Python Applications for Robotics

Spring 2021

Instructor:	Zeid Kootbally	**Final Project:**	Visual Servoing.
Email:	zeidk@umd.edu	**Due date:**	05/05/2021 by 7 pm

Contents

1 Updates

Updates added to this pdf document:

- 04/28/2021
  - Section 10 was added to help with issues for attaching the marker to the robot.
  - Section 5.1 was updated with a mention to a broadcaster.
- 04/25/2021
  - Section 5.2: The behavior of **follower** has been updated. Instead of waiting a certain amount of time we are now tasking **follower** to rotate in place.
- 04/24/2021
  - Section 2: Added instructions to clone the package gazebo_ros_link_attacher in your workspace.
  - Section 5.2.1: Modified the list of expected outputs when your nodes are running.

2 Prerequisites

To complete this project you need the following:

- Debian ROS packages:

  ```
  sudo apt install ros-<DISTRO>-fiducial-msgs
  ```

  ```
  sudo apt install ros-<DISTRO>-aruco*
  ```

  ```
  sudo apt install ros-<DISTRO>-image-transport-plugins
  ```

  ```
  sudo apt install ros-<DISTRO>-gazebo-ros
  ```

  - Update the package list: `rospack profile`
  - You may also need to clone the following package in your workspace:

    ```
    git clone  https://github.com/pal-robotics/gazebo_ros_link_attacher
    ```

  - Disclaimer: I probably forgot some packages. Shoot me an email if you can not run the launch files that come with the package I am providing to you.
- The package final_project_x. This is the only package you need in your catkin workspace for the final project.
- You need the yaml file you generated in RWA3.
- You need the two files generating from mapping.

3 Introduction

It is highly recommended you perform this project as a group. You can stay in the same group from RWA3 or you can create a new group.

The final project consists of two waffle turtlebots, named **leader** and **follower**. The goal of this assignment is to make **follower** follows **leader** using visual servoing. Visual servoing, also known as vision-based robot control,

is a technique which uses feedback information extracted from a vision sensor (visual feedback) to control the motion of a robot. Visual servoing is possible with the waffle model since it comes with a depth camera which also publishes RGB images. We can observe the live stream from the camera in the bottom left corner of rviz. We will use the **follower** camera to perceive the **leader** robot in the environment, and then try to follow it. To make it easier, a marker has been attached to **leader**. **follower** has to use its camera to find this marker and follow the marker by keeping a safe distance.

You will use move_base to drive both robots.

- For **leader**, read the yaml file created in RWA3, which contains location information, and tasks the robot to reach each location in successive order.
- Task **follower** to follow **leader** while keeping a safety distance of 0.5 m.

4 Final Project Package

All you need to start the simulation comes with the final_project_x package.

4.1 Rename Package

First, you have to rename this package. Replace x with your group name or with lastname_firstname (if you are doing the project alone). After renaming this package, rename any instance of final_project_x in the following files:

- *final_project_x*→`CMakeLists.txt`
- *final_project_x*→`package.xml`
- *final_project_x*→*launch*→`follower_move_base.launch`
- *final_project_x*→*launch*→`leader_move_base.launch`
- *final_project_x*→*launch*→`multiple_robots.launch`
- *final_project_x*→*launch*→`navigation.launch`
- *final_project_x*→*launch*→`single_robot.launch`

4.2 Add Previously Generated Files

- Inside the *maps* folder place the two files you generated in RWA3 during mapping (yaml and pgm).
- Inside the *yaml* folder place the yaml file (which contains location information) you created in RWA3.

4.3 Package Content

Below is a description of the structure for the package final_project_x.

- *launch* folder: All the launch files needed to start the simulation environment with the two robots, start amcl, and move_base are located in this folder. More on some of these launch files below.
- *maps*: This folder is empty for now. Place your map files in this folder.
- *models* and *photos*: These folders consist of all the models and photos needed for the Gazebo world. These folders were copied from the original aws-robomaker-small-house-world package. Do not modify anything in these folders.

- *nodes*: This folder has two Python files. One file consists of a node to attach a marker to **leader**. The other file provides examples on how to use **move_base** with the robots. For the project you will need to create Python files in this folder.
- *param*: Information needed for the planners and costmaps can be found in this folder. These files were copied from the official turtlebot package and were modified to include two robots instead of one. Do not modify anything in this folder.
- *rviz*: Pre-configured RViz files are located in this folder. When RViz starts it will load those files so the options, topics, etc are already configured. Do not modify anything in this folder.
- *worlds*: This folder consists of the world file to be loaded in Gazebo. It is the same world we used in RWA3. Do not modify anything in this folder.
- *yaml*: This folder is empty. As mentioned previously, place the yaml file you created through ROS service calls in RWA3.
- *doc*: Folder to store `readme.txt` to provide instructions on how to run the project.

4.3.1 Spawning Robots

Starting the simulation and spawning the robots are performed with:

```
roslaunch final_project_x multiple_robots.launch
```

Spawning multiple robots is quite tricky. We start each robot individually by placing them in a different namespace. In the snippet below **leader** and **follower** are placed in their own namespace. Placing each robot in its own namespace ensures that topics, parameters, tf frames, etc are specific to each robot. For instance, if you want to send velocities to control each robot you will need to use either `/leader/cmd_vel` or `/follower/cmd_vel`. If both robots subscribe to `/cmd_vel` then sending velocity data to this topic will move both robots at the same time (which is not something we want to do in this project).

We also need to have a specific value for each robot for the parameter `tf_prefix`. If we do not re-define `tf_prefix` then the default one (`tf`) is used. Both robots will have the same frames (e.g., `/base_footprint`) and you will not be able to do any transform (are we talking about `/base_footprint` for **leader** or **follower**?). By reassigning `tf_prefix`, the tf tree will now show `/leader_tf/base_footprint` and `/follower_tf/base_footprint`.

```
<!-- leader -->
<group ns="leader">
    <param name="tf_prefix" value="leader_tf" />
    <include file="$(find final_project_x)/launch/single_robot.launch" >
        <arg name="init_pose" value="-x 2.999509 -y 0.869111 -z 0" />
        <arg name="robot_name"  value="leader" />
        <arg name="has_marker" value="true" />
    </include>
</group>

<!-- follower -->
<group ns="follower">
    <param name="tf_prefix" value="follower_tf" />
    <include file="$(find final_project_x)/launch/single_robot.launch" >
        <arg name="init_pose" value="-x 2.212427 -y 0.869111 -z 0" />
        <arg name="robot_name"  value="follower" />
```

```
            <arg name="has_marker" value="false" />
        </include>
    </group>
```

`single_robot.launch` consists of definitions and operations for only one robot. In the code snippet above we are calling this file twice, once for each robot.
Figure 1 shows the tf tree for the simulated environment.

4.3.2 Attaching ArUco Marker

Visual servoing is done using an Aruco marker which we place on **leader** (see Figure 2). This marker can be observed with a 2D camera, and have its 3D pose within the world extracted from the image. You can go to chev.me/arucogen and get an image by specifying a marker id and size. The model for the marker can be found in the *models* folder. In `multiple_robots.launch` we are using the argument `has_marker` to specify should have this marker attached to it (**leader** in this case). `single_robot.launch` will start the node **spawn_marker_on_robot** if a marker is assigned to a robot. This node is located in *nodes*/`spawn_marker_on_robot`. In a nutshell, to attach the marker to the robot, the node does the following:

- Pause the simulation (to make sure the robot is not moving while the marker is being attached to the robot).
- Get the Gazebo model name (**leader** in this case).
- Attach the marker using the Gazebo plugin **ros_link_attacher_plugin** (see line 5 in `small_house.world`).
- Unpause the simulation.

4.3.3 Starting Navigation

So far we just spawned robots in Gazebo. To do autonomous navigation we need to start amcl and move_base. We have created launch files to start these nodes for each robot. In a second terminal, enter:

```
roslaunch final_project_x navigation.launch
```

4.3.4 Detecting ArUco Marker

To detect the ArUco marker we use the the node **aruco_detect** from the aruco_detect package, as seen in `multiple_robots.launch`. **aruco_detect** subscribes to a camera topic, and looks for markers in the images it receives. If a marker is detected, the node publishes its 3D transformation.

```
<!-- start marker detector -->
 <node name="aruco_detect" pkg="aruco_detect" type="aruco_detect">
   <param name="image_transport" value="compressed" />
   <param name="publish_images" value="true" />
   <param name="fiducial_len" value="0.15" />
   <param name="dictionary" value="0" />
   <param name="do_pose_estimation" value="true" />
   <param name="ignore_fiducials" value="" />
   <param name="fiducial_len_override" value="" />
   <remap from="/camera/compressed" to="/follower/camera/rgb/image_raw/compressed" />
   <remap from="/camera_info" to="/follower/camera/rgb/camera_info" />
 </node>
```

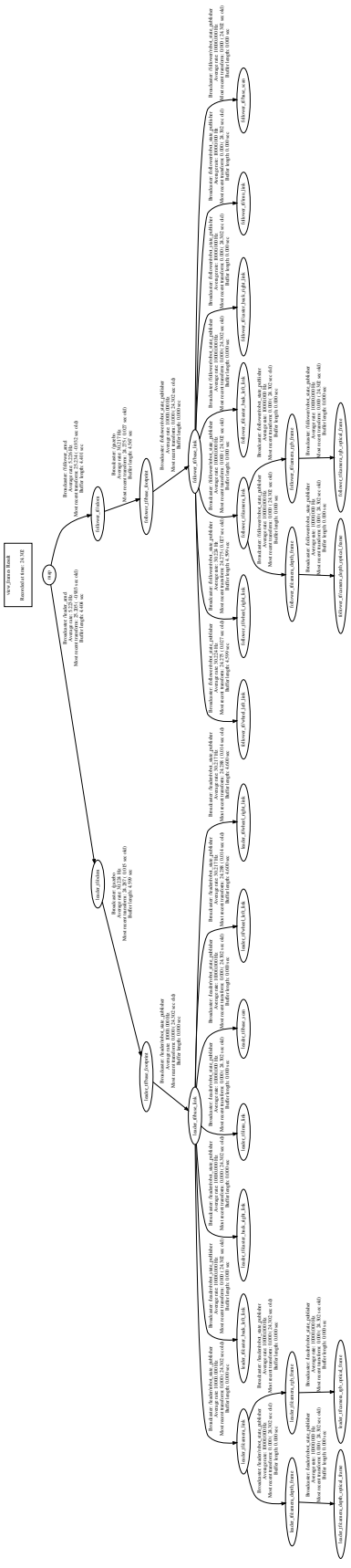

Figure 1: tf tree for current application.

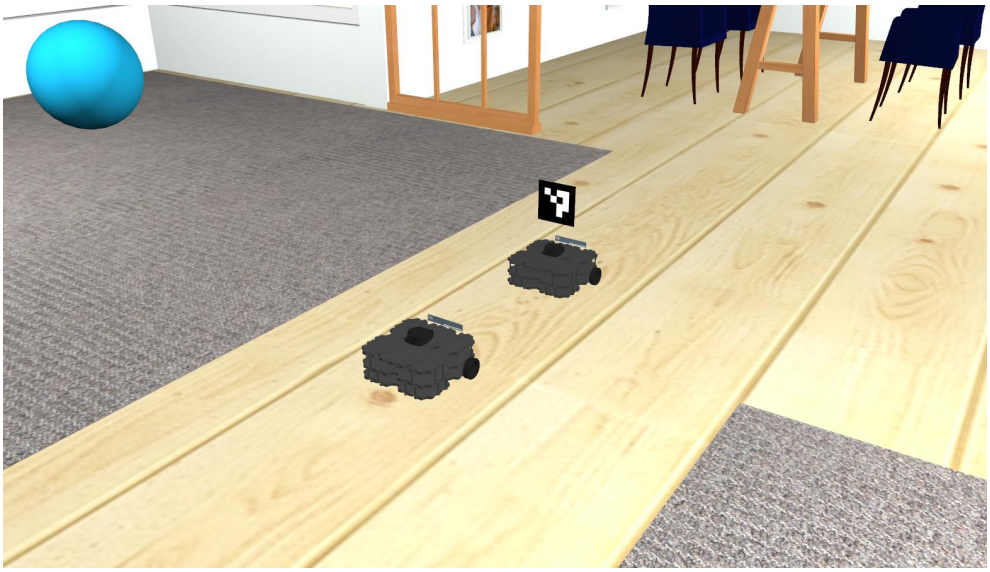

Figure 2: Marker attached to **leader**.

In the snippet above we are tasking **aruco_detect** to listen to **/follower/camera/rgb/image_raw/**. This node will publish on the topic **/fiducial_transforms**.

After starting the simulation we can listen to this topic:

```
roslaunch final_project_x multiple_robots.launch
roslaunch final_project_x navigation.launch
rostopic echo fiducial_transforms
```

A message published on this topic is shown below:

```
header:
  seq: 1308
  stamp:
    secs: 51
    nsecs: 948000000
  frame_id: "follower_tf/camera_rgb_optical_frame"
image_seq: 1319
transforms:
  -
    fiducial_id: 0
    transform:
      translation:
        x: -0.0475116704523
        y: -0.186742861661
        z: 0.666488540435
      rotation:
        x: 0.999999855045
        y: 7.48915858126e-05
        z: -0.000212491994457
```

```
    w: 0.000489027967743
image_error: 0.000756484689191
object_error: 9.11063144018e-06
fiducial_area: 73756.442833
```

As we can see, the ArUco marker was detect in the camera frame of the follower. The marker id is also provided (this matches the id from chev.me/arucogen). In order to follow **leader**, we need to send the goal pose of the marker in the map frame.

5 Tasks for Final Project

For the project you need to write two main nodes:

- The first node reads your yaml file with locations and tasks **leader** to go to each waypoint (using **move_base**). Note: The robots are not aware of each other. If you send **leader** to a goal pose then it may bump into **follower**. You can use laser beams (RWA2) to detect the other robot. You can also hardcode some smaller waypoints between **leader** current pose and the goal pose to avoid **follower**.
- The second node does the appropriate transforms from the camera frame to the map frame and generates a goal in the map frame for **follower**. Using **move_base**, task **follower** to reach the goal. Make sure **follower** stays behind **leader** within a safe distance of 0.5 m.
- OOP is not a requirement but you can explore the idea of using OOP for the project.

5.1 Transforms

The transforms needed to express the marker in the map frame is shown in Figure 3. You need to compute $[T]^{map}_{marker}$ which is a transform composition as shown in Equation 1.

$$[T]^{map}_{marker} = [T]^{map}_{robot}[T]^{robot}_{camera}[T]^{camera}_{marker} \tag{1}$$

Note that $[T]^{robot}_{camera}$ is a static transform, meaning the transform between the camera and the robot will always be the same. Since the camera is rigidly attached to the robot it will always be in the same pose on the robot.

$[T]^{camera}_{marker}$ can be retrieved from the topic **fiducial_transforms**. However `listener.lookupTransform` can only find the transform between 2 existing frames (existing in the tf tree). If you want to get the transform between the marker and **follower** you need to use a broadcaster. In other words, read the transform you receive from **fiducial_transforms**, broadcast it in the frame **follower_tf/camera_rgb_optical_frame**, finally do `listener.lookupTransform` between the new broadcast frame and map. The result will give you the pose of the marker in the map frame.

5.2 Behavior for leader and follower

leader has to reach each location stored in your yaml file.

- For each location in the yaml file to visit:
  - Use **move_base** to task **leader** to go to the goal.

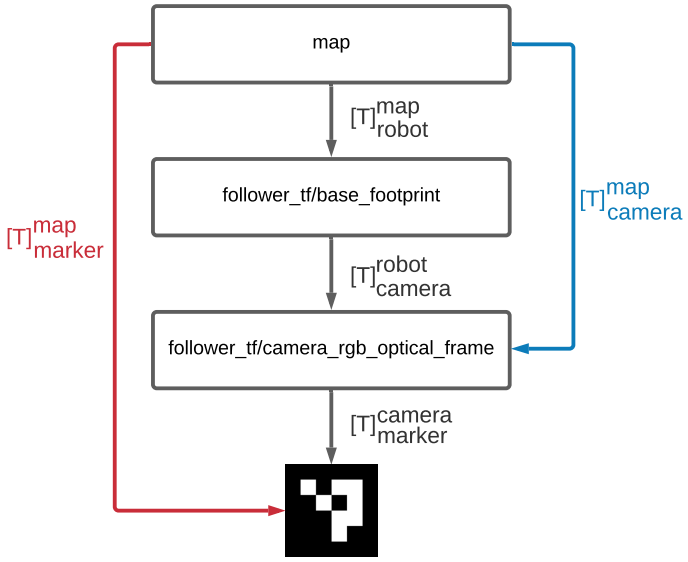

Figure 3: Transforms from marker to map.

  - Create/update a parameter on the parameter server (more information here), which contains information on the location **leader** is trying to reach. This parameter can be a simple string (e.g., "livingroom") or a list of numbers which represent the pose of the location **leader** is heading to. More information on parameter types can be found here.

The goal of the control algorithm for **follower** is to follow a marker by maintaining a safe distance. For every fiducial transformations message, the control loop for **follower** will be as follows:

- If marker is not detected:
  - Make **follower** rotate in place 360°. If marker is still not found after a full rotation, read the parameter from the parameter server and use **move_base** to start moving **follower** towards the location (we will call this behavior "exploration"). If **follower** loses the marker we cannot let this robot stay idle as it may never detect the marker. By moving **follower** to the location where the target is expected to be found, **follower** has a bigger chance of finding the marker again. If **follower** has reached the location and the marker is still not found then read the parameter again as there is a good chance **leader** is heading towards the next location and this information has been updated on the parameter server.
- If marker is detected:
  - If "exploration" is in progress then stop "exploration".
  - Compute the goal pose.
  - Task **follower** to reach the goal.

5.2.1 What to Output in the Terminal

Do not clutter the terminal with useless outputs. Below is a list of outputs that we expect to see in the terminal while your nodes are running. Remember to use one of the `rospy.log*` methods to output in your terminal.

- Outputs for **leader**:
  - "Going to <spot>" (replace spot with one of the spots stored in your yaml file).

  - "<spot> reached" when **leader** has reached the spot.
- Outputs for **follower**:
  - "Following marker" when **follower** follows the marker.
  - "Looking for marker" when **follower** can not find the marker (**follower** is in exploration mode).
  - "Marker found" when **follower** finds the marker while in exploration mode.

6 Submission

- Submit only final_project_x (after renaming it).
- Make sure your code is documented or you will lose points. Since this is the final project, non-documented code will be severely penalized. Same thing goes for not following PEP8.
- An extension is possible (see Section 8) but unlikely.
- Create a folder *doc* in the package and create a `readme.txt` file. Write the instructions on how to run your project. You will be penalized if instructions are not provided.

7 Grading Rubric

- Maximum points: 120 pts. Note: Due to the large number of students in this class we will not have time for a presentation from each student or from each group. The 40 pts that were originally allocated to presentation have been split between the live demo and the final report.

+40 pts – **leader** reaches each waypoint stored in the yaml file.

+60 pts – **follower** manages to follow the fiducial marker using the approach described in Section 5.2.

+20 pts – **Documentation**: Code properly documented and PEP8 guideline followed. There is no need to generate any documentation.

-20 pts – **yaml file**: Points will be deducted if you hardcode locations in your Python code instead of retrieving them from the yaml file. However, you can read this yaml file once and store its content in a data structure in your Python code. In any case you have to read this file.

-50 pts – **follower** does not follow the fiducial marker but only goes to the goal described in the yaml file. While running your code we will look at the topic **fiducial_transforms**. If we see transforms being published but **follower** ignores the marker then we will deduct points.

8 Extension

This assignment is fairly doable when working as a team. If a large number of students need more time then we will "try" to move last class to a later day. This extension needs to take into account the time needed to grade this project for each student/group. We also need to take into account the deadline for entering your grades in the system.

9 Last Class

For last class I will need some volunteers to give us a live demo. We will try to keep it under 2h. If there are no volunteers then I will randomly pick the names myself. Participation is important in this class. Even though the live demo is not graded it will help with the final grade if you are not too far from the next letter grade.

10 Troubleshooting

Some students have issues with attaching the ArUco marker to the turtlebot. Here are some suggestions on how to fix this issue:

- Move the directory *aruco_marker* to *~/.gazebo/models* in your Home directory.
- Edit the following in `spawn_marker_on_robot`:

```
while not rospy.is_shutdown() and rospy.Time.now() == rospy.Time(0):
    time.sleep(5)
```

11 Credits

- Nick Lamprianidis (for fiducial marker).
- theconstructsim (idea for splitting launch files).
- ROS answers (various things).