The design of our industrial robot involved a structured approach that included problem formulation, engineering problem-solving, and the application of appropriate analysis and modelling methods. our primary role in the project focused on the programming tasks, ensuring the implementation of the robot's control system, QR code recognition, navigation algorithms, and communication between ROS and the web interface.

0.0.1 Problem formulation

The key engineering challenges we addressed through programming were:

1. *Navigation System with junction detection*: Implementing a control algorithm to follow a black line and detect a junction.
2. *Load and Unload Identification*: Developing a QR code recognition system.
3. *Communication Between ROS and Web Interface*: Ensuring seamless data transfer between the robot's ROS-based system and a web-based user interface.

0.0.2 Engineering problem-solving

0.0.2.1 Navigation System (Line Following and junction detection Algorithm)

The robot needed to detect and follow a black line accurately. To achieve this, I explored two potential solutions:

- Computer Vision Approach: Using OpenCV to process camera input and detect the black line.
- Sensor-Based Approach: Using simulated infrared sensors in ROS.

We opted for the *computer vision approach* due to its flexibility in real-world applications. I implemented an algorithm using *OpenCV in Python,* which:

- Captured images from the robot's camera.
- Applied image processing techniques (grayscale conversion, thresholding, edge detection).
- Used contour detection to track the black line and adjust the robot's movement accordingly.

0.0.2.2 Load and Unload Point Detection (QR Code Scanning)

To recognize loading and unloading points, I implemented a *QR code detection system* using *QReader library* for detecting and decoding QR codes.

The system was designed to extract location data from QR codes and send it to the navigation system to adjust the robot's path.

0.0.2.3 Communication Between ROS and Web Interface

Since ROS does not natively communicate with web technologies, I implemented a *real-time communication system* using a *WebSockets (Socket.IO)* to enable bidirectional data transfer and *ROSBridge* to translate ROS messages into JSON format for the web interface.

This setup allowed users to monitor the robot's status and send commands through a web-based interface built with *React.js.*

0.0.2.4 Application of Theoretical and Applied Knowledge

The work applied key theoretical and practical knowledge across multiple domains. Computer vision and image processing techniques were utilized for line detection and QR code recognition, ensuring accurate navigation and object identification. Robotics and control algorithms were implemented to facilitate smooth and efficient movement. Networking and web communication played a crucial role in establishing real-time data transfer between ROS and the web interface, enabling seamless interaction. Additionally, software engineering best practices were followed to structure ROS nodes effectively, optimize system performance, and debug potential issues, ensuring a well-organized and robust implementation.

Through this structured approach, I successfully developed the necessary programming solutions for the industrial robot, ensuring accurate navigation, QR code recognition, and seamless communication between the robot and the web interface.

0.0.3 User interface

The UI is structured into modular React components, each dedicated to displaying specific robot-related information. These components dynamically consume state from the *Zustand core*, ensuring real-time synchronization between the ROS updates and UI rendering. This modular approach enhances maintainability and flexibility, allowing for seamless updates to the user interface as data changes, while also ensuring the UI remains responsive and in sync with the backend processes.

The system enables real-time robot movement control with built-in process validation, ensuring smooth operation during task execution. It supports live camera and QR code feed display, providing real-time visual feedback. Dynamic map updates are implemented based

on the robot's movement and scanned locations, offering continuous situational awareness. Task execution and monitoring are integrated, including cargo loading and unloading, with clear updates on progress. Interactive gear control is featured, with process-state validation to prevent gear changes during active tasks, ensuring safe and efficient operation throughout the system.

0.0.3.1 Design Hierarchy

The application follows a structured GUI hierarchy where the *App* component serves as the central entry point, interacting with a *Core Store* for managing state. Within the App, the *MainLayout* organizes key sections, including a *NavBar*, a *Task* section, and a *Remote* module. The *Remote* module further breaks down into essential components such as *Camera, Map, RemoteButton,* and *Gear & Speed*, suggesting it handles functionalities related to remote control and navigation. This modular design ensures a clear separation of concerns, making the application easier to maintain and extend. The overall structure is visually represented in fig. 1 below.

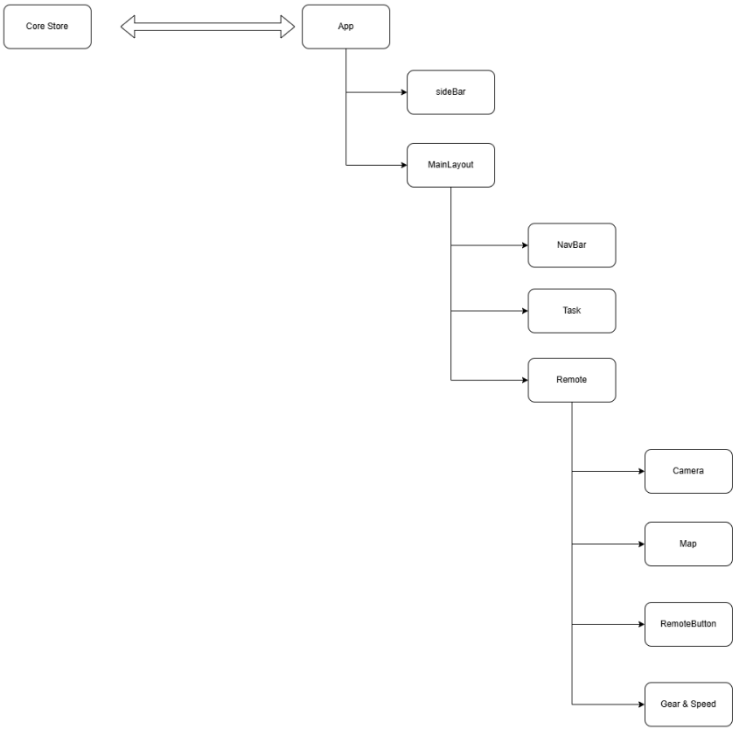

Figure 1: GUI hierarchy

0.0.3.2 Output

The graphical user interface of the robot, segmented into several functional areas for ease of operation:

1. *Menu*: On the left side, we have a straightforward menu with two options: ”Remote Control” and ”Task.” This menu allows users to navigate between different functionalities of the system easily (see fig. 2).

2. *Remote Control Tab*: This section is the heart of the interface and is further divided into three parts (see fig. 3):

    - *Main Camera View*: Dominating the top centre is a large grey box labelled ”Main Camera.” This is likely where the live feed from the camera appears, giving users a visual of the controlled environment.
    - *Control Panel*: Located below the camera view, this panel includes directional buttons (*up, down, left, right*) and a central button labelled ”S” for emergency stop There's also a toggle switch marked ”A” and ”M” and a speed indicator displaying ”0.00 m/s.” These controls are probably used to manoeuvre the remote vehicle or robot.
    - *Map*: On the right side, we see a white area with a grid of blue squares, a red and green marker, likely indicating the position of the robot or points of interest. There are two additional buttons at the bottom, a blue one with a ”+” sign and a red one with a ”-” sign, possibly for zooming in and out.

3. Task Tab: designed to manage tasks and monitor activities. The interface is divided into distinct sections for better functionality (fig. 4):

    - *Mapping Button*: This button is used to send the mapping task to the robot.
    - *Task Display*: Under the ”Mapping” section, there's a white text box labelled ”Task display,” which displays tasks and other relevant information.
    - *Main Display Area*: The majority of the screen is occupied by a large dark area that serves as the main display or workspace for the system's operations.
    - *Send Task Button*: At the bottom of the screen, this button allows users to send tasks to the robot.

This interface offers a clear, user-friendly design for effectively managing the robot.

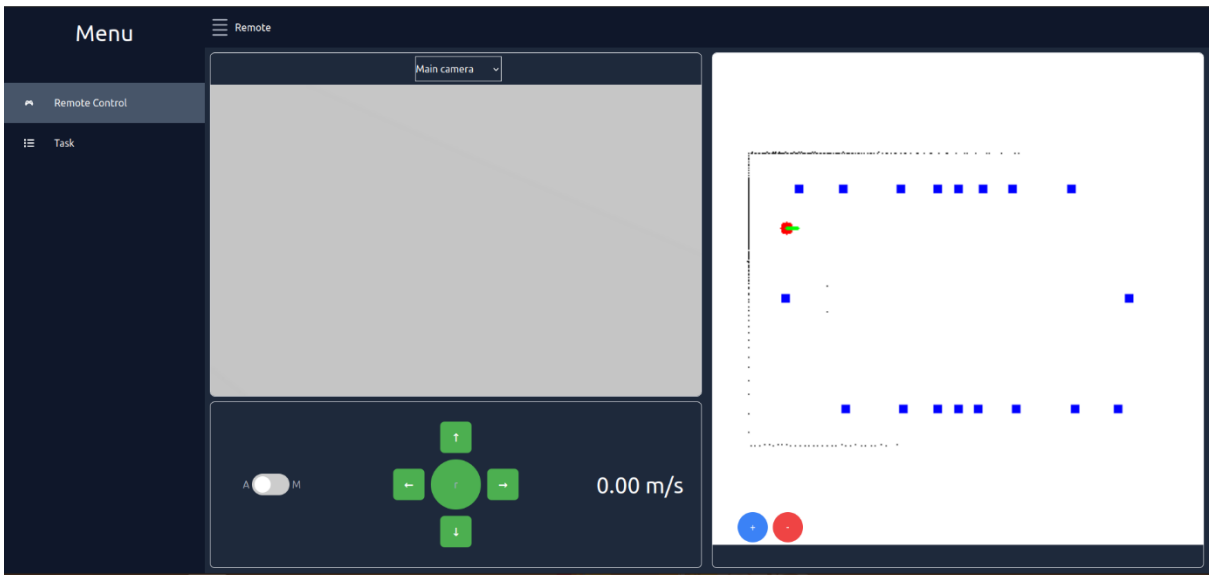

Figure 2: Opened menu & remote control Tab

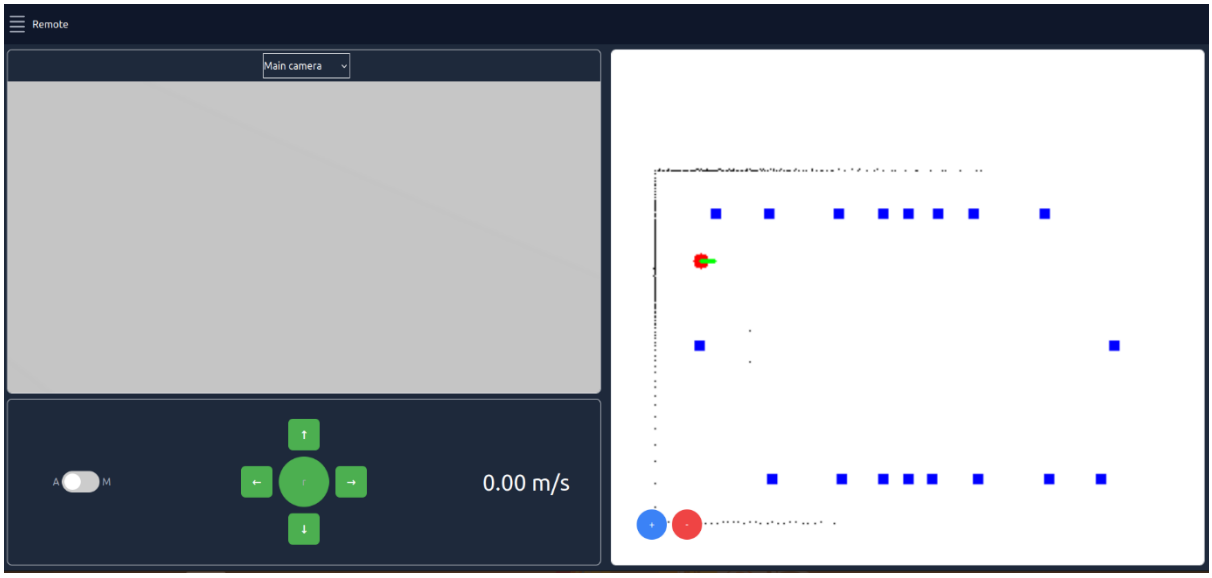

Figure 3: Remote control tab

Figure 4: Task Tab before mapping

0.0.3.3 Codes and background implementation

0.0.3.3.1 Core Store Implementation – The Most Important Element

a centralized state management system using *Zustand* to handle application state, WebSocket communication, and real-time data updates.

Key Responsibilities and Implementation Details:

State management was handled using *Zustand* by creating a global store (core, see fig. 1) to manage the application's state efficiently. This included tracking the connection status through *connectedSocket* and *connectedRos*, as well as managing the UI state with menu and *menuOpened*. Real-time data, such as *cameraData*, *camera_qr_data, mapData, speed, gear, zone,* and *station*, was also stored to ensure seamless updates. Additionally, the application's process state was maintained using *processState*. To facilitate state updates, setter functions like *setmenu* and *setmenuOpened* were implemented, allowing for specific property modifications.

Code Block 0.0.1: All states of core element with their default value

```
export const core = create((set,get) =>
({ connectedSocket : false,
    connectedRos : false,
    menu : "Remote",
    menuOpened : false,
    socket : null,
    cameraData : null,
```

```
    camera_qr_data : null,
    speed : 0,
    mapData : null,
    gear : 0,
    zone : null,
    station : null,}))
```

WebSocket integration was implemented using Socket.IO to enable real-time communication between the frontend and backend. The *initializeSocket* function was developed to establish a connection with the WebSocket server at `http://localhost:5000` (see fig. 2). It managed connection and disconnection events, ensuring the *connectedSocket* state was updated accordingly. Additionally, it listened for real-time data streams, including *camera_feed, camera_qr_feed, map_feed, speed, gear_state, task_response, mapping_output,* and *processState*, allowing for seamless data exchange and synchronization across the application.

Code Block 0.0.2: socket connection

```
const socket = io('http://localhost:5000');
set({ socket });
```

Real-time data handling was implemented to ensure seamless updates and synchronization across the application. Camera feeds were processed and stored as base64-encoded images in the state variables *cameraData* and *camera_qr_data*, enabling real-time visualization. The *mapData* state was continuously updated with real-time map images received from the backend. Additionally, the robot's speed and gear state were tracked through WebSocket events, updating the speed and gear properties accordingly. Task responses from the backend were displayed using react-hot-toast, providing instant user feedback and enhancing the overall user experience.

Code Block 0.0.3: socket connection

```
camera_feed : async function(data){
        set({cameraData : data.image})
    },
    camera_qr_feed : async function(data){
        set({camera_qr_data : data.image})
    },
    map_feed : async function(data){
```

```
        set({mapData : data.image})
    },
```

he frontend and backend. The *sendTask* function was implemented to transmit task sequences, such as loading and unloading, to the backend via WebSocket. Task data, including *task_name* and params, was structured and emitted through the *robot_task* event. For mapping, the mapping function was triggered to initiate the process by emitting a *robot_task* event with the necessary task data. Additionally, the *mapping_output* event was processed to serialize and store zone and station data in the state variables zone and station, ensuring efficient real-time updates.

Serialization of mapping data was handled through the serialize function (see fig. 4), which processed raw location data received from the backend. This function filtered and organized intersection and station data into structured formats, ensuring clarity and usability. The processed data was then stored in the state variables zone and station, enabling seamless integration with the UI for real-time visualization and interaction.

Code Block 0.0.4:

```
serialize : async function(locations){
      let lcs = [], st = []
      locations.forEach(location => {
        let nm = location.location.split("_")
        if(nm[0] == "intersection"){
          if (nm[2] == "x" || nm[2] == "y"){
          }else{
            let found = false
            for(let i = 0; i < lcs.length; i++){
              if (lcs[i].location == nm[1] + " " + nm[2]){
                found = true
              }
            }
            if (!found){
              lcs.push({
                location : nm[1] + " " + nm[2],
                x : location.x, y : location.y,
              })
            }
          }
        }else if (nm[0] == "station"){
```

```
        let found = false
          for(let i = 0; i < st.length; i++){
            if (st[i].location == nm[0] + " " + nm[1]){
              found = true
            }
          }
          if (!found){
            st.push({location : nm[0] + " " + nm[1],
            x : location.x, y : location.y,
          })
        }
      }
    });
    set({zone : lcs})
    set({station : st})
  },
```

Gear control was managed through the *changeGear* function, allowing the robot's gear state to toggle between *automatic* and *manual* modes. To ensure safe operation, gear changes were restricted while an active process was running *(processState === "Processing")*. Additionally, react-hot-toast was used to provide user feedback, notifying users when gear changes were blocked due to ongoing tasks.

Direction control was handled through the *sendDirection* function, which transmitted movement commands such as *UP, DOWN, LEFT*, and *RIGHT* to the backend via WebSocket. To maintain system integrity, movement commands were restricted during active processes *(processState === "Processing")*, preventing conflicts with ongoing tasks. User feedback was provided to ensure clarity when movement commands were blocked.

The outcome was the successful delivery of a scalable and efficient state management system that seamlessly integrates with WebSocket communication. This integration enabled real-time updates for critical data, including camera feeds, map data, speed, and gear state, significantly enhancing the application's interactivity and responsiveness. Additionally, the system provided a robust foundation for task management, mapping, and robot control, ensuring a smooth and intuitive user experience while maintaining high performance and reliability.

0.0.3.3.2 App Component

the root App component serves as the entry point for the application.

The application was enhanced with several key integrations to improve functionality and user experience. *React Router (BrowserRouter)* was integrated to enable client-side routing, allowing for seamless navigation between pages. Conditional rendering logic was added to display a loading spinner (Loader component) while waiting for the socket connection to be established, ensuring users are informed during the connection process. The layout was structured using flexbox for a responsive design, dividing the interface into a SideBar and MainLayout component for better organization. Additionally, React Hot Toast (Toaster) was integrated to provide user notifications and feedback, enhancing interactivity and responsiveness throughout the application.

The outcome:

The creation of a scalable and modular application structure, designed to support real-time communication capabilities seamlessly. By implementing loading states and real-time notifications, the application ensures a smooth user experience, keeping users informed and engaged during the connection process and throughout their interactions with the system. This structure not only enhances performance but also improves the overall user interface, making it responsive and intuitive. See ()code block 0.0.5) for implementation.

Code Block 0.0.5: App Component

```
function App() {
  const { initializeSocket, connectedSocket, connectedRos } = core();

  useEffect(() => {
    initializeSocket();
  }, [initializeSocket]);

  if (!connectedSocket) {
    return <Loarder />;
  }

  return (
    <BrowserRouter>
      <div className="flex flex-row w-full h-full">
        <SideBar />
        <MainLayout />
      </div>
      <Toaster />
```

```
    </BrowserRouter>
  );
}

export default App;
```

0.0.3.3.3 MainLayout Component

This component manages the main content area and routing logic. See code block 0.0.6 for implementation.

Key Responsibilities:

The application was structured with React Router using Routes and Route to define navigation paths. A default route (/) was set to render the Remote component, while additional routes were configured for the /remote path to display the Remote component and the /task path for the Task component. Conditional rendering was implemented to display a loading spinner (Loader component) when the socket connection is not yet established, ensuring a smooth user experience during the initial connection phase. The layout was designed to be responsive using CSS, allowing the content area to adjust dynamically based on the viewport size. Additionally, a *NavBar* component was integrated at the top of the layout to provide consistent navigation across the application, enhancing usability and accessibility.

Outcome:

Was the creation of a centralized and reusable layout structure for the application, promoting maintainability and scalability. By implementing proper loading states and routing, seamless navigation was ensured, allowing users to transition between different sections smoothly. This structure enhanced the overall user experience, providing consistent and responsive design elements across the application, while also ensuring that users are kept informed during the socket connection process.

Code Block 0.0.6: MainLayout Component

```
const MainLayout = () => {
    const { connectedSocket, menuOpened } = core();
    return (
    <div className="w-full h-full overflow-hidden">
        <NavBar />
        <div className="w-full h-[calc(100%-64px)]">
        <Routes>
            <Route path="/" element={connectedSocket ? <Remote /> :
            ↪  <Loader />} />
            <Route
            path="/remote"
            element={connectedSocket ? <Remote /> : <Loader />}
            />
            <Route path="/task" element={connectedSocket ? <Task /> :
            ↪  <Loader />} />
        </Routes>
        </div>
    </div>
    );
};
export default MainLayout;
```

0.0.3.3.4 Remote Component

Remote component serves as the central interface for remote control functionality.

The application features a dual-panel layout implemented using *CSS Flexbox*, enhancing responsiveness and user experience. The left panel integrates a live camera feed alongside a control interface equipped with directional buttons (*RemoteButton*) for navigation. A gear-switching mechanism is incorporated, utilizing a toggle switch to alternate between *automatic (A)* and *manual (M)* modes. This functionality leverages *React's useEffect* and *useRef hooks* for efficient state management. Throughout, responsive and dynamic UI updates are ensured, adapting seamlessly to user interactions and state changes. See code below for implementation.

Code Block 0.0.7:

```
const Remote = (props) => {
    const {sendDirection, changeGear, gear} = core()
    const gearref = React.createRef()
```

```
    const onReset = function(e){ sendDirection("STOP") }
    const switchGear = function(e){
        e.preventDefault()
        changeGear()
    }
    useEffect(() => { gearref.current.checked = gear == 1}, [gear])
return (
<div className=
 ↪  "w-full h-full overflow-hidden flex flex-row bg-slate-800">
  <section className="w-1/2 p-2">
    <div className="w-full h-2/3 flex flex-col items-center
    border rounded-lg overflow-hidden">
      <Camera/>
    </div>
    <div className="w-full h-[calc((100%/3)-8px)]
    border rounded-lg mt-2 flex flex-row">
      <div className=" w-1/4 h-full flex items-center justify-center">
        <label className="switch-label mr-1">A</label>
          <label className="switch">
            <input type="checkbox" name="switch"
            id = "switch" ref={gearref} value={gear == 1}
             ↪  onClick={switchGear}/>
              <span className="slider round"></span>
          </label>
        <label className="switch-label ml-1">M</label>
      </div>
      <div className=
       ↪  "circle-section flex items-center justify-center  w-2/4 h-full"
       ↪  >
        <div className="circle-container relative ">
          <button className="center-circle" onClick =
           ↪  {onReset}>r</button>
            <RemoteButton direction="UP"/>
            <RemoteButton direction="DOWN"/>
            <RemoteButton direction="LEFT"/>
            <RemoteButton direction="RIGHT"/>
        </div>
      </div>
      <div className=" w-1/4 h-full"><Speed/></div>
    </div>
```

```
  </section>
  <section className="w-1/2 flex items-center justify-center p-2">
    <div  className="w-full h-full border rounded-lg flex flex-col
    items-center relative overflow-hidden">
      <Map ></Map>
    </div>
            </section>
    </div>
    );
};
export default Remote;
```

0.0.3.3.5 Camera Component

The Camera component displays live camera feeds and enable camera switching functionality.

A dynamic camera feed display was implemented by processing base64-encoded image data from the backend, supporting both the Main Camera and QR Code Camera. *React's useEffect* was utilized to ensure the image source (*imgRef*) updated whenever new camera data (*cameraData* or *camera_qr_data*) was received (see code block 0.0.8). A camera selection dropdown was added to enhance user interaction, allowing users to switch between the main camera and QR code camera views. Memory management was efficiently handled by revoking object URLs when the component unmounted or the camera data changed. The component was styled using CSS to ensure the camera feed was responsive and fit seamlessly within the layout.

Code Block 0.0.8: UseEffect which update camera image each render

```
useEffect(function(){
        if(cam === "front"){
            if (cameraData) {
                const imageUrl = `${cameraData}`
                imgRef.current.src = imageUrl;

                // Clean up the object URL when component is unmounted
                return () => {
                    URL.revokeObjectURL(imageUrl);
                };
            }
```

```
        }
        else if (cam === "back"){
            if (camera_qr_data) {
                const imageUrl =
                ↪  `${camera_qr_data}`
                imgRef.current.src = imageUrl;

                // Clean up the object URL when component is unmounted
                return () => {
                    URL.revokeObjectURL(imageUrl);
                };
            }
        }
    },[cameraData, camera_qr_data])
```

0.0.3.3.6 Map Component

This component displays and interact with a dynamic map interface. included integrating a base64-encoded map image (*mapData*) to display real-time map data. Zoom functionality was implemented using buttons for zooming in (+) and out (-) (see code below), as well as mouse wheel support for smooth zooming. A loading state was added to display a placeholder message while the map data was being fetched. Using *useEffect*, the map image was dynamically updated whenever new *mapData* was received (see code below), ensuring real-time updates. CSS transformations (scale) were applied to enable smooth zooming transitions, and floating zoom buttons were designed for an intuitive user experience, styled with CSS for a modern look.

Code Block 0.0.9:

```
    const zoomIn = () => {
        setZoom((prevZoom) => Math.min(prevZoom + 0.1, 7)); // Max zoom
        ↪  7x
    };

    const zoomOut = () => {
        setZoom((prevZoom) => Math.max(prevZoom - 0.1, 1)); // Min zoom
        ↪  1x (original size)
    };

    const handleWheel = (event) => {
```

```
        // Prevent the page from scrolling when zooming
        event.preventDefault();

        if (event.deltaY < 0) {
            zoomIn(); // Zoom in when scrolling up
        } else {
            zoomOut(); // Zoom out when scrolling down
        }
    };

    useEffect(() => {
        if (mapData) {
            const imageUrl = `${mapData}`;

            // Set image src only if imgRef.current is available
            imgRef.current.src = imageUrl;
            if(!mapData){
                setLoading(true);
            }
            setLoading(false)
            return () => {
                URL.revokeObjectURL(imageUrl);
            };
        }
    }, [mapData]);
```

0.0.3.3.7 Task Component

the Task component manages task creation and submission for the application. It is responsible for managing task creation and submission for the application. The responsibilities included designing and implementing a task creation interface that allowed users to select zones and define tasks, such as loading or unloading. The component dynamically built and displayed task sequences based on user input and integrated validation logic (see code below) to ensure task sequences adhered to predefined rules. For example, it prevented invalid task combinations like unloading without loading first and limited the number of tasks to a maximum of six. React hooks (*useState, useEffect, useCallback, useRef*) were utilized for state management and user interaction handling. Toast notifications (react-hot-toast) were integrated to provide real-time feedback for errors and successful actions. A mapping button

was implemented to trigger mapping functionality and dynamically display available zones. The component was styled using CSS and Tailwind CSS to achieve a clean and responsive design, ensuring a smooth and intuitive user experience while efficiently managing tasks within the application.

Code Block 0.0.10:

```
const set = () => {
        if (params.length === 6) return
         ↪  toast.error("You can't load and unload more than 6 times");
        if (selected === "") return toast.error("Select a zone");
        let type = ref.current.value === "Loading" ? "L" : "U";
        if (type === "U") {
            let param = params[params.length - 1];
            if (param) {
                let key = Object.keys(param)[0];
                if (param[key] !== "Loading") {
                    return toast.error(
                     ↪  "You can't unload without loading");
                } else if (key === selected) {
                    return toast.error(
                     ↪  "You can't unload from a zone you've loaded from"
                     ↪  );
                }
            } else { return toast.error("You need to load first");}
        }else if( type === "L"){
            let param = params[params.length - 1];
            if (param) {
                let key = Object.keys(param)[0];
                if (param[key] === "Loading") {
                    return toast.error("You can't load twice");
                }
            }
        }
        setTask(`${task}${selected}(${type}) -> `);
        setParams([...params, { [selected]: ref.current.value }]);
        setSelected("");
    };
```

0.0.4 Robot control framework

The Robot Operating System (ROS) serves as the robot's main controller, managing all operations seamlessly at all times and facilitating communication with the GUI.

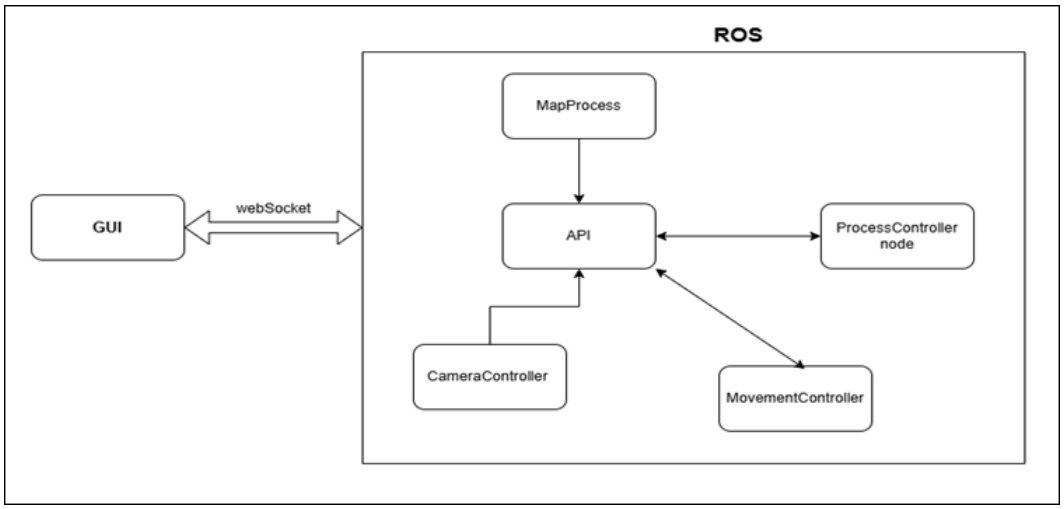

Figure 5: general architecture

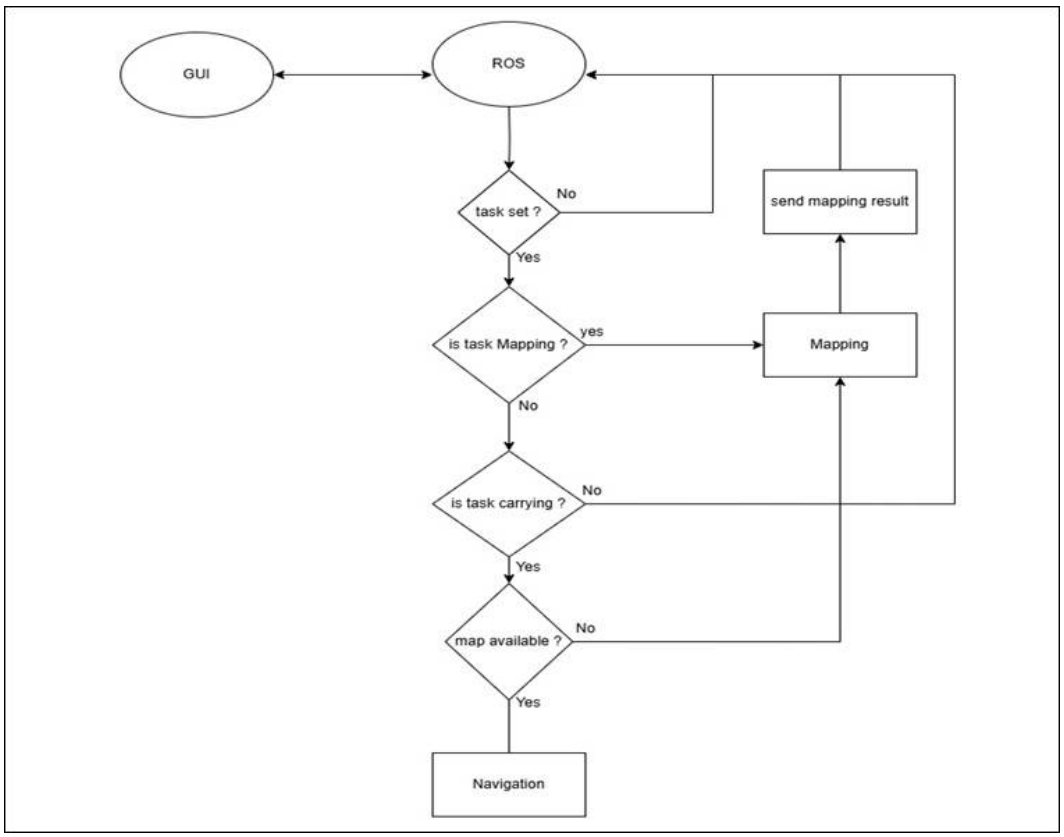

Figure 6: general architecture

0.0.4.1 Nodes

0.0.4.1.1 ie_API_Server

This node serves as a critical bridge between the ROS (Robot Operating System) ecosystem and a frontend application, leveraging *WebSocket* communication to enable seamless interaction. It facilitates real-time data streaming, robot control, and task management, ensuring smooth and efficient communication between the backend and frontend (see fig. 5).

Code Block 0.0.11: API initial value

```
class ie_API_Server:
  def __init__(self):
      self._bridge = CvBridge()
      self.sio = socketio.async_server.AsyncServer(async_mode='asgi',
      ↪  cors_allowed_origins="*")
      self.app = socketio.asgi.ASGIApp(self.sio)
      self.mc_pub = rospy.Publisher("manual_controller", Int32,
      ↪  queue_size=10)
      self.cam_sub = rospy.Subscriber("camera_feed", Image,
      ↪  self.run_async_cameraFeedCallback)
      self.cam_qr_sub = rospy.Subscriber("camera_qr_code_feed",
      ↪  Image, self.run_async_cameraQrFeedCallback)
      self.sensors_sub = rospy.Subscriber("sensor_data",
      ↪  SensorDataMap, self.sensorsCallback)
      self.speed_sub = rospy.Subscriber("speed_value", Float32,
      ↪  self.run_async_speedCallback)
      self.map_sub = rospy.Subscriber("map_feed", Image,
      ↪  self.run_async_mapFeedCallback)
      self.process_sub = rospy.Subscriber("ProcessState", String,
      ↪  self.run_async_processState)
      rospy.Service('output', taskMessage,
      ↪  self.run_async_taskFinished)
      rospy.Service('mapping_output', mappingOutput,
      ↪  self.run_async_mappingOutput)
      rospy.Service('gear_changed', robotGear,
      ↪  self.run_async_gearChanged)
      self.sio.on("connect", self.onConnect)
      self.sio.on("disconnect", self.onDisconnect)
      self.sio.on("moveDirection", self.movement)
      self.sio.on("message", self.message)
      self.sio.on("robot_task", self.taskCallback)
```

```
        self.sio.on("robot_gear", self.gearCallback)
```

The node integrates *WebSocket communication* using *Socket.IO*, establishing a real-time, bidirectional channel that supports multiple clients. This allows users to connect, disconnect, and interact with the robot in real time. The WebSocket connection is the backbone of the system, enabling instant data exchange and control commands between the frontend and the ROS backend.

Code Block 0.0.12: API initial value

```
  # Register event handlers
self.sio.on("connect", self.onConnect)
self.sio.on("disconnect", self.onDisconnect)
self.sio.on("moveDirection", self.movement)
self.sio.on("message", self.message)
self.sio.on("robot_task", self.taskCallback)
self.sio.on("robot_gear", self.gearCallback)
```

For *real-time data streaming*, the node subscribes to various ROS topics to capture and process critical information. It subscribes to *camera_feed* and *camera_qr_code_feed* to receive live camera images, which are then converted from ROS Image messages to OpenCV format using *CvBridge*. These images are encoded as *base64 strings* and streamed to the frontend via WebSocket. Similarly, the node subscribes to the *map_feed* topic to receive real-time map images, processes them, and streams them to the frontend in the same manner. Additionally, it subscribes to the *speed_value* topic to capture the robot's speed and emits speed updates to the frontend in real time, ensuring users have up-to-date information on the robot's movement.

The node also handles *robot control* by listening for specific events from the frontend. For movement control, it listens for *moveDirection* events (e.g., *UP, DOWN, LEFT, RIGHT, STOP*) and translates these into corresponding commands published to the *manual_controller* ROS topic, enabling direct control of the robot's movement. For gear control, it listens for *robot_gear* events to toggle between automatic and manual modes. This is achieved by calling the *change_gear* ROS service and emitting the updated gear state to the frontend, ensuring the user interface reflects the current state of the robot.

Task management is another key functionality of the node. It listens for *robot_task* events from the frontend, which trigger tasks such as loading, unloading, or mapping. The node converts task data into a ROS-compatible format (using the *TaskData* message) and calls the *robot_task* service to execute the task(see code block 0.0.12). It also listens for task feedback from the ROS backend via the *output* service, emitting task responses (e.g., success or

failure messages) to the frontend. This ensures users receive timely updates on task progress and outcomes, enhancing transparency and usability.

For *mapping functionality*, the node listens for mapping data from the ROS backend via the *mapping_output* service. It processes and serializes this data, which includes information such as locations and coordinates, and emits it to the frontend. This allows users to visualize the robot's environment and track its movements in real time.

Finally, the node monitors the robot's *process state* by subscribing to the *ProcessState* topic. It captures updates on the robot's current state (e.g., Processing, Stationary) and emits these updates to the frontend in real time. This ensures users are always aware of the robot's operational status, enabling better decision-making and control.

Code Block 0.0.13:

```
    async def taskCallback(self, sid, task):
      t = TaskData()
      t.task_name = task['task']["task_name"]
      t.params = [
          Param(zone=key, type=value)
          for key, value in task['task']["params"].items()
      ]
      rospy.wait_for_service('robot_task')
      robot_task = rospy.ServiceProxy('robot_task', robotTask)
      robot_task.wait_for_service(10)
      try:
          response = robot_task(t)
          print(response)
          await self.sio.emit(
              "task_response",
              {"response": response.message},
              to=sid
          )
      except rospy.ServiceException as exc:
          print("Service did not process request: " + str(exc))

  def run_async_taskFinished(self, output):
      try:
          loop = asyncio.new_event_loop()
          asyncio.set_event_loop(loop)
          loop.run_until_complete(self.taskFinished(output))
      except Exception as e:
```

```
            rospy.logerr(f
             ↪ "Error before proccessing and emitting taskFinished: {e}")
        return taskMessageResponse(True)

    async def taskFinished(self, output):
        await self.sio.emit("task_response", {"response": output.message})

    def run_async_mappingOutput(self, output):
        try:
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)
            loop.run_until_complete(self.mappingOutput(output))
        except Exception as e:
            rospy.logerr(f
             ↪ "Error before proccessing and emitting taskFinished: {e}")
        return mappingOutputResponse(True)

    async def mappingOutput(self, output):
        print(output)
        data = [
            {"location": kv.location, "x": kv.x, "y": kv.y}
            for kv in output.locations
        ]
        print(data)
        await self.sio.emit("mapping_output", {"locations": data})
```

0.0.4.1.2 movementController

This node manages the robot's movement in both *manual* and *autonomous* modes. It processes control commands, handles gear switching, and ensures smooth transitions between states.

Key Features and Functionality:

The node supports two modes of operation:

- *Manual Mode*: The robot is controlled via user commands, such as increasing or decreasing linear or angular velocity.
- *Autonomous Mode*: The robot is controlled by an external system, like a navigation stack, with manual control disabled.

To facilitate seamless switching between these modes, the *twist_mux* package can be utilized. This package subscribes to multiple *cmd_vel* topics.

The Movement Control component was designed to subscribe to the *manual_controller* topic to receive movement commands, such as *UP, DOWN, LEFT, RIGHT,* and *STOP*. The component adjusted the robot's linear and angular velocity based on the received commands and then published the resulting velocity commands to the *cmd_vel* topic.

Gear Switching system provides a ROS service (*change_gear*) to switch between manual and autonomous modes, updates the gear state and notifies the GUI via another ROS service (*gear_changed*) (see fig. 5). Velocity Interpolation Implements smooth interpolation for angular velocity to prevent abrupt changes in robot movement. Uses a waiting period and linear interpolation to gradually reduce angular velocity to zero.

Process State Monitoring Subscribes to the *ProcessState* topic to monitor the robot's current state (e.g., Processing, Stationary). Resets movement commands if the robot is in a Processing state. Speed Calculation Calculates the robot's total velocity by combining linear and angular velocity components. Publishes the calculated speed to the *speed_value* topic for real-time monitoring.

Technical Details:

- ROS Integration:
  - Publishers:
    - cmd_vel: Publishes velocity commands for the robot.
    - speed_value: Publishes the robot's current speed.
  - Subscribers:
    - manual_controller: Receives movement commands.
    - ProcessState: Monitors the robot's state.
    - cmd_vel: Receives velocity commands from the autonomous system.
  - Services:
    - change_gear: Handles gear switching between manual and autonomous modes.
- Data Structures:
  - cmd_vel Struct: Stores the robot's linear and angular velocity (see code block 0.0.12) components.Manages interpolation and waiting states for smooth velocity transitions.Provides methods to increase/decrease velocity and reset commands.

- Interpolation Logic: Uses timestamps (ros::Time) to track the last update time and interpolation duration. Implements a waiting period (wait_duration) before starting interpolation. Linearly interpolates angular velocity towards zero over a specified duration (interpolation_duration), see code block 0.0.11 for implementation.
- Velocity Calculation: Combines linear and angular velocity components to calculate the robot's total velocity. Accounts for the robot's wheel radius (radius) to convert angular velocity to linear velocity.
- Error Handling: Logs warnings for invalid commands or attempts to control the robot in autonomous mode. Ensures smooth transitions between states to prevent abrupt movements.

Code Block 0.0.14: Gear shift

```
bool changeGear (ie_communication::robotGear::Request &req,
 ↪  ie_communication::robotGear::Response &res){
  movedata.gear = req.state;
  if(movedata.gear == 0){
      std::cout <<
       ↪  "The robot is in autonomous mode, manual control is disabled"
       ↪   <<
       ↪  std::endl;
  }else{
      std::cout <<
       ↪  "The robot is in manual mode, autonomous control is disabled"
       ↪   <<
       ↪  std::endl;
  }

  ros::NodeHandle n;
  ros::ServiceClient client =
   ↪  n.serviceClient<ie_communication::robotGear>("gear_changed");
  ie_communication::robotGear srv;

  srv.request.state = movedata.gear;

  if(client.call(srv)){

  }else{
      ROS_ERROR("Failed to update the gear to GUI");
```

```
    }

    res.message = true;
    return true;
}
```

Code Block 0.0.15: Linear interpolation

```
  void updateAngular() {
        ros::Time current_time = ros::Time::now();
        ros::Duration time_since_update = current_time -
         ↪  last_update_time;

        if (waiting_to_interpolate) {
            if (time_since_update.toSec() >= wait_duration) {
                // Start interpolation after waiting period
                angular_start = az;
                interpolation_start_time = current_time;
                angular_interpolating = true;
                waiting_to_interpolate = false;
            }
        }

        if (angular_interpolating) {
            ros::Duration time_since_start = current_time -
             ↪  interpolation_start_time;
            float t = time_since_start.toSec() /
             ↪  interpolation_duration;

            if (t >= 1.0) {
                az = 0.0; // Stop interpolation after duration
                angular_interpolating = false;
            } else {
                az = angular_start * (1.0 - t); // Linearly interpolate
                 ↪  towards zero
            }
        }
    }
```

Code Block 0.0.16: Cmd_vel structure

```
struct cmd_vel {
  int gear = 0;
  float lx = 0.0;
  float ly = 0.0;
  float lz = 0.0;
  float ax = 0.0;
  float ay = 0.0;
  float az = 0.0;

  ros::Time last_update_time; // Last time angular velocity was
  ↪ updated
  ros::Time interpolation_start_time; // Time when interpolation
  ↪ starts
  bool angular_interpolating = false; // Flag to indicate
  ↪ interpolation
  bool waiting_to_interpolate = false; // Flag to indicate waiting
  ↪ period
  float angular_start = 0.0; // Starting value for interpolation
  float interpolation_duration = 0.8; // Duration for interpolation
  ↪ (in seconds)
  float wait_duration = 0.3; // Time to wait before starting
  ↪ interpolation (in seconds)
  float radius = 0.16;
  void changeGear(const std_msgs::Int32::ConstPtr& msg) {
      gear = msg->data;
  }
  void increaseLinear() {
      lx += 0.01;
  }

  void decreaseLinear() {
      lx -= 0.01;
  }

  void increaseAngular() {
      az += 0.1;
      updateAngularState();
  }
```

```
    void decreaseAngular() {
        az -= 0.1;
        updateAngularState();
    }

    void reset() {
        lx = 0.0;
        ly = 0.0;
        lz = 0.0;

        ax = 0.0;
        ay = 0.0;
        az = 0.0;

        angular_interpolating = false;
        waiting_to_interpolate = false;
    }

    void updateAngularState() {
        last_update_time = ros::Time::now();
        angular_interpolating = false; // Stop interpolation if active
        waiting_to_interpolate = true; // Set waiting state
    }


    float getVelocity(){
        if (gear == 1){
            float linear_velocity_rotational = az * radius;
            float tVelocity = sqrt((pow(lx, 2) +
             ↪  pow(linear_velocity_rotational, 2)));
            return std::round(tVelocity * 100.0f) / 100.0f;
        }else{
            float linear_velocity_rotational = acvl.angular.z * radius;
            float tVelocity = sqrt((pow(acvl.linear.x, 2) +
             ↪  pow(linear_velocity_rotational, 2)));
            return std::round(tVelocity * 100.0f) / 100.0f;
        }
    }
};
```

0.0.4.1.3 CameraController

This node functions as a *camera feed manager*, playing a crucial role in subscribing to raw camera feeds, processing them, and publishing the processed feeds to other ROS topics for further use, such as display or analysis. It subscribes to two raw camera feeds: */camera/rgb/image_raw*, which is the primary camera feed used for general purposes like navigation or object detection, and */camera_2/camera_2/image_raw*, the secondary camera feed often utilized for specialized tasks such as QR code detection. The node then publishes the processed feeds to two ROS topics: *camera_feed* for the primary camera feed and *camera_qr_code_feed* for the secondary camera feed. This ensures that downstream nodes or applications have access to the necessary camera data for their respective tasks, see (code block 0.0.16) for implementation.

The node also incorporates *camera state management* to ensure efficient operation. It uses a *boolean flag (_camState)* to enable or disable feed publishing, ensuring that feeds are only published when the camera is active and data is available. Additionally, it tracks camera availability *(_cam_available*) to prevent publishing when no data is being received. This state management mechanism helps optimize resource usage and prevents unnecessary processing. To handle potential issues, the node implements *robust error handling* for image conversion and publishing. Errors, such as failed image conversions or publishing failures, are logged using *rospy.logerr*, making it easier to debug and maintain the system.

From a technical perspective, the node integrates seamlessly with the ROS ecosystem. It subscribes to */camera/rgb/image_raw* and */camera_2/camera_2/image_raw* to receive raw images from the primary and secondary cameras, respectively. It then publishes the processed images to *camera_feed* and *camera_qr_code_feed* for further use. For image processing, the node uses *CvBridge* to convert ROS Image messages into OpenCV-compatible formats. While the node processes and publishes the images, it does not apply additional filtering or transformations, ensuring the images remain in their original state unless modified by downstream nodes.

To ensure smooth operation and responsiveness, the node is initialized in a *separate thread*. This threading approach allows the node to handle image processing and publishing without blocking the main thread, ensuring efficient performance even under high workloads. By combining ROS integration, state management, error handling, and threading, this camera feed manager node provides a reliable and efficient solution for managing and streaming real-time camera data within the ROS ecosystem. Its design ensures that downstream applications receive the necessary camera feeds promptly and accurately, making it an essential component for systems that rely on real-time visual data.

Code Block 0.0.17: CameraProcess node

```
import rospy
from ie_communication.srv import camState, camStateResponse
import cv2
from cv_bridge import CvBridge
from sensor_msgs.msg import Image

class CameraController:

    def __init__(self):
        self._pubcam1 = rospy.Publisher("camera_feed", Image,
        ↪  queue_size=10)
        self._pubcam2 = rospy.Publisher("camera_qr_code_feed", Image,
        ↪  queue_size=10)
        rospy.Subscriber("/camera/rgb/image_raw",Image,
        ↪  self._camera1Process)
        rospy.Subscriber("/camera_2/camera_2/image_raw",Image,
        ↪  self._camera2Process)
        self._bridge = CvBridge()
        self._camState = True
        self._cam_available = False
        self._cam1Frame = None
        self._cam2Frame = None

    def _camera1Process(self, data):
        try:
            self._cam1Frame = data
            self._cam_available = True
            self._provideCamFeed()
        except Exception as e:
            rospy.logerr(f"Error converting image: {e}")

    def _camera2Process(self, data):
        try:
            self._cam2Frame = data
            self._cam_available = True
            self._provideCam2Feed()
        except Exception as e:
            rospy.logerr(f"Error converting image: {e}")
```

```
    def _provideCamFeed(self) -> None:
        if self._camState and self._cam_available:
            try:
                self._pubcam1.publish(self._cam1Frame)
            except Exception as e:
                rospy.logerr(f"Error publishing image feed: {e}")

    def _provideCam2Feed(self) -> None:
        try:
            self._pubcam2.publish(self._cam2Frame)
        except Exception as e:
            rospy.logerr(f"Error publishing image from cam2 feed: {e}")
```

0.0.4.1.4 map_process

This node is responsible for processing laser scan data, robot position, and QR code positions to generate and publish a dynamic map for visualization and navigation. It subscribes to the */scan* topic to receive laser scan data, which is processed to detect obstacles and update the local map. The robot's position and orientation are tracked using *TF2*, which retrieves the robot's translation and rotation in the map frame. These coordinates are then converted into pixel coordinates for visualization on the map. Additionally, the node integrates QR code positions by reading them from an SQLite database (*qr_code.db*) every 5 seconds. These QR code positions are displayed on the map as blue rectangles, providing a comprehensive view of the environment.

Code Block 0.0.18: Update Map from Scan Function

```
  def update_map_from_scan(self):
    if self.robot_position is None or self.robot_rotation is None:
       return
    yaw = self.get_yaw_from_quaternion(self.robot_rotation)

    for i, range_val in enumerate(self.latest_scan.ranges):
        if range_val < self.latest_scan.range_min or range_val >
        ↪  self.latest_scan.range_max:
            continue  # Skip invalid range values

  angle=self.latest_scan.angle_min+i* self.latest_scan.angle_increment
```

```
        endpoint_x = self.robot_position.x + range_val * np.cos(yaw +
        ↪  angle)
        endpoint_y = self.robot_position.y + range_val * np.sin(yaw +
        ↪  angle)

        endpoint_x_pixel = int((endpoint_x - self.map_origin_x) /
        ↪  self.map_resolution)

        endpoint_y_pixel = int((endpoint_y - self.map_origin_y) /
        ↪  self.map_resolution)

        if 0 <= endpoint_x_pixel < self.map_width and 0 <=
        ↪  endpoint_y_pixel < self.map_height:

                self.map_image[endpoint_y_pixel, endpoint_x_pixel] = 0
```

The node generates a *2D map image* that includes several key elements: obstacles derived from laser scan data (represented as black pixels), the robot's current position (shown as a red circle) and orientation (indicated by a green line), QR code positions (displayed as blue rectangles), and the robot's path (traced as a green line over time). This map image is published to the *map_feed* topic, enabling real-time visualization for users or other nodes. The map is dynamically updated at a fixed rate of 10 Hz using a timer callback, ensuring it reflects the latest scan data, robot position, and QR code positions. The map parameters, such as resolution (5 cm per pixel), dimensions (*400x400 pixels*), and origin *(-10.0, -10.0 meters*), are predefined, and the map is initialized as free space (white pixels) before being updated with obstacles (black pixels) from laser scans.

In summary, this node provides a dynamic and real-time map visualization by integrating laser scan data, robot position, and QR code positions. Its robust processing and integration capabilities ensure accurate and up-to-date map generation, making it a vital component for navigation and visualization tasks in the ROS ecosystem.

Code Block 0.0.19: Function Drawing Map

```
    def process_scan(self):
        self.map_image.fill(255)

        robot_x_pixel = int((self.robot_position.x -
        ↪  self.map_origin_x) / self.map_resolution)
```

```
        robot_y_pixel = int((self.robot_position.y -
        ↪  self.map_origin_y) / self.map_resolution)

        if not (0 <= robot_x_pixel < self.map_width and 0 <=
        ↪  robot_y_pixel < self.map_height):
            return
      self.update_map_from_scan()
      map_image_color=cv2.cvtColor(self.map_image, cv2.COLOR_GRAY2BGR)

      if len(self.path_history) > 1:
          for i in range(1, len(self.path_history)):
                cv2.line(map_image_color, self.path_history[i - 1],
                ↪  self.path_history[i], (0, 255, 0), 2)

      for qr_x, qr_y in self.qr_code_positions:
          qr_x_pixel=int((qr_x-self.map_origin_x)/ self.map_resolution)
          qr_y_pixel=int((qr_y-self.map_origin_y)/ self.map_resolution)
            if 0 <= qr_x_pixel < self.map_width and 0 <= qr_y_pixel <
            ↪  self.map_height:
                cv2.rectangle(map_image_color,
(qr_x_pixel - self.qr_code_size, qr_y_pixel -
↪  self.qr_code_size),(qr_x_pixel + self.qr_code_size, qr_y_pixel +
↪  self.qr_code_size),(255, 0, 0), -1)

        cv2.circle(map_image_color, (robot_x_pixel, robot_y_pixel), 5,
        ↪  (0, 0, 255), -1)
        yaw = self.get_yaw_from_quaternion(self.robot_rotation)
        robot_x_end = robot_x_pixel + int(np.cos(yaw) * 10)
        robot_y_end = robot_y_pixel + int(np.sin(yaw) * 10)
        cv2.line(map_image_color, (robot_x_pixel, robot_y_pixel),
        ↪  (robot_x_end, robot_y_end), (0, 255, 0), 2)

        map_image_color = np.flipud(map_image_color)
        map_image_color = cv2.rotate(map_image_color,
        ↪  cv2.ROTATE_90_COUNTERCLOCKWISE)
        output_path = "/tmp/robot_map_with_position.png"
        cv2.imwrite(output_path, map_image_color)
        self.send_image(output_path)
```

0.0.4.2 Classes

0.0.4.2.1 Task

The Task class is a base class that encapsulates common functionality for robot tasks, such as: Robot *control* (movement, turning, stopping), *Obstacle detection and avoidance* using LIDAR data, *Image processing* for line following and QR code detection, *State management* for task execution and transitions.

Key Features and Functionality:

The robot control system offers a comprehensive set of methods for managing basic robot movements. These include *_move_forward*, which propels the robot forward at a constant speed, as well *as _turn_left* and *_turn_right* for executing left and right turns, respectively. Additionally, the system features a *_U_turn* method for performing a 180° turn and a *_move* function that adjusts the robot's movement based on error, particularly useful for tasks like line following. To halt the robot, the *stop* method is available.

For obstacle detection and avoidance, the system subscribes to the */scan* topic to receive LIDAR data, enabling it to monitor the surroundings. It employs the *check_for_obstacles* function (see code block 0.0.17) to detect obstacles in the robot's path. When an obstacle is detected, the system utilizes a contouring algorithm called *contour_obstacle(see* code block 0.0.17*)* to navigate around it. This algorithm involves turning 90° left, moving forward, turning 90° right, and then returning to the original path. To ensure precise turns, the system tracks the robot's orientation using odometry data, accessed through the *_get_yaw* method. This combination of movement control and obstacle avoidance ensures efficient and accurate navigation in dynamic environments.

Code Block 0.0.20: Obstacle avoidance procedure

```
def contour_obstacle(self):
      self._contourningStep += 1
      if self.timer:
          self.timer.shutdown()
      self.stop()  # Stop the robot

      if self._contourningStep == 1:
          rospy.loginfo("Contouring obstacle: Step 1")
          self.turn_angle(90)
          self.contour_obstacle()
      elif self._contourningStep == 2:
          rospy.loginfo("Contouring obstacle: Step 2")
          self._move_forward()
```

```
            while not self.is_obstacle_behind():
                rospy.sleep(0.1)
            self.turn_angle(-90)  # Turn right 90°
            self.contour_obstacle()
        elif self._contourningStep == 3:
            rospy.loginfo("Contouring obstacle: Step 3")
            self._move_forward()
            while not self.is_obstacle_behind():
                rospy.sleep(0.1)
            self.turn_angle(-90)
            self.contour_obstacle()
        elif self._contourningStep == 4:
            rospy.loginfo("Contouring obstacle: Step 4")
            self._move_forward()
            self._turnSide = "left"
            self.moveToTurnPosition = True
            self._obstacleInFront = False
```

Code Block 0.0.21: Function checking if the robot overpass the obstacle

```
    def is_obstacle_behind(self):
        if self.scan_data is None:
            return False
        robot_length = 0.52
        obstacle_distance_threshold = robot_
        sector_start_angle = np.deg2rad(225)  # 225 degrees
        sector_end_angle = np.deg2rad(270)   # 270 degrees
        angle_increment = self.scan_data.angle_increment
        num_ranges = len(self.scan_data.ranges)
        start_index = int((sector_start_angle -
         ↪  self.scan_data.angle_min) / angle_increment)
        end_index = int((sector_end_angle - self.scan_data.angle_min) /
         ↪  angle_increment)
        min_distance = float('inf')
        for i in range(start_index, end_index):
            if 0 < self.scan_data.ranges[i] < min_distance:
                min_distance = self.scan_data.ranges[i]
        print(f"min distance : {min_distance}")
        return min_distance > obstacle_distance_threshold and
         ↪  min_distance < (obstacle_distance_threshold + 0.5)
```

Code Block 0.0.22: Obstacle Detection

```
  def check_for_obstacles(self, event):
    if self.scan_data is None or len(self.scan_data.ranges) == 0:
      return

print("Checking for obstacles using LIDAR data")
  front_angle_range = 30  # Degrees
    front_angle_range_rad = np.deg2rad(front_angle_range)
    angle_increment = self.scan_data.angle_increment
    num_ranges = len(self.scan_data.ranges)
    min_angle = self.scan_data.angle_min
    start_index = max(0, int((min_angle - front_angle_range_rad / 2) /
    ↪  angle_increment))
    end_index = min(num_ranges - 1, int((min_angle +
    ↪  front_angle_range_rad / 2) / angle_increment))
    min_distance = float('inf')
    for i in range(start_index, end_index):
      if 0 < self.scan_data.ranges[i] < min_distance and
      ↪  np.isfinite(self.scan_data.ranges[i]):
         min_distance = self.scan_data.ranges[i]
         obstacle_distance_threshold = 0.3
    if min_distance < obstacle_distance_threshold:
        self._obstacleInFront = True
        rospy.loginfo(f"Obstacle detected at {min_distance:.2f}
        ↪  meters!")
        self.stop()  # Stop the robot
        self._obstacleChecker += 1

    if self._obstacleChecker >= 5:
      rospy.loginfo(
      ↪  "Obstacle is still present. Calling contour_obstacle function."
      ↪  )
      self._obstacleChecker = 0

    if not self._contourningObstacle:
        self._contourningObstacle = True
        self.contour_obstacle()
    else:
        self._obstacleInFront = False
        self._obstacleChecker = 0
```

```
        rospy.loginfo("No obstacle detected. Moving.")
```

The image processing component of the system is designed to enhance the robot's navigation and interaction capabilities by subscribing to multiple camera topics, such as */camera/rgb/image_raw* and */camera_qr_code_feed*. This allows the robot to perform tasks like line following and QR code detection. For line following, the system detects black lines using techniques such as *thresholding* and *contour detection*, enabling the robot to accurately track and follow predefined paths. Additionally, the system incorporates QR code detection using the *QReader library*, which decodes QR codes to extract relevant information or commands. To ensure precise alignment with detected lines, the system implements the *_adjust_orientation* method, which dynamically adjusts the robot's movement based on real-time line detection data. Together, these features enable the robot to navigate complex environments and interact with visual markers effectively.

The task state management system is responsible for monitoring and controlling the execution of tasks. It tracks the task's current state using the *_running* variable, which indicates whether the task is active or inactive. The system provides two key methods: *start* to initiate the task and *stop* to halt it, ensuring flexibility and control over task execution. Additionally, it emits signals such as *finishedSignal* and *failedSignal* to notify the system when a task has been successfully completed or has encountered a failure. These signals enable the system to respond appropriately, whether by transitioning to the next task or handling errors, ensuring smooth and efficient operation.

The system incorporates *robust error handling* to manage potential issues across various components, including ROS callbacks, image processing, and task execution. Errors and task statuses are logged using *rospy.loginfo* for informational messages and *rospy.logerr* for error reporting, ensuring transparency and ease of debugging.

To enhance efficiency and responsiveness, the system leverages *asynchronous execution* through the *asyncio* library. The *_execute* method is designed to run tasks in a non-blocking manner, allowing the robot to perform multiple operations simultaneously without interruptions. This approach ensures smooth operation and maintains system responsiveness, even during complex or time-consuming tasks.

Technical Details:

The system is tightly integrated with ROS (Robot Operating System) to facilitate seamless communication and control. It utilizes *subscribers* to receive critical data from various sensors and topics:

- */odom*: Provides odometry data, enabling the system to track the robot's position and orientation.

- */scan*: Delivers LIDAR data, which is essential for obstacle detection and avoidance.
- */camera/rgb/image_raw, /camera_qr_code_feed*, and other camera topics: Supply image feeds for tasks like line following and QR code detection.

On the output side, the system employs *publishers* to send commands and data:

- */cmd_vel*: Publishes velocity commands to control the robot's movement, such as forward motion, turns, and stops.
- */error*: Publishes error values for debugging and monitoring system performance.
- */position_joint_controller/command*: Sends commands to control lifting mechanisms or other actuators, if applicable.

This integration ensures efficient data flow and real-time control, enabling the robot to perform complex tasks with precision and reliability.

0.0.4.2.1.1 Navigation system

The navigation system is composed of three key functions: *_adjust_orientation*, *check_side_pixels*, and *check_straight_pixels*. These methods work together to enable the robot to follow lines and detect junctions effectively, ensuring smooth and accurate navigation in its environment.

The *_adjust_orientation* method is responsible for adjusting the robot's orientation based on the detected line and the number of black pixels in the camera image. It ensures the robot stays centered on the line or makes appropriate turns at junctions. This method takes several input parameters, including the error (deviation of the line from the center of the image), the angle of the detected line, the number of black pixels in the region of interest, the binary mask of the image, and the dimensions of the bounding box around the detected line. If the robot is moving to a lift position, it adjusts its orientation to reach the target. Otherwise, it checks the number of black pixels to determine the robot's position relative to the line. If the number of black pixels is within a defined threshold, the robot moves forward while adjusting its orientation based on the error. If the number of black pixels exceeds the threshold, the robot uses *check_side_pixels* and *check_straight_pixels* to determine if it's at a junction. Based on the detected line configuration, the robot decides whether to turn left, right, or move forward. The method then publishes velocity commands to the */cmd_vel* topic to adjust the robot's movement.

Code Block 0.0.23: Adjust Orientation Function

```
    def _adjust_orientation(self, error, angle, pixels, mask , w_min,
     ↪  h_min):
        if self._obstacleInFront:
            return None
        if self.moveToTurnPosition:
            self._move(error)
            return
        if self._making_turn:
            print("error", error)
            if error < 60 or error > -60:
                self.stop()
                self._making_turn = False
            return
        if self._making_u_turn:
            print("error", error)
            if error < 60 or error > -60:
                self.stop()
                self._making_u_turn = False
            return
        if (w_min < 100 and h_min < 100):
            self._move_forward()
            return None
        if (pixels <= self._black_pixels + (self._black_pixels *
         ↪  self._threshold)) and  (pixels >= self._black_pixels -
         ↪  (self._black_pixels * self._threshold)):
            self._move(error)
            return None
        if pixels > (self._black_pixels + (self._black_pixels *
         ↪  self._threshold)):
            left_pixels, right_pixels, onLeft, onRight =
             ↪  self.check_side_pixels(mask)
            if onLeft or onRight:
                top_pixels_remaining, top_pixels_side,
                 ↪  bottom_pixels_side, onTop, hastoStop =
                 ↪  self.check_straight_pixels(mask)
                if hastoStop:
                    self.needMakeDecision = True
                    self.junction_decision(onLeft, onRight, onTop)
                    if top_pixels_remaining == 0:
```

```
                    self._move_forward()
                return [left_pixels, right_pixels,
                 ↪  top_pixels_remaining, top_pixels_side,
                 ↪  bottom_pixels_side]
            self._move_forward()
            return [left_pixels, right_pixels, 0, 0, 0]
        elif pixels < (self._black_pixels - (self._black_pixels *
         ↪  self._threshold)):
            self._move(error)
            return None
    self._move(error)
    return None
```

The *check_side_pixels* method checks the number of black pixels on the left and right sides of the image to detect junctions or turns. It takes the binary mask of the image as input and divides the image into left and right halves, excluding the middle region. It counts the number of black pixels in each half and determines if the number of black pixels on either side exceeds a threshold, indicating a potential turn. The method returns the number of black pixels on the left and right sides, along with flags indicating whether a turn is detected. This information is used *by _adjust_orientation* to make decisions at junctions.

Code Block 0.0.24: Check Side Pixels Function

```
def check_side_pixels(self, mask):
    onLeft = False
    onRight = False
    height, width = mask.shape

    # Defining middle region of the mask
    middle_start = (width // 2) - (self._middle_width // 2)
    middle_end = (width // 2) + (self._middle_width // 2)

    # Create a mask for the middle region and remove it from the
     ↪  original mask
    middle_mask = np.zeros_like(mask)
    middle_mask[:, middle_start:middle_end] = 255
    masked_binary = cv2.bitwise_and(mask, cv2.bitwise_not(middle_mask))

    # Split the masked binary into left and right halves
    left_half = masked_binary[:, :width // 2]
```

```
    right_half = masked_binary[:, width // 2:]

    # Count the black pixels in both halves
    left_pixels = np.sum(left_half == 255)
    right_pixels = np.sum(right_half == 255)

    # Define a threshold for the number of black pixels to consider
    ↪ left or right
    threshold_lr = (self._black_pixels // 3) - 20

    # Check if the left or right region has sufficient black pixels
    if left_pixels > threshold_lr:
        onLeft = True
    if right_pixels > threshold_lr:
        onRight = True

    return left_pixels, right_pixels, onLeft, onRight
```

The *check_straight_pixels* method checks the number of black pixels in the top and bottom halves of the image to detect straight paths or junctions. It also takes the binary mask of the image as input and divides the image into top and bottom halves, excluding the middle region. It counts the number of black pixels in each half and determines if the number of black pixels in the bottom half exceeds the top half, indicating a potential junction. If a junction is detected, it checks the continuity of the line in the top half to determine if the robot should move forward. The method returns the number of black pixels in the top and bottom halves, along with flags indicating whether the robot should stop or move forward.

Code Block 0.0.25: Check Straight Pixels Function

```
def check_straight_pixels(self, mask):
    onTop = False
    hastoStop = False
    height, width = mask.shape
    middle_start = (width // 2) - (self._middle_width // 2)
    middle_end = (width // 2) + (self._middle_width // 2)

    # Create a mask for the middle region and remove it from the
    ↪ original mask
    middle_mask = np.zeros_like(mask)
```

```
        middle_mask[:, middle_start:middle_end] = 255
        masked_binary = cv2.bitwise_and(mask, cv2.bitwise_not(middle_mask))

        # Split the masked binary into top and bottom halves
        top_half = masked_binary[:height // 2, :]
        bottom_half = masked_binary[height // 2:, :]

        # Count the white pixels in the top and bottom halves
        top_pixels = np.sum(top_half == 255)
        bottom_pixels = np.sum(bottom_half == 255)

        # Check if bottom pixels are greater than or equal to top pixels,
        ↪  implying a need to stop
        if bottom_pixels >= top_pixels:
            hastoStop = True
            middle_line = mask[:, middle_start:middle_end]
            height, _ = middle_line.shape

            # Split the middle line into top and bottom portions
            top_half = middle_line[:height // 2, :]
            bottom_half = middle_line[height // 2:, :]

            # Zero out the black pixels in the bottom portion
            bottom_half[bottom_half == 255] = 0

            # Count remaining white pixels in the top half
            top_pixels_remaining = np.sum(top_half == 255)
            continuity_threshold = 0.3 * self._black_pixels

            # Check if top pixels remaining are above the threshold
            if top_pixels_remaining >= continuity_threshold:
                onTop = True

        return top_pixels_remaining, top_pixels, bottom_pixels, onTop,
        ↪  hastoStop
```

Together, these methods form a robust navigation system that allows the robot to follow lines accurately and make informed decisions at junctions. The *_adjust_orientation* method continuously adjusts the robot's movement based on real-time data, while *check_side_pixels*

and *check_straight_pixels* provide critical information about the robot's surroundings. This combination ensures the robot can navigate complex environments with precision and reliability, adapting its behavior based on the detected line and junction configurations.

0.0.4.2.1.2 Interaction Between Methods:

Line Following: The *_adjust_orientation* method uses *check_side_pixels* and *check_straight_pixels* to determine if the robot is at a junction or should continue following the line. If the robot is on a straight path, it adjusts its orientation based on the error and moves forward.

Junction Detection: When the number of black pixels exceeds the threshold, *check_side_pixels* and *check_straight_pixels* are called to detect junctions. Based on the results, the robot makes a decision to turn left, right, or move forward.

Navigation: *The junction_decision* method (in child classes like *Mapping* and *Carrying*) uses the output of *check_side_pixels* and *check_straight_pixels* to navigate junctions and reach goal zones.

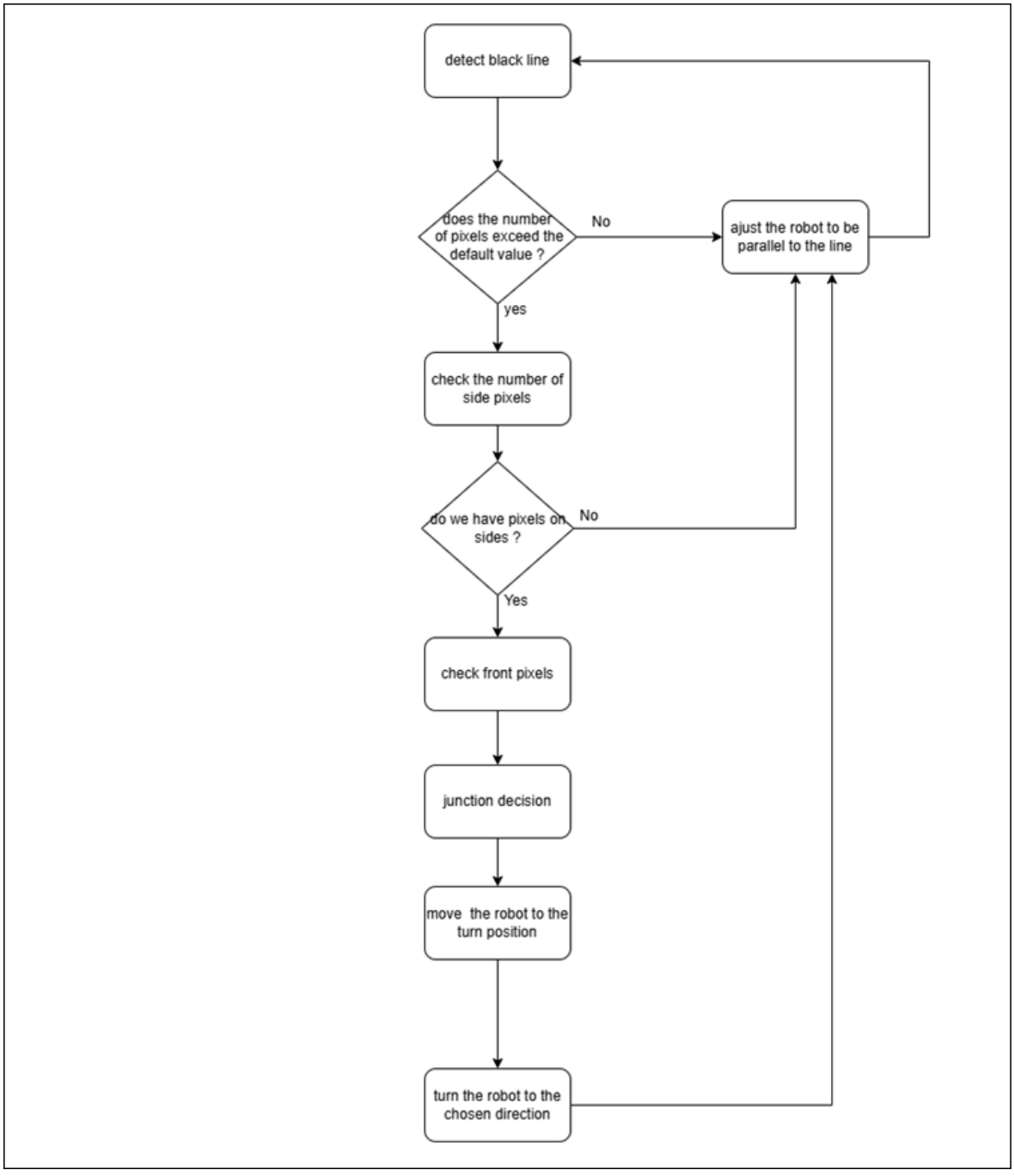

Figure 7: line following & junction flowchart

0.0.4.2.1.3 Example Scenario

- *Following a Straight Line: The robot detects a line with a small error and a number of black pixels within the threshold. The _adjust_orientation method adjusts the robot's orientation and moves it forward (see* code block 0.0.25 Robot following the line and detecting a junction*)* .
- *Approaching a Junction: The robot detects a significant increase in black pixels, indicating a junction.* The *check_side_pixels* method detects more black pixels on the left side, suggesting a left turn. The *check_straight_pixels* method confirms that the robot should stop and make a decision. The *junction_decision* method instructs the robot to turn left, so the robot move to the turn positrion (see fig. 7 the robot is at the turn position), and turn to the choosen direction (see fig. 8 The robot turns to the chosen direction).
- Navigating a Complex Path: The robot uses a combination *of check_side_pixels, check_straight_pixels,* and *junction_decision* to navigate through multiple junctions and reach its goal.

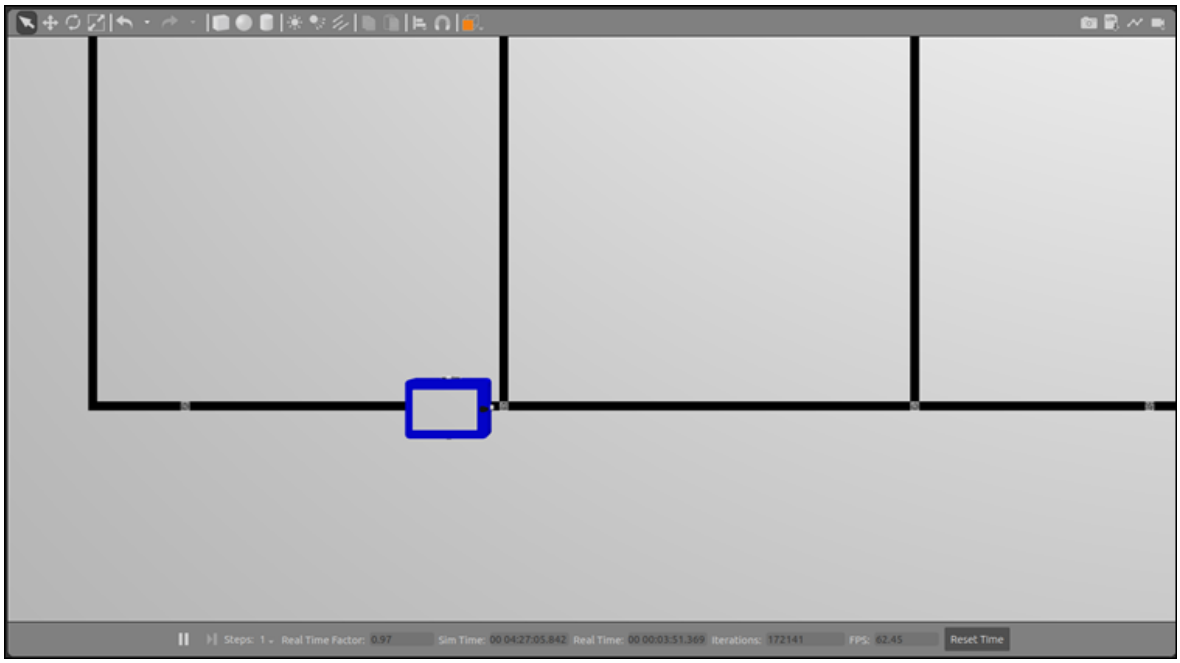

Figure 8: Robot following the line and detecting a junction

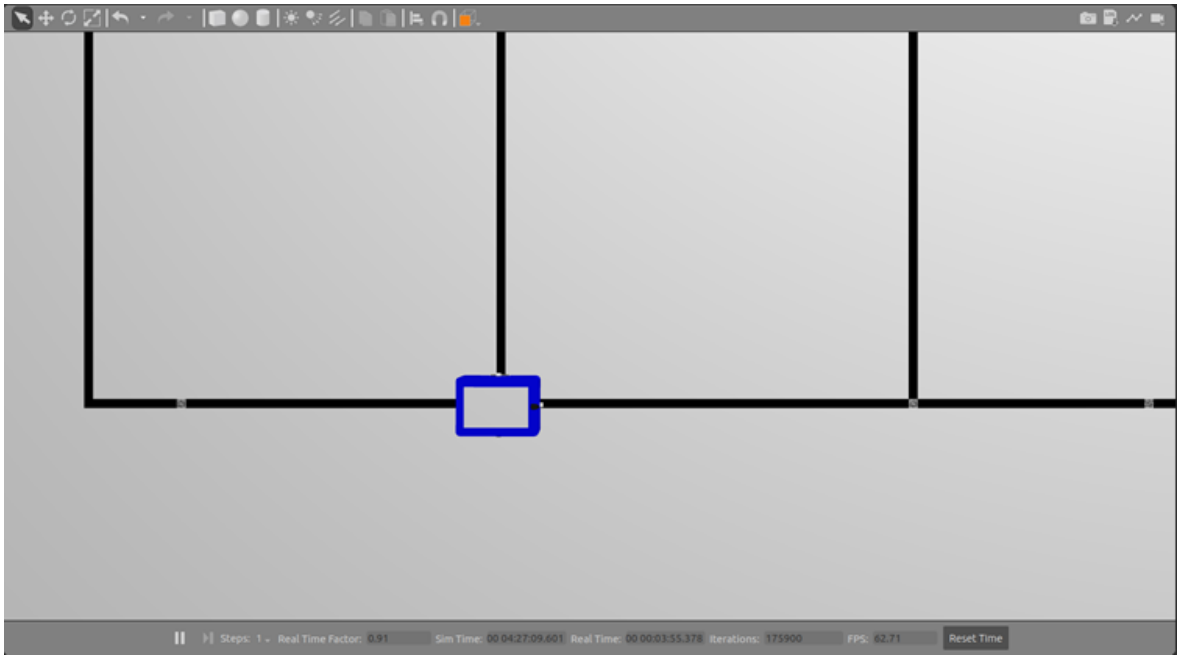

Figure 9: the robot is at the turn position

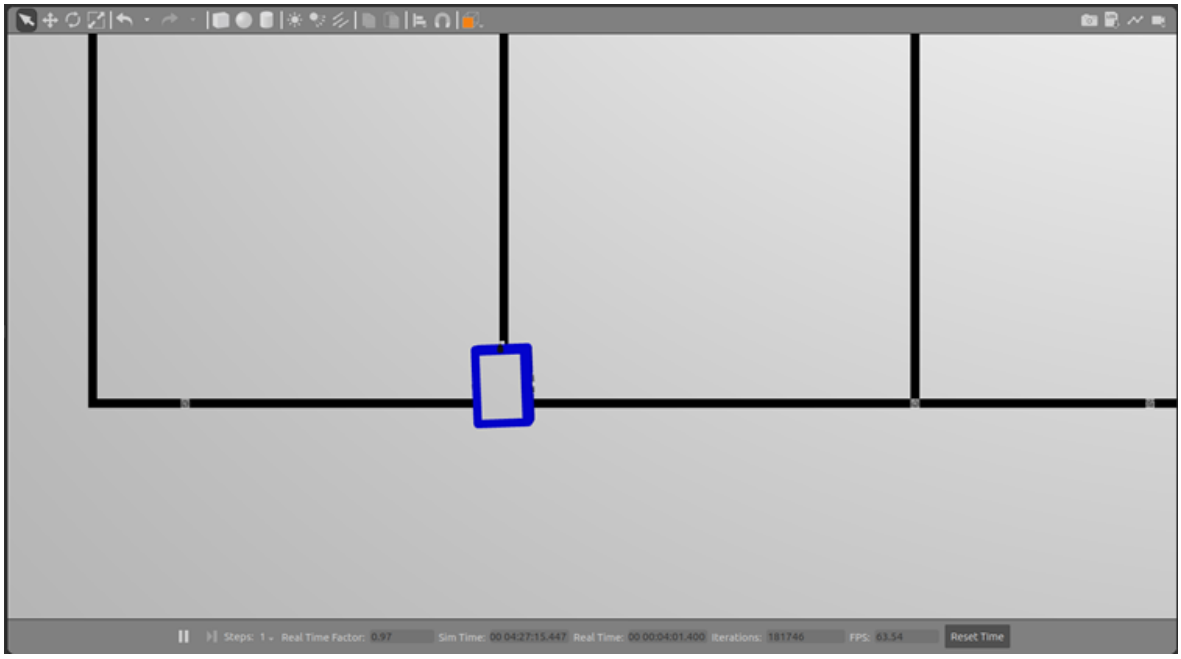

Figure 10: The robot turns to the chosen direction

0.0.4.2.1.4 Essentials parameters

The parameters (*self.param, self._black_pixels, self._sideBlackPixels, self._threshold, self._middle_width, self.distanceToQr, and obstacle_distance_threshold*) are critical to the functionality of the Task class. These values were carefully tuned through *extensive testing and experimentation* to ensure optimal performance. Let's break down each parameter, its purpose, and how it was determined:

_black_pixels: Represents the expected number of black pixels in the camera image when the robot is following a straight the line. With a default Value of 493850, this value was determined by analyzing the camera image when the robot is perfectly aligned on the line. This parameter is used as a reference value for line-following logic. If the number of black pixels deviates significantly from this value, the robot adjusts its orientation.

_sideBlackPixels: Represents the expected number of black pixels on the sides of the camera image when the robot is approaching a turn. Its Value is 414556, Determined by analysing the camera image when the robot is approaching a junction or turn.

It's used to detect when the robot should initiate a turn (e.g., when the number of black pixels on one side exceeds this threshold).

_threshold: Defines the acceptable deviation from self._black_pixels for line-following.

The correct value found after test is 0.119 (11.9%). This parameter determined through experimentation to balance sensitivity and robustness. If the number of black pixels deviates by more than 11.9% from self._black_pixels, the robot adjusts its orientation.

_middle_width: Defines the width of the region of interest (ROI) in the camera image for line detection. Its value is 580 (pixels width). Set to focus on the central portion of the camera image, where the line is most likely to be.Adjusted to ensure the robot can detect the line even if it deviates slightly from the center.

distanceToQr: Represents the distance (in meters) from the robot to the QR code when it is detected. With a value of 0.3869 meters, determined by measuring the distance at which the QR code is reliably detected and decoded and ensures the robot stops at an appropriate distance from the QR code for accurate processing.

obstacle_distance_threshold: Defines the minimum distance (in meters) at which an obstacle is considered too close and requires avoidance. Its value is 0.3 (meters).

Determined through testing to balance safety and efficiency. A smaller value could risk collisions, while a larger value could cause unnecessary detours.

0.0.4.2.1.5 How These Values Were Determined:

- *Iterative Testing*: Each parameter was initially set to a rough estimate based on theoretical calculations or prior experience. The robot was then tested in various scenarios

(e.g., line following, obstacle avoidance, QR code detection) to observe its behaviour.

- *Incremental Adjustments*: Parameters were adjusted incrementally to improve performance. For example: Adjusting *self._threshold* to reduce oscillations during line following.
- *Real-World Validation*: The robot was tested in real-world environments (e.g., with varying lighting conditions, uneven surfaces, and different obstacle configurations) to ensure robustness.
- Trade-Offs: Some parameters required trade-offs. For example: A larger *obstacle_distance_threshold* increases safety but may result in longer detours.

0.0.4.2.1.6 Child Classes: Mapping and Carrying

The child classes (Mapping and Carrying) will extend the Task class to implement task-specific logic.

Mapping: Focuses on exploring the environment, detecting QR codes, and building a map.

Carrying: Focuses on transporting items between specified zones.

0.0.4.2.2 Mapping

The *Mapping class* is designed to explore the environment, detect QR codes, and store their positions in a database. It inherits from the *Task class*, leveraging its core functionality such as robot control, obstacle avoidance, and line following, while adding mapping-specific logic to fulfill its unique role. This inheritance allows the Mapping class to reuse existing methods and focus on extending functionality for environment exploration and QR code detection.

One of the key features of the Mapping class is *QR code detection*, which is handled by the *_check_qr* method. This method detects QR codes using image processing techniques and stores the detected QR codes along with their positions in a dictionary (*self.qrcodes*). To prevent duplicate detections, the class compares new QR codes with the last detected one (*self._lastQrCode*), ensuring that each QR code is only recorded once. This avoids redundancy and improves the efficiency of the mapping process.

For *environment exploration*, the class utilizes the *junction_decision* method to navigate junctions systematically. When a junction is detected (e.g., left, right, or top), the robot makes a decision to turn or move forward based on the exploration strategy. This ensures that the robot explores all possible paths in the environment, leaving no area unmapped.

The systematic approach guarantees comprehensive coverage, which is essential for accurate mapping.

The Mapping class also integrates with an *SQLite database* (qr_code.db) to store detected QR codes and their positions. The *_store* method is responsible for inserting QR code data into the database, while the *_checkExistingQrCodes* method checks for existing QR codes to avoid redundant entries. This ensures that the database remains up-to-date and free of duplicates. The database schema includes a table named *qr_code* with columns for id (primary key), name (QR code identifier), position_x (X-coordinate), and position_y (Y-coordinate), providing a structured way to store and retrieve mapping data.

When the mapping process is complete, the class emits a signal (*fsignal*) containing the detected QR codes, notifying other system components of the task's completion. It also calls the parent class's *_task_finished* method to clean up resources and notify the system, ensuring a smooth transition to the next task. The use of the *signalslot library* enables decoupled communication between the Mapping class and other components, enhancing modularity and flexibility.

The class implements *robust error handling* to manage potential issues during database operations, such as table creation or data insertion. Errors are logged using *rospy.logerr*, providing detailed information for debugging and maintenance. Additionally, the class ensures that the database connection is properly closed, preventing resource leaks and ensuring data integrity.

0.0.4.2.2.1 How It Works

The *Mapping class* operates through a well-defined workflow that ensures systematic environment exploration, QR code detection, and data storage. Here's how it works in detail:

Initialization

The Mapping class begins by initializing itself through the parent class's constructor using super().__init__("mapping"). This sets up the core functionality inherited from the *Task class*, such as robot control, obstacle avoidance, and line following. Additionally, the class initializes the *fsignal*, a signal used to notify other components when the mapping task is complete. This signal is essential for decoupled communication within the system.

Start Mapping

The mapping process is initiated by calling the *start* method. This method first checks for existing QR codes in the SQLite database using the *_checkExistingQrCodes* method. If QR codes are already present in the database, the task completes immediately by emitting the *fsignal* with the existing data. This avoids redundant mapping and saves time. If no QR codes are found, the robot begins exploring the environment, systematically navigating through the space to detect and record new QR codes.

QR Code Detection

QR code detection is handled by the *_check_qr* method. When a QR code is detected, the method processes it by converting its position to global coordinates using the *_calculate_distance* function. This ensures that the QR code's location is accurately recorded relative to the robot's environment. The detected QR code and its position are then stored in the *self.qrcodes* dictionary. To prevent duplicate detections, the method compares the new QR code with the last detected one (*self._lastQrCode*). If the QR code has already been recorded, it is ignored, ensuring that only unique QR codes are stored.

Code Block 0.0.26: Mapping Qr Code Checker

```
def _check_qr(self, decoded_text):
  if not self._processQrCode :
    self._processQrCode = True
    if decoded_text[0] == "None" or decoded_text[0] is None:
        self._processQrCode = False
        return
    if decoded_text[0] != self._lastQrCode:
        if len(self.qrcodes) >= 1:
          if (decoded_text[0] == list(self.qrcodes.keys())[0]):
              self._processQrCode = False
              self._task_finished(message="Mapping process finished")
              return
        self.hasDetectedQrRecently = False
        position = self._calculate_distance(self.robot_pose)
        self.qrcodes[decoded_text[0]] = position
        self._lastQrCode = decoded_text[0]
        self._processQrCode = False
    self._processQrCode = False
```

Junction Navigation

The *junction_decision* method plays a critical role in navigating the robot through the environment. When the robot encounters a junction, this method determines the appropriate action based on the type of junction detected:

- If a *top junction* is detected, the robot moves forward to continue exploration.
- If a *left or right junction* is detected, the robot prepares to turn, ensuring that all paths are systematically explored. This method ensures comprehensive coverage of the environment, leaving no area unmapped.

Code Block 0.0.27: Junction decision of mapping class

```
def junction_decision(self, onLeft, onRight, onTop):
    tm = 3
    if onTop:
    print("On top")
    tm = 1
    self._move_forward()
    self.timer= rospy.Timer(rospy.Duration(tm),self.resume_processing)
    elif onLeft:
        self._turnSide = "left"
        self.moveToTurnPosition = True
    elif onRight:
        self._turnSide = "right"
        self.moveToTurnPosition = True
```

Task Completion

Once the mapping process is complete, the *_task_finished* method is called to wrap up the task. This method performs several key actions:

1. It stores the detected QR codes in the SQLite database using the *_store* method, ensuring that the data is saved for future use.

2. It emits the *fsignal* with the QR code data, notifying other system components that the task is complete.

3. It calls the parent class's *_task_finished* method to clean up resources and notify the system, ensuring a smooth transition to the next task.

0.0.4.2.2.2 Impact on the System

The Mapping class enables the robot to *autonomously explore its environment*, *detect QR codes*, and *store their positions* for future use. It leverages the parent class's functionality for robot control and obstacle avoidance, demonstrating the power of *inheritance* and *code reuse*. The integration with SQLite ensures *persistent storage* of mapping data, enabling other tasks (e.g., carrying) to use this information.

0.0.4.2.3 Carrying

The *Carrying class* is a specialized child class of the *Task class*, designed to handle the transportation of items between specified zones. It extends the parent class by adding

carrying-specific functionality, such as QR code navigation, lifting mechanism control, and complex junction navigation. By leveraging the core functionality of the Task class—such as robot control, obstacle avoidance, and line following—the Carrying class focuses on implementing logic tailored to item transportation tasks.

0.0.4.2.3.1 Key Features and Functionality

Task Initialization:

The Carrying class initializes by accepting a list of tasks (params) that specify the zones and actions to be performed, such as loading or unloading. It calls the parent class's constructor with the task name "carrying" to set up the core functionality. This initialization ensures the robot is ready to execute the carrying task with the necessary parameters.

QR Code Navigation:

The class uses QR codes to navigate to goal positions. It implements the *_check_qr* method to detect QR codes and determine if the robot has reached a goal zone. By comparing the detected QR code with the target zone, the robot can confirm its location and proceed with the next action, such as loading or unloading.

Lifting Mechanism:

The Carrying class controls a lifting mechanism to handle items during loading and unloading. The *_lift* method is responsible for interpolating the lifting motion over a specified duration, ensuring smooth and precise movement. Commands for the lifting mechanism are published to the */position_joint_controller/command* topic, enabling real-time control.

Junction Navigation:

The class extends the *junction_decision* method to handle complex navigation decisions. When the robot encounters a junction, it determines the best direction to reach the goal zone based on QR code positions and intersections. The class uses a *shortest path algorithm* to choose between turning left, right, or making a U-turn, ensuring efficient navigation.

Task Completion:

The class tracks the progress of the task using *self._currentTask*. Once all goals have been reached, the task is marked as complete, and a signal (*fsignal*) is emitted to notify the system. This ensures that other components are aware of the task's completion and can take appropriate action.

0.0.4.2.3.2 How It Works

Initialization:

The Carrying class initializes by calling the parent class's constructor and setting up the task parameters. It prepares the robot for the carrying task by loading QR code positions and

configuring the lifting mechanism.

Navigation and QR Code Detection.
The robot navigates through the environment using QR codes as markers. The *_check_qr* method detects QR codes and determines if the robot has reached a goal zone. If the target zone is reached, the robot proceeds with loading or unloading.

Lifting Mechanism Control:
The *_lift* method controls the lifting mechanism, interpolating its motion to ensure smooth operation. This method is called during loading and unloading to handle items efficiently.

Junction Navigation:
The *junction navigation* is a critical component of the Carrying class, responsible for handling navigation decisions at junctions. Its purpose is to determine the best action—such as turning left, right, or moving forward—based on the detected QR codes, the robot's current position, and the goal zone. This method ensures the robot navigates efficiently and reaches its destination while avoiding unnecessary detours.

The method takes three key parameters: *onLeft* Indicates whether a left turn is detected, *onRight* Indicates whether a right turn is detected and *onTop* Indicates whether a straight path is detected. These parameters provide information about the robot's immediate environment, helping it decide the best course of action.

The method's logic is divided into two main scenarios: when no QR code is detected and when a QR code is detected.

No QR Code Detected: If no QR code has been scanned recently, the robot makes a decision based on the detected junctions:

- If a *left or right turn* is detected, the robot prepares to turn in the corresponding direction.
- If a *straight path* is detected, the robot moves forward to continue its journey.

This logic ensures the robot continues exploring or navigating even when no QR codes are present, maintaining progress toward its goal.

QR Code Detected: If a QR code has been scanned, the robot checks whether it corresponds to a *goal zone* or an *intersection*:

- If the QR code is a *goal zone*, the robot moves to the lifting position and performs the required action, such as loading or unloading. This marks a key milestone in the task.
- If the QR code is an *intersection*, the robot uses the *_chooseSide* or *_chooseSideToGo* method to determine the best direction to reach the goal zone. These methods calculate the optimal path based on the robot's current position, the goal zone's location, and the available routes.

This logic ensures the robot makes informed decisions at intersections, minimizing travel time and energy consumption.

Based on the decision-making process, the method publishes velocity commands (self.msg) to the */cmd_vel* topic. These commands execute the chosen action, such as turning left, turning right, or moving forward. This ensures the robot's movements are precise and aligned with the navigation strategy.

Code Block 0.0.28: Junction Decision Function Of Carrying Process

```
def junction_decision(self, onLeft, onRight, onTop):
    print("Making decision")
    print(self.hasDetectedQrRecently)
    self.stop()
    tm = 3
    current = self._params[self._currentTask]

    if not self.hasDetectedQrRecently:  # no qrCode has been scanned
        if onLeft and (not onTop) and (not onRight):
            self._turnSide = "left"
            self.moveToTurnPosition = True
        elif onRight and (not onTop) and (not onLeft):
            self._turnSide = "right"
            self.moveToTurnPosition = True
        elif onTop:
            tm = 2
            self._move_forward()
            self.timer = rospy.Timer(rospy.Duration(tm),
             ↪  self.resume_processing, oneshot=True)
        else:  # if the robot can't go straight, failed the task
            self._task_failed(
             ↪  "The robot is stuck : should go straight")

    elif self._lastQrCode:  # qrCode has been scanned
        if self._lastQrCode in self.qrcodes:
            if self._isStationSide(self._lastQrCode):
                tm = 2
                self._move_forward()
                self.timer = rospy.Timer(rospy.Duration(tm),
                 ↪  self.resume_processing, oneshot=True)
            else:
                if self.goal_in_db:
```

```
            current = self._params[self._currentTask]
            zArr = current["zone"].split(" ")
            goal = f"{zArr[0]}_{zArr[1]}_center"
            goalPos = self.qrcodes.get(goal, (None, None))
            side = self._chooseSideToGo(onTop, onRight, onLeft,
             ↪  goalPos)
            if side == "S":
                tm = 2
                self._move_forward()
                self.timer = rospy.Timer(rospy.Duration(tm),
                 ↪  self.resume_processing, oneshot=True)
            elif side == "R":
                self._turnSide = "right"
                self.moveToTurnPosition = True
            elif side == "L":
                self._turnSide = "left"
                self.moveToTurnPosition = True
            elif side == "B":
                tm = 6
                self._U_turn()
                self.timer = rospy.Timer(rospy.Duration(tm),
                 ↪  self.resume_processing, oneshot=True)
            elif side == "0":
                return
        else:
            intersections =
             ↪  self._getIntersection(current['zone'])
            if self._lastQrCode in [intersections[0][0],
             ↪  intersections[1][0]]:
                if onLeft:
                    self._turnSide = "left"
                    self.moveToTurnPosition = True
                elif onRight:
                    self._turnSide = "right"
                    self.moveToTurnPosition = True
                else:
                    self._task_failed(
                     ↪  "The robot is stuck : should turn")
            else:
                if onLeft and (not onTop) and (not onRight):
```

```
                    self._turnSide = "left"
                    self.moveToTurnPosition = True
                elif onRight and (not onTop) and (not onLeft):
                    self._turnSide = "right"
                    self.moveToTurnPosition = True
                else:
                    print("Not a corner, using shortest path")
                    side = self._chooseSide(onTop, onRight,
                     ↪  onLeft, intersections)
                    if side == "S":
                        tm = 2
                        self._move_forward()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                    elif side == "R":
                        self._turnSide = "right"
                        self.moveToTurnPosition = True
                    elif side == "L":
                        self._turnSide = "left"
                        self.moveToTurnPosition = True
                    elif side == "B":
                        tm = 6
                        self._U_turn()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                    elif side == "0":
                        tm = 6
                        self._U_turn()
                        self.timer =
                         ↪  rospy.Timer(rospy.Duration(tm),
                         ↪  self.resume_processing,
                         ↪  oneshot=True)
                        rospy.logerr("Wrong direction")
                        return
        else:
            tm = 2
```

```
            self._move_forward()
            self.timer = rospy.Timer(rospy.Duration(tm),
             ↪  self.resume_processing, oneshot=True)
            rospy.logerr(
             ↪  "The robot is stuck : doesn't know what to do")
```

Task Completion: Once all goals have been reached, the task is marked as complete, and the *fsignal* is emitted to notify the system. The parent class's cleanup methods are called to ensure resources are released and the system is ready for the next task.

0.0.4.2.3.3 Interaction Between Methods

- *Junction Detection*: The junction_decision method detects junctions and determines if the robot should turn left, right, or move forward. If a QR code is detected, it uses _chooseSide or _chooseSideToGo to determine the best direction to reach the goal zone.
- *Path Planning*: The _chooseSide method calculates the shortest path to the goal zone based on the positions of intersections. The _chooseSideToGo method calculates the shortest path to a specific target position (e.g., a goal zone).
- *Navigation*:
  - Based on the chosen direction, the robot executes the corresponding action (e.g., turn left, turn right, move forward).

0.0.4.2.3.4 Example Scenario

1. Approaching a Junction: The robot detects a junction with options to turn left, right, or move forward. The *junction_decision* method calls *_chooseSide* to determine the best direction to reach the goal zone.
2. Calculating the Shortest Path: The *_chooseSide* method calculates the Euclidean distance to each intersection and chooses the direction with the minimum distance. If the goal zone is directly ahead, the robot moves forward. If it's to the left or right, the robot turns accordingly.
3. Executing the Action: The robot executes the chosen action (e.g., turn left, turn right, move forward) and continues navigating toward the goal zone.

0.0.4.2.3.5 Impact on the System

- These methods enable the robot to *autonomously navigate junctions* and *reach goal zones efficiently*, ensuring reliable and efficient task execution.
- They demonstrate the importance of *path planning* and *decision-making algorithms* in robotics, showcasing your ability to implement complex navigation logic.

Code Block 0.0.29: Adjust Orientation Function of Carrying Class

```
def _adjust_orientation(self, error, angle, pixels, mask, w_min,
 ↪  h_min):

        if not self.isLifting :
            if self.moveToLiftPosition:
                if self._distanceToLifePosition() > 0.005:
                    self._move(error)
                    return None
                else:
                    print("Lift position reached")
                    self.stop()
                    self.isLifting = True
                    self._lift()
                return None
            else:
                return super()._adjust_orientation(error, angle,
                 ↪  pixels, mask, w_min, h_min)
```

Code Block 0.0.30: Lift function

```
def _lift(self, duration=3):
    current = self._params[self._currentTask]
    liftType = current["type"]
    start = 0 if liftType == "Loading" else 1
    end = 1 if liftType == "Loading" else 0

    start_time = rospy.Time.now()
    end_time = start_time + rospy.Duration.from_sec(duration)
    rate = rospy.Rate(100)
    while rospy.Time.now() < end_time and not rospy.is_shutdown():
        current_time = rospy.Time.now()
        elapsed = (current_time - start_time).to_sec()
```

```
        fraction = elapsed / duration
        fraction = min(fraction, 1.0)  # Fixed the missing parenthesis
        ↪ here
        current_value = start + fraction * (end - start)
        self.liftPub.publish(current_value)
        rate.sleep()

    self.liftPub.publish(end)
    self.isLifting = False
    self.moveToLiftPosition = False

    self._currentTask += 1
    if(self._currentTask == len(self._params)):
        self.stop()
        self._task_finished(message="All goals have been reached")
```

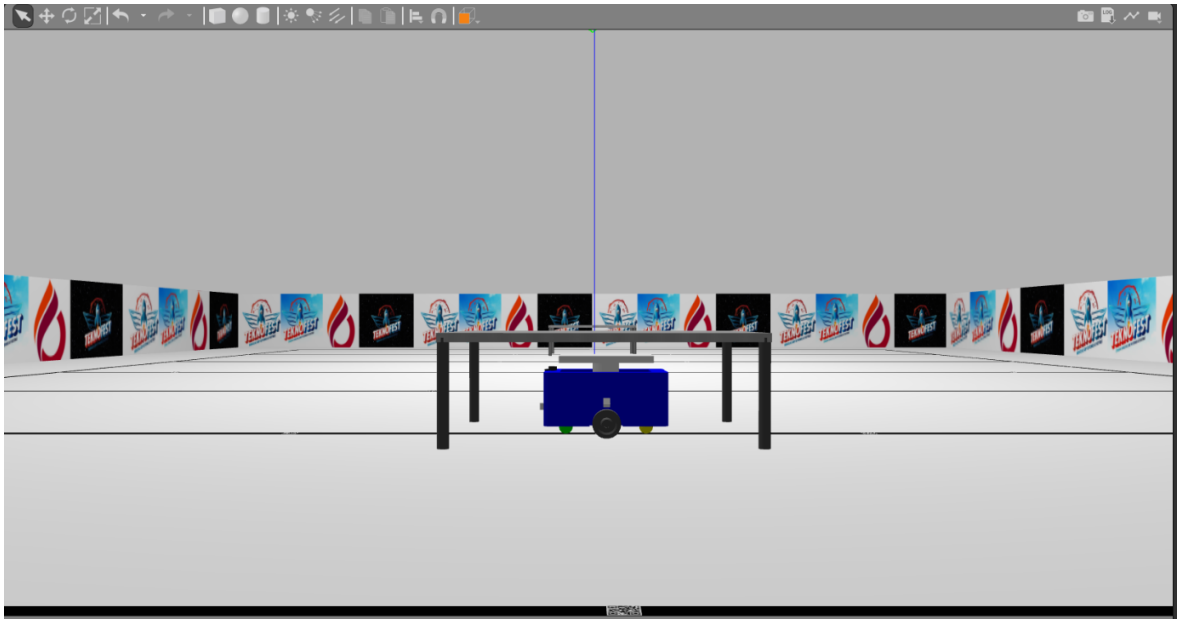

Figure 11: the robot ready to lift a charge