{Learn, Create, Innovate};

Mobile Robotics

Open Loop Control

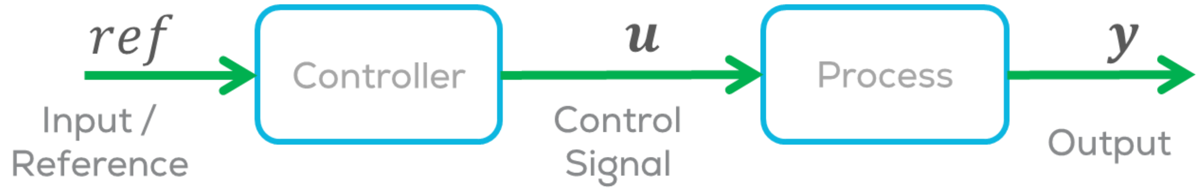
MCR²

Manchester **Robotics**

Open Loop Control

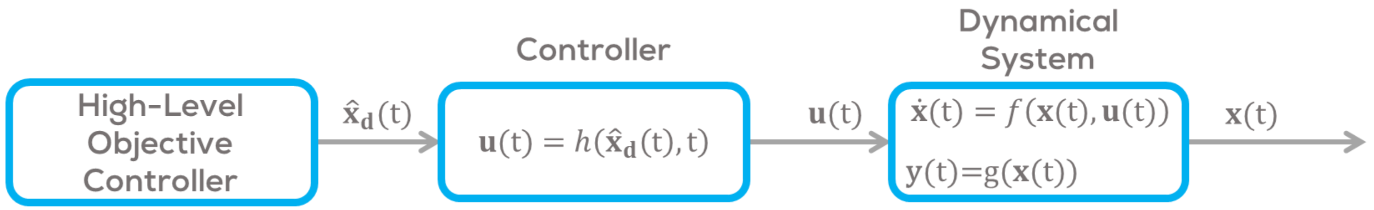
- Open Loop Control is a type of control where the output is not measured or feedback for correction.

- In other words, Open Loop Control System is a system in which the control action is independent of the output of the system.



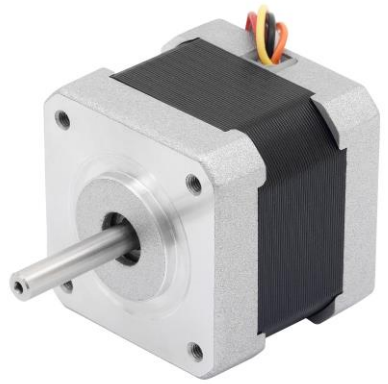
Open Loop Control

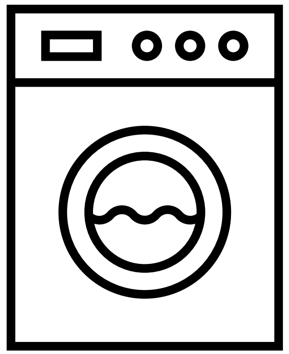
- In this type of control, the output is regulated by varying the input or reference.
- The output of the system is determined by the current state of the system and the inputs that are received from the controller.


High-Level Objective Controller
$\hat{\mathbf{x}}_{\mathbf{d}}(t)$
Controller
$\mathbf{u}(t) = h(\hat{\mathbf{x}}_{\mathbf{d}}(t), t)$
$\mathbf{u}(t)$
Dynamical System
$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t))$
$\mathbf{y}(t) = g(\mathbf{x}(t))$
$\mathbf{x}(t)$

Open Loop Control

- Some examples of this type of control systems are:
  - Windows, window blinds,
  - washing machines,
  - microwave ovens,
  - hair drier, bread toaster, door lock systems,
  - some stepper motors,
  - turning on/off lights,
  - some remote-controlled applications, etc.



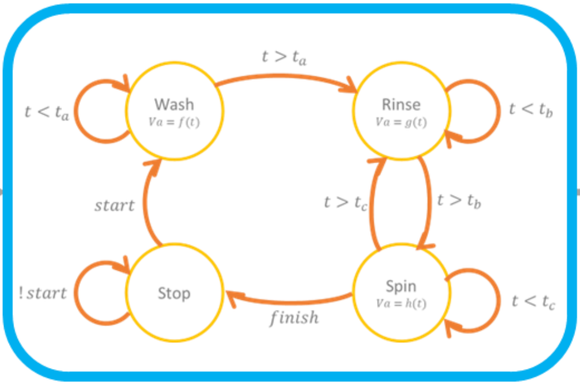


Open Loop Control

Pre-Programmed State Machine

Washing Machine Loaded

User Input (cycle selection)



$V_a(t)$

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = (GK_m x_3 - mga\cos(x_1))\left(\dfrac{1}{G^2 J_m + J + ma^2}\right) \\ \dot{x}_3 = \dfrac{V_a}{L} - \dfrac{K_b G}{L}x_2 - \dfrac{R}{L}x_3 \end{cases}$$

$$y(t) = [0\ 1\ 0]\cdot\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$y(t)$

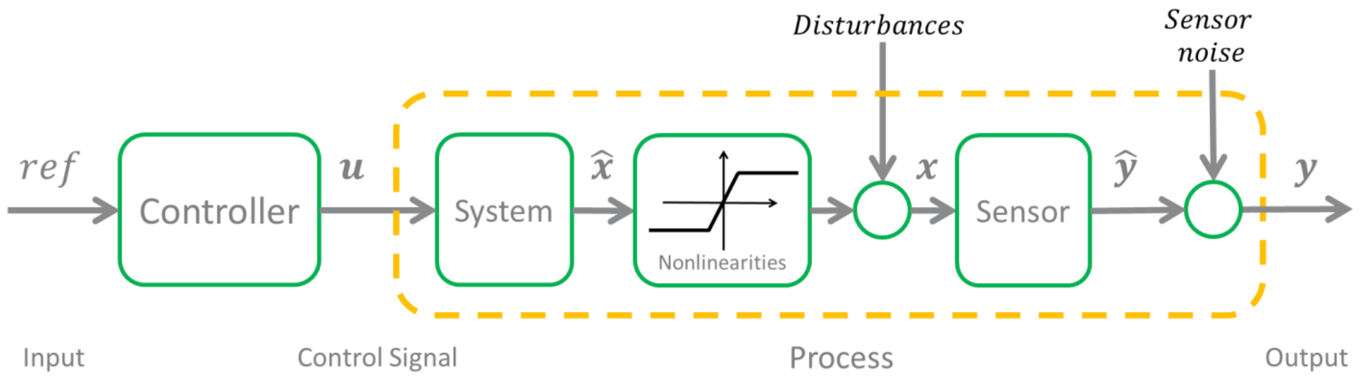
Where:

- $x_1 = \theta_L$: Drum angular Position
- $x_2 = \dot{\theta}_L$: Drum angular Velocity
- $x_3 = i_a$: Motor Armature Current
- L, R: Motor's Armature Inductance and Resistance
- K_b, K_m: Motor's Electromagnetic Constants
- V_a: Motor input voltage
- G: Gear Ratio
- g: Gravity
- a: Radius to the centre of mass
- m: Load mass
- J_m: Inertia of the motor
- J: Inertia of the Drum

Open Loop Control

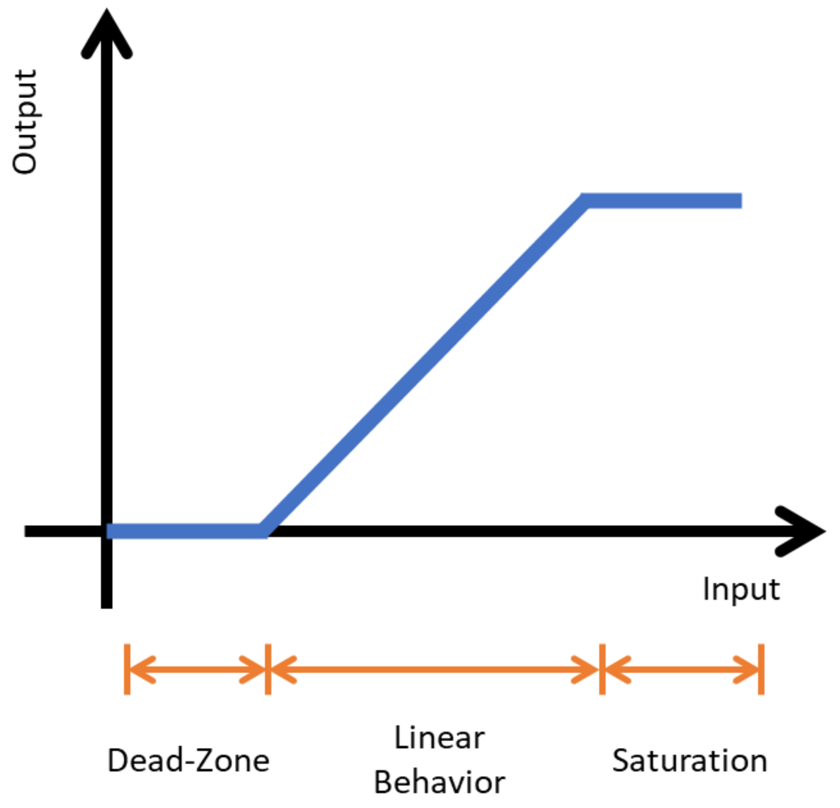
- Every process, present disturbances, nonlinearities and noise.
- Disturbances and noise come from the environment that surrounds the system.
  - Friction, slippage, obstacles on the environment, different types of environment, etc.
  - The system may change over time, for example a valve that accumulates rust over time.

$$\dot{x} = f(x(t), t)$$



Open Loop Control

- Nonlinearities are an intrinsic characteristic of a system in which the output does not linearly follow the input (output not proportional to the input).
    - For the case of the of most systems such as motors, or robots, the nonlinear behavior can be seen as a saturation and dead-zone in the motors and the robot itself.



Open Loop Control

Advantages

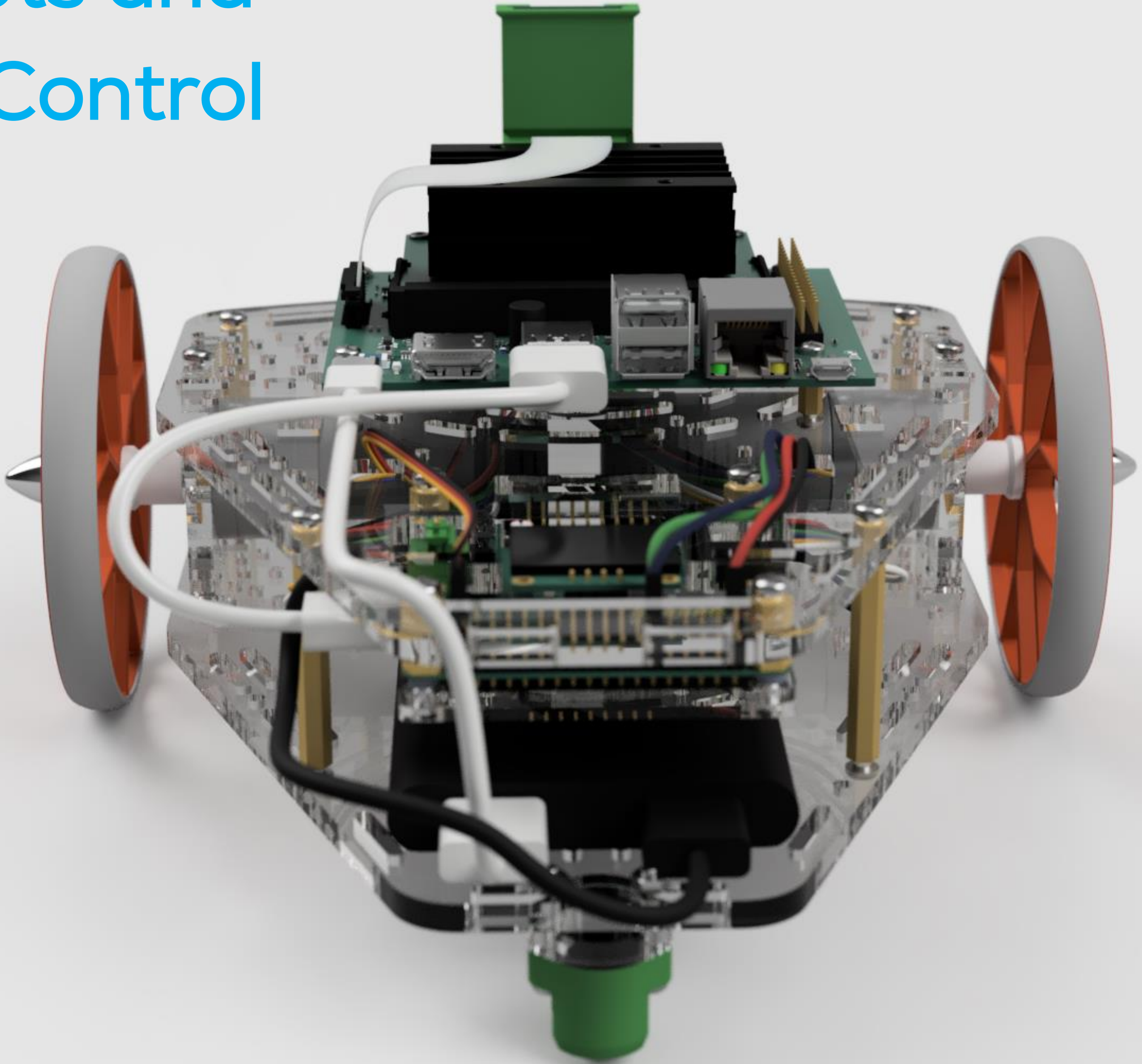
- Simple to design and implement, provided that the user has some experience with the system.
- The cost for design, implement and maintain is relatively low compared with other controllers.
- Maintenance is considered simple, no high technical level required (for most of the controllers).
- The behavior of the controller is quite stable, provided that the system is in a controlled environment, such that the process does not present big disturbances, or the process is not dangerous when unsupervised.

Open Loop Control

Disadvantages:

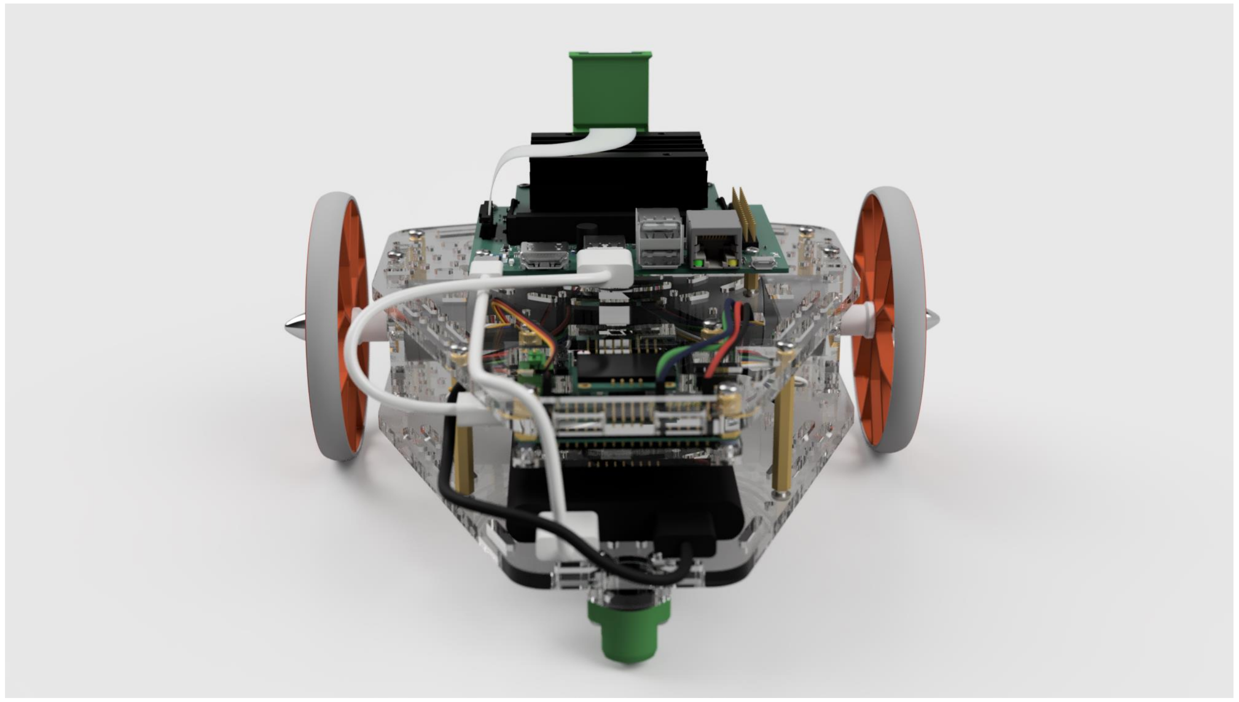
- This type of control is not robust against disturbances (cannot correct the output in the presence of a disturbance).
- An open-loop system has no self-regulation or control action over the output value.
- The input and controller depend on the experience of the user.
- Requires re-calibration often.
- Prone to errors in the output and control signal.
- Does not take into consideration changes on the process over time.
- Also, there is no chance to correct the transition errors in open loop systems so there is more chance to occur errors.

Mobile Robots and Open Loop Control



Mobile Robots and Open Loop Control

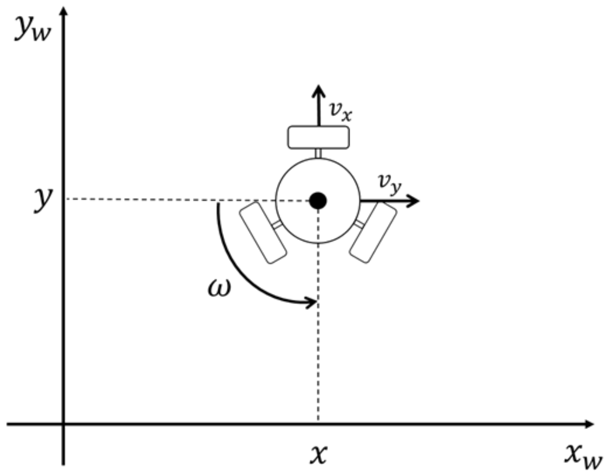
- Open loop control is commonly used in basic mobile robots' tasks.
- Repetitive tasks such as moving goods from one position to another, simple trajectory following algorithms, cleaning, etc.
- Some challenges when using this type of controllers are usually due to Slippage, wheel misalignment, and environmental variations.



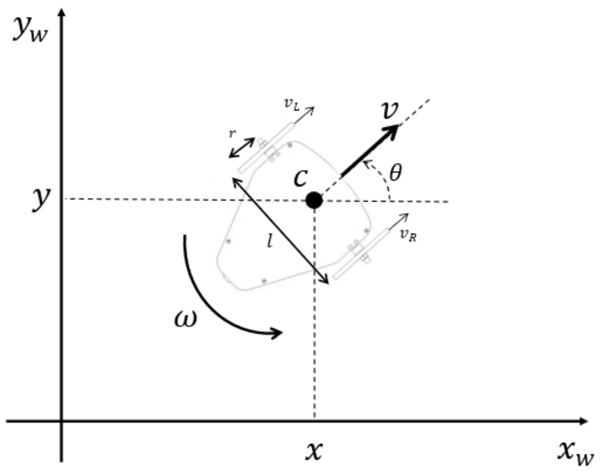
Mobile Robots

Mobile robots can be classified as "Holonomic" or "Nonholonomic“.

- This classification depends on the relationship between controllable and total degrees of freedom of a robot.
- Holonomic Robots: If the controllable degree of freedom is equal to total degrees of freedom, then the robot
- Nonholonomic Robots: If the controllable degree of freedom is less than the total degrees of freedom. Such systems are therefore called underactuated. Differential Drive Systems fall into this category.



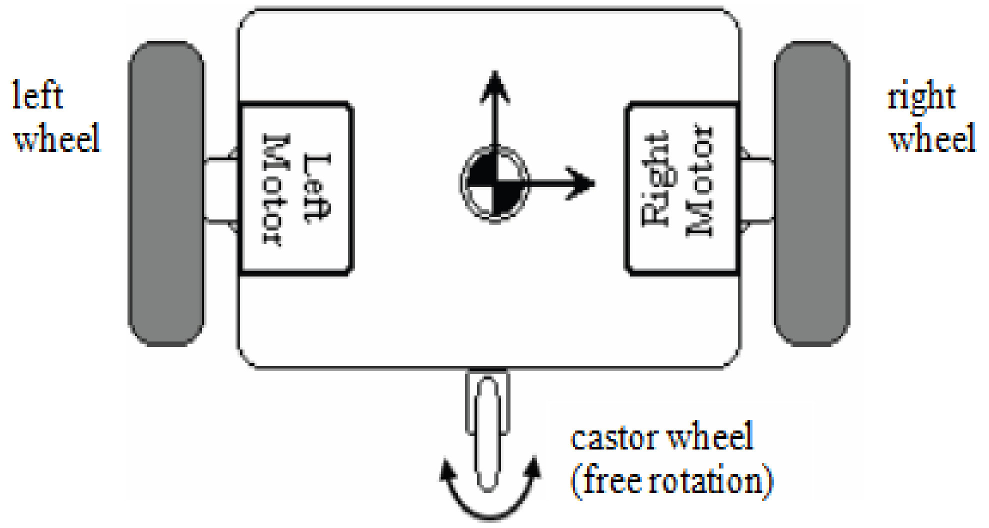
Holonomic Robot



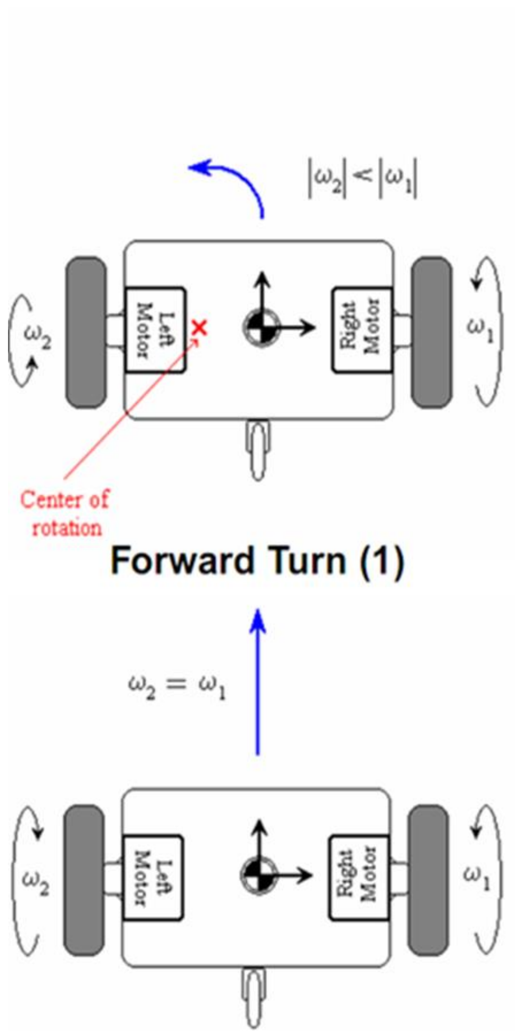
Nonholonomic Robot

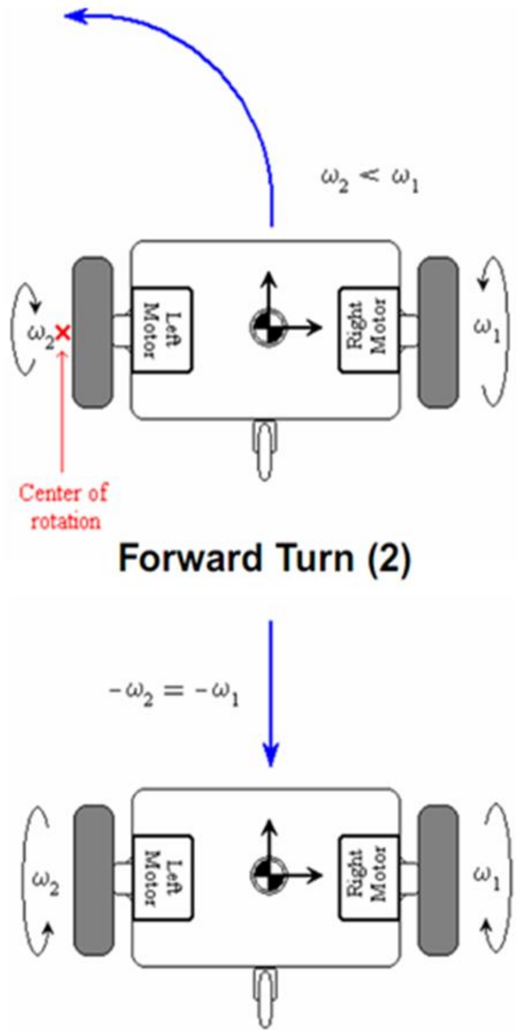
Differential drive robots

- Also known as differential wheeled robots, these are mobile robots whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels, thereby requiring no additional steering motion.



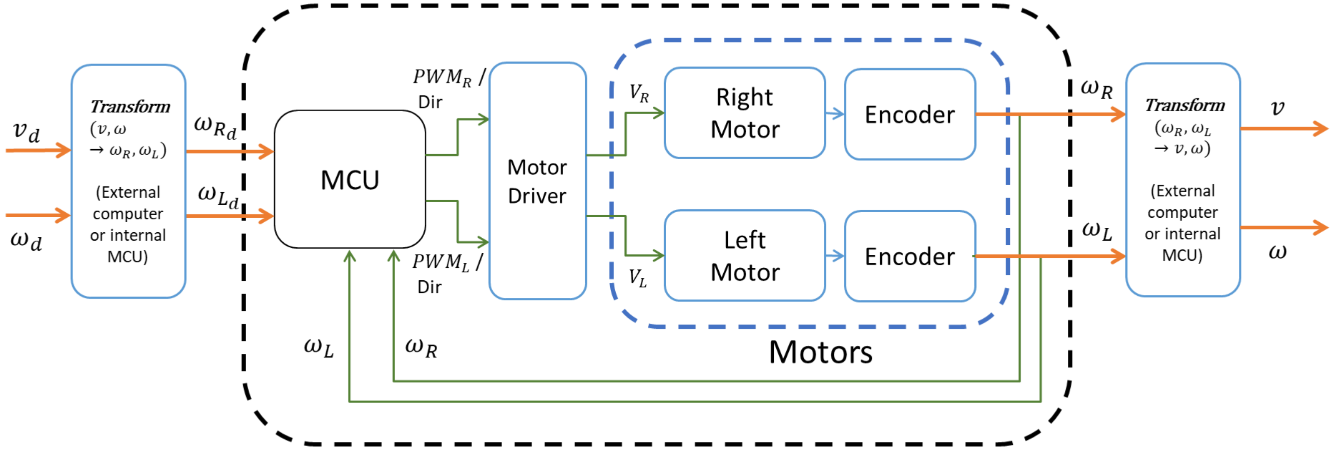
Differential drive robots



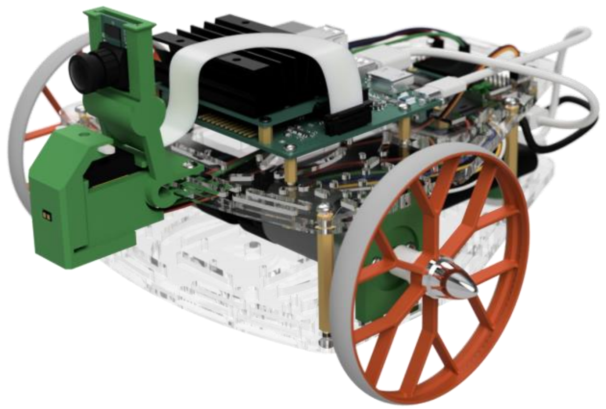


Differential drive robots

- A mobile robot, such as the Puzzlebot, requires the combination and usage of sensors and actuators to work.
- Usually, in mobile robots a closed control loop control for each of the motors is present, to reach the required velocities, set by the user.
- In robotics this is called low level control or inner control.
- For the case of a wheeled mobile robot is a common practice to implement a PID control for each motor.

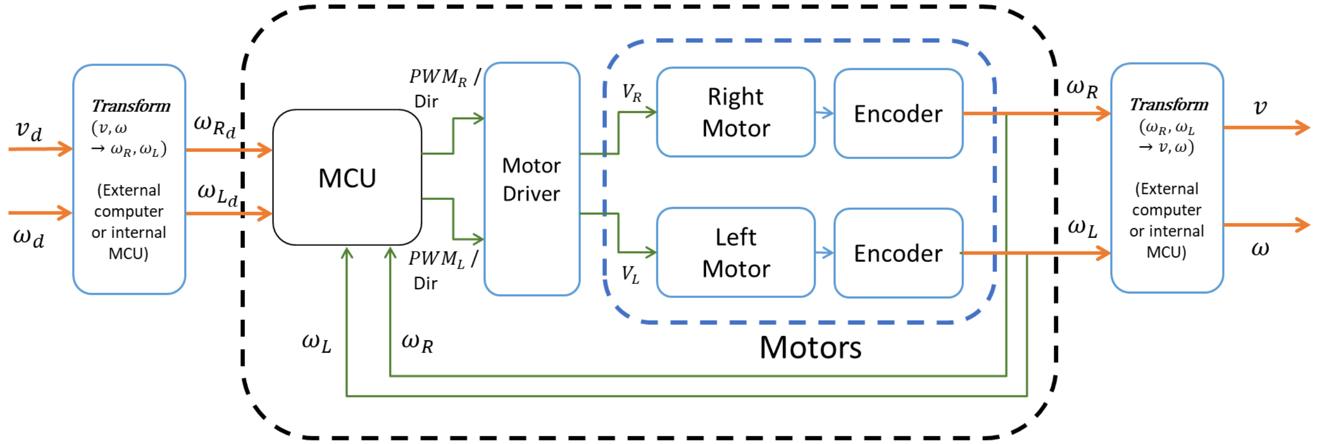


Real Robot Diagram*

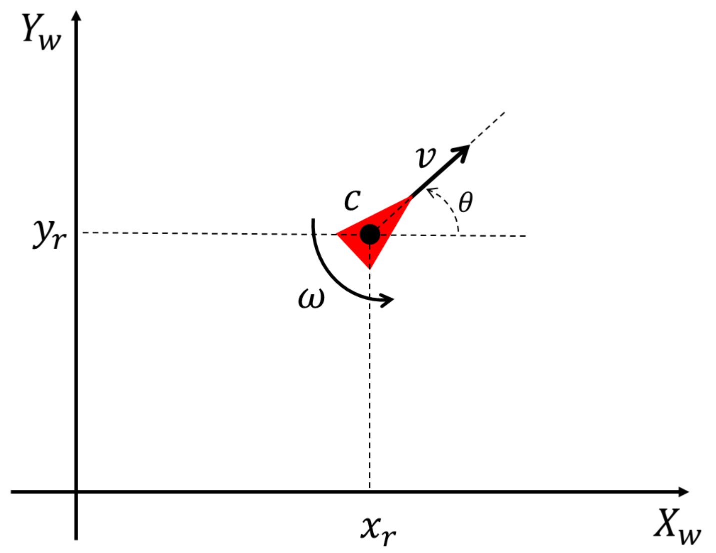


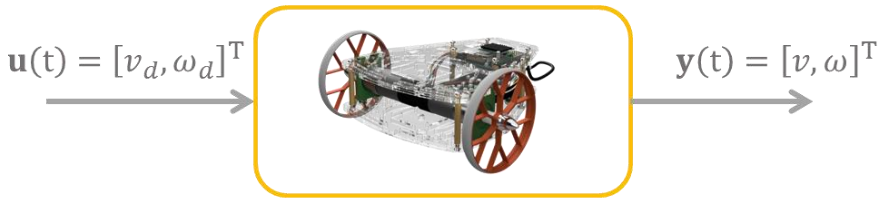
Differential drive robots

- It can be observed that the inputs to the robot are the linear and angular speeds of the robot i.e., $\mathbf{u}(\mathrm{t}) = [v_d, \omega_d]^T$
- The outputs are the wheel velocities $[\omega_r, \omega_l]$ that can be transformed into robots' linear and angular velocities $\mathbf{y}(\mathrm{t}) = [v, \omega]^T$.
- The inner loop controller and actuators will determine the behavior of the robot.



Real Robot Diagram*

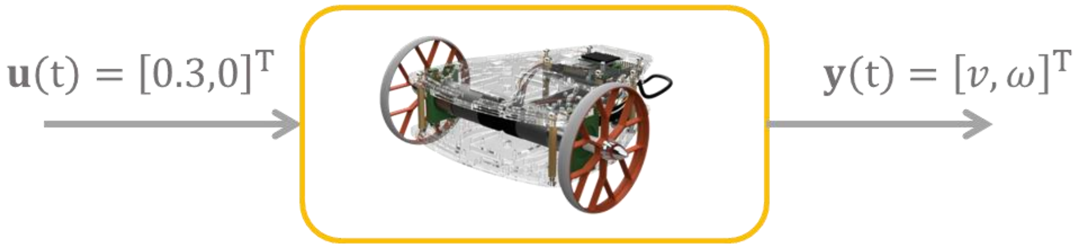


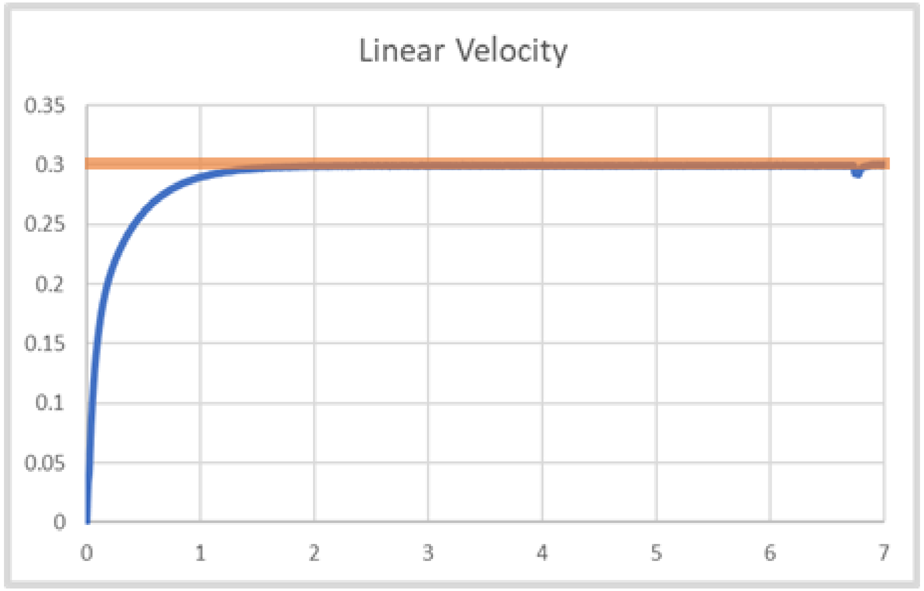

$\mathbf{u}(\mathrm{t}) = [v_d, \omega_d]^\mathrm{T}$
$\mathbf{y}(\mathrm{t}) = [v, \omega]^\mathrm{T}$

Mobile Robots and Open Loop Control

- Let the system (in this case the Puzzlebot) to be represented as a black box and let only constant linear velocity of $v_d = 0.3\frac{m}{s}$ to be the input to the system (step input).
- It can be observed in the plot that the output of the system (blue) matches the input reference (orange) at steady state.
- Therefore, the gain of the system is said to be 1.

  **This gain is only for the robot system for any other system the system's gain must be estimated.*

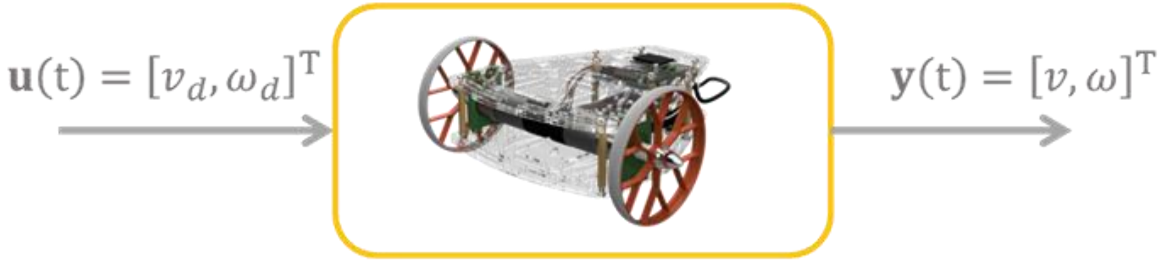




Mobile Robots and Open Loop Control

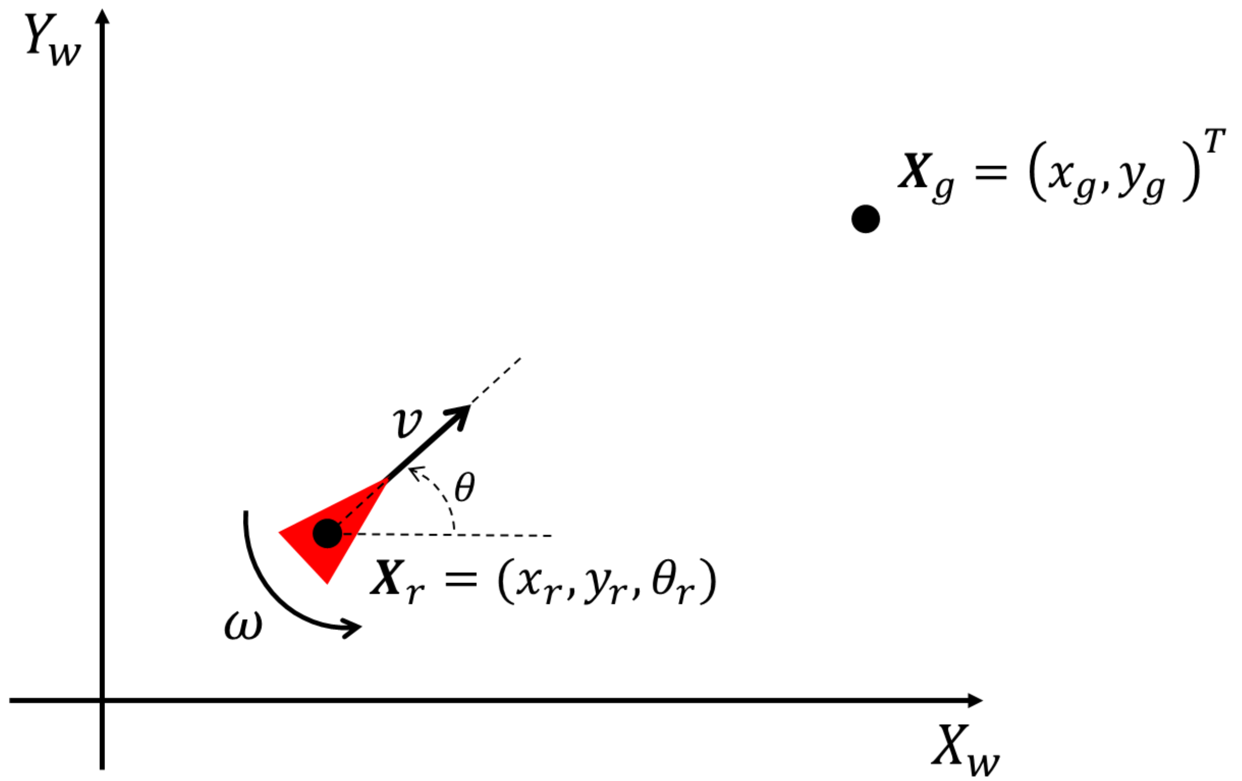
Analysis

- A more in-depth analysis of this situation reveals that this is due to the inner controllers of the robot.
  - If the inner controllers were poorly tunned, the gain might not be 1.
- The inner controllers allows the robot to follow the reference after some transient.
- The same thing happens with the angular speed.


$\mathbf{u}(t) = [v_d, \omega_d]^T$
$\mathbf{y}(t) = [v, \omega]^T$

Mobile Robots and Open Loop Control

- Knowing that the system's output will follow the input after a transient, a question arises…
- Can we use this information to move the robot a certain distance?
- The answer is …
  - YES! Since we know the input and output to be the linear and angular velocities and the gain of the system it is possible to use this information to move the robot a certain distance.



Y_w
X_w
$\boldsymbol{X}_g = (x_g, y_g)^T$
v
θ
ω
$\boldsymbol{X}_r = (x_r, y_r, \theta_r)$

Mobile Robots and Open Loop Control

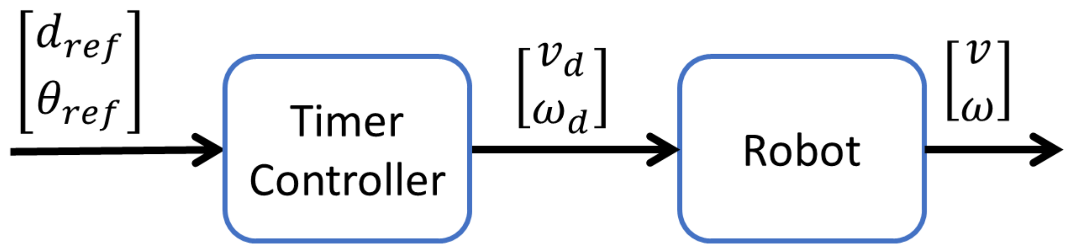
- Since we know that the output follows the input, it is possible to approximate the distance and angle traveled by using the following formula.

$$d = v \cdot dt$$
$$\theta = \omega \cdot dt$$

  where dt is the time since the input was applied.

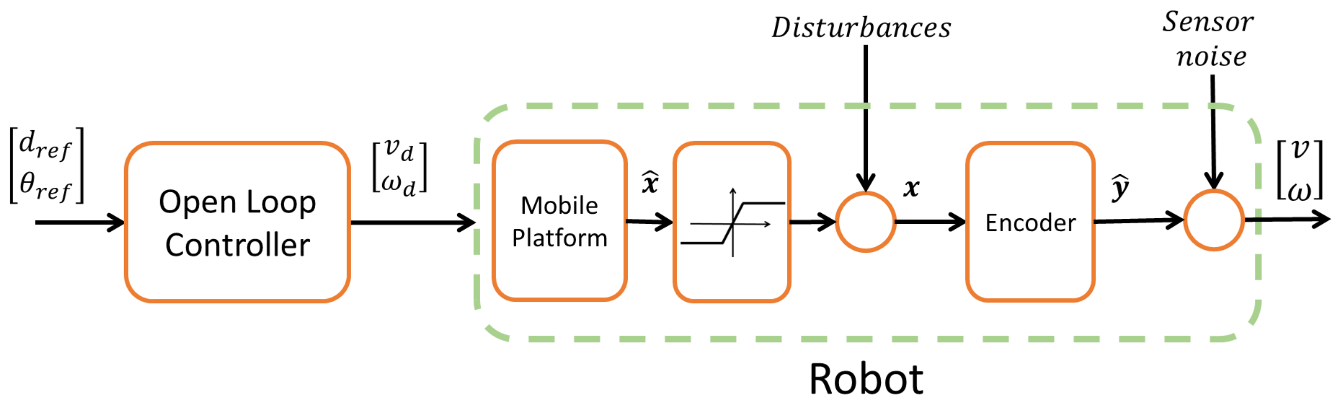
- For this case, the user can fix the values of v and ω and just set a dt (timer) to move a certain distance.

This is an Open Loop Control in Mobile Robots!.


$\begin{bmatrix} d_{ref} \\ \theta_{ref} \end{bmatrix}$
Timer Controller
$\begin{bmatrix} v_d \\ \omega_d \end{bmatrix}$
Robot
$\begin{bmatrix} v \\ \omega \end{bmatrix}$

Open Loop Control: Mobile Robot

- As with any Open Loop Controller, when designing this controller, the nonlinear behaviour, perturbations, environment and noise should be taken into consideration.
- Disturbances are due to the mechanical unbalance of the wheels, wheel slippage with the floor, defects of productions, changes in the environment, etc.
- Noise can be found in the encoder reading, the noise can be due to electromagnetic noise or radiation, mechanical unbalance of encoders, etc.

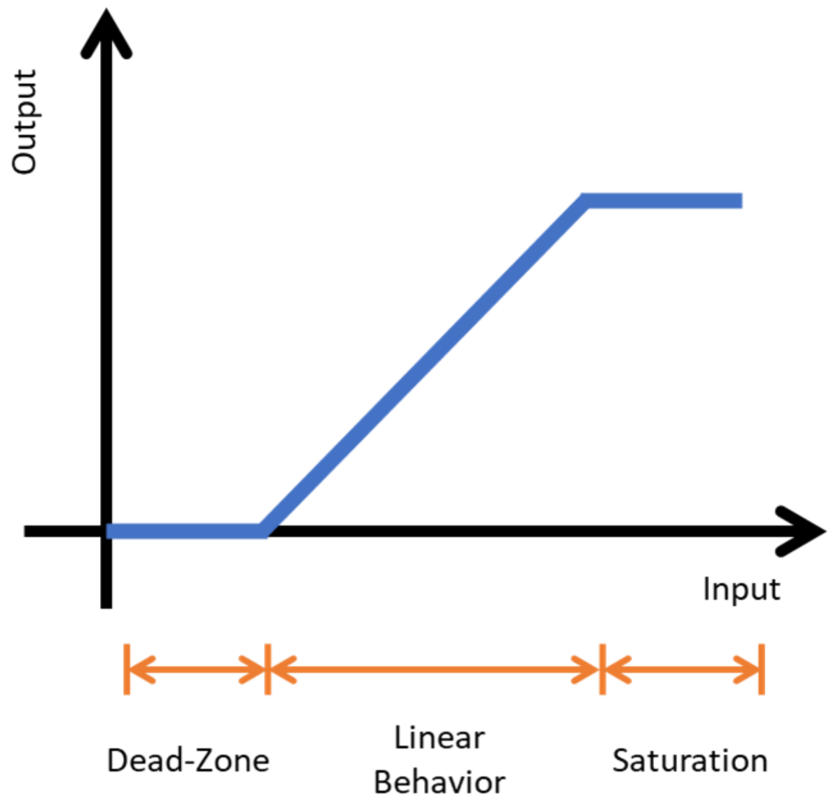


$\begin{bmatrix} d_{ref} \\ \theta_{ref} \end{bmatrix}$
Open Loop Controller
$\begin{bmatrix} v_d \\ \omega_d \end{bmatrix}$
Mobile Platform
$\hat{x}$
Disturbances
x
Encoder
$\hat{y}$
Sensor noise
$\begin{bmatrix} v \\ \omega \end{bmatrix}$
Robot

Open Loop Control: Mobile Robot

Nonlinearities

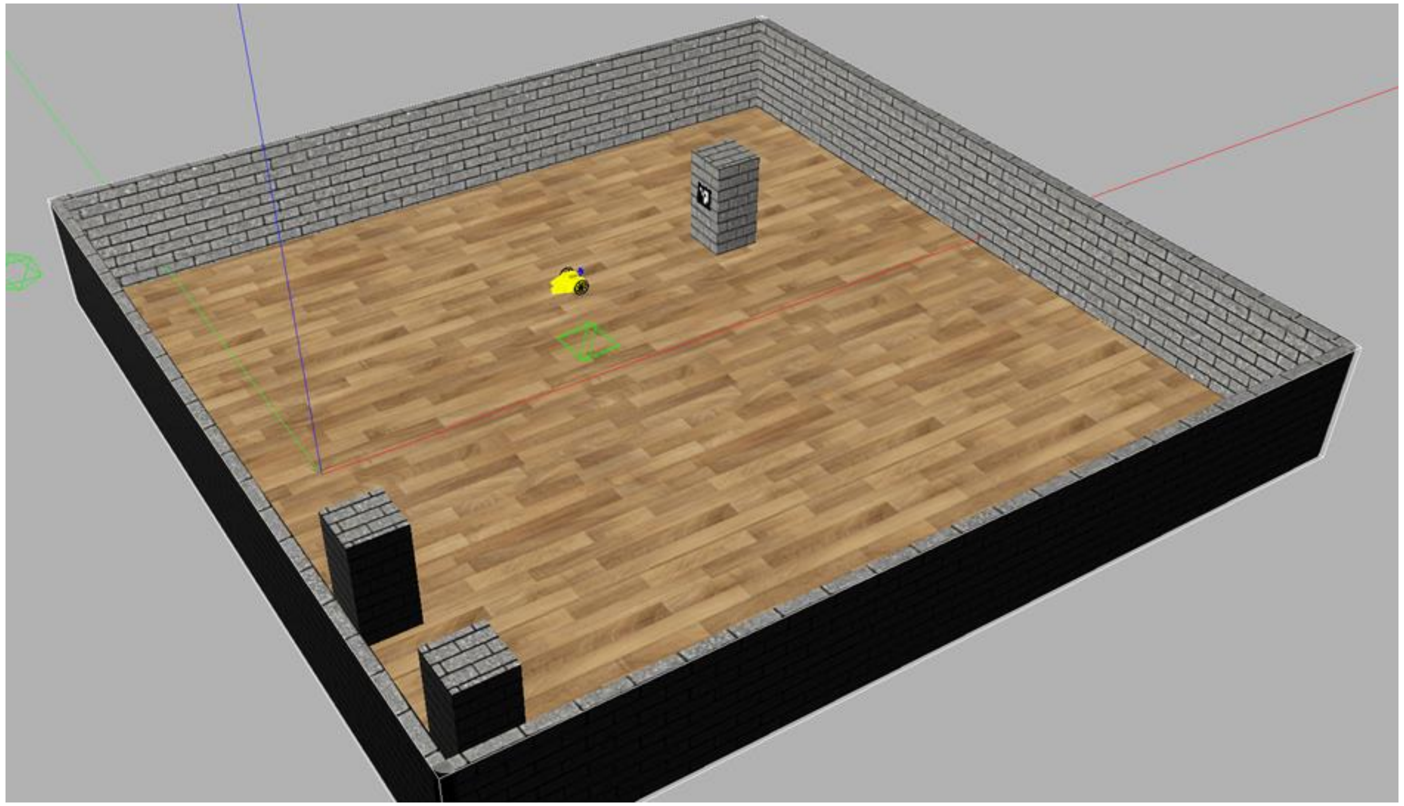
- In this case, the real mobile robot is a nonlinear system, due to the maximum and minimum velocities of the motors therefore it presents saturation and dead-zone.
    - *Other nonlinear behaviour are present, but they are not relevant in this case.
- The linear behaviour is present within a region in which the actuators (motors) output are proportional to the input.



Open Loop Control: Mobile Robot

Simulation

- Careful! When using a simulator to develop an open loop control.
- Depending on the simulator, the dynamical behavior of the robot can be linear, nonlinear or kinematic and might or might not consider the noise and disturbances .
- Use simulators with caution.



{Learn, Create, Innovate};

Open Loop Control

Activity 1: Simple Control for the Puzzlebot

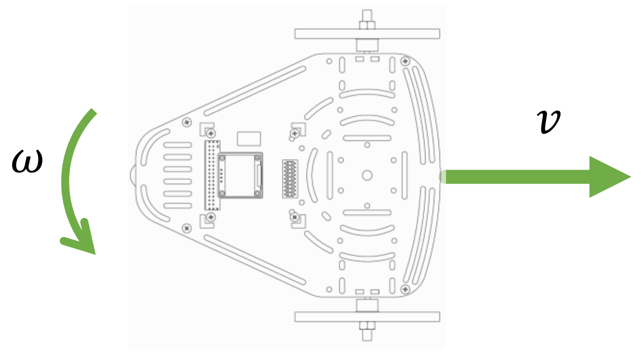
Introduction

Introduction

- The following activity will help you build a simple controller for the Puzzlebot
- This activity consist of creating a node that sends commands to the real robot and the gazebo simulation and move in a straight line for a period of time.
- This activity will use the Puzzlebot for testing.
- For more information about the Puzzlebot and how to use it please go to the presentation "MCR2_Puzzlebot_Jetson_Ed_ROS2"

Activity 1: Moving a Puzzlebot

Straight line



Drive the robot 2 meters in a straight line, turn and come back.

Objectives:

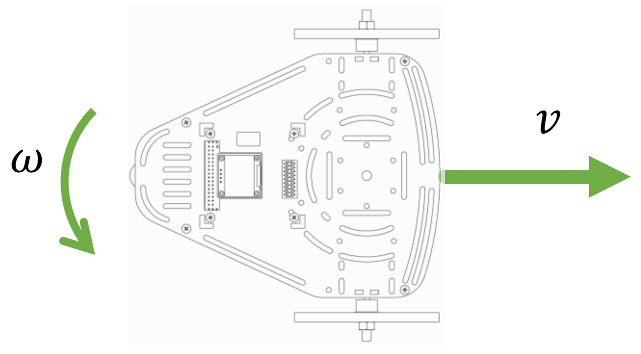
- Create a ROS node that sends commands to the robot gazebo simulation and move in a straight line for 2 meters.
- The Puzzlebot requires the commands to be published on a Twist() Message on the /cmd_vel topic.

```
geometry_msgs/Twist

geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

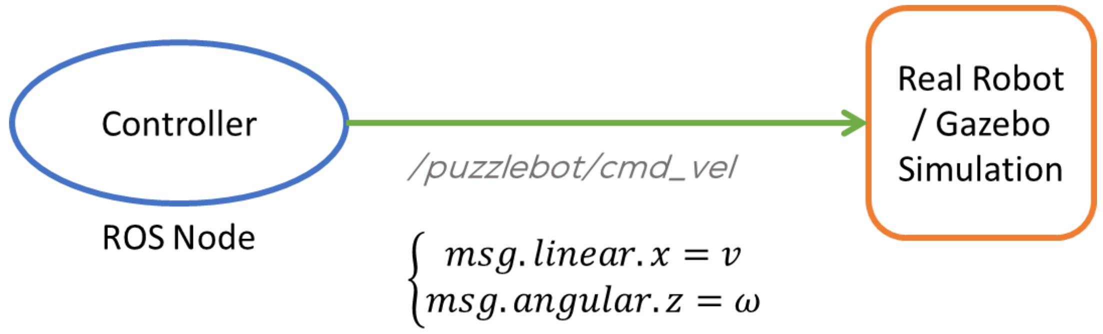
Activity 1: Moving a Puzzlebot

Information

- The Puzzlebot being a nonholonomic DDR, only accepts commands on the "linear x" and "angular z" parts of the message. Corresponding to linear velocity v and heading angular velocity ω.



```
geometry_msgs/Twist

geometry_msgs/Vector3 linear
  float64 x
geometry_msgs/Vector3 angular
  float64 z
```



$$\begin{cases} msg.linear.x = v \\ msg.angular.z = \omega \end{cases}$$

Activity 1: Moving a Puzzlebot

Instructions

For this project two options are given to the students, using the template or making a package from scratch.

Template:

- Download the template from Activity 1, add it to your workspace and compile it using Colcon.

Create a package:

- Make a new package called "puzzlebot_control", with the following packages: rclpy, std_msgs, geometry_msgs, python3-numpy, ros2launch

```
$ ros2 pkg create --build-type ament_python
puzzlebot_control --node-name open_loop_ctrl --dependencies
geometry_msgs python3-numpy rclpy ros2launch std_msgs --
description 'Puzzlebot controllers' --maintainer-name 'Mario
Martinez' --maintainer-email 'mario.mtz@manchester-
robotics.com' --license Apache-2.0
```

- Give executable permission to the file

```
$ cd ~/src/puzzlebot_control/puzzlebot_control/
$ sudo chmod +x open_loop_ctrl.py
```

- Open the file "`open_loop_ctrl.py`"

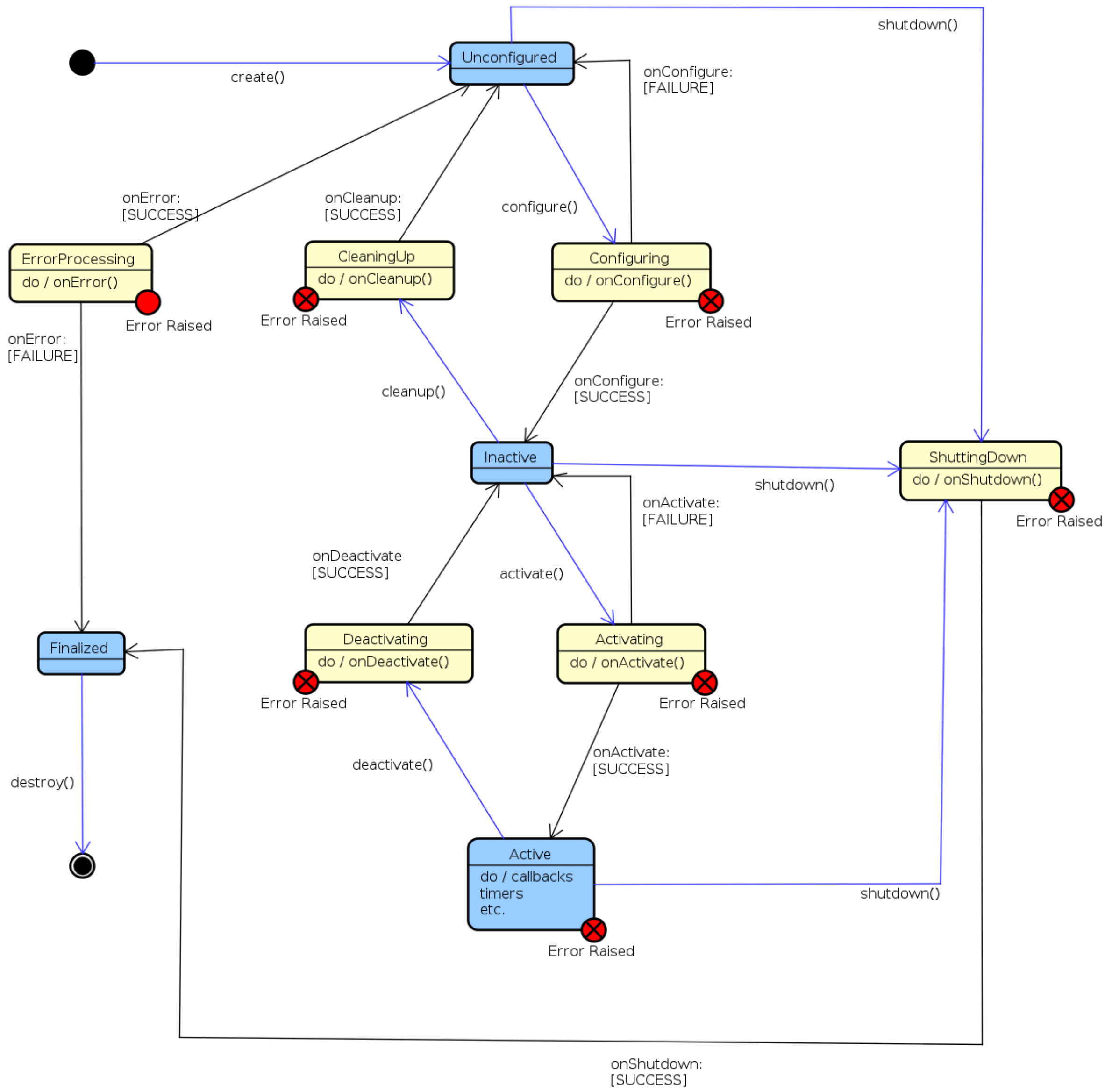
State Machines

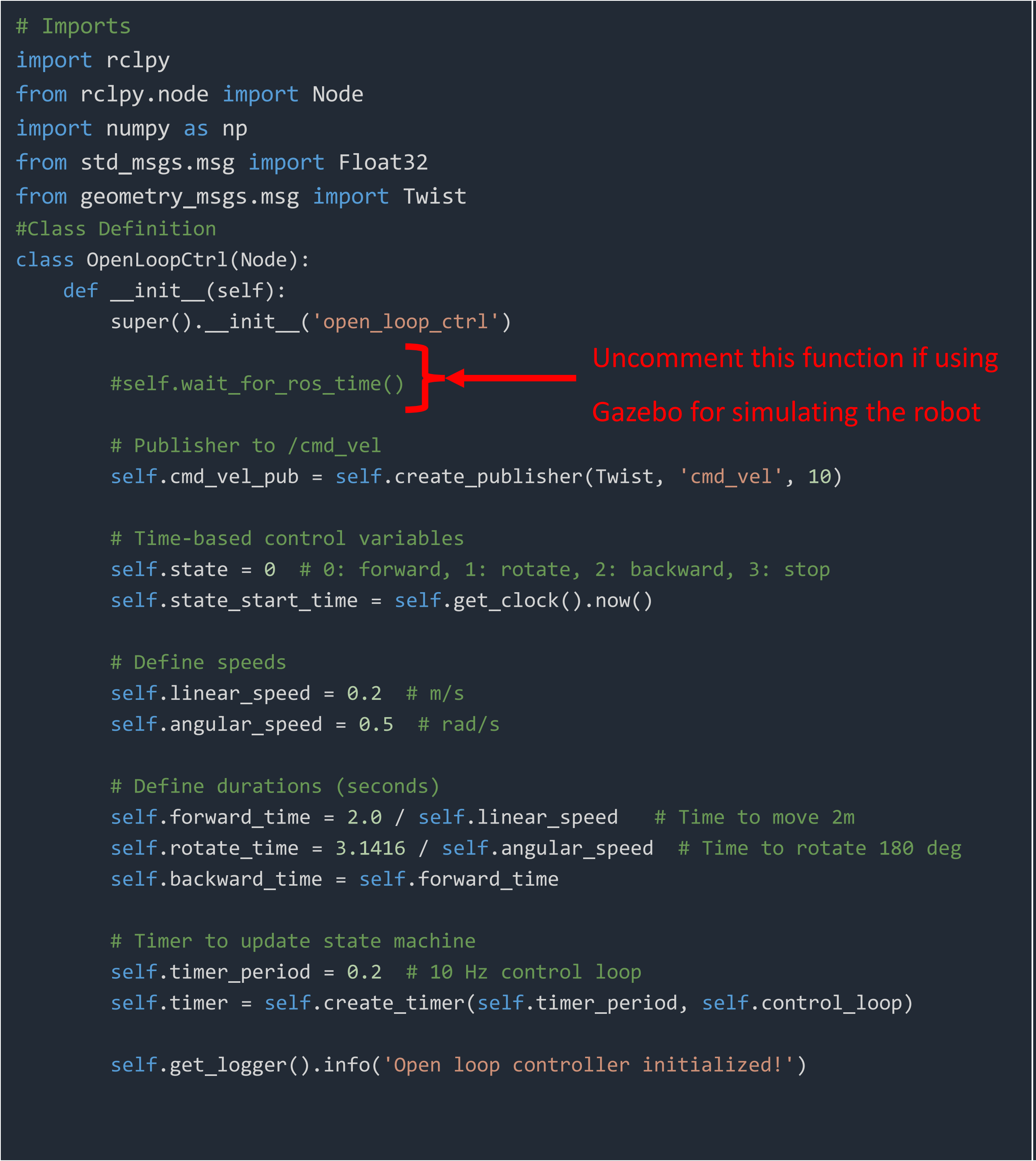
Definition

- A state machine is a computational model that transitions between predefined states based on inputs or conditions.
- Finite State Machines (FSM) – Fixed number of states.

Why use state machines?

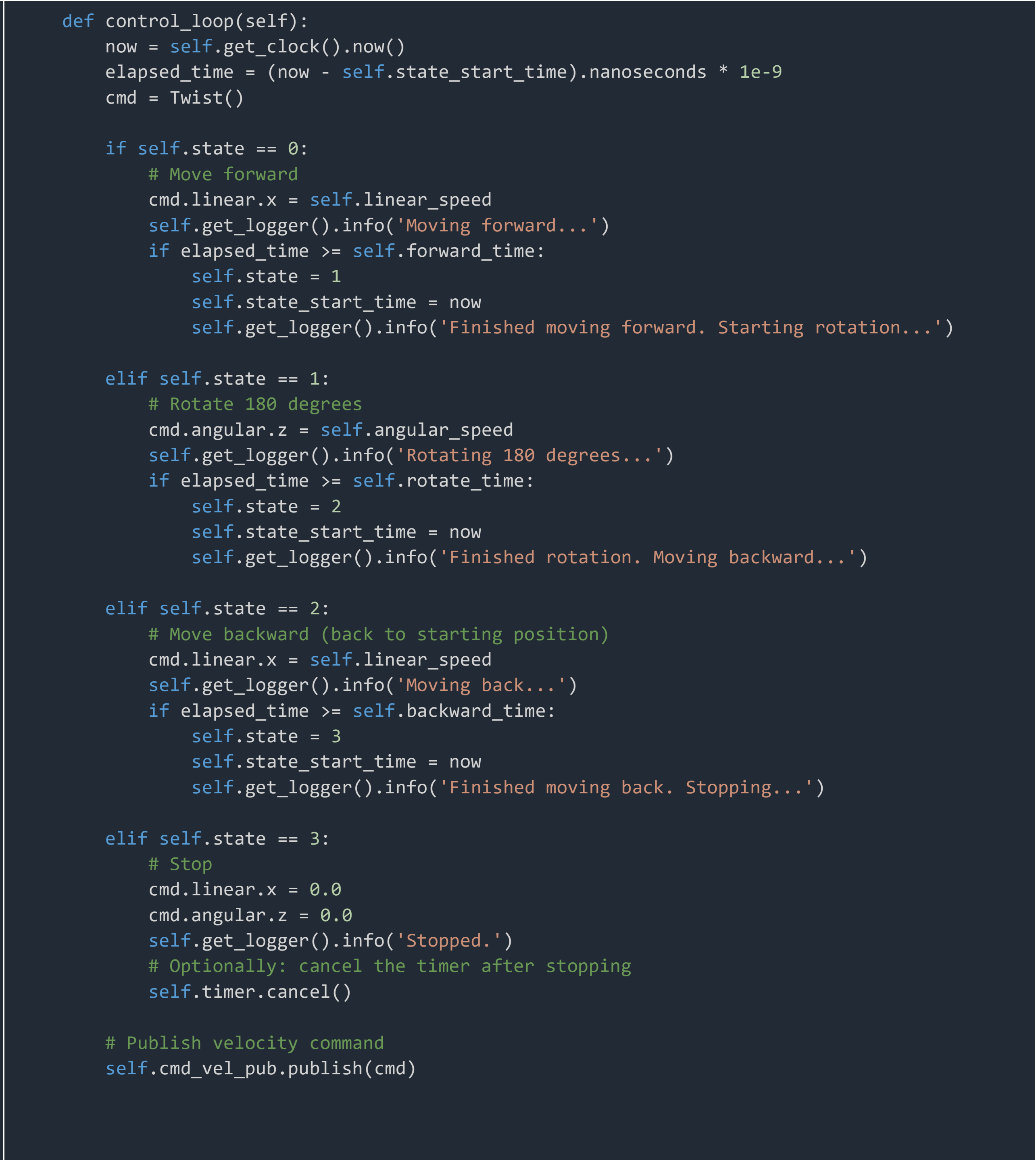
- Structured way to control robotic behaviour.
- Ensures modular, scalable, and readable code.





```python
# Imports
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist
#Class Definition
class OpenLoopCtrl(Node):
    def __init__(self):
        super().__init__('open_loop_ctrl')

        #self.wait_for_ros_time()

        # Publisher to /cmd_vel
        self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)

        # Time-based control variables
        self.state = 0  # 0: forward, 1: rotate, 2: backward, 3: stop
        self.state_start_time = self.get_clock().now()

        # Define speeds
        self.linear_speed = 0.2  # m/s
        self.angular_speed = 0.5  # rad/s

        # Define durations (seconds)
        self.forward_time = 2.0 / self.linear_speed   # Time to move 2m
        self.rotate_time = 3.1416 / self.angular_speed  # Time to rotate 180 deg
        self.backward_time = self.forward_time

        # Timer to update state machine
        self.timer_period = 0.2  # 10 Hz control loop
        self.timer = self.create_timer(self.timer_period, self.control_loop)

        self.get_logger().info('Open loop controller initialized!')
```

Uncomment this function if using Gazebo for simulating the robot



```python
    def control_loop(self):
        now = self.get_clock().now()
        elapsed_time = (now - self.state_start_time).nanoseconds * 1e-9
        cmd = Twist()

        if self.state == 0:
            # Move forward
            cmd.linear.x = self.linear_speed
            self.get_logger().info('Moving forward...')
            if elapsed_time >= self.forward_time:
                self.state = 1
                self.state_start_time = now
                self.get_logger().info('Finished moving forward. Starting rotation...')

        elif self.state == 1:
            # Rotate 180 degrees
            cmd.angular.z = self.angular_speed
            self.get_logger().info('Rotating 180 degrees...')
            if elapsed_time >= self.rotate_time:
                self.state = 2
                self.state_start_time = now
                self.get_logger().info('Finished rotation. Moving backward...')

        elif self.state == 2:
            # Move backward (back to starting position)
            cmd.linear.x = self.linear_speed
            self.get_logger().info('Moving back...')
            if elapsed_time >= self.backward_time:
                self.state = 3
                self.state_start_time = now
                self.get_logger().info('Finished moving back. Stopping...')

        elif self.state == 3:
            # Stop
            cmd.linear.x = 0.0
            cmd.angular.z = 0.0
            self.get_logger().info('Stopped.')
            # Optionally: cancel the timer after stopping
            self.timer.cancel()

        # Publish velocity command
        self.cmd_vel_pub.publish(cmd)
    def wait_for_ros_time(self):
        self.get_logger().info('Waiting for ROS time to become active...')
        while rclpy.ok():
            now = self.get_clock().now()
            if now.nanoseconds > 0:
                break
            rclpy.spin_once(self, timeout_sec=0.1)
        self.get_logger().info(f'ROS time is active!')


#Main
def main(args=None):
    rclpy.init(args=args)

    node = OpenLoopCtrl()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        node.destroy_node()

#Execute Node
if __name__ == '__main__':
    main()
```

When connecting Gazebo with ROS, the user Nodes must wait until Gazebo Publishes the topic "/clock" to synchronize the Gazebo simulation time with ROS time.

Otherwise, the nodes might start without Gazebo and could cause instability.

"For calculations of time durations when using simulation time, clients should always wait until the first non-zero time value has been received before starting."

More information here.

Activity
Some tips and tricks

- Use `self.get_clock().now()` to get the current time.
- If the robot is not moving, check your topics with `ros2 topic echo` and `ros2 topic pub`
- Ensure your python file is executable: `sudo chmod +x <path_to_file>.py`
- Ensure you source your file: `source install/setup.bash`

Activity 1: Results

- Save and recompile the project.
- Build the program using “colcon build”

```
$ colcon build
$ source install/setup.bash
```

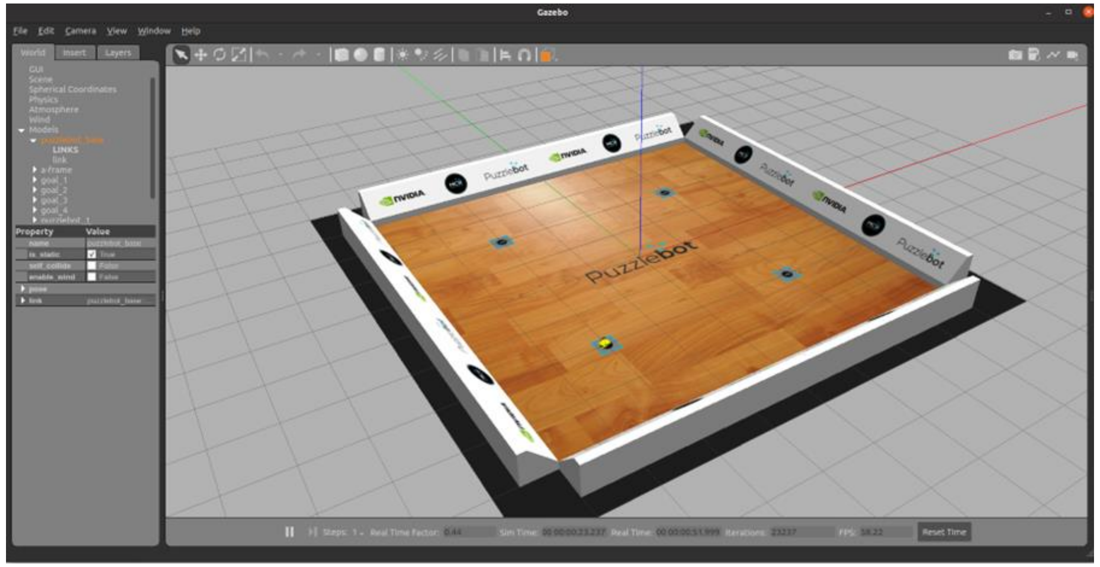
- Connect to the Puzzlebot via Wi-Fi
- If using Gazebo run the gazebo simulator, start the simulation as follows:

```
$ ros2 launch puzzlebot_gazebo gazebo_example_launch.py
```

- Open another terminal and run the Activity

```
$ ros2 run puzzlebot_control open_loop_ctrl
```

- The robot should move forward, rotate 180 deg. and come back to the initial position.



Thank you

{Learn, Create, Innovate};

T&C

Terms and conditions

{Learn, Create, Innovate};

Terms and conditions