[Lab 3]

SLAM configuration and Navigation

Resources

[1] GitHub of the lab: https://github.com/emarche/Mobile-robotics-4S009023-UniVR

Requirements

- Setup of L1, L2, L3

Bug

Bug algorithms are simple (local) motion planning algorithms.

They rely on few assumptions:

- The robot can track (or retrieve) its position and orientation
- The robot can detect obstacles
- We have a defined goal (i.e., its position).

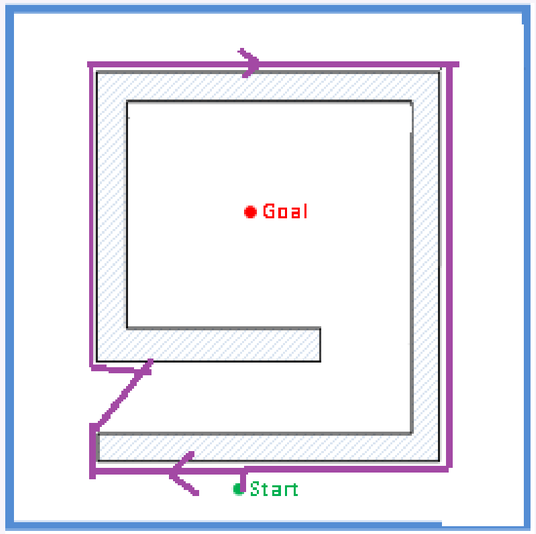
Bug0

The Bug0 algorithm is a simple, greedy algorithm that goes toward the goal until it hits an obstacle, then goes around the obstacle until it can go toward the goal again.

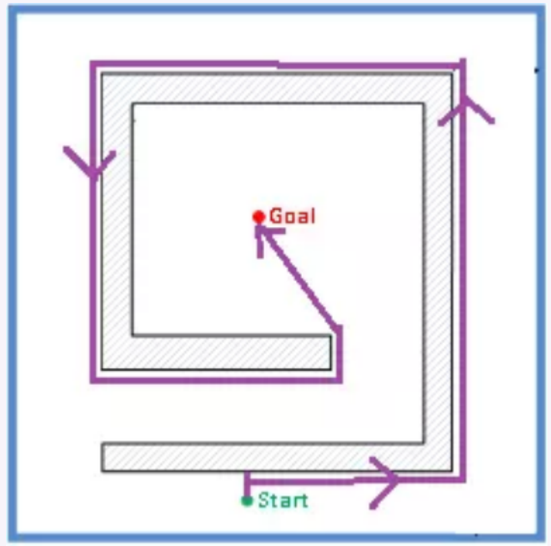
```
While not at Goal:
    Move towards Goal
    If hit obstacle:
        Turn (right\left)
        Move around obstacle until path to Goal is clear
    end if
end while
```

Left/Right Bug0

The Bug0 algorithm can fail in some environments, depending on the turning strategy (i.e., always turn right or left).



Here turning left fails, ending up turning in circles.

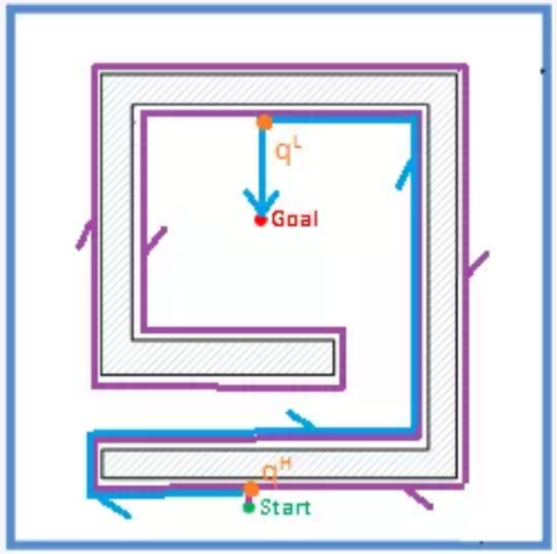


While turning right can navigate to the goal.

Bug1

There are some possible improvements for Bug0 that allows to solve the previous issues.

- Bug1 is an exhaustive search algorithm: on hitting an obstacle, it will follow around the entire obstacle looking for the point on the obstacle that is closest to the goal. It will then follow the obstacle back to that point before moving towards the goal again.



Another improvement is Bug2, which tracks and follow the line between the initial position of the robot and the goal.

Lidar setup for Bug0

For simplicity we will try to implement Bug0 with a rospy node.

First, modify the turtlebot3_description files to visualize laser scans in Gazebo:
workspace/src/turtlebot3/turtlebot3_description/urdf/turtlebot3_waffle_pi.urdf.xacro

```
<robot name="turtlebot3_waffle_pi_sim" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:arg name="laser_visual"  default="true"/>
  <xacro:arg name="camera_visual" default="false"/>
  <xacro:arg name="imu_visual"    default="false"/>
```

Now if you launch a simulation environment (e.g., turtlebot3_world), you will see the scans.

Lidar setup

We will also modify the n° of scans and angle, to simplify the process of coding bug. In the same file you can change these settings.

```
<scan>
  <horizontal>
    <samples>360</samples>
    <resolution>1</resolution>
    <min_angle>0.0</min_angle>
    <max_angle>6.28319</max_angle>
  </horizontal>
</scan>
```

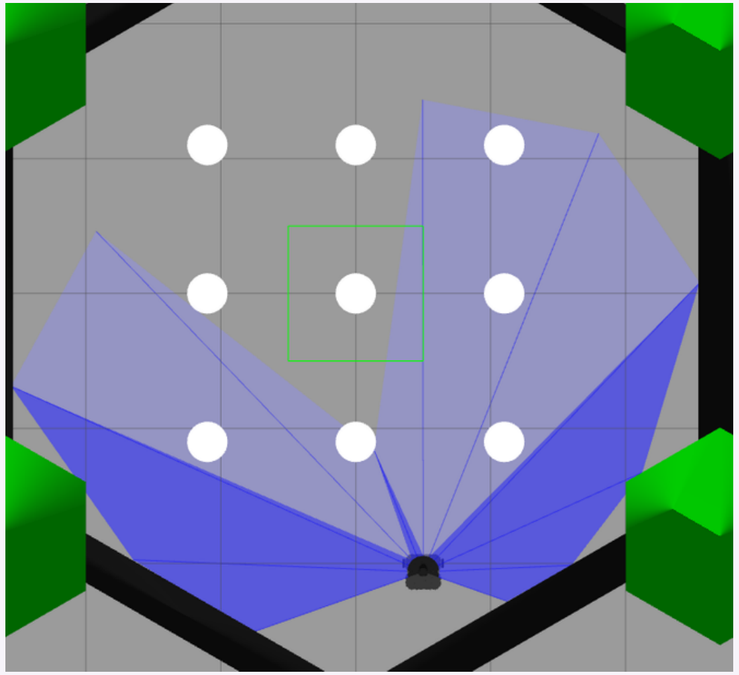
The simulated LDS-01 returns 360 scans sampled in range [0, 6.28] rad, i.e., [0, 360] deg.

```
<scan>
  <horizontal>
    <samples>11</samples>
    <resolution>1</resolution>
    <min_angle>-1.91</min_angle>
    <max_angle>1.91</max_angle>
  </horizontal>
</scan>
```

We are interested in frontal and side obstacles so we will sample 11 scans in range [-1,91, 1.91] rad, , i.e., [-110, 110] deg.

Lidar setup

Launching again the simulation environment you will see new setting for the lidar and visualize the lidar topic (i.e., rostopic echo /scan) to retrieve the index of each scan.



We note that the value at index 0 corresponds to the rightmost scan, and then indexes proceed in anti-clockwise order.

Environment

You will find the Gazebo environment for the Bug algorithm in the course's GitHub:

- "tb3_bug.launch" -> turtlebot3_simulations/turtlebot3_gazebo/launch
- "turtlebot3_bug" folder -> turtlebot3_simulations/turtlebot3_gazebo/models
- "turtlebot3_bug.world" -> turtlebot3_simulations/turtlebot3_gazebo/worlds

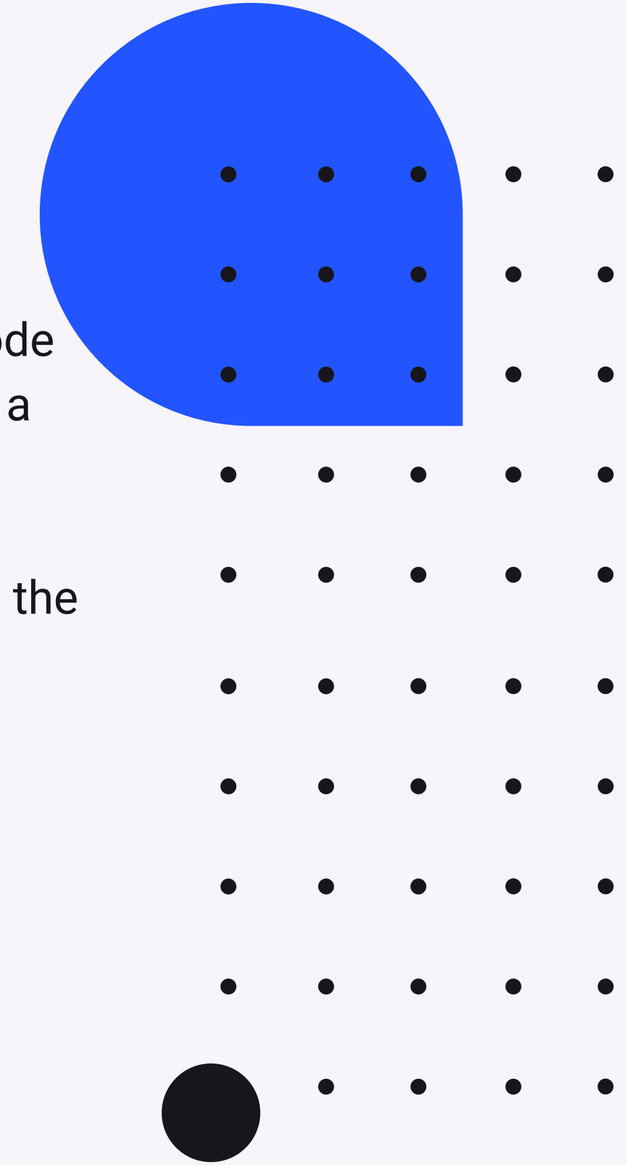
Hence, you can launch the simulation environment with the usual command:

```
roslaunch turtlebot3_gazebo turtlebot3_bug.launch
```

Python Bug0

We provide an initial Python script, "turtlebot3_bug.py" with useful functions to code the Bug0 algorithm. You can place the script in any folder and simply launch it as a python program.

Notice that if you compiled ROS packages using python 2, you will have to launch the program using python2



class Bug0

The class Bug0 initializes different useful components such as desired goal position and threshold for reaching, robot velocities, ...
Moreover it subscribes to the cmd_vel topic to publish velocity messages, and create a listener for the odometry frame to retrieve robot's odometry.

```
self.cmd_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)     # subscribe as publisher
to cmd_vel for velocity commands
self.tf_listener = tf.TransformListener()
self.odom_frame = 'odom'

try:
    self.tf_listener.waitForTransform(self.odom_frame, 'base_footprint', rospy.Time(),
    rospy.Duration(1.0))
    self.base_frame = 'base_footprint'
except (tf.Exception, tf.ConnectivityException, tf.LookupException):
    try:
        self.tf_listener.waitForTransform(self.odom_frame, 'base_link', rospy.Time(), rospy.
        Duration(1.0))
        self.base_frame = 'base_link'
    except (tf.Exception, tf.ConnectivityException, tf.LookupException):
        rospy.loginfo("Cannot find transform between odom and base_link or base_footprint")
        rospy.signal_shutdown("tf Exception")
```

Get Odometry

The function to get robot's odometry is already implemented and returns robot position and rotation.

```
def get_odom(self):
    try:
        (trans, rot) = self.tf_listener.lookupTransform(self.odom_frame, self.base_frame, rospy.
        Time(0))
        rot = euler_from_quaternion(rot)

    except (tf.Exception, tf.ConnectivityException, tf.LookupException):
        rospy.loginfo("TF Exception")
        return

    return Point(*trans), np.rad2deg(rot[2])
```

Get Scan

The function to get robot's scan values is already implemented and returns the scan values.

```
def get_scan(self):
    scan_topic = rospy.wait_for_message('scan', LaserScan)    # wait for a scan msg to retrieve
    recent sensor's reading

    # back_right, right, front, left, back_left
    #scan_filter = [scan.ranges[0], scan.ranges[1], scan.ranges[5], scan.ranges[-2], scan.ranges
    [-1]]
    scan = [scan_topic.ranges[i] for i in range(len(scan_topic.ranges))]
    for i in range(len(scan)):        # cast limit values (0, Inf) to usable floats
        if scan[i] == float('Inf'):
            scan[i] = 3.5
        elif math.isnan(scan[i]):
            scan[i] = 0

    return scan
```

Get Goal Information

The function to get info on the specified goal is already implemented and returns the distance of the robot from the goal, and the required angle to face the goal straightly.

```
def get_goal_info(self, tb3_pos):
    distance = sqrt(pow(self.goal_x - tb3_pos.x, 2) + pow(self.goal_y - tb3_pos.y, 2))  #
    compute distance wrt goal
    heading = atan2(self.goal_y - tb3_pos.y, self.goal_x- tb3_pos.x)    # compute heading to
    the goal in rad

    return distance, np.rad2deg(heading)     # return heading in deg
```

Publishing Velocity

A code sample to publish velocity commands is provided in the demo function "move".
Just remember that angular velocity is specified in rad/s, and linear velocity is in m/s and has a maximum value of 0.21

```
def move(self):
    move_cmd = Twist()
    move_cmd.linear.x = 0.2
    move_cmd.angular.z = 0.15
    self.cmd_pub.publish(move_cmd)
```

Exercise

The goal is to implement the Bug0 algorithm with the turn left strategy in the provided environment.