Robot Navigation Using ArUco Markers

Professor in charge:
Prof. Sachit Rao

Group members:

Rajan Khade (MT2024077)
Aditya AV (MT2024009)
Pralay D. Saw (MT2024119)

May 15, 2025

Contents

1 Project Summary

This project implements an autonomous robot navigation system using ArUco markers in a simulated environment. The robot uses a camera to detect ArUco markers placed in the environment, which serve as navigation waypoints. Based on the detected markers, the robot determines its position and orientation, then navigates to the next marker according to predefined instructions.

The system is implemented using ROS 2 (Robot Operating System) for communication between different components, Gazebo for physics simulation, and OpenCV for computer vision tasks including ArUco marker detection. The robot follows a sequence of actions (move forward, turn clockwise, turn counterclockwise) based on the markers it detects, effectively navigating from a starting point to a destination by following the marker trail.

This approach demonstrates a practical application of computer vision in robotics, specifically for indoor navigation where GPS might not be available or reliable.

2 Tools and Technologies

The project utilizes several key tools and technologies:

- **ROS 2 (Robot Operating System)** - Version: Jazzy
  A flexible framework for writing robot software, providing hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more.

- **Gazebo** - Harmonic
  A robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces for simulating robots in complex indoor and outdoor environments.

- **OpenCV** - Version: 4.6
  An open-source computer vision and machine learning software library with more than 2500 optimized algorithms for real-time computer vision tasks.

- **Python** - Version: 3.12
  The primary programming language used for implementing the navigation logic and computer vision processing.

- **ArUco Library** - Part of OpenCV
  A minimal library for ArUco marker detection, which are square fiducial markers that can be used for camera pose estimation and tracking.

3 ROS Concepts

3.1 Key ROS Components

- **Nodes**: Processes that perform computation. In our project, the main nodes include:

  - `navigation_03`: Our custom node that processes camera images and controls robot movement
  - `ros_gz_bridge`: Bridges communication between ROS 2 and Gazebo
  - `robot_state_publisher`: Publishes the robot's state to the TF tree

- **Topics**: Named buses over which nodes exchange messages. Key topics in our system:
  - `/camera/image`: Camera image data
  - `/camera/camera_info`: Camera calibration information
  - `/cmd_vel`: Robot velocity commands
  - `/joint_states`: Robot joint state information

- **Messages**: Typed data structures for communication between nodes
  - `sensor_msgs/Image`: For camera images
  - `sensor_msgs/CameraInfo`: For camera parameters
  - `geometry_msgs/Twist`: For velocity commands

3.2 URDF and SRDF

- **URDF (Unified Robot Description Format)**: XML format for representing robot models
  - Defines the robot's physical structure (links, joints)
  - Specifies visual and collision properties
  - Includes inertial parameters for physics simulation

- **SRDF (Semantic Robot Description Format)**: Complements URDF with semantic information
  - Defines groups of joints and links
  - Specifies end effectors and other semantic structures
  - Used primarily for motion planning

In our project, the robot description is defined in the `my_robot_description` package, which contains the URDF files that specify the robot's structure, including its base, wheels, and camera.

4 Robot Structure and Physics

4.1 Robot Description

The robot is defined using URDF (Unified Robot Description Format) and Xacro (XML Macros), which are XML-based formats used in ROS to describe the physical properties of a robot. The robot model is organized in several Xacro files in the `my_robot_description/urdf` directory, each handling different aspects of the robot structure.

4.1.1 Main Robot Structure

The main robot description is defined in `my_robot.urdf.xacro`, which includes other Xacro files to build the complete robot model:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="my_robot">

  <!-- Include common properties -->
  <xacro:include filename="common_properties.xacro" />

  <!-- Include mobile base -->
  <xacro:include filename="mobile_base.xacro" />

  <!-- Include camera -->
  <xacro:include filename="camera.xacro" />

  <!-- Include Gazebo-specific elements -->
  <xacro:include filename="mobile_base_gazebo.xacro" />

  <!-- Create the mobile base -->
  <xacro:mobile_base/>

  <!-- Attach camera to the robot -->
  <xacro:camera parent="base_link" prefix="camera">
    <origin xyz="0.15 0 0.05" rpy="0 0 0"/>
  </xacro:camera>

</robot>
```

Listing 1: Main Robot URDF Structure (my_robot.urdf.xacro)

4.1.2 Mobile Base

The mobile base is defined in `mobile_base.xacro`, which creates a differential drive robot with two wheels:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="mobile_base">
    <!-- Base Link -->
    <link name="base_link">
      <visual>
        <geometry>
          <box size="0.3 0.3 0.1"/>
        </geometry>
        <material name="${green}"/>
      </visual>
      <collision>
        <geometry>
          <box size="0.3 0.3 0.1"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="1.0"/>
        <inertia ixx="0.01" ixy="0.0" ixz="0.0" iyy="0.01" iyz="0.0"
   izz="0.01"/>
```

```
    </inertial>
  </link>

  <!-- Right Wheel -->
  <link name="right_wheel">
    <visual>
      <geometry>
        <cylinder radius="0.05" length="0.04"/>
      </geometry>
      <material name="${black}"/>
    </visual>
    <collision>
      <geometry>
        <cylinder radius="0.05" length="0.04"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz=
"0.0001"/>
    </inertial>
  </link>

  <joint name="right_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <axis xyz="0 0 1"/>
    <origin xyz="0.0 -0.15 0" rpy="0 1.5707 0"/>
  </joint>

  <!-- Left Wheel -->
  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder radius="0.05" length="0.04"/>
      </geometry>
      <material name="${black}"/>
    </visual>
    <collision>
      <geometry>
        <cylinder radius="0.05" length="0.04"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz=
"0.0001"/>
    </inertial>
  </link>

  <joint name="left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <axis xyz="0 0 1"/>
    <origin xyz="0.0 0.15 0" rpy="0 1.5707 0"/>
  </joint>

  <!-- Caster Wheel -->
```

```xml
    <link name="caster_wheel">
      <visual>
        <geometry>
          <sphere radius="0.05"/>
        </geometry>
        <material name="${black}"/>
      </visual>
      <collision>
        <geometry>
          <sphere radius="0.05"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="0.1"/>
        <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz=
  "0.0001"/>
      </inertial>
    </link>

    <joint name="caster_wheel_joint" type="fixed">
      <parent link="base_link"/>
      <child link="caster_wheel"/>
      <origin xyz="-0.13 0 -0.05" rpy="0 0 0"/>
    </joint>
  </xacro:macro>

</robot>
```

Listing 2: Mobile Base Definition (mobile_base.xacro)

4.1.3 Camera

The camera is defined in `camera.xacro`, which creates a camera link and the necessary joints to attach it to the robot:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="camera" params="parent prefix *origin">
    <!-- Camera Link -->
    <link name="${prefix}_link">
      <visual>
        <geometry>
          <box size="0.03 0.03 0.03"/>
        </geometry>
        <material name="${blue}"/>
      </visual>
      <collision>
        <geometry>
          <box size="0.03 0.03 0.03"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="0.1"/>
        <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz=
  "0.0001"/>
      </inertial>
    </link>

    <!-- Camera Joint -->
    <joint name="${prefix}_joint" type="fixed">
      <parent link="${parent}"/>
      <child link="${prefix}_link"/>
      <xacro:insert_block name="origin"/>
    </joint>

    <!-- Camera Optical Frame -->
    <link name="${prefix}_optical_frame"/>

    <joint name="${prefix}_optical_joint" type="fixed">
      <parent link="${prefix}_link"/>
      <child link="${prefix}_optical_frame"/>
      <origin xyz="0 0 0" rpy="-1.5707 0 -1.5707"/>
    </joint>
  </xacro:macro>

</robot>
```

Listing 3: Camera Definition (camera.xacro)

4.1.4 Gazebo-Specific Configuration

The `mobile_base_gazebo.xacro` file contains Gazebo-specific configurations for the robot, including plugins for the differential drive controller and camera sensor:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- Gazebo plugin for differential drive -->
  <gazebo>
    <plugin name="differential_drive_controller" filename="
   libgazebo_ros_diff_drive.so">
      <update_rate>30</update_rate>
      <left_joint>left_wheel_joint</left_joint>
      <right_joint>right_wheel_joint</right_joint>
      <wheel_separation>0.3</wheel_separation>
      <wheel_diameter>0.1</wheel_diameter>
      <max_wheel_acceleration>1.0</max_wheel_acceleration>
      <max_wheel_torque>20</max_wheel_torque>
      <command_topic>cmd_vel</command_topic>
      <publish_odom>true</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>
      <odometry_topic>odom</odometry_topic>
      <odometry_frame>odom</odometry_frame>
      <robot_base_frame>base_link</robot_base_frame>
    </plugin>
  </gazebo>

  <!-- Gazebo plugin for camera -->
  <gazebo reference="camera_link">
    <sensor type="camera" name="camera">
      <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>640</width>
          <height>480</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.
   so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>camera</cameraName>
        <imageTopicName>image</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_optical_frame</frameName>
        <hackBaseline>0.07</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
      </plugin>
    </sensor>
  </gazebo>

</robot>
```

Listing 4: Gazebo Configuration (mobile_base_gazebo.xacro)

4.1.5 Common Properties

The `common_properties.xacro` file defines common properties used throughout the robot description, such as materials and constants:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- Colors -->
  <xacro:property name="black" value="Gazebo/Black"/>
  <xacro:property name="white" value="Gazebo/White"/>
  <xacro:property name="red" value="Gazebo/Red"/>
  <xacro:property name="green" value="Gazebo/Green"/>
  <xacro:property name="blue" value="Gazebo/Blue"/>
  <xacro:property name="grey" value="Gazebo/Grey"/>

  <!-- Material definitions -->
  <material name="${black}">
    <color rgba="0.0 0.0 0.0 1.0"/>
  </material>

  <material name="${white}">
    <color rgba="1.0 1.0 1.0 1.0"/>
  </material>

  <material name="${red}">
    <color rgba="1.0 0.0 0.0 1.0"/>
  </material>

  <material name="${green}">
    <color rgba="0.0 1.0 0.0 1.0"/>
  </material>

  <material name="${blue}">
    <color rgba="0.0 0.0 1.0 1.0"/>
  </material>

  <material name="${grey}">
    <color rgba="0.5 0.5 0.5 1.0"/>
  </material>

</robot>
```

Listing 5: Common Properties (common_properties.xacro)

The robot description is modular and follows best practices for ROS robot modeling:

- **Separation of concerns**: Different aspects of the robot (base, camera, Gazebo configuration) are in separate files
- **Use of Xacro macros**: Parameterized macros allow for reusable components
- **Proper inertial properties**: Mass and inertia tensors are defined for physics simulation
- **Gazebo integration**: Specific plugins for differential drive and camera sensor

This modular approach makes the robot description easier to maintain and extend. The camera is properly attached to the robot base with the correct optical frame orientation for computer vision tasks, which is essential for the ArUco marker detection used in navigation.

4.2 Physics Simulation

Gazebo handles the physics simulation, which includes:

- **Rigid Body Dynamics**: Simulates the robot's movement based on applied forces
- **Collision Detection**: Prevents the robot from passing through obstacles
- **Friction and Gravity**: Realistic interaction with the environment

The simulation world is defined in `my_robot_bringup/worlds/my_world.sdf`, which specifies the environment including walls, floor, lighting, and ArUco marker placements.

5 ArUco Markers

5.1 What are ArUco Markers?

ArUco markers are square fiducial markers that consist of a black border and an inner binary matrix that determines its identifier (ID). They are designed for easy detection and pose estimation in computer vision applications.

Key features of ArUco markers:

- Unique ID encoded in the binary pattern
- Fast detection even in challenging conditions
- Accurate pose estimation when camera is calibrated
- Robust to partial occlusion and varying lighting conditions

5.2 Marker Generation

In our project, ArUco markers are generated using OpenCV's ArUco library. The following code snippet demonstrates how we generate the markers:

```python
import cv2
import cv2.aruco as aruco
import matplotlib.pyplot as plt

# Define dictionary
aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_250)

# Generate marker image
for i in range(10):
    marker_id = i
    marker_size = 170  # Pixels
    marker_img = cv2.aruco.generateImageMarker(aruco_dict, marker_id,
   marker_size)

    # Save the marker image
    plt.imsave(f"images/marker_{marker_id}.png", marker_img, cmap="gray
   ")
```

Listing 6: ArUco Marker Generation Code

The generated markers are then used to create 3D models for the Gazebo simulation environment. These models are placed at strategic locations in the simulated world to guide the robot's navigation.

6 System Architecture

The system architecture is represented by the ROS node graph, which shows how different components communicate with each other. The following figure illustrates the complete system architecture:

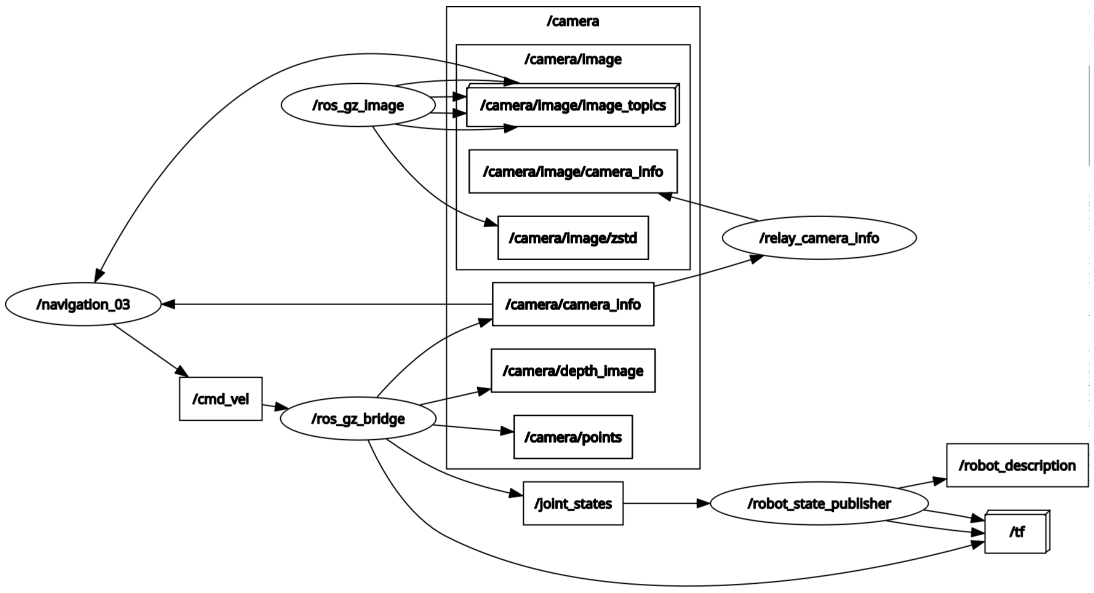


Figure 1: ROS Node Graph showing the communication between different components

The system architecture consists of several key components that work together to enable the robot's navigation using ArUco markers:

- **/navigation_03 Node**: This is the central node of our system that:
  - Subscribes to camera images from `/camera/image` topic
  - Processes these images to detect ArUco markers
  - Makes navigation decisions based on detected markers
  - Publishes velocity commands to the `/cmd_vel` topic to control the robot's movement
- **/camera Namespace**: Contains all camera-related topics and data:
  - `/camera/image`: Raw image data from the robot's camera
  - `/camera/camera_info`: Camera calibration parameters needed for pose estimation
  - `/camera/image/camera_info`: Detailed camera information
  - `/camera/image/image_topics`: Additional image-related topics
  - `/camera/image/zstd`: Compressed image data
  - `/camera/depth_image`: Depth information (if using a depth camera)
  - `/camera/points`: Point cloud data (if available)
- **/ros_gz_bridge Node**: This critical node bridges communication between ROS 2 and Gazebo:
  - Converts Gazebo sensor data to ROS messages (e.g., camera images)
  - Converts ROS commands to Gazebo control inputs (e.g., wheel velocities)

  - Handles topics like `/joint_states` for robot state information
  - Manages the `/cmd_vel` topic for robot motion control

- **/ros_gz_image Node**: Specialized bridge for image data:
  - Converts Gazebo camera images to ROS image messages
  - Ensures efficient transfer of high-bandwidth image data
  - Connects directly to the camera image topics

- **/relay_camera_info Node**: Relays camera information:
  - Ensures camera calibration data is available to all nodes that need it
  - Connects `/camera/image/camera_info` to other parts of the system

- **/robot_state_publisher Node**: Publishes the robot's state:
  - Receives joint state information from `/joint_states`
  - Computes and publishes the robot's kinematic state
  - Publishes to `/robot_description` and `/tf` topics
  - The `/tf` topic is crucial for coordinate transformations between different parts of the robot

Data Flow in the System:

1. The Gazebo simulation generates camera images and joint states
2. The `/ros_gz_bridge` and `/ros_gz_image` nodes convert these to ROS messages
3. The `/navigation_03` node receives camera images and processes them to detect ArUco markers
4. When markers are detected, the navigation node calculates the robot's position and determines the next action
5. The navigation node publishes velocity commands to `/cmd_vel`
6. The `/ros_gz_bridge` converts these commands to Gazebo control inputs
7. Gazebo updates the robot's position based on these commands
8. The `/robot_state_publisher` updates the TF tree with the new robot state
9. The cycle repeats as the robot navigates through the environment

This architecture demonstrates the power of ROS's modular design, allowing different components to work together seamlessly. The separation of concerns between perception (camera), decision-making (navigation), and actuation (velocity commands) creates a robust and maintainable system.

7 Processing Images from Camera

Before implementing the navigation algorithm, we need to process the images from the robot's camera to detect ArUco markers. This is a crucial step that enables the robot to perceive its environment and make navigation decisions.

7.1 Image Processing Pipeline

The image processing pipeline involves several steps:

1. Receiving raw camera images from the ROS topic
2. Converting ROS image messages to OpenCV format
3. Processing the images to detect ArUco markers
4. Estimating the pose of detected markers
5. Visualizing the results for debugging

7.2 Implementation

The image processing is implemented in the `process_camera_images.py` file. Here's the main function that demonstrates the core functionality:

```python
def main(args=None):
    print("OpenCV version: %s" % cv2.__version__)

    rclpy.init(args=args)
    node = ProcessCameraImages()

    try:
        node.display_image()  # Run the display loop
    except KeyboardInterrupt:
        pass
    finally:
        node.stop()  # Ensure the spin thread and node stop properly
        node.destroy_node()
        rclpy.shutdown()
```

Listing 7: Camera Image Processing Main Function

The main processing happens in the `ProcessCameraImages` class, which:

- Creates a ROS 2 node that subscribes to camera images
- Uses OpenCV and the ArUco library to process these images
- Detects ArUco markers and estimates their poses
- Visualizes the results in a window

7.3 Key ArUco Functions

The ArUco library provides several key functions that we use for marker detection and pose estimation:

- `aruco.getPredefinedDictionary()`: Selects the marker dictionary to use (we use DICT_4X4_250)
- `aruco.DetectorParameters_create()`: Creates parameters for the marker detector
- `aruco.detectMarkers()`: Detects markers in an image and returns their corners and IDs
- `aruco.estimatePoseSingleMarkers()`: Estimates the 3D pose of markers using camera parameters
- `aruco.drawDetectedMarkers()`: Visualizes detected markers on the image
- `cv2.drawFrameAxes()`: Draws 3D axes to visualize marker poses

7.4 Image Processing Code

Here's a simplified version of the image processing function:

```python
def process_image(self, img):
    """Image processing task."""
    # Convert to grayscale for marker detection
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detect ArUco markers
    corners, ids, rejected = aruco.detectMarkers(gray, aruco_dict,
  parameters=aruco_params)

    # If markers are detected
    if ids is not None:
        # Draw detected markers on the image
        aruco.drawDetectedMarkers(img, corners, ids)

        # Estimate pose of each marker
        rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(
            corners, self.marker_length, self.camera_matrix, self.
  dist_coeffs
        )

        # Draw axes for each detected marker
        for i in range(len(ids)):
            cv2.drawFrameAxes(
                img, self.camera_matrix, self.dist_coeffs,
                rvecs[i], tvecs[i], self.marker_length * 0.5
            )

            # Print marker information
            print(f"Marker ID: {ids[i][0]}, Distance: {tvecs[i
  ][0][2]:.3f} m")

```

```
    return img
```

Listing 8: Image Processing Function

This image processing pipeline is essential for the robot's navigation, as it provides the necessary information about marker positions and orientations. The navigation algorithm then uses this information to make decisions about the robot's movement.

8 Pseudocode

Algorithm 1 Robot Navigation Using ArUco Markers

```
1: Initialize robot and camera parameters
2: Define marker dictionary and detection parameters
3: Define navigation plan (marker IDs and corresponding actions)
4: Set initial target marker ID and action
5: while robot is running do
6:     Capture camera image
7:     Convert image to grayscale
8:     Detect ArUco markers in the image
9:     if markers detected then
10:        Draw detected markers on the image for visualization
11:        Estimate pose of each marker
12:        if target marker is detected then
13:            Calculate distance to target marker
14:            Calculate marker center position in image
15:            if current action is "turn_cw" or "turn_ccw" then
16:                if marker is centered in image then
17:                    Change action to "move"
18:                end if
19:            else if current action is "move" then
20:                if distance to marker is small enough then
21:                    Update target marker ID and action based on navigation plan
22:                end if
23:            end if
24:        end if
25:    end if
26:    Set robot velocity based on current action:
27:    if action is "move" then
28:        Set linear velocity forward
29:    else if action is "turn_cw" then
30:        Set angular velocity clockwise
31:    else if action is "turn_ccw" then
32:        Set angular velocity counter-clockwise
33:    else if action is "idle" then
34:        Stop all movement
35:    end if
36:    Publish velocity command
37: end while
```

9 Navigation Algorithm

The navigation algorithm is implemented in the `navigation_03.py` file. Here are some key components of the implementation:

9.0.1 Marker Detection and Pose Estimation

```python
def process_image(self, img):
    """Image processing task."""
    gray = cv2.cvtColor(self.latest_frame, cv2.COLOR_BGR2GRAY)
    corners, ids, rejected = aruco.detectMarkers(gray, aruco_dict, parameters=aruco_params)

    if ids is not None:
        aruco.drawDetectedMarkers(self.latest_frame, corners, ids)
        rvecs, tvecs, * = aruco.estimatePoseSingleMarkers(corners, self.marker_length, self.camera_matrix, self.dist_coeffs)

        for i in range(len(ids)):
            cv2.drawFrameAxes(self.latest_frame, self.camera_matrix, self.dist_coeffs, rvecs[i], tvecs[i], self.marker_length * 0.5)
```

Listing 9: Marker Detection and Pose Estimation

9.0.2 Navigation Logic

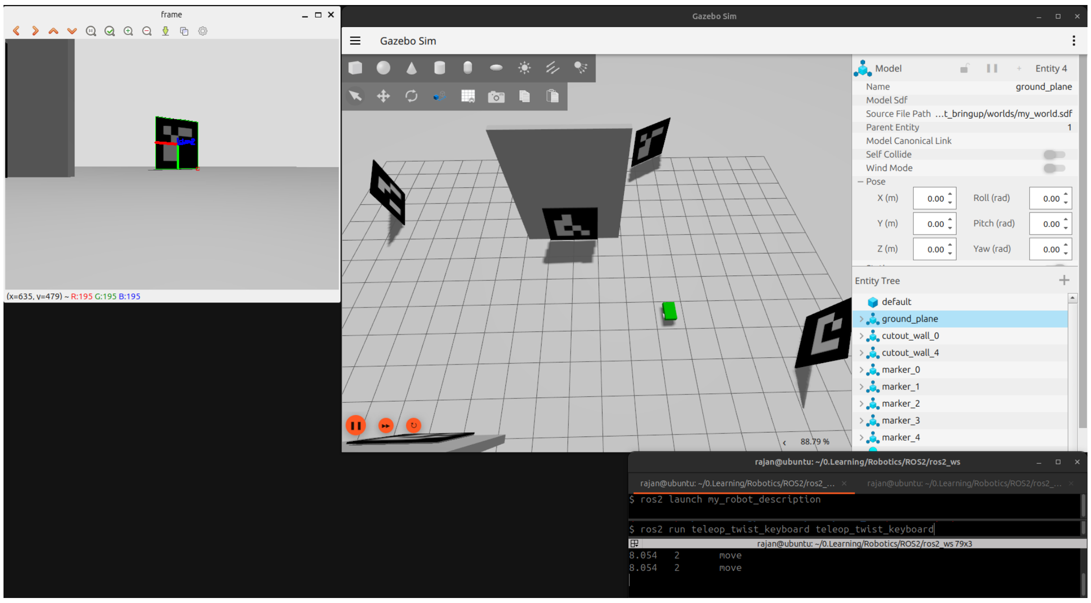
```python
if ids is not None:
    ids_flattened = ids.flatten().tolist()
    requred_id = aruco_data[self.target_id]["next_id"]  # next id to look for
    marker_center = [0, 0]  # this is just to avoid error
    if(requred_id in ids_flattened):
        index = ids_flattened.index(requred_id)
        print(f"{round(tvecs[index][0][2], 3):<6} {aruco_data[self.target_id]['next_id']:>2} {self.target_action:>10}")
        marker_center = self.get_marker_center_on_image(rvecs[index][0], tvecs[index][0])
        if self.target_action in ["turn_cw", "turn_ccw"]:
            # get marker to center of image or directly in front of the camera
            if marker_center[0] > 310 and marker_center[0] < 330:
                self.target_action = "move"
        elif self.target_action == "move":
            # if we are close enough to the marker, then move to the next id and it's action
            if tvecs[index][0][2] <= 4:
                self.target_id = aruco_data[self.target_id]["next_id"]
                self.target_action = aruco_data[self.target_id]["next_action"]
```

Listing 10: Navigation Decision Logic

10 Simulation and Results

10.1 Gazebo Simulation Environment

The simulation environment is set up in Gazebo, with ArUco markers placed at strategic locations. The robot navigates through this environment by detecting and following the markers.


frame
(x=635, y=479) ~ R:195 G:195 B:195
Gazebo Sim
Gazebo Sim
88.79 %
rajan@ubuntu: ~/0.Learning/Robotics/ROS2/ros2_ws
rajan@ubuntu: ~/0.Learning/Robotics/ROS2/ros2_...
rajan@ubuntu: ~/0.Learning/Robotics/ROS2/ros2_...
rajan@ubuntu: ~/0.Learning/Robotics/ROS2/ros2_ws 79x3

Model | Entity 4
Name: ground_plane
Model Sdf
Source File Path: ...t_bringup/worlds/my_world.sdf
Parent Entity: 1
Model Canonical Link
Self Collide
Wind Mode
Pose
X (m) 0.00 | Roll (rad) 0.00
Y (m) 0.00 | Pitch (rad) 0.00
Z (m) 0.00 | Yaw (rad) 0.00

Entity Tree
- default
- ground_plane
- cutout_wall_0
- cutout_wall_4
- marker_0
- marker_1
- marker_2
- marker_3
- marker_4

```
$ ros2 launch my_robot_description
```

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

```
8.054   2       move
8.054   2       move
```

Figure 2: Gazebo simulation environment with ArUco markers and robot

In the screenshot above, we can see:

- The simulated environment with ArUco markers placed on walls
- The robot (small green cube) in the environment
- The Gazebo interface showing the simulation controls and entity tree
- The camera view (left panel) showing what the robot's camera sees, including detected markers

10.2 Navigation Performance

The robot successfully navigates through the environment by:

1. Detecting ArUco markers using its camera
2. Determining its position relative to the markers
3. Following the predefined navigation plan
4. Moving from one marker to the next until reaching the destination

The navigation algorithm handles different scenarios:

- When a marker is not in view, the robot turns to search for it

- When a marker is detected but not centered, the robot adjusts its orientation
- When a marker is centered and the robot is close enough, it moves to the next marker in the sequence

11 Demo and Source Code

Below you'll find the source code for this project and a live demonstration video.

- **GitHub Repository:** Robot Navigation
- **Demo Video:** Click here to watch the demo on YouTube

12 Conclusion

This project successfully demonstrates autonomous robot navigation using ArUco markers in a simulated environment. The implementation leverages ROS 2 for communication, Gazebo for physics simulation, and OpenCV for computer vision tasks.

Key achievements:

- Implemented a complete navigation system using visual markers
- Demonstrated effective integration of ROS 2, Gazebo, and OpenCV
- Created a modular and extensible architecture
- Achieved reliable marker detection and pose estimation
- Implemented a robust navigation algorithm

Future improvements could include:

- Adding obstacle avoidance capabilities
- Implementing more sophisticated path planning
- Extending to multi-robot scenarios
- Testing in more complex environments
- Deploying on physical robots