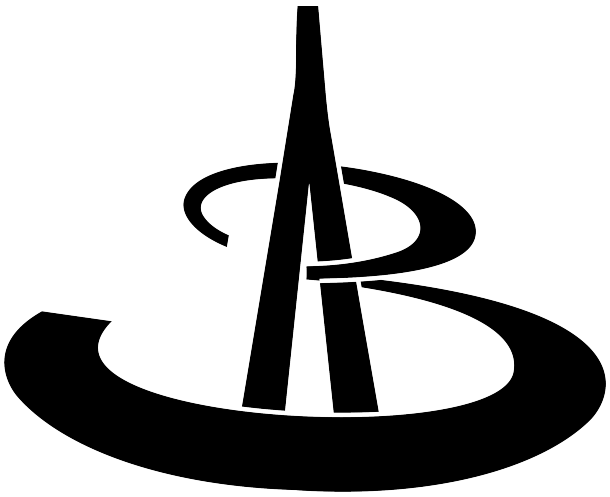

Inter IIT Tech Meet 13.0

Report for Mid-Prep Problem Statement

BharatForge:

Centralized Intelligence for Dynamic Swarm Navigation

Team ID: 57

December 2024

Contents

1. Problem Understanding

1.1 Overview

The problem centers on addressing the constraints of GPS-independent navigation for Autonomous Mobile Robots (AMRs) in industrial settings. The present dependence on ArUco markers for navigation and object detection is laborious and lacks scalability, rendering it inappropriate for dynamic environments such as warehouses and manufacturing lines. The objective is to develop a resilient, scalable system enabling robots to move independently while coordinating efficiently as a swarm.

1.2 Constraints and Requirements

A primary issue in this project was navigating indoor locations absent of GPS support, requiring the creation of a reliable alternative for accurate navigation and localization. The existence of both static and dynamic impediments, such as mobile machinery and unpredictable human movements, increased the complexity of the system's design. Furthermore, the system required scalability, ensuring optimal performance in larger settings with increased robot quantities, necessitating careful optimization of computational resources and memory management. Addressing these limits was essential for attaining an adaptive and flexible system.

1.3 Initial Analysis and Key Challenges

Modifying navigation algorithms to accommodate frequent alterations in dynamic settings was a principal problem in this study. The robots depended on inbuilt sensors and algorithms to identify and react to dynamic barriers such as humans or machinery in real time, assuring continuous functioning. Meanwhile, sustaining effective communication within the swarm was essential for cooperative navigation. Each robot was required to distribute updated environmental data and dynamically coordinate tasks to prevent collisions and enhance travel routes.

Moreover, the management of resource utilization became progressively vital as the system expanded to more extensive habitats and swarms. Complex optimization techniques were needed to make it possible for real-time processing, memory efficiency, and algorithmic precision to work together without any problems. All of these issues made it clear how important it is to have a strong, scalable architecture that can meet the needs of industrial automation.

2. Methodology and Approach

2.1 Swarm Navigation and Environment Setup

2.1.1 Initial Exploration

For the initial map exploration, we use a single robot. The robot leverages the open-source navigation stack, `nav2`, to navigate unexplored areas. While exploring, it detects objects in the environment and records their coordinates.

Object Coordinate Assignment

The YOLO model provides object coordinates in the image frame. By dividing these coordinates by the image size, the laser angle (α) pointing to the object is determined. Using odometry data (x_1, y_1, yaw) and the laser distance (d), the object coordinates (x, y) in the map frame are calculated as:

$$x = d \cdot \cos(\alpha) + x_1$$

$$y = d \cdot \sin(\alpha) + y_1$$

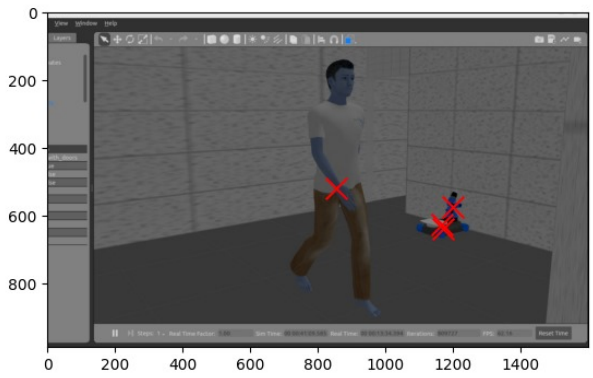

Figure 2.1: Object dectection by the YOLO model

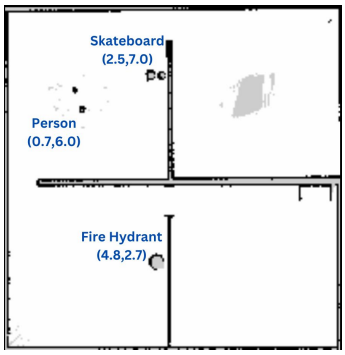

Figure 2.2: Global coordinates of objects assigned by robot

2.1.2 Robot Swarm Architecture

The robots aim to distribute themselves efficiently to ensure optimal task execution efficiency. The details of how this distribution is calculated is discussed in 2.2.3.

2.2 AI/ML Models and Algorithms

We leveraged artificial intelligence & machine learning models and algorithms for the following purposes.

2.2.1 Object Detection

For object detection through bots' onboard cameras, we have implemented a class for performing object detection using the **YOLOv8** model, specifically the lightweight version **yolov8n.pt**, which is designed for rapid inference while maintaining high accuracy. The input to our model is an image of what our bot can see at this moment. The model detects objects in the view and outputs their location. The output is a list of objects detected, where each object is represented by its coordinates and class label.

2.2.2 Clustering of detected object coordinates

We have made use of clustering for getting an approximate centroid for a particular object (as there can be multiple coordinates for a given object due to slight variations in the coordinates' readings), so that whenever we are supposed to reach that particular object, we can use its centroid.

For clustering, we have made use of the **elbow method**, wherein we approximate that the given data points can be clustered into at most 10 clusters. We fit multiple **K-means clustering** models on the given data points with the number of clusters (K) varying from 1 to 10. For each model, we store the inertia of our model.

In the context of k-means clustering, **inertia** is a metric used to measure how well the clusters have been formed. It represents the sum of squared distances between each data point and the centroid of its assigned cluster.

We also have a threshold set, the main use of this threshold is that we get to know the optimum value of K for our model. So when the ratio of (inertia of our model at num_clusters=K+1) and (inertia of our model at K) goes below the threshold, we make use of the cluster assigned by our model having num_clusters set to K.

Figure 2.3: Variation in inertia with respect to number of clusters

2.2.3 Distributing Bots Evenly Across the Map

When the bots are idle (unassigned), we aim to spread them evenly across the map to increase the likelihood of having a bot near a task when it's assigned.

To achieve this, we select points uniformly across the accessible regions of a **grayscale map**, emphasizing areas with higher intensity. The selection process incorporates a weighted probability distribution and spatial constraints such as a minimum distance between points (exclusion radius) and safe distance from walls. The key steps involved are:

- **Weighted Image Filtering:** Apply a `Gaussian filter` to the image to smooth it and reduce noise, emphasizing central regions and making areas of interest more prominent.

- **Binary Thresholding:** Threshold the filtered image to create a binary map, where intensity values above a certain threshold represent accessible areas (white), and others represent obstacles (black).
- **Distance Transform Calculation:** Compute a `distance transform` to measure the distance from each accessible pixel to the nearest obstacle. This ensures selected points are sufficiently far from walls, based on the exclusion radius.
- **Weighted Random Point Selection:** Select points within the accessible regions using a weighted probability distribution. The probability of selecting a point is proportional to the square of its intensity value, favoring brighter (higher intensity) areas.
- **Enforcing Exclusion Constraints:** Ensure that each selected point satisfies the minimum distance constraint from other points (exclusion radius) and maintains a safe distance from walls as determined by the distance transform. If a point violates these constraints, it is discarded, and the selection process repeats.
- **Intensity Reduction Around Selected Points:** After a valid point is selected, subtract a Gaussian distribution from the surrounding area in the image. This reduces the intensity around the point, decreasing the probability of nearby points being selected and promoting an even distribution.

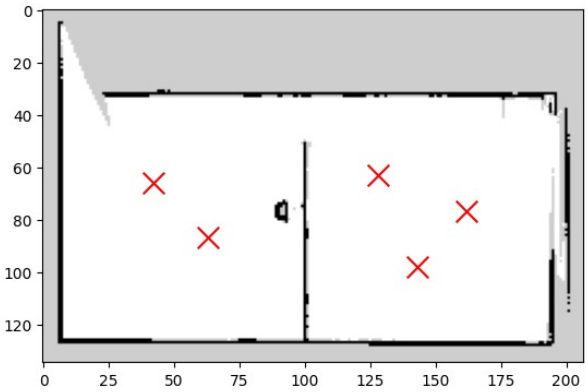

Figure 2.4: Demo with distributing 5 points evenly across the map.

2.3 The Central Server

The central server is responsible for managing tasks and allocating them to robots in an efficient and effective manner. It coordinates the overall operation by utilizing several key classes and algorithms, which are detailed below.

2.3.1 Class Structures

Task Class

The `Task` class represents a task that needs to be performed by a robot. Each task includes the following major attributes:

- `task_id`: A unique identifier for the task.
- `object_type`: The type of object associated with the task.
- `required_capability`: The capability required to perform the task (e.g., `'carry'`, `'inspect'`, `'repair'`).
- `final_place`: (Optional) The final destination place for the object.

- `priority`: The priority level of the task.
- `wait_time`: The total time the task has been waiting to be assigned.

Robot Class

The `Robot` class models a robot that can perform tasks. The key attributes and methods include:

- `robot_id`: A unique identifier for the robot.
- `position`: The current position of the robot on the grid.
- `capabilities`: A list of capabilities that the robot possesses.
- `reliability`: The probability that the robot will successfully complete a task.
- `current_task`: The task currently assigned to the robot.
- `task_queue`: A queue of `SimpleTask` instances representing the steps to complete the task.
- `calculate_bid()`: Calculates a bid value for a task based on multiple parameters.

TaskAllocator Class

The `TaskAllocator` class is responsible for managing tasks and allocating them to robots. Its key components are:

- `robots`: A list of all robots in the system.
- `tasks`: A list of all tasks in the system.
- `object_positions`: A mapping of object types and place names to their positions.
- `allocate_tasks()`: Allocates tasks to robots based on bids.

2.3.2 Allocation Algorithm

The allocation algorithm is auction-based, where robots bid on tasks they are capable of performing. The bidding process involves calculating a bid value for each robot-task pair, using several parameters:

Bid Parameters

The bid value is calculated as a weighted sum of several factors:

- **Distance** (w_{distance}): The distance between the robot's current position and the task's position.
- **Capability Score** ($w_{\text{capability}}$): A factor penalizing robots that lack the required capability.
- **Capability Potential** ($w_{\text{cap\_potential}}$): The sum of priorities for the robot's unused capabilities.
- **Battery Level** (w_{battery}): A factor considering the robot's battery level.
- **Reliability** ($w_{\text{reliability}}$): A factor penalizing less reliable robots.

The bid value B for a robot is calculated as:

$$\begin{aligned} B = {} & w_{\text{distance}} \times \text{distance} \\ & + w_{\text{capability}} \times \text{capability\_score} \\ & + w_{\text{cap\_potential}} \times \text{capability\_potential} \\ & + w_{\text{obstacle}} \times \text{dynamic\_obstacle\_prob} \\ & + w_{\text{reliability}} \times (1 - \text{reliability}) \end{aligned}$$

After receiving bids from all the unassigned bots, we choose the bot with the lowest bid and assign the task to it.

2.3.3 Scheduling Algorithm

The scheduling algorithm ensures that tasks are allocated efficiently while considering priority and wait time.

Wait Time and Priority

- **Wait Time**: The time a task has been waiting to be assigned. Tasks with higher wait times are prioritized.
- **Priority**: A static priority level assigned to tasks. Higher priority tasks are given preference.
- **Wait Time Update**: The wait time is increased based on the task's priority (`wait_time += priority * 10`).

Task Reassignment

- Tasks that fail due to robot failure or other issues are marked as `unassigned` and re-enter the task queue.
- The wait time and priority of these tasks are updated to reflect their increased urgency.

In every scheduling cycle, the tasks with the higher wait times are selected first for allocation.

2.4 Chat Interface

The chat interface provides a user-friendly way for operators to interact with the task allocation system. It allows users to input commands in natural language, which are then interpreted and executed by the system. The interface is built upon several key components, including a system prompt for the AI assistant, server implementation using Flask, session management, and secure parsing of the assistant's responses.

2.4.1 System Prompt

A carefully crafted system prompt guides the LLM to understand and process user commands accurately. The system prompt includes information about the system's capabilities, available objects, locations, and the formats for the commands that the assistant can interpret. It also provides examples to illustrate how the assistant should structure its responses. The prompt ensures that the assistant outputs responses in a consistent and parse-able format.

2.4.2 Server Implementation

The server is implemented using Flask, a lightweight web framework for Python. It exposes two main routes:

- `/login`: Allows users to log in by providing a username. A unique session ID is generated for each user and stored in a cookie.
- `/chat`: Handles user commands sent to the system. It retrieves the user's session ID from the cookie, processes the command, and returns the assistant's response along with any actions taken.

2.4.3 Session Management

To maintain user-specific conversation histories, the server generates a unique session ID for each user upon login. This session ID is stored in a cookie on the client's side and automatically sent with subsequent requests. The server uses this session ID to retrieve and update the user's conversation history, ensuring personalized interactions with the assistant.

2.4.4 Assistant Response Parsing

The assistant's responses are expected to be in predefined formats as defined in the system prompt, such as:

- `Task(object_type='part', required_capability='carry', final_place='assembly_area', priority=0.1)`
- `AddRobot(position=(2, 3), capabilities=['carry', 'inspect'], battery_level=0.9, reliability=0.95)`

The server uses regular expressions to extract key-value pairs from these responses. For instance, it extracts the parameters inside the parentheses and creates a dictionary of arguments. Numeric values and collections are safely parsed using `ast.literal_eval()`, which evaluates strings containing Python literals without executing arbitrary code.

2.4.5 Integration with Central Server

The chat interface integrates with the central server's task allocator and robot management system. When the assistant's response is parsed and valid commands are extracted, and it calls the corresponding functions using shared `TaskAllocator` instance. This allows tasks to be added, robots to be registered, and objects or locations to be incorporated into the system in real-time.

2.5 Overall Workflow

The system comprises four main components: Chat Interface, Task Allocator and Scheduler, Autonomous Navigation, and Simulation World. The workflow proceeds as follows:

- **Chat Interface:** Users provide prompts, which are processed to generate tasks. Responses are updated based on simulation feedback.
- **Task Allocator and Scheduler:** Maintains a central database for object and robot data. Tasks with maximum wait time are assigned to bots based on a bidding process, selecting the bot with the lowest bid.
- **Simulation World:** Loads the environment, spawns multiple bots, and facilitates navigation through map updates and the Nav2 stack.
- **Autonomous Navigation:** Handles world loading, bot spawning, autonomous navigation, and saving the updated map data.

The components interact via data exchange and feedback loops, ensuring seamless task execution and navigation.

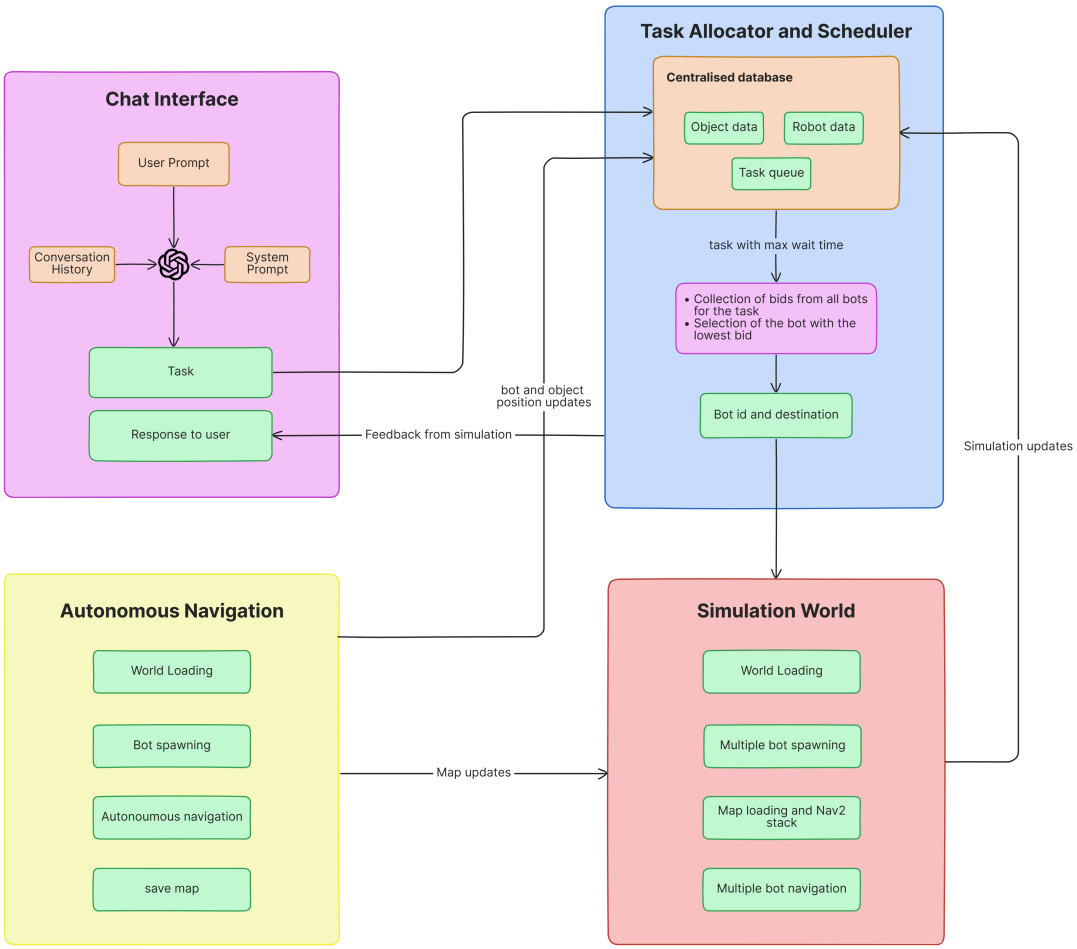

Figure 2.5: Diagram illustrating the overall workflow of our system.

3. Evaluation and Testing

3.1 Test Scenarios

3.1.1 Navigation in a 10x10 world with 3 dynamic obstacles and 5 different objects

- **Navigation of smaller swarms:** Deployed 4 robots in the world.

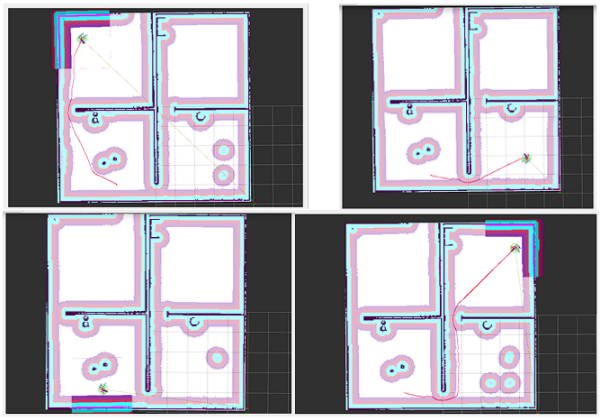

Figure 3.1: Navigation for a small swarm(4 bots)

- **Navigation of larger swarms:** Deployed 6 robots in the world.

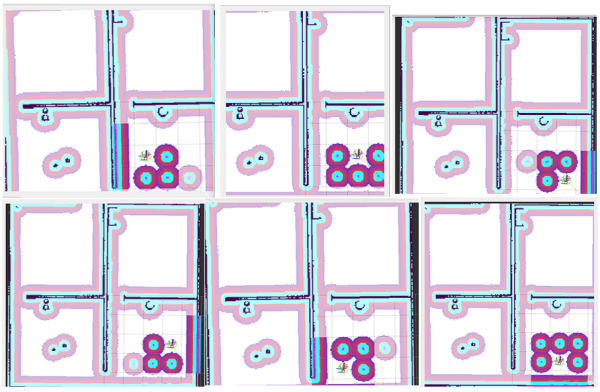

Figure 3.2: Navigation for a larger swarm(6 bots)

4. Conclusion

4.1 Challenges and Lessons Learned

We encountered challenges ranging from detecting the autonomous exploration map to determining the coordinates of objects for navigation. **Autonomous navigation was difficult** as the robot had to know where to explore next. Another challenge was **optimizing autonomous map exploration with multiple robots**. While exploring and creating the map, we were also keeping track of objects detected using computer vision (YOLO model), but we were getting **multiple coordinates with some error for the same object** in the world. Then we had to spawn multiple robots and make them navigate at the same time, while the navigation goals for different bots depend on the number of tasks, distance of task from each robot, and number of ways the task can be completed (for a task of fetching a fire extinguisher, we need to consider all available fire extinguishers present in the environment). **Optimizing robot positions so that they cover equal ground** was another challenge.

Autonomous exploration can be done by finding out **frontiers** in the unexplored map and exploring them until no frontier is left. For multiple robot map exploration, we could implement a **map merging mechanism** while individual robots are exploring and creating respective maps. For the problem of getting multiple coordinates with some error for the same object during exploration, we are using **K-Means Clustering** for aggregation. Multiple robot spawning was handled by changing **ROS2 namespaces** of each robot. The **Navigation2 stack** was used to handle simultaneous robot navigation on the built map. **Task assignment** was done by considering navigation distances from each robot to tasks. For optimizing robot positions, after applying an **exclusion radius** to walls and obstacles, the remaining area was **equally divided among the robots**.

4.2 Future Scope and Recommendations

4.2.1 Use of 3D LiDAR and Depth Cameras

3D LiDAR creates precise environmental maps, enabling robots to navigate complex and scalable setups effectively. Depth cameras enhance obstacle detection by providing spatial data, improving navigation in dynamic environments. Despite their benefits, high costs, increased computational demands, and integration challenges may hinder widespread adoption.

4.2.2 Distributed Computing for Resource Efficiency

Distributed computation optimizes resource management by allocating processing tasks among robots, hence decreasing delay and enhancing performance. However, ensuring flawless communication and synchronization across remote nodes continues to pose challenges.

5. Appendix

Appendix 1 : Dynamic K-Means function

```
import rclpy
from rclpy.node import Node
from custom_interfaces.srv import Kmeans
import numpy as np
from sklearn.cluster import KMeans as SKLearnKMeans
import matplotlib.pyplot as plt

# Dynamic KMeans function
def dynamic_kmeans(data, threshold=0.6, max_clusters=10):
    # Ensure the number of clusters doesn't exceed the number of samples
    n_samples = len(data)
    max_clusters = min(max_clusters, n_samples)  # Limit the max clusters

    inertia_values = []
    cluster_range = range(1, max_clusters + 1)

    for k in cluster_range:
        kmeans = SKLearnKMeans(n_clusters=k, random_state=42)
        kmeans.fit(data)
        inertia_values.append(kmeans.inertia_)

    optimal_num_clusters = 1
    for i in range(1, len(inertia_values)):
        # Avoid division by zero
        if inertia_values[i - 1] == 0:
            relative_improvement = 0
        else:
            relative_improvement = (inertia_values[i - 1] - inertia_values[i]) / \\
            inertia_values[i - 1]

        # If the relative improvement is smaller than the threshold, stop
        if relative_improvement < threshold:
            break
        optimal_num_clusters = i + 1

    # Fit the KMeans with the optimal number of clusters
    kmeans = SKLearnKMeans(n_clusters=optimal_num_clusters, random_state=42)
    kmeans.fit(data)

    centroids = kmeans.cluster_centers_
    labels = kmeans.labels_
```

```
    return optimal_num_clusters, centroids, labels



class KMeansServer(Node):
    def __init__(self):
        super().__init__('kmeans_server')
        self.srv = self.create_service(Kmeans, 'process_kmeans', self.process_kmeans_callback)
        self.get_logger().info("Server is running")

    def process_kmeans_callback(self, request, response):
        # Combine x and y coordinates into 2D array
        data = np.column_stack((request.x_coordinates, request.y_coordinates))

        # Apply KMeans
        optimal_num_clusters, centroids, labels = dynamic_kmeans(data)

        # Prepare response
        response.optimal_num_clusters = optimal_num_clusters
        response.centroids_x = centroids[:, 0].tolist()
        response.centroids_y = centroids[:, 1].tolist()
        response.labels = labels.tolist()

        self.get_logger().info(f"Processed KMeans: {optimal_num_clusters} clusters")

        return response

def main(args=None):
    rclpy.init(args=args)
    kmeans_server = KMeansServer()
    rclpy.spin(kmeans_server)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

6. References

1. Daniel Morgan, Giri P. Subramanian, Soon-Jo Chung, and Fred Y. Hadaegh, “Swarm assignment and trajectory optimization using variable-swarm, distributed auction assignment, and sequential convex programming,” 2016.

2. *ROS 2 Documentation*, available online: https://docs.ros.org/en/rolling/.

3. *Nav 2 Documentation*, available online: https://navigation.ros.org/.

4. *TurtleBot 3 Documentation*, available online: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.