 110°(H) x 70°(V) x 120°(D) field of view, with H, V, and D standing for Horizontal, Vertical, and Diagonal.

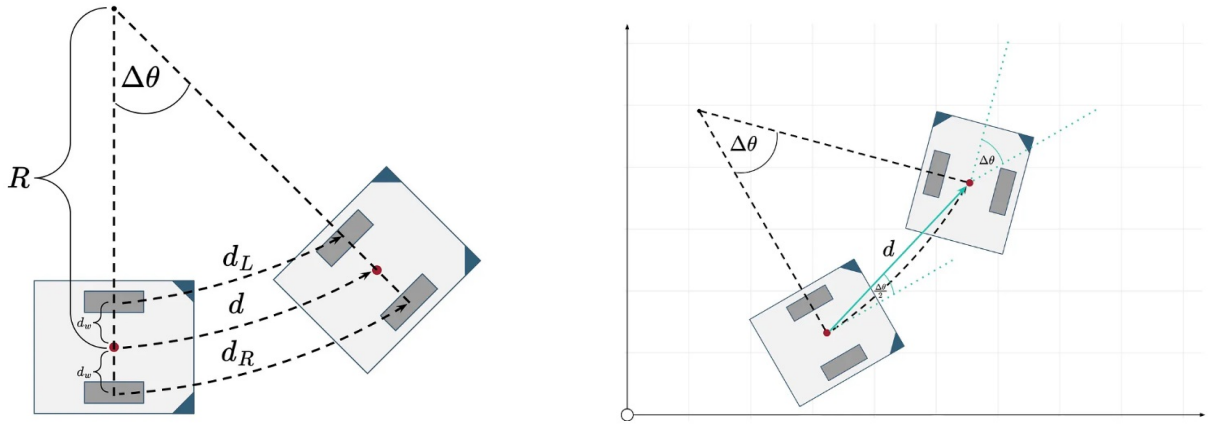
Figure 3.16: ZED 2 stereo camera

- **Ultrasonic Sensors**: HC-SR04 (**Figure 3.17**) is an ultrasonic sensor which emits sound waves and measure the time taken for the echoes to return. This provides a range from 2 cm to 400 cm.

Figure 3.17: HC-SR04 ultrasonic sensor

3.5.2 Wheel Odometry Model

The odometry model calculates the robot's position using the rotation data from the encoders attached to its wheels. The equations below demonstrate how the x,y coordinates of the robot's base footprint frame are calculated with respect to the odom frame. The geometric model for the differential driving method is shown in **Figure 3.18**



(a) (b)

Figure 3.18: Differential-drive geometric model

d_L = distance traveled by the left wheel
d_R = distance traveled by the right wheel
d_w = distace between reference point and the wheel
d = distance traveled by the reference point
$\Delta\theta$ = change in the angle of rotation
R = radius of the curve containing the reference point

d_l and d_r can be directly measured using encoders, while d_w can be measured using a ruler. To calculate other variables, geometric relationships can be derived.

$$R = d_L(\frac{2d_w}{d_R - d_L}) + d_w \tag{3.6}$$

$$d = \frac{d_L + d_R}{2} \tag{3.7}$$

$$\Delta\theta = \frac{d_R - d_L}{2d_w} \tag{3.8}$$

$$\Delta x = \frac{d_L + d_R}{2}\cos(\theta_{t-1} + \frac{\Delta\theta}{2}) \tag{3.9}$$

$$\Delta y = \frac{d_L + d_R}{2}\sin(\theta_{t-1} + \frac{\Delta\theta}{2}) \tag{3.10}$$

$$x_t = x_{t-1} + \frac{d_L + d_R}{2}\cos(\theta_{t-1} + \frac{\Delta\theta}{2}) \tag{3.11}$$

$$y_t = y_{t-1} + \frac{d_L + d_R}{2}\sin(\theta_{t-1} + \frac{\Delta\theta}{2}) \tag{3.12}$$

3.5.3 Fusion of Odometry

In mobile robotics, IMU data with wheel encoder readings enhances odometry estimates significantly. This fusion process combines the strengths of each sensor type to provide a more robust and accurate localization system, particularly in environments where either sensor alone might underperform due to specific limitations.

- **Wheel Encoders**: These devices measure the rotation of the wheels to estimate the robot's displacement. They are directly tied to the mechanical movement but can be affected by slippage or uneven terrain.
- **IMU**: This sensor measures linear acceleration and angular velocity. It helps track orientation, tilt, and sudden shifts in movement, which are not directly measurable through encoders.

Data synchronization is essential in ensuring that data from the IMU and the wheel encoders are accurately aligned for fusion, typically achieved through time-stamping each data set at the source and aligning these stamps during processing. Advanced filtering techniques such as the Kalman Filter(KF) or the Extended Kalman Filter (EKF) are utilized to integrate these data streams effectively, which aids in reducing noise and compensating for sensor biases. By fusing the IMU and encoder data, the system can effectively minimize common errors, such as the susceptibility of wheels to slip and skid on uneven or slippery surfaces, and the tendency for IMUs to drift over time. This integrated approach ensures a balance where the weaknesses of one sensor are compensated by the strengths of another, enhancing overall system reliability and accuracy.

3.5.4 Mapping with SLAM Toolbox

The SLAM Toolbox[16] is a powerful framework used in robotics to create maps of unknown environments while simultaneously localizing the robot within those maps. By integrating sensor data such as laser scans and odometry, the SLAM Toolbox[16] enables the robot to build accurate and detailed representations of its surroundings in real-time. The SLAM Toolbox[16] employs a probabilistic approach to map creation, utilizing techniques such as feature extraction, loop closure detection, and optimization algorithms to continuously refine the map as the robot explores its environment.

3.5.5 Localization with AMCL

Adaptive Monte Carlo Localization (AMCL) is a probabilistic method that uses a particle filter to estimate a robot's 2D position by representing the robot's pose with a collection of particles. Each particle corresponds to a potential pose of the robot. With each new time step, the robot gathers sensor data and matches it against a pre-existing map to evaluate the likelihood of each particle's pose. Particles that are less likely are eliminated, and those with higher likelihoods are used in a resampling process to create a new particle set for the subsequent time step. This resampling process helps maintain a particle distribution

that closely approximates the robot's actual pose, compensating for uncertainties in movement and limited visibility of the environment. **Figure 3.19** shows the block diagram for localization.

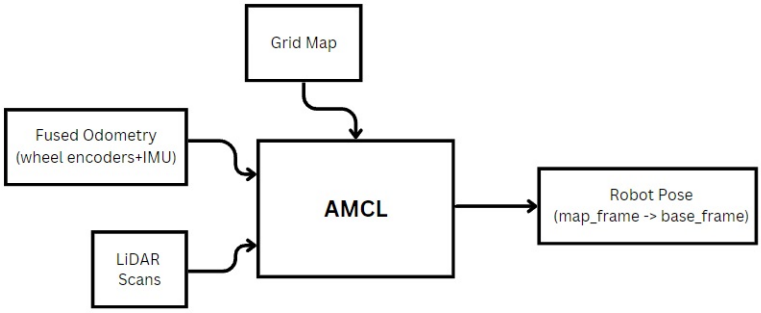


Figure 3.19: Block diagram of localization

3.5.6 Obstacles Detection

Positive obstacles refer to any objects or features that protrude upwards from the ground level and pose a physical barrier to a robot's movement. These can include walls, people, furniture, and more. Negative obstacles involve any kind of drop-off or lower areas that can pose risks of falling or tipping. These can include cliffs, ditches, pits, and uneven surfaces, all of which require careful navigation to ensure safety.

Obstacle detection is a critical component of the mobile robot receptionist project, ensuring safe navigation and interaction within its operational environment. This section explores how the system identifies both positive obstacles and negative obstacles using an array of sensors, including 2D LiDAR, the ZED 2 stereo camera, and ultrasonic sensors.

2D LiDAR

2D LiDAR serves as the primary sensor during the mapping and localization processes, providing precise distance measurements by emitting laser beams and detecting their reflections off surfaces. It is highly effective in identifying both dynamic and static positive obstacles within the robot's surroundings.

ZED 2 Stereo Camera

Because LiDAR can only detect obstacles within a specific plane, the ZED2 camera acts as a secondary method for obstacle detection, especially useful during navigation. This camera is crucial not only for detecting positive but also negative obstacles. By generating a depth map and 3D point cloud from stereo images, it provides additional details about the shape and size of obstacles, thereby enhancing the robot's ability to navigate around them. The 3D point cloud data received from the ZED2 is first filtered using a voxel grid filter to reduce noise. Subsequently, the data is processed to identify positive and negative obstacles and then converted into the specific **LaserScan** message type in ROS.

The corresponding ground level points of a staircase are detected as illustrated in the **Figure 3.20** below.

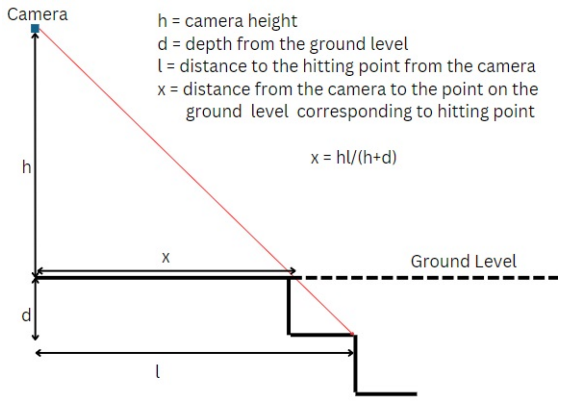


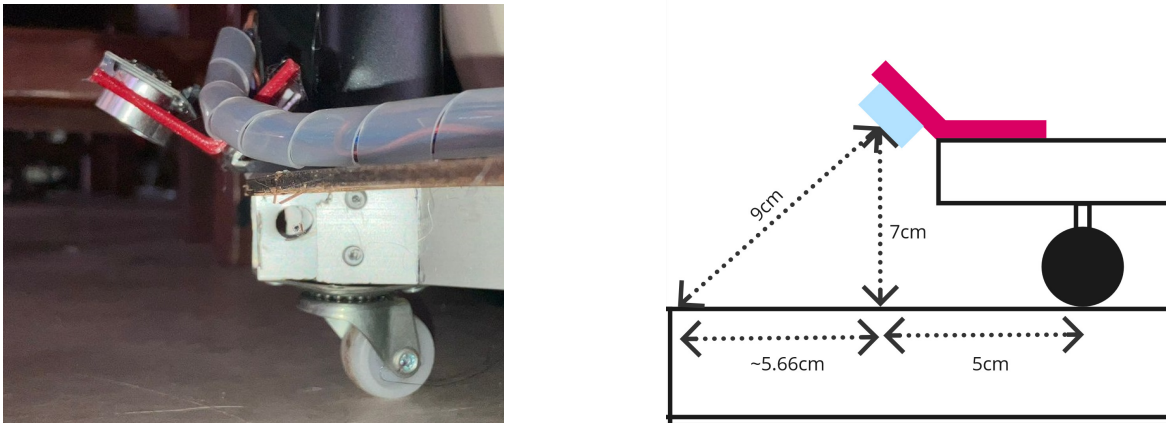
Figure 3.20: Downward staircase detection

Ultrasonic Sensors

These sensors are specifically used to detect negative obstacles when the ZED2 camera may fail to do so. Positioned to look downwards, these sensors trigger an immediate halt to the robot's movements, managed by a low-level driving system (Arduino Mega), which takes precedence over commands from the main processor (Jetson AGX Xavier) when a drop is detected. This safety mechanism ensures that the robot stops immediately and remains stationary, requiring manual intervention to reset and move away from the hazard before it can resume navigation. This approach provides a critical safety layer, ensuring the robot does not proceed over an edge potentially resulting in damage, thereby enhancing its operational integrity and safety in complex environments.

The robot is equipped with three ultrasonic sensors. Each ultrasonic reading needs 50ms of time. All sensors are read in one loop. Therefore, it takes $50 \times 3 = 150$ms to update all the readings.

Let's assume it takes additional 50ms to execute serial readings and other actuation. If we assume the robot moves at it's maximum speed $v_{max}\ ms^{-1}$, The robot must be able to detect a cliff and stop within the 50ms + 150ms = 200ms of time. **Figure 3.21** shows the measurements for the front caster wheel with the front down-looking ultrasonic sensor.



(a) Actual sensor placement (b) Dimensions

Figure 3.21: Ultrasonic sensor placement

Therefore, The maximum speed for the robot can be calculated as follows.

$$v_{max} = \frac{(5cm + 5.66cm)}{200ms} = 0.533ms^{-1} \tag{3.13}$$

3.5.7 Autonomous Navigation with Navigation2

Overview

Navigation2(NAV2)[17] is an advanced navigation system designed specifically for ROS2, the Robot Operating System. It serves as the successor to the original Navigation Stack used in ROS1, incorporating significant improvements to adapt to the changes and capabilities of ROS2. NAV2 is built to support a wider variety of robots, including both small and large mobile robots in dynamic and complex environments.

NAV2[17] leverages ROS2 features such as real-time support, improved security, and better resource management. It includes modules for load, serve, and store maps, localization, path planning, path following, and obstacle avoidance. The modular architecture of NAV2[17] allows for easy customization and integration with different types of sensors and robotic platforms. This makes it a versatile and powerful tool for developers looking to implement sophisticated navigation capabilities in robotic applications ranging from industrial automation to service robots in both indoor and outdoor settings. **Figure 3.22** shows the block diagram of NAV2[17].

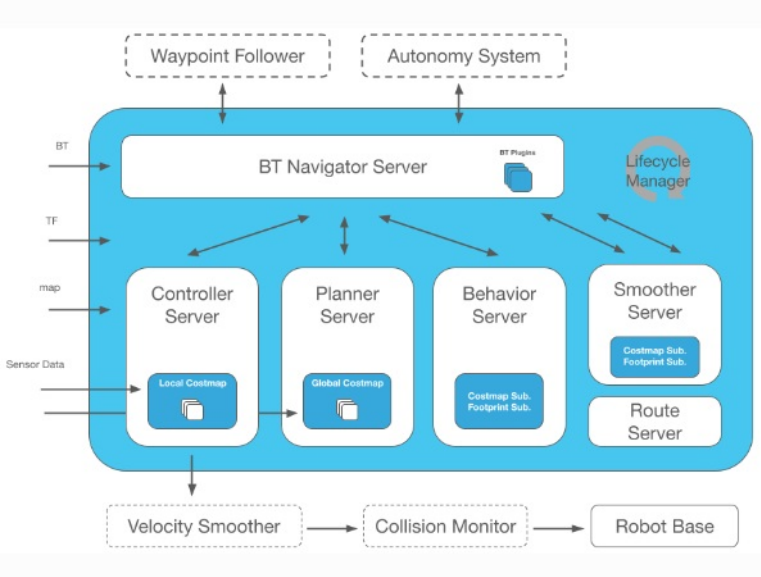


Figure 3.22: Block diagram of NAV2

Plugins Selection

In the Navigation2 (Nav2[17]) system for ROS2, plugins play a crucial role in customizing and extending the navigation stack to meet specific robotic application requirements. The system architecture is designed to be modular, with each component potentially using different plugins that implement various functionalities. Here are some key types of plugins utilized in Nav2[17] for this project:

- **Planner Plugins**: These plugins are responsible for generating a path for the robot to follow from its starting point to the destination. Common examples include the NavFnPlanner and SmacPlanner, which provide different approaches to path planning such as A* and other search-based algorithms. For our project, NavFn planner is used with Dijkstra's expansion.

- **Controller Plugins**: Once a path is planned, controller plugins are used to follow the path. These plugins determine how the robot moves along the planned path, handling the dynamics and kinematics of the robot to ensure smooth navigation. For our project, the DWBLocalPlanner (Dynamic Window Approach) is used which handles obstacle avoidance in real-time as the robot follows the path.

- **Recovery Plugins**: These plugins are invoked when the robot encounters a problem, such as being stuck or deviating from the path. Recovery plugins attempt to resolve these issues and return the robot to normal operation. For our project, recovery behaviors include 'spin', 'wait' and 'back_up'.

- **Costmap Plugins**: Costmap is a grid that represents the space around the robot with costs assigned to each cell based on the presence of obstacles. In the project, two obstacle layers, one for the LiDAR and another for the ZED 2 camera,are implemented in both the global costmap and local costmap. Additionally, an inflation layer is utilized

in both cost maps to ensure safe navigation clearance around obstacles. The obstacle layer, which incorporates the global costmap, completes the robot's understanding of the environment by providing a fixed, unchanging map. This configuration enhances the robot's ability to plan and adjust its path effectively, considering both permanent fixtures in the environment and dynamic changes

- **Goal Checker Plugins**: These plugins are responsible for determining when the goal has been reached. They can define custom conditions under which a goal is considered achieved, allowing for precise control over when a navigation task is completed. 'SimpleGoalChecker' is used as the goal checker plugin in our project.

- **Progress Checker Plugins**: Progress checkers monitor the robot's progress towards the goal. If the robot isn't making the expected progress (e.g., if it's stuck), these plugins can trigger recovery behaviors to correct the course. 'SimpleProgressChecker' is used as the progress checker plugin in our project.

Tuning Parameters

Tuning navigation parameters in ROS2, particularly when using the Nav2[17] stack, is a critical aspect of achieving optimal performance in robot navigation. All the navigation parameters can be found **here.**[18]

3.6 Face Re-identification

One of the main feature of the reception robot is to remember people who already interacted with the robot. For this procedure we have to incorporate a face recognition model with a data base which stores information about each person. Following are the key components that need to be included in the face recognition procedure.

- **Face detection**: To do a face identification, the algorithm should get a face in the image frame. In this implementation face detection is done from the OpenCV face detection using using Haar Feature-based Cascade Classifiers[19].

- **Feature extraction**: The feature extraction for a face can be done using a face recognition model which is trained on human faces. ArcFace[9] is used as the base model for the face recognition feature.

When a new user is interacting with the robot, it should remember that person in the next time. This procedure should be able to compare the previous data with new data and save new data when a new user comes. All of these actions can be achieved through the library available for Python called Deepface[20]. Following flow chart (**Figure 3.23**) shows the basic flow of the facial re-identification process.

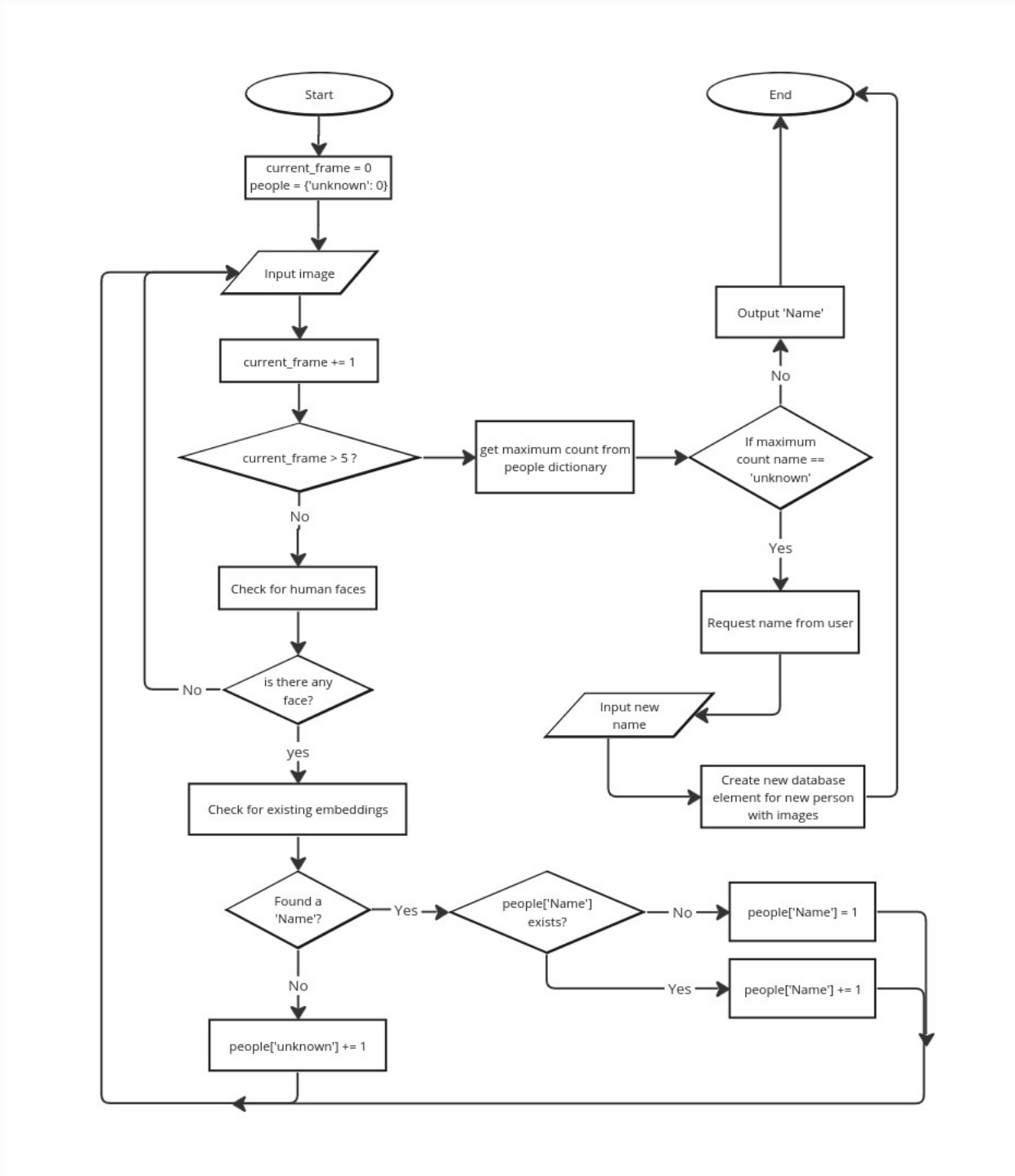


Figure 3.23: Basic flow for face re-identification

The used model for the face detection uses a different method to calculate similarity between the images than the usual use cases. Instead of euclidean distance, ArcFace calculates the geodesic distance on the hyper sphere. **Figure 3.24** illustrates this measurement graphically.

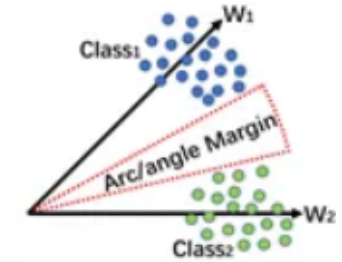


Figure 3.24: Distance measuring method in ArcFace

Simply it uses the cosine distance between the two embedding vectors generated from the model to obtain a loss function during training process. The paper on ArcFace[9] describes how this distance is used for training process. In this robot's implementation, the embedding of the image will be obtained from the trained model and measure the similarity(distance) between images using cosine distance to identify the person interacts with the robot.

3.7 Gestures

The gestures of the robot will be fall into two main categories, namely, hand gestures and neck movement.

3.7.1 Hand Gestures

Design Specifications and Requirements

- Increase DoF
- Add handshake gesture
- Improve realisticness of the the 'Ayubowan' gesture and showing direction gestures

Hardware Implementation

- Added wrist joints.It increased DoF from 4 to 6
- Orientation of elbow joint to get a realistic 'Ayubowan' gesture.

Since 2 servos were added, the masses of arms were changed. So re-calculated all the torques to check whether the existing servos were suitable. **Figure 3.25** is referred for the required torque calculations.

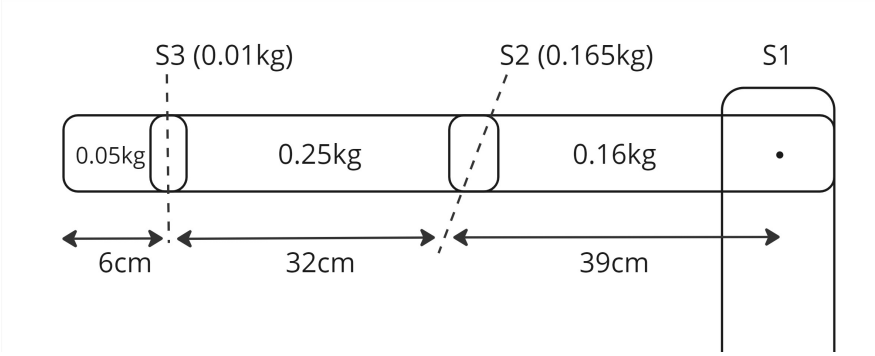


Figure 3.25: Hands dimensions and masses

S1 - Shoulder Joint
Required torque = 33 kgcm
Used Servo specs,

- Model = CM-785HB
- Torque = 77 kgcm
- Load = 42%

S2 - Elbow Joint
Required torque = 6 kgcm
Used Servo specs,

- Model = SpringRC SR811
- Torque = 6 kgcm
- Load = 23%

So we concluded that we can use same servos for the elbows and shoulders.

S3 - Wrist Joint
Required torque = 0.15 kgcm
Used Servo specs,

- Model = MG90S
- Torque = 2 kgcm
- Load = 7.5%

Firmware Implementation

The previous group had used a Micro Maestro 6-Channel USB Servo Controller from Polulu for the servo controlling part. But at the initial stage of the project we had planned to implement a biceps joint as well. Then we need altogether 8 DoF. Since the servo driver was for 6 channels without using that, we tried to run servos from the microcontroller by using available libraries for controlling servos' speed. However there was an issue with the smoothness because those libraries did not contain a parameter to control the smoothness. Then only parameter available was speed. So some speeds we observe discontinued motion. So we implemented our own servo library which had 2 parameters called delay and step rather than having one parameter as speed (speed = step/delay)

Working principal of the implemented servo library

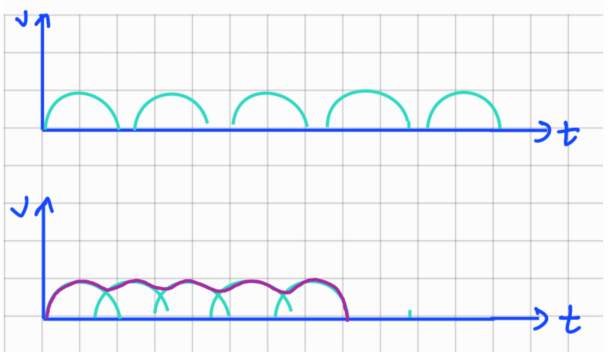


Figure 3.26: Servo library speed controlling mechanism

Normal servos do not contain speed input. The only input is a PWM signal which is proportional to the target angle. Since internally it has PID controller, it do different angle changes in different maximum speeds. Taking advantage of this, instead of directly giving the target angle we divide the total required angle difference to a set of steps and give as sub-targets with delay. By adjusting the delay we can minimize the discontinuity as shown in **Figure 3.26**. The library was implemented in a fully non-blocking way using timers then we can use that same micro-controller for other tasks as well.

Due to physical implementation limitations with the available hardware resources we had to remove the biceps joints from our design. Then we fixed that elbow joint (**Figure 3.27**) axis in such a way that we can use it for both handshake and 'Ayubowan' gestures.

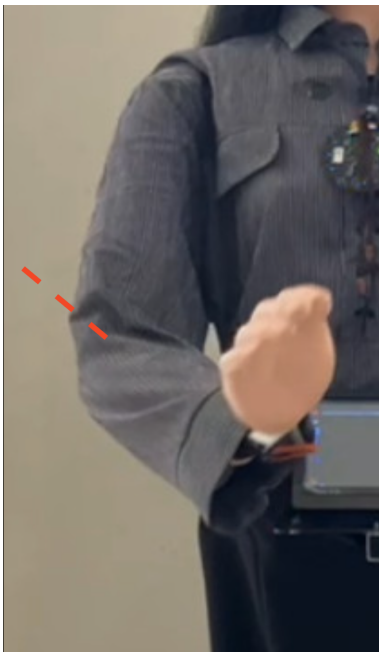


Figure 3.27: Elbow joint axis

Then we just only need to control 6 DoF. So we decided to use the available Micro Maestro 6-Channel USB Servo Controller for the final implementation of the hand gestures.

Micro Maestro 6-Channel USB Servo Controller[21]

- Three control methods: USB, TTL (5 V) serial, and internal scripting
- 0.25 μs output pulse width resolution (corresponds to approximately 0.025° for a typical servo, which is beyond what the servo could resolve)
- Pulse rate configurable from 33 Hz to 100 Hz
- Wide pulse range of 64 μs to 3280 μs
- Individual speed and acceleration control for each channel
- Channels can be optionally configured to go to a specified position or turn off on startup or error
- Channels can also be used as general-purpose digital outputs or analog inputs
- A simple scripting language lets you program the controller to perform complex actions even after its USB and serial connections are removed

A free configuration and control program called Maestro Control Center was available for this driver. It can be used to configure (**Figure 3.28**) the speed and acceleration of each servo.

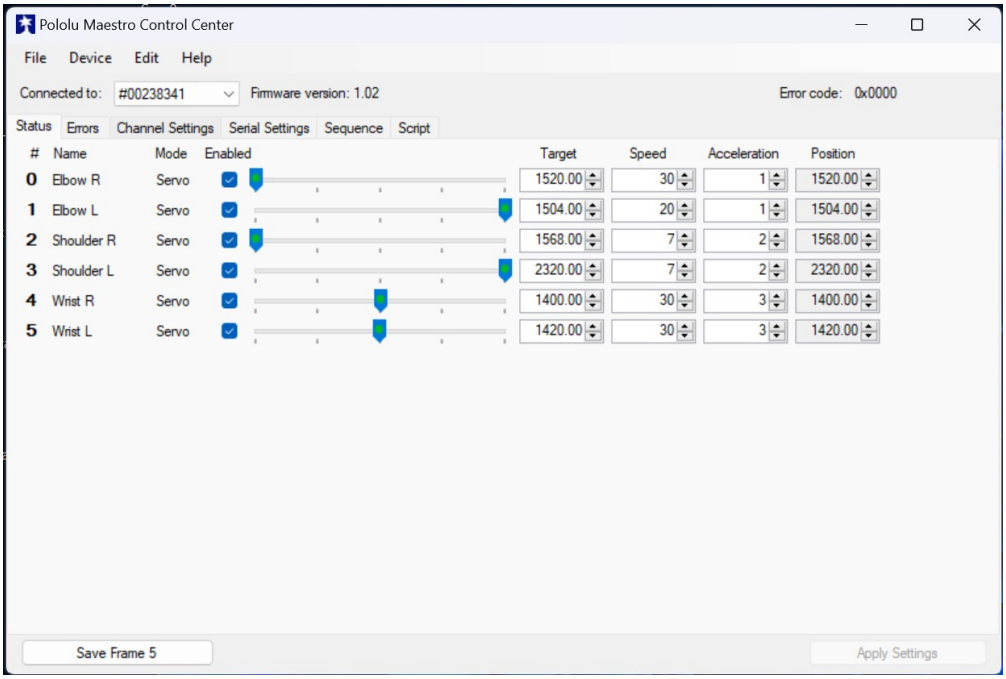


Figure 3.28: Maestro control center - configuration

Also, it supports recording frames which are sets of angles of each servo. A set of frames can be saved as a sequence. (**Figure 3.29**)

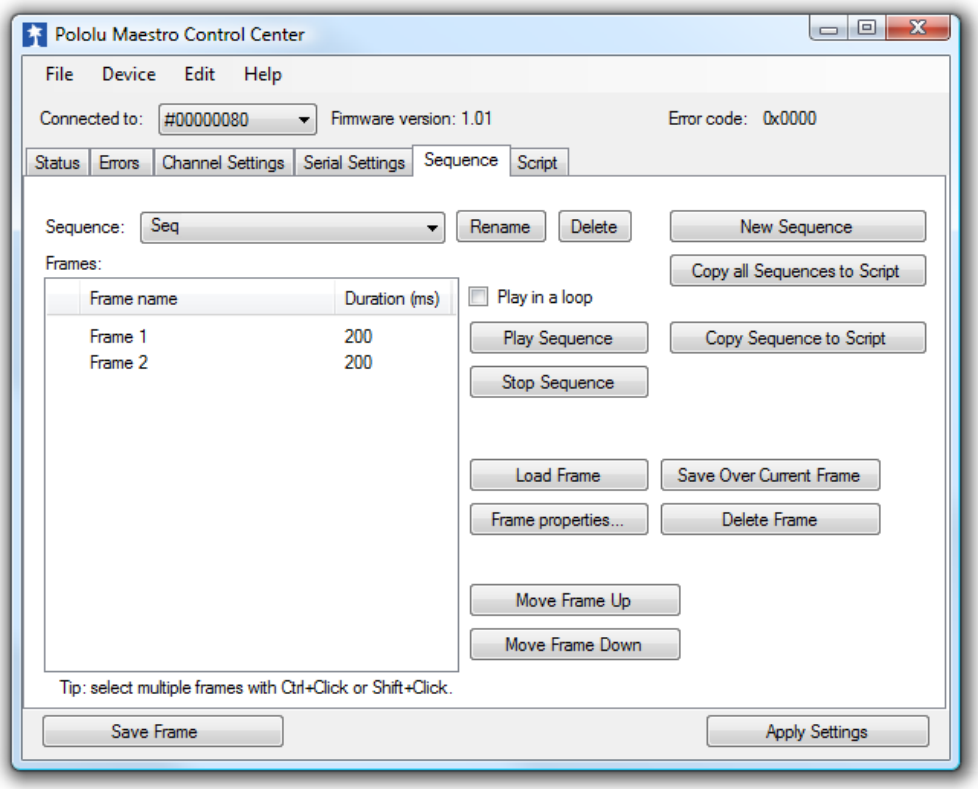


Figure 3.29: Maestro control center - sequence planner

All the sequences can be loaded into a script (**Figure 3.30**) and that can be stored in the servo driver's memory. Then by using a serial command, those sequences can be executed. Using that feature we saved each gesture as a sequence. Then by sending a serial command to the servo driver, we could execute the relevant gesture.

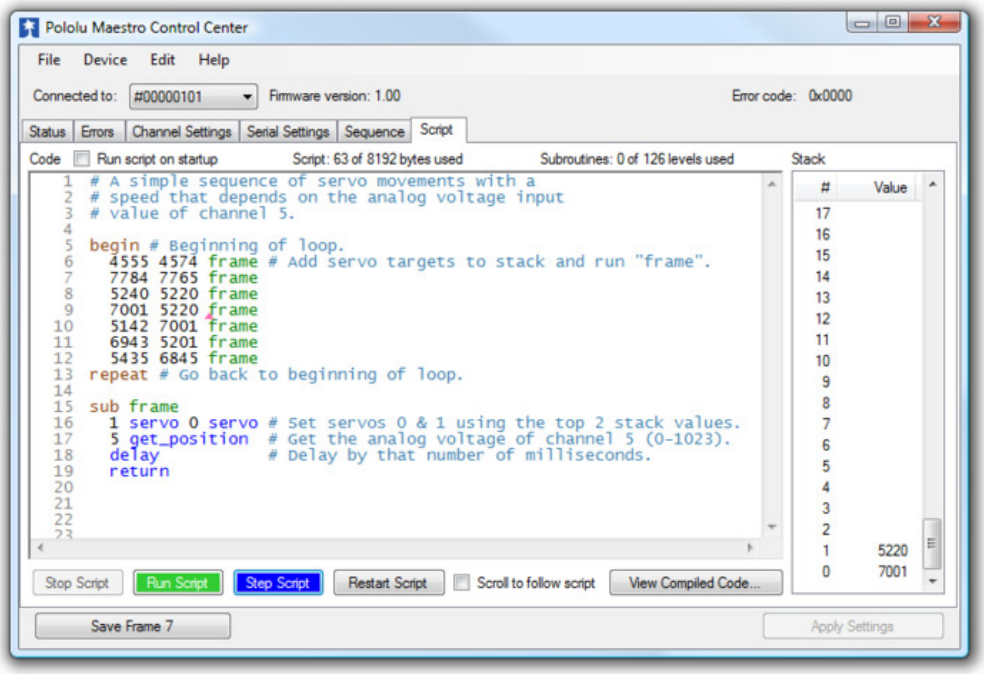


Figure 3.30: Maestro control center - script

Even though the servo driver itself had a USB-TTL converter we had to use a separate FTDI USB-TTL converter due to device conflict occurred only when we tried to connect that servo driver through a USB hub. We had to use HUB because the system had several USB components. To send the serial commands from the system we used the PySerial library.

3.7.2 Neck Gesture

The neck could be rotated to the user's direction by detecting user's angle using the camera. To control the speed of the neck joint servo our own implemented library[14] was used.

3.8 Conversation Handling

The conversation handling pipeline of the smart mobile robot receptionist is designed to facilitate natural interaction between the robot and users. It consists of five interconnected sub-processes: real-time voice capturing, speech-to-text conversion, natural language understanding and generation, text-to-speech conversion, and user's verbal input classification. Each sub-process plays a crucial role in transforming spoken language into actionable information for the robot and generating appropriate responses back to the user. Since we aimed to implement the whole conversation handling processes locally on the robot's processing unit (single board computer) Jetson AGX Xavier. We utilized machine learning models and other algorithms that can be implemented on the Jetson AGX Xavier. **Figure 3.31** shows the main conversation pipeline.

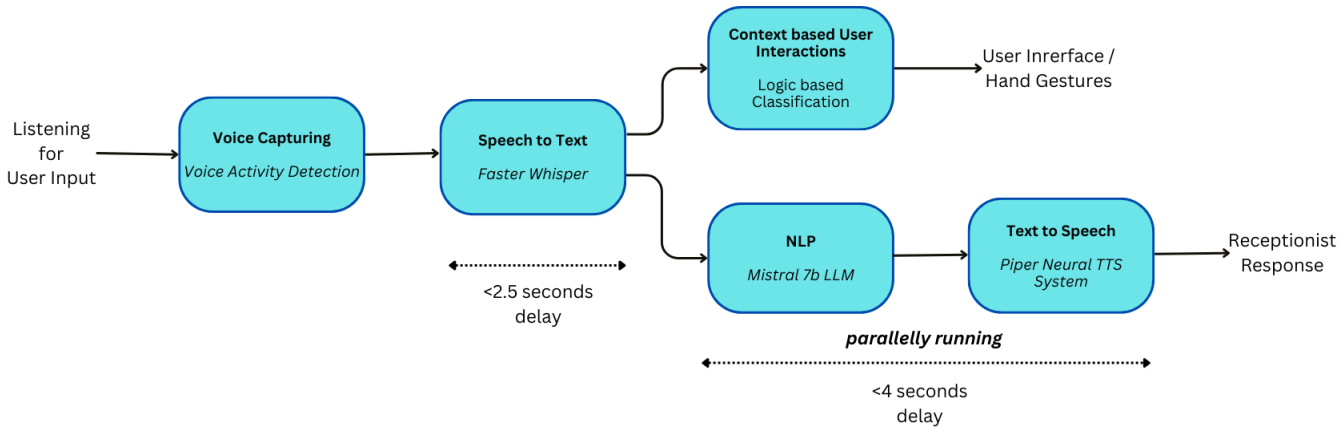


Figure 3.31: Conversation pipeline

3.8.1 Voice Capturing

In our smart mobile robot receptionist system, maintaining clear communication between the user and the robot is important, even with a distance of 2 to 3 feet between them during interaction. To ensure that the microphone effectively captures the user's voice from this distance, we use the **Respeaker Mic Array v2.0**[22] (**Figure 3.32**) far-field microphone array. This specialized hardware enables clear and accurate audio capture, essential for converting the user's spoken input into text.

However, to feed only voice-specific audio clips into the speech-to-text model, it's vitally important to accurately identify when the user begins and ends speaking. We achieve this by implementing a mechanism to capture audio clips containing solely the user's voice. using the **WebRTC Voice Activity Detection**[23] library, we can identify whether a given short duration of audio corresponds to human speech, allowing us to extract voice-only segments for further processing. This approach ensures that our system effectively captures and processes the user's verbal commands, facilitating seamless interaction between the user and the robot.

Respeaker Mic Array v2.0

Figure 3.32: Respeaker mic array v2.0

We utilize the Respeaker Mic Array v2.0[22] (**Figure 3.32**) as our audio-capturing device. This device boasts four omni-directional microphones, enabling it to pick up voices from various angles. Its far-field capability allows it to detect voices up to 4 meters away, ensuring clear communication even across larger spaces. Additionally, the device comes equipped with a built-in XMOS-XVF 3000 audio processor, which enhances audio quality by providing noise suppression and acoustic echo cancellation functionalities. These features collectively contribute to improving the accuracy and clarity of voice capture within our smart mobile robot receptionist system.

WebRTC Voice Activity Detector

WebRTC Voice Activity Detector[23] library utilizes a Gaussian Mixture Model (GMM) to identify voice in audio streams. This works in stages: first, features like Mel-Frequency Cepstral Coefficients (MFCCs), Spectral Energy, and Zero-Crossing Rate are extracted from the audio. A pre-trained GMM, built on a massive dataset of speech and non-speech examples, then analyzes these features. The GMM essentially compares how well the features match the distributions it learned for speech and non-speech datasets. Based on this comparison, the VAD calculates the likelihood of speech being present. By employing a threshold, the VAD ultimately decides whether the audio clip contains speech or not. While not perfect, this probabilistic approach based on GMMs allows WebRTC VAD to effectively distinguish between human voice and background noise or other audio sources.

This library operates through a Voice Activity Detection (VAD) constructor that requires two arguments. The first specifies the audio sample rate, ensuring compatibility with the chosen format. The sampling rates, the library supports are 8kHz, 16kHz, 32kHz, ans 48kHz. The second argument defines the VAD level, acting as a sensitivity control. A level of 0 prioritizes capturing all potential speech, even if accompanied by background noise. Conversely, a level of 3 prioritizes eliminating noise but might risk missing quieter speech segments. The VAD object itself offers a process method that analyzes 10ms, 20ms, or 30ms audio blocks (depending on the sample rate) and returns a binary output: true if speech is detected and false otherwise. This process facilitates the segmentation of the captured audio stream, isolating user speech for optimal speech-to-text conversion in the next step of the conversation handling pipeline.

Real-time Voice Capturing

Figure 3.33 represents the real-time voice capturing flowchart. Many thresholds and parameters within our system's voice-capturing workflow were established through many voice-capturing tests. These tests were instrumental in fine-tuning the system's performance to ensure smooth and accurate voice capturing. By systematically adjusting various thresholds and parameters based on the outcomes of these tests, we were able to optimize the system's ability to capture voice inputs effectively. This iterative process allowed us to strike a balance between sensitivity and specificity, resulting in a robust voice-capturing mechanism capable of delivering reliable performance across various user interactions.

1. **20ms audio data block** - We opted for 20ms audio blocks in our implementation of the voice capturing, a decision grounded in the WebRTCVAD[23] library's native

support for 10ms, 20ms, and 30ms audio block sizes. This selection aligns with the library's average supported value and strikes a balance between computational efficiency and accuracy in voice activity detection. By leveraging this standard block size, we ensure compatibility with the library's capabilities while maintaining a pragmatic approach to processing audio data. This choice reflects our commitment to utilizing established practices and optimizing performance within our system's voice processing pipeline.

2. **400ms long audio data buffer** - To determine when a user starts or stops speaking, we used a 400ms long audio data buffer. This buffer length allowed us to strike a good balance between responsiveness and accuracy in detecting speech. By giving the system 400ms audio data (twenty 20ms audio blocks), we ensured it could accurately identify when the user began or ended their speech. This approach resulted in smooth and reliable voice capturing, improving the overall user experience.

3. **Value 15 as the speech starting threshold** - After conducting numerous tests, we determined that if more than 15 out of the 20 consecutive audio blocks contain speech, we can confidently conclude that a user has initiated communication with the robot. This threshold was chosen based on its ability to consistently deliver smooth and accurate results across various testing scenarios. By establishing this criterion, we ensure that the system reliably detects the onset of user speech, facilitating seamless interaction between the user and the robot.

4. **Value 3 as the speech ending threshold** - Based on testing, we determined that if fewer than 3 out of the 20 consecutive audio blocks contain speech, we can confidently infer that the user has concluded their interaction with the robot. This threshold was chosen based on its ability to consistently produce smooth and accurate recording results across various testing scenarios. By setting this criterion, we ensure that the system accurately identifies the end of user speech, facilitating seamless transitions and efficient processing of user inputs within the interaction flow.

5. **Value 2 as the VAD aggressiveness parameter** - The WebRTCVAD[23] library facilitates three possible values as the VAD (Voice Activity Detection) aggressiveness parameter. A level of 0 prioritizes capturing all potential speech, even if accompanied by background noise. Conversely, a level of 3 prioritizes eliminating noise but might risk missing quieter speech segments. We selected level 2 as the VAD aggressiveness parameter based on its balanced performance in speech detection. After evaluating different aggressiveness levels through testing, we found that level 2 offered a satisfactory balance between accuracy and smoothness in speech detection. This choice yielded consistent and reliable results across various testing scenarios, enhancing the overall effectiveness of our system in accurately capturing user speech inputs.

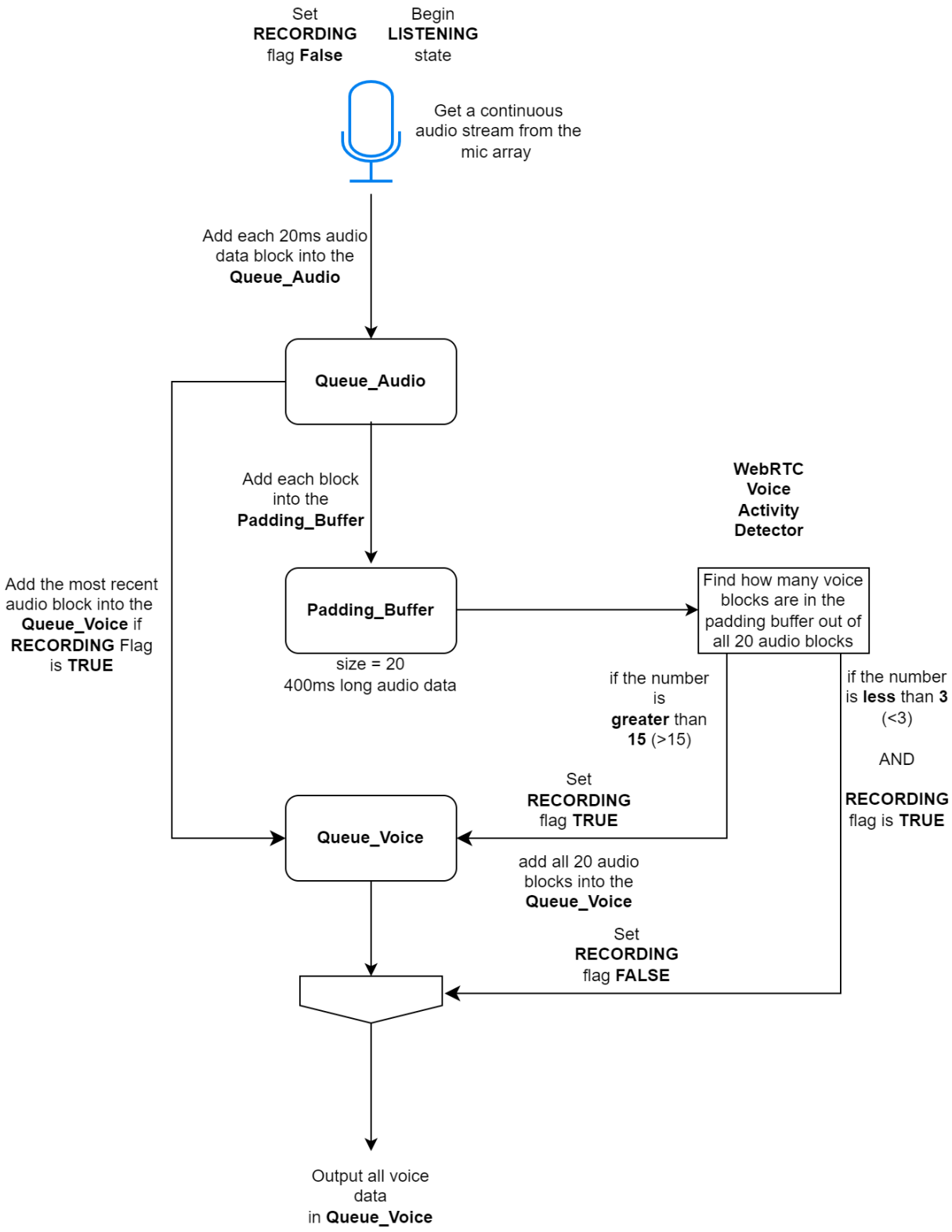


Figure 3.33: Voice capturing flow

3.8.2 Speech-to-Text Conversion

After capturing the user's voice data, understanding its context becomes crucial for subsequent processing stages. To convert the speech data into text, we employ the open-source **faster-whisper**[24] small.en model. This model is a reimplementation of OpenAI's Whisper[25] [26] model, utilizing CTranslate2[27] as a fast inference engine for Transformer models. The decision to use faster-whisper was motivated by its superior performance metrics compared to the original Whisper model. Notably, this implementation achieves up to four times faster inference speed while maintaining the same level of accuracy and requiring less memory. By leveraging faster-whisper, we ensure efficient and effective conversion of speech data to text, enhancing the overall performance and responsiveness of our system.

Whisper is a powerful automatic speech recognition (ASR) system trained on a vast dataset of 680,000 hours of speech from various sources on the web. This extensive dataset gives Whisper an edge, making it more resilient to different accents, background noise, and technical terms in speech-to-text conversion. **Figure 3.34** shows the Whisper architecture. A comparison for different Whisper models is shown in **Table 3.2**

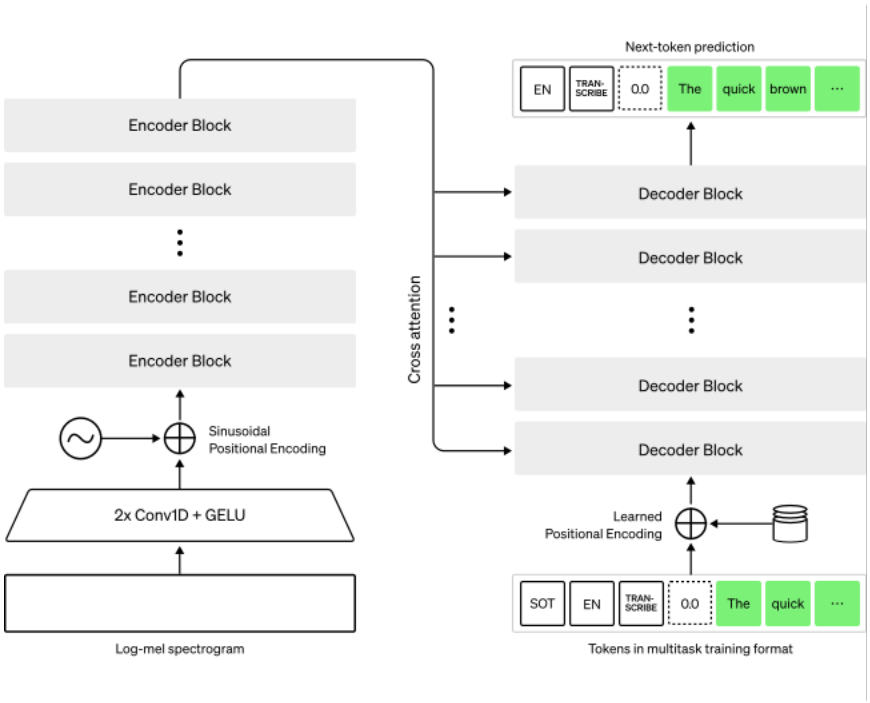


Figure 3.34: The Whisper architecture is a simple end-to-end approach, implemented as an encoder-decoder Transformer. Input audio is split into 30-second chunks, converted into a log-Mel spectrogram, and then passed into an encoder. A decoder is trained to predict the corresponding text caption, intermixed with special tokens that direct the single model to perform tasks such as language identification, phrase-level timestamps, multilingual speech transcription, and to-English speech translation.

Size	Parameters	English-only model	Multilingual model	Required VRAM	Relative speed
tiny	39 M	`tiny.en`	`tiny`	~1 GB	~32x
base	74 M	`base.en`	`base`	~1 GB	~16x
small	244 M	`small.en`	`small`	~2 GB	~6x
medium	769 M	`medium.en`	`medium`	~5 GB	~2x
large	1550 M	N/A	`large`	~10 GB	1x

Table 3.2: There are five Whisper model sizes, four with English-only versions, offering speed and accuracy tradeoffs. Above are the names of the available models and their approximate memory requirements and inference speed relative to the large model; actual speed may vary depending on many factors including the available hardware.

We opted for Whisper's small, English-only model, specifically the small.en variant. During testing, this model demonstrated notably accurate results while exhibiting significantly low latency when deployed on the Jetson Xavier platform. This combination of accuracy and efficiency underscores the suitability of the small.en model for real-time speech processing applications. The robust performance observed on the Jetson Xavier platform highlights the model's compatibility with resource-constrained environments, making it a promising choice for deployments where both accuracy and speed are essential considerations.

3.8.3 Natural Language Understanding and Generation

For our natural language understanding and generation tasks, we leverage **Mistral**[5], a Large Language Model (LLM) with 7 billion parameters, deployed on the Jetson AGX Xavier platform. Large Language Models represent the cutting edge in language understanding and generation capabilities, offering state-of-the-art performance in comprehending and generating human-like text. With Mistral's extensive parameterization, we achieve superior linguistic prowess, enabling our system to interpret and produce natural language with remarkable accuracy and fluency.

To facilitate inference with this large language model, we employ the exllamv2[28] library, which streamlines the inference process, ensuring efficient utilization of computational resources on the Jetson AGX Xavier platform. This integration of advanced language models and inference tools empowers our system to deliver sophisticated natural language understanding and generation capabilities, enriching the user experience and expanding the scope of potential applications.

Custom Instruction Dataset for LLM Finetuning

To fine-tune[6] the Mistral[5] 7 billion Large Language Model (LLM) for our specific application, we curated a custom dataset comprising approximately 9000 samples. This dataset is meticulously crafted to include fundamental information about the Electronic and Telecommunication Department, along with detailed descriptions of various locations within the department and their respective floor placements. By capturing these essential details, such as departmental infrastructure and spatial organization, our dataset provides a rich context

for training the language model to understand and generate responses relevant to our domain. This focused dataset serves as a crucial resource for enhancing the model's proficiency in accurately interpreting user queries and providing contextually appropriate responses tailored to the department's operations and facilities. A sample from the dataset is shown in **Figure 3.35**.

```
Instruction: can you give me directions to the analog lab of the Department of
               Electronic and Telecommunication at the University of Moratuwa?

Response: The analog lab of the Department of Electronic and Telecommunication
               at the University of Moratuwa is located on the 2nd floor.



Instruction: Is there a bio-medical laboratory at the Department of Electronic
               and Telecommunication at the University of Moratuwa?

Response: Yes, the bio-medical lab of the Department of Electronic and Telecommunication
               at the University of Moratuwa is located on the ground floor.
```

Figure 3.35: Two samples from the dataset

Mistral 7-Billion Parameter Large Language Model

For our language processing unit, we chose to implement the Mistral[5] 7 billion Large Language Model (LLM), developed by Mistral AI[29] company. At the time of initiating the development of our robot's language processing capabilities, Mistral stood out as the top-performing 7 billion parameter model available, making it the ideal choice for our project. **Table 3.3** shows the architecture of the model. **Table 3.4** shows the performance of the Mistral 7B compared with other LLMs.

Parameter	Value
dim	4096
n_layers	32
head_dim	128
hidden_dim	14336
n_heads	32
n_kv_heads	8
window_size	4096
context_len	8192
vocab_size	32000

Table 3.3: Model architecture

Model	Modality	MMLU	HellaSwag	WinoG	PIQA	Arc-e	Arc-c	NQ	TriviaQA	HumanEval	MBPP	MATH	GSM8K
LLaMA 2 7B	Pretrained	44.4%	77.1%	69.5%	77.9%	68.7%	43.2%	24.7%	63.8%	11.6%	26.1%	3.9%	16.0%
LLaMA 2 13B	Pretrained	55.6%	**80.7%**	72.9%	80.8%	75.2%	48.8%	**29.0%**	**69.6%**	18.9%	35.4%	6.0%	34.3%
Code-Llama 7B	Finetuned	36.9%	62.9%	62.3%	72.8%	59.4%	34.5%	11.0%	34.9%	**31.1%**	**52.5%**	5.2%	20.8%
Mistral 7B	Pretrained	**60.1%**	**81.3%**	**75.3%**	**83.0%**	**80.0%**	**55.5%**	**28.8%**	**69.9%**	**30.5%**	47.5%	**13.1%**	**52.2%**

Table 3.4: Mistral 7B performance metrics. Mistral 7B outperforms Llama 2 13B on all metrics

Mistral[5] 7 billion comes in two versions: the base model and the instruction fine-tuned model. While the base model predicts the next token without reasoning, the instruction fine-tuned model can successfully follow instructions and generate meaningful responses. To tailor Mistral to our project's needs, we acquired the instruction fine-tuned model and further fine-tuned it using a custom dataset specific to the Electronic and Telecommunication Department. Leveraging the Parameter-Efficient Fine-Tuning[6] method, we optimized the model's performance while minimizing computational and storage costs. This approach also mitigated the risk of catastrophic forgetting, a phenomenon observed during full fine-tuning of LLMs. In inference testing, the fine-tuned model demonstrated strong performance, particularly when prompted appropriately, affirming its efficacy for our language processing tasks.

Model Inference

For our model inference needs, we use the ExLlamaV2[28] library, tailored for running local Large Language Models on modern consumer GPUs. Specifically, we utilize the .exl2 format, employing a 4-bit quantized Mistral[5] 7 billion model. With the ExLlamaV2 library, our model achieves impressive performance, generating 10-15 tokens per second on the Jetson AGX Xavier platform. In particular, the library offers a unique feature compared to other inference libraries: the ability to retrieve generated tokens one by one from the model, rather than waiting for the entire response generation to finish. This incremental token retrieval enhances responsiveness and facilitates smoother interactions within our system, ultimately contributing to an enhanced user experience.

3.8.4 User Input Classification

To facilitate seamless interaction between the user and the robot, we developed a system capable of discerning whether the user is referencing a specific location within the department and requesting directions to that location. This information is crucial for providing relevant responses and guiding the robot's navigation system, user interface (UI) of the robot's screen, and hand gesture controller. Initially, we experimented with a fine-tuned BERT[30] model for text classification to identify user queries about locations and direction requests. However, running BERT models concurrently with Mistral LLM slowed down the conversation-handling process. Consequently, we opted for a logic-based classifier, which efficiently identifies keywords indicative of location-related queries, such as "guide me", "direction", "analog lab", "where" etc. By analyzing the converted text from user speech and filtering for these keywords, we can accurately classify whether the user is referencing a specific location and requesting directions. This streamlined approach optimizes the conversation handling flow, ensuring prompt and effective responses while mitigating computational overhead.

3.8.5 Text-to-Speech Conversion

The text-to-speech functionality within our system operates through two parallel processes, each contributing to the seamless conversion of text phrases into spoken audio. Initially, the

Large Language Model (LLM) generates tokens sequentially, where a token represents a word, symbol, or punctuation mark. As the LLM generates text, it identifies the end of phrases or sentences marked by punctuation such as full stops, commas, or exclamation marks. Upon encountering such punctuation, the text phrase is added to the convert_to_speech queue. Subsequently, the audio creation process utilizes the Piper package to generate an audio clip corresponding to the text phrase in the queue. Once created, the audio file name is enqueued into the play_audio queue. Simultaneously, the play_audio process retrieves audio files from the play_audio queue and plays them, ensuring a continuous flow of spoken responses. This parallel architecture optimizes the text-to-speech conversion process, minimizing delays and enhancing the overall user experience by providing prompt and fluent verbal interactions.

Voice of the robot

For text-to-audio generation within our system, we employ the Piper[31] package, a high-performance neural text-to-speech solution renowned for its natural sound quality and diverse voice options. Piper offers the capability to execute voice models ranging from 5 million to 20 million parameters through the utilization of onnxruntime[32]. These voice models, trained using the VITS[33] auto-encoder, ensure superior audio output fidelity and voice consistency. Typically, Piper takes approximately 1.5 to 2 seconds to generate an audio clip for a given text input using a specified voice model. This efficient processing time enables rapid conversion of text phrases into high-quality audio, facilitating smooth and responsive verbal interactions within our system.

3.9 User Interface

The user can interact with the receptionist using the user interface which is displayed using an 800x480 LCD display with a touch panel. Since both UI and UX are important first we designed the basic appearance and the flow of the pages using the Figma tool as in **Figure 3.36**. Figma[34] is an enterprise-level professional tool for UI/UX designing.

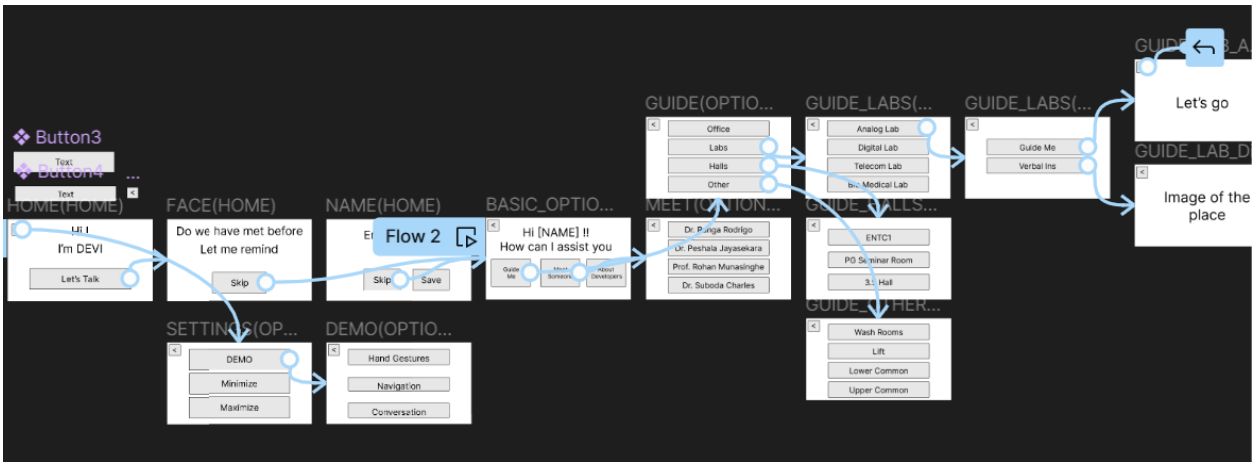


Figure 3.36: Figma UI/UX design

We evaluated the available options to implement a UI in an Ubuntu system. QT and GTK are the frameworks which were available as the best options. Among them, QT was selected by considering the elegant look, performance, and usage by the developers.

3.9.1 UI Architecture & Implementation

As depicted in **Figure 3.37** a custom architecture for the UI implementation was designed.

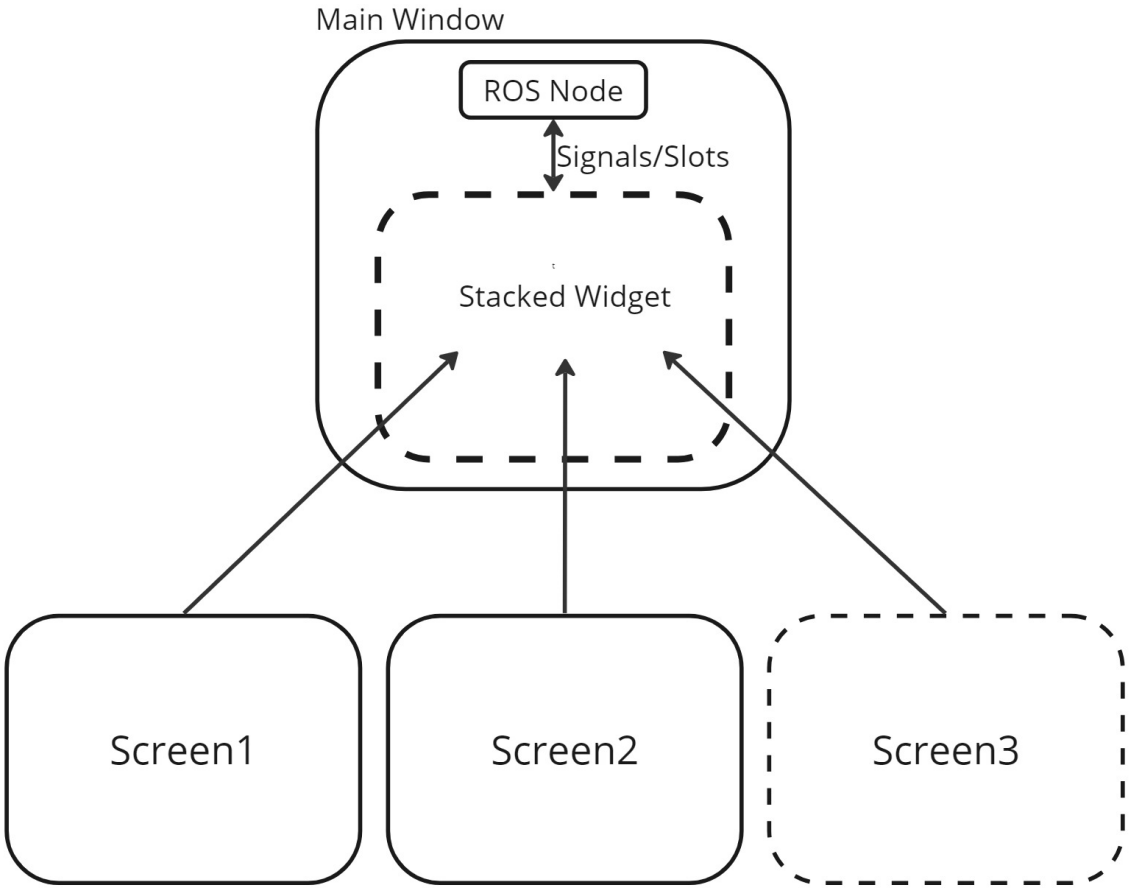


Figure 3.37: UI architecture

- Screens and Pages
  After observing some pages have the same structure, we introduced something called screens which is kind of a template for a page. So several pages can be generated at run time using a screen. Page is any specific instant of the UI. This approach made the design efficient, flexible and scalable. 11 screens and 18 pages were implemented in the final design.

- Main Window
  The main window is the parent container which has 2 components mainly.

– Stacked Widget
This widget is capable of holding screens with loaded pages in a stacked manner. That stacked behaviour was used to implement the back button feature. When we go to a new page, it loads on top of the current page facilitating to go back by removing the topmost page. Home button was also implemented that cleared the stack except the home screen.

– ROS Node
This was used to communicate with the ROS nodes in the main system. It has multiple subscribers as well as multiple publishers. It runs on a separate thread facilitating to communication with other ROS nodes in the system without blocking the UI thread. Components in the main window and components in the screens loaded into the stacked widget can communicate bidirectionally with the ROS Node using signals and slots. Signals and slots are used in the QT framework for internal communication between threads. Signals are sent from button clicks and user inputs are sent to other ROS nodes through the publishers in the ROS Node and subscribers listen for the data from the other ROS nodes and send them to the slots to make the necessary changes in the UI. This ROS Node implementation based on ros2_qt_demo[35] public repository

– Blinking Listening Icon
To improve the user experience we added a blinking indication button as depicted in **Figure 3.38** to inform the user whether the robot is listening or not. Because the robot is speaking it is not listening. Also by clicking it, we can forcefully make the robot stop listening and start listening after stopping forcefully.

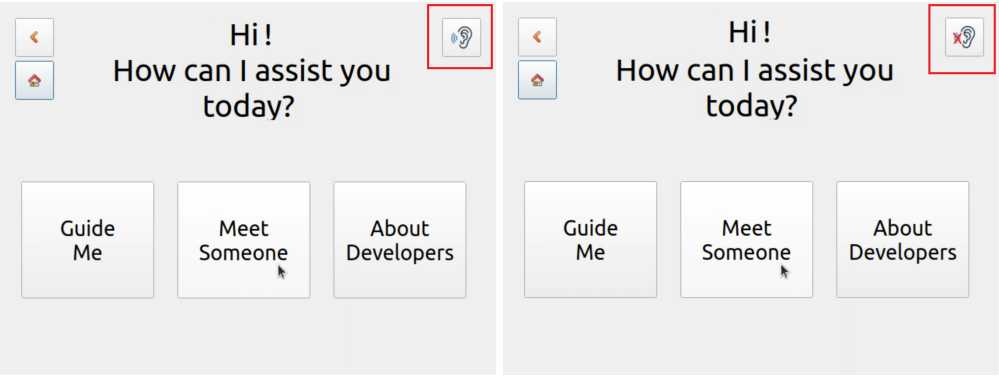


Figure 3.38: Listening indication icon

– Blinking Border
Some situation needs the special attention of the user for example while in a voice conversation, it may ask the user for confirmation through UI for verbal instruction or navigate and guide him to a specific location. That kind of screen is displayed with a blinking border to get the user's attention as depicted in **Figure 3.39**.

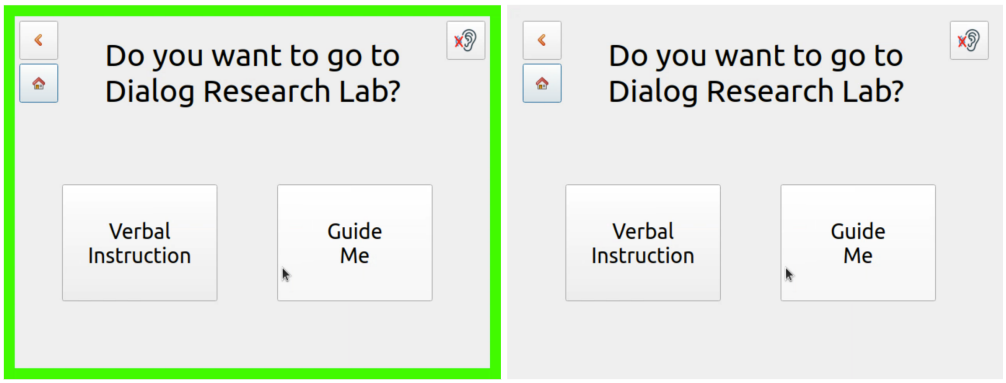


Figure 3.39: Blinking border

We added a settings page to test the functionalities and change the resolutions. Changing the resolution was necessary because for debugging purposes we connect to the Jetson board through a remote viewer.

3.10 Mobile App

Android application was developed basically for admin purposes. Android Studio IDE with Java was used to develop the app. The app has 2 main features.

3.10.1 Main Features

Joystick

This feature was used to tele-operate the robot in the mapping stage.

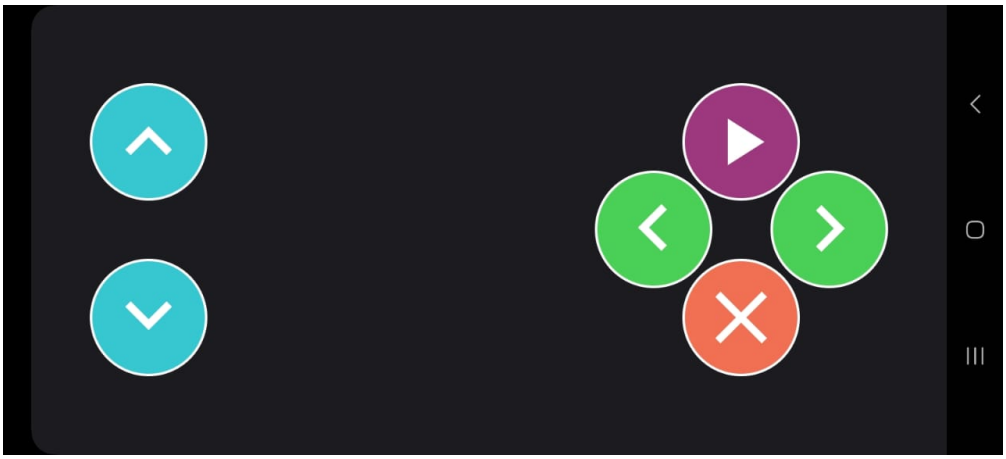


Figure 3.40: Joystick feature of the app

The UI for the joystick features is shown in **Figure 3.40**. Those buttons are multi-touch supported, the robot can go forward or backwards while turning. This gives smooth control of the robot.

Map

After the mapping stage, the created map can be loaded into this map in the app. As in **Figure 3.41** it shows the real-time location of the robot which is sent from the robot to the app. Also, we can set a specific point and command the robot to go to that specific location. At this level, this feature can be moved into a specific location. But since this project will continue in future this feature can be used to summon the robot for a specific location or make the robot bring or take something from there.

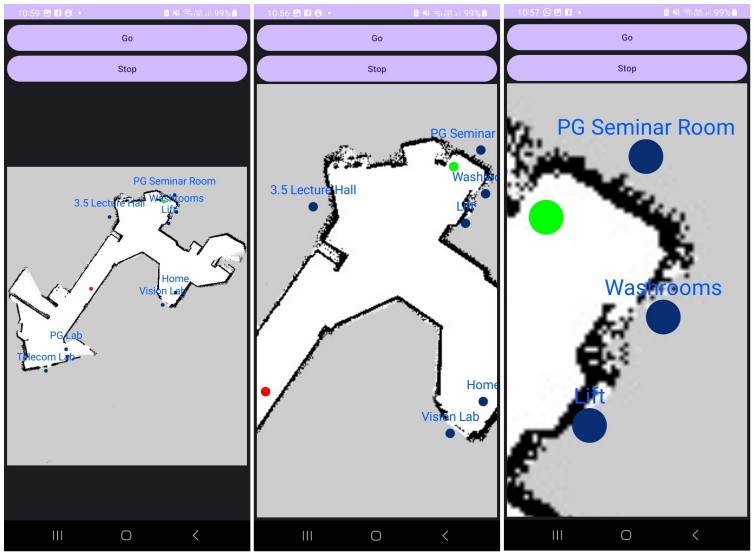


Figure 3.41: Map feature with zooming option

Custom Map View

Custom map view was implemented from scratch based on our requirements. Location coordinates sent from the robot needed to map into pixel coordinates in the map. The current scale and translation state of the map image can be obtained as a transformation matrix. This matrix is bound with pinch and move touch gestures. Based on those gestures this matrix recalculated. Using that matrix we convert location coordinates sent from the robot to density-independent pixels[36] of the map image and display as a point and vice versa. With the touch input, we get pixel coordinates then we convert them to density-independent pixel coordinates and then convert them to location coordinates and send them to the robot when we need to give a target from the app to the robot Also, needed to display location labels with the name of the location when we give the coordinates and locations after generating the map in the mapping stage. The coordinate point conversation also applies here

3.10.2 Real-time Communication with the Robot

We needed to do real-time communication between the robot and the map to send joystick data and target location and to get the real-time location of the robot and display it on

the map. To accomplish this we designed a data flow as depicted in **Figure 3.42**. We used Firebase services to establish real-time communication. So it connects through the internet. Under Firebase services we used the real-time database feature. Firebase is a totally software-related service so no ROS-related stuffs were available. So we have to create an interconnection between ROS and Firebase from scratch. We implemented a ROS node that listens for the real-time database changes which are sent from the app and sends them to the system using ROS topics. Also, it could subscribe to ROS topics and send those data to the real-time database. The app could listen to those changes and display them on the map.

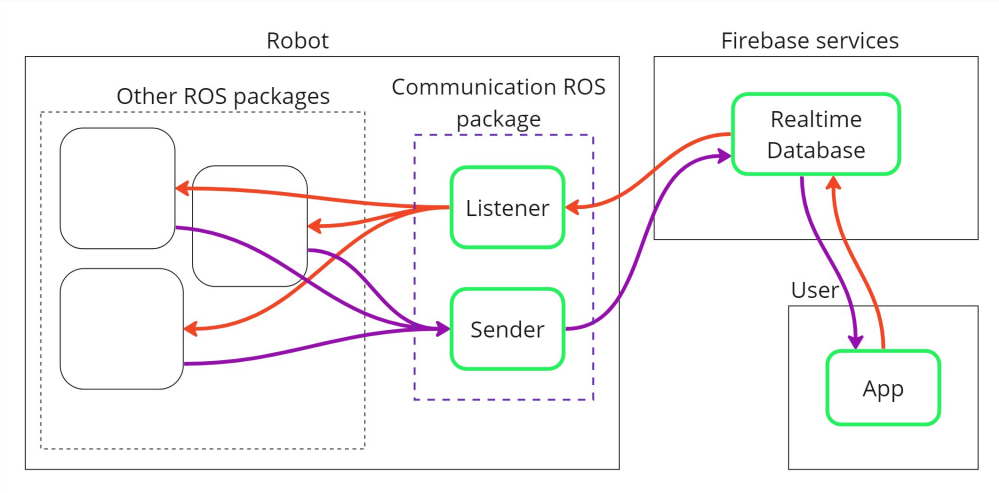


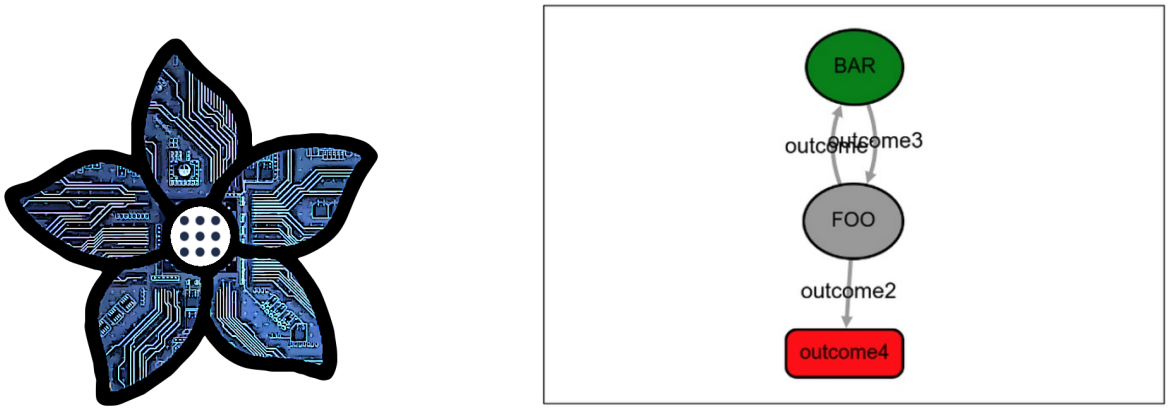
Figure 3.42: Real-time communication method of the app

3.10.3 Responsive design

The app was designed in a fully responsive manner, it can operate on Android devices with any aspect ratio and screen resolution. We obtained it by following best practices in UI/UX design and getting the benefit of using density-independent pixels

3.11 Finite State Machine

The implementation of the Finite State Machine (FSM) within the robot's system was achieved using YASMIN(Yet Another State MachINe) **Figure 3.43**. YASMIN is a robust framework designed to manage complex state transitions in dynamic environments. It provides a structured way to handle the various states of the robot, such as idle, navigation, interaction with users. By employing YASMIN, the robot can seamlessly switch between different operational modes, responding adaptively to its environment and user inputs. This implementation ensures that the robot behaves predictably while maintaining the flexibility to react to unexpected situations or changes in its operational context. The use of a state machine like YASMIN helps in maintaining a clear separation of concerns within the robot's control logic, facilitating easier maintenance and scalability of the system.



(a) Logo (b) Demo states

Figure 3.43: YASMIN

3.11.1 Features

- ROS 2 Integration: YASMIN is fully integrated with ROS 2, allowing for seamless interaction with other components of the Robot Operating System.
- Language Support: It supports both Python and C++, offering flexibility in programming language choice for development and prototyping.
- Rapid Prototyping: Enables fast development cycles, which is critical in dynamic project environments.
- Predefined States: Comes with default states tailored for ROS 2 action and service clients, streamlining the setup process.
- Data Sharing: Utilizes blackboards to facilitate data sharing across different states and state machines, enhancing the communication within the framework.
- Control Features: State machines can be easily canceled or stopped, providing robust control over the state execution, which is essential for handling unexpected scenarios or errors.
- Monitoring Tool: Includes a web viewer that allows developers and operators to monitor the execution of state machines in real time, providing a valuable tool for debugging and system assessment.

3.11.2 States

Several states were defined at the project's outset to control its operations and interactions. These states include:

- **Idle**: The robot remains in this state until it receives a trigger from the user interface, waiting passively for activation.

- **Conversation**: Upon activation, the robot initiates interaction with users by greeting them, employing face recognition for identity verification, and re-identifying known individuals.
- **Navigation**: This state is engaged when the robot needs to travel to a user-specified destination, which can be set through the UI or remotely via a mobile app by an administrator.
- **Back Home**: After completing its navigation state, the robot returns to its home base.
- **Guide**: During interactions, particularly when providing verbal instructions, the robot uses hand gestures to enhance communication.
- **Switching Power Mode**: This state involves managing the robot's power sources, switching from wall power to battery power when navigation begins, and reverting back to wall power once it returns home.

3.12 Power Management

3.12.1 Power Budget

The robot has mainly 3 power-consuming states based on functioning devices. So power budget was calculated for those 3 states separately.

Stationary - Fully Functioning (Wall Powered)

Table 3.5: Stationary - fully functioning state power budget

Component	Voltage (V)	Current (mA)	Power (W)
Nvidia Jetson	14	2000	28
Microcontroller	5	80	0.4
Servos	5	17000	85
Ultrasonic sensors	5	15	0.075
Zed2 Stereo Camera	5	380	1.9
Lidar	5	400	2
Touch Screen	12	200	2.4
Microphone	5	180	0.9
Total			$\sim 120W$

Moving (Battery Powered)

Table 3.6: Moving state power budget

Component	Voltage (V)	Current (mA)	Power (W)
Nvidia Jetson	14	1100	15.4
Microcontroller	5	80	0.4
Ultrasonic sensors	5	15	0.075
Motors	12	15000	180
Zed2 Stereo Camera	5	380	1.9
Lidar	5	400	2
Touch Screen	12	200	2.4
Microphone	5	180	0.9
Total			$\sim 200W$

Stationary - Idle (Wall Powered)

Table 3.7: Stationary - idle state power budget

Component	Voltage (V)	Current (mA)	Power (W)
Nvidia Jetson	14	1000	14
Microcontroller	5	80	0.4
Zed2 Stereo Camera	5	380	1.9
Touch Screen	12	200	2.4
Microphone	5	180	0.9
Total			$\sim 20W$

3.12.2 Power Sources

Direct Power Supply

By considering Stationary - Fully Functioning Power State
Power = 120W
Voltage =14V

Assuming 0.85 efficiencies of buck converters Required Current = (120/0.85)/14 =$\sim 10A$

Selected power supply (**Figure 3.44**) specifications

- Input Voltage = 230VAC
- Output Voltage = 14VDC (max)

- Output Current = 30A (max)

Figure 3.44: Power supply

Battery

Power = 200W
Voltage =14V

Assuming 0.85 efficiencies of buck converters Required current rating (200/0.85)/14 $=\sim 16A$

Required battery specifications by considering operating time

Table 3.8: Required battery specifications

Parameter	Value
Current (A)	16A
Voltage (V)	14.8V
Operating time (h)	0.5h
Capacity (Ah)	8Ah
Discharge Rate	2C

But we had to use an available battery. So the used battery was 4S6P Li-ion 8650 Battery custom battery pack (**Figure 3.45**).

- Nominal Voltage: 3.6v x 4 = 14.4V
- Nominal Capacity: 3Ah x 6 = 18Ah
- Discharge Rate: $20x6/18 =\sim 6C$

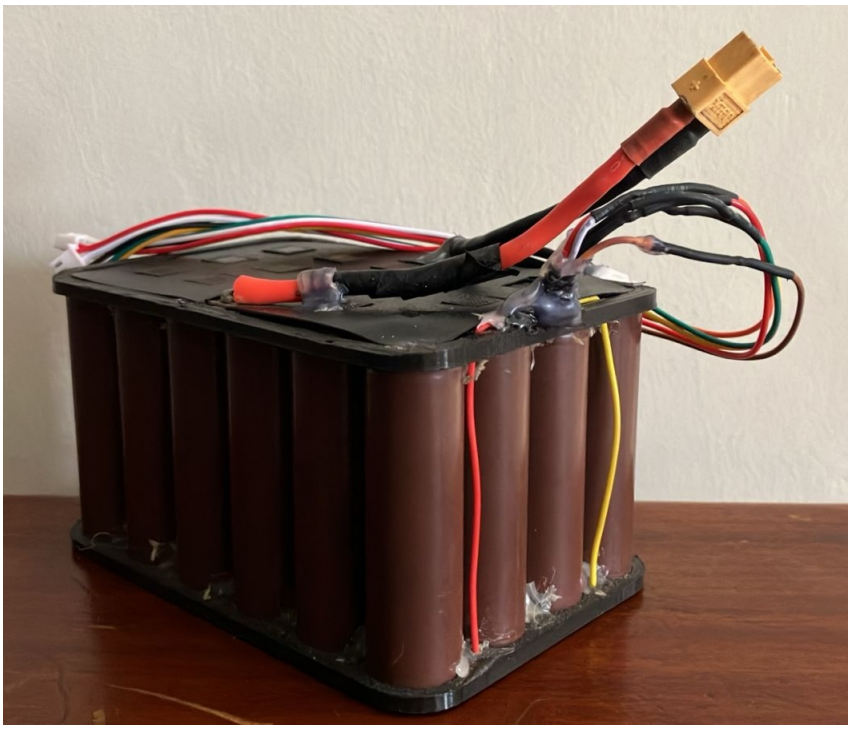


Figure 3.45: Used battery pack

3.12.3 Power Source Switching System

Since the robot needs to switch the power source in runtime an automatic power source switching system was designed using Schottky diodes as in **Figure 3.46**. Relays were used for additional control from the robot system.

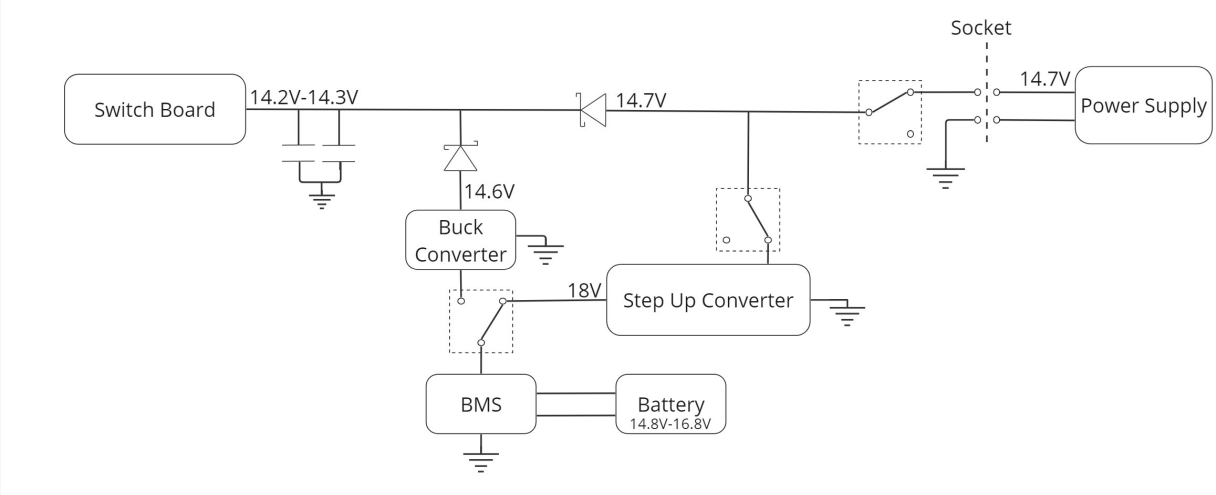


Figure 3.46: Power source switching system