Gazebo and RViz

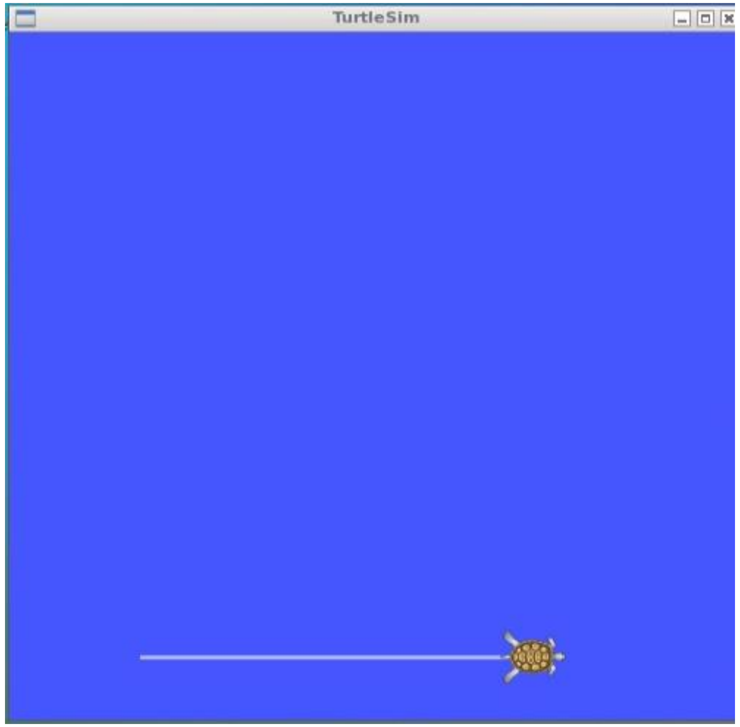
Carmine Tommaso Recchiuto

Launch files

The turtlebot_controller package has been updated!

https://github.com/CarmineD8/turtlebot_controller

> roslaunch turtlebot_controller exercise3



ROS – Launch Files

<remap>

Remapping allows you to "trick" a ROS node so that when it thinks it is subscribing to or publishing to /some_topic it is actually subscribing to or publishing to /some_other_topic, for instance.

Sometimes, you may need a message on a specific ROS topic which normally only goes to one set of nodes to also be received by another node. If able, simply tell the new node to subscribe to this other topic. However, you may also do some remapping so that the new node ends up subscribing to /needed_topic when it thinks it is subscribing to /different_topic.

This could be accomplished like so:

<remap from="/different_topic" to="/needed_topic"/>

The remap tag can be used within a <node> tag, and in that case it will remap will apply just to that specific node, or generally in the launch file, and in this case it will apply to the lines following the remap.

ROS – Launch Files

How to set parameters in the launch file?

- ✓ Tags <param> and <rosparam>

The tag <param> defines a parameter to be set on the Parameter Server. The <param> tag can be put inside of a <node> tag, in which case the parameter is treated like a private parameter.

Es.

```
<param name="publish_frequency" type="double" value="10.0" />
```

Type may be str, int, double, bool or yaml.

ROS – Launch Files

How to set parameters in the launch file?

✓ Tags <param> and <rosparam>

Similarly, the tag <rosparam> gives the possibility of loading or deleting parameters from the ROS Parameter Server.

```
<rosparam command="load" file="$(find rosparam)/example.yaml" />
<rosparam command="delete" param="my/param" />
```

Example

```
<launch>

    <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node">
        <param name="background_b" type="int" value="0"/>
        <param name="background_g" type="int" value="0"/>
        <param name="background_r" type="int" value="255"/>
    </node>
    <node name="harmonic_server" pkg="my_srv" type="harmonic_server" />
    <node name="exercise3" pkg="turtlebot_controller" type="exercise3" />

</launch>
```

or

```
<launch>

    <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node"/>

    <param name="turtlesim_node/background_b" type="int" value="0"/>
    <param name="turtlesim_node/background_g" type="int" value="0"/>
    <param name="turtlesim_node/background_r" type="int" value="255"/>

    <node name="harmonic_server" pkg="my_srv" type="harmonic_server" />
    <node name="exercise3" pkg="turtlebot_controller" type="exercise3" />

</launch>
```

ROS – Robotic simulations

[We are ready to start our first simulation of a robot in a 3D simulation environment: CarmineD8/robot_description: Package for working with mobile robot simulations with ROS and Gazebo (github.com)*]

Let’s take the package robot_description. As first step, let’s see the structure of the package.

```
.
├── CMakeLists.txt
├── config
│   ├── sim.rviz
│   └── sim2.rviz
├── include
│   └── robot_description
├── launch
│   ├── sim.launch
│   ├── sim2.launch
│   └── sim_w1.launch
├── package.xml
├── src
├── urdf
│   ├── robot2.gazebo
│   ├── robot2.xacro
│   ├── robot2_laser.gazebo
│   └── robot2_laser.xacro
└── worlds
    ├── world01.world
    └── world02.world
```

config -> configuration files for simulation
launch -> roslaunch files
urdf -> robot description files
worlds -> environments for simulation
scripts -> ros nodes, we will see it later

*do not forget to switch on the branch Noetic

ROS – Rviz and Gazebo

✓ roslaunch robot_description sim.launch

We have now two windows, Rviz and Gazebo

Rviz is a tool for ROS Visualization. It's a 3-dimensional visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions.

Gazebo is the 3D simulator for ROS

The robot may be controlled using ROS topics (/cmd_vel) (a nice tool is teleop_twist_keyboard, which may be launched with rosrun teleop_twist_keyboard teleop_twist_keyboard.py). When moving the robot around, information coming from sensors may be visualized in Rviz (ex: odom, or cameras).

ROS – Rviz and Gazebo

Let's check more carefully the launch file.

- ✓ We add the robot description in the ROS parameter server
- ✓ We launch the simulation in an empty world
- ✓ We launch the node RVIZ, together with some additional nodes
- ✓ We spawn our robots in the simulation

ROS – Rviz and Gazebo

More details about steps 2 and 3!

Gazebo

- Dynamic simulation based on various physics engines (ODE, Bullet, Simbody and DART)
- Sensors (with noise) simulation
- Plugin to customize robots, sensors and the environment
- Realistic rendering of the environment and the robots
- Library of robot models
- ROS integration

Advanced features

- Remote &cloud simulation
- Open source

ROS – Rviz and Gazebo

Gazebo is composed by:

- A server gzerver for simulating the physics, rendering and sensors
- A client gzclient that provides a graphical interface to visualize and interact with the simulation

The client and the server communicate using the gazebo communication library

This may be seen by analyzing the launch file included (empty_world.launch in the gazebo_ros package)

- Two different nodes are started, one for the GzServer, and one for the GzClient
- You may also notice all parameters defined in the launch file

ROS – Rviz and Gazebo

When launching Rviz, three nodes are actually executed:

- joint_state_publisher
- robot_state_publisher
- rviz

- *joint_state_publisher*: the package reads the robot_description parameter from the parameter server, finds all of the non-fixed joints and publishes a JointState message with all those joints defined. If GUI is present, the package displays the joint positions in a window as sliders.

- *robot_state_publisher*: the package uses the URDF specified by the parameter robot_description and the joint positions from the topic joint_states to calculate the forward kinematics of the robot and publish the results via *tf*.

ROS – Rviz and Gazebo

- ✓ Rviz is executed by specifying a configuration file, which sets the elements that we want to display in the simulation.
- ✓ In the example, we specify the fixed frame (odom) and that we want to visualize the robot structure and the output of the camera.
- ✓ Topics or visualization elements may be added by selecting them from the add menu.
- ✓ By selecting "odom" as fixed frame, we may visualize the movement of the robot also in Rviz. This may be more evident, by adding the visualization of the tf

ROS – Rviz and Gazebo

- ✓ The sim2.launch roslaunch file corresponds to the same simulation, but with a slightly different robot: it has a laser sensor instead of a camera.

- ✓ The launch file is thus similar to the previous one, but we are now loading a different urdf file as robot_description parameter in the ROS parameter server, and we are starting Rviz with a different configuration file: indeed, we are going to visualize the laser sensor instead of the camera output.

- ✓ Please notice that, differently from images, the laser output may be seen directly in the corresponding frame

ROS – Rviz and Gazebo

- ✓ Finally sim_w1.launch uses a different environment for the simulation (environments have been stored in the folder worlds).
- ✓ Here in the launch file we explicitly launch the gazebo client and the server (we cannot include anymore the empty_world.launch).
- ✓ The world has been defined with a default value, so this may be overridden when launching the simulation (es. roslaunch robot_description sim_w1.launch world:=world01)

Mobile robots: planning the motion

✓ To actually plan the motion of our robot in an environment we need to process the output of the sensors

- The node *reading_laser.py* converts the 720 readings contained inside the LaserScan msg into five distinct readings. Each reading is the minimum distance measured on a sector of 60 degrees (total 5 sectors = 180 degrees).

- Moving the robot in the environment we may check if the laser data are correctly updated

- rosrun robot_description reading_laser.py

Mobile robots: planning the motion

- ✓ Let’s now use the information for controlling the robot in the environment. For example, we can let the robot move around but avoiding obstacles!

- ✓ obstacle_avoidance.py implements a very simple behaviour: if an obstacle is detected on the front (or front-right or front-left) rotate until there are no obstacles perceived. If the obstacle is perceived on the right, than rotate on the left, and viceversa.

- rosrun robot_description obstacle_avoidance.py

Mobile robots: planning the motion

✓ Let's complicate a little bit the behaviour: I want now a robot which follow the walls!

- The functions defined are:

  - main : This is the entry point for the algorithm, it initializes a node, a publisher and a subscriber. Depending on the value of the state_ variable, a suitable control action is taken (by calling other functions). This function also configures the frequency of execution of control action using Rate function.

  - clbk_laser : This function is passed to the Subscriber method and it executes when a new laser data is made available. This function writes distance values in the global variable regions_ and calls the function take_actions

  - take_action : This function manipulates the state of the robot. The various distances stored in regions_ variable help in determining the state of the robot.

Mobile robots: planning the motion

- ✓ Let’s complicate a little bit the behaviour: I want now a robot which follow the walls!

- The functions defined are:

  - find_wall : This function defines the action to be taken by the robot when it is not surrounded by any obstacle. This method essentially makes the robot move in a anti-clockwise circle (until it finds a wall).

  - turn_left : When the robot detects an obstacle it executes the turn left action

  - follow_the_wall : Once the robot is positioned such that its front and front-left path is clear while its front-right is obstructed the robot goes into the follow wall state. In this state this function is executed and this function makes the robot to follow a straight line

- ✓ rosrun robot_description wall_follow.py

Mobile robots: planning the motion

- ✓ But <u>what</u> if I want to reach a specific point in space?

- I need a node able to control the robot: go_to_point.py (please notice that in this case the robot is non-holonomic, so things are more complex than in assignment 1)

- It's implemented as a State machine: there are finite number of states that represent the current situation (or behavior) of the system. In our case, we have three states

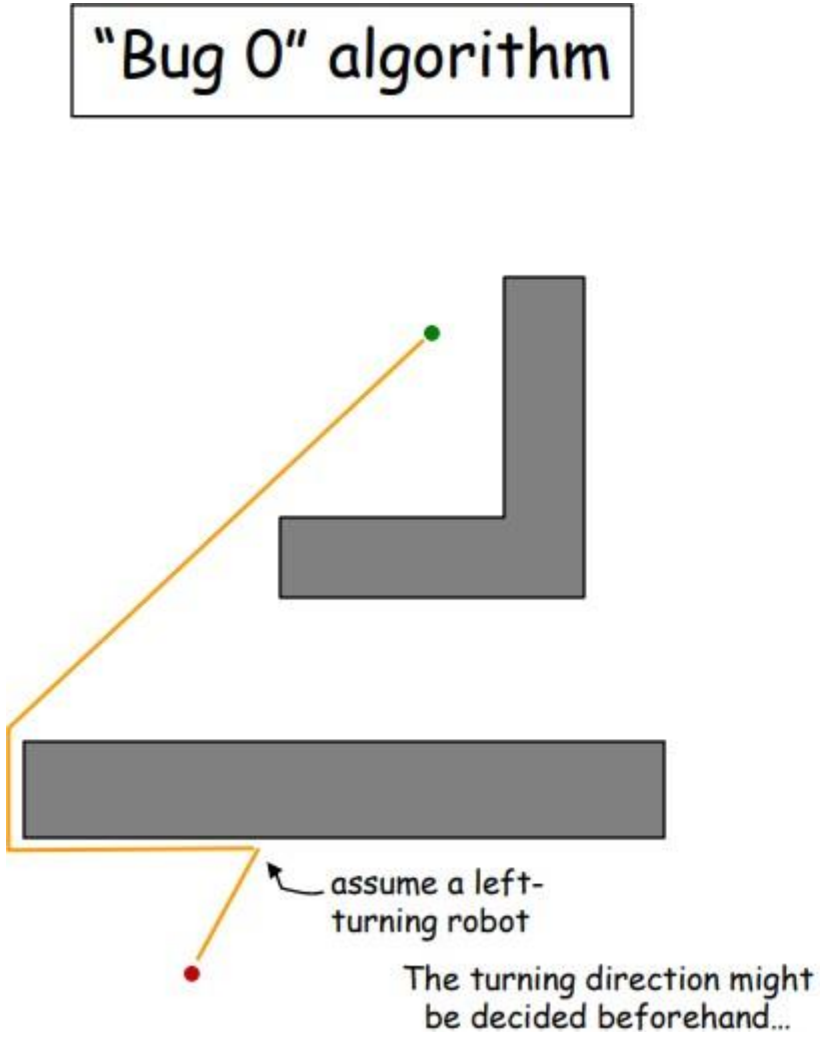
- **Fix Heading** : Denotes the state when robot heading differs from the desired heading by more than a threshold (represented by yaw_precision_ in code)

- **Go Straight** : Denotes the state when robot has correct heading but is away from the desired point by a distance greater than some threshold (represented by dist_precision_ in code)

- **Done** : Denotes the state when robot has correct heading and has reached the destination.

    - ✓ rosrun motion_plan go_to_point.py

Mobile robots: planning the motion



1) head toward goal

2) follow obstacles until you can head toward the goal again

3) continue

Mobile robots: planning the motion

- ✓ We are almost there. What if I want to put together the two behaviors?

- ✓ Services! We may modify the two scripts in order to advertise two services that may be available to the bug0 algorithm

- ✓ We will obtain what it is usually called the **Bug 0 algorithm,** which drives the robot towards a points (goal), but If while doing so if the robot detects an obstacle it goes around it.

- ✓ Let's first modify the go_to_point.py script:

  - ❑ The target position is now retrieved from the ROS parameter server
  - ❑ I add a service server that, when called, set a global variable to True o False
    - ❑ If the variable is true than the algorithm is executed, otherwise nothing is done

- ✓ The same is done for the wall_follow.py script.

Mobile robots: planning the motion

Finally, I need to implement a client able to call the two services.

This has been done in bug.py, which is used to alternative call the two services, depending on the obstacles detected and on the robot’s heading. A launch file has been also built!

roslaunch robot_description bug0.launch des_x:=0 des_y:=8

```
<launch>
   <arg name="des_x" />
   <arg name="des_y" />
   <param name="des_pos_x" value="$(arg des_x)" />
   <param name="des_pos_y" value="$(arg des_y)" />
   <node pkg="robot_description" type="follow_wall_service.py" name="wall_follower" output="screen" />
   <node pkg ="robot_description" type="go_to_point_service.py" name="go_to_point" output="screen" />
<node pkg ="robot_description" type="bug.py" name="go_to_point" output="screen" />
</launch>
```

Assignment

Problem here! The task is blocking and I cannot do anything while the robot is reaching the target.

This is one of the long-running tasks that should be implemented with an action server!

The package assignment_2_2022: https://github.com/CarmineD8/assignment_2_2022
provides an implementation of the same node as an action server

What should you do here?

- Create a new package, in which you will develop three nodes:
  - (a) A node that implements an action client, allowing the user to set a target (x, y) or to cancel it. The node also publishes the robot position and velocity as a custom message (x,y, vel_x, vel_z), by relying on the values published on the topic /odom;
  - (b) A service node that, when called, prints the number of goals reached and cancelled;
  - (c) A node that subscribes to the robot's position and velocity (using the custom message) and prints the distance of the robot from the target and the robot's average speed. Use a parameter to set how fast the node publishes the information.
- Also create a launch file to start the whole simulation. Set the value for the frequency with which node (c) publishes the information.

Assignment

Additional Requirements:

- ***Only for node (a):*** Create a flowchart of your code, or describe it in pseudocode (Pseudocode Examples (unf.edu))

- Add some comments to the code

- Use functions to avoid having a single block of code

- Publish the new package on your own repository. The flowchart (or the pseudocode) should be added to the ReadMe of the repository. (consider using Markdown syntax to write your readme: Basic Syntax | Markdown Guide)

- **Deadline: 09/01/2023**

Evaluation

- Code performance
- Code structure and clarity
- Respect of the requirements
- Organization of the repository (e.g., README in which you describe what the code does (possibly with flowchart or pseudocode), how to run the code, possible improvements, …)