MTE 431

Mobile Robots

Fall 2023

Design and Control of an Omnidirectional Mobile Robot

Submitted by

Mostafa Ahmed	120200214
Mohamed Khaled	120210346
Omar Elshamly	120190093
Osama Khalil	120210051
Radwa Waheed	120200228

Submitted to

Prof. Haitham El-Hussieny

Table of Contents

Table of Figures

1. Introduction

Omnidirectional mobile robots represent a groundbreaking advancement in robotics, offering unparalleled agility and maneuverability. These robots are designed with wheels or mechanisms that allow them to move effortlessly in any direction, providing the ability to navigate complex environments with ease. Unlike traditional robots limited to forward and backward movements, omnidirectional robots can move forward, backward, sideways, and rotate seamlessly, making them ideal for applications requiring precise and dynamic motion control. Their versatility finds applications in various industries, including logistics, manufacturing, healthcare, and research, where their unique mobility capabilities contribute to increased efficiency, flexibility, and adaptability in performing tasks. This project seeks to enable the control of an omnidirectional mobile robot through a joystick using ROS2 and Arduino integration.

2. Robot Design and Kinematic Model

2.1. Design

The robot was implemented with four Mecanum wheels connected to the robot platform as shown in figure (1). The wheels are attached to both sides having the rollers oriented at angles of + 45^o and - 45^o.

2.2. Kinematic Model

To consider the kinematic model, it is assumed that the robot is placed on a plane surface where (O, x, y) is the inertial reference frame and (G, x_R, y_R) is a local coordinate frame fixed on the robot at its center of mass, which coincides with its geometric center G.

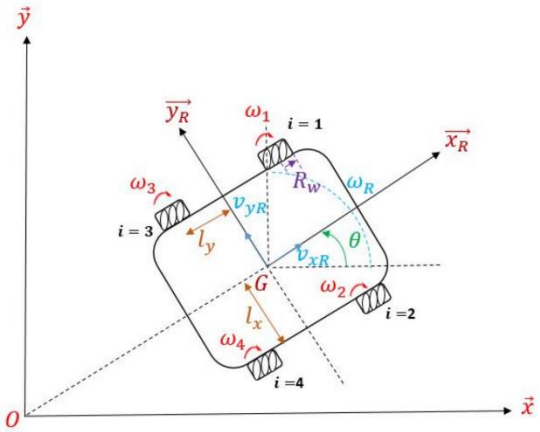

Figure (2): 4-Mecanum Wheeled Robot Geometry.

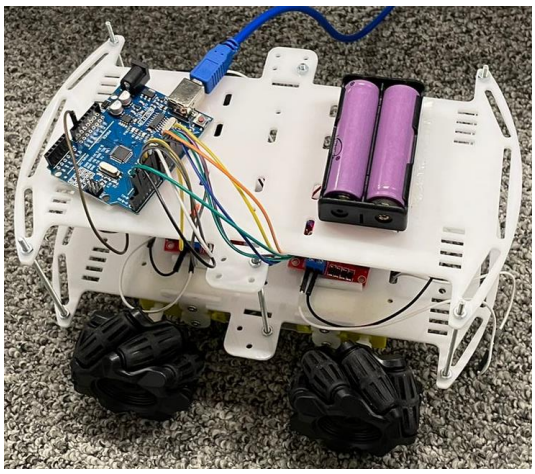

Figure (1): Robot Assembly.

The alpha, beta, and gamma angles were computed from the SolidWorks design as illustrated in figure (3).

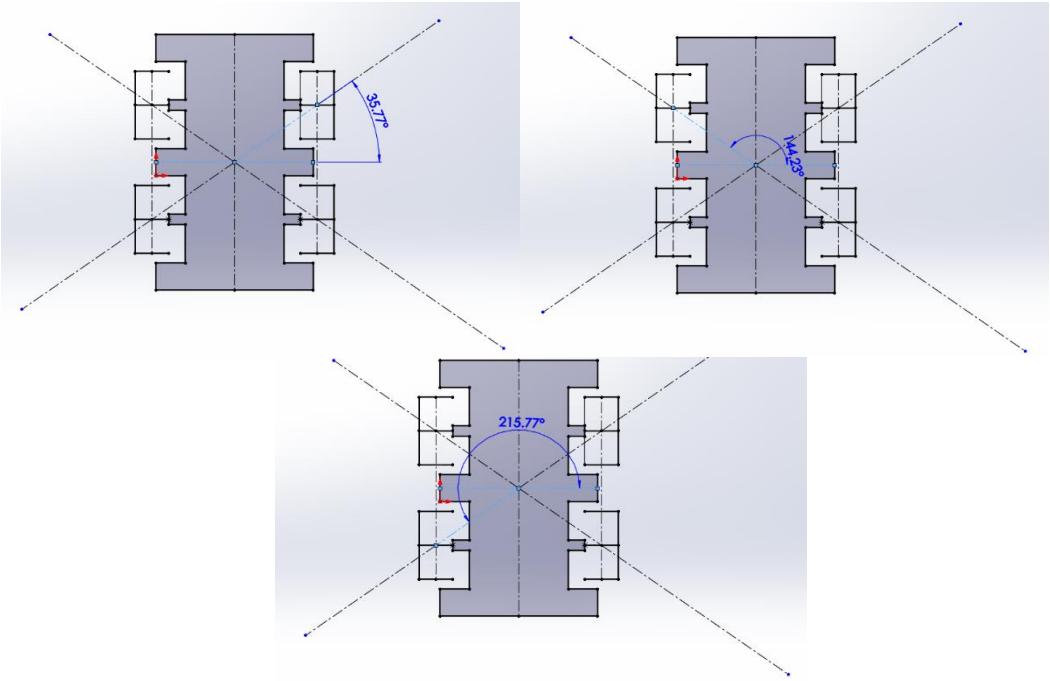

Figure (3): Alpha, Beta, and Gamma Measurements.

	alpha	**beta**	**gamma**
Left Front Wheel	35.77	144.23	45
Right Front Wheel	144.23	35.77	-45
Right Back Wheel	215.77	35.77	45
Left Back Wheel	324.23	-144.23	-45

Table (1): Alpha, Beta, and Gamma Values.

The inverse kinematic model is given by:

$$\boldsymbol{\omega = J_2^{-1}\, J_1\, \varepsilon' R}$$

Where ω is the wheels speeds matrix, J_2^{-1} is the radius matrix, J_1 is the rolling constrains matrix, and $\varepsilon' R$ is the robot speeds with respect to the robot frame.

The J_2^{-1}, J_1, $\varepsilon' R$ matrices are given in the following code to compute the wheels speeds continuously:

```
# Calculate motor speeds
robot_frame_speed = np.array([[vx], [vy], [vtheta]], dtype=float)

alpha = np.array([np.deg2rad(35.77), np.deg2rad(144.23), np.deg2rad(215.77),
np.deg2rad(324.23)], dtype=float)
```

```
beta = np.array([np.deg2rad(144.23), np.deg2rad(35.77), np.deg2rad(-35.77),
np.deg2rad(-144.23)], dtype=float)

gamma = np.array([np.deg2rad(45), np.deg2rad(-45), np.deg2rad(45), np.deg2rad(-
45)], dtype=float)

r = 0.0325
L = 0.0937

rolling_constraints = np.array([
[np.sin(alpha[0] + beta[0] + gamma[0]), -np.cos(alpha[0] +  beta[0] + gamma[0]),
-L * np.cos(beta[0] + gamma[0])],
[np.sin(alpha[1] + beta[1] + gamma[1]), -np.cos(alpha[1] + beta[1] + gamma[1]), -
L * np.cos(beta[1] + gamma[1])],
[np.sin(alpha[2] + beta[2] + gamma[2]), -np.cos(alpha[2] + beta[2] + gamma[2]), -
L * np.cos(beta[2] + gamma[2])],
[np.sin(alpha[3] + beta[3] + gamma[3]), -np.cos(alpha[3] + beta[3] + gamma[3]), -
L * np.cos(beta[3] + gamma[3])],
], dtype=float)

radius_matrix = np.array([
                [r * np.cos(gamma[0]), 0, 0, 0],
                [0, r * np.cos(gamma[1]), 0, 0],
                [0, 0, r * np.cos(gamma[2]), 0],
                [0, 0, 0, r * np.cos(gamma[3])]
                ], dtype=float)

motor_speeds = np.linalg.inv(radius_matrix) @ rolling_constraints @
robot_frame_speed
```

2.3. The Main Directions of Movement of the Robot

The robot's main directions of movement, together with the indication of the rotation of each wheel, are shown in figure (4). For example, the forward motion is achieved when all four wheels are driven in the same direction – forward (Fig. 4 a).

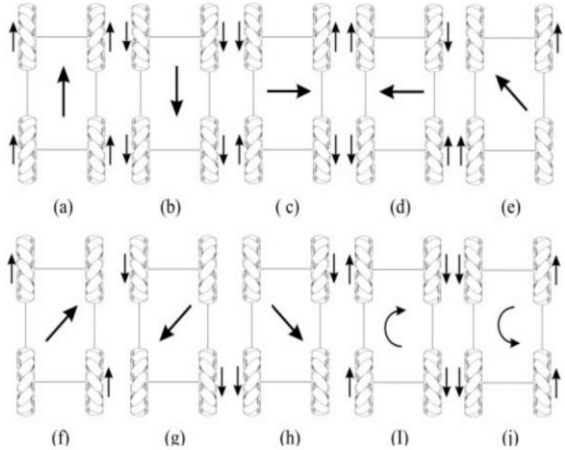

Figure (4): The Main Locomotion Direction of The Robot and The Way They are Achieved.

3. Hardware Setup

3.1. Components

Component	Quantity
Arduino UNO	1
Mecanum Wheel	4
Motor Driver, L298	2
Battery Holder	1
Lithium Battery	2

3.2. Hardware Connection

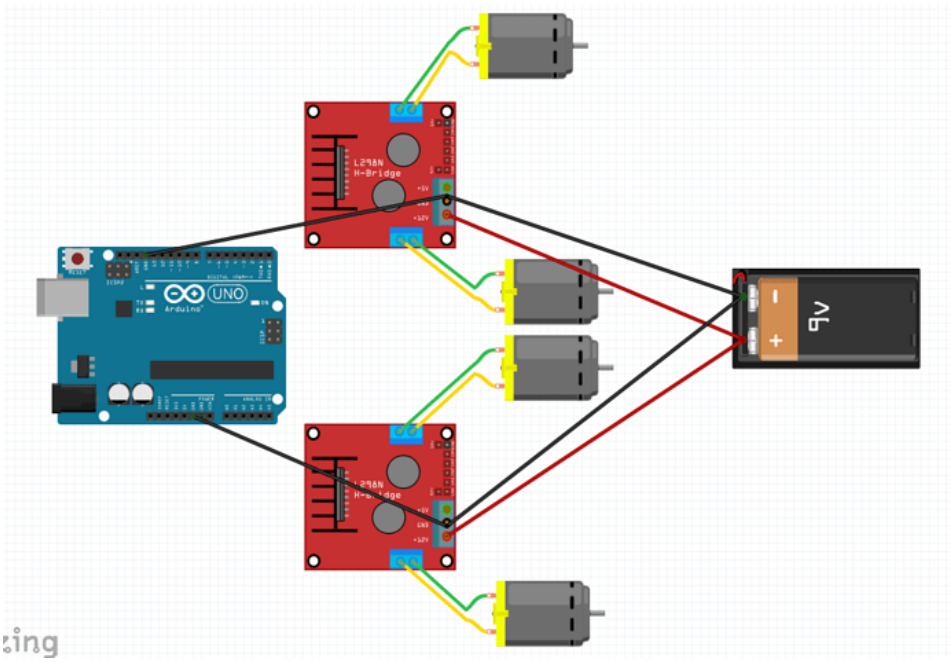

Figure (5): Hardware Connection.

4. Software Development

4.1. Building the URDF

We started by constructing the robot_core.xacro file, subsequently, the gazebo_control file. The gazebo_control_omni.xacro is a plugin that allows the robot to move in omni-directions inside gazebo.

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

    <xacro:include filename="robot_core.xacro" />
    <xacro:include filename="gazebo_control_omni.xacro" />
    <xacro:include filename="colors.xacro" />

</robot>
```

Figure (6): robot.urdf.xacro.

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

    <gazebo>
<plugin name="object_controller" filename="libgazebo_ros_planar_move.so">


      <!-- Set control loop update rate -->
      <update_rate>100</update_rate>
      <!-- Set odom publish rate -->
      <publish_rate>10</publish_rate>

      <!-- Set if odom required -->
      <publish_odom>true</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>

      <!-- Frame IDs -->
      <odometry_frame>odom</odometry_frame>
      <robot_base_frame>base_link</robot_base_frame>

      <!-- Set odom covariance -->
      <covariance_x>0.0001</covariance_x>
      <covariance_y>0.0001</covariance_y>s
      <covariance_yaw>0.01</covariance_yaw>

     </plugin>

    </gazebo>


</robot>
```

Figure (7): gazebo_control_omni.xacro.

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

<!-- BASE LINK -->
<link name="base_link">
</link>

<!-- BASE_FOOTPRINT LINK -->
<joint name="base_footprint_joint" type="fixed">
  <parent link="base_link"/>
  <child link="base_footprint"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>

<link name="base_footprint">
</link>

<!-- Chassis -->
<joint name="chassis_joint" type="fixed">
  <parent link="base_footprint"/>
  <child link="chassis"/>
</joint>

<link name="chassis">
  <visual>
    <geometry>
      <box size="0.5 0.3 0.2"/>
    </geometry>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <box size="0.5 0.3 0.2"/>
    </geometry>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
  </collision>
  <inertial>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <mass value="0.5"/>
    <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1"/>
  </inertial>
</link>
<gazebo reference="chassis">
  <material>Gazebo/Blue</material>
</gazebo>
```

Figure (8): robot_core.xacro.

Additionally, the joystick node was implemented to navigate the robot in gazebo, publishing the robot velocities on /cmd_vel topic.

```yaml
joy_node:
  ros__parameters:
    device_id: 0
```

```
    deadzone: 0.05
    autorepeat_rate: 20.0

teleop_node:
  ros__parameters:
    axis_linear:
      x: 3
    scale_linear:
      x: -3.5
    scale_linear_turbo:
      x: -50.0

    axis_linear:
      y: 1
    scale_linear:
      y: 3.5
    scale_linear_turbo:
      y: 50.0

    axis_angular:
      yaw: 0
    scale_angular:
      yaw: 3.5
    scale_angular_turbo:
      yaw: 50.0


    enable_button: 7
    enable_turbo_button: 6

    require_enable_button: true
```

4.2. Publisher and Subscriber Node

This node subscribes to the /cmd_vel topic to receive velocity commands. Upon receiving a Twist message (linear and angular velocities), it calculates the corresponding motor speeds based on mecanum wheel kinematics. The calculated motor speeds are then published on the motor_velocity topic as a new Twist message to Arduino. The script continuously spins, handling incoming messages and updating the motor velocities accordingly. If an error occurs during message processing, it logs a warning message.

```
import rclpy
from geometry_msgs.msg import Twist
import numpy as np

class MinimalVelPubSubNode:
    def __init__(self):
        rclpy.init()
```

```
        self.node = rclpy.create_node('minimal_vel_pub_sub')
        self.subscription = self.node.create_subscription(Twist, '/cmd_vel',
self.chatter_callback, 10)
        self.publisher = self.node.create_publisher(Twist, 'motor_velocity', 10)

    def chatter_callback(self, msg):
        try:
            # Extract linear x, linear y, and angular z velocities
            vx = msg.linear.x
            vy = msg.linear.y
            vtheta = msg.angular.z

            # Calculate motor speeds
            robot_frame_speed = np.array([[vx], [vy], [vtheta]], dtype=float)
            alpha = np.array([np.deg2rad(35.77), np.deg2rad(144.23),
np.deg2rad(215.77), np.deg2rad(324.23)], dtype=float)
            beta = np.array([np.deg2rad(144.23), np.deg2rad(35.77), np.deg2rad(-
35.77), np.deg2rad(-144.23)], dtype=float)
            gamma = np.array([np.deg2rad(45), np.deg2rad(-45), np.deg2rad(45),
np.deg2rad(-45)], dtype=float)
            r = 0.0325
            L = 0.0937
            rolling_constraints = np.array([
                [np.sin(alpha[0] + beta[0] + gamma[0]), -np.cos(alpha[0] +
beta[0] + gamma[0]), -L * np.cos(beta[0] + gamma[0])],
                [np.sin(alpha[1] + beta[1] + gamma[1]), -np.cos(alpha[1] +
beta[1] + gamma[1]), -L * np.cos(beta[1] + gamma[1])],
                [np.sin(alpha[2] + beta[2] + gamma[2]), -np.cos(alpha[2] +
beta[2] + gamma[2]), -L * np.cos(beta[2] + gamma[2])],
                [np.sin(alpha[3] + beta[3] + gamma[3]), -np.cos(alpha[3] +
beta[3] + gamma[3]), -L * np.cos(beta[3] + gamma[3])],
            ], dtype=float)
            radius_matrix = np.array([
                [r * np.cos(gamma[0]), 0, 0, 0],
                [0, r * np.cos(gamma[1]), 0, 0],
                [0, 0, r * np.cos(gamma[2]), 0],
                [0, 0, 0, r * np.cos(gamma[3])]
            ], dtype=float)
            motor_speeds = np.linalg.inv(radius_matrix) @ rolling_constraints @
robot_frame_speed

            # Create and publish Twist message with calculated motor speeds
            motor_twist = Twist()
            motor_twist.linear.x = motor_speeds[0, 0]
            motor_twist.linear.y = motor_speeds[1, 0]
            motor_twist.linear.z = motor_speeds[2, 0]
            motor_twist.angular.z = motor_speeds[3, 0]
```

```
            self.publisher.publish(motor_twist)

            self.node.get_logger().info('Published Motor Speeds: %s' %
motor_twist)
        except Exception as e:
            self.node.get_logger().warn('Error processing cmd_vel message: %s' %
str(e))

    def spin(self):
        rclpy.spin(self.node)

    def destroy_node(self):
        self.node.destroy_node()
        rclpy.shutdown()

def main(args=None):
    node = MinimalVelPubSubNode()
    node.spin()
    node.destroy_node()

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

4.3. Arduino Code

This code subscribes to the /motor_velocity topic to receive velocity commands. Upon receiving Twist messages, it calculates motor speeds based on mecanum wheel kinematics and adjusts each wheel's motion accordingly. The motors are controlled using PWM signals, allowing the robot to move in various directions. The code continuously loops, handling incoming messages and controlling the robot's movement.

```
#include <ros.h>
#include <geometry_msgs/Twist.h>

ros::NodeHandle  nh;     // Initialize ROS node handle

// Define motor pin configurations
const int motor_left_front = 5;
const int motor_right_front = 6;
const int motor_left_back = 10;
const int motor_right_back = 11;
int motorA1 = 3;
int motorA2 = 4;
int motorB1 = 8;
int motorB2 = 7;
int motorC1 = 9;
```

```
int motorC2 = 12;
int motorD1 = 2;
int motorD2 = 13;

// Callback function for receiving Twist messages
void cmdcallback(const geometry_msgs::Twist& twist){
    int speed_left_front = float(twist.linear.x);
    int speed_right_front = float(twist.linear.y);
    int speed_right_back = float(twist.linear.z);
    int speed_left_back = float(twist.angular.z);

    // Constrain speeds to ensure they are within valid range
    speed_left_front = constrain(speed_left_front, -255, 255);
    speed_right_front = constrain(speed_right_front, -255, 255);
    speed_right_back = constrain(speed_right_back, -255, 255);
    speed_left_back = constrain(speed_left_back, -255, 255);


      // Motor control logic for each wheel based on speed direction
      if (speed_left_front > 0) {
        digitalWrite(motorA1, LOW);
        digitalWrite(motorA2, HIGH);
      }
      else if (speed_left_front < 0){
        digitalWrite(motorA1, HIGH);
        digitalWrite(motorA2, LOW);
      }
      else {
        digitalWrite(motorA1, LOW);
        digitalWrite(motorA2, LOW);
      }


      if (speed_right_front > 0){
        digitalWrite(motorB1, LOW);
        digitalWrite(motorB2, HIGH);
      }
      else if (speed_right_front < 0){
        digitalWrite(motorB1, HIGH);
        digitalWrite(motorB2, LOW);
      }
      else {
        digitalWrite(motorB1, LOW);
        digitalWrite(motorB2, LOW);
      }


      if (speed_left_back > 0 ){
```

```
        digitalWrite(motorC1, LOW);
        digitalWrite(motorC2, HIGH);
      }
      else if (speed_left_back < 0){
        digitalWrite(motorC1, HIGH);
        digitalWrite(motorC2, LOW);
      }
      else {
        digitalWrite(motorC1, LOW);
        digitalWrite(motorC2, LOW);
      }


      if (speed_left_back > 0){
        digitalWrite(motorD1, LOW);
        digitalWrite(motorD2, HIGH);
      }
      else if (speed_left_back < 0){
        digitalWrite(motorD1, HIGH);
        digitalWrite(motorD2, LOW);
      }
      else {
        digitalWrite(motorD1, LOW);
        digitalWrite(motorD2, LOW);
      }

    // Set motor speeds using PWM
    analogWrite(motor_left_front, abs(speed_left_front));
    analogWrite(motor_right_front, abs(speed_right_front));
    analogWrite(motor_left_back, abs(speed_left_back));
    analogWrite(motor_right_back, abs(speed_right_back));
}

ros::Subscriber<geometry_msgs::Twist> sub("/motor_velocity", &cmdcallback);


void setup()
{
    // Set pin modes for motors
    pinMode(motor_left_front, OUTPUT);
    pinMode(motor_right_front, OUTPUT);
    pinMode(motor_left_back, OUTPUT);
    pinMode(motor_right_back, OUTPUT);
    pinMode(motorA1, OUTPUT);
    pinMode(motorA2, OUTPUT);
    pinMode(motorB1, OUTPUT);
    pinMode(motorB2, OUTPUT);
    pinMode(motorC1, OUTPUT);
```

```
    pinMode(motorC2, OUTPUT);
    pinMode(motorD1, OUTPUT);
    pinMode(motorD2, OUTPUT);

  // Initialize ROS node and subscribe to Twist messages
  nh.initNode();
  nh.subscribe(sub);
}


void loop()
{
  // Spin ROS node and handle incoming messages
  nh.spinOnce();
  delay(1);
}
```

4.4. ROS 2 to Arduino Communication

In order for ROS 2 to communicate with the Arduino, a bridge between ROS 1 and ROS 2 was stablished via using ros1-bridge package. The ros1-bridge is a ROS 2 package that provides nodes to bridge topics between ROS 1 and ROS 2. Consequently, using rosserial_arduino package directly, which provides a communication bridge between the ROS framework and microcontrollers, particularly Arduino-based systems.

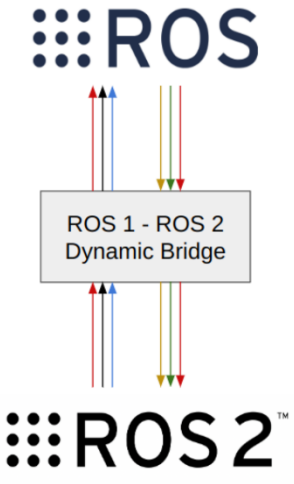

Figure (9): ROS 2 to ROS 1 Bridge.

5. References

[1] Design and development of an autonomous omni-directional mobile robot ... (n.d.). https://www.researchgate.net/publication/269294739_Design_and_development_of_an_autonomous_omni-directional_mobile_robot_with_Mecanum_wheels

[2] Joshnewans. (n.d.). Joshnewans/articubot_one. GitHub. https://github.com/joshnewans/articubot_one

[3] Ros 1 - Ros 2 Bridge. Robotics Knowledgebase. (2023, May 3). https://roboticsknowledgebase.com/wiki/interfacing/ros1_ros2_bridge/