BLG456E, Robotics, Fall 2020

Assignment #1: Exploration

Due Date: See Ninova.

Individual or Group of 2

Summary

Assignment: Program a ROS node to control a Turtlebot to move, avoid obstacles, and explore. The Turtlebot will be simulated in Gazebo.

Submission type: A single C++ or Python file containing code controlling your exploring robot. The file to be submitted is explorer_node_a1_456_cp.cpp or explorer_node_a1_456_python.py. A skeleton for these files should be found in an archive supplied with the assignment description.

More information: In this assignment, you will design a ROS node which is able to move the robot platform in 2D in an unknown (simulated) environment. The robot should move around the simulated environment autonomously while avoiding obstacles using it's laser scanner. You can use the /scan topic to read laser scans and the /cmd_vel_mux/input/navi topic to send movement messages to the robot platform. You should alter the provided skeleton so that the file explorer_node_a1_456_cpp.cpp or explorer_node_a1_456_python.py contains code to control your exploring robot. For this assignment, a report is not necessary, but your code needs to be well-commented and clear, and you may include a readme.txt.

*If you have acquired snippets from online or other sources, document clearly **close to them in the code** from where you obtained them. Any copied code that is more than a snippet or that was not shared freely **is plagiarism** and can result in zero marks on the assignment and possible referral.*

Table of Contents

Marking criteria

- Moving the robot (beyond what the skeleton does).
- Accessing the laser scan (beyond what the skeleton does).
- Moving the robot in a marginally intelligent way.
- Moving the robot somewhat intelligently.
- Marginally successful object avoidance.
- Somewhat successful object avoidance.
- Marginally intelligent exploration.
- Somewhat intelligent exploration.
- Accessing the map.
- Clear code.
- Clear documentation.

If your program does not compile you get zero marks and fail the course (note the VF requirement).

Bonuses avaiable for: Extended use of the existing mapping subsystem for exploration, interesting robot behaviour (e.g. wall-following), brief intelligent analysis of problem, etc. If you think you deserve a bonus, ensure it is documented, preferably in the readme.

Assignments not submitted according to requirements will not be evaluated.

Step 1 – Set up and test the simulator

ROS & Turtlebot Setup

Before starting, make sure you have the necessary tools. For this assignment you will need the Ubuntu 16.04 (or a related flavour) operating system, ROS Kinetic and the Turtlebot Gazebo package package. For instructions on how to get these, see the document “How to install ROS Kinetic and the Turtlebot Simulation” under “Sınıf Dosyaları in Ninova”.

After installing Ubuntu or Kubuntu or Lubuntu or Xubuntu 16.04, the quick way is to run the following commands (with an internet connection):

- `wget https://bitbucket.org/damienjadeduff/456_kinetic_turtlebot/raw/master/install_456_students.sh`
- `chmod u+x install_456_students.sh`
- `./install_456_students.sh`

You may need to type your password during installation.

Set up your workspace

Set up ROS variables

In this section you are going to working in a terminal, so open a terminal. Next, ensure you have set up the ROS environment variables by running:

- `source /opt/ros/kinetic/setup.bash`

More details on why you should need to do this are given at:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment#Managing_Your_Environment

Create ROS workspace

To set up your ROS workspace, follow the instructions at:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment#Create_a_ROS_Workspace

The ROS workspace is where your robot code will live.

The main commands you will need are:

- `mkdir -p ~/catkin_ws/src`
- `cd ~/catkin_ws/src`
- `catkin_init_workspace`
- `cd ~/catkin_ws`
- `catkin_make`

Once this this is completed, **don't forget to run the following command whenever you create a new terminal session**, so that your ROS environment variables are set up appropriately. If you created your ROS workspace in `~/catkin_ws` then run:

- `source ~/catkin_ws/devel/setup.bash`

You can also add that command to the bottom of the file called `~/.bashrc` so that you don't have to run it manually each time you create a new terminal instance.

Basic knowledge

It is advised that you go over the beginner level tutorials in the ROS web site. In particular:

- Navigating the ROS Filesystem
- Creating a ROS Package
- Building a ROS Package
- Understanding ROS Nodes
- Understanding ROS Topics
- Writing a Simple Publisher and Subscriber (C++)
- Or Writing a Simple Publisher and Subscriber (Python)

Test the simulator

Important reminder: Before doing anything with ROS, ensure your environment variables are set up appropriate to your workspace by running (if your workspace is in catkin_ws):

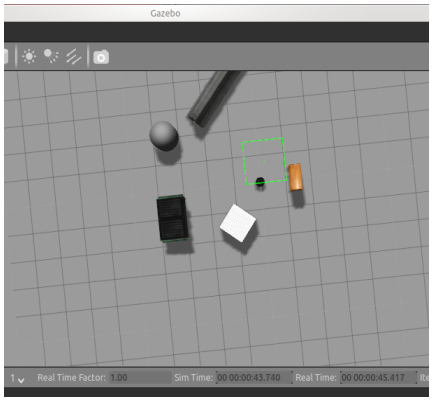
```
source ~/catkin_ws/devel/setup.bash
```

(remember this can be automated by putting the command in .bashrc)

The robot controller that you write will send messages to and receive messages from the simulated Turtlebot robot. To start the simulation, run the following command in a terminal:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

You should see the Turtlebot robot in the Gazebo GUI amongst a scattering of objects:



'Gazebo' is the name of the simulator that ROS uses to simulate robots.

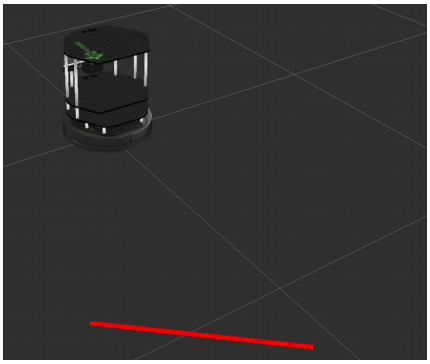
(You can use the Ctrl-C key combination in the terminal in order to close down ROS and all the ROS nodes you started with this command.)

In order to see what your robot sees, in a new terminal window run

```
roslaunch turtlebot_rviz_launchers view_robot.launch
```

Don't forget to first ensure that the setup.bash script mentioned above has been run so that roslaunch will work.

This allows you to visualise the contents of the robots sensors and its own self-model using the 'rviz' program. This same visualiser would be used even if it were a real and not simulated robot that you were using. If you turn on the LaserScan display you will also see the result of the robot's laser scanning as a line of red markers. See the image below.



In order to drive the robot around in its simulated world, you can run the following command and enter directions from the keyboard according to the displayed instructions:

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Only keyboard commands that you enter into that terminal will be registered. You can run the teleoperation application alongside the automated robot controller your write in your assignment to override its behaviour.

It is normal to place the rviz window, the simulator window, the teleoperation terminal, and other terminals alongside each other on your computer display. Most windowing systems have keyboard shortcuts for making this easier and there are tools like *terminator* or *tmux* that may help.

Step 2 – Compile and run the provided skeleton code

What is in the skeleton code

Once you have created your envionment, you can compile and check the skeleton code. The skeleton code consists of two packages - a referee package (`a1_456_referee`), and a ROS package that will contain your answer code (`a1_456_answer`).

Inside the referee package the only file that you will use directly is:

- **A 'launch file'** `a1.launch` that will be used to launch, with appropriate parameters:
  - The simulator.
  - The RViz visualiser.
  - The referee program.
  - The explorer program.
  - A mapping subsystem.

  You can launch it by running:

```
roslaunch a1_456_referee a1.launch
```

Inside the package that will contain your code (a1_456_answer) the following files exist:

- **An explorer program** explorer_node_a1_456_cpp.cpp that you will edit in order to complete the assignment (or, alternatively, explorer_node_a1_456_python.py).
- **A cmake configuration file** CMakeLists.txt for configurating compilation of the C++ node (cmake is a cross-platform make configuration utility).
- **A ROS package manifest** package.xml for giving the ROS build system more information about the package.

You don't need to make use of the mapping subsystem but using it can help a lot with knowing where in the map has not been visited.

Compile the skeleton code

Important reminder 2: Before doing anything with ROS, ensure your environment variables are set up appropriate to your workspace by running (if your workspace is in catkin_ws):

```
source ~/catkin_ws/devel/setup.bash
```

(remember this can be automated by putting the command in .bashrc)

Download and unzip the skeleton code in the file a1_456_v1.3.zip (that can be found on the assignment page on Ninova) so that it exists in the src directory of your workspace:

```
unzip a1_456_v1.3.zip -d ~/catkin_ws/src
```

E.g. if your workspace were in catkin_ws, you would expect to see the following new directories:

```
~/catkin_ws/src/a1_456_answer
~/catkin_ws/src/a1_456_referee
```

Now run the following commands to compile your code (in the C++ case):

```
cd ~/catkin_ws
catkin_make
```

The ROS build system, “catkin” will automatically determine how to compile your code with the right dependencies. At the end you should see some messages, including:

```
Built target referee_node_a1_456_cpp
Built target gazebo_map_publisher
Built target explorer_node_a1_456
```

The executables will be in ~/catkin_ws/devel but you don't run them directly.

Run the skeleton code

Again, ensuring that your ROS environment variables are set up (see above), run the following command to run the simulator, mapping subsystem, and referee:

```
roslaunch a1_456_referee a1.launch
```

You can use the Ctrl-C key-combination in the terminal in order to close down ROS and all the ROS nodes you started with this command.

To run the controller, open up a new terminal window and run the following command:

```
rosrun a1_456_answer explorer_node_a1_456_cpp
```

or

```
rosrun a1_456_answer explorer_node_a1_456_python.py
```

You are now in a position to edit the file explorer_node_a1_456_cpp.cpp or explorer_node_a1_456_python.py in order to improve your controller. Simply edit the code in the file, rerun catkin_make (in the C++ case), and rerun the above command.

> **Note**: The first time you run this you will need an internet connection so that the simulator (Gazebo) can download the maps and objects that it needs. This may take a little while. To ensure it is working you may check how much data is being transferred over your internet connection.

More help

Examine the ROS environment

To get a deeper understanding of what is happening when the simulation and rviz is running, read:

http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

Try running (in turn) the following commands *while the simulation and/or rviz is running* to get an idea for what is going on behind the scenes:

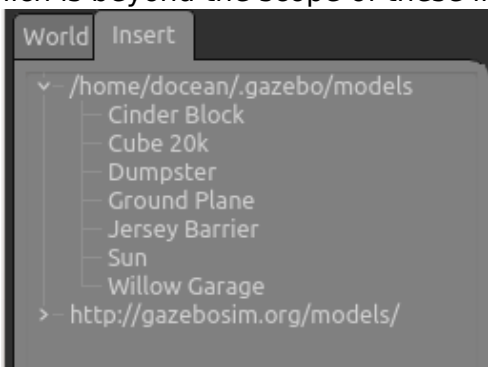
```
rqt_graph
rostopic list
rostopic info <TOPIC NAME>
rostopic echo <TOPIC NAME>
rosnode list
rosnode info <NODE NAME>
```

Where <TOPIC NAME> and <NODE NAME> can be replaced with a topic or node name that you obtained when you ran rostopic list or rosnode list.

Editing the simulated world

In order to produce more exciting scenarios for your robot, you can use the simple object insertion and manipulation toolbar or the “Insert” panel to insert new objects in Gazebo:

In order to save and re-use these new scenarios, however, you will have to create your own “launch” file, which is beyond the scope of these instructions.



Buggy Gazebo

If Gazebo crashes, it may not have fully crashed. It may just be the GUI that crashed. You usually can get the GUI back by running, in yet another terminal window:

```
rosrun gazebo_ros gzclient
```

ROS messages

The ROS 'node' that you are writing will 'subscribe' to topics to which the turtlebot is 'publishing' 'messages' (and so receive those messages).

- Your node skeleton has been written to subscribe to laser scans from the `/scan` topic.

  These messages will be of type sensor_msgs/LaserScan.

  Within the C++ code, the type for the laser scans in the callback will be:

  ```
  const sensor_msgs::LaserScan::ConstPtr&
  ```

  Within the Python code, the type for the laser scans in the callback will be:

  ```
  sensor_msgs.msgs.LaserScan
  ```

The ROS 'node' that you are writing will 'publish' 'messages' to topics to which the turtlebot is 'subscribing'.

- Your node skeleton has been written to publish to twists on the `/cmd_vel_mux/input/navi` topic.

  These messages will be of type geometry_msgs/Twist.

  Within the C++ code, the type for the twists to be published will be:

  ```
  geometry_msgs::Twist
  ```

  Within the Python code, the type for the twists to be published will be:

  ```
  geometry_msgs.msg.Twist
  ```

> **Important**: The robot will ignore your commands if the teleoperation program is open. You need to close it for your robot to start moving.

More information about these types could be found by examining the header or Python files but the easiest thing to do is to look at the documentation for the messages given above.

Also, an IDE with code completion (like *kdevelop* for C++ or *spyder* for Python) can help a lot.

Also see the ROS tutorials suggested in the 'Basic knowledge' section above.

How to use laser scan data

Laser scan data will be available within the callback function that your program registers when it subscribes to the /scan topic by creating a ros::Subscriber (C++) or rospy.Subscriber (Python) object.

C++: As the datatype provided to the callback is usually in the form of a (smart) pointer, you will need **->** to access it.

The data structures available withn the message object are described in the message documentation:

http://docs.ros.org/indigo/api/sensor_msgs/html/msg/LaserScan.html

In particular, the float32[] ranges datatype is available as a normal std::vector inside your C++ program (tuple in your Python program) and consists of a series of numbers representing how far away each part of the scan found an object. You can find the size and contents through normal debugging methods or examining the comments in the code.

How to send movement messages

You can publish a message through a ros::Publisher (C++) or rospy.Publisher (Python) object. The message that you will be publishing is a “twist”, which is a combination of linear and angular velocity, but for three dimensional objects. As we are only concerned with two dimensions, you will only need to set the **x** and **y** compoments of the linear velocity and the **z** component of the angular velocity (anyway, the robot is only a wheeled mobile robot so could not follow many possible 3D motions that would require flight for example).

As with the laser scan, you can access the fields of the geometry_msgs/Twist message, which will be “linear” and “angular”, which themselves are messages of type geometry_msgs/Vector3. To see what fields are available in a Vector3 message, see the message definition:

http://docs.ros.org/api/geometry_msgs/html/msg/Vector3.html

If you are using Python you can use commands like type() or dir(), the inspection capabilities of your spyder ipython console, etc. to determine what attributes existing objects have.

Advanced Topics

Perhaps you are curious, perhaps you want every piece of information you can get your hands on to win the competition and humiliate your classmates, or perhaps you like reading. This section is for you.

Understand the map

The assignment itself does not require you to access the map. If you want to use it, the information in this section can help.

The map is provided in an OccupancyGrid type message.

Information about how to use the occupancy map message can be found here:

http://docs.ros.org/indigo/api/nav_msgs/html/msg/OccupancyGrid.html

In order to access map information you can use the following expressions (C++ - Python is similar):

Height in pixels: map_msg.info.height
Width in pixels: map_msg.info.width
Value (at X,Y): map_msg.data [X*map_msg.info.width + Y]
Meaning of value:
0 = *fully free space.*
100 = *fully occupied.*
-1 = *unknown.*
Coordinates of cell 0,0 in /map frame:
map_msg.info.origin
The size of each grid cell:
map_msg.info.resolution

Additional information about how slam_gmapping, which is being used by the robot to make the map, works can be found here:

http://wiki.ros.org/slam_gmapping

The map message contains the origin coordinates of the map msg.info.origin which corresponds to map entry map[0][0] in the global frame of reference. Each map entry (cell) is map.info.resolution size in both width and height.

Obviously the robot does not have access to the complete map (the robot does not know the contents of the simulation/world only what its own sensors show) so this estimate can sometimes be wrong.

For more sophisticated way of thinking about the relationship between the map frame of reference (/map) and the robot base (/base_link) it is suggested you study the following topics:

- Introduction to tf.
- Writing a tf listener (C++).

If, in the skeleton code, you set the value of the (const bool) variable chatty_map to true, it will print an ASCII representation to terminal of the current map. This will quickly get untenable for bigger maps, but the rviz session that is loaded with a1.launch will also show the current map as known/estimated by the robot.