SC 627 - Motion Planning and Coordination of Autonomous Vehicles

ROS Assignment

Chinmay Makarand Pimpalkhare
200100115

Assigment 1 | Implementation of Bug-1 Algorithm

1 Assignment 1

1.1 Bug1 Algorithm Pseudocode

```
(1) align towards the goal
until obstacle:
    move straight
if obstacle:
    rotate to align with boundary
    follow boundary as closely as possible
    complete one round along the boundary
after completion, once again move around the path to reach the closest point
after reaching closest point, go back to (1)
```

1.2 Explanation of functions used in code

1.2.1 laser_data_callback

```
def laser_data_callback(msg):
    global wait_, yaw_, closest_obstacle_angle_, closest_obstacle_dist_, dist_along_zero_
    global left_closest_, right_closest_
    global left_dist_, right_dist_
    dist_along_zero_ = msg.ranges[0]
    left_closest_ = min(msg.ranges[45:135])
    right_closest_ = min(msg.ranges[225:315])
    left_dist_ = msg.ranges[90]
    right_dist_ = msg.ranges[270]
    closest_obstacle_dist_ = min(msg.ranges)
    closest_obstacle_angle_ = msg.ranges.index(closest_obstacle_dist_)
    wait_ = True
```

Here we are taking in the laser data readings. The functions are taking in arguments from different angles across the scope of the sensor, and we are finding out the directions along which the distance is smallest and so on. This will help us when we want to rotate the bot to align it with the surrounding boundary and so on.

1.2.2 ret_distance

```
def ret_distance(P1, P2):
    dist = math.sqrt((P1.x - P2.x)**2 + (P1.y - P2.y)**2)
    return dist
```

This function is used to return the distance between two points.

1.2.3 fine_forward_movement

```
def fine_forward_movement(k):
    global current_position_, goal_position_, state_
    initial_position = current_position_
    for i in range (1,100000):
        for j in range (1, 10):
            move_the_bot.linear.x = k*(10 - j)*(j - 1)
            move_the_bot.angular.z = 0.0
            publish_to_cmd_vel.publish(move_the_bot)
            final_position = current_position_
            print('Iteration number: ', 10*(i-1) + j, 'Distance moved:', ret_distance(initial_position, final_position))
    move_the_bot.linear.x = 0
    move_the_bot.angular.z = 0
    publish_to_cmd_vel.publish(move_the_bot)
    return
```

This code is being used to move the bot by a short distance. We first increase the speed of the bot by starting from 0, let it peak at some value, and then let it slow down again. This stops the bot from veering and making it follow a straight path. Also the speed of the bot is controlled by the parameter `k`.

1.2.4 calculate_goal_angle_in_global_frame

```
def calculate_goal_angle_in_global_frame():
    global goal_position_, current_position_
    delta_x = goal_position_.x - current_position_.x
    delta_y = goal_position_.y - current_position_.y
    beta = math.atan2(delta_y, delta_x)*180//math.pi
    beta = (beta + 360) % 360
    return beta
```

This function calculates the angle needed by the bot to reach its final position.

1.2.5 is_reached_goal

```

def is_reached_goal():
    global current_position_
    global goal_position_

    if ret_distance(goal_position_, current_position_) < 0.1:
        return True
    return False
```

This function tells us whether the bot has reached the final position or not

1.2.6 move_to_perp_orientation

```
def move_to_perp_orientation(angle):
    global yaw_
    global yaw_tol_
    global closest_obstacle_angle_
    global closest_obstacle_dist_
    final_orientation = ret_final_orientation()
    yaw_error = abs(final_orientation - yaw_) if abs(final_orientation - yaw_)<=180 else 360 - abs(final_orientation - yaw_)
    # print("final_orientation: ", final_orientation)
    # while abs(yaw_error)>= yaw_error_allowed_ or abs(closest_obstacle_angle_ - 90) < abs(closest_obstacle_angle_ - 270):
    while abs(closest_obstacle_angle_ - angle)  >= yaw_tol_:
        move_the_bot.linear.x = 0
```

```
        # move_the_bot.angular.z = 0.01*yaw_error
        move_the_bot.angular.z = 0.01*(closest_obstacle_angle_ - angle)
        publish_to_cmd_vel.publish(move_the_bot)
        final_orientation = ret_final_orientation()
        print('close angle', closest_obstacle_angle_)
        print('Moving to perp')
        # yaw_error = abs(final_orientation - yaw_) if abs(final_orientation - yaw_)<=180 else 360 - abs(final_orientation - yaw_)
        # print("turning: ", yaw_error, " closest_obstacle_angle: ", closest_obstacle_angle_)
    move_the_bot.linear.x = 0.1
    move_the_bot.angular.z = 0.0
    publish_to_cmd_vel.publish(move_the_bot)
    # print("stopped, ", yaw_error)
# Turn to align with the goal
```

This function makes the bot rotate by 90 degrees and move perpendicularly to its original position.

1.2.7 action_1

```
def action_1():
    global current_position_, goal_position_, yaw_, yaw_tol_, state_
    print('Entered action 1')
    beta = calculate_goal_angle_in_global_frame()
    while (abs(beta - yaw_)) >= yaw_tol_:
        move_the_bot.linear.x = 0.0
        move_the_bot.angular.z = 0.1*min((beta - yaw_), 10)
        publish_to_cmd_vel.publish(move_the_bot)
        print('Action1: Rotating, required angle = ', beta - yaw_ )
    state_ = 2
    return
```

This code aligns the bot with the line along which is should move to reach the goal in the least possible time in the absence of any obstacles in its space.

1.2.8 action_2

```
# Move straight in the absence of any obstacles
def action_2():
    global state_, slow_down_dist, turn_after_slow_dist
    global closest_obstacle_angle_, closest_obstacle_dist_, dist_along_zero_
    print('Entered action 2')
    damper_1 = 0.4 if closest_obstacle_dist_ < 1 else 0
    damper_2 = 0.05  if closest_obstacle_dist_ < 0.5 else 0
    damper_3 = 0.1 if closest_obstacle_dist_ < 0.3 else 0
    while (dist_along_zero_ >= slow_down_dist and closest_obstacle_dist_ >= robot_radius ) :
        move_the_bot.linear.x = 0.5 - damper_1 - damper_2 - damper_3
        move_the_bot.angular.z = 0.0
        publish_to_cmd_vel.publish(move_the_bot)
        print('Action 2: Moving straight, closest obstacle straight ahead is at ', dist_along_zero_)
    move_the_bot.linear.x = 0.0
    move_the_bot.angular.z = 0.0
    publish_to_cmd_vel.publish(move_the_bot)
    state_ = 3
    return
```

This code makes the bot move in straight line if there are no obstacles in its way. I have made use of some variables that detect how far an obstacle is and accordingly slow it down. If an obstacle is near then the bot should slow down more.

1.2.9 action_4

```
def action_4():
    global current_position_, goal_position_, epsilon_, closest_obstacle_angle_,
    closest_obstacle_dist_
    global state_, count_
    print('Entered action 4')
    follow_start = current_position_
    subgoal_position = current_position_
    count_ = 0
    while count_ < 4:
        while (left_dist_ <= 1):
            if ret_distance(goal_position_, current_position_) < ret_distance(
    subgoal_position, goal_position_):
                subgoal_position = current_position_
            move_the_bot.linear.x = 0.1
            move_the_bot.angular.z = -0.01
            publish_to_cmd_vel.publish(move_the_bot)
            print(count_, 'wall following and left distance equals ', left_dist_)
        #fine_forward_movement(0.0002)
        #fine_forward_movement(0.0003)
        move_to_perp_orientation(45)
        fine_forward_movement(0.0011)
        move_to_perp_orientation(43)
        fine_forward_movement(0.001)
        count = count + 1
    move_the_bot.linear.x = 0.0
    move_the_bot.angular.z = 0.0
    publish_to_cmd_vel.publish(move_the_bot)
    state_ = 5
    count_ = 0
    return
```

This code makes the bot follow the walls as closely as possible. There are some issues with this part of the code because of which the bot is running into issues. This is the part that needs to be dealt with the most closely. It is not working properly in edge cases such as when the bot hits a block at the corner, so I am trying to correct that. I have also added another code I used for the same.

1.2.10 circumnavigation_function

```
def action_4():
    global goal_position_, state_, current_position_
    global closest_obstacle_angle_, closest_obstacle_dist_
    global circ_entry_goal_dist_, circ_entry_point_, circ_exit_goal_dist_
    global circ_exit_point_, circ_start_goal_dist_,  circ_start_point_
    while ret_distance(circ_exit_point_, current_position_) > 0.09:
        if abs(closest_obstacle_angle_ - 90) <= circum_yaw_tol_:
            move_the_bot.linear.x = 0.05
            move_the_bot.angular.z = 0.0
            publish_to_cmd_vel.publish(move_the_bot)
        else:
            while abs(closest_obstacle_angle_ - 90) >= circum_yaw_tol_:
                move_the_bot.linear.x = 0.0
                move_the_bot.angular.z = 0.1
                publish_to_cmd_vel.publish(move_the_bot)
    state_ = 5
    move_the_bot.linear.x = 0.0
    move_the_bot.angular.z = 0.0
    publish_to_cmd_vel.publish(move_the_bot)
    return
```

This is the most tricky part of the code

1.3 Issues Faced during code

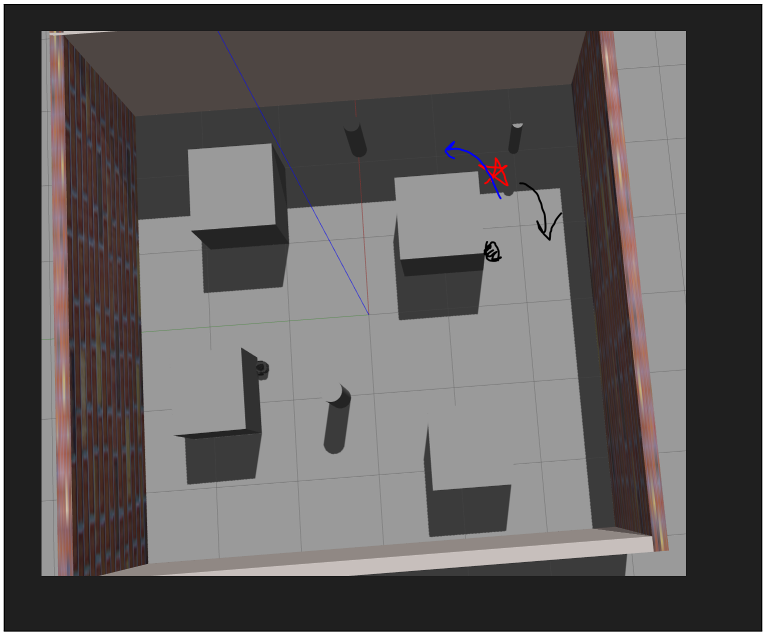

Figure 1: Sometimes the bot was getting attracted to the wall and now following the path like it should

this was an issue, where the parameters had to be tuned very much in order to continue following the wall.

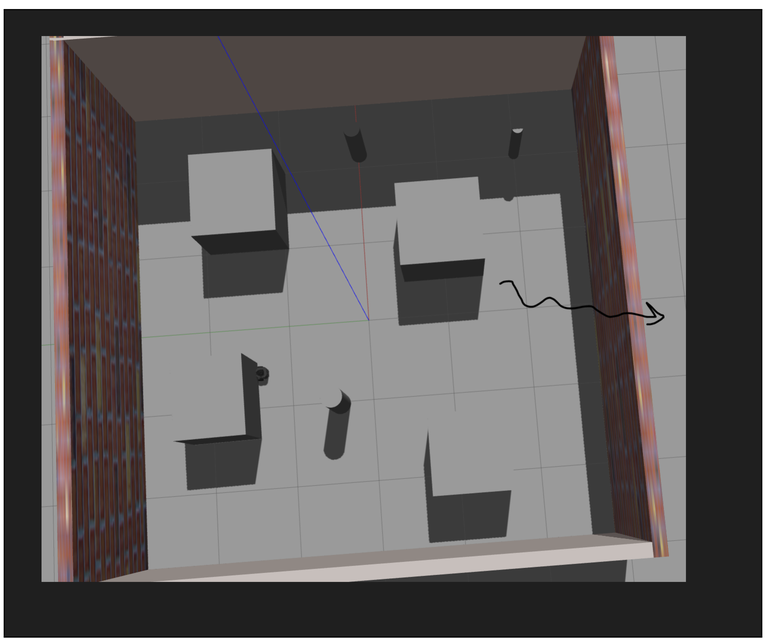

Figure 2: In some cases, the bot was exiting the grid

1.4 Things I learnt

Here I have tried to document the things that I learned while I was doing the assignment

First, we shall try and understand what the initial statements mean.

```
#!/usr/bin/env python3
#!/usr/bin/env python
```

The following two lines are **shebang lines**. A shebang line defines where the interpreter is located. In this case, the /env is used when we want the shebang to use whatever python executable is in the user's `$PATH`. This may cause the file to run differently depending upon which user decides to use it. Also note that we must define the paths for `python` and `python3` separately.

```
import rospy
```

The next line imports rospy into the code. rospy is a pure python client library for ROS. It favours implementation and development speed over runtime performance which enables codes and algorithms to be prototyped and tested quickly within ROS.

```
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
```

We shall try to understand each of the four lines, line by line. The first line imports Point(), which is an object containing the location of the particle. We can define a position variable, say `curr_pos` as a Point() and then access its components through `curr_pos.x` and so on.
The next line allows us to obtain the LaserScan data. This data can be generally read by using a command such as

```
$ rostopic echo /kobuki/lase/scan -n1
```

where the -n1 flag makes sure that the information is output only once. Generally, we can see that the topic uses the kind of message that it does using the command

```
rosmsg show sensor_msgs/LaserScan
```

The `angle_min` and `angle_max` indicate the angle range(from -90 to 90 degree in this case) that the LaserScan is measuring and the ranges is an array which gives you the distance measured for each angle bin.
Next, we have the `Twist` message. A Twist message contains two variables `angular` and `linear` of type Vector3, which contain 3 variables x,y,z of type float64.
The last line here refers to the `Odometry` message which gives us the information about the distances travelled by the robot, so that we can determine the new position from the old position. In order to print the current odometry, one can run

```
rostopic echo /odom -n1
```

The most important part is `pose`, which tells us the current position of the robot.
If one wanted to make the robot move, what they could do is publish a command velocity message to the robot, possibly in the following form:

```
rostopic pub /cmd_vel geometry_msgs/Twist
"linear:
    x: 0.2
    y: 0.0
```

```
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.2"
```

After that the code we have written also imports the transformations package, because this shall be used in the conversion of roll-pitch-yaw to euler angles.
The next line is somewhat important.

```
from std_srvs.srv import *
```

std_srvs contains two service types called Empty and Trigger, which are common service patterns for sending a signal to a ROS node. For the Empty service, no actual data is exchanged between the client and the service. The Trigger service adds the possibility to check if the triggering was successful or not.

Services in ROS In ROS, topics are used to handle communications between nodes. Topics can connect nodes to many other nodes. There are no acknowledgements between the nodes. In some cases, it is useful if some data is sent only when we request for it. ROS services implement these type of request-response type of communications. They consist of two message types: one for requesting the data, and the other for the response. These services are mainly used for event-based ROS executions. Services are defined in the same package as messages, in their own /srv folder. One ROS node is the **service server**. It advertises a service, and makes it available for other nodes. Another node is the **service client**. It sends a request message to the server once the service is needed.

ROS Publisher and Subscriber A ROS publisher is a type of node (program) which creates a particular type of ROS message and sends/publishes the message over a channel called a topic. On the other hand, a ROS subscriber subscribes to the topic so that it receives the messages whenever any message is published to the topic. A publisher can publish to one or more topic and a subscriber can subscribe from one or more topics. Further, a publisher and a subscriber are not aware of each other's existence. This idea decouples the production of information from its consumption.

```
#Function that can publish messages at a given rate
def messagePublisher():

    #define a topic to which the messages will be published
    message_publisher = rospy.Publisher('messageTopic', String, queue_size=10)

    #initialize the Publisher node.
    #Setting anonymous=True will append random integers at the end of our publisher node
    rospy.init_node('messagePubNode', anonymous=True)

    #publishes at a rate of 2 messages per second
    rate = rospy.Rate(2)

    #Keep publishing the messages until the user interrupts
    while not rospy.is_shutdown():
    message = 'bhaya meko bachao'

    #display the message on the terminal
    rospy.loginfo('Published: ' + message)
```

```

    #publish the message to the topic
    message_publisher.publish(message)

    #rate.sleep() will wait enough until the node publishes the message to the topic
    rate.sleep()
```

Similarly we can write a code to subscribe to the data which has been published

```
#Callback function to print the subscribed data on the terminal
def callback_str(subscribedData):
     rospy.loginfo('Subscribed: ' + subscribedData.data)

#Subscriber node function which will subscribe the messages from the Topic
def messageSubscriber():
    #initialize the subscriber node called 'messageSubNode'
    rospy.init_node('messageSubNode', anonymous=False)

    #This is to subscribe to the messages from the topic named 'messageTopic'
    rospy.Subscriber('messageTopic', String, callback_str)


    #rospy.spin() stops the node from exiting until the node has been shut down
    rospy.spin()
```