Introduction to Robotics 046212, Spring 2021
Computer Exercise 1

Orr Krupnik 302629027

April 12, 2021

1 Part I

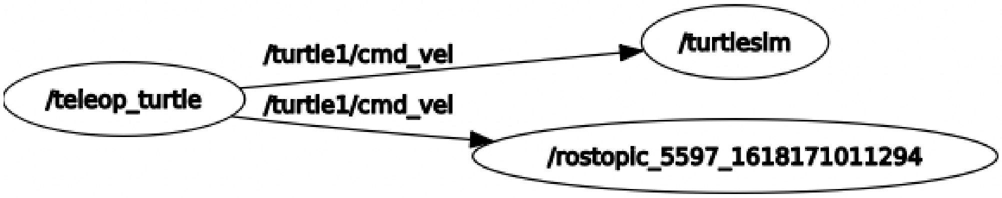
Topics and Services

Task 1. To find the name of the first node, I used `rosnode info`. To kill the turtle simulator node, I used `rosnode kill turtlesim`.

Task 2. The command for publishing a Twist message was:

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
x: 1.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0"
```

Task 3. Attached is the updated `rqt_graph` after running `rostopic echo`:



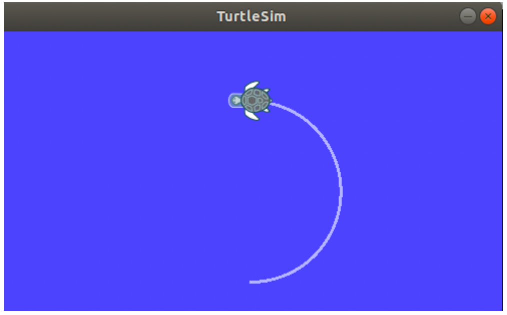
Task 4. The required commands are:

1. To clear the pen, use `rosservice call /clear`.
2. The command is `rosservice call /turtle1/set_pen "r: 0, g: 255, b: 0, width: 0, 'off': 0"`, using the arguments to set the color and width, or turn the pen off.

3. One option is to use `rosservice call reset`. Another option is to use the `teleport_absolute` service, with the correct coordinates (which seem to be around $(6.0, 6.0)$ but not exactly).

Writing our First Node

Below is a plot of a π arch drawn by the turtle:



And below is the `my_node.py` code:

```python
#!/usr/bin/env python

import numpy as np
import rospy
from std_srvs.srv import Empty, Trigger, TriggerResponse
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from std_msgs.msg import Float32

class Draw(object):
    def __init__(self):

        # Initialize the node
        rospy.init_node('drawer', anonymous=False)

        # Subscribe to pose topic
        rospy.Subscriber('/turtle1/pose', Pose, self.pose_callback)

        # Subscribe to draw_arch topic
        rospy.Subscriber('/draw_arch', Float32, self.draw_arch_callback)

        # Create publisher for cmd_vel topic
        self.cmd_vel_pub = rospy.Publisher('/turtle1/cmd_vel', Twist,
    queue_size=10)

        # Create service callback to pause_drawing
        rospy.Service('pause_drawing', Trigger, self.pause_callback)

        # Create service callback to resume_drawing
        rospy.Service('resume_drawing', Trigger, self.resume_callback)

        # Create handle to the reset service
        self.reset_service = rospy.ServiceProxy('/reset', Empty)

        # Indicator that node is now in the process of drawing
        self.is_busy = False

    def pose_callback(self, msg):
        self.current_angle = msg.theta

    def draw_arch_callback(self, msg):

        # Check availability for drawing a new arch
        if self.is_busy:
            rospy.loginfo(’Currently drawing, new request is ignored’)
            return

        rospy.loginfo(’Recieved draw request’)
        self.is_busy = True

        # Reset turtlesim
        self.reset_service()
        self.current_angle = None

        # Initialize member variables
        self.pose = None
        self.allowed_to_draw = True
        if 0 <= msg.data < np.pi:
            desired_angle = msg.data
        elif np.pi < msg.data <= 2 * np.pi:
            desired_angle = msg.data - 2 * np.pi
        else:
            raise Exception(’Angle must be in range [0, 2PI]’)

        # Wait for first published pose before drawing
        while self.current_angle == None:
            rospy.sleep(0.01)
        rospy.loginfo(’Start drawing’)

        while not rospy.is_shutdown():

            if abs(self.current_angle - desired_angle) < 0.01:
                twist_msg = Twist()
              twist_msg.linear.x = 0
              twist_msg.linear.y = 0
              twist_msg.linear.z = 0
              twist_msg.angular.x = 0
              twist_msg.angular.y = 0
              twist_msg.angular.z = 0

    self.cmd_vel_pub.publish(twist_msg)
                break

            if not self.allowed_to_draw:
                continue

            twist_msg = Twist()
            twist_msg.linear.x = 1
            twist_msg.linear.y = 0
            twist_msg.linear.z = 0
            twist_msg.angular.x = 0
            twist_msg.angular.y = 0
            twist_msg.angular.z = 0.5

            rospy.sleep(0.01)
            self.cmd_vel_pub.publish(twist_msg)

        rospy.loginfo('Finished drawing')
        self.is_busy = False

    def pause_callback(self, reqt):
        self.allowed_to_draw = False
        return TriggerResponse(success=True, message='paused drawing')

    def resume_callback(self, req):
        self.allowed_to_draw = True
        return TriggerResponse(success=True, message='resumed drawing')

if __name__ == '__main__':
    Draw()
    rospy.spin()
```

Launch Files

Below is the launch file created in this section:

```
<?xml version="1.0"?>

<launch>
  <node name='teleop' pkg='turtlesim' type='turtle_teleop_key' />
  <group ns="turtlesim1">
    <node name='sim' pkg='turtlesim' type='turtlesim_node'/>
    <node name='relay' pkg='topic_tools' type='relay' args='/turtle1/cmd_vel /turtlesim1/turtle1/cmd_vel'/>
  </group>

  <group ns="turtlesim2">
    <node name='sim' pkg='turtlesim' type='turtlesim_node'/>
    <node name='relay' pkg='topic_tools' type='relay' args='/turtle1/cmd_vel /turtlesim2/turtle1/cmd_vel'/>
  </group>
</launch>
```

ROS Parameters

Below is the launch file created in this section (with my selection of color parameters):

```
<?xml version="1.0"?>

<launch>
```

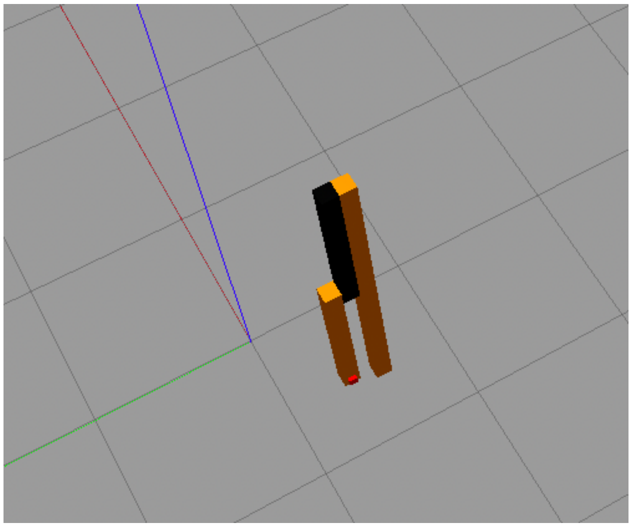
```
  <group ns="turtlesim1">
    <param name="background_r" value="255.0" />
    <node name="call_clear" pkg="rosservice" type="rosservice" args="call
   clear" />
  </group>

  <group ns="turtlesim2">
    <param name="background_b" value="0.0" />
    <node name="call_clear" pkg="rosservice" type="rosservice" args="call
   clear" />
  </group>

</launch>
```

2 Part II: Gazebo

Task 1. Screenshot of the RRBOT from an arbitrary pose:



I may have accidentally moved it slightly from the origin – Gazebo was very slow to react, so dragging around the scene may have also dragged the RRBOT.

Task 2. I queried the `/gazebo/model_states` topic using `rostopic echo /gazebo/model_states`. The resulting pose for the can is

```
position:
  x: 0.994453969112
  y: 0.0238124137028
  z: 0.0328405733862
orientation:
  x: -0.700230458086
  y: -0.103859283941
  z: -0.0583170985601
  w: 0.703910271784
```

Another option is to call the `get_model_state` service using:
`rosservice call /gazebo/get_model_state "model_name:  'coke_can3'"`.

Task 3. The can was placed near the end effector, slightly towards the positive direction of the x axis. Therefore, I set the RRBOT end effector velocity to a high linear value (10.0) using the `twist` part of the `/gazebo/set_model_state` arguments:

```
rosservice call /gazebo/set_model_state '{model_state:
{ model_name: rrbot,
pose: { position: { x: 0.0, y: 0.0 ,z: 0 },
orientation: {x: 0, y: 0, z: 0, w: 1.0 } },
twist: {
    linear: {x: 10.0 , y: 0 ,z: 0 } ,
    angular: { x: 0.0 , y: 0 , z: 0.0 } } ,
reference_frame: world } }'
```

The resulting can pose was:

```
position:
  x: 3.24013868067
  y: 0.271657234878
  z: 0.0328494642726
orientation:
  x: -0.427784834718
  y: 0.563973314078
  z: -0.565680981918
  w: 0.423012130901
```