Ast x Oru Robotics Summer School

Andreas Persson, David Caceres Dominguez, Pedro Zuidberg Dos Martires

Örebro University

github.com/pedrozudo/astxoru-roboticssummerschool

Recap from Last Week

+ ROS

Recap from Last Week

- `Sensor` class for...
- Reading BrickBi3 sensor
- Publishing ROS message

```python
#!/usr/bin/env python
import rospy                          # Import the ROS Python library
from std_msgs.msg import Bool         # Import Bool message type from standard messages
import brickpi3                       # Import the BrickPi3 drivers

'''A class for handling sensor(s).'''
class Sensor:
    def __init__(self): # Class constructor

        # Create a publisher
        self.pub = rospy.Publisher('/touch/reading', Bool, queue_size=10)

        # Create BrickPi3 instance
        self.BP = brickpi3.BrickPi3()

        # Configure for a touch sensor on connector S1
        self.BP.set_sensor_type(self.BP.PORT_1, self.BP.SENSOR_TYPE.TOUCH)

    # Method for reading and publishing sensor values
    def read(self):
        try:
            value = self.BP.get_sensor(self.BP.PORT_1)
            self.pub.publish(value)
        except brickpi3.SensorError:
            pass

    # Method for "unconfigure" all sensors and motors
    def reset(self):
        self.BP.reset_all()
```

Recap from Last Week

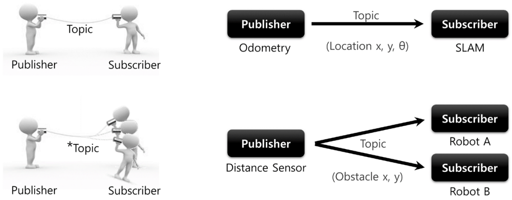
- Sensor reading published as...
- `Bool` message type
- From standard messages (`std_msgs`)

```python
#!/usr/bin/env python
import rospy                          # Import the ROS Python library
from std_msgs.msg import Bool         # Import Bool message type from standard messages
import brickpi3                       # Import the BrickPi3 drivers

'''A class for handling sensor(s).'''
class Sensor:
    def __init__(self): # Class constructor

        # Create a publisher
        self.pub = rospy.Publisher('/touch/reading', Bool, queue_size=10)

        # Create BrickPi3 instance
        self.BP = brickpi3.BrickPi3()

        # Configure for a touch sensor on connector S1
        self.BP.set_sensor_type(self.BP.PORT_1, self.BP.SENSOR_TYPE.TOUCH)

    # Method for reading and publishing sensor values
    def read(self):
        try:
            value = self.BP.get_sensor(self.BP.PORT_1)
            self.pub.publish(value)
        except brickpi3.SensorError:
            pass

    # Method for "unconfigure" all sensors and motors
    def reset(self):
        self.BP.reset_all()
```

Recap from Last Week

In fact, there are many different ROS message types...

- `std_msgs` standard messages
- `sensor_msgs` sensor messages
- `geometry_msgs` geometric primitives
- `nav_msgs` navigation messages
- ...



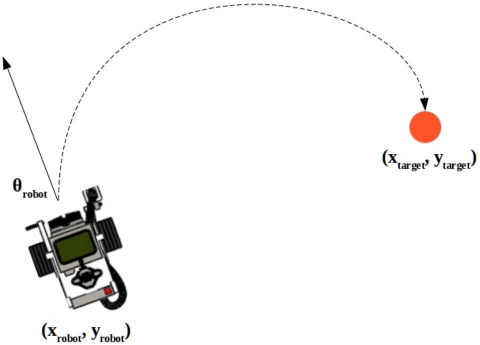
*Topic not only allows 1:1 Publisher and Subscriber communication, but also supports 1:N, N:1 and N:N depending on the purpose.

Navigation

- *GoTo Behaviour*
- Make the robot go from point **A** to point **B**

Navigation

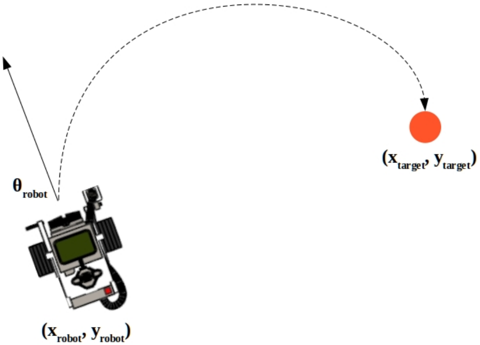
- *Whats needed?*
- Position and orientation of the **robot**
- Position of the **target** (position)
- Robot wheel configuration and dimensions



- Also, see Luis slides: `github.com/pedrozudo/astxoru-roboticssummerschool/lectures_luis/slides.pdf`

Navigation

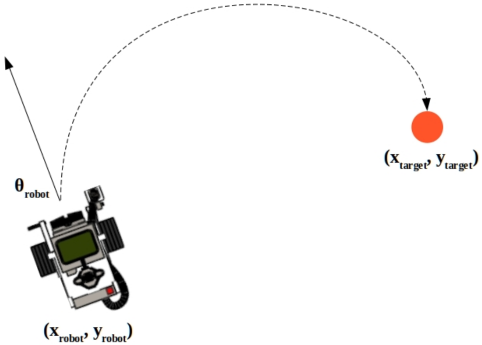
- **Task 1:**
  1. Assume that the **robot start** at position and orientation `(0.0, 0.0, 0.0)`, and;
  2. Write the *GoTo behaviour* that makes the robot go the a given arbitrary **target position**



Navigation

- Theoretical background (simplified):
  - V - linear velocity
  - ω - angular velocity
  - r - wheel radius
  - b - width between wheels

- Angular velocity for left resp. right wheel:
  - $\omega_L = (V - \omega * b/2)/r$
  - $\omega_R = (V + \omega * b/2)/r$

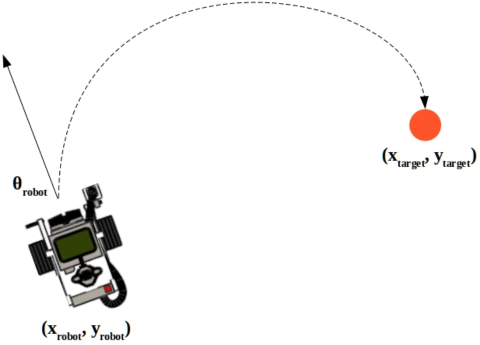


Navigation

- $\delta_x = x_{target} - x_{robot}$
- $\delta_y = y_{target} - y_{robot}$

- V - linear velocity:
  - ...can be set arbitrary, or;
  - Proportional to the *distance*: $d = \sqrt{\delta_y^2 + \delta_x^2}$

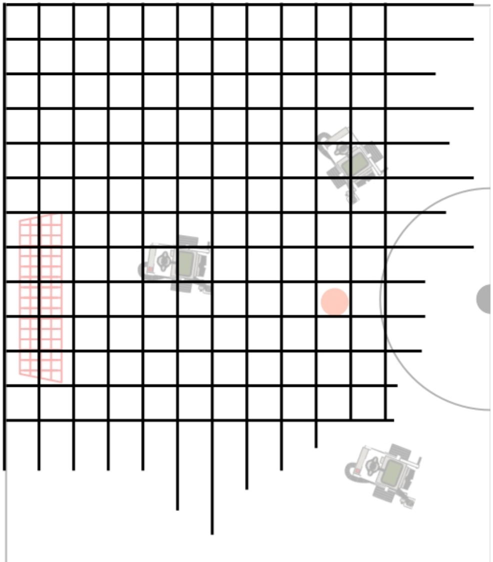
- ω - angular velocity:
  - $\omega = atan2(\delta_y, \delta_x) - \theta_{robot}$
  - $\omega = \begin{cases} \omega - 2\pi & \text{if } \omega > \pi \\ \omega + 2\pi & \text{elif } \omega < -\pi \\ \omega & \text{otherwise} \end{cases}$



Path Planning

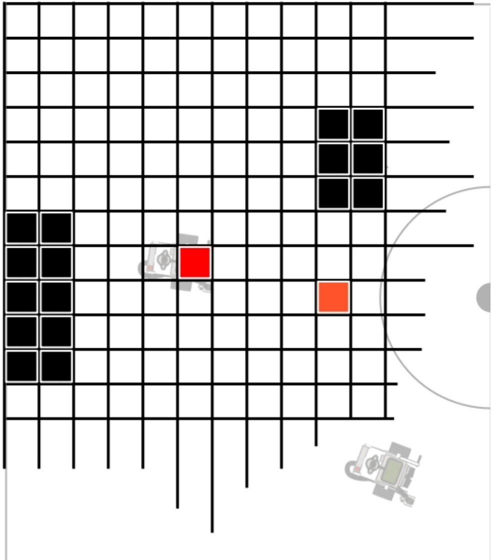
Path Planning

- *Gridmap* representation
- Game plane: `4.16 x 2.60 m`
- Represented by 2D matrix
- *Question though, which resolution?*



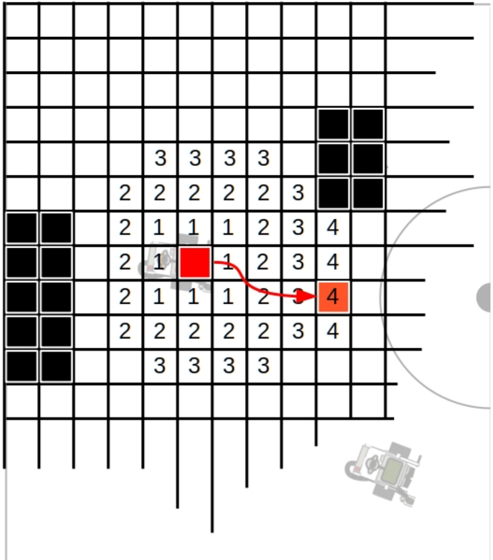
Path Planning

- *Populate* the *gridmap* with:
  - Occupied cells
  - Cell of robot(s)
  - Target cell(s) (e.g. the ball)



Path Planning

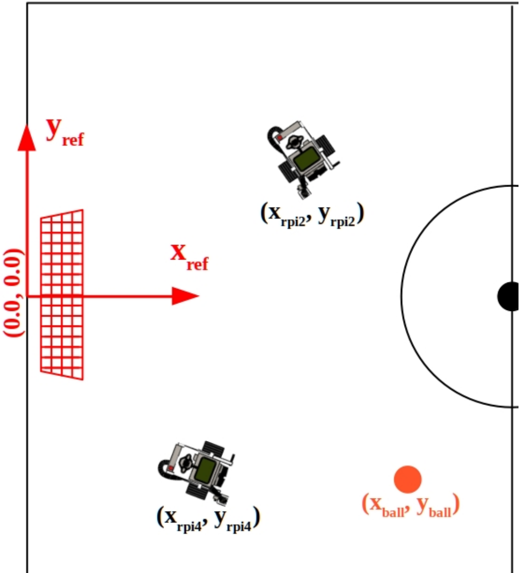
- *Find* the path from robot to target
- *Pathfinding* algorithms:
  - Dijkstra's search algorithm
  - A* search algorithm
  - ...
- *Implementation of A* algorithm* (`astar.py`) *can be found on GitHub!*



Path Planning

- Positions given by *AR-tags* for...
  - Each robot
  - Each goal
  - Ball

- Two separate reference frames (**A** and **B**) with *origin in each goal cage*



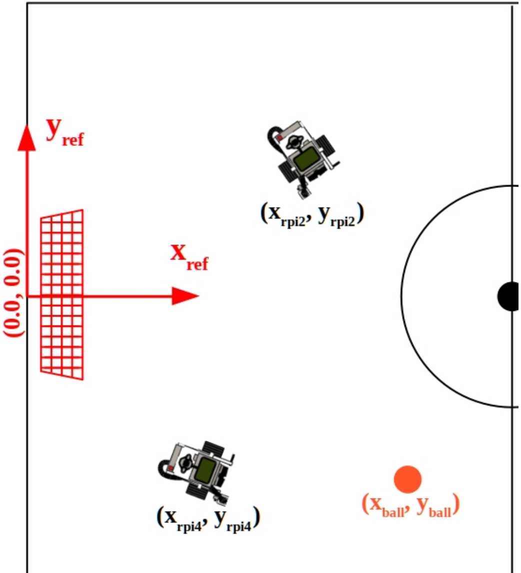
Path Planning

- Positions given as (`Point2D`) message type (from `geometry_msgs`)
- Positions published on separate ROS topics, e.g.:

  `/a/ball/position`

- See all available ROS topics (in Terminal):

  `rotopic list`



Path Planning

- **Task 2:**
  1. Get the position of robots, goal cages, and the ball through *ROS subscribers*;
  2. Translate the positions into a *gridmap representation*, and;
  3. Plan and follow a path using your *GoTo behaviour* (from **Task 1**).