Robotic Navigation in a ROS/Gazebo Environment

@ "BIP2026: Embedded systems:ROS in Embedded Design for Mobile Robots"
Wrocław University of Science and Technology

Jorge Barreiros (jmsousa@isec.pt)
Coimbra Polytechnic University, ISEC
Coimbra, Portugal

Wrocław University of Science and Technology
isec Politécnico de Coimbra
HELHa Haute École Louvain en Hainaut
POLITÉCNICO DE PORTALEGRE

Before we start...

- Find the code and commands:

https://github.com/jorgebarreiros-aet/navigation

- Always change the ROS topic names when running code from examples (to avoid conflict with other students)
  - Eg: /cmd_vel -> my_name/cmd_vel

The plan for today...

- ROS & Gazebo Integration
  - ROS 2 Publishing and Subscribing from Python
  - The ROS-Gazebo Bridge
- Navigation to Select Positions
  - Navigation to Specific Positions with Velocity Control

ROS & Gazebo Integration

Interfacing ROS2 topics with the Gazebo Simulator

From Python to ROS2

- A program can publish ROS2 messages containing relevant information into specific topics. For example:
  - A sensor can publish data readings into a `/sensor/data` topic.
  - A control system can publish the desired velocity into `/cmd_vel` topic. .
- Registered topic consumers will receive those messages and take appropriate action. For example:
  - A control system might read sensor data (from a topic such as `/sensor/data`) and take appropriate control action.
  - A motor controller can read desired velocity (from a topic such as `/cmd_vel`) and drive the motor appropriately,

From Python to ROS2

- A typical ROS program consists of the following operations:
  1. Initialization
  2. Create one or more ROS nodes
  3. Process node callbacks
  4. Shutdown

From Python to ROS2

- Let's walk through a simple Python program that sends information to ROS2 topic.
- This program will send a Twist message, containing desired linear and angular velocities, into a `/cmd_vel` topic.
- It highlights:
  - The lifecycle of a new ROS2 node
  - The creation of a new publisher for a certain topic & message type.
  - The use of a timer to achieve a periodic publication rate.
  - The publication of a message to the topic

From Python to ROS2

Python client library for ROS 2

The Twist message definition

Class that represents a ROS 2 node

We define a new ROS2 node type called `TwistPublisher`

We will create a publisher that will write a Twist message to the /cmd_vel topic. We will store up to 10 messages if subscribers are not able to keep up.

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist


class TwistPublisher(Node):

    def __init__(self):
        super().__init__('twist_publisher')
        # Create a publisher for the /cmd_vel topic
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 10)
        # Create a timer that calls the publish_twist function every 0.5 seconds
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.publish_twist)
        self.get_logger().info('Twist Publisher Node Started')
```

From Python to ROS2

The `publish_twist` method is called back periodically from the timer.

Set desired values

Publish message

```python
def publish_twist(self):
    msg = Twist()
    # Set linear velocity (forward)
    msg.linear.x = 0.5
    # Set angular velocity (rotation around Z axis)
    msg.angular.z = 0.5
    self.publisher_.publish(msg)
    self.get_logger().info(
        f'Publishing: linear.x={msg.linear.x}, angular.z={msg.angular.z}'
    )
```

Create a new twist message.

The main function implements typical lifecycle of a ROS2 application

```python
def main(args=None):
    rclpy.init(args=args)
    node = TwistPublisher()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()
```

From Python to ROS2

Initialize the library
Execute the node and block.
Keeps executing callbacks.

Standard Python script entry-point

```python
def main(args=None):
    rclpy.init(args=args)
    node = TwistPublisher()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Create a new TwistPublisher node

Cleanup. Destroy node and shutdown library.

From Python to ROS2

- To run our program, we setup ROS first (once) and the run it on a terminal

```
jorge@UbuntuNoble:~/gzfiles/example$ source /opt/ros/jazzy/setup.bash
jorge@UbuntuNoble:~/gzfiles/example$ python3 ./twist_publisher.py
[INFO] [1771174418.766045646] [twist_publisher]: Twist Publisher Node Started
[INFO] [1771174419.546033479] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771174419.760747112] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771174420.258109492] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771174420.763963691] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771174421.262968595] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771174421.760035074] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
```

From ROS2 to Python

- Let's now consider an example of a Python program *consuming* information from a ROS topic.
- We will read odometry information from a ROS2 topic.
- The program highlights:
  1. Creating a subscriber and subscription call-back method.
  2. The odometry message format
  3. Converting data from quaternion to yaw

From ROS2 to Python

- Looking at the imports, we can see that the only novelty is that we are now using the `Odometry` message type, rather than `Twist`..

```
import math
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
```

From ROS2 to Python

- Skipping ahead to the main entry point, we see there is nothing new here...
  - The lifecycle is the same
- We are creating a new `OdomObserver` node.

```python
def main(args=None):
    rclpy.init(args=args)
    node = OdomObserver()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

From ROS2 to Python

- A new `OdomObserver` node type is created:
  - When initialized, a subscriber is created that will monitor the "`model/vehicle_blue/odometry`" topic for `Odometry` messages. The history depth is 10.
  - The callback method `on_odom` will be called whenever a new message is posted.

```
class OdomObserver(Node):
    def __init__(self):
        super().__init__("odom_observer")
        # Depth 10 is fine for observation
        self.sub = self.create_subscription(Odometry, "/model/vehicle_blue/odometry", self.on_odom, 10)
        self.get_logger().info("Listening to /odom (nav_msgs/msg/Odometry) ...")
```

From ROS2 to Python

- The `on_odom` method of the `OdomObserver` will be called to process published `Odometry` messages.
- Odometry messages have the following structure:

```
nav_msgs/Odometry
    pose.pose.position
    pose.pose.orientation
    twist.twist.linear
    twist.twist.angular
```

- xyz position
- rotation (quaternion!)
- Linear velocity (along xyz axes)
- Angular velocity (around xyz axes)

From ROS2 to Python

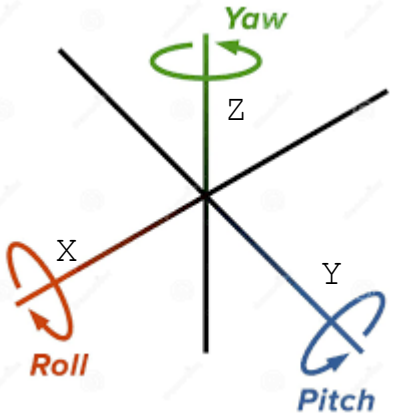
- The `on_odom` method of the `OdomObserver` will be called to process published `Odometry` messages.

Get position and orientation
Convert from quaternion to yaw
Get linear and angular velocity

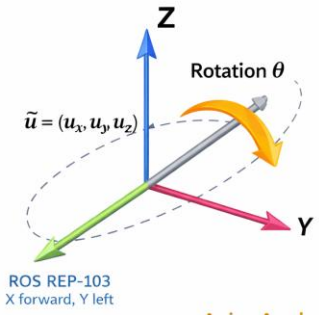
```
on_odom(self, msg: Odometry) -> None:
p = msg.pose.pose.position
q = msg.pose.pose.orientation
yaw = yaw_from_quaternion(q.x, q.y, q.z, q.w)
v = msg.twist.twist.linear
w = msg.twist.twist.angular
yaw_deg = math.degrees(yaw)
self.get_logger().info(
    f"x={p.x:.3f} m, y={p.y:.3f} m, yaw={yaw_deg:.1f} deg | "
    f"v_x={v.x:.3f} m/s, w_z={w.z:.3f} rad/s"
)
```

Roll, Pitch, Yaw and Quaternions

- **Roll, Pitch, Yaw:** Rotation around the X, Y, and Z axis



- **Quaternion:** Based on rotation around an arbitrary unit axis vector $\{u_x,u_y,u_z\}$.
  - The vector and angle are **indirectly** encoded



Quaternion

Rotation:

$q = (x, y, z, w)$

$x = u_x \sin(\theta/2)$

$y = u_y \sin(\theta/2)$

$z = u_z \sin(\theta/2)$

$w = \cos(\theta/2)$

Axis–Angle → Quaternion Encoding

From ROS2 to Python

- The `yaw_from_quaternion` method computes yaw from a (x,y,z,w) quaternion.
  - *The yaw represents the orientation of a mobile robot moving on a plane.*

Required math to retrieve Yaw from Quaternion.

```python
def yaw_from_quaternion(x: float, y: float, z: float, w: float)
    """
    Convert quaternion (x,y,z,w) to yaw (rotation around Z) in r
    Assumes a standard ROS ENU frame and that roll/pitch are sma
    """
    # yaw (Z-axis rotation)
    siny_cosp = 2.0 * (w * z + x * y)
    cosy_cosp = 1.0 - 2.0 * (y * y + z * z)
    return math.atan2(siny_cosp, cosy_cosp)
```

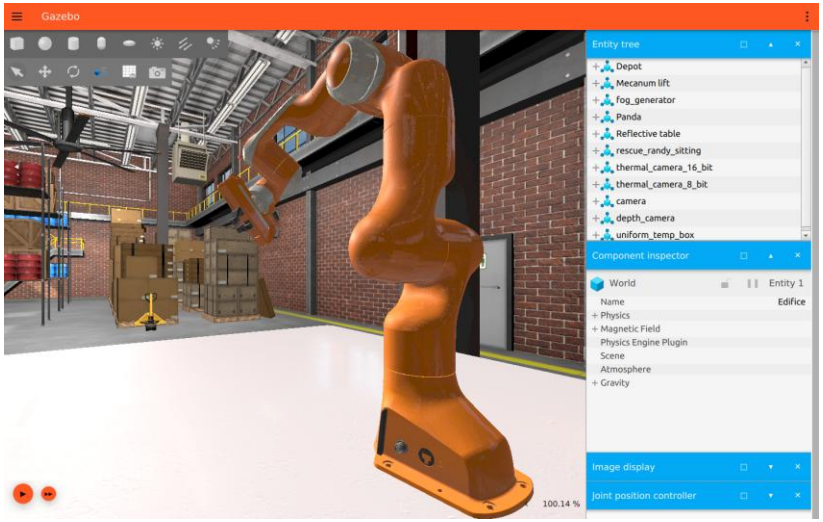
From Python to ROS2

- We can run our program on a terminal (after initializing ROS2 if necessary), and it will be "paused" waiting for odometry messages.
  - None are being sent yet, so nothing happens.

```
jorge@UbuntuNoble:~/gzfiles/example$ python3 ./twist_odom.py
[INFO] [1771184583.448488526] [odom_observer]: Listening to /odom (nav_msgs/msg/Odometry) ...
```

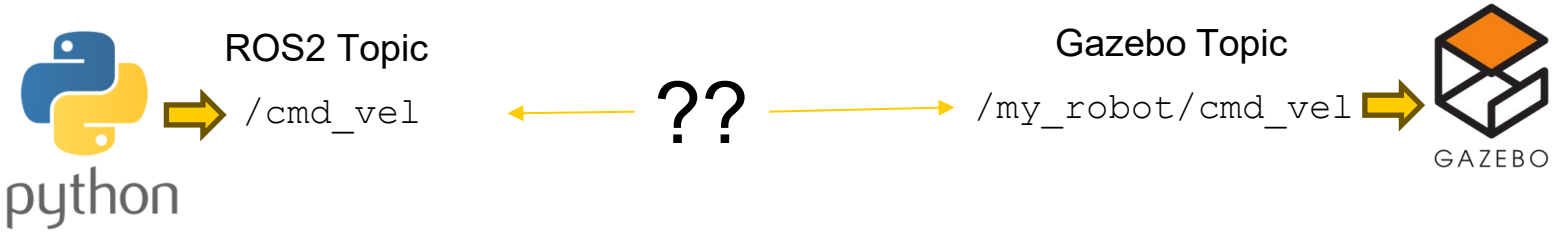
Gazebo Refresher

- Gazebo is a robotics simulator that has its own internal topics messaging system.
  - It simulates an entire world including robots, the environment and components like sensors.
  - **Check the online presentation and material for additional details.**



Connecting ROS2 to Gazebo

- Our goal is to make sure our Python code can control the simulation objects in Gazebo...



Connecting ROS2 to Gazebo

- We achieve this by running a **ROS bridge** that will perform the necessary adaptation between both systems.
  - *The bridge allows bidirectional communication*

Connecting ROS2 to Gazebo

- To properly work, the bridge must know:
  - The names of the topics on each end.
  - The type of messages on both sides.
  - The direction of communication (ROS to Gazebo, Gazebo to ROS, or birectional).
- One way to specify this is to create a YAML file with all the required information

Connecting ROS2 to Gazebo

- This example file should be self-explanatory.

Bridge.yaml

The topics to connect on both sides are named (both are `/cmd_vel` on this example).

Types of message on both ends

Direction of communication (ROS_TO_GZ, GZ_TO_ROS, or BIDIRECTIONAL)

```
- ros_topic_name: "/cmd_vel"
  gz_topic_name: "/cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "gz.msgs.Twist"
  direction: ROS_TO_GZ
```

Connecting ROS2 to Gazebo

- We can run the bridge with the configuration specified in the bridge.yaml file:
  - The bridge will propagate messages between both systems as determined in its configuration.

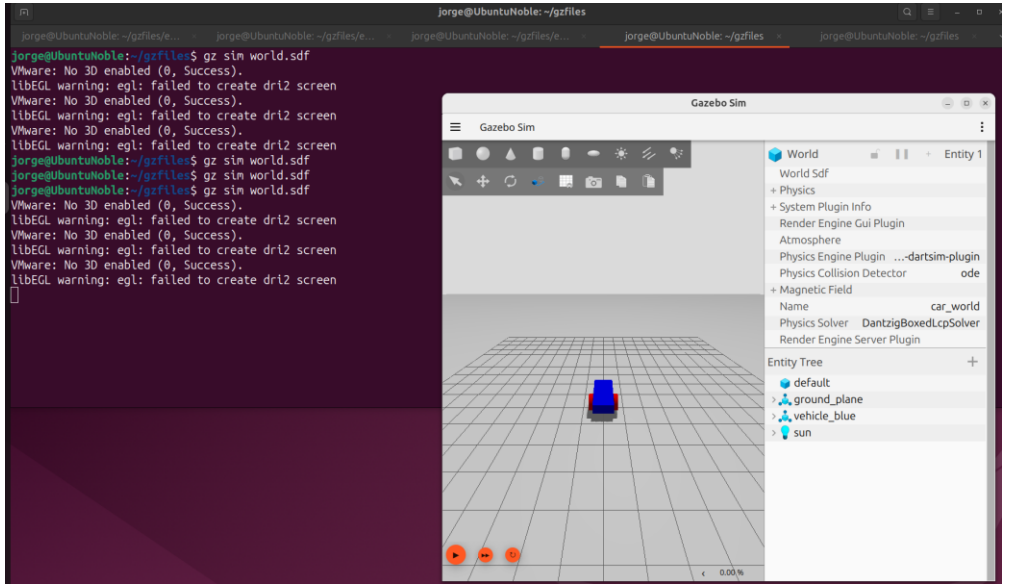
```
jorge@UbuntuNoble:~/gzfiles/example$ ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge config_file:=./bridge.yaml
[INFO] [launch]: All log files can be found below /home/jorge/.ros/log/2026-02-15-21-11-03-134615-UbuntuNoble-4596
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [bridge_node-1]: process started with pid [4599]
[bridge_node-1] [INFO] [1771189864.450921520] [ros_gz_bridge]: Creating ROS->GZ Bridge: [/cmd_vel (geometry_msgs/msg/Twist) -> /cmd_vel (gz.msgs.Twist
)] (Lazy 0)
```

Putting it all together

- We will now integrate all we learned so far by:
  - Start the Gazebo Simulation in one terminal
  - Running the Twist Publisher code in another terminal
  - Running the Odometry Subscriber in another terminal
  - Running the Gazebo Bridge in another terminal
  - Run the simulation

Running Gazebo

- Open a new terminal and run the Gazebo Simulator using the provided `world.sdf` file:
  - This file defines one mobile robot, with differential control, moving on a horizontal plane.
  - Refer to the online presentation for more details on Gazebo, the SDF files and its contents.



Running the publisher

- Open a new terminal, setup ROS2 and run the twist publisher

```
jorge@UbuntuNoble:~/gzfiles/example$ python3 ./twist_publisher.py
[INFO] [1771193515.476877724] [twist_publisher]: Twist Publisher Node Started
[INFO] [1771193515.962664273] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771193516.460319026] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771193516.962610307] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
[INFO] [1771193517.462277276] [twist_publisher]: Publishing: linear.x=0.5, angular.z=0.5
```

Running the Odometry Subscriber

- Open a new terminal, setup ROS2, and run the odometry subscriber

```
jorge@UbuntuNoble:~/gzfiles/example$ python3 ./twist_odom.py
[INFO] [1771193598.031381307] [odom_observer]: Listening to /odom (nav_msgs/msg/Odometry) ...
```

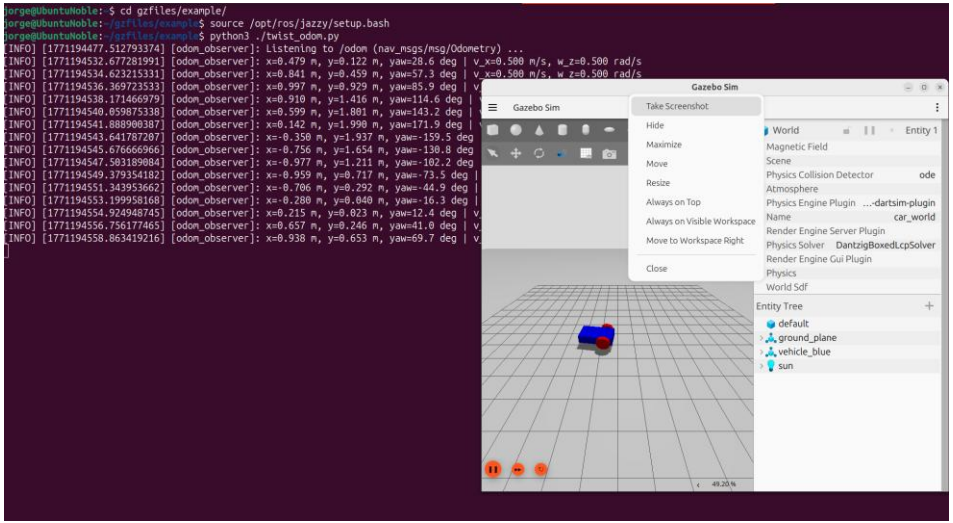
Running the Bridge

- Open a new terminal, setup ROS2, and run the bridge with file `bridge2.yaml`. This sets up two bridges
  - A bridge to send Twist commands from ROS to Gazebo
  - A bridge to read Odometry information from Gazebo to ROS

```
jorge@UbuntuNoble:~/gzfiles/example$ ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge config_file:=./bridge2.yaml
[INFO] [launch]: All log files can be found below /home/jorge/.ros/log/2026-02-15-22-15-18-266683-UbuntuNoble-6478
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [bridge_node-1]: process started with pid [6481]
[bridge_node-1] [INFO] [1771193719.615783586] [ros_gz_bridge]: Creating ROS->GZ Bridge: [/cmd_vel (geometry_msgs/msg/Twist) -> /cmd_vel (gz.msgs.Twist
)] (Lazy 0)
[bridge_node-1] [INFO] [1771193719.626143847] [ros_gz_bridge]: Creating GZ->ROS Bridge: [/model/vehicle_blue/odometry (gz.msgs.Odometry) -> /odom (nav
_msgs/msg/Odometry)] (Lazy 0)
[bridge_node-1] [INFO] [1771193720.612618599] [ros_gz_bridge]: Passing message from ROS geometry_msgs/msg/Twist to Gazebo gz.msgs.Twist (showing msg o
nly once per type)
```

Starting the Simulation

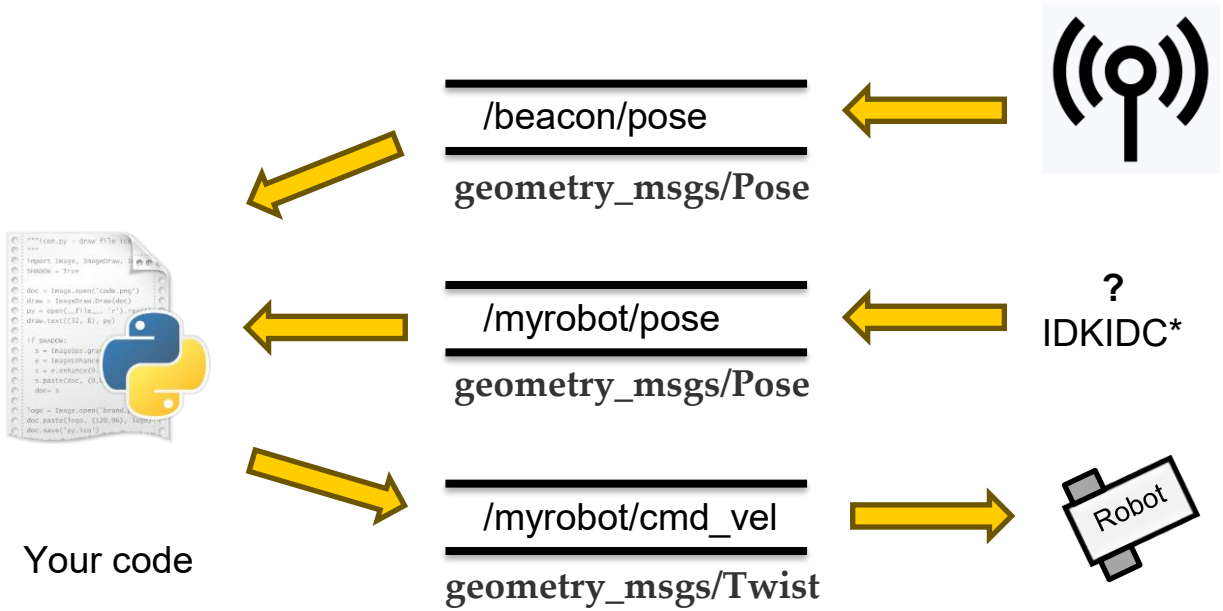
- Once all the components are up and running, start the simulation (press play on bottom left of Gazebo):
  - The simulated mobile robot should start moving with the velocity sent by the twist publisher
  - The odometry subscriber will start printing out the odometry information:
    - X,Y position, yaw (rotation on Z axis)
    - Linear velocity and angular velocity on Z axis



Navigation

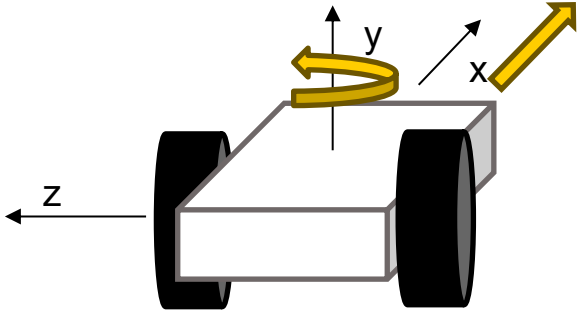
- Our objective is to navigate the robot so that it moves to (or nearby) the position of a beacon.
- The architecture of our setup is as follows:
  - The estimated position of the robot is periodically published in topic `/myrobot/pose`
  - The estimated position of a tag (beacon) is periodically published in topic `/beacon/pose`
  - The robot's velocity can be controlled by publishing an appropriate Twist to the `/myrobot/cmd_vel` topic.

System Architecture



Navigation

- Planar robots can be velocity controlled on linear X and angular Z.
  - Other commands don't make physical sense.



How to move towards a specific position?

- Find the direction and rotate in the appropriate direction
- Manage the linear velocity depending on the distance to target:
  - *Subjected to velocity limits for safety!*

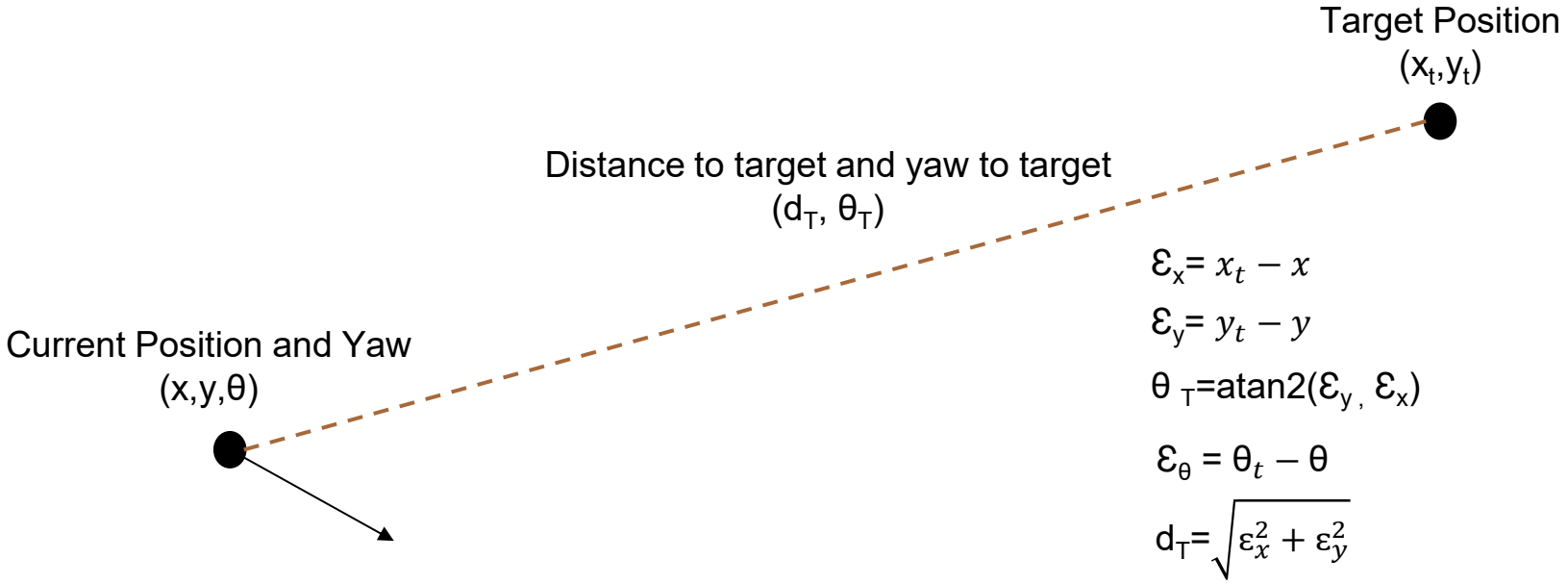
Adjusting Direction

- We cannot directly specify a direction of movement.
- We can only specify a desired angular velocity (ɵ')
- How should we proceed, given our current position and yaw, and target position?

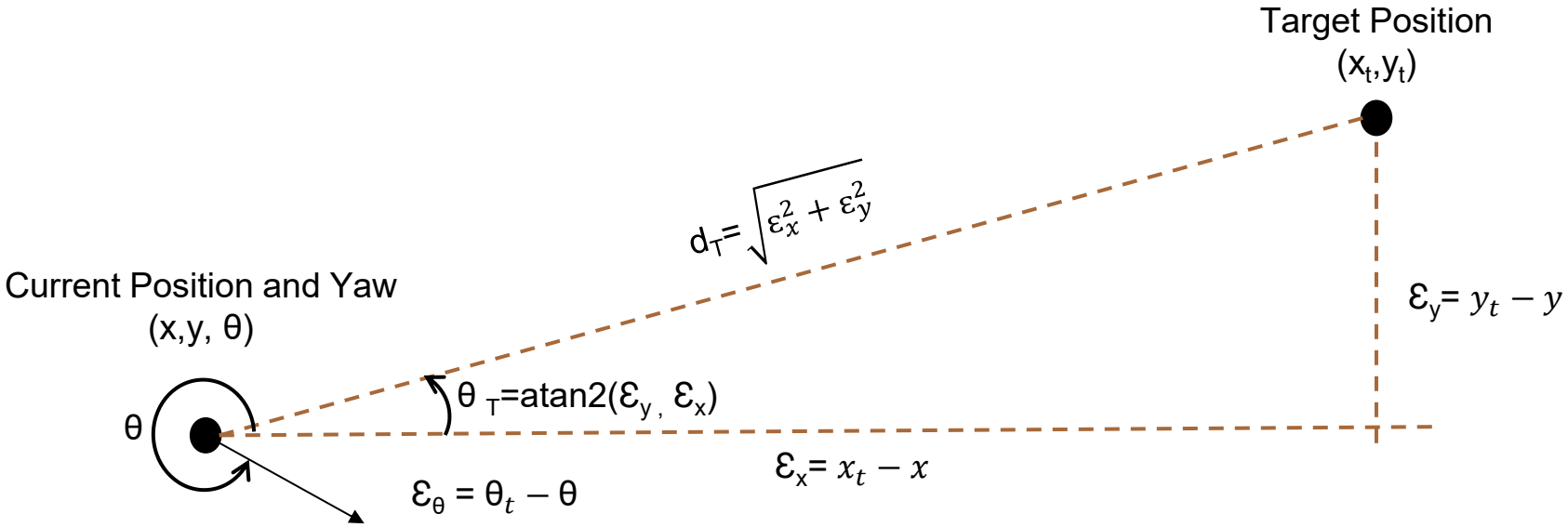
Target Position
(x_t, y_t)

Current Position and Yaw
(x,y,ɵ)

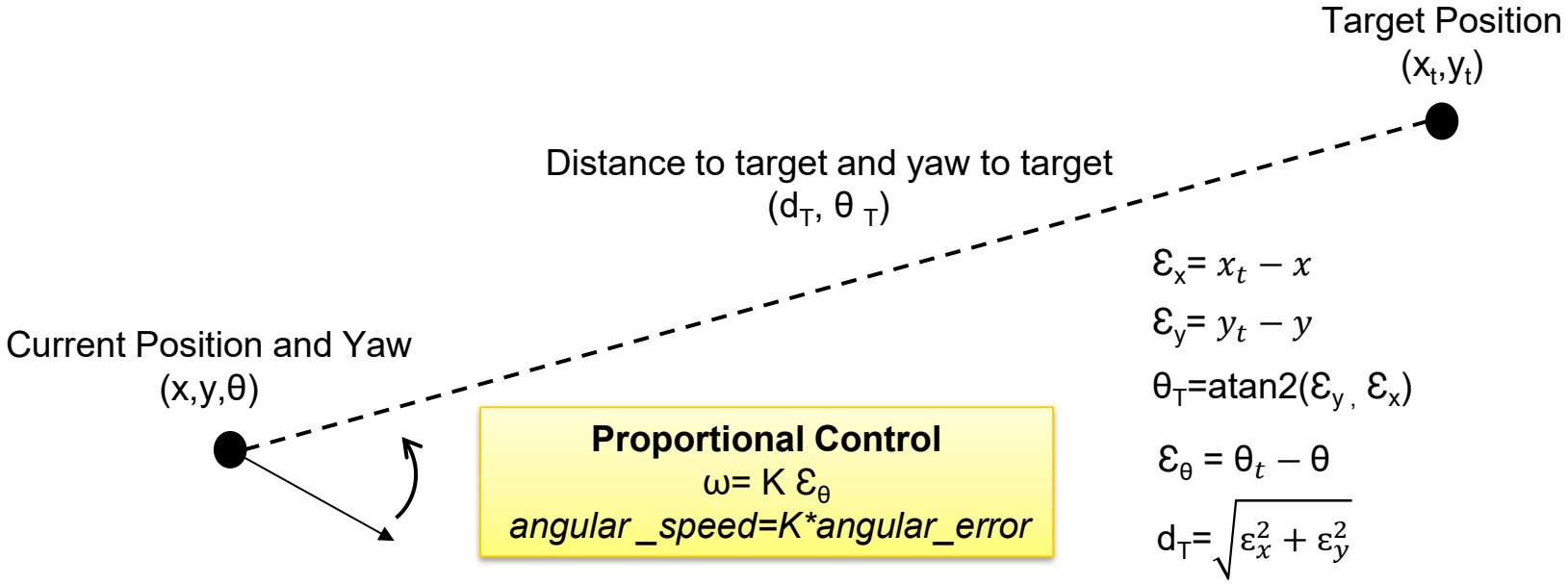
Adjusting Direction



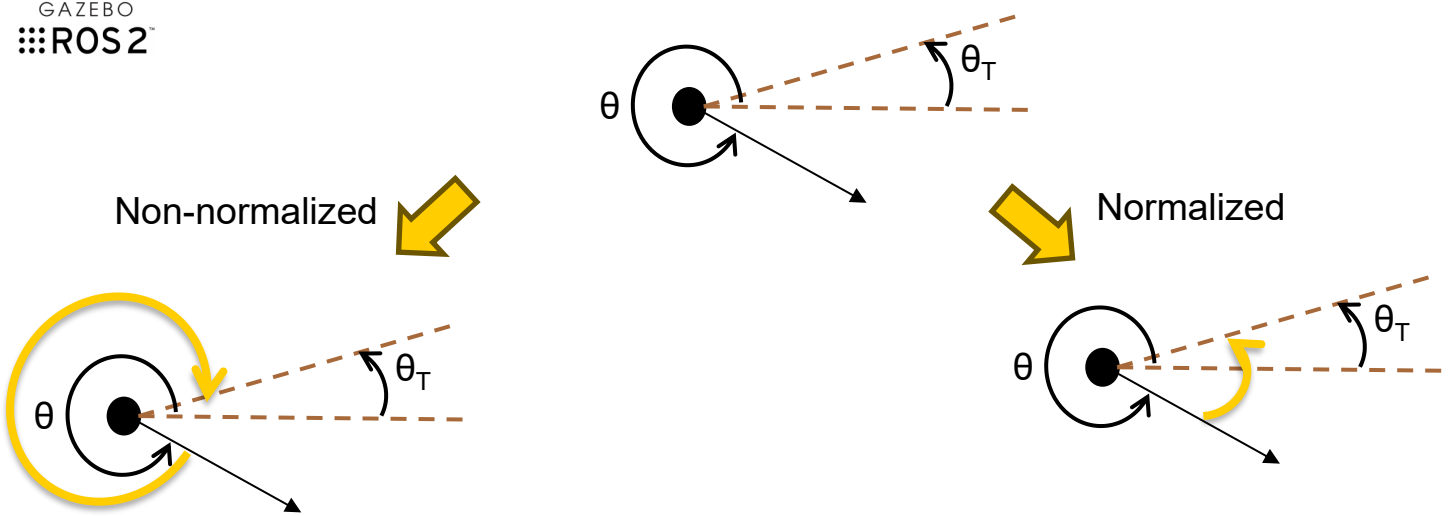
Adjusting Direction



Adjusting Direction



Adjusting Direction

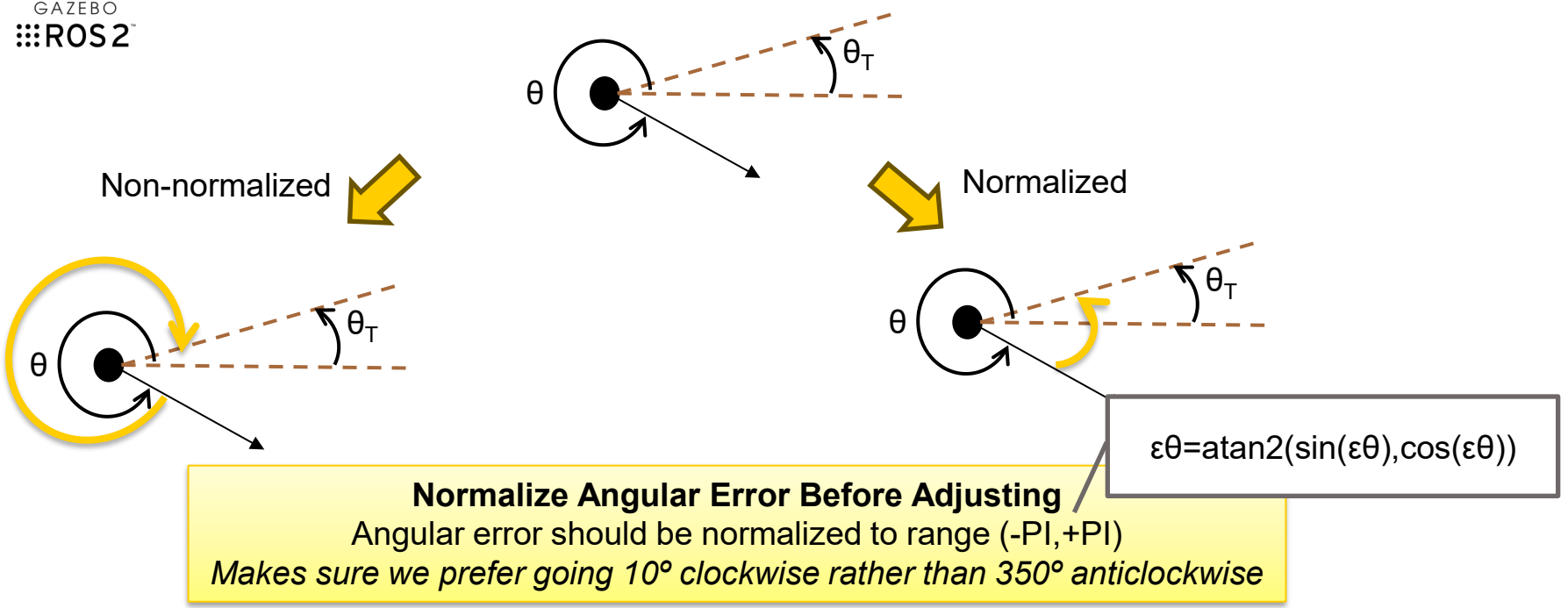


Normalize Angular Error Before Adjusting

Angular error should be normalized to range (-PI,+PI)

Makes sure we prefer going 10º clockwise rather than 350º anticlockwise

Adjusting Direction

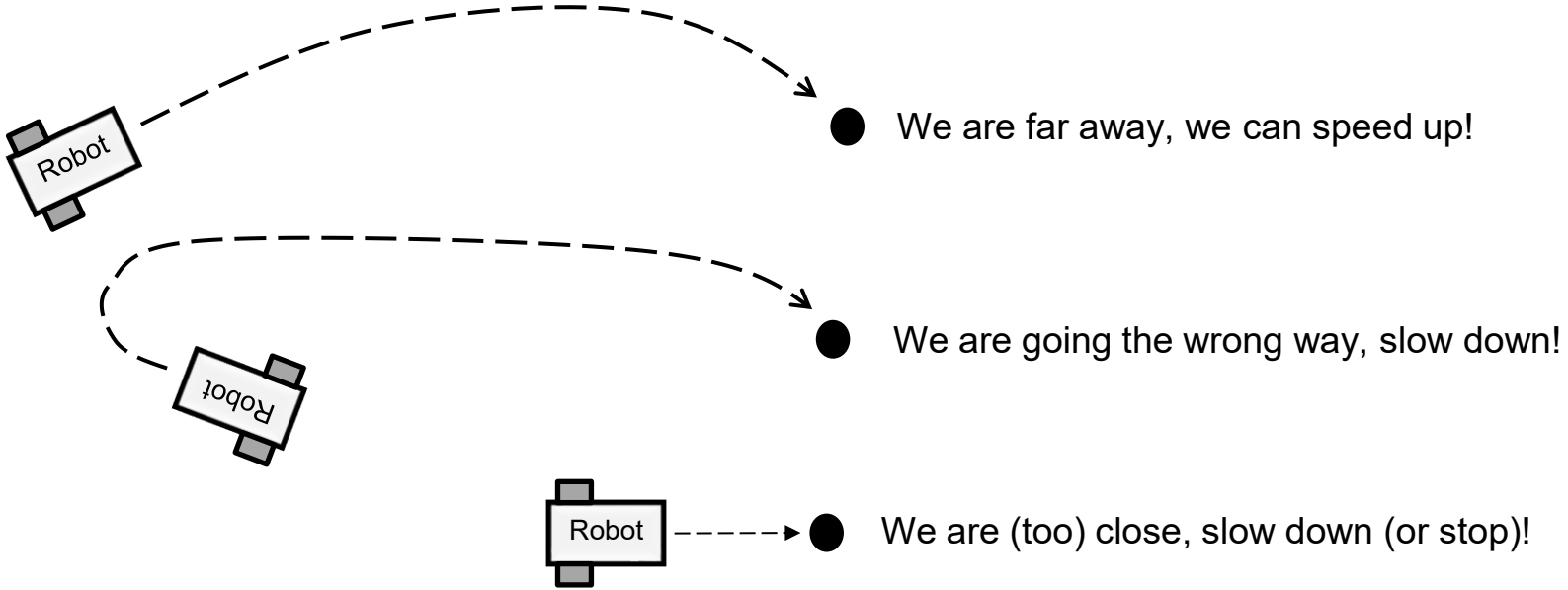


Normalize Angular Error Before Adjusting

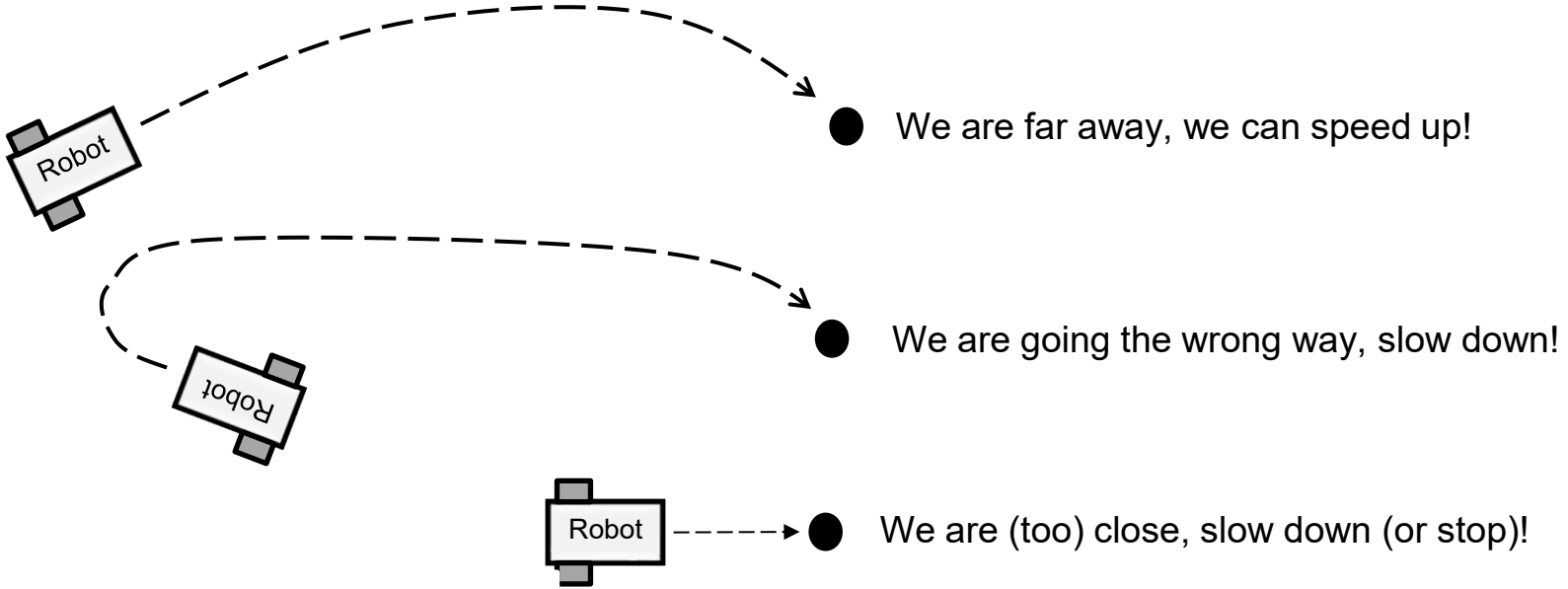
Angular error should be normalized to range (-PI,+PI)

Makes sure we prefer going 10º clockwise rather than 350º anticlockwise

Managing Linear Speed

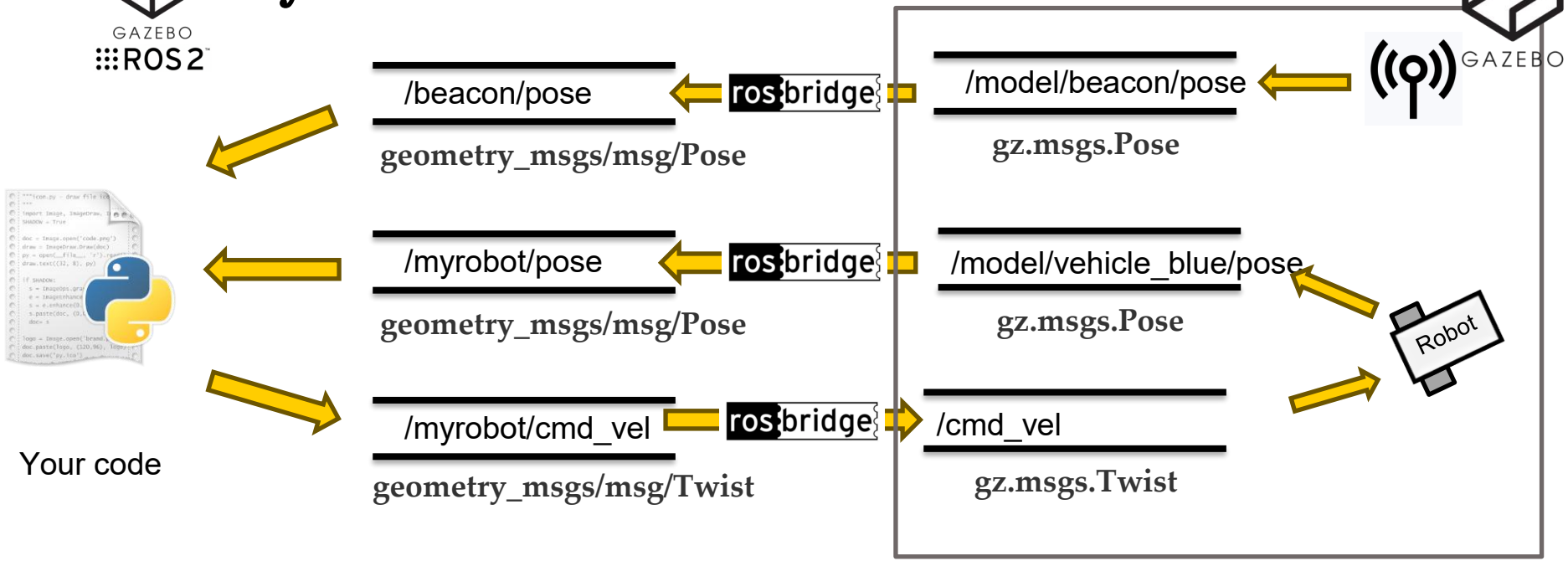


Managing Linear Speed



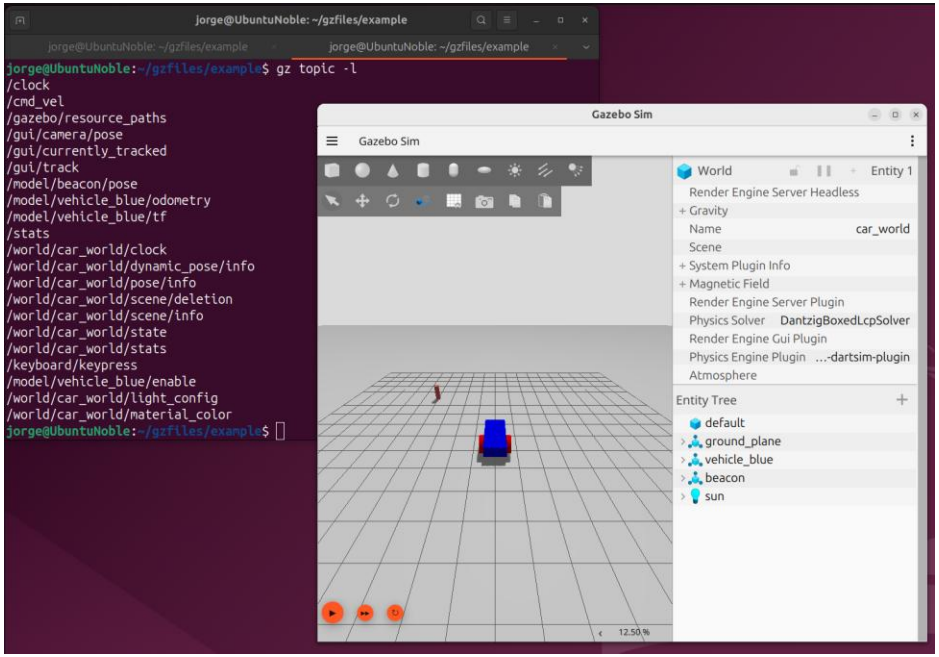
$$v = clip(Kd * dT, 0, vmax) * max(0, cos(\varepsilon\theta))$$

System Architecture With Simulation



Creating the Beacon Simulation

- We will add a model beacon to our simulation world file `worldWithBeacon.sdf`.
  - It is a model with cylinder geometry
  - It uses the `PosePublisher` plugin to publish its pose (`gz.msgs.Pose`) on the `/model/beacon/pose` Gazebo topic.



(see all the terminal commands in slide 53

System Architecture With Simulation

Your code

/beacon/pose

geometry_msgs/msg/Pose

ros bridge

/model/beacon/pose

gz.msgs.Pose

/myrobot/pose

geometry_msgs/msg/Pose

ros bridge

/model/vehicle_blue/pose

gz.msgs.Pose

Robot

/myrobot/cmd_vel

geometry_msgs/msg/Twist

ros bridge

/cmd_vel

gz.msgs.Twist

Getting Robot Position

- We also need to make sure that the pose of the robot is made available. We resort to the same technique:
  - We use the Odometry information we have already published from Gazebo and extract position and rotation information.
- We have everything we need setup to send the velocity command too.

System Architecture With Simulation

Your code

/beacon/pose

geometry_msgs/msg/Pose

ros bridge

/model/beacon/pose

gz.msgs.Pose

GAZEBO

/myrobot/pose

geometry_msgs/msg/Pose

ros bridge

/model/vehicle_blue/pose

gz.msgs.Pose

Robot

/myrobot/cmd_vel

geometry_msgs/msg/Twist

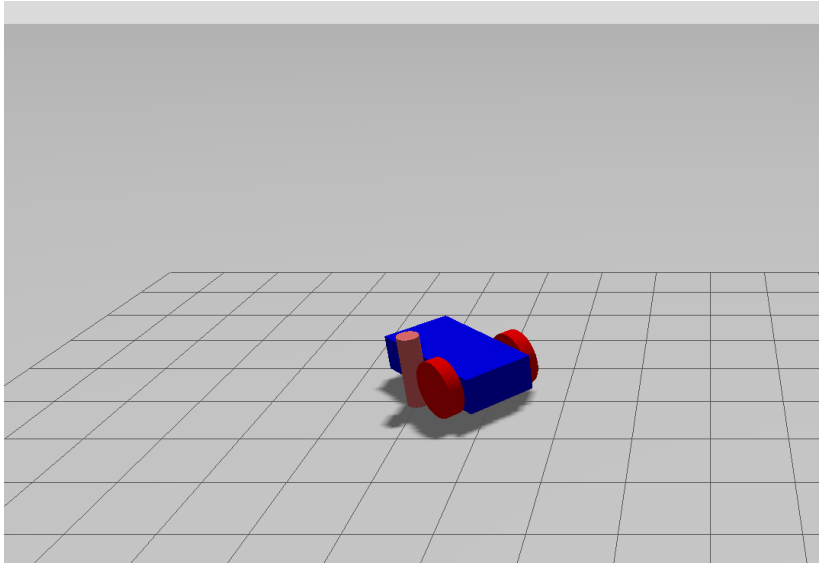
ros bridge

/cmd_vel

gz.msgs.Twist

Setting up the Bridge and Executing

- To tie everything together, we now need to configure the bridge to make the necessary adaptations between ROS and Gazebo topics.
- Then run the controller and watch the robot move to the beacon and stop.



Additional Considerations

- Control rate matters
  - Too low – jerky movement
  - Too high – computational effort, can acting on old data.
  - Should not be directly tied to sensor data acquisition rate (these can change, be too low, too high)
- Data acquisition rate matters
  - Too low – controller does not act on recent data
- Act on most recent data
  - Older unprocessed messages do not really matter, only most up to date information is relevant
- Don't act on stale data
  - If most recent data is too old, don't act on it -> network loss, etc

Running Everything

Start the controller

Terminal 1

```
source /opt/ros/jazzy/setup.bash
python3 navigator5.py
```

Terminal 3

```
gz sim worldBeacon.sdf
```

Terminal 2

```
source /opt/ros/jazzy/setup.bash
ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge config_file:=./bridgeFinal.yaml
```

Running Everything

Terminal 1

```
source /opt/ros/jazzy/setup.bash
python3 navigator5.py
```

Setup the bridge

Terminal 3

```
gz sim worldBeacon.sdf
```

Terminal 2

```
source /opt/ros/jazzy/setup.bash
ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge config_file:=./bridgeFinal.yaml
```

Running Everything

Start the simulator

Terminal 1

```
source /opt/ros/jazzy/setup.bash
python3 navigator5.py
```

Terminal 3

```
gz sim worldBeacon.sdf
```

Terminal 2

```
source /opt/ros/jazzy/setup.bash
ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge config_file:=./bridgeFinal.yaml
```

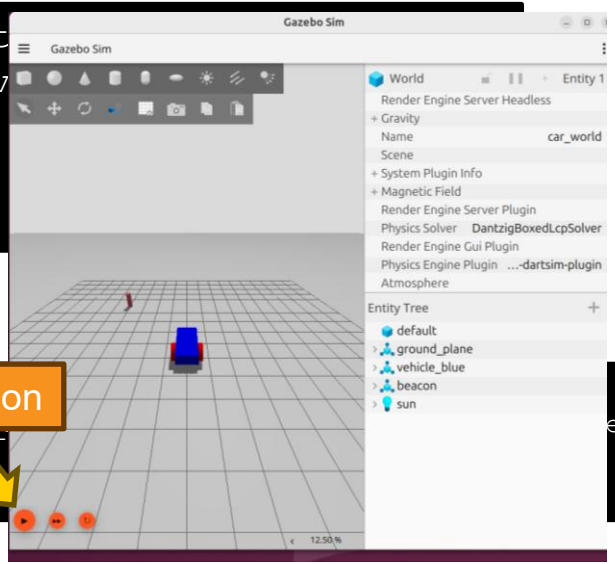
Running Everything

Terminal 1

```
source /opt
python3 nav
```

Terminal 3

```
gz sim worldBeacon.sdf
```



Terminal 2

```
s
r                                                    e_name:=ros_gz_bridge config_file:=./bridgeFinal.yaml
```

Run the simulation