ECE4310: Programming for Robotics, Project 1 Technical Report

Project 1: Service Robot Auto-delivery with AR-tags

1st Zixing JIANG
School of Science and Engineering
the Chinese University of Hong Kong, Shenzhen
119010130@link.cuhk.edu.cn

***Abstract*—This text is a technical report of ECE4310 Project 1. In this project, students are asked to implement a navigation program on a service robot to perform automatic delivery. The robot is visually guided by several AR-tags. The whole experimental setup is implemented on ROS with Gazebo simulation environment. The project handout and material are available at this link.**

***Index Terms*—Service Robot, Navigation, AR-tags, ROS, Gazebo**

I. Introduction

In this project, students are asked to program a service robot to perform an auto-delivery task in a simulation environment shown in Fig. 1. There are several AR-tags in the environment. These AR-tags are the robot's delivery destinations. The robot is asked to follow all the AR-tags in the room by a specific order, which is encoded by the id of these tags. As shown in Fig. 2, the id (from 0 to 7) of a tag denotes the spatial relative direction of next tags respect to current tag. The robot is born with a initial tag. The robot is asked to search all the tags in the room by order according to the encoded directional information. The id of the last tag is 8, which denotes termination.

This report is organized as follows. In section II, I will introduce the overall architecture of the algorithm and details of each modules. In section III, I will give some instructions and comments on how to reproduce the project and run my code and go through some demos, and in section IV, I will discuss the feature and further improvement of my implementation.

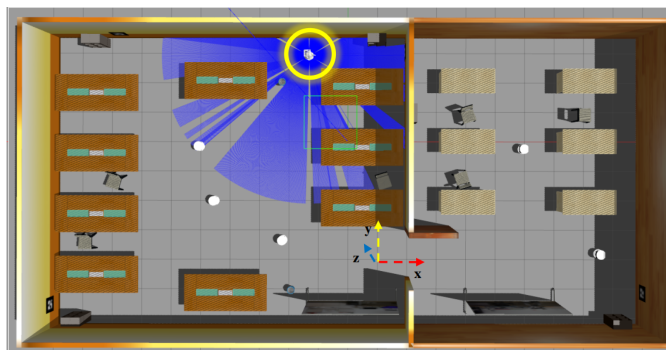
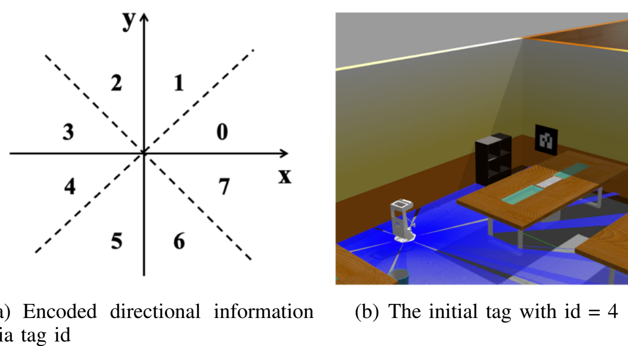


Fig. 1. Gazebo simulation environment for the auto-delivery task. The service robot is highlighted by the yellow circle. Coordinate frame of this simulation is illustrated in the figure, where the z axis points to the ground. The origin of this environment locates at the center of the green rectangle.



(a) Encoded directional information via tag id

(b) The initial tag with id = 4

Fig. 2. Encoded directional information in the AR-tag.

II. Navigation Algorithm

The whole robot navigation algorithm can be divided into three level. From high to low they are: *perception*, *planning*, and *actuation*.

A. Perception

To perform the auto-delivery task, the robot must sense the environment it locates. For example, the robot uses laser to detect the obstacles in the environment. What I want emphasis in this part is AR-tag recognition and robot coordinate frame transform. The former tells us the destination and the later help us calculate a path to follow. These two values act as input of the subsequent planning module. Without them, the system won't work.

1) AR Marker: The AR-tag in the environment is recognized by the depth camera equipped on the robot. The camera publish the recognition result on ROS with a frequency. In my implementation, a subscriber is assigned to the topic "/ar_pose_marker" to listen the to data. The listened data with "AlvarMarker" data structure consists of the id and pose respect to the virtual camera link. A callback function "update_markers_detected" is used to handle this data, as shown below. This callback function will store all the necessary

information of a tag (id, pose) in to a dictionary named "detected_markers". This dictionary will be updated every time the AR-tags information is updated, i.e. the callback is called.

```
# perception: camera and AR tag
self.camera = rospy.Subscriber("/ar_pose_marker",
    AlvarMarkers, self.update_markers_detected)
self.detected_markers = {}
```

Listing 1. AR-tag listener

```
# acquire currently detected AR tag
# --------------------------------
# tag, ID, pose in odom
def update_markers_detected(self, msg):
  if len(msg.markers) != 0:
  for marker in msg.markers:
    id  = marker.id
    pose_camera = marker.pose.pose
    pose_odom = self.camera_to_odom(marker.pose)
    self.detected_markers[id] = pose_odom
```

Listing 2. AR-tag callback

The pose recognized by camera is respect to camera link. For convenience when navigating, this pose is transformed from camera link to odom. This transformation is performed by "camera_to_odom", as shown below.

```
# transfer tag's position from virtual_camera_link
    to odom
def camera_to_odom(self, pose_camera_link):
  tf_buffer = tf2_ros.Buffer(rospy.Duration(1200.0))
  tf_listener = tf2_ros.TransformListener(tf_buffer)
  transform = tf_buffer.lookup_transform(
    "odom",
    "virtual_camera_link",
    rospy.Time(0),
    rospy.Duration(1.0)
  )
  pose_camera_link.pose.position.y = -
    pose_camera_link.pose.position.y
  pose_camera_link.pose.position.x -= 1.5
  pose_odom = tf2_geometry_msgs.do_transform_pose(
    pose_camera_link, transform)
  return pose_odom.pose
```

Listing 3. Transform between camera link and odom

Since the image frame and camera frame is opposite, before the transformation, the y-directional pose in camera link should change sign. What's more, to avoid collision when navigating, the x-directional pose in camera link is minus by 1.5 m.

2) Robot Pose: The robot's pose in odom link is subscribed from the Gazebo simulator, as shown below.

```
# perception: robot's current pose
self.pose_reader = rospy.Subscriber("/gazebo/
    model_states", ModelStates, self.
    update_robot_pose)
self.robot_pose_x = None
self.robot_pose_y = None
self.robot_pose_yaw = None
```

Listing 4. Robot pose listener

In ROS, the orientations is described by quaternions by default. Thus, for convenience when setting navigation goal, the quaternions is converted to Eular angle.

```
def update_robot_pose(self, msg):
self.robot_pose_x = msg.pose[-1].position.x
self.robot_pose_y = msg.pose[-1].position.y
_, _, self.robot_pose_yaw = euler_from_quaternion([
    msg.pose[-1].orientation.x, msg.pose[-1].
    orientation.y, msg.pose[-1].orientation.z, msg.
    pose[-1].orientation.w])
pass
```

Listing 5. Robot pose callback and Quaternions-Eular angle convertion

B. *Planning*

The responsibility of planning unit is to compute a navigation goal based on current robot state and the perception data feed. For certain tag in process, there are two possible states navigate to this tag or search from this tag. Based on this, a logic flow can be drawn as shown in Fig. 3. Based on the flow chart, pseudocode can be derived.

Algorithm 1 Planning

```
if is tag_in_process = Null then
  pick a not delivered tag
end if
if is tag_in_process delivered then
  if tag_in_process = 8 then
    Finished
    return
  end if
  for detected tag that != tag_in_process and tag does not
  delivered do
    if directional condition statisfied then
      set tag as new tag_in_process
      return
    end if
  end for
  compute goal based on directional condition
  return
else
  set goal as this tag's pose
end if
```

For more information about implementation, please refer to the source code.

C. *Actuation*

Move_base is selected as the navigator of this algorithm, as shown below.

```
# actuation: navigate through move_base
self.navigator = actionlib.SimpleActionClient("
    move_base", MoveBaseAction)
rospy.loginfo("Waiting for move_base action server
    ...")
self.navigator.wait_for_server(rospy.Duration(60))
rospy.loginfo("Connected to move base server")
```

Listing 6. Navigator

Since we convert quaternions to Eular angles in perception, the actuation function takes in x-directional position, y-directional position, and yaw angle as input. After target goal set, the Eular angle will be converted back to quaternions and feed in to move_base, as shown below.

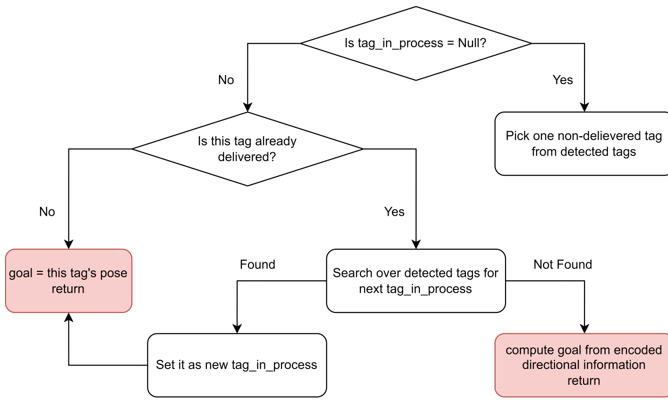


Fig. 3. Logic flow of planning

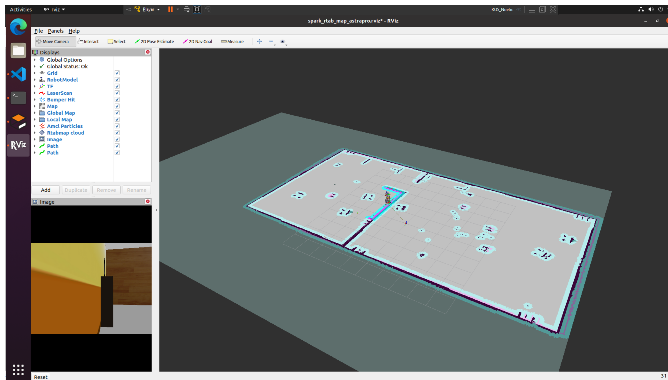


Fig. 4. Gmapping

```
# move to the planned goal
# ------------------------
def move(self, target_x, target_y, target_yaw):
  global goal
  goal = MoveBaseGoal()
  goal.target_pose.header.frame_id = "odom"
  goal.target_pose.header.stamp = rospy.Time.now()
  p = Point(target_x, target_y, 0.0)
  q_angle = quaternion_from_euler(0, 0, target_yaw,
    axes='sxyz')
  q = Quaternion(*q_angle)
  goal.target_pose.pose = Pose(p, q)
  self.navigator.send_goal(goal)
  wait = self.navigator.wait_for_result()
  if not wait:
    rospy.logerr("Action server not available!")
  rospy.signal_shutdown("Action server not available
    !")
  else:
    return self.navigator.get_result()
```

Listing 7. Move base

For convenience when publishing goal, a global map of the simulation environment is established by Grid-mapping algorithm, as shown in Fig 4.

The planning-actuation pair executes repeated in the main control loop.

III. Demonstration

To reproduce the project from the submitted source file, please follow these instructions:

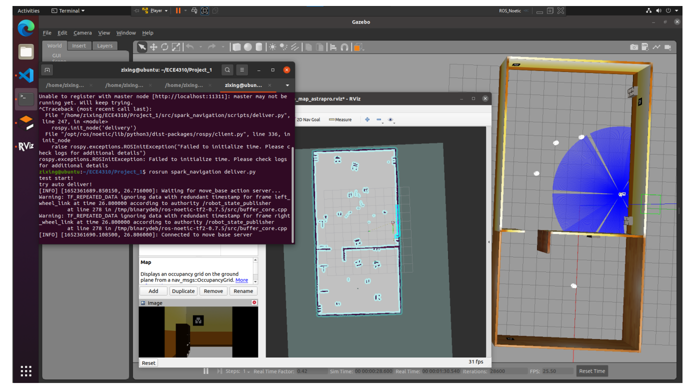


Fig. 5. Auto Delivery

1) Go to and build the workspace

```
cd P_roject1 && catkin_make
```

2) Start Gazebo simulation

```
. scripts/gazebo.bash
```

3) Start virtual camera

```
. scripts/camera.bash
```

4) Start navigation stack

```
. scripts/camera.bash
```

5) run auto-deliver

```
. scripts/deliver.bash
```

If successful, you will the interface shown in Fig. 5. You may access this link to see a video demo. This program is expected to run for about 10 minutes.

IV. Discussion

In conclusion, in this project, an auto delivery algorithm is implemented. The running results shows the algorithm works well. However, there is still something can be improved. First, the algorithm subscribe robot's pose from gazebo. However, for real application, this information may not available and we have to apply position estimation. Second, for the encoded directional information calculation, it's highly related to the layout of the environment. If the environment changes, the algorithm may fail.