INTRODUCTION

Autonomous navigation is a rapidly growing field and has become a pivotal area of research and application in robotics, with numerous applications in industries such as transportation, manufacturing, and warehousing. One of the fundamental tasks for autonomous robots is the ability to navigate their environment independently with minimal human intervention, that is a crucial factor or a key milestone for the development of intelligent systems capable of interacting with the real world. Among the various techniques used for autonomous navigation, line following remains one of the most essential and widely researched methods, especially for mobile robots, where the robot must track a specific line on the ground. While it may appear straightforward, line following involves a variety of challenges, including sensor calibration, precise control of the robot's motors, and handling of unpredictable environmental factors.For this project the primary objective was eventually designing and developing a simulated robot capable of accomplishing a specific task while following a line and avoiding obstacles and maintain accurate control over its movements within a simulated environment using the **Robot Operating System (ROS)**, which is widely used in both research and industry due to its flexibility, modularity, and wide range of libraries and tools. ROS provides a powerful framework for developing and simulating robotic applications.

METHODS USED TO DESIGN:

It is necessary outline the methodology used to design the robot, its navigation system, and the simulation environment. The method should cover both the hardware (even though it's simulated) and software design, as well as the control algorithms used to ensure the robot follows the line autonomously.

The design of the autonomous line-following robot involved a series of methodical steps to ensure the successful integration of hardware (simulated), software (ROS-based), and control algorithms to enable autonomous navigation. The following sections describe the approach taken to design and implement the robot and its navigation system in the simulation environment.

0.1 FOR LINE DETECTION AND FOLLOWING:

0.1.1 Camera-based Vision Method:

Using **cameras** for line-following is an advanced approach in robotics, where the robot uses visual information to detect and track a line or path. This technique, often referred to as **vision-based line following**, involves utilizing computer vision algorithms to process images captured by the robot's camera and determine the robot's position relative to the line. To use a camera for line-following, the camera is needed to be mounted on the robot, typically facing downward or slightly tilted to capture the surface on which the line is drawn which is the case with the robot used for this project.

This process was accomplished through many steps via simulation such :

SETTING UP THE SIMULATION ENVIRONMENT

The simulation platform was set as with the following components:

- **Robot Model**: The robot in the simulation should have a camera mounted on it. The robot model could be a differential-drive robot, a mobile platform with wheels, or a more complex robot with additional sensors. **The robot model used is a differential-drive mobile robot.**
- **Camera Sensor**: Simulated cameras in the environment will capture images of the ground, where the line is located. The camera was set up to view downward, directly in front of the robot.
- **Line (Path)**: lines on the ground can be created in the simulation. The line can be straight, curved, or even follow a more complex path. For this project black lines were drawn within the world created in Gazebo.
- **Simulator**: The simulation platform used is:

  Gazebo: Commonly used with **ROS** (Robot Operating System) for robot control and visualization.

IMAGE PROCESSING FOR LINE DETECTION

After setting up the simulated robot and camera, the robot will need to process the images captured by the camera to detect the line. The core part of this system involves **image processing** and **vision algorithms**.

- **Steps for Image Processing:**
- **Capture Image from the Camera**:

  In the simulation, it is necessary to get access to the camera feed in real-time. With **ROS**, there can be subscriber to the camera's image topic such **/camera/rgb/image_raw**

- **Convert the Image to Grayscale**:

  grayscale conversion can be used to reduce the complexity of the image. Since the line will typically have a contrasting colour (black or white), working with a grayscale image makes it easier to detect the line.

- **Thresholding** will convert the grayscale image into a binary image where the line is white (or black) and the background is the opposite colour. You can use a simple threshold or adaptive thresholding for better results in varying lighting conditions.
- **Detect the Line**:

Once the image is binary, **edge detection**, **contour detection**, or **Hough Transform** can be used cto detect the line. For instance, you can use **Hough Line Transform** to detect straight lines. For the project they were combined and used together for line detection.

- **Determine the Robot's Position Relative to the Line**:

  The centroid of a detected line is calculated t hoo measure or estimate w far the robot is from the centre of the line in the camera frame.

0.1.2 1.2 Control Algorithm for Line Following

Once the line is detected, the robot needs to be controlled to follow it. The basic principle is to adjust the robot's steering based on its position relative to the line. PID Controller algorithm can be used to keep the robot following line in a steady way.

The **PID (Proportional-Integral-Derivative)** controller is commonly used in line-following robots.

- **Proportional**: The steering correction is proportional to how far the robot is from the line's centre.
- **Integral**: This helps eliminate accumulated errors over time.
- **Derivative**: This predicts and reacts to the rate of change in the error (speed of deviation).

0.1.3 1.3 Integrating the Line Following with the Simulator

Finally, the image processing and control algorithms needed to be integrated into the simulation environment. With ROS this integration is usually made through Gazebo. When **ROS** is used the robot cab ne set up with a simulated camera and subscribe to the camera feed. ROS provides libraries like CV BRIDGE to convert the ROS image messages into

0.2 FOR BUILDING MAP

0.2.1 LIDAR

The map was built using SLAM (Simultaneous Localization and Mapping) in a simulation thanks to the lidar sensor. Using LIDAR (Light Detection and Ranging) with SLAM (Simultaneous Localization and Mapping) is a common and effective approach for building a map of an environment while simultaneously localizing the robot within that environment. In ROS, LIDAR is often used as the primary sensor for SLAM, especially for 2D SLAM algorithms like GMapping and Hector SLAM. The algorithm used for SLAM was GMapping in this project.The lidar can be used to avoid obstacles as well.

0.2.2 QR Code Scanning

QR Code Detection with OpenCV was used, the OpenCV's QRCodeDetector detects and decode QR codes in the robot's camera feed. The QR code are used to help the robot to navigate autonomously, the QR code information to know its current position and calculate the closest path to the destination.

After building the map , it will be saved for autonomous navigation purpose.

Chapter 1

Realistic constraints

1.1 COMPUTATIONAL-POWER-AND-RESOURCES

1.1.1 cpu-and-gpu-limitations

- *CPU Limitations*: Simulations, especially those involving complex algorithms like SLAM or sensor fusion, can be CPU-intensive. The computational demand increases with the complexity of the robot's tasks, such as real-time mapping, localization, or decision-making. A limited CPU performance could cause delays, lag, or even make real-time processing difficult.
- *GPU Constraints*: If you're using computer vision algorithms (e.g., QR code recognition, camera-based line-following), a GPU can greatly accelerate image processing. However, not all simulations leverage GPU power, and if the GPU is not sufficient or not utilized properly, the simulation could run slower or experience bottlenecks.

1.1.2 memory-ram-constraints

- *High Memory Usage*: Simulations that involve large environments, dense sensor data (such as LIDAR point clouds), or high-resolution images can require a significant amount of RAM. If the memory usage exceeds the available capacity, it can lead to slow performance, crashes, or even the inability to run the simulation at all.
- *Data Storage for Sensor Data*: Storing large amounts of sensor data (e.g., from LIDAR, cameras, IMUs) generated during a simulation can strain the system's storage, especially if you need to log or save sensor outputs for later analysis. In a large-scale simulation, this could be a limiting factor.

1.1.3 real-time-performance

- *Real-Time Simulation Constraints*: Achieving real-time performance in simulations can be difficult, especially when dealing with complex tasks such as real-time SLAM or path planning. The simulation might not run at the required frame rate (e.g., 30Hz or higher), leading to delays in robot control and decision-making.
- *Simulation Time vs Real Time*: If the simulation is not running at real-time speed (1:1 with physical time), it can make it harder to validate time-sensitive behaviors. For

example, a robot using SLAM may not update its map fast enough in a simulation that runs too slowly, affecting navigation and decision-making in real-world applications.

1.2 COMPLEXITY-OF-THE-SIMULATED-ENVIRONMENT

1.2.1 size-and-detail-of-the-simulation

- *Environmental Complexity*: Large and complex simulated environments with many dynamic obstacles, detailed textures, and various types of surfaces can be computationally expensive. More detailed environments require more processing power to render, simulate physics, and handle sensor data.
- *Realistic Physics*: Physics engines in simulations (e.g., Gazebo, V-REP) simulate the interactions between the robot and the environment, including forces like friction, gravity, and object collisions. While necessary for realistic testing, these physics simulations can be computationally demanding, especially in dynamic environments with many objects.

1.2.2 dynamic-objects-and-movements

- *Real-Time Object Simulation*: When simulating environments with moving objects (e.g., pedestrians or vehicles), the computation required to simulate their motion and interactions with the robot can add significant overhead. This becomes particularly challenging if multiple objects are moving in complex patterns at the same time.

1.3 SIMULATING-SENSOR-DATA

1.3.1 a.-sensor-data-processing-load

- *LIDAR and Point Cloud Data*: LIDAR sensors generate large amounts of data, especially in 3D environments. Processing the point cloud data in real time requires significant computing resources, particularly when applying algorithms like SLAM to build and update maps. The more data points and the larger the map, the more processing power is required.
- *Camera Data for Visual Processing*: Cameras generate high-resolution images that need to be processed for tasks like QR code detection or line-following. Processing these images (especially with advanced computer vision techniques such as feature detection or deep learning) can be computationally expensive. Depending on the image resolution and the complexity of the detection algorithms, this could put a strain on the available computing resources.

1.3.2 real-time-sensor-fusion

- *Combining Data from Multiple Sensors*: If you are fusing data from multiple sensors (e.g., combining LIDAR, IMU, camera data for SLAM), the computational load increases significantly. Sensor fusion algorithms, such as Kalman Filters or particle

filters, need to process data from all sensors in real time, which may not be feasible with limited computational resources.

1.4 ALGORITHMIC-CONSTRAINTS

1.4.1 a.-computational-cost-of-slam

- *SLAM Algorithm Complexity*: Algorithms used for SLAM (like Extended Kalman Filters, GraphSLAM, or particle filters) are computationally expensive, especially when the robot has to map a large area in real-time. As the environment grows, the number of calculations needed to maintain and update the map increases, potentially leading to slower performance or the need for more powerful hardware.

- *Map Size and Resolution*: The higher the resolution and accuracy of the map, the more computational resources are required to store and update it. Large maps with high-detail features demand significant memory and processing power to handle updates, store the data, and process localization.

1.4.2 b.-path-planning-and-decision-making

- *Complex Path Planning Algorithms*: Path planning algorithms, such as A*, RRT, or D* algorithms, may require substantial computational resources depending on the complexity of the environment. If your robot is navigating a large or cluttered environment, calculating collision-free paths in real-time becomes a challenge, especially if there are many dynamic obstacles to avoid.

- *Decision-Making Algorithms*: When the robot makes decisions based on sensor input, running machine learning algorithms or decision trees to determine the next action (e.g., go towards a QR code or follow a line) can also be computationally expensive. Depending on the complexity of the logic, this could limit the speed and responsiveness of the simulation.

1.5 SIMULATION-SOFTWARE-LIMITATIONS

1.5.1 simulation-engine-efficiency

- *Performance of the Simulation Engine*: Different simulation platforms (Gazebo, V-REP, Webots, etc.) have varying levels of efficiency. Some simulators might be highly optimized for certain types of robots or algorithms, while others may be slower or require more resources to simulate complex systems. Choosing the right simulation platform for your application is critical for balancing performance and accuracy.

- *Plugin Overhead*: Many simulators rely on plugins for additional functionality (e.g., camera sensors, LIDAR). The more plugins you add or the more complex the plugin behavior, the more computational overhead is introduced. This can affect simulation performance, especially in large-scale environments or real-time applications.

1.5.2 parallelization-and-multithreading

- *Limited Parallel Processinh*: Not all simulation environments or algorithms are well-optimized for parallel processing, which means that tasks may not fully utilize multi-core CPUs or GPUs. For example, in simulations involving real-time SLAM, the ability to parallelize sensor data processing or SLAM computation can significantly reduce the computational load, but not all algorithms or environments support this effectively.

- *Threading Issues*: Running multiple processes (e.g., sensor readings, actuator control, SLAM) in parallel in a simulation may introduce issues such as deadlock or race conditions if not properly managed. This can lead to slower simulation times or unresponsive behavior in your robot model.

1.6 SIMULATING-REAL-TIME-CONSTRAINTS

1.6.1 real-time-control-vs.-simulation-speed

- *Synchronization Issues*: Achieving true real-time performance is difficult in a simulation. Many simulations allow you to adjust the time scale (e.g., running the simulation faster than real time for testing), but this introduces the issue that the control algorithms might not have the same timing constraints they would in a real robot. Ensuring that the simulation runs with a fixed time step, synchronized with the robot's control loops, is essential but can be computationally demanding.

- *Hardware-in-the-loop (HIL) Simulation*: When integrating real hardware with a simulator (e.g., for testing SLAM on real sensors), the latency and real-time computation required can cause performance bottlenecks. This is especially true if communication between the simulation and physical robot hardware is slow or not synchronized properly.