Helwan University Faculty of Engineering

Computer & Systems Engineering Department

Automated Warehouse System

BSE. Senior Year Project

Submitted By:

Mohamed Magdy Mohsen

Seif Mostafa Abdalhalim

Tarek Mahmoud Ali

Supervised By:

Prof. Dr. Amr ElSayed

July 2023

Acknowledgment

Our heartfelt appreciation and utmost gratitude are extended to our Graduation Project Supervisor, Assoc. Prof. Amr El Sayed. His expert advice, wealth of knowledge, and insightful dialogues have been an invaluable resource throughout our entire journey.

Prof. El Sayed's patience and guidance in his methodology have not only enlightened us but also empowered us to reach new academic heights. His steadfast belief in our potential has inspired us to deliver our very best in this graduation project.

ABSTRACT

This book provides a comprehensive examination of the challenges and advancements in large warehouse systems, emphasizing the critical need for adopting automated or semi-automated systems to enhance operational efficiency and reduce reliance on manual labor. With the continuous growth of warehouse facilities on a global scale, there is an increasing realization of the limitations and drawbacks of manual warehouse management methods, highlighting the compelling advantages offered by robotic systems.

The existing manual approach to warehouse management is characterized by several shortcomings. It is limited by human precision, resulting in output inconsistencies and inefficiencies. Manual methods are also associated with elevated operational costs, primarily due to the significant portion of the budget allocated to labor salaries. Additionally, human errors in package tracking and inventory management can lead to delayed shipments and mismanaged inventory. Moreover, manual methods struggle to predict and anticipate the necessary supplies, leading to inefficiencies in stock management. Furthermore, the reliance on manual labor introduces safety risks and potential injuries to workers, as evidenced by reports of serious injuries in warehouse facilities.

To address these challenges, this book explores three key automated processes for warehouse management. Firstly, it examines the concept of delivery notification, whereby the delivery car or truck notifies the system in advance about incoming packages or placed orders. This step aims to automate the process from the moment the package is received by the warehouse, reducing the need for human interaction.

Secondly, the book delves into the implementation of an automated convoy system, which facilitates the seamless transfer of shipments from the delivery truck to the robotic arm. This system ensures efficient scheduling and workflow management, minimizing delays and optimizing the utilization of resources.

Lastly, the book explores the concept of an autonomous robot equipped with advanced object detection capabilities. This robotic arm swiftly detects and transports each package to the designated area corresponding to its category, ensuring accurate and efficient sorting and shelving.

Contents

Table of figures

List of Tables

Chapter 1
INTRODUCTION

Warehouse management is a critical aspect of supply chain operations, involving the efficient and effective handling, storage, and movement of goods within a facility. However, warehouses often face various challenges and problems that can hinder their productivity, increase costs, and affect customer satisfaction. Addressing these problems is crucial for optimizing warehouse operations and maintaining a competitive edge in toaday's fast-paced business environment. Some require a huge amount of storage space with big warehouses. Some require the mediocre kind of space and others require less space. In the global market, these industries are moving towards building a better economy. 43% of small businesses either do not track inventory or use a manual method. Another survey by Peoplevox found that 34% of businesses have delayed shipping because the products mentioned in the order were not actually in stock. Warehouses owners do not take into consideration the seasonal demand pattern, leaving some demanded items unfulfilled while wasting resources on unneeded items. According to Kane Logistics, labor is one of the biggest expenses; owners spend from 50-70% of the overall warehousing budget on labor salary. Work environment in warehouses is not safe for the Workers. According to a new report released by the Strategic Organizing Center (SOC), Workers at Amazon facilities sustained more than 34,000 serious injuries on the job in 2021—leading to a serious injury rate at Amazon warehouses that is more than twice as high as the rate at non-Amazon warehouses. Over 145,000 workers across the U.S. work in warehouses, as the Occupational Safety and Health Administration reports. Sadly, about one in 20 of these workers will suffer injuries in a given year, according to data from the U.S. Bureau of Labor Statistics.

Chapter 2
ANALYSIS AND REQUIREMENTS

2.1 System Requirements

System stack begins when the delivery truck notifies the robotic warehouse that a shipment has been made. the fully-automated and trained robotic arm would take the packages and arrange them according to their tags and category. The smart warehouse system would then track each package's place and plan a path for different robots to pick the packages onto their suitable sections. Upon getting a shipment request from clients the robot network would then plan a subset of the working robots to handle this request.

2.2 Functional Requirements

REQ-X	**Requirement description**
REQ-1	Communication Interface: The system should have a communication interface to receive shipment notifications from the delivery truck, such as an API or messaging system.
REQ-2	Automated Convey: • The system requires a automated and trained robotic arm capable of handling packages. • The robotic arm should be able to identify package tags and categories accurately.
REQ-3	Robot Arm: • It must detect the position of the package and its pose. • Calculate the robot joint angles to reach the package.
REQ-4	Package Tracking: • The system needs to track the location of each package within the warehouse. • It should provide real-time updates on the package's current position and its movement history.

REQ-5	Path Planning: • The system should have path planning algorithms to determine the most efficient routes for robots to pick up packages from their respective locations. • The path planning algorithms should consider factors such as distance, traffic, and congestion within the warehouse.
REQ-6	Robot Network: • The system requires a network of robots capable of picking up packages and transporting them to suitable sections. • The robot network should have the capability to handle multiple shipment requests concurrently. • It should be able to dynamically allocate robots to handle different shipment requests based on availability and workload.
REQ-7	Shipment Request Handling: • The system needs to handle shipment requests from clients. • It should be able to receive shipment details, such as package quantities, delivery deadlines, and delivery locations. • The system should have the capability to prioritize and allocate suitable robots to handle each shipment request.
REQ-8	1. Automation and Integration: • The system should automate the process of package handling, tracking, and path planning. • It should integrate with various sensors, cameras, and control systems to ensure seamless operation and coordination.
REQ-9	Mobile robot:

	• The mobile robot should be able to move in dynamic environment

2.3 Nonfunctional Requirements

REQ-X	**Requirement description**
REQ-10	Scalability and Performance: • The system should be designed to handle a large volume of shipments and packages efficiently. • It should be scalable to accommodate future growth and expansion. • The system should have sufficient processing power, storage capacity, and network bandwidth to handle the workload.
REQ-11	Reliability and Fault Tolerance: • The system should be reliable, ensuring high availability and minimal downtime. • It should have mechanisms in place to handle failures, such as backup systems, redundancy, and fault-tolerant algorithms. • The system should be able to recover from errors or disruptions without significant impact on operations
REQ-12	2. Security:

	• The system should incorporate security measures to protect sensitive data, prevent unauthorized access, and ensure data integrity. • It should have authentication and access control mechanisms to restrict system access to authorized personnel only

Table 1Functional and nonfunctional requirements

2.4 System Stakeholders

Warehouse Managers: Oversee warehouse operations and ensure system efficiency.

Delivery Companies: Collaborate for smooth shipment coordination.

Warehouse Operators: Responsible for system operation and maintenance.

IT Department: Manages system configuration, maintenance, and reliability.

Robotics Engineers: Design, develop, and maintain robotic components.

Clients and Customers: Utilize warehouse services and rely on system performance.

Maintenance and Support Teams: Handle system maintenance and troubleshooting.

System Integrators: Ensure seamless integration of hardware and software.

Regulatory Bodies: Oversee compliance with safety and labor regulations.

End Users: Benefit from system performance as package senders or recipients.

2.5 System Usecase

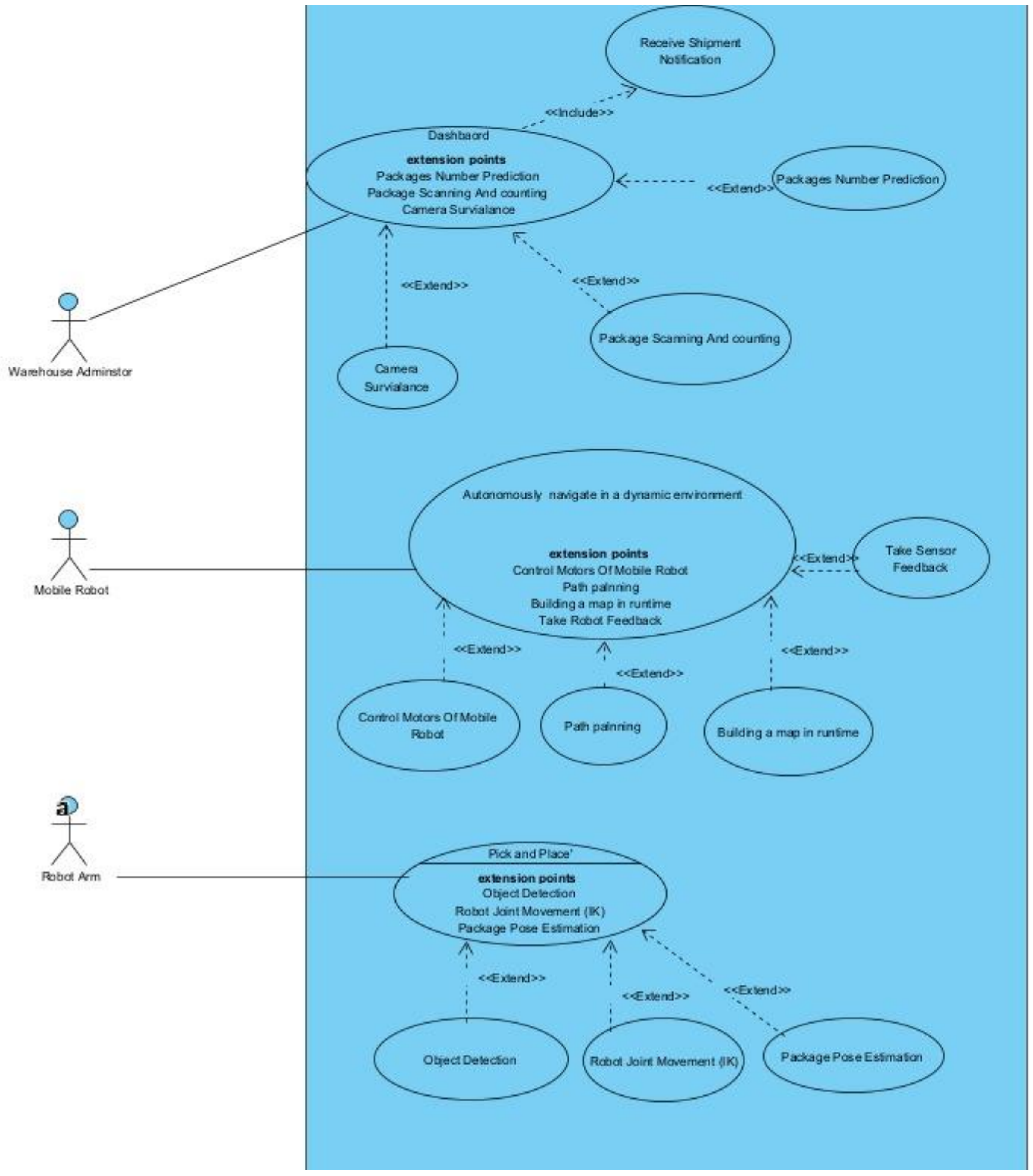


Figure 1 Usecase diagram

Chapter 3
Background

3.1 Machine Learning

Machine learning is the study of computer algorithms that improve automatically through experience and using data. It is seen as part of artificial intelligence. Machine learning algorithms build a model based on sample data, known as "training data," in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a variety of applications, where it is difficult or unfeasible to develop traditional algorithms to perform the required tasks. A subset of machine learning is closely related to computational statistics, which focuses on making predictions using computers; But not all machine learning is statistical learning. The study of Mathematical Optimization provides methods, theory, and application areas in the field of machine learning. Data mining is a related field of study, with an emphasis on exploratory data analysis through unsupervised learning, and machine learning in its application across business problems, is also referred to as predictive analytics.

3.1.1 Supervised Learning

Supervised learning algorithms construct a mathematical model of a set of data that include both the inputs and the desired. outputs the data is known as sampled data and consists of a set of training examples. Each training example has one or many inputs. and a desired output, also known as a supervisory signal. In the mathematical model, each training example is displayed by a vector or array, at sometimes it is called a feature vector, and the training data is displayed by a matrix. Through reduplicate optimization of an objective function, supervised learning algorithms learn a function that can be used to guess the output. associated with new inputs. An optimal task will allow the

algorithm to correctly determine the output for inputs that were. not a piece of the training data. An algorithm that enhances the accuracy of its outputs or predictions overtime is said to have. learned to implement that task.
Similarity learning is a wide range of supervised machine. learning relative to regression and classification, but the objective is to learn from examples using a common function that measures. how similar or related two objects are. It has applications in recommendation systems, visual identity tracking, and face verification and ranking.

3.1.2 Unsupervised Learning

Supervised learning refers to using a set of input variables to predict the value of a labeled output variable. It requires labeled data. (Think of this like an answer key that the model can use to evaluate its performance). Conversely, unsupervised learning refers to inferring. underlying patterns from an unlabeled dataset without any reference to labeled outcomes or predictions.

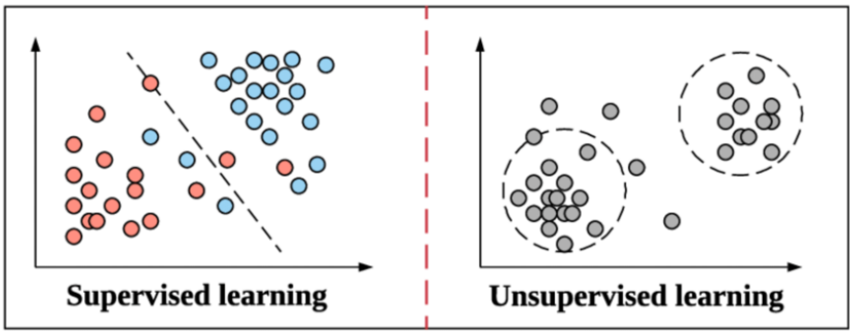


Figure 2 Comparison between supervised and unsupervised learning

There are several methods of unsupervised learning, but Clustering is far and away the most used unsupervised learning. technique. Clustering refers to the process of automatically grouping. together data points with similar characteristics and assigning them to "Clusters."

3.1.3 Reinforcement Learning

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like. situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or

penalties for the actions it performs. Its goal is to maximize the total. reward.

Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.

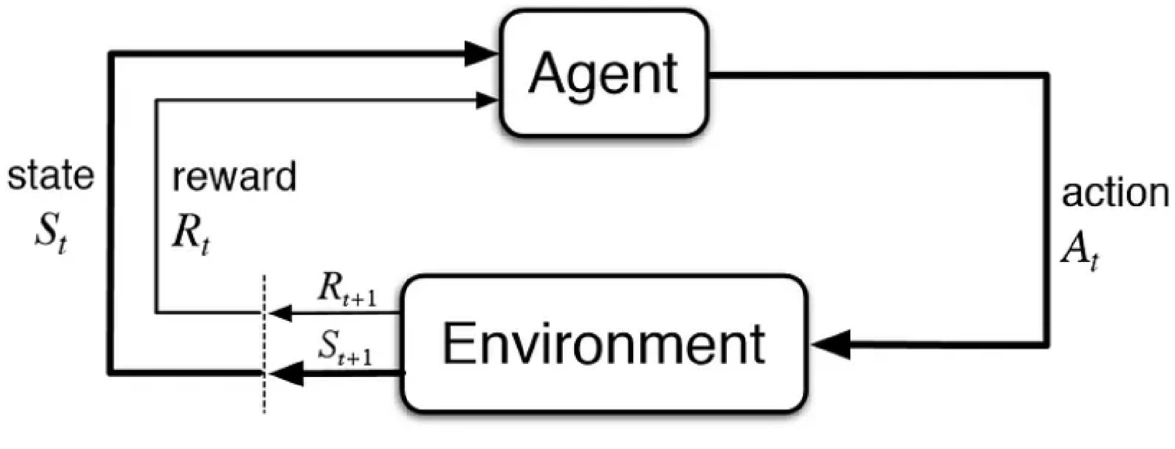


Figure 3 Reinforcement Learning

Distinguishing between reinforcement learning, deep learning,
and machine learning is hard. In fact, there should be no clear divide.
between machine learning, deep learning, and reinforcement learning.
It is like a parallelogram – rectangle – square relation, where machine
Learning is the broadest category and the deep reinforcement learning.
the narrowest one.
In the same way, reinforcement learning is a specialized application.
of machine and deep learning techniques, designed to solve problems.
in a particular way.

3.2 Computer Vision

3.2.1 RosCamera Interface

Before delving into the details of our Computer Vision (CV) modules, it is necessary to discuss the creation of an interface that mimics the OpenCV library's physical camera within the simulation environment. A more comprehensive examination of the virtual camera used in the simulation, which transmits rendered scene data in JPEG format to our RosCamera class, will be presented in the subsequent simulation section.

The sudo code below showcases the ImageReceiver and RosCamera classes.
```
1. Import required libraries: rclpy, Node from rclpy.node,
CompressedImage from sensor_msgs.msg, cv2, numpy, and threading

2. Define a class ImageReceiver:
   1. Define __init__ function with parameters self, save_image (default
False), show_img (default False), and img_topic (default empty string)
       1. Initialize the node 'image_receiver'
       2. Create a subscription for CompressedImage with the image
topic, a callback function for image, and QoS profile depth 10
       3. Initialize attributes show_img and save_img with the values
passed
       4. Initialize attribute img with an empty numpy array

   2. Define a callback function image_callback with parameter self and
msg
```

```
        1. Check if the image format in msg is 'jpeg'
            1. Decompress the image
            2. If show_img is True, display the image
            3. If save_img is True, save the decompressed image to a file
        2. If the image format is not 'jpeg', log a warning about
unsupported image format

3. Define a class RosCamera:
    1. Define __init__ function with parameters self, topic, and
show_img (default False)
        1. Initialize rclpy
        2. Create an instance of ImageReceiver with True, show_img, and
topic as parameters
        3. Create a new thread to spin the ros node and start it

    2. Define a function spin_ros_node with parameter self to spin the
ros node continuously

    3. Define a function GetCameraData with parameter self to get the
image data from the image receiver node

    4. Define a function stop with parameter self
        1. Destroy the image receiver node
        2. Shutdown rclpy
        3. Join the thread
```

This pseudocode representation demonstrates how the RosCamera class acts as an interface, facilitating interaction between the simulation environment and the computer vision modules. By leveraging this structure, raw visual data from the virtual environment is channeled effectively into the analysis pipeline, enabling the generation of meaningful insights.

3.2.2 Object Tracking and counting.

In this part of the thesis, the focus is on object tracking and counting using machine learning. To achieve this, two key techniques were used; object detection with YOLOv5s/m and centroid tracking.

YOLOv5: An Overview

YOLOv5, standing for "You Only Look Once version 5," is an object detection model. The goal of object detection is to identify "what" objects are present in an image and "where" they are located. This dual task of simultaneously recognizing and localizing objects sets it apart from image classification, which only identifies what objects are in an image, without specifying their locations.

YOLO, as the name suggests, takes a unique approach to object detection. It looks at the entire image only once (hence "You Only Look Once") rather than using a sliding window or region proposal methods to identify multiple objects in the image. This gives YOLO a speed advantage over other object detection models, making it particularly well-suited for real-time object detection.

The YOLOv5 version is not an official version from the original YOLO authors but a repository from Ultralytics. Despite this, it's gained popularity due to its performance and speed.

YOLOv5 Model Architecture

YOLOv5's architecture is composed of three main parts: the backbone, the neck, and the head.

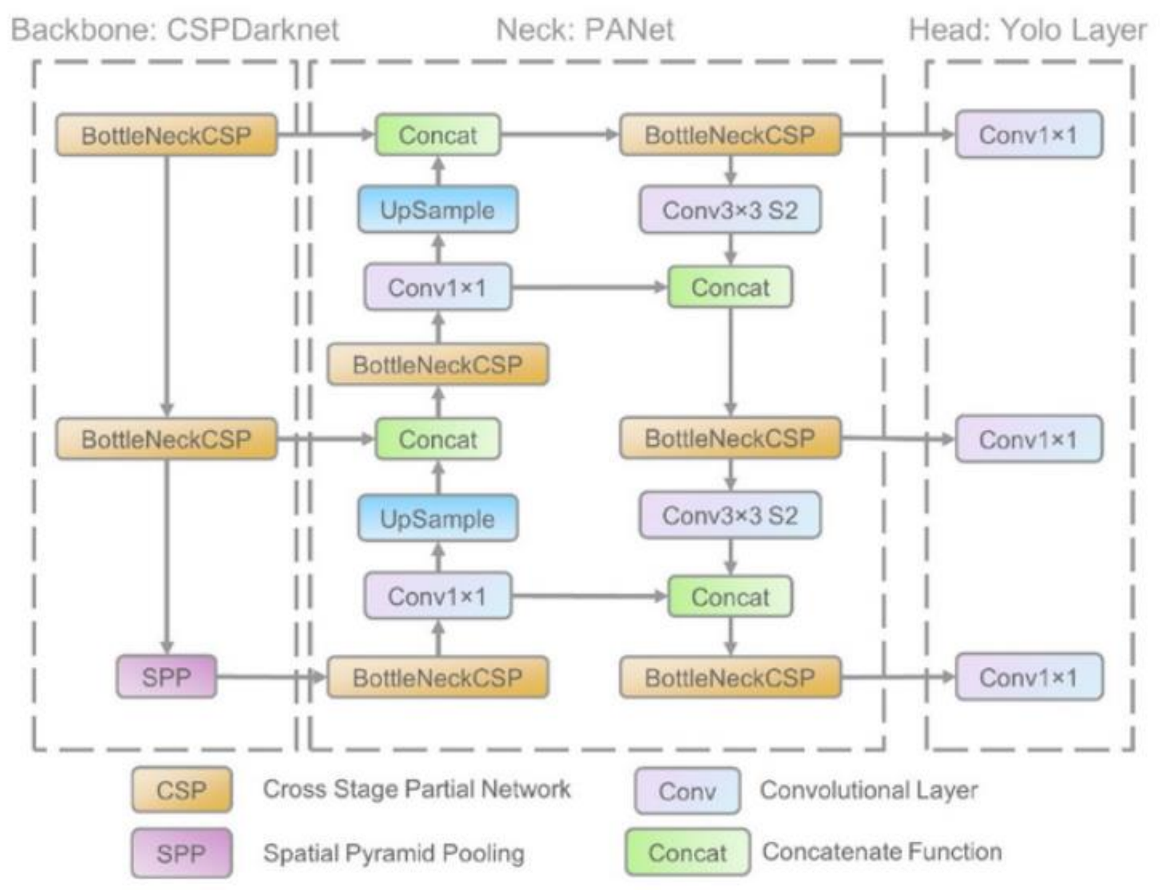


Figure 4 Network architecture for YOLO v5

Backbone (CSPDarknet53)

The backbone is essentially the feature extractor and consists of convolutional layers that process the input image and extract low-level features such as edges, colors, and textures. The backbone used in YOLOv5 is CSPDarknet53, which is a variant of Darknet53 that uses the Cross Stage Partial Network (CSPNet) strategy. This strategy reduces redundancy in feature computation, enhancing efficiency without compromising performance.

Neck (PANet & FPN)

The neck part, also known as the feature pyramid network (FPN), is responsible for multiscale feature extraction. In simple terms, objects in images can come in different sizes. Some objects might be close to the camera (and therefore appear large), while others might be far away (and appear small). Multiscale feature extraction allows the model to detect objects of varying sizes.

FPN accomplishes this by integrating low-resolution, semantically strong features with high-resolution, semantically weak features. The PANet (Path

Aggregation Network) further enhances this by enabling information flow across different layers.

Head

The head of the model is the detection part that uses the processed feature maps from the backbone and neck to predict class probabilities and bounding box coordinates. The head uses anchor boxes, which are predefined boxes with certain height and width ratios, to accommodate different object shapes.

YOLOv5 also uses a regression technique to adjust these anchor boxes to better fit the actual objects.

3.2.3 YOLOv5: Object Detection Process

The object detection process in YOLOv5 can be broken down into the following steps:

Pre-processing: The input image is resized to a square format without altering the aspect ratio. This ensures that the detection algorithm works consistently across different sized images.

Feature Extraction: The processed image is passed through the CSPDarknet53 backbone to extract low-level features. These features are then passed through the PANet and FPN to extract multi-scale features.

Detection: The feature maps are passed to the detection head, where the anchor boxes are used to predict the class probabilities and bounding box coordinates.

Post-processing: The raw output from the detector includes a number of potential detections. However, many of these are false positives, or multiple detections for the same object. Thus, a technique called non-maximum suppression (NMS) is used to remove these duplicate detections and retain only the most probable one.

Output: The model outputs the class labels and bounding box coordinates for each detected object.

YOLOv5's performance, speed, and ease of use make it an excellent choice for object detection tasks, especially for real-time applications. It also provides a number of different sized models (YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) that balance speed and accuracy according to your specific need

3.2.4 Object Detection with YOLOv5s/m

To detect the moving packages, we first used the YOLOv5s/m model, a very efficient and accurate model for object detection. This model detects the packages and outputs a bounding box for each package moving on the conveyor.

The class ObjectDetection is implemented in Python and uses the PyTorch library to load and use the YOLOv5s/m model for object detection. The class has an initializer __init__ that accepts parameters such as minimum confidence and box color and initializes the model and some other class variables.

The class also has a DetectObjects method which accepts an image frame as a parameter, runs the frame through the YOLO model, and returns an array of positions and labels, as well as the final image. The Draw_boxes_on_frame function is used to draw bounding boxes on the frame.

Pseudo code for the class is as follows:

```
Class ObjectDetection
  Initialize with parameters: minimum confidence, box color
  Load the YOLO model
  Define classes using the model's names

  Function DetectObjects takes an image frame
    Pass image to the model
    Get labels and coordinates from the model's output
    Return results, labels, and coordinates

  Function Draw_boxes_on_frame takes frame, labels, coordinates
```

```
    Loop over each label
      If confidence is above the minimum confidence
        Draw a rectangle on the frame
    Return the frame


  Function ToLabel converts a label from integer to string using the
classes
```

3.2.5 Object Centroid Tracker

After detecting the packages, we need to track their centers and see if they enter the warehouse by passing across a certain threshold. For this, an object centroid tracker was implemented.

The TrackableObject class represents a trackable object, with an ID, class, list of centroids (for storing history), and a boolean to determine if it has been counted.

The ObjectTracker class maintains the list of tracked objects and manages their registration and deregistration. This class uses the Euclidean distance between the old centroid and the new centroid to match the detected objects to the already tracked objects.

Pseudo code for the ObjectTracker class would be as follows:

```
Class ObjectTracker
  Initialize with maximum disappear frames
  Create an ordered dictionary for objects and disappeared frames


  Function Register accepts centroid, label
    Register a new object with the next available ID


  Function Deregister accepts object ID
```

```
        Remove an object from the tracking list

    Function Update accepts bounding boxes, labels, frame width, frame height
        Calculate new centroids for bounding boxes
        If no object is being tracked, register new objects
        Else, calculate minimum distance between new and old centroids
        Update the matched objects, and register or deregister objects as needed
```

3.2.6 Object Counting

For object counting, a script is implemented which applies the object detection and tracking techniques discussed above. The script starts a video stream and initializes the object detection and tracking classes. In each frame, it detects objects, updates the object tracker, draws a line on the frame, and iterates over each tracked object to draw its trajectory and ID.

The main novelty in this script is that it calculates the direction of movement for each object, and if the object crosses the middle line of the frame, the total count of objects is incremented or decremented based on the direction of movement.

Pseudo code for the script is as follows:

```
Class ObjectCounting
Set width, height, tail length, total frames, and total objects
Initialize video stream
Initialize object detector and tracker

While true
   Read frame from video stream
   Detect objects in the frame
   Update the object tracker
   Draw a line on the frame
```

```
    Iterate over each object in tracked objects
      Get object label and ID
      Draw the object trajectory
      If the object has not been counted and crosses the midline
        Increment or decrement total objects based on direction
    Display the frame
    Increment total frames
  Close video stream and destroy all windows
```

```

In conclusion, the combination of YOLOv5s/m for object detection and centroid tracking for object tracking provides a robust and efficient solution for tracking and counting objects in a video stream.

### 3.2.7 Qr-code Detection and decoding

The script begins by importing necessary libraries such as cv2, pyzbar, and a custom module RosCamera , which we previously discussed . The cv2 library is used for image processing, pyzbar for decoding barcodes and QR codes, and RosCamera for fetching camera data from ROS (Robot Operating System).

cv2.QRCodeDetector() is a class in OpenCV that provides methods to detect and decode QR codes in an image. The detectAndDecodeMulti() function is used to detect and decode multiple QR codes in a single frame.

In the while loop, the camera data is continuously fetched from RosCamera. For each frame, if it is not empty, it uses cv2.QRCodeDetector() to detect and decode any QR codes present, and pyzbar to detect barcodes. If any QR codes are detected, it prints the decoded information, which typically contains the data embedded in the QR code and its type.

For each detected QR code, it prints the QR code's type and data, and highlights the QR code in the frame with a green color if the QR code is valid, or a red color otherwise. If the QR code is valid, it uses pyzbar to further decode the QR code and prints the decoded text.
```

If 'q' is pressed on the keyboard, it breaks the loop and destroys the window, ending the program.

Pseudo code :
```
Import necessary libraries

Initialize QR Code Detector
Print "starting ros camera"
Initialize RosCamera with the specified topic
Print "Camera initialized"

While True:
    Fetch the camera data
    If the camera data frame is not empty:
        Use QRCodeDetector to detect and decode QR codes in the frame
        Decode barcodes in the frame
        Print detected barcodes
        If any QR code is detected:
            Print decoded QR code information
            For each decoded QR code and corresponding points in the frame:
                If the decoded QR code is valid:
                    Print QR code type and data
                    Highlight the QR code in the frame with green color
                    Decode the QR code using pyzbar
                    If the decoded QR code is valid:
                        Print the decoded QR code text
                    And send a signal for the simulation via ROS 2 that a package has been    scanned
                Else:
```

```
              Highlight the QR code in the frame with red color
          Display the frame with highlighted QR codes
      If 'q' is pressed on the keyboard:
        Break the loop


    Destroy the window
```

3.2.8 Fire Detection

In the context of modern warehousing, one must pay paramount attention to safety due to the potential presence of hazardous materials within the facility. The proximity at which items are stored increases the propensity for incidents, particularly fires, to rapidly escalate, thereby placing the infrastructure, personnel, and inventory at significant risk.

Addressing this exigent challenge, we have integrated a fire detection capability into our system, utilizing the robust and efficient YOLOv5 model, which we previously discussed in depth. This added class aims to provide an early warning system, alerting personnel to initiate fire safety protocols and mitigate potential damages.

Training a YOLO model to recognize a new class - fire in our case - involves several steps, which we will now discuss.

The initial stage entails gathering a fire dataset comprising various instances of fire in different environments, scales, and lighting conditions. This dataset should be sufficiently diverse and representative of real-world scenarios that the model could encounter in an actual warehouse.

Next, the images in the dataset need to be annotated, labeling the regions where fire is present. This process forms the 'ground truth' against which the model's predictions will be evaluated and refined during the training process. Several open-source tools like LabelImg or Roboflow can be utilized for this purpose.

With the annotated fire dataset, we then proceed to re-train our YOLOv5 model using transfer learning. This approach allows us to leverage the pre-trained weights of the model, which has already learned to detect numerous objects from diverse backgrounds. Transfer learning can significantly speed up the training process and improve the model's performance, particularly when the new dataset is relatively small.

During training, the model learns to associate the features of the input images with the corresponding labels - in this case, the presence of fire. It does this by iteratively adjusting the weights of the model to minimize the difference between its predictions and the ground truth. The training process continues until the model's performance on a validation set, which it has not seen during training, meets our satisfaction.

Post-training, we conduct rigorous testing on separate, unseen images to ensure that the model can generalize well to new data. Any inaccuracies are scrutinized and if needed, adjustments are made to the model or the training data, and the model is re-trained.

Incorporating fire detection into our system therefore presents a proactive solution to potentially catastrophic events, enhancing the overall safety of warehouse operations. This is a testament to the versatility and efficacy of the YOLOv5 model, demonstrating its wide applicability in real-world scenarios beyond traditional object detection tasks.

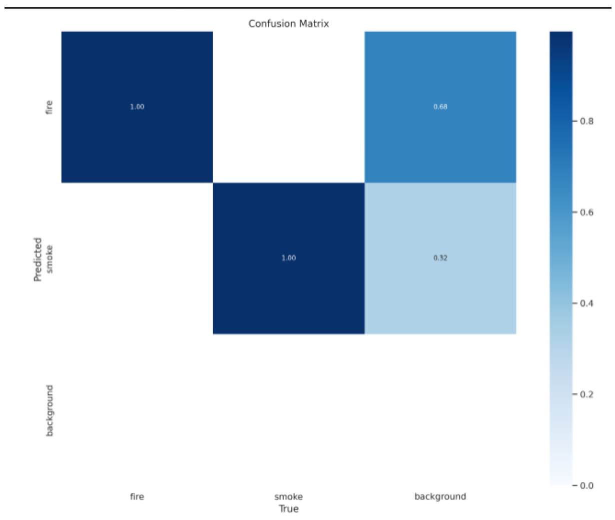


Figure 5 Network architecture for YOLO v5

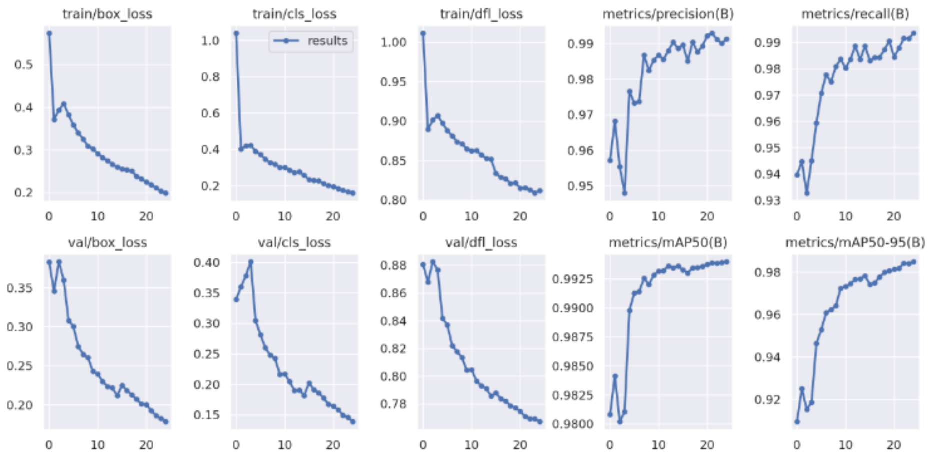


Figure 6 Yolo custom training metrics

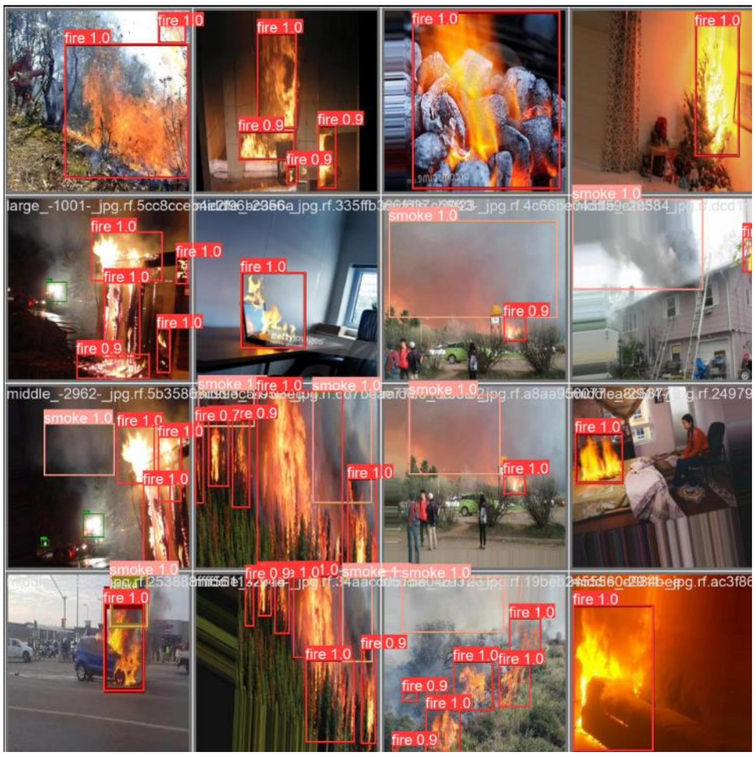


Figure 7 Yolo custom dataset validation

Chapter 4
Simulation

4.1 Gazebo vs Unity

Both Unity and Gazebo are powerful tools used for simulating robotic systems. However, each comes with its own unique features and capabilities, making them suitable for different applications. Below, we provide a comparative analysis between Unity and Gazebo for the purpose of robotics simulation:

4.1.1 Realism and Visual Quality

Unity: Known for its high-fidelity graphics, Unity is capable of producing photorealistic environments with its real-time rendering and ray tracing features. This high visual and sensor fidelity makes it an excellent choice for simulations where visual realism is crucial.

Gazebo: While Gazebo also supports high-quality rendering, its focus is more on physical accuracy than visual realism. It is capable of simulating illumination (both directional and spotlights), and offers a variety of cameras (like depth, multicamera, and GPU-lens camera).

4.1.2 Physics Engine

Unity: Unity employs the PhysX engine developed by Nvidia. It includes a new articulation joint system and support for the Temporal Gauss-Seidel Solver (TGS), which helps in producing more realistic and accurate physics interactions.

Gazebo: Gazebo can utilize multiple physics engines, including ODE (Open Dynamics Engine), Bullet, Sim body, and DART. This flexibility allows users to select an engine based on the specific requirements of their simulations.

4.1.3 Ease of Use and Learning Curve

Unity: With its user-friendly interface, abundant resources, and large community support, Unity is generally easier to learn and use, especially for those who are already familiar with game development.

Gazebo: Gazebo has a steeper learning curve, and its interface might not be as intuitive for beginners. However, it offers extensive documentation and has a strong community of robotics researchers and developers.

4.1.4 Assets and Environment Modelling

Unity: Unity has a vast Asset Store with millions of assets that can be used to design and model environments, which can significantly speed up prototyping. It also supports a broader range of file formats for importing custom models.

Gazebo: Gazebo also offers a variety of models in its model database. However, it has less variety compared to Unity. Modelling environments may require more manual effort and time in Gazebo.

4.1.5 Real-time Simulation

Unity: Unity provides real-time simulation and is optimized for performance on a variety of platforms, including mobile devices.

Gazebo: Gazebo's simulations are more computationally intensive due to their focus on physical realism. As a result, they may not always run in real-time, especially for complex simulations.

4.2 Simulation tool: Unity 3D

For the successful completion of our graduation project, we leveraged Unity, a powerful and widely used game engine, as our primary simulation tool. This approach allowed us to simulate various components of our project, including the motors, sensors, algorithms, and even the robots themselves. This comprehensive simulation environment was essential to the efficient testing and validation of multiple scenarios, specifically in the context of a multi-agent system. It served as a cost-effective and practical solution, eliminating the immediate need for constructing physical robotic hardware.

Prior to physically realizing these solutions, it is of the utmost importance that roboticists and developers have at their disposal an experimentation platform that mirrors the real-world environment as accurately as possible. This includes replicating the robot's physical interactions with said environment. By enabling this level of simulation, developers are given the opportunity to assess the robot's performance in terms of localization, motion planning, and control, all within a safe and modifiable virtual space.

Unity, our chosen platform, offers a multitude of advantages for this type of work. Firstly, it enables the creation and modeling of highly customizable real-world situations, allowing us to design and run simulations that accurately reflect the complex environments our robots would potentially operate in.

Further, Unity boasts high-fidelity physics capabilities, featuring support for the PhysX SDK 4 and advancements such as the TGS solver and a new articulation joint system. These physics enhancements are integral to creating and exploring real-world robotic behaviors and interactions within the simulated environments.

Lastly, Unity offers high visual and sensor fidelity, real-time rendering, and ray tracing capabilities. These are essential for creating photorealistic replicas of real-world environments. In a robotics simulation, realism is key as it directly impacts the quality and applicability of the simulation results.

In addition to Unity, we also incorporated the Robot Operating System 2 (ROS 2) into our simulation workflow. ROS 2 served as a bridge between the simulated robots in Unity and the practical robotic applications, helping us emulate how an actual robot would interact with the ROS 2 package.

Eventually, the developed system was deployed on a physical robot, acting as a proof of concept to validate the theories, designs, and algorithms that were initially tested in our Unity-based simulation environment. This crucial step helped us ensure that our concepts and design could indeed be applied in real-world scenarios, illustrating the essential role that simulation plays in the design, testing, and deployment of robotic systems.

4.3 ROS-TCP Connection

The ROS TCP Connector is an open-source package that functions as a TCP endpoint in Unity for receiving and sending ROS messages. This package was developed with the objective of enabling Unity projects to interact with ROS systems without the need for ROS# or any other third-party plugins.

Developed in C#, the ROS TCP Connector relies on TCP communication to ensure reliable, ordered, and error-checked delivery of a stream of bytes. This is instrumental in robot simulations and controls were dropped or out-of-order packets can significantly affect the system's performance.

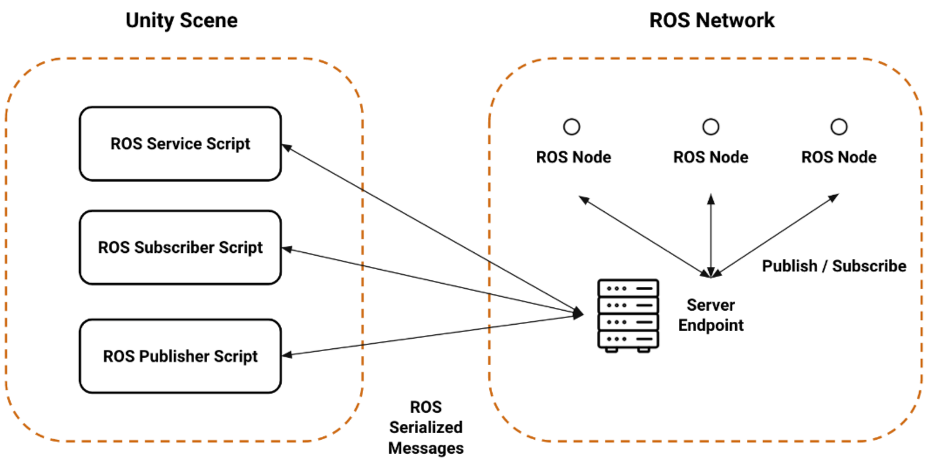


Figure 8 ROS–Unity Integration

4.3.1 Technical Overview

At its core, the ROS TCP Connector consists of two main components: the ROS TCP Endpoint and the Message Generation system.

ROS TCP Endpoint: This is a ROS node that serves as the main communication point between ROS and Unity. It is responsible for receiving connections from Unity, parsing the incoming messages, and then publishing those messages to the correct ROS topics. It can also subscribe to any ROS topics and send the data back to Unity.

Message Generation system: This system enables automatic conversion between ROS message types and Unity message types. This is achieved through a custom message generation pipeline that uses Python scripts to parse ROS .msg and .srv files, and then generates corresponding C# classes to be used in Unity.

4.3.2 Applications and Advantages

With the ROS TCP Connector, developers can create complex robotic simulations in Unity that can interact directly with ROS systems. This allows the utilization of Unity's advanced rendering and physics capabilities for creating realistic and accurate simulations, which can be invaluable for prototyping, testing, and development of robotic systems.

One key advantage of the ROS TCP Connector is its ability to handle custom ROS messages. It can generate the necessary C# classes for any ROS message types that are not included in its standard library, which makes it a highly flexible tool for developing complex ROS-Unity applications.

Another significant benefit of the ROS TCP Connector is that it enables real-time simulations in Unity to interact with ROS. This makes it possible to test and develop ROS-based systems in a visually rich and interactive environment, which can be especially beneficial for projects involving autonomous robots, where visualization and interaction are critical.

4.4 Simulated Sensors

4.4.1 Camera Sensor

We previously Discussed in the Computer Vision section about our ROSCamera interface, now we will be talking about the implementation of that particular class from the simulation.

The ImagePublisher class inherits from MonoBehaviour, Unity's base class for all scripts, and it uses multiple namespaces, including standard collections and Unity-specific libraries. It also uses the Unity.Robotics.ROSTCPConnector namespace, which provides the ROSConnection class for communicating with ROS.

Key variables in the class include the ROSConnection object 'ros', which manages the connection to ROS, and a Camera object 'imageCamera'. The resolution of the image and its quality level are defined, as well as a string to specify the ROS topic where the image data will be published.

In the Start() method, which is a Unity-specific method that is called before the first frame update, the ROSConnection instance is fetched and registered as a publisher of CompressedImageMsg messages on the defined topic. The method also initializes a new Texture2D and a RenderTexture, which are used to capture the image from the camera view. The CompressedImageMsg object 'compressedImage' is then initialized, with its 'frame_id' field set to a specified value and its 'format' field set to "jpeg".

A callback function, UpdateImage(), is registered with the Camera.onPostRender event. This function is called after the camera has finished rendering, which ensures that the most up-to-date view from the camera is captured. In this function, the current camera view is read into the Texture2D object, encoded into JPEG format with the specified quality level, and then assigned to the 'data' field of 'compressedImage'. The updated CompressedImageMsg is then published to the ROS topic.

4.4.2 Laser Sensor

The LaserScanPublisher class inherits from MonoBehaviour, which is Unity's base class for scripts that need to interact with the Unity game engine. It uses namespaces that include standard collections, Unity-specific libraries, and the Unity.Robotics.ROSTCPConnector namespace for establishing communication with ROS.

Key variables in the class include the ROSConnection object 'ros', which maintains the connection with ROS, a Laser object 'laser' that represents the simulated LiDAR sensor, and a string 'laserTopicName' to define the ROS topic where the Laser Scan data will be published.

The Start() method, a Unity-specific method that is called before the first frame update, gets the instance of ROSConnection, and initializes the LaserScanMsg object 'laserScan'. This message object contains all the data pertinent to a laser scan, including the angle range, time increments, minimum and maximum range values, range measurements, and intensity values. The InvokeRepeating() function is also called in the Start() method to repeatedly call the PublishScan() method at a rate specified by 'publishRate'.

The PublishScan() method updates the 'header' field of 'laserScan' with the current time, obtains the current laser scan ranges by calling the GetCurrentScanRanges() method of the 'laser' object, and assigns these ranges to the 'ranges' field of 'laserScan'. The updated LaserScanMsg is then sent to the specified ROS topic using the Send() method of the 'ros' object.

4.4.3 Robot TF Publisher

The TransformTreeNode class is a custom class within the Unity.Robotics.SlamExample namespace. It maintains a reference to a GameObject and a list of child nodes in the transformation tree. The Transform property of the class returns the transformation of the GameObject associated with the node. The class also includes methods for converting a node's transformation to a ROS-compatible TransformStampedMsg and populating child nodes.

The class constructor takes a GameObject as a parameter, initializes its Children list, and populates it by calling the PopulateChildNodes() method. This method iterates over all the children of the input GameObject's Transform. If a child GameObject has a UrdfLink component attached, indicating it's a part of the transform tree, it creates a new TransformTreeNode for that GameObject and adds it to the Children list.

The ToTransformStamped() method converts a TransformTreeNode object's Transform into a ROS-compatible TransformStampedMsg, which represents a transformation between two coordinate frames in a time-stamped format. This helps to communicate the transformation data between the Unity environment and ROS.

4.4.4 Use of TF in ROS

In the Robot Operating System (ROS), the Transform (TF) package is a key component that allows the user to maintain the relationship between multiple coordinate frames over time. It's essentially a tree of coordinate frames, where each frame is connected to one or more other frames. For example, in a robotic system, there might be separate coordinate frames for the robot's base, its manipulator arm, its gripper, etc.

With TF, you can ask questions like, "Where was the robot's gripper relative to its base five seconds ago?", or "What is the current relative position of the robot's camera and its wheel?". This is important for complex robotics operations, where knowing the precise spatial relationship between different parts of the robot is crucial.

In conclusion, the TransformTreeNode class in the Unity environment performs a similar role to TF in ROS, helping to maintain a structured understanding of the spatial relationships between different objects or points in a simulation. By offering the ability to convert these transformations into a format compatible with ROS, it further enables a powerful interface between Unity-based robotics simulations and ROS.

4.4.5 ROS Clock Publisher

Finally, to keep the Simulation and ROS in sync, we need to send the Simulation time continuously . The main responsibility of this class is to continuously publish the current Unity simulation time (which could be system clock time or simulated time) as a ROS Clock message. The frequency of publication is controlled by the PublishRateHz parameter.

In detail, upon initialization (Start method), an instance of ROSConnection is fetched or created. This ROSConnection instance is used to register a publisher for the topic "clock", with message type ClockMsg.

The Update method, which is called once per frame in Unity, checks if it is time to publish another message using the ShouldPublishMessage boolean property. If it is time, PublishMessage is invoked. This method assembles a ClockMsg containing the current time in seconds and nanoseconds, updates the LastPublishTimeSeconds field, and publishes the message to the "clock" topic.

The class also contains a ClockMode property, which determines whether to use the system clock or Unity's game time as the source for the published ROS clock messages. This mode can be set in the Unity editor but not changed during runtime.

4.5 Simulated Robots

4.5.1 Robot Arm

Our first robot we simulated was the robot arm, used in picking and placing objects from one place to another , this is done by writing our own implementation of Jacobian IK.

Inverse Kinematics

Inverse kinematics (IK) plays a pivotal role in robotics, particularly in the control of robotic arms. This paper provides an in-depth look at the implementation of the Jacobian method for inverse kinematics using a six-degrees-of-freedom (DOF) robotic arm. The approach centers around the C# script utilized in the Unity engine, focusing on the 'JointController' class, which implements the IK solver. This paper seeks to dissect the code, offering an insight into the key components and their roles in the IK process.

Introduction:

Inverse Kinematics (IK) refers to the computational process of determining the joint parameters necessary to place the end-effector (the device at the end of a robotic arm) in a desired pose. The Jacobian IK method, named after the Jacobian matrix, is a popular approach to solving IK problems, primarily due to its effectiveness in real-time applications.

In this context, the Jacobian matrix represents the derivative of the end-effector's position concerning its joint angles. The Jacobian IK method employs this matrix to iteratively adjust joint angles, reducing the discrepancy between the current and desired end-effector position until a predetermined error threshold is reached.

Methodology:

The central class for the IK solver is the 'JointController'. This class contains the core components of the Jacobian IK method, specifically the forward kinematics, Jacobian matrix calculation, error calculation, and joint angle correction.

Forward Kinematics:

The first step in the IK process is the computation of forward kinematics, which is the task of determining the position and orientation of the end-effector given the joint angles. In the 'JointController' class, the 'ForwardKinematics()' method is responsible for this task.

The method computes the world position of each joint by iteratively transforming the local position of each joint to world coordinates using the quaternion representation of rotations. The quaternion rotation is calculated from the current joint angle and the axis of rotation.

Error Calculation:

The 'CalcErr()' method computes the position and orientation error between the current end-effector position and the desired position. The position error is simply the vector difference between the target and current position. The orientation error is calculated by creating a quaternion representing the desired orientation, and then taking the inverse of the current orientation quaternion and multiplying it with the desired orientation quaternion.

Jacobian Matrix Calculation:

The Jacobian matrix is calculated in the 'CalcJacobian()' method. The linear velocity component of the Jacobian is obtained by calculating the cross product between the rotation axis of each joint and the vector from the joint to the end effector. The angular velocity component of the Jacobian is simply the rotation axis of each joint. The Jacobian matrix is then constructed by stacking the linear and angular velocity components.

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_A} & \frac{\partial p_x}{\partial \theta_B} & \frac{\partial p_x}{\partial \theta_C} \\ \frac{\partial p_y}{\partial \theta_A} & \frac{\partial p_y}{\partial \theta_B} & \frac{\partial p_y}{\partial \theta_C} \\ \frac{\partial p_z}{\partial \theta_A} & \frac{\partial p_z}{\partial \theta_B} & \frac{\partial p_z}{\partial \theta_C} \end{bmatrix}$$

Figure 9 Jacobian Matrix

Joint Angle Correction:

The Jacobian matrix is used to iteratively adjust the joint angles in the 'CalcIK()' method. First, the Jacobian pseudo-inverse is computed. Then, the method multiplies the pseudo-inverse with the position and orientation error to calculate the change in joint angles. The joint angles are updated by adding the change in joint angles to the current joint angles.

Conclusion:

The implementation of the Jacobian method in the 'JointController' class provides an efficient solution for inverse kinematics in a 6-DOF robotic arm. By iteratively adjusting the joint angles, the Jacobian IK method allows for real-time control of the robotic arm, enabling the arm to accurately reach its target position and orientation. The presented script provides a valuable resource for developers and researchers interested in employing the Jacobian IK method in robotics applications.

4.5.2 Autonomous Mobile Robot

we will discuss an autonomous mobile robot equipped with a virtual laser sensor and an AGV (Automated Guided Vehicle) controller class designed to manage the robot's movement. The navigation algorithm is designed to output two floating point numbers representing the linear and angular velocities.

Sensory Systems

The mobile robot is equipped with a virtual laser sensor. This sensor is responsible for detecting and relaying information about obstacles in the robot's environment. The sensor's role is crucial in path planning and navigation, providing data that the robot can use to avoid obstacles and reach its target destination.

AGV Controller Class

The AGV Controller Class is a critical part of the robot's control mechanism, which translates navigation outputs into actual motion. It accomplishes this by managing the robot's motor operations to achieve the target velocities set by the navigation algorithm. The final output of the navigation algorithm is a pair of floating-point numbers, each representing the robot's desired linear and angular velocities.

Code Implementation

Written in Unity's C# language, the AGV Controller class consists of several components. It communicates with the robot through either keyboard inputs or via ROS (Robot Operating System) messages, depending on the chosen control mode.

At the start, the AGV Controller initializes the articulation bodies of the two wheels and sets the parameters of the wheel joints, which includes the force limit and damping. It also subscribes to the ROS topic "cmd_vel" to receive commands if the control mode is set to ROS.

Two significant methods exist in this class: KeyBoardUpdate() and ROSUpdate(). KeyBoardUpdate() responds to inputs from the keyboard, changing the robot's speed and rotation based on user commands. Conversely, ROSUpdate() processes the ROS messages and updates the robot's motion accordingly.

A critical function in this class is RobotInput(float speed, float rotSpeed). This function takes the linear and rotational speeds as inputs (in m/s and rad/s respectively). It ensures these values do not exceed the maximum limit. The function then computes the rotation of each wheel by considering the speed, rotational speed, and other characteristics of the robot such as track width and wheel radius. These computations allow the robot to turn effectively and manage its linear velocity. The computed wheel rotations are then applied to the wheels, effectively controlling the robot's movement.

In conclusion, this autonomous mobile robot combines sophisticated sensory systems with a powerful control class. The virtual laser sensor allows for environmental awareness, necessary for safe and efficient navigation. The AGV Controller class converts the navigation algorithm's output into actual robot motion. It can receive commands from both manual keyboard inputs and ROS messages, making it versatile for different control scenarios. These aspects work in harmony to provide a robust foundation for autonomous navigation. Future work could investigate refining the control algorithms or adding other types of sensors to improve navigation in complex environments.

Navigation Approach A: nav2 stack

The application of autonomous navigation principles to mobile robots necessitates a reliable and efficient mechanism to generate, process, and follow a pathway towards a set target. In our research, we utilized the Nav2 Stack as our first approach to autonomous navigation. Although an in-depth discussion of the Nav2 Stack is beyond the scope of this section, we will elaborate on the simulation components used for this approach, namely, the Goal Sender, AGV Controller's ROS mode, LaserScan, and TF (transform) poses of the robot.

Goal Sender

The Goal Sender plays an integral role in setting the target position for the robot. It operates by sending a stampedTransform message to the robot's topic using the Unity TCP ROS Connector. This message specifies the robot's goal pose, marking the destination the robot aims to reach. The Goal Sender class's simplicity belies its crucial function in guiding the autonomous robot towards its goal.

AGV Controller's ROS mode

As outlined in the previous discussion of the AGV Controller class, it operates in two modes: Keyboard and ROS. For the Nav2 Stack approach, we primarily rely on the ROS mode. In this mode, the AGV Controller reads the target linear and angular velocities from the cmd_vel ROS topic.

The cmd_vel topic is typically used by ROS-based packages to dictate a robot's movement. In our case, these velocities are the result of a PID (Proportional Integral Derivative) controller. This controller is a common technique in control systems, as it helps reduce the error between the robot's current state and its desired state by adjusting the control inputs.

Laser Scan

LaserScan is an essential component for the robot's environmental perception. It sends messages on the scan topic, representing sensor data from the robot's onboard laser scanner. This data provides crucial insights into the robot's surroundings, aiding in obstacle detection and avoidance and thereby ensuring safe navigation.

TF Poses

The robot's TF (transform) poses are sent to the Nav2 Stack as feedback. TF is a package in ROS that lets the user keep track of multiple coordinate frames over time. It maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc., between any two coordinate frames at any desired point in time.

In this context, the robot's TF poses, which describe its position and orientation in the coordinate frame, are critical for monitoring the robot's real-time state and comparing it with the desired state. Any deviation can then be corrected using the PID controller, enabling the robot to follow the set path towards its goal.

Conclusion

In summary, the application of Nav2 Stack for autonomous navigation involved an orchestration of different simulation components, each playing a vital role in the process. The Goal Sender sets the target, the AGV Controller (in ROS mode) dictates the robot's movements, Laser Scan provides environmental feedback, and the TF poses keep track of the robot's state.

Navigation Approach B: Reinforcement Learning (Examination of Proximal Policy Optimization Algorithms for Autonomous Navigation)

Deep reinforcement learning has shown significant potential in a wide array of domains, from gaming to real-world applications like autonomous navigation. An exemplar among the diverse algorithms in this field is Proximal Policy Optimization (PPO), proposed by John Schulman et al. (2017) from OpenAI. This paper examines the technical aspects of PPO, its advantages, and its application to autonomous navigation, demonstrating the immense promise this technique holds in robotics.

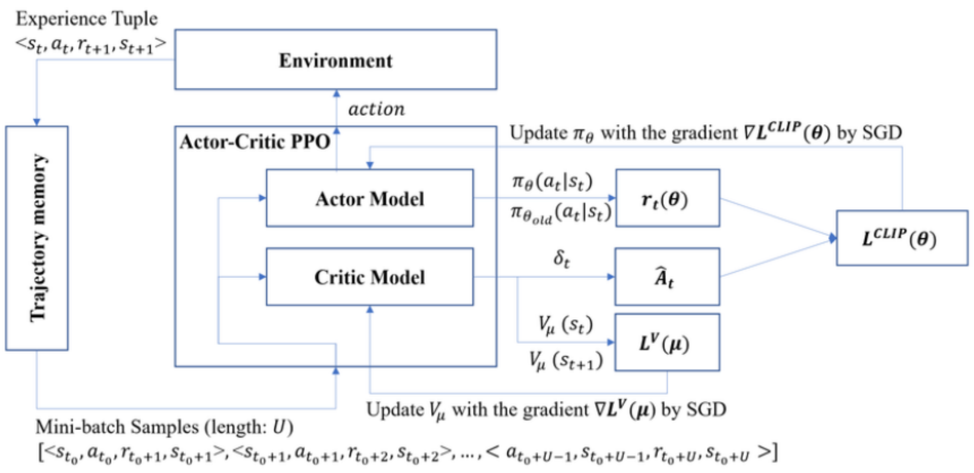
Reinforcement learning (RL) deals with sequential decision-making problems, whereby an agent learns to perform actions to maximize cumulative reward in a given environment. Policy optimization algorithms are a subset of RL, which directly optimize the policy of the agent. However, they often face stability and efficiency issues. Proximal Policy Optimization (PPO) was proposed to address these challenges. By restraining policy updates to be relatively small, PPO improves the stability of training while maintaining substantial performance.

Proximal Policy Optimization: An Overview

PPO algorithms leverage the strengths of first-order policy optimization methods, such as stochastic gradient ascent, and second-order methods, like Trust Region Policy Optimization (TRPO), without their respective drawbacks of sample inefficiency and computational expense.

The primary technical novelty of PPO is the use of a surrogate objective function for policy update. This function adds a constraint to policy updates, limiting the difference between the new and old policy at each step (as measured by the KL-divergence), to avoid harmful large updates. It employs a clipped version of the policy ratio (new policy probability over old policy probability), which eliminates the excessive incentive to move probability mass to or from actions whose estimated advantage has changed.

The PPO Algorithm



Experience Tuple
$<s_t, a_t, r_{t+1}, s_{t+1}>$
Environment
action
Actor-Critic PPO
Trajectory memory
Actor Model
Critic Model
Update π_θ with the gradient $\nabla L^{CLIP}(\theta)$ by SGD
$\pi_\theta(a_t|s_t)$
$\pi_{\theta_{old}}(a_t|s_t)$
$r_t(\theta)$
$L^{CLIP}(\theta)$
δ_t
$\hat{A}_t$
$V_\mu(s_t)$
$V_\mu(s_{t+1})$
$L^V(\mu)$
Update V_μ with the gradient $\nabla L^V(\mu)$ by SGD
Mini-batch Samples (length: U)
$[<s_{t_0}, a_{t_0}, r_{t_0+1}, s_{t_0+1}>, <s_{t_0+1}, a_{t_0+1}, r_{t_0+2}, s_{t_0+2}>, \ldots, < a_{t_0+U-1}, s_{t_0+U-1}, r_{t_0+U}, s_{t_0+U} >]$

Figure 10 The actor-critic proximal policy optimization (Actor-Critic PPO) algorithm process.

The PPO algorithm follows a procedure akin to policy iteration. Starting with an initial policy, it collects a batch of trajectories by interacting with the environment, estimates the advantages of the state-action pairs in the trajectories, and optimizes the clipped surrogate objective function using stochastic gradient ascent. It performs multiple epochs of optimization over the same batch of trajectories, updating the policy after each epoch.

The surrogate objective function is constructed to provide a lower bound to the expected future rewards. By optimizing this surrogate function, the policy is updated in a way that is expected to increase the future rewards. The clipping mechanism then assures that the updated policy does not deviate excessively from the old policy.

PPO Advantages and Challenges

PPO algorithms have several advantages that make them a promising choice for a wide range of applications. They maintain a balance between sample complexity, computational complexity, and ease of implementation. They are relatively robust to hyperparameter settings, and they perform comparably or better than state-of-the-art policy optimization methods on a variety of tasks.

However, PPO also faces challenges. Despite the clip in the objective function, the agent can sometimes still get stuck in suboptimal policies. Also, PPO, like all deep reinforcement learning methods, requires a considerable amount of data to learn effectively, which may be challenging in complex environments or when learning from scratch.

Application to Autonomous Navigation

In our second approach to autonomous navigation, we utilized PPO to train the robot. PPO's advantages, particularly its ability to efficiently use samples and its robustness, make it an appealing choice for training an autonomous agent.

With an appropriate reward function, the robot learns to navigate towards the target while avoiding obstacles. It is a challenging problem due to its high-dimensional state and action spaces and the delayed and sparse reward structure. However, through the iterative process of trial and error and subsequent policy optimization, the robot gradually learns an effective policy to navigate in the environment.

Observations

Observations are collected in the CollectObservations() function. The agent observes the following data:

Robot position: The current position of the robot in a 2D plane, (x, z).

Robot rotation: The current rotation of the robot, given as Euler angles.

Target position: The position of the target in the same 2D plane, (x, z).

Direction vector: A normalized vector pointing from the robot's current position to the target.

Reward Function

The reward function is determined within the OnActionReceived() and OnCollisionEnter(), OnTriggerStay(), OnCollisionStay() functions.

Distance reward: Every time an action is taken, the agent is rewarded based on its distance to the target. The closer the robot is to the target, the higher the reward.

Time penalty: A small penalty is deducted for each time step to encourage the agent to reach the goal as quickly as possible.

Boundary collision penalty: If the agent collides with a boundary (tagged as "Boundries"), a significant negative reward is given, and the episode ends.

Goal reward: If the agent reaches the goal (collides with an object tagged as "Goal"), it receives a large positive reward. The agent's status is then marked as having reached the target.

Collision penalty: If the agent stays in collision with an object (tagged as "CollisionObjects"), it receives a moderate negative reward.

Heuristic Function

The Heuristic() function allows manual control of the agent for testing purposes. It takes the vertical and horizontal input from the keyboard and feeds it into the action array. The vertical input is mapped to linear velocity and the horizontal input to angular velocity.

OnEpisodeBegin Function

This function resets the state of the robot at the beginning of each new episode. The current episode reward is reset to 0 and the hasReachedTarget flag is set to false. If a multi-agent scenario is in place, it also calls the corresponding function.

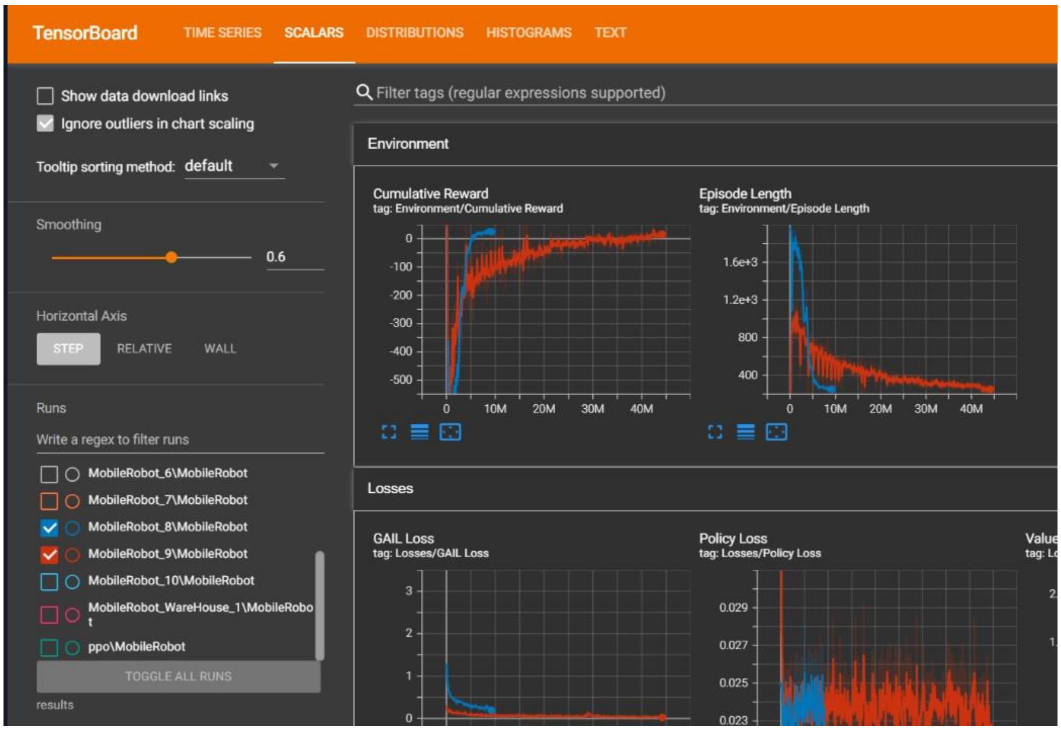
setRobotVelocity Function

The setRobotVelocity() function is used to send the calculated velocities (linear and angular) to the robot controller. The velocities are passed through the RobotInput() function of the robot controller.

The overall aim of this agent behaviour is to help the robot learn to reach the goal as quickly as possible, avoiding obstacles and staying within boundaries, by optimizing the actions based on the rewards provided by the environment.

Results

It took roughly 40 Million steps before applying Imitation learning using Demonstration Recorder that unity provides.
after using the imitation learning method with reward percentage of 10% , the network reached its goal in approximately 10 Million steps (Roughly 4 times faster)



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs
Write a regex to filter runs
☐ MobileRobot_6\MobileRobot
☐ MobileRobot_7\MobileRobot
☑ MobileRobot_8\MobileRobot
☑ MobileRobot_9\MobileRobot
☐ MobileRobot_10\MobileRobot
☐ MobileRobot_WareHouse_1\MobileRobot
☐ ppo\MobileRobot

Figure 11 PPO with imitation acceleration

Conclusion

The Proximal Policy Optimization algorithm has proven to be a robust and efficient policy optimization method. Its balance between efficiency, performance, and ease of implementation makes it a powerful tool for complex tasks such as autonomous navigation. Despite facing challenges, it has shown itself to be a promising avenue for further exploration and improvement. Future work may focus on algorithmic advancements to further enhance the robustness and efficiency of PPO, and on exploiting its strengths for more complex and practical tasks in the real world.

Chapter 5
ROS2

ROS2, also known as Robot Operating System 2, represents a significant milestone in the evolution of robotics operating systems. It serves as an advanced framework that builds upon the successes of its predecessor, ROS, providing a robust platform for academic research in the field of robotics. With its emphasis on improved interoperability, real-time capabilities, security, and performance, ROS2 offers a scalable, flexible, and efficient solution for developing cutting-edge robotic applications. Its modular architecture, extensive tooling, and integration with other frameworks make ROS2 an invaluable resource for researchers seeking to explore novel algorithms, experiment with hardware-software interactions, and collaborate on collaborative research projects. Supported by a vibrant community and gaining popularity in academia, ROS2 empowers researchers to push the boundaries of robotics and drive innovation in this rapidly evolving field.

ROS 2 is designed to be used in a wide variety of robotics applications, including autonomous vehicles, industrial robots, and medical robots. It is a powerful platform that can be used to build complex and sophisticated robotics applications, while introducing new features and capabilities.

1. Evolution of Robotics: ROS2 represents a significant milestone in the evolution of robotics, offering a more robust, scalable, and efficient platform for developing advanced robotic systems.
2. Improved Interoperability: One of the key focuses of ROS2 is enhanced interoperability, enabling seamless communication and collaboration between different robots, components, and systems, promoting a standardized ecosystem.
3. Real-time Capabilities: ROS2 introduces real-time features, enabling the development of applications that require precise timing and synchronization, critical for applications such as autonomous vehicles and industrial automation.
4. Multi-platform Support: ROS2 expands its compatibility by supporting various operating systems, including Linux, Windows, and macOS, broadening its accessibility and allowing developers to leverage their preferred environments.
5. Modular Architecture: With a modular architecture, ROS2 enables developers to create more flexible and scalable robotic systems by decoupling components and facilitating independent development and deployment.
6. Security and Safety: ROS2 emphasizes security and safety by incorporating features such as access control, authentication, and

encryption, addressing the growing concerns of deploying robots in sensitive environments.

7. Industry Adoption: ROS2 has gained substantial traction in the robotics industry, with major players endorsing and contributing to its development, driving its widespread adoption and ensuring a rich ecosystem of libraries and tools.
8. Improved Performance: By leveraging the latest advancements in networking and communication protocols, ROS2 enhances performance, minimizing latency and maximizing throughput, vital for real-time applications.
9. Simplified Development Process: ROS2 introduces an improved development workflow, empowering developers with powerful tools and standardized APIs, simplifying the creation, testing, and deployment of robotic applications.
10. Integration with Other Frameworks: ROS2 offers seamless integration with other software frameworks, such as OpenAI Gym and TensorFlow, enabling developers to combine the power of machine learning and robotics.
11. Support for Heterogeneous Systems: ROS2 enables the integration of heterogeneous systems, allowing robots to interact with a wide range of hardware devices, sensors, and actuators, fostering innovation and versatility.
12. Data Distribution Service (DDS) Integration: ROS2 integrates with the Data Distribution Service (DDS) standard, providing a reliable and efficient mechanism for real-time data exchange, ensuring robust communication in distributed systems.
13. Community-driven Development: Similar to its predecessor, ROS2 benefits from a vibrant and passionate community of developers, researchers, and enthusiasts, contributing to its growth, documentation, and support.
14. Seamless Migration: While ROS2 represents a major leap forward, efforts have been made to ensure a smooth transition from ROS to ROS2, allowing existing ROS users to leverage their knowledge and codebase.
15. Flexibility for Resource-constrained Systems: ROS2 supports resource-constrained systems, enabling the development of lightweight and efficient robotic applications suitable for small-scale platforms.
16. Extensive Tooling: ROS2 provides an extensive suite of development and debugging tools, including visualization tools, simulation environments, and monitoring utilities, facilitating the development and troubleshooting process.
17. Standardization and Compatibility: With a focus on standardization, ROS2 promotes compatibility across different robotic systems, ensuring interoperability between hardware, software, and frameworks.

18. Advanced Middleware: ROS2 utilizes a more advanced middleware infrastructure, allowing for fine-grained control over communication, supporting both traditional publish-subscribe and service-request paradigms.
19. Academic and Research Integration: ROS2 has gained significant popularity in the academic and research communities, serving as a robust platform for experimentation, algorithm development, and collaborative research projects.
20. Future Innovations: ROS2 lays the foundation for future innovations in robotics, providing a flexible and scalable platform that can adapt to the evolving needs of the robotics industry and empower the development of groundbreaking robotic applications.

5.1 Slam

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in robotics that addresses the challenges of autonomously mapping an unknown environment while simultaneously determining the robot's precise location within that environment. SLAM plays a crucial role in enabling robots to navigate and interact intelligently with their surroundings. By utilizing sensor data, such as laser range finders, cameras, or depth sensors, SLAM algorithms enable robots to build accurate maps of their environment while estimating their own position and orientation relative to that map. SLAM has wide-ranging applications in various domains, including autonomous vehicles, robotics, augmented reality, and virtual reality. This introduction provides an overview of the key concepts, techniques, and applications of SLAM, highlighting its significance in advancing the field of robotics and autonomous systems, SLAM is used in a wide variety of applications, including:

- Autonomous vehicles: SLAM is used in autonomous vehicles to build a map of the environment and to track the vehicle's location in the map. This allows the vehicle to navigate safely and efficiently.
- Robotics: SLAM is used in robotics to build a map of the environment and to track the robot's location in the map. This allows the robot to perform tasks such as navigation, object manipulation, and exploration.
- 3D reconstruction: SLAM can be used to reconstruct 3D models of environments. This is useful for applications such as virtual reality and augmented reality.

5.1.1 Slam tool-box

The SLAM Toolbox is a comprehensive suite of software tools and libraries designed to facilitate the development and implementation of Simultaneous Localization and Mapping (SLAM) algorithms. SLAM Toolbox plays a vital role in empowering researchers, engineers, and roboticists to tackle the challenges of mapping unknown environments and accurately localizing robots within those environments. With its rich set of functionalities and extensive support for various sensor types, the SLAM Toolbox has become a go-to resource in the field of robotics for robust mapping and localization solutions.

1. Mapping Capabilities: The SLAM Toolbox offers a wide range of mapping capabilities, allowing users to create detailed representations of the environment in real-time. It supports various mapping techniques such as occupancy grid mapping, feature-based mapping, and point cloud mapping, enabling users to choose the most suitable approach for their specific application.
2. Sensor Fusion: The SLAM Toolbox excels in sensor fusion, seamlessly integrating data from multiple sensors such as lidar, cameras, IMUs, and odometry, to enhance mapping and localization accuracy. By combining information from different sensor modalities, the toolbox enables robust and reliable mapping and localization in dynamic and challenging environments.
3. Localization Algorithms: The SLAM Toolbox provides a collection of state-of-the-art localization algorithms, ranging from probabilistic filters like Extended Kalman Filter (EKF) and Particle Filter (PF), to optimization-based methods such as Iterative Closest Point (ICP) and Graph-SLAM. These algorithms allow precise estimation of the robot's pose within the map, even in the presence of noise, uncertainties, and motion complexities.
4. Real-time Performance: The SLAM Toolbox is designed to deliver real-time performance, efficiently processing sensor data and updating the map and robot pose estimates in a timely manner. By leveraging optimized algorithms and parallel computing techniques, the toolbox ensures high-speed mapping and localization capabilities, enabling robots to operate in dynamic environments and make timely decisions.
5. Simulation and Visualization: Many SLAM Toolboxes provide simulation environments and visualization tools, allowing users to simulate robot motion and sensor data to evaluate and debug SLAM algorithms. Visualizations help users understand the mapping process, inspect the quality of the map, and analyze localization performance, facilitating algorithm development and validation.
6. Hardware Integration: The SLAM Toolbox offers interfaces and support for a wide range of robotic platforms and sensors, enabling seamless integration with existing hardware setups. This flexibility allows researchers and developers to deploy SLAM algorithms on various robotic systems, from ground-based robots to aerial drones, utilizing different sensors based on their application requirements.

7. Education and Learning: The SLAM Toolbox serves as an educational resource for students and enthusiasts interested in understanding and experimenting with SLAM techniques. Its user-friendly interfaces, documentation, and code examples enable newcomers to grasp the fundamentals of SLAM and gain practical experience in mapping and localization algorithms.

5.2 Localization

Localization refers to the process of determining and estimating the precise position and orientation (pose) of a robot or object in a given environment. Localization is a fundamental problem in robotics and plays a crucial role in enabling robots to navigate, interact, and perform tasks autonomously. By leveraging sensor measurements, such as odometry, GPS, laser range finders, or cameras, localization algorithms estimate the robot's pose relative to a known map or in relation to the surrounding objects.

1. Pose Estimation: Localization algorithms estimate the pose of a robot by fusing sensor measurements with prior knowledge or maps of the environment. The goal is to determine the robot's position (x, y) and orientation (theta) accurately.
2. Sensor Integration: Localization algorithms make use of sensor data from various sources, such as odometry, GPS, inertial measurement units (IMUs), lidar, or cameras. By combining information from multiple sensors, the algorithm can compensate for individual sensor limitations and improve the accuracy and robustness of the localization estimate.
3. Probabilistic Approaches: Many localization algorithms employ probabilistic methods, such as Kalman filters, particle filters, or graph-based approaches. These algorithms model the uncertainty associated with sensor measurements and motion dynamics, providing a probabilistic distribution over possible robot poses.
4. Map-based Localization: In map-based localization, the robot matches its sensor measurements against a pre-built map of the environment. This can involve feature-based matching, where distinctive landmarks or visual features are compared, or scan matching, where sensor readings are aligned with the map to determine the robot's pose.
5. Feature-based Localization: Feature-based localization algorithms identify and match distinctive features in the environment to estimate the robot's pose. These features can be visual, such as keypoints in images, or geometric, such as edges or corners in laser scans.

6. Monte Carlo Localization (Particle Filters): Particle filters, also known as Monte Carlo Localization, employ a probabilistic sampling approach to estimate the robot's pose. By representing the robot's possible poses as a set of particles, the filter resamples and weights these particles based on the sensor measurements and motion model, resulting in an accurate pose estimate.
7. Graph-based Localization (Graph-SLAM): Graph-SLAM approaches formulate the localization problem as an optimization on a graph, where nodes represent robot poses, and edges encode constraints between poses based on sensor measurements. By optimizing the graph, the algorithm estimates the most likely set of poses that best align with the measurements and the map.
8. Real-time Localization: Real-time localization is crucial for dynamic environments or applications that require continuous pose updates. To achieve real-time performance, localization algorithms often employ efficient data structures, parallel processing, or optimization techniques to process sensor measurements quickly and update the robot's pose estimate in real-time.

Localization is a critical component of autonomous navigation, robotic mapping, and human-robot interaction. Accurate localization enables robots to navigate safely, avoid obstacles, plan optimal paths, and perform complex tasks in a variety of settings, including outdoor environments, indoor spaces, or even in unstructured and unknown environments. By solving the localization problem, robots can operate autonomously, adapt to changing conditions, and interact seamlessly with their surroundings, paving the way for advancements in robotics and intelligent systems.

5.2.1 AMCL

AMCL stands for Adaptive Monte Carlo Localization. It is a probabilistic localization algorithm that is used in ROS for robots moving in 2D. AMCL works by representing the robot's pose as a distribution of particles, where each particle represents a possible pose of the robot.

AMCL uses a particle filter to update the distribution of particles as the robot moves and takes sensor measurements. The particle filter is a probabilistic algorithm that uses a set of particles to represent the uncertainty in the robot's pose. As the robot moves, the particle filter is updated to reflect the new sensor measurements.

AMCL is a very robust localization algorithm that can be used in a variety of environments. It is also very efficient, which makes it suitable for real-time applications.

Here are some of the features of AMCL:

- Probabilistic: AMCL uses a probabilistic approach to localization, which means that it can handle uncertainty in the robot's pose.
- Adaptive: AMCL is adaptive, which means that it can adjust its behavior to the environment and the robot's motion.
- Efficient: AMCL is efficient, which makes it suitable for real-time applications.

AMCL has proven to be a robust and widely adopted localization algorithm in the robotics community. Its adaptive resampling, probabilistic framework, and integration with sensor models make it suitable for real-world scenarios where accurate pose estimation and adaptability to changing environments are crucial for successful robot autonomy.

5.3 ROS2 Controller

ROS2 Controller, an integral component of the Robot Operating System 2 (ROS2), represents a significant advancement in the field of robot control. Building upon the successes of its predecessor, ROS2 Controller provides an enhanced and versatile framework for implementing and managing controllers in robotic systems. With a focus on modularity, flexibility, and integration with other ROS2 components, ROS2 Controller empowers researchers and academics to explore and develop advanced control strategies, enabling precise and efficient manipulation of robots across a wide range of applications.

1. Modular Control Architecture: ROS2 Controller adopts a modular control architecture that allows for the seamless integration and interchangeability of control algorithms, enabling researchers to experiment with various approaches for robot control. This modular design facilitates the implementation and evaluation of novel control techniques, promoting innovation and advancement in the academic community.

2. Extensive Control Interfaces: ROS2 Controller provides a rich set of control interfaces, enabling researchers to interact with and control individual joints, actuators, and effectors of the robot. This fine-grained control granularity allows for precise manipulation and dexterity, essential for conducting sophisticated tasks and experiments in areas such as robotic manipulation and grasping.

3. Control Abstraction and Reusability: ROS2 Controller abstracts the underlying hardware details, providing a standardized interface for interacting with different robot platforms. This abstraction layer promotes code reusability and portability across different robotic systems, facilitating comparative studies, benchmarking, and generalizability of control algorithms and strategies.

4. Integration with Sensor Feedback: ROS2 Controller seamlessly integrates with sensor feedback, enabling researchers to leverage sensor data, such as vision, proprioceptive, or force/torque measurements, for closed-loop control. This integration fosters the development of adaptive control approaches, enabling robots to respond and adapt to environmental changes and task requirements in real-time.

5. Real-time Control: ROS2 Controller incorporates real-time capabilities, enabling precise control and synchronization of robot actions in time-critical applications. This feature is particularly valuable in domains such as robot-assisted surgery, human-robot interaction, or autonomous vehicles, where timely and accurate control responses are vital.

6. Safety and Fault Tolerance: ROS2 Controller emphasizes safety and fault tolerance in robot control. It provides mechanisms to handle error conditions, recover from faults, and implement safety protocols to ensure the wellbeing of humans and the integrity of the robotic system. This aspect is critical for academic research involving human-robot collaboration, physical human-robot interaction, and safe robot operation.

7. Integration with ROS2 Ecosystem: ROS2 Controller seamlessly integrates with other ROS2 components, including perception, planning, and simulation modules. This tight integration enables researchers to develop comprehensive control systems that incorporate state estimation, path planning, and high-level decision-making, promoting interdisciplinary research and development in robotics.

In conclusion, ROS2 Controller represents a significant step forward in robot control for academic research. With its modular architecture, extensive control interfaces, integration with sensor feedback, and real-time capabilities, ROS2 Controller enables researchers to explore innovative control strategies and algorithms in various application domains. By leveraging the power of the ROS2 ecosystem and open collaboration, ROS2 Controller fosters academic advancements, promoting interdisciplinary research and pushing the boundaries of robot control in pursuit of more intelligent, capable, and adaptive robotic systems.

ROS2 Controller subsystem is responsible for managing and executing control commands to actuate the robot's joints or effectors. While ROS2 Controller itself is a core component, there are several subsystems or components that work together to enable its functionality. These subsystems include:

1. Controller Manager: The Controller Manager is responsible for the overall management and coordination of controllers within the ROS2 system. It handles the loading, unloading, and switching of controllers based on commands received from higher-level modules or user inputs.
2. Hardware Interface: The Hardware Interface acts as a bridge between the ROS2 Controller and the robot's hardware. It provides a layer of abstraction that allows the controller to communicate with the low-level hardware, such as actuators, motors, or sensors, in a standardized manner. The Hardware Interface translates the control commands from the Controller into appropriate low-level commands for the robot hardware.
3. Joint State Controller: The Joint State Controller is responsible for reading and publishing the current state of the robot's joints. It receives joint position, velocity, and effort feedback from the robot's sensors or encoders and publishes this information for use by other modules, such as perception or planning components.
4. Joint Trajectory Controller: The Joint Trajectory Controller is a type of controller that enables the execution of trajectory-based control commands. It receives desired joint trajectories as inputs and generates control signals to drive the robot's joints along those trajectories. The Joint Trajectory Controller handles the interpolation and execution of the desired joint trajectories, allowing smooth and precise motion control.
5. Cartesian Controller: The Cartesian Controller enables control of the robot's end-effector or manipulator in Cartesian space. It takes high-level commands, such as desired positions or orientations in 3D space, and generates the necessary joint commands to achieve the desired end-effector motion. Cartesian Controllers are often used in tasks such as robot manipulation, pick-and-place operations, or object tracking.
6. Force/Torque Controller: The Force/Torque Controller enables control of contact forces and torques exerted by the robot on the environment or objects it interacts with. This controller is commonly used in applications such as force-controlled grasping, compliant motion, or physical human-robot interaction. It allows the robot to exert precise forces and torques while maintaining contact stability.
7. Adaptive Control Subsystems: ROS2 Controller can also integrate with adaptive control subsystems, enabling the development and implementation of advanced control techniques that adapt to changing environmental conditions, uncertainties, or system dynamics. Adaptive

control subsystems provide mechanisms for adjusting control parameters or strategies based on real-time feedback and adaptation algorithms.

These subsystems work together within the ROS2 ecosystem to enable effective control of robots. By integrating these components, ROS2 facilitates the development and deployment of versatile control strategies that can be tailored to various robot platforms, applications, and research objectives.

5.4 Navigation

Navigation is the process of guiding and controlling the movement of an entity, such as a robot, vehicle, or individual, to reach a desired destination while avoiding obstacles and maintaining a safe and efficient trajectory. It involves various key aspects:

Perception: Navigation relies on perception systems that utilize sensors like cameras, lidar, radar, or sonar to sense and understand the surrounding environment. These sensors provide data on obstacles, landmarks, or other relevant features.

Mapping: Mapping involves creating a representation of the environment. Robots build maps using sensor data, capturing the spatial layout of obstacles, landmarks, and other pertinent information. Maps aid in path planning, obstacle avoidance, and localization.

Localization: Localization determines the robot's position and orientation (pose) within the known map or environment. By fusing sensor data and map information, localization algorithms estimate the robot's pose, enabling it to determine its location relative to the goal and plan appropriate paths.

Path Planning: Path planning generates collision-free paths from the robot's current pose to the desired goal. Algorithms, such as Dijkstra's algorithm, A* search, or rapidly-exploring random trees (RRT), determine optimal or feasible paths considering environmental constraints, robot dynamics, and other factors.

Obstacle Avoidance: Obstacle avoidance ensures safe navigation by detecting and avoiding obstacles in the robot's path. By utilizing sensor data, robots detect and react to obstacles to circumvent potential collisions.

Motion Control: Motion control executes the desired trajectory or path generated during planning. It involves controlling the robot's actuators, such as motors or wheels, to achieve precise and smooth motion while accounting for dynamics, stability, and kinematic constraints.

Human-Robot Interaction: Navigation also encompasses interaction with humans. This includes interpreting human instructions or commands, understanding and following social conventions, or coordinating movements with human partners in shared spaces.

Autonomous Decision-Making: Advanced navigation systems incorporate decision-making algorithms to handle complex scenarios. This may involve dynamically replanning paths in response to changing conditions, adapting to unforeseen obstacles, or prioritizing goals based on predefined criteria.

Navigation plays a crucial role in applications such as autonomous vehicles, mobile robots, unmanned aerial vehicles (UAVs), and virtual characters in gaming and virtual reality. Its objective is to enable safe, efficient, and goal-directed movement, contributing to the autonomy and effectiveness of intelligent systems.

5.5 Nav2 stack

The Nav2 stack in ROS2 is a comprehensive software package that provides a full navigation system for autonomous mobile robots. It builds upon the successes of its predecessor, the ROS Navigation Stack, and is specifically designed for the ROS2 ecosystem. The Nav2 stack encompasses a collection of nodes, libraries, and tools that enable robots to autonomously navigate in complex environments while avoiding obstacles and reaching their desired goals.

Key components and features of the Nav2 stack include:

1. Navigation Planning: The Nav2 stack offers advanced path planning algorithms, such as Dijkstra's algorithm or A* search, to generate collision-free paths for the robot. It can handle various map representations, including occupancy grids or point clouds, and supports global and local planning strategies.
2. Obstacle Avoidance: The Nav2 stack incorporates obstacle avoidance techniques to ensure safe navigation in dynamic environments. It uses sensor data, such as laser scans or depth images, to detect obstacles and adapt the robot's trajectory to avoid collisions.
3. Localization: Nav2 provides localization capabilities to determine the robot's pose (position and orientation) within the map. It supports different localization techniques, including odometry, visual odometry, or simultaneous localization and mapping (SLAM), enabling accurate robot position estimation.
4. Sensor Fusion: The Nav2 stack incorporates sensor fusion techniques to fuse data from various sensors, such as lidar, cameras, or IMUs, for improved perception and navigation performance. Sensor fusion enhances the accuracy of obstacle detection, mapping, and localization.
5. Map Representation: Nav2 supports various map representations, including occupancy grids or point clouds, to represent the robot's environment. It enables the creation and updating of maps in real-time,

allowing the robot to navigate within both static and dynamic environments.

6. Path Execution: The Nav2 stack includes path execution components responsible for controlling the robot's motion to follow the planned path. It provides interfaces to actuate the robot's actuators, such as wheels or motors, to achieve precise and smooth motion along the desired trajectory.
7. Behavior Trees: Nav2 incorporates behavior trees as a high-level control mechanism, enabling the definition and execution of complex robot behaviors. Behavior trees allow for the modular and hierarchical composition of robot actions, providing flexibility and scalability in designing autonomous robot behaviors.
8. Simulation and Visualization: The Nav2 stack includes simulation environments and visualization tools, allowing users to simulate robot navigation and visualize the robot's perception, planned paths, and executed trajectories. This facilitates the development, testing, and debugging of navigation algorithms and behaviors.
9. Configuration and Parameterization: Nav2 provides configurable parameters and settings to customize the navigation system according to the robot's capabilities, environment, and specific requirements. This flexibility allows users to fine-tune the system and optimize navigation performance.

The Nav2 stack serves as a powerful toolset for researchers, developers, and roboticists working on autonomous mobile robot navigation in ROS2. Its comprehensive set of features, modular design, and compatibility with the ROS2 ecosystem enable the development and deployment of robust navigation systems for a wide range of robotic applications, including service robots, autonomous vehicles, or unmanned aerial vehicles (UAVs).

5.5.1 Nav2 tools

- Load, serve, and store maps (Map Server)
- Localize the robot on the map (AMCL)
- Plan a path from A to B around obstacles (Nav2 Planner)
- Control the robot as it follows the path (Nav2 Controller)
- Smooth path plans to be more continuous and feasible (Nav2 Smoother)
- Convert sensor data into a costmap representation of the world (Nav2 Costmap 2D)
- Build complicated robot behaviors using behavior trees (Nav2 Behavior Trees and BT Navigator)
- Compute recovery behaviors in case of failure (Nav2 Recoveries)
- Follow sequential waypoints (Nav2 Waypoint Follower)

- Manage the lifecycle and watchdog for the servers (Nav2 Lifecycle Manager)
- Plugins to enable your own custom algorithms and behaviors (Nav2 Core)

5.6 Multi agent

A multi-agent system refers to a framework where multiple autonomous agents or robots work together in a coordinated manner to achieve common goals or perform complex tasks. These agents can be physical robots or virtual entities operating in a shared environment. The key characteristics of a multi-agent system include autonomy, communication, coordination, and collaboration among the agents.

1. Autonomy: Each agent in a multi-agent system operates independently, making decisions based on local perceptions, available information, and its own objectives. Autonomy allows agents to adapt to changing conditions and make local decisions that contribute to the overall system behavior.
2. Communication: Agents in a multi-agent system communicate with each other to exchange information, coordinate their actions, and share knowledge. ROS2 provides a messaging infrastructure that allows agents to publish messages on specific topics and subscribe to topics of interest. This communication mechanism enables agents to share sensor data, state information, task assignments, and other relevant information necessary for coordination.
3. Coordination: Coordination mechanisms enable agents to work together and synchronize their actions to achieve collective goals. This involves negotiating tasks, synchronizing movements, sharing resources, and resolving conflicts. Agents can use coordination algorithms, distributed consensus protocols, or higher-level coordination frameworks provided by ROS2 to facilitate cooperation and achieve coordination.
4. Collaboration: Collaboration in a multi-agent system involves agents actively cooperating and sharing responsibilities to accomplish complex tasks. Agents can divide a task into subtasks and distribute them among the team members. Collaborative behaviors can include information sharing, task allocation, task handoff, cooperative sensing, or joint decision-making.

Implementing multi-agent systems using namespaces in ROS2 provides a structured approach to organizing and managing multiple agents operating in the same environment. Namespaces allow agents to operate independently while sharing common resources and communication channels. steps guide on how to implement multi-agent systems using namespaces in ROS2:

1. Define Agent Namespaces: Start by assigning unique namespaces to each agent in the system. Namespaces can be defined based on the agents' identities, roles, or any other relevant criteria. For example, you can assign namespaces like **/agent1**, **/agent2**, **/robot1**, or **/robot2** to represent different agents or robots.
2. Launch Agent Nodes: Create launch files or launch scripts to launch the nodes specific to each agent within their respective namespaces. Specify the namespace for each node to ensure that they operate within the correct scope. This can be done by setting the **namespace** attribute when launching the nodes.
3. Publish and Subscribe to Namespaced Topics: To facilitate communication between agents, use namespaced topics. Namespaced topics prevent conflicts between agents operating in the same environment. When publishing a message, use a topic name that includes the agent's namespace as a prefix. For example, if Agent 1 wants to publish a message, it can publish to a topic like **/agent1/topic_name**. If Agent 2 wants to subscribe to that message, it can subscribe to the same topic using **/agent1/topic_name**. This ensures that messages are routed correctly within the multi-agent system.
4. Use Namespaced Services: Agents may need to provide or consume services within their namespaces. Define services within the agent's namespace and ensure that the service clients use the correct namespaced service name when making service requests. For example, if Agent 1 provides a service named **/agent1/service_name**, other agents can access this service by specifying the complete namespaced service name.
5. Namespace Parameter Server: The parameter server in ROS2 allows storing and retrieving parameters dynamically. Utilize namespaces to organize parameters specific to each agent. For example, Agent 1's parameters can be stored under the **/agent1/** namespace, and Agent 2's parameters can be stored under the **/agent2/** namespace. Access parameters using the appropriate namespace to ensure correct parameter retrieval and update.
6. Visualization and Debugging: Utilize visualization tools such as RViz2 to visualize the state, sensor data, and trajectories of each agent within their respective namespaces. Configure RViz2 to display the information from

the correct namespace for each agent, allowing for visualization and debugging of individual agents as well as the overall multi-agent system.

7. Coordination and Communication: Implement coordination and communication mechanisms within and between agents using namespaced topics, services, and parameter server access. Develop message formats and protocols for inter-agent communication. Agents can publish messages to namespaced topics to communicate information, exchange commands, or share updates. They can also utilize namespaced services to request or provide specific functionalities. Furthermore, agents can access namespaced parameters to share information or modify shared configurations.

5.7 Solution

For our mobile robot we need to use these technologies mentioned earlier where the robot is able to execute its task which is carrying a package from the convey to its intended place

To be able to do so, firstly we need the mobile robot to have a map of the environment it’s working in for this mapping and specifically slam tool-box package is used.

5.7.1 Mapping

For mapping we need the robot to be able to move and see the environment as well, here we are using Kinect camera to make the robot see.

Kinect camera:

Also known as an RGBD camera, is a type of depth-sensing camera that combines both RGB (Red, Green, Blue) color imaging and depth sensing capabilities in a single device. It captures both color information and depth measurements, providing a richer representation of the environment compared to traditional RGB cameras.

The term "RGBD" stands for Red, Green, Blue, and Depth, highlighting the two essential components of the camera's functionality:

1. RGB Imaging: The RGB component of the Kinect camera captures color information, similar to a standard color camera. It consists of three-color channels (red, green, and blue), enabling the camera to capture the visual appearance of the scene and objects in their true colors.
2. Depth Sensing: The depth component of the Kinect camera captures the distance of objects from the camera, providing depth measurements for each pixel in the image. It achieves this by emitting infrared light or using structured light patterns and then calculating the distance based on the time-of-flight or the deformation of the light patterns.

Odometry data:

Also known as wheel odometry or encoder data, refers to the measurements and estimates of the robot's movement and position based on the rotation of its wheels or other motion sensors. Odometry is commonly used in robotics for estimating the robot's relative displacement and tracking its position over time.

Key aspects of odometry data include:

1. Wheel Encoders: Odometry data is typically obtained from wheel encoders, which are sensors attached to the robot's wheels. The encoders measure the wheel rotation and provide information about the distance traveled, wheel velocities, and direction of motion.
2. Distance Traveled: Odometry data includes measurements of the distance traveled by the robot, typically expressed in meters or another unit of length. It can provide estimates of the robot's translational movement along the x, y, and z axes.
3. Wheel Velocities: Odometry data often includes the angular velocities of the robot's wheels, representing the rotational speed of each wheel. These velocities are used to calculate the robot's linear and angular velocities in its local coordinate frame.
4. Pose Estimation: By integrating the odometry data over time, it is possible to estimate the robot's pose, which includes its position (x, y, z) and orientation (roll, pitch, yaw) in the world coordinate frame or a reference frame.

Odometry data provides valuable information about a robot's movement and serves as a crucial component in navigation, path planning, and control algorithms. While it offers estimates of the robot's motion.

By combining RGB and depth information and odometry data, we can now perform mapping with our mobile robot.

This can be done by sending the RGBD data and Odom data to slam tool-box, but before this we need to pass the RGBD data to node in ROS2 called depth to scan which takes the depth data and converts it to scan data to pass it to the slam tool-box package to be able to construct a map, slam toolbox also need odometry data to construct a map

After constructing the map, we need to save this map and here comes the first Nav2 tool Map Server which takes the constructed map as input and output the map in YAML format which can be used further in localization and navigation tasks.

5.7.2 Localization

After mapping the environment where the robot is operating the robot needs to know where its exactly placed in the environment, here we use localizing techniques to localize the robot, we are using AMCL and there is how:

1. We load the map that we created earlier in the YAML format that is supported by AMCL.
2. Configure AMCL Parameters: Configure the AMCL parameters to suit your robot and environment. These parameters are typically defined in a YAML or XML file. The configuration includes setting the number of particles, selecting sensor models (e.g., laser, camera), specifying motion models, and setting up any relevant covariance matrices or thresholds.
3. Launch AMCL Node: Launch the AMCL node using the **ros2 launch** command, providing the necessary configuration file. This will start the AMCL node and load the specified map and parameters. Ensure that the robot's sensor data is being published and available for AMCL to subscribe to.
4. Publish Sensor Data: Publish sensor data to the appropriate topics for AMCL to consume. This means laser scan data that is being published from the depth to laser node, ensure that the sensor data is correctly calibrated and synchronized with the robot's odometry data.
5. Monitor Localization: Monitor the AMCL node's output to observe the localization performance. This includes visualizing the estimated robot pose, analyzing the particle distribution, and checking the overall localization accuracy. ROS2 provides visualization tools like RViz2 for visualizing robot states and debugging the localization process.
6. Refine Parameters: Fine-tune the AMCL parameters as needed to improve localization performance. Adjust parameters related to sensor noise, motion model, or resampling to achieve more accurate and robust localization results. Iteratively modify and evaluate the parameters until satisfactory performance is achieved.

5.7.3 Navigation

We have performed mapping and localizing so we are now able to perform navigation where the robot actually executes the task, there are some nav2 nodes we need to use to perform navigation:

- **controller server**

The controller server is responsible for executing trajectory control commands, which specify the desired robot motion, such as velocity and acceleration, to achieve the planned path.

- **smoother server**

The smoother server implements a trajectory smoother to generate smoother and more natural robot motions. It takes the output trajectories from the controller server and applies smoothing algorithms to reduce jerk and improve motion quality.

- **planner server**

The planner server generates a collision-free path from the robot's current position to a specified goal location. It uses algorithms like A* (A-star) or Dijkstra's algorithm to search for an optimal or near-optimal path based on the environment's map and other constraints.

- **behavior server**

The behavior server handles higher-level behaviors and decision-making for the robot's navigation. It manages the execution of complex navigation behaviors, such as obstacle avoidance, dynamic replanning, or task-specific actions.

- **bt navigator**

The bt navigator stands for Behavior Tree Navigator. It utilizes behavior trees, a popular control architecture in robotics, to handle high-level task planning and decision-making. The bt navigator interprets and executes behavior tree structures, coordinating the actions of other navigation components to achieve the desired robot behavior.

- **waypoint follower**

The waypoint follower node is responsible for following a set of predefined waypoints or navigation goals. It takes the planned path from the planner server and issues control commands to the controller server to navigate through the waypoints sequentially.

- **velocity smoother**

The velocity smoother smooths the velocity commands sent to the robot's actuators to achieve smoother and more controlled motion. It helps reduce sudden changes in velocity and acceleration, resulting in improved path tracking and overall navigation performance.

After tuning and launching these nodes the robot should now take a path and start executing this to perform its task.

At this point we have successfully implemented one robot in our environment but the problem is that we need multiple robots or what we know call multiple agents to be able to optimize the work in the warehouse environment so we have to implement multi agent to this system (add other robots to the environment), working with a single robot is simple, it also has some limitations when compared to multi-agent systems in a warehouse environment. Here are some disadvantages of using a single robot in a warehouse compared to a multi-agent system:

1. Limited Scalability: A single robot has limitations in terms of its capacity and capabilities. As the warehouse size or workload increases, a single robot may struggle to handle the growing demands efficiently. In

contrast, a multi-agent system allows for easy scalability by adding more robots to distribute the workload and accomplish tasks in parallel.

2. Single Point of Failure: In a single robot system, the failure or malfunction of the robot can lead to a complete halt in operations until the issue is resolved. This single point of failure can result in significant downtime and impact the overall efficiency of warehouse operations. In a multi-agent system, if one robot encounters a problem, the other robots can continue working, minimizing the impact on productivity.
3. Limited Task Flexibility: A single robot is typically designed for specific tasks or workflows within the warehouse. It may have limitations in adapting to changing requirements or performing diverse tasks efficiently. Multi-agent systems, on the other hand, allow for task flexibility, as different agents can specialize in various tasks, enabling more versatile and adaptable operations.
4. Longer Response Times: With a single robot, response times can be slower due to limited resources and the need to complete one task before moving on to the next. In a multi-agent system, multiple robots can handle tasks simultaneously, reducing response times and enabling faster completion of warehouse operations.
5. Bottlenecks and Congestion: In a single robot system, the robot may become a bottleneck if the workload exceeds its capacity. It can result in congestion and delays, especially in areas with high traffic or complex task dependencies. Multi-agent systems can distribute tasks among multiple agents, helping to avoid bottlenecks and improve overall efficiency.
6. Lack of Redundancy: Single robot systems lack redundancy, which means there is no backup in case of robot failure or maintenance. If the robot requires maintenance or repairs, the entire warehouse operation may need to be halted until the robot is back in service. Multi-agent systems provide redundancy, where other agents can continue operations even if one agent is temporarily unavailable.
7. Limited Coverage and Range: Single robots may have limitations in terms of coverage and range, restricting their ability to handle large warehouses efficiently. Multi-agent systems can cover larger areas and handle more extensive operations by distributing tasks among multiple agents and leveraging their combined capabilities.

5.7.4 Multiagent implementation

Having our mobile robot being able to navigate in our environment and execute tasks we have to support our system with other agent, this can be achieved by the following:

1. Give our robot a namespace as agent_1 to be able to distinguish between different robots.
2. Launch the mapping node for only one robot so we don’t get contrasting maps.
3. Launch AMCL node for each agent to be able to localize itself inside the environment.
4. Implement a tasking algorithm to take the task to be executed as an input and outputs the most efficient agent to execute the task.
5. Launch navigation node for the agent that is chosen to perform the task.

In conclusion, implementing multi-agent systems in a warehouse environment using ROS2 provides several advantages over using a single robot. By leveraging mapping, localization, and navigation capabilities, combined with the power of multiple agents, the efficiency, scalability, and flexibility of warehouse operations can be significantly improved.

The solution involves utilizing technologies such as SLAM for mapping the environment, using RGBD cameras like the Kinect camera to capture visual and depth information, and odometry data for estimating the robot's movement and position. With the map constructed, localization can be achieved using AMCL, which enables the robot to determine its precise location within the environment.

For navigation, a set of ROS2 nodes is employed, including the controller server, smoother server, planner server, behavior server, bt navigator, waypoint follower, and velocity smoother. These nodes collaborate to plan collision-free paths, control trajectory execution, and handle higher-level behaviors and decision-making.

To implement multi-agent systems, each robot is assigned a unique namespace to ensure independent operation while sharing common resources. By launching specific nodes within their respective namespaces, agents can communicate through namespaced topics and services, enabling coordinated actions and information exchange. The use of namespaces also facilitates scalability, fault tolerance, and redundancy by adding more agents to distribute workload and minimize single points of failure.

The implementation of multi-agent systems in a warehouse environment overcomes limitations associated with single robot systems, such as limited scalability, single points of failure, and task inflexibility. Multi-agent systems offer improved response times, reduced congestion, and increased coverage, enabling more efficient and robust warehouse operations.

By implementing multi-agent concepts, such as task allocation algorithms, each agent can be assigned specific tasks based on efficiency criteria. This optimizes task execution and allows the system to dynamically adapt to changing demands.

Chapter 6
Embedded systems

The project stack is divided into two main components. The first component is an automated conveyor system that efficiently delivers packages within the warehouse. This system utilizes advanced technologies to transport items swiftly and accurately, reducing the time required for manual handling and improving overall productivity.

The second component of the project stack is a mobile robot designed to facilitate the movement of packages from the automated conveyor to their designated storage locations. This robot is equipped with intelligent navigation capabilities, enabling it to navigate through the warehouse autonomously while avoiding obstacles. Its primary function is to transport packages from the conveyor system to the precise storage location where they should be placed.

The mobile robot, a physical implementation of the robotic system, comprises two main parts. The first part utilizes the STM32F4 Black Pill as its foundation. It is integrated with various sensors and motor drivers, enabling precise control and data acquisition for the robot's operation.

The second part of the embedded system features the Raspberry Pi 4. This component is responsible for handling more complex tasks within the robot. Specifically, it is connected to a Kinect device equipped with a depth camera. The Kinect camera captures the environment in which the robot operates, enabling it to construct a detailed map. The Raspberry Pi 4 also serves as the platform for running the ROS 2 server, facilitating seamless communication with other ROS 2 packages.

The STM32F4 Black Pill serves as the backbone of the robot's hardware, providing essential interfaces and control capabilities. Through its connectivity with sensors and motor drivers, it enables the robot to gather data from the environment and execute precise movements. On the other hand, the Raspberry Pi 4, in conjunction with the Kinect and ROS 2 server, enhances the robot's perception and decision-making abilities.

By combining these two components, the mobile robot achieves a robust and versatile system for navigating its surroundings. The integration of sensor data from the STM32F4 and the Kinect's depth camera, processed by the Raspberry Pi 4, allows the robot to create an accurate map of its environment. This information serves as a foundation for executing tasks, avoiding obstacles, and efficiently moving within the designated space.

Furthermore, the ROS 2 server running on the Raspberry Pi 4 enables seamless communication and integration with other ROS 2 packages. This facilitates the exchange of information and coordination with external systems, expanding the capabilities and adaptability of the mobile robot.

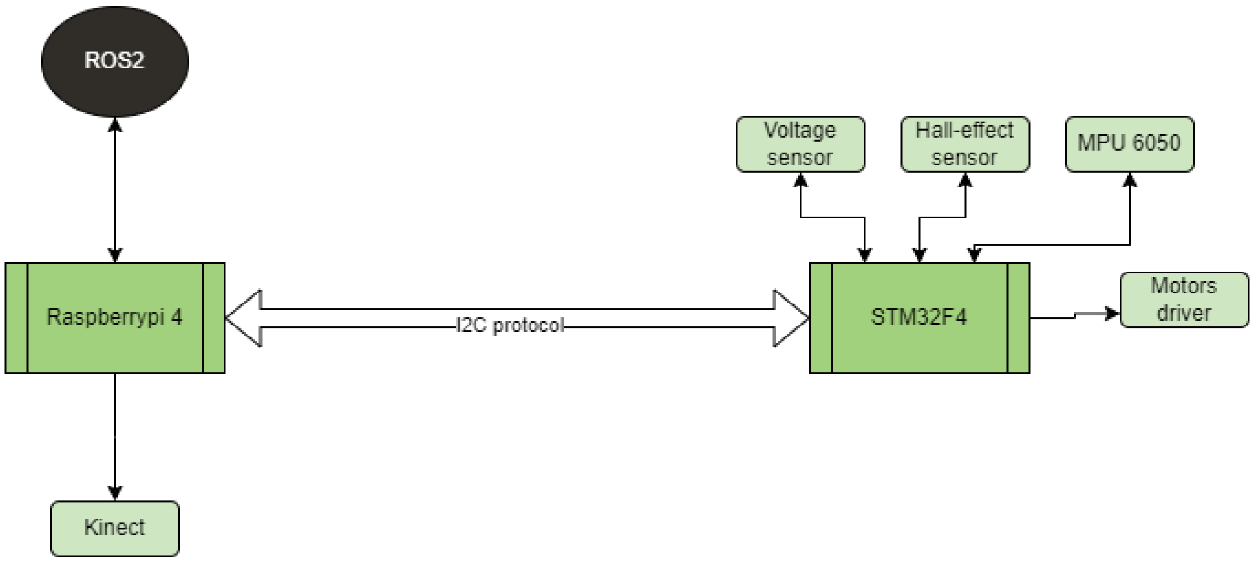


Figure 12 embedded systems block diagram

6.1 Layerd Architecture

Layered architecture is a software design pattern that divides the software into layers, each of which provides a specific set of services. This pattern helps to improve the **modularity**, **reusability**, and **scalability** of the software.

In embedded systems, layered architecture is often used to divide the software into three layers:

MCAL (Microcontroller Abstraction Layer): The MCAL layer provides a high-level interface to the microcontroller's hardware. This layer abstracts away the details of the hardware, making it easier for the upper layers to interact with the hardware.

HAL (Hardware Abstraction Layer): The HAL layer builds on the MCAL layer and provides a more abstract interface to the hardware. This layer hides the details of the specific microcontroller that is being used, making the software portable to different microcontrollers.

APP (Application Layer): The APP layer is the top layer of the architecture. This layer contains the application-specific code.

6.1.1 MCAL Layer

The MCAL layer provides a high-level interface to the microcontroller's hardware. This layer abstracts away the details of the hardware, making it easier for the upper layers to interact with the hardware.

The MCAL layer typically includes drivers for the microcontroller's peripherals, such as the UART, SPI, and I2C interfaces. It also includes functions for managing the microcontroller's memory, timers, and interrupts.

The MCAL layer is typically written in assembly language or C. This is because the MCAL layer needs to have direct access to the hardware, and assembly language and C are the only languages that can provide this level of access.

6.1.2 HAL Layer

The HAL layer builds on the MCAL layer and provides a more abstract interface to the hardware. This layer hides the details of the specific microcontroller that is being used, making the software portable to different microcontrollers.

The HAL layer typically includes functions for initializing the microcontroller's peripherals, configuring the peripherals, and reading and writing data to the peripherals.

The HAL layer is typically written in C . This is because the HAL layer needs to be portable to different microcontrollers, and C the most portable programming languages.

6.1.3 APP Layer

The APP layer is the top layer of the architecture. This layer contains the application-specific code.

The APP layer typically includes functions for implementing the application's features, such as reading sensor data, controlling actuators and displaying data on a display.

6.1.4 Advantages of Layered Architecture

The layered architecture pattern has several advantages in embedded systems:

Modularity: The software is divided into independent layers, which makes it easier to maintain and update the software.

Reusability: The layers can be reused in different applications, which saves time and effort.

Scalability: The architecture can be scaled up or down to meet the needs of different applications.

Portability: The software can be ported to different microcontrollers, as long as the HAL layer is updated to support the new microcontroller.

6.1.5 Project Layered Architecture

In mobile robot the needed modules in MCAL are

- RCC
- GPIO
- Timers
- SYSTICK
- ADC
- EXTI
- NVIC
- I2C

For HAL Layer the modules are

- DC_MOTOR
- MPU
- VOLTAGE_SENSOR
- HALL_EFFECT

For APP Layer there is only one module is the Robot control.

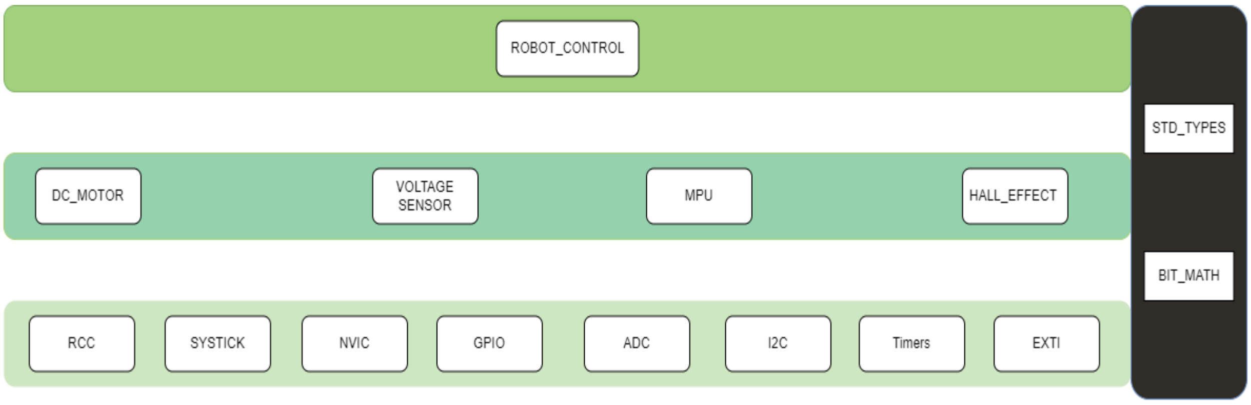


Figure 13 Layered architecture

6.2 Microcontroller

The STM32F4 Black Pill is a powerful microcontroller board that offers a range of features and capabilities for various embedded system applications. This compact board is based on the STM32F4 series microcontroller, which is part of STMicroelectronics' STM32 family. With its rich set of peripherals, processing power, and extensive community support, the STM32F4 Black Pill has gained popularity among developers and hobbyists alike.

6.2.1 Overview of STM32F4 Black Pill:

The STM32F4 Black Pill board combines the STM32F4 microcontroller with a convenient form factor, making it suitable for prototyping, development, and integration into embedded projects. The microcontroller itself is built around the ARM Cortex-M4 processor, which provides high performance and efficiency for a wide range of applications.

6.2.2 Key Features and Specifications:

The STM32F4 Black Pill board offers a host of features that make it versatile and suitable for various projects. Some of its notable features include:

1. **Microcontroller**: The board is powered by the STM32F407VET6 microcontroller, which operates at a clock frequency of up to 168 MHz. It integrates a rich set of peripherals, including GPIO pins, timers, ADCs, DACs, UART, SPI, I2C, USB, and more.
2. **Memory**: The board provides ample memory resources, including 512 KB of Flash memory for program storage and 192 KB of RAM for data storage and processing. This allows for the implementation of complex algorithms and applications.
3. **Connectivity**: The STM32F4 Black Pill supports various communication interfaces, such as USB OTG (On-The-Go), UART, SPI, and I2C. These

interfaces enable seamless connectivity with other devices, sensors, and communication networks.

4. **Expansion Options**: The board features a standard 20-pin JTAG/SWD header for debugging and programming, as well as additional GPIO pins for connecting external modules, sensors, and expansion boards.

Benefits and Applications:

The STM32F4 Black Pill offers several advantages that make it a popular choice for embedded system development:

Performance: The Cortex-M4 processor, coupled with its high clock frequency and advanced instruction set, delivers exceptional processing power, enabling real-time and computationally intensive applications.

Peripheral Integration: The microcontroller's rich set of peripherals allows for seamless integration with various sensors, actuators, and communication interfaces, expanding the range of possible applications.

Extensive Software and Community Support: The STM32F4 series is well-supported by a vast community of developers, providing access to a wide range of software libraries, examples, and resources. This support ecosystem simplifies development and accelerates the learning curve for users.

Versatility: The STM32F4 Black Pill is suitable for a wide range of applications, including robotics, home automation, industrial control systems, Internet of Things (IoT) devices, and more. Its flexibility and scalability make it an ideal choice for both prototyping and production-level projects.

As a result of the aforementioned advantages and capabilities, the STM32F4 Black Pill has been selected to be connected with the sensors needed for the mobile robot and motors driver for robot motion.

6.3 Sensors

Sensors used in mobile robot are MPU6050, Voltage sensor, and Hall effect Sensor.

6.3.1 MPU6050:

The MPU6050 is a popular motion-tracking sensor module that combines a 3-axis gyroscope and a 3-axis accelerometer in a single integrated circuit. Developed by InvenSense (now a part of TDK), the MPU6050 offers accurate motion sensing capabilities and is widely used in various applications such as robotics, drones, gaming, wearable devices, and motion-based user interfaces.

Key Features:

The MPU6050 motion tracking sensor module offers several key features that make it a popular choice among developers:

Gyroscope and Accelerometer Integration: The MPU6050 combines a 3-axis gyroscope and a 3-axis accelerometer on a single chip, allowing simultaneous measurement of rotational and linear motion. This integration simplifies the design process and reduces the board space required.

Digital Motion Processing: The sensor module includes a built-in Digital Motion Processor (DMP) that offloads complex motion processing tasks from the host microcontroller, reducing the computational burden and providing calibrated motion data ready for use.

Low Power Consumption: The MPU6050 is designed to operate at low power, making it suitable for battery-powered applications. It includes power-saving features such as sleep mode and an on-chip temperature sensor for thermal management.

Wide Sensing Range: The gyroscope offers programmable full-scale ranges from ±250 degrees per second to ±2000 degrees per second, while the accelerometer provides programmable full-scale ranges from ±2g to ±16g. This wide sensing range allows the sensor to adapt to different motion dynamics.

Working Principles:

The MPU6050 utilizes a combination of microelectromechanical systems (MEMS) technology and sensor fusion algorithms to measure and interpret motion accurately. The gyroscope measures the rotational rate around each of the three axes, while the accelerometer measures linear acceleration in the same axes. These measurements are processed by the built-in DMP, which combines the data from both sensors to provide fused motion information.

The sensor fusion algorithms in the DMP utilize sensor data fusion techniques such as Kalman filtering or complementary filtering to combine the gyroscope and accelerometer measurements effectively. This fusion process compensates for the limitations of individual sensors and provides accurate motion tracking data, including orientation, angular velocity, and linear acceleration.

6.3.2 Controling MPU6050 via STM32F4

Hardware Setup:

- Connect the VCC and GND pins of the MPU6050 to the 3.3V and GND pins of the STM32F4 microcontroller, respectively.
- Connect the SDA and SCL pins of the MPU6050 to the corresponding I2C pins of the STM32F4 microcontroller.
- Configure I2C Pins as pull up opendrain

Configure the I2C Peripheral:

- Enable the I2C peripheral by setting the appropriate bits using RCC.
- Configure the I2C pins in the GPIO mode, ensuring the correct alternate function is selected.

Initialize the I2C peripheral with the desired parameters, such as the clock speed and addressing mode.

Initialize the MPU6050

- Write the desired configuration values to the MPU6050 registers using the I2C interface. This includes setting the sample rate, range, and other parameters based on your application requirements.
- check if the sensor is responding by reading the **“WHO_AM_I (0x75)”** Register. If the sensor responds with **0x68**, this means it’s available and good to go.
- Next we will wake the sensor up and in order to do that we will write to the **“PWR_MGMT_1 (0x6B)” Register**
- On writing (0x00) to the PWR_MGMT_1 Register, sensor wakes up and the Clock sets up to 8 MHz.
- Setting the Data output Rate or Sample Rate. This can be done by writing into **“SMPLRT_DIV (0x19)” Register**. This register specifies the divider from the gyroscope output rate used to generate the Sample Rate for the MPU6050. As the formula says Sample Rate = Gyroscope Output Rate / (1 + SMPLRT_DIV). Where Gyroscope Output Rate is 8KHz, To get the sample rate of 1KHz, we need to use the SMPLRT_DIV as ‘7’.
- Now configure the Accelerometer and Gyroscope registers and to do so, we need to modify **“GYRO_CONFIG (0x1B)” and “ACCEL_CONFIG (0x1C)”Registers**.
- Writing (0x00) to both of these registers would set the Full scale range of ± 2g in ACCEL_CONFIG Register and a Full scale range of ± 250 °/s in GYRO_CONFIG Register along with Self-test disabled. This completes the initialization of the MPU6050.

Read Sensor Data

- We can read 1 BYTE from each Register separately or we can just read 6 BYTES all together starting from ACCEL_XOUT_H Register.
- The ACCEL_XOUT_H (0x3B) Register stores the higher Byte for the acceleration data along X-Axis and Lower Byte is stored in ACCEL_XOUT_L Register. So we need to combine these 2 BYTES into a 16 bit integer value. As the following.
  ACCEL_X = (ACCEL_XOUT_H <<8 | ACCEL_XOUT_L)
- Similarly we can do the same for the ACCEL_YOUT and ACCEL_ZOUT Registers. These values will still be the RAW values and we still need to convert them into proper ‘g’ format. for the Full-Scale range of **± 2g**, the sensitivity is **16384 LSB/g**. So to get the ‘**g**‘ value, we need to divide the RAW from **16384**.
- Reading Gyro Data is similar to the Acceleration case. We will start reading 6 BYTES of data from the GYRO_XOUT_H Register, Combine

the 2 Bytes to get 16 bit integer RAW values. As we have selected the Full-Scale range of ± 250 °/s, for which the sensitivity is 131 LSB /°/s, we have to divide the 0RAW values by 131.0 to get the values in dps (°/s).

6.3.3 Voltage Sensor

Voltage sensor play a crucial role in the mobile robot by providing accurate measurements of voltage levels. It is essential components in determining the power level of each mobile robot to detect weather it can operate or it need to be charged. This helps in making the system full automated. It can measure from 0 to 25 volt. The mobile robot operating power is 12 volt so this sensor range is better to our application.

Working Principle

Voltage sensors operate based on the principle of converting electrical voltage into a measurable signal. The most common method involves using a voltage divider circuit to scale down the input voltage to a level suitable for measurement. This scaled-down voltage is then converted into an analog signal that can be processed by STM32f4 as all its gpio pins supports working as ADC channel with 12 bit channel resolution so it can provide us with accurate data about battery level.

6.3.4 Controling Voltage sensor using STM32f4

Hardware Setup:

- Connect source of power of the robot (battery) to the input of the sensor
- Connect the the analog pin to the GPIO pin which is configured as ADC channel and GND to GND of STM3F4

Software Configuration:

To control this sensor to be able to read battery level reading you need first to confiure the ADC in STM32 that can be done through initialization the ADC module with the specified channel as the input by configuring the ADC clock cycles, prescale value, sampling time, and enables the ADC.

After initialization to read the ADC data from the specified channel. It starts the conversion, waits for the conversion to complete, clears the end of conversion flag, and returns the converted data from the data register (DR). Hal layer contains separate module for voltage sensor that calls the function of the ADC and to map the data in data register to voltage level from 0 to 12 volt.

6.3.5 Hall effect sensor

A hall effect sensor is a type of sensor that detects the presence and magnitude of a magnetic field using the Hall effect. The Hall effect is the production of a voltage difference across a conductor when it is placed in a magnetic field.

Hall effect sensors are used in a wide variety of applications, including motor control, position sensing, and current sensing. In motor control, hall effect sensors are used to provide feedback on the rotor position. This feedback is used to control the motor's speed and direction of rotation.

Advantages of Hall Effect Sensors:

Hall effect sensors have a number of advantages over other types of sensors for motor control, including:

High accuracy: Hall effect sensors are very accurate, and they can provide accurate feedback on the rotor position even at high speeds.

Reliable: Hall effect sensors are very reliable, and they can withstand harsh environments.

Low cost: Hall effect sensors are relatively inexpensive, which makes them a cost-effective solution for motor control.

In mobile robot it is nessceary to ensure that the motor is moving in the needed dirction to its destination with the needed speed for the motors. So we used dc motors that has already established hall effect sensors. It has Hall effect sensors established at the end of the motor.

6.3.6 Controling Hall effect sensor using STM32f4

Hardware Setup:

- Connect the power of the sensor to the power pins in STM32F4 (GND & 3.3).
- Connect the output of the sensors to the GPIO Pins and configure them as input pull up

Software Configuration:

To calculate the speed of the motor we need to count the number of pulses generated by one sensor. Counting the number of pulses in one second. That can be done by enabling interrupt on the GPIO pin connected to the sensor rising edge , and in the ISR of the interrupt will update number of pulses through one second.

3By the end of this second Calculation of the RPM can be done by this formula

RPM = Number of Pulses * number of pulses per rotation

number of pulses per rotation is already given in the dc motor datasheet.

Determing the dirction of the rotation of the motor can be one by checkinh which sensor leading nd which one is laing by that the direction can be detected.

6.4 Motors

The mobile robot works with 2 DC motor of model **GM25-370-CE**

6.4.1 Features:

- Model: GM25-370-CE
- Nominal voltage: 12 VDC
- No load speed: 250 RPM
- Gears reduction ratio: 1:10
- Hall resolution: 224.4 PPR
- Output shaft: D-shaped, 6mm diameter
- Dimensions: 32mm (diameter) x 78.8mm (length)
- Weight: 20g

The GM25-370-CE is a small, high-torque DC gear motor with a D-shaped output shaft. It is suitable for our application for the movement of the mobile robot.

6.5 Motors Driver

The L298 is a popular integrated circuit (IC) commonly used as an H-bridge motor driver. It allows bidirectional control of DC motors and provides an easy way to control the speed and direction of motors in robotics and automation projects.

6.5.1 Features:

Basic Functionality: The L298 is a dual H-bridge IC, meaning it can control two DC motors independently. Each H-bridge consists of four transistors, allowing for bidirectional control of current flow to the motor. This configuration enables control over the motor's rotation direction and speed.

Voltage and Current Ratings: The L298 can handle a wide range of supply voltages, typically between 4.5V and 46V. The maximum operating current is around 2A per channel, but this can be increased by adding external transistors.

Pin Configuration: The L298 IC comes in a multiwatt15 package and consists of 15 pins. The important pins include two enable pins (ENA and ENB) to control the motor speed, four input pins (IN1, IN2, IN3, and IN4) for setting the motor direction, and four output pins (OUT1, OUT2, OUT3, and OUT4) to connect to the motor terminals.

Motor Control Modes: The L298 can operate in different control modes, including two-wire control and three-wire control. In two-wire control, the enable pins are connected together, and the motor speed is controlled by a single PWM (Pulse-Width Modulation) signal. In three-wire control, each motor channel has a separate enable pin, allowing independent speed control.

Built-in Protection: The L298 includes built-in protection features to prevent damage to the IC and connected components. It has built-in diodes (also known as flyback diodes or freewheeling diodes) to protect against voltage spikes generated by the motor. These diodes help to prevent damage to the IC when the motor is turned off.

Heat Dissipation: Since the L298 can handle significant currents, it can generate heat during operation. It is recommended to use a heat sink or other cooling methods to dissipate heat to ensure the IC operates within its temperature limits.

6.5.2 Controling L298 motor driver via STM32F4

Hardware Setup:

- Connect the two pins of each motor to one of the two output terminals
- Connect the input pins to two digital pins in STM32F4 to control the direction of each motor(IN1, IN2 for motor1 & IN3,IN4 for motor).
- Connect the two pwm channels to the enable pins EN1 & EN2.
- Connect the battery to the power of the hbridge.

Software Configuration:

PWM is used to control motors speed. In PWM, the signal is a square wave with a fixed period. The duty cycle represents the percentage of time the signal is in the "ON" state (high voltage) compared to the total period. By adjusting the duty cycle, the effective voltage or current delivered to the load can be controlled.

Pwm control is devided into 2 main functions the first that initialise the timer to work as pwm by passing the channel needed.

The second function that starts the pwm by loading the register by the value that output the needed duty cycle passed in function.

To provide full abstraction due to layered architecture the module DCMotor is used in initialising the PWM and configure the pins of the motors direction. It also has another function that is used to pass move the motor In one direction with specified dutycycle.

6.6 Raspberrypi 4

The Raspberry Pi 4 is a single-board computer developed by the Raspberry Pi Foundation. It was released in June 2019 as the successor to the Raspberry Pi 3 Model B+. The Raspberry Pi 4 offers improved performance, increased memory options, and enhanced multimedia capabilities compared to its predecessors. Here's some detailed information about the Raspberry Pi 4:

6.6.1 Specifications

Processor: Broadcom BCM2711 quad-core ARM Cortex-A72 (64-bit) CPU @ 1.5GHz

GPU: Broadcom VideoCore VI

Memory: 4GB

Storage: MicroSD card slot (supports SDXC cards up to 2TB)

Connectivity: Dual-band 802.11ac wireless LAN, Bluetooth 5.0, Gigabit Ethernet

USB Ports: Two USB 2.0 ports, two USB 3.0 ports

Video Outputs: Two micro HDMI ports (supporting up to 4Kp60)

Audio: 3.5mm audio jack, two micro HDMI ports for audio/video output

GPIO: 40-pin GPIO header

Power: USB-C connector for power input (5V/3A recommended)

Performance: The Raspberry Pi 4 offers a significant performance boost compared to its predecessors. The Cortex-A72 CPU cores are more powerful, providing faster execution of tasks. The increased memory options (up to 8GB) allow for better multitasking and the ability to run more memory-intensive applications.

6.6.2 Ubunto jammy 22.04 Operatin system

The Raspberry Pi 4 is compatible with various operating systems but for this application the most suitable operating system is **ubuntu jammy 22.04**.

Features:

Stability and Long-Term Support: LTS releases of Ubuntu, including those with version numbers ending in 04, are designed to provide stability and long-term support. They typically receive five years of support, including security updates and bug fixes. This makes them suitable for production environments and enterprise use.

Newer Software Versions: Ubuntu LTS releases tend to introduce updated versions of various software packages compared to the previous LTS release. This includes newer desktop environments, applications, and development tools. These updates bring improved features, performance enhancements, and bug fixes.

Enhanced Security: Ubuntu LTS releases prioritize security and offer regular security updates throughout their support period. This helps to ensure that your system remains protected against emerging threats and vulnerabilities.

Hardware Compatibility: With each new Ubuntu release, hardware support is expanded, enabling compatibility with a wider range of devices. The LTS releases usually incorporate newer kernel versions, which often include improved support for the latest hardware components.

Desktop Environment Improvements: Ubuntu typically includes a default desktop environment (such as GNOME) with each release. New LTS versions often come with updates to the desktop environment, offering new features, visual enhancements, and improved performance.

Developer-Friendly: Ubuntu is widely used by developers due to its extensive package repository, tools, and community support. LTS releases ensure a stable development environment, making it easier for developers to create and deploy their applications.

Community and Ecosystem: Ubuntu has a large and active community, which means there is a wealth of resources, tutorials, and support available. The Ubuntu ecosystem provides a rich set of software, including applications, libraries, and frameworks, making it easier to find and utilize tools for various tasks.

Supports ROS2: Ubuntu provides strong support for ROS2, with its compatibility, community, and development tool integration. It is a widely used and recommended choice for running ROS2 applications.

6.7 I2C Comunication protocol

I2C is a widely used synchronous serial communication protocol. It is a multi-master, multi-slave protocol designed for short-distance communication between integrated circuits on a PCB (Printed Circuit Board). It allows for efficient data transfer using only two wires: a data line (SDA) and a clock line (SCL).

6.7.1 Communication Basics:

Master-Slave Relationship: I2C supports a master-slave relationship, where the master initiates and controls the communication with one or more slave devices.

Addressing: Each slave device on the I2C bus is identified by a unique 7-bit or 10-bit address. The master sends the address of the slave it wants to communicate with.

Synchronous Communication: Communication in I2C is synchronous, meaning the clock signal generated by the master controls the timing of data transfer.

Bi-directional Data Line (SDA): The SDA line is used for both transmitting and receiving data between the master and the slave(s).

Clock Line (SCL): The SCL line carries the clock signal generated by the master to synchronize data transfer.

6.7.2 Data Transfer:

Start and Stop Conditions: Communication starts with a Start condition (SDA transitioning from high to low while SCL is high) and ends with a Stop condition (SDA transitioning from low to high while SCL is high).

Data Bits: Data is transferred in 8-bit chunks (bytes) with the most significant bit (MSB) sent first.

Acknowledgment (ACK/NACK): After receiving each byte of data, the receiver (either master or slave) sends an Acknowledgment (ACK) bit to indicate successful reception. A Non-Acknowledgment (NACK) is sent if there is an error or the receiver cannot accept more data.

Clock Stretching: Slaves can hold the SCL line low to slow down the communication, known as clock stretching. This allows slower devices to keep up with the transfer rate of the master.

6.7.3 I2C Modes:

Standard Mode: Supports data rates up to 100 kbps.

Fast Mode: Supports data rates up to 400 kbps.

6.7.4 Implementation of I2C In mobile robot

In mobile robot I2C is used in 2 different modules. The first in Controling MPU6050 that acts as slave for STM32F4. The second is detecated for communocation between raspberrypi 4 and STM32F4 for sending the data of the sensors as it is connected to STM32F4 to raspberrypi, and then raspberry pi sends back the motors speed and dirictions.

6.8 System State mashine

state machines offer a structured and efficient approach to modeling and implementing complex behavior in embedded systems. They enhance system understanding, modularity, event-driven behavior, fault tolerance, resource efficiency, scalability, testability, and debugging capabilities. Utilizing state machines in embedded systems promotes robustness, reliability, and maintainability throughout the system's lifecycle.

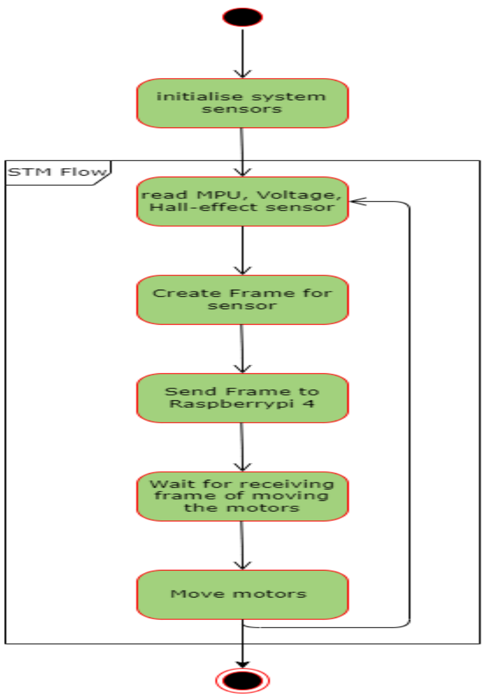

initialise system sensors
STM Flow
read MPU, Voltage, Hall-effect sensor
Create Frame for sensor
Send Frame to Raspberrypi 4
Wait for receiving frame of moving the motors
Move motors

Figure 15 STM State machine

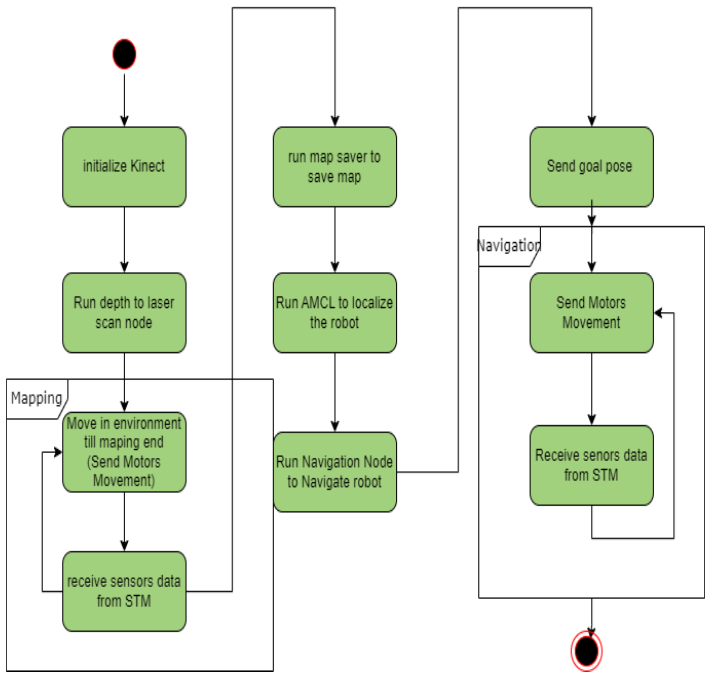

initialize Kinect
Run depth to laser scan node
Mapping
Move in environment till maping end (Send Motors Movement)
receive sensors data from STM
run map saver to save map
Run AMCL to localize the robot
Run Navigation Node to Navigate robot
Send goal pose
Navigation
Send Motors Movement
Receive senors data from STM

Figure 14 RapberryPi state machine

Chapter 7
Diagrams

7.1 Sequence diagram

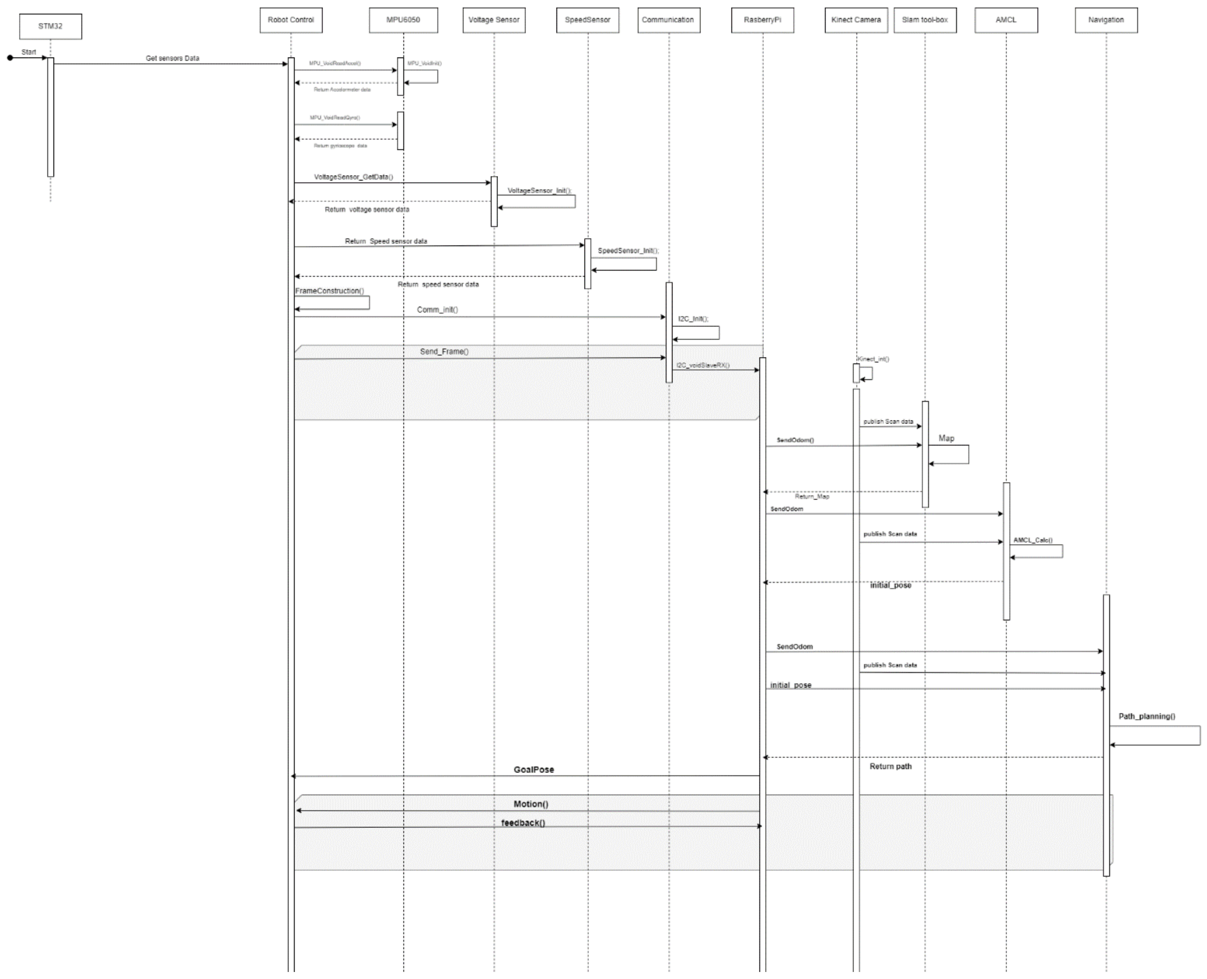


Figure 16 System Sequence diagram

7.2 Class diagram

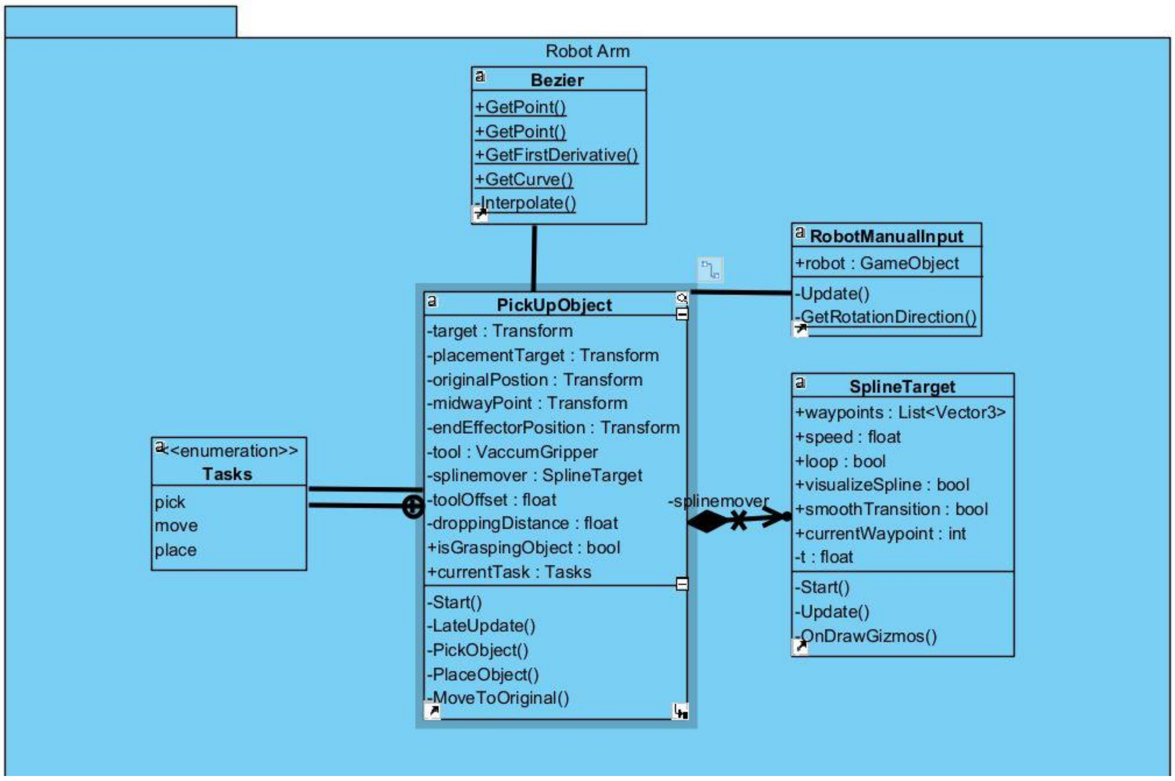


Figure 17 Class diagram 1

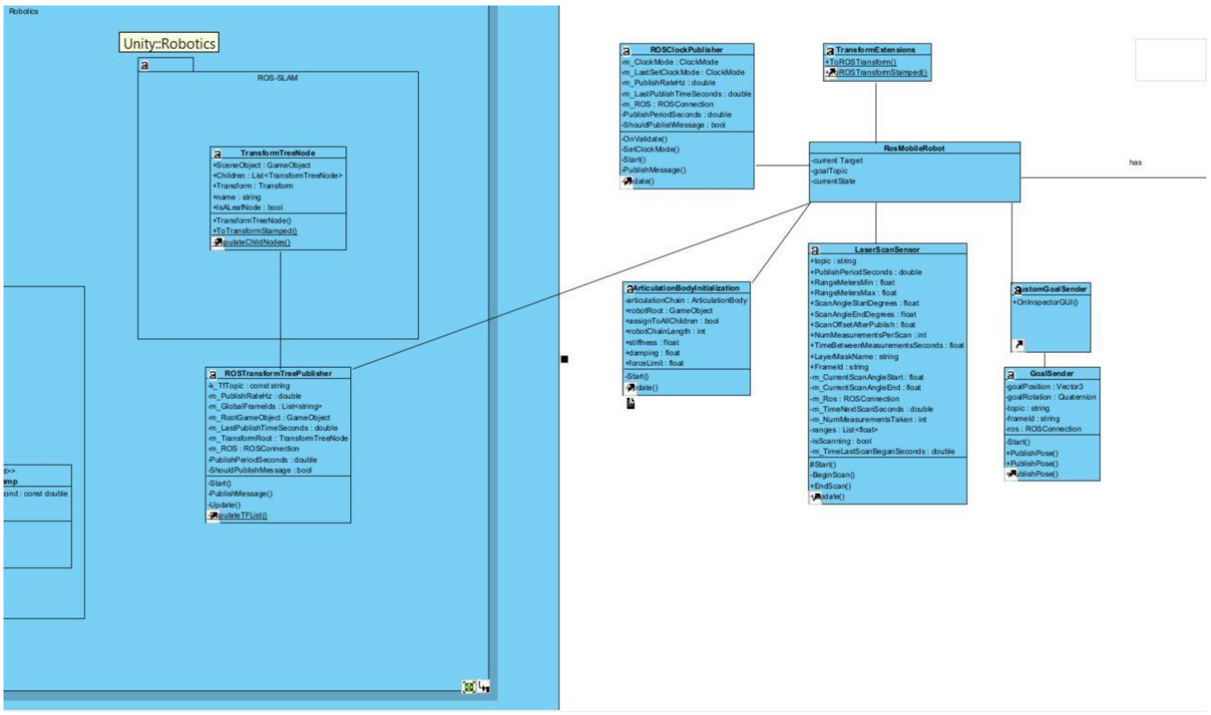


Figure 18 Class diagram

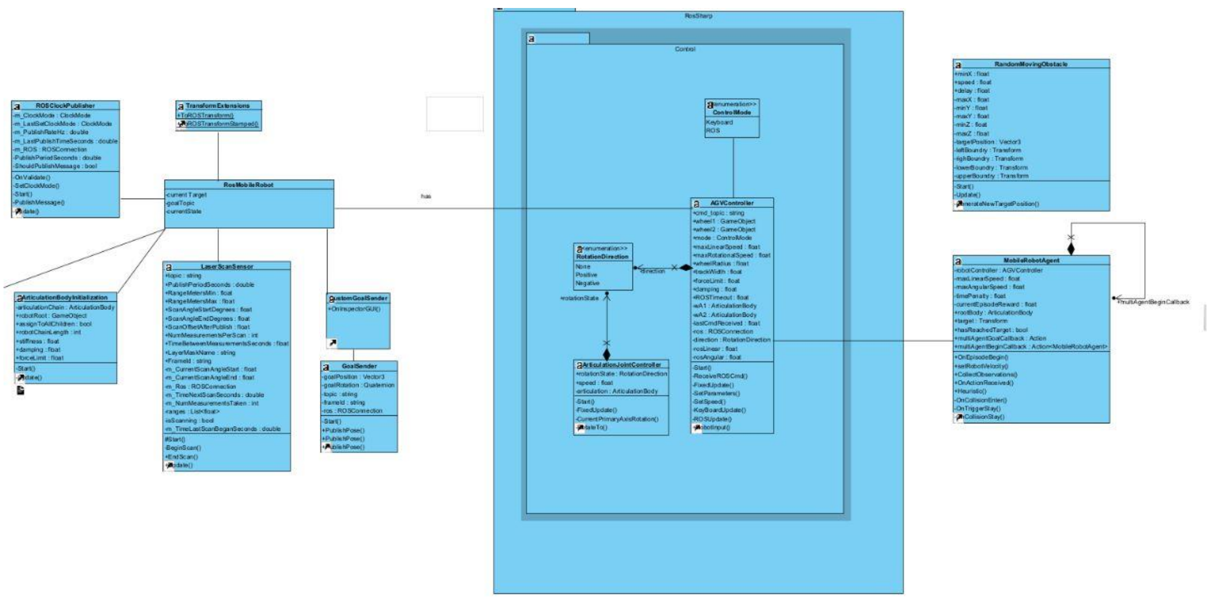


Figure 19 Class diagram3

Chapter 8
Testing

Testing is an essential part of software development and ensures that a system or software meets its intended functionality, quality, and performance requirements. Testing can be classified into different levels, including unit testing, integration testing, and system testing. Each level of testing serves a specific purpose in verifying the software's behavior and identifying potential defects or issues.

Unit Testing:

Unit testing focuses on testing individual units or components of software in isolation. The goal is to verify that each unit functions correctly as per its design and requirements. It involves testing small, independent code segments such as functions, methods, or classes. Unit testing typically employs testing frameworks and techniques to provide automated and repeatable tests. By isolating and testing units individually, developers can detect and fix defects early in the development process.

Integration Testing:

Integration testing verifies the interactions and interfaces between different units or components of software. It aims to identify defects that may arise when integrating individual units into a larger system. Integration testing ensures that the units work together seamlessly and that data flows correctly between them. It tests the integration points, communication protocols, and dependencies between units. The goal is to uncover defects caused by incompatible interfaces, incorrect data transformations, or improper interactions between components.

System Testing:

System testing is conducted on the complete software system as a whole. It validates the behavior of the entire system against its functional and non-functional requirements. System testing involves testing the system's functionalities, performance, reliability, security, and usability. It aims to ensure that the software system meets the desired business goals and performs as expected in the target environment. System testing can include various techniques such as functional testing, performance testing, usability testing, and security testing.

Unit testing, integration testing, and system testing complement each other and collectively contribute to the overall software quality. They help identify defects at different levels of granularity, from individual units to the entire system. By conducting these testing levels systematically, developers can increase the confidence in the software's correctness, reliability, and robustness before deploymen

8.1 Unit Testing

Table 2 Test case 1

8.1.1 Testing Embedded Systems Modules

Test Case ID	1. VoltageSensor_Init()		
Test Case Description	Testinng initialzation of the channel connected to the voltage sensor		
Pre-Requisite	Availability of the ADC module. Proper initialization and configuration of the microcontroller or development board.	**Post-Requisite**	The ADC module should be successfully initialized with the provided ADC channel.
Test Execution Steps:			

S.No	**Inputs**	**Expected Output**	**Test Result**
1	ADC_Channel = 0	Initialize ADC Channel 0	Passed
2	ADC_Channel = 1	Initialize ADC Channel 1	Passed
3	ADC_Channel = 8	Initialize ADC Channel 8	Passed

Table 3 Test case 2

Test Case ID	2. u8 VoltageSensor_GetData()		
Test Case Description	Testing data received fom voltage sensor		
Pre-Requisite	The ADC module or function ADC_ReadChannel() is properly configured and	**Post-Requisite**	The VoltageSensor_GetData() function correctly calculates the voltage output based on the ADC

	returns valid readings within the expected range of 0 to 1275.		reading and returns the expected voltage value.

Test Execution Steps:			
S.No	**Inputs**	**Expected Output**	**Test Result**
1	ADC reading = 0	5	Passed
2	ADC reading = 1275	1.96	Passed
3	ADC reading = 500	2	Passed

Table 4 Test case 3

Test Case ID	3. void SpeedSensor_Init();		
Test Case Description	Initialize the hall effect sensor		
Pre-Requisite	Availability of the speed sensor module. Proper hardware connections and configuration.	**Post-Requisite**	The speed sensor module should be successfully initialized.
Test Execution Steps:			

S.No	Inputs	Expected Output	Test Result
1	No input required	Speed sensor initialized	passed
2	No input required	Speed sensor already initialized	passed

Table 5 Test case 4

Test Case ID	4. SpeedSensor_GetSpeed()		
Test Case Description	Return the RPM depending on the number of pulses		
Pre-Requisite	• Availability of a speed sensor connected to the system. • Proper hardware connections and configuration for the speed sensor.	**Post-Requisite**	The speed sensor should return the RPM of the motor successfully
Test Execution Steps:			

S.No	Inputs	Expected Output	Test Result
1	No input required	Non-zero RPM value	Passed
2	No input required	zero RPM value	Passed

Table 6 Test case 5

Test Case ID	5. void MPU_VoidInit(void)
Test Case Description	Initialise the MPU sensor and check the connectivity

Pre-Requisite	Proper hardware connections and configuration for the MPU. Properly configured I2C bus and relevant I2C library or driver.	**Post-Requisite**	Sensor must be initialised successfully

Test Execution Steps:			
S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input required	MPU sensor initialized	Passed
2	No input required	MPU Sensor already initialized	Passed

Table 7 Test case 6

Test Case ID	6. void MPU_VoidReadAccel(void)		
Test Case Description	Get the accelerometer readings in the three axis		
Pre-Requisite	• Proper hardware connections and configuration for the MPU • Properly configured I2C bus and relevant I2C library or driver.	**Post-Requisite**	Sensor must be initialised successfully

Test Execution Steps:			
S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input required	Accelerometer data is read successfully	Passed

Table 8 Test case 7

Test Case ID	7. void MPU_VoidReadGyro(void)		
Test Case Description	Get the accelerometer readings in the three axis		
Pre-Requisite	• Proper hardware connections and configuration for the MPU • Properly configured I2C bus and relevant I2C library or driver.	**Post-Requisite**	Sensor must be initialised successfully
Test Execution Steps:			

S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input required	Accelerometer data is read successfully	Passed

Table 9 Test case 8

Test Case ID	8. void DCMotor_Init(MotorsNo Motor)		
Test Case Description	It must initialise the specified DC motor		
Pre-Requisite	• Availability of DC motors to be initialized. • Proper hardware connections	**Post-Requisite**	• Successful initialization of the specified DC motor.

	and configuration for the DC motors. • The microcontroller or system should support the necessary interfaces or protocols for motor control (PWM, GPIO).		• The motor driver circuit should be ready to control the motor, allowing for actions such as starting, stopping, and changing the motor speed or direction

Test Execution Steps:

S.No	Inputs	Expected Output	Test Result
1	Motor = Motor1	Initialization of Motor 1	Passed
2	Motor = Motor2	Initialization of Motor 2	Passed

Table 10 Test case 9

Test Case ID	9. void DCMotor_Move(Directions Direction,MotorsNo Motor, DUTYCYCLES dutycycle)		
Test Case Description	It must the move the specified DC motor with the given duty cycle		
Pre-Requisite	• Availability of DC motors to be initialized. • Proper hardware connections and configuration for the DC motors.	**Post-Requisite**	• Successful initialization of the specified DC motor. • The motor driver circuit should be ready to control the motor,

	• The microcontroller or system should support the necessary interfaces or protocols for motor control (PWM, GPIO).		allowing for actions such as starting, stopping, and changing the motor speed or direction

Test Execution Steps:

S.No	**Inputs**	**Expected Output**	**Test Result**
1	Direction = Backward	Move motor backward	Passed
2	Direction = Forward	Move motor Forward	Passed
3	Motor = Motor1	Move motor 1	Passed
4	Motor = Motor2	Move motor 2	Passed
5	DutyCyle= 0%	Move motor with PWM = 0%	Passed
6	DutyCyle= 50%	Move motor with PWM = 50%	Passed
7	DutyCyle= 80%	Move motor with PWM = 80%	Passed
8	Invalid values	Motor won’t move	Passed

8.1.2 Simulation Testing

Table 11 Test case 10

Test Case ID	10		
Test Case Description	Testing the publishing of pose information		
Pre-Requisite	Function requires two inputs - pos of type Vector3 and rot of type Quaternion. Valid input values should be provided. 2. Valid ROS instance should be running and accessible 3. frameId and topic variables should be correctly defined 4. An instance of the Clock should be available and accessible to get the current time for the timestamp.	**Post-Requisite**	Pose information should be successfully published to the specified ROS topic.

Test Case Number	**Input (pos, rot)**	**Expected Output (poseMsg)**	**Test Result**
1	pos = (0,0,0), rot = (0,0,0,1)	poseMsg.pose.position = (0,0,0), poseMsg.pose.orientation = (0,0,0,1)	Passed
2	pos = (1,1,1), rot = (0,0,0,1)	poseMsg.pose.position = (1,1,1), poseMsg.pose.orientation = (0,0,0,1)	Passed

Test Case Number	**Input (pos, rot)**	**Expected Output (poseMsg)**	**Test Result**
3	pos = (-1,-1,-1), rot = (0,0,0,1)	poseMsg.pose.position = (-1,-1,-1), poseMsg.pose.orientation = (0,0,0,1)	Passed
4	pos = (0,0,0), rot = (1,0,0,0)	poseMsg.pose.position = (0,0,0), poseMsg.pose.orientation = (1,0,0,0)	Passed
5	pos = (1,1,1), rot = (1,0,0,0)	poseMsg.pose.position = (1,1,1), poseMsg.pose.orientation = (1,0,0,0)	Passed
6	pos = (-1,-1,-1), rot = (1,0,0,0)	poseMsg.pose.position = (-1,-1,-1), poseMsg.pose.orientation = (1,0,0,0)	Passed

Table 12 Test case 11

Test Case ID	11		
Test Case Description	Testing the publishing of pose information using a Transform target		
Pre-Requisite	Function requires one input - target of type Transform. Valid input value should be provided 2. Valid ROS instance should be running and accessible 3. frameId and topic variables should be correctly defined 4. An instance of the Clock should	**Post-Requisite**	Pose information should be successfully published to the specified ROS topic.

	be available and accessible to get the current time for the timestamp		

Table 13 Test case 12

Test Case ID	Input (Transform target)	Expected Output	Test Result
1	Null	Exception (e.g., ArgumentNullException)	Passed
2	Transform object with position (1,1,1) and rotation (0,0,0,1)	PoseStampedMsg object with position (1,1,1), orientation (0,0,0,1), and current timestamp published to ROS	Passed
3	Transform object with position (-2,-2,-2) and rotation (0,0,0,-1)	PoseStampedMsg object with position (-2,-2,-2), orientation (0,0,0,-1), and current timestamp published to ROS	Passed
4	Transform object with position (0,0,0) and rotation (1,1,1,1)	PoseStampedMsg object with position (0,0,0), orientation (0.5,0.5,0.5,0.5), and current timestamp published to ROS	Passed

Test Case ID	12		
Test Case Description	Testing RobotInput function with valid speed and rotational speed values within maximum limits		
Pre-Requisite	Function requires two inputs - speed and rotSpeed of type float. Valid input values should be provided. 2. Properties of the robot such as maxLinearSpeed,	**Post-Requisite**	Robot's wheel speeds are correctly set and result in expected linear and rotational movements

	maxRotationalSpeed, wheelRadius, and trackWidth should be properly set. 3. SetSpeed() function should be available and working correctly		

Test Case ID	Input	Expected Output	Test Result
1	`speed` = 0.5 m/s, `rotSpeed` = 0 rad/s	`wheel1Rotation` and `wheel2Rotation` should be set such that it reflects a linear speed of 0.5 m/s and a rotational speed of 0 rad/s	Passed
2	`speed` = 2 m/s (assuming this is greater than `maxLinearSpeed`), `rotSpeed` = 0 rad/s	`wheel1Rotation` and `wheel2Rotation` should be set such that it reflects a linear speed equal to `maxLinearSpeed` and a rotational speed of 0 rad/s	Passed
3	`speed` = 0.5 m/s, `rotSpeed` = 2 rad/s (assuming this is greater than `maxRotationalSpeed`)	`wheel1Rotation` and `wheel2Rotation` should be set such that it reflects a linear speed of 0.5 m/s and a rotational speed equal to `maxRotationalSpeed`	Passed
4	`speed` = 0 m/s, `rotSpeed` = 0 rad/s	Both `wheel1Rotation` and `wheel2Rotation` should be set to zero, indicating the robot has stopped	Passed

Table 14 Test case 13

Test Case ID	13		
Test Case Description	Testing the IK function		
Pre-Requisite	The terations, joint array, minAngle and maxAngle arrays should be defined and properly initialize	**Post-Requisite**	The robot arm angles should be correctly adjusted according to the calculated inverse kinematics. prevAngle should be updated with the new angles.

Test Case ID	**Input**	**Expected Output**	**Test Result**
1	Providing a set of `joint` values, `minAngle` and `maxAngle` that will produce an error less than 1E-3	`angle` values should be adjusted according to the calculated inverse kinematics, `prevAngle` should be updated with the new angles, and no changes should be made to the original joint values.	Passed
2	Providing a set of `joint` values, `minAngle` and `maxAngle` that will produce an error greater than 1E-3	The `angle` values should be reset to their `prevAngle` values.	Passed
3	Providing a set of `joint` values, `minAngle` and `maxAngle`	The `angle` values should be reset to their `prevAngle` values.	Passed

Test Case ID	Input	Expected Output	Test Result
	that will result in calculated angles out of the limits		

Table 15 Test case 14

Test Case ID	14.Kinect camera testing		
Test Case Description	Publish depth and point to cloud data from Kinect camera		
Pre-Requisite	• ensure that a custom rviz config file exists at the specified file path before running the tests.	**Post-Requisite**	• After executing the test cases, verify that the generated launch description includes the expected rviz config file path as specified in the test case.

Test Execution Steps:

S.No	Inputs	Expected Output	Test Result
1	Empty input	The launch description contains three entities with the expected properties	Passed
2	Valid input	The launch description includes the expected launch arguments and nodes	Passed

6.1.3 Sensors gathering node testing.

Table 16 Test case 15

Test Case ID	15.Sensor data testing		
Test Case Description	Test continuous message publishing of the sensor data retrieved from the STM		
Pre-Requisite	• ensure that data is being received correctly from STM where sensors are interfaced.	**Post-Requisite**	• Verify that messages are continuously published. • Verify graceful loop exit and proper shutdown.
Test Execution Steps:			

S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input	Messages are continuously published until a keyboard interrupt signal is received	Passed
2	Keyboard interrupt signal	The publishing loop exits gracefully, and the ROS 2 node	Passed

		is properly destroyed and shut down	

6.1.4 Slam tool-box testing

Table 17 Test case 16

Test Case ID	16.Mapping testing		
Test Case Description	Test that slam tool-box successfully generates a map		
Pre-Requisite	• Kinect camera send depth camera data to slam toolbox	**Post-Requisite**	• Verify that a map is being constructed.
Test Execution Steps:			
S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input	The generated launch description includes the default values for use_sim_time and slam_params_file. No map is constructed.	Passed
2	Custom input: Custom values for use_sim_time and slam_params_file	The generated launch description includes the custom values for use_sim_time and slam_params_file. The launch description contains the DeclareLaunchArgument and Node entities with the	Passed

		expected custom properties.	

6.1.5 AMCL testing

Table 18 Test case 17

Test Case ID	17.Localization testing		
Test Case Description	Test that robot is able to perform localization successfully		
Pre-Requisite	• Robot is existing in environment. • A map is provided.	**Post-Requisite**	• Particles are projected around the robot locating its actual position in the environment.

Test Execution Steps:			
S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input	The generated launch description should include the default values for all launch arguments and nodes.	Passed
2	Custom input: Custom values for use_sim_time and slam_params_file	The generated launch description should include the custom values for launch arguments and nodes.	Passed

6.1.6 Navigation testing

Table 19 Test case 18

Test Case ID	18.Navigation testing		
Test Case Description	Test that robot is able to perform navigation successfully		
Pre-Requisite	• Robot is localized properly. • Goal pose is within the environment.	**Post-Requisite**	• none

Test Execution Steps:

S.No	**Inputs**	**Expected Output**	**Test Result**
1	No input	The generated launch description should include the default values for all launch arguments and nodes.	Passed
2	Custom test case with specific launch arguments and nodes	The generated launch description should include the custom values for launch arguments and nodes.	Passed

Chapter 9
FUTURE WORK

- **Hardware dynamic obstacle avoidance**

Hardware Dynamic Obstacle Avoidance: While we have successfully implemented dynamic obstacle avoidance in simulation, our next step is to translate this capability into hardware. This involves integrating sensors and navigation algorithms to enable real-time obstacle detection and avoidance in physical warehouse environments.

- **Shelf robot**

Shelf Robot: We aim to develop and deploy a specialized robot, known as a shelf robot, which will be responsible for retrieving packages from the mobile robot and accurately shelving them. This will enhance the efficiency and organization of the warehouse system by automating the process of stocking and retrieving items.

- **Package tracking**

Package Tracking: Another aspect of our future work is the implementation of package tracking. This feature will enable precise tracking of the movement of packages, from their arrival to their final placement in the warehouse. Accurate package tracking is crucial for effective inventory management in an autonomous warehouse system, reducing the risk of mismanaged inventory and ensuring seamless order fulfillment.

- **Delivery notification**

In our future plans, we aim to achieve full automation of the delivery process. This entails developing a system where the delivery vehicle simply notifies our system about an incoming package or placed order. From that point onwards, the entire process will be seamlessly executed without any human intervention or interaction with the system. Our objective is to streamline and automate the delivery process, eliminating the need for manual intervention and optimizing efficiency.

Chapter 10
CONCLUSION

In conclusion, the rapid growth of large warehouse systems worldwide necessitates the adoption of automated or semi-automated solutions to overcome the limitations of manual warehouse management. Manual methods suffer from various drawbacks, including delays in shipping, inadequate inventory tracking, high labor costs, and safety risks for workers. To address these challenges, alternative solutions such as robotic systems have been introduced by companies like ABB Ltd and Amazon.

Our proposed solution focuses on three key components: delivery notification, an automated convoy system, and an autonomous robot equipped with advanced object detection capabilities. By implementing these automated processes, warehouse management can achieve significant benefits such as improved delivery efficiency, accurate inventory tracking, reduced labor costs, enhanced safety measures, and precise supply forecasting. These advancements will optimize warehouse operations, enhance productivity, improve cost-effectiveness, and ensure the well-being of workers.

Chapter 11
REFERENCES

1. Anggraeni, P., Mrabet, M., Defoort, M., & Djemai, M. (2018). *Development of a wireless communication platform for multiple-mobile robots using ROS*. https://doi.org/10.1109/ceit.2018.8751845

2. Endres, F., Hess, J., Sturm, J., Cremers, D., & Burgard, W. (2014). 3-D mapping with an RGB-D camera. *IEEE Transactions on Robotics*, *30*(1), 177–187. https://doi.org/10.1109/tro.2013.2279412

3. García, A., Martín, F., Guerrero, J. M., Rodríguez, F. J., & Matellán, V. (2023). *Portable Multi-Hypothesis Monte Carlo localization for mobile robots*. https://doi.org/10.1109/icra48891.2023.10160957

4. Cheng, P. D. C., Indri, M., Sibona, F., De Rose, M., & Prato, G. (2022). Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. https://doi.org/10.1109/etfa52439.2022.9921458

5. Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, ChristopherSTAN, Liu Changyu, Laughing, Adam Hogan, lorenzomammana, tkianai, yxNONG, AlexWang1900, Laurentiu Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Hatovix, Jake Poznanski, … yzchen. (2020). ultralytics/yolov5: v3.0 (v3.0). Zenodo.

6. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. ArXiv. /abs/1707.06347

7. Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2021). The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games. ArXiv. /abs/2103.01955