University of California, Riverside

EE 106

PROGRAMMING PRACTICAL ROBOTS

Spring 2024
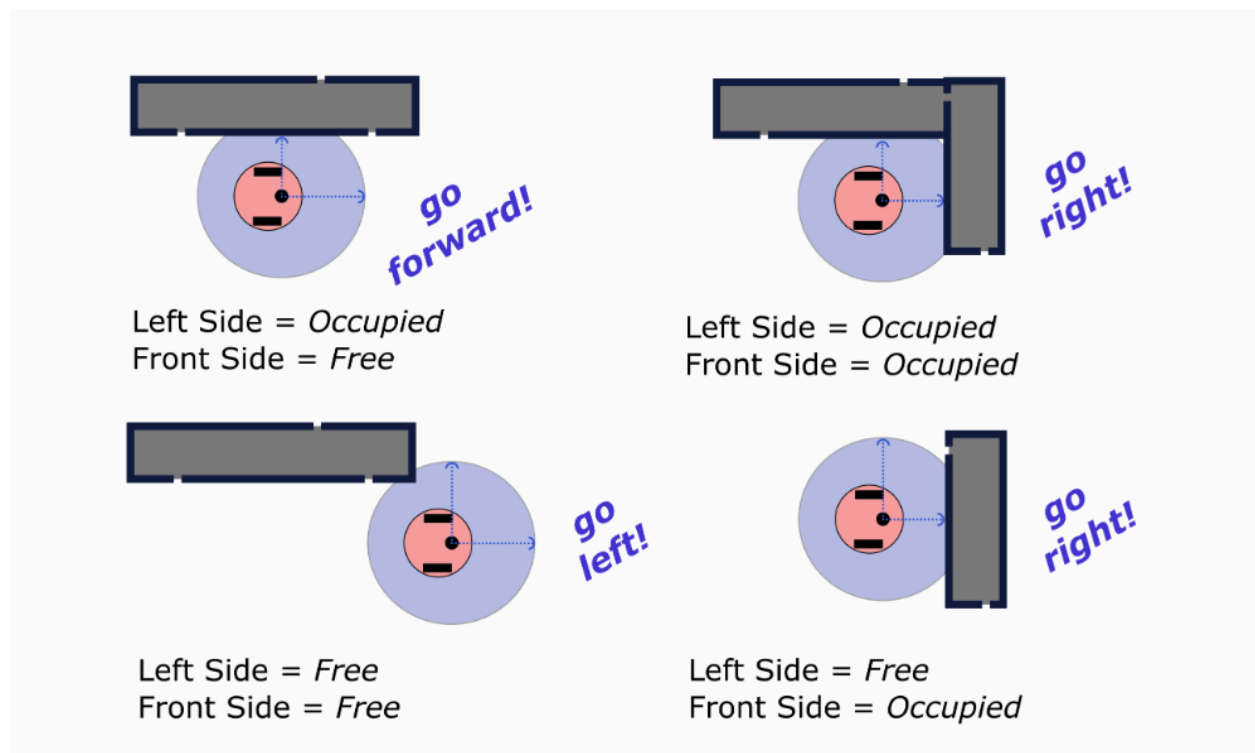

Lab 7 Report

05/18, 2024

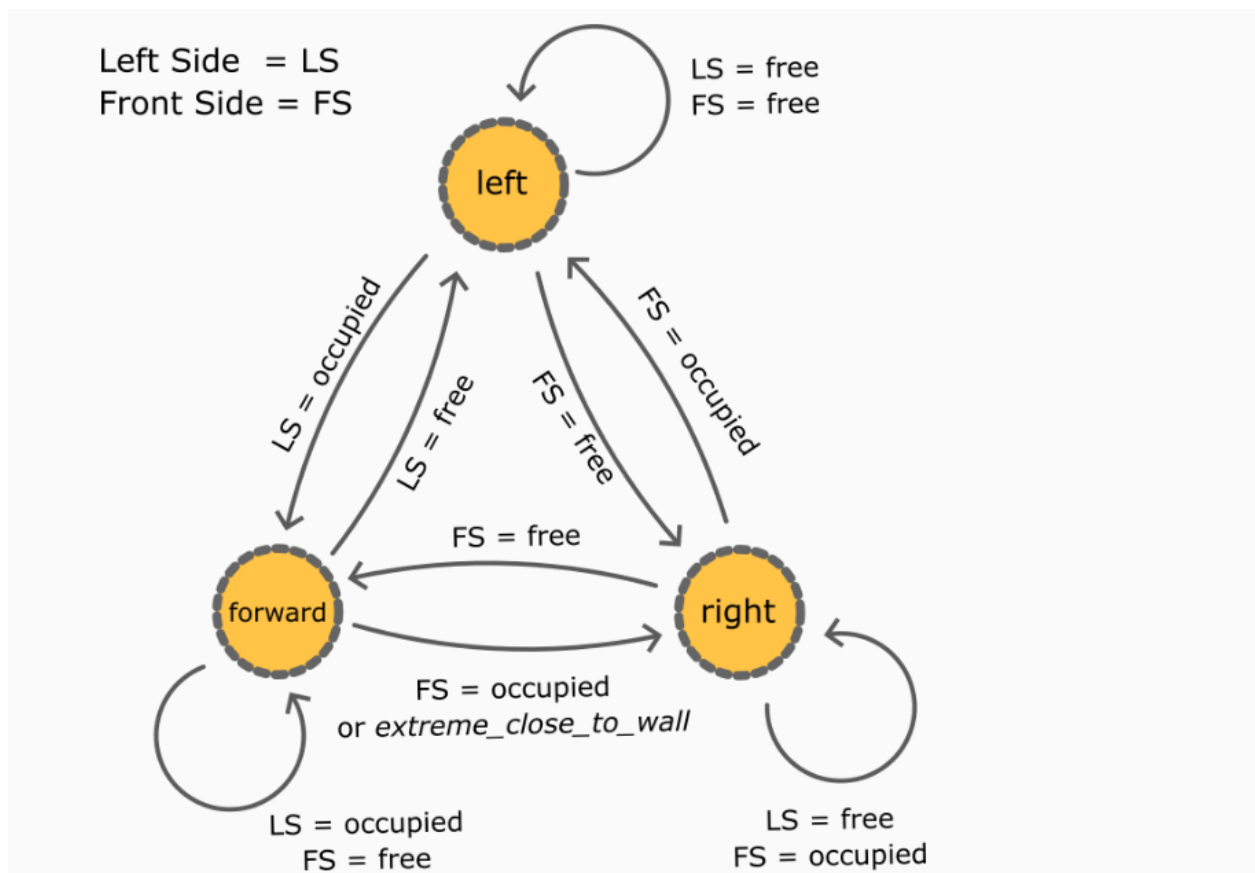| Name	SID	Section	Group Number
Thong Thach Hung Nguyen	862224662 862221239	21	04

1. Problem Statement

The objective of this lab is to understand the finite state machine of the robot so the robot can move in the left-wall following based on Finite State Machine. By detailing the challenges associated with navigation and sensing, and outlining the methods to address them, we will also introduce the concept of a Finite State Machine (FSM). This FSM will be integrated into the development of our behavior framework. From that we can build the state for the robot to go left, go right, or go straight. The goal/objective of the robot is to follow the left wall.

The finite state machine:

go forward!

Left Side = *Occupied*
Front Side = *Free*

go right!

Left Side = *Occupied*
Front Side = *Occupied*

go left!

Left Side = *Free*
Front Side = *Free*

go right!

Left Side = *Free*
Front Side = *Occupied*

Left Side	Front Side	Action
Free	Free	Left
Free	Occupied	Right
Occupied	Free	Forward
Occupied	Occupied	Right

Left Side = LS
Front Side = FS

LS = free
FS = free

FS = occupied

LS = occupied

LS = free

FS = free

FS = free

FS = occupied

left

FS = free

forward

right

FS = occupied
or *extreme_close_to_wall*

LS = occupied
FS = free

LS = free
FS = occupied

2. Design Idea

The main goal of this lab is to create the open list for all the possible nodes that are connected to the start node

Once all the nodes are planned, the next step is to calculate the cost to take from each node to the other node

If one of the node is the obstacle or the obstacle is inside the closed_list, then it will be skipped

If the current node has a better path then it will add to the set or update to the cost of the node and add it to the parent node. If the node is found the shortest path, it will update the current node to that parent cost-to-go amount

It will keep repeating again, and again until it reaches the goal

As the motion planning starts to mapping out all the possible nodes and paths, until it reaches the goal. Motion planning will calculate all the cost-to-go to each nodes and starts to eliminate all the longest cost to go from one node to other node, only leaves with small cost. After that, it will update to the current node. More explanation will be based on the "How to run code section".
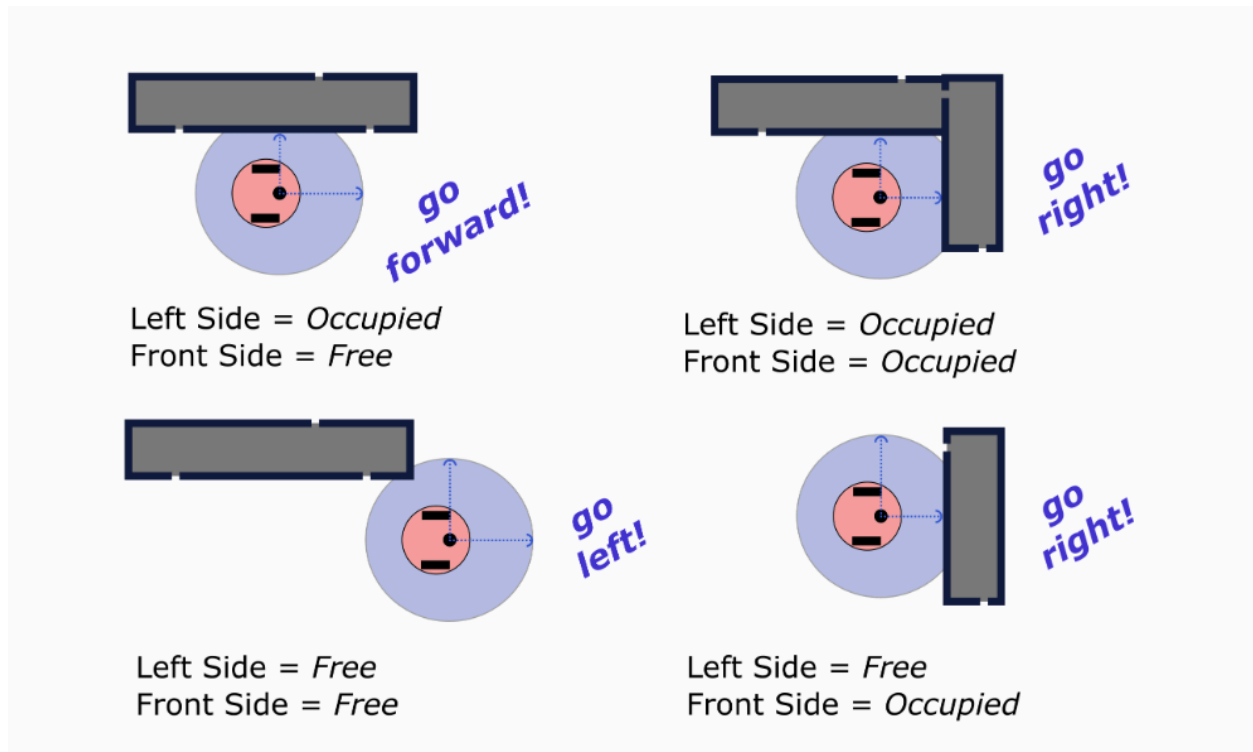
3. Results

For complex world result:
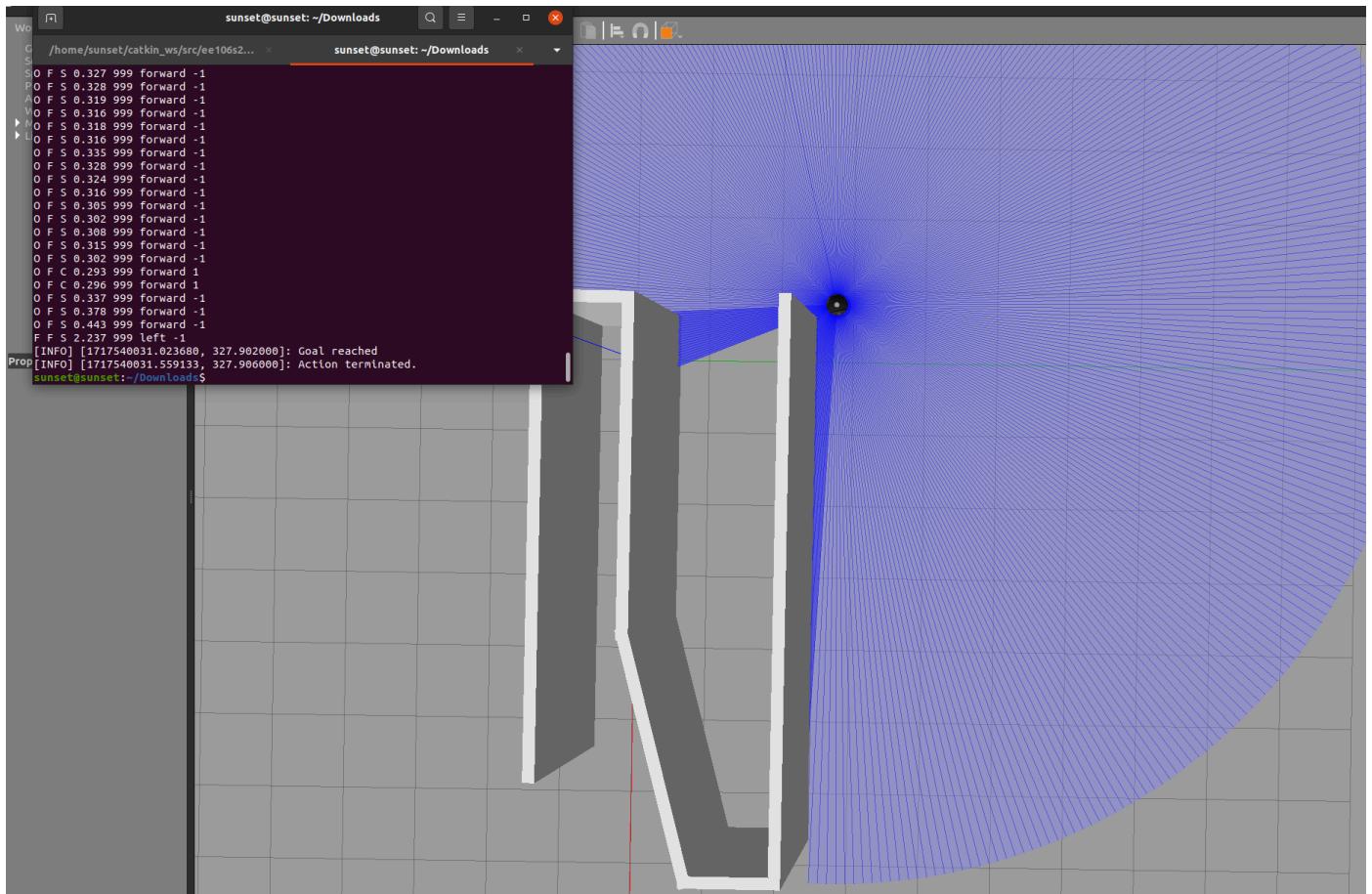
For more complex world result:
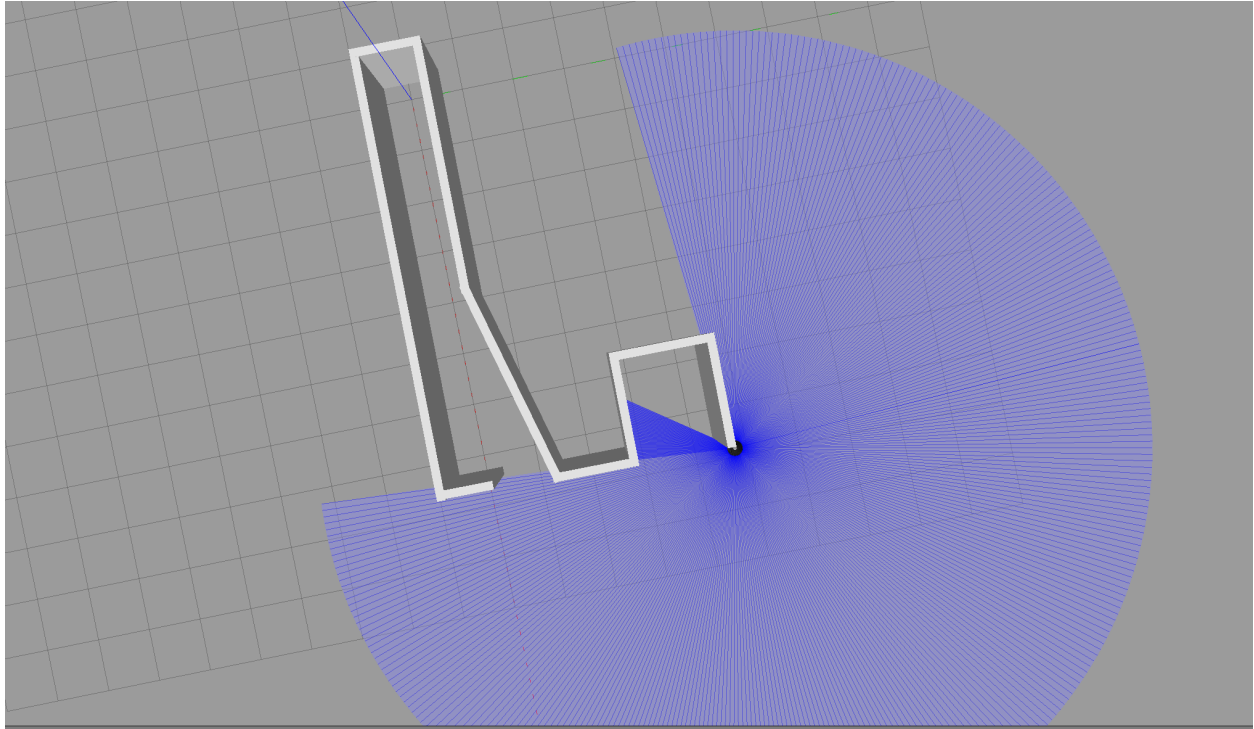
4. Appendix (optional)

1) How to run the code

Understanding the finite state machine for the left wall following:

Left Side = *Occupied*
Front Side = *Free*

Left Side = *Occupied*
Front Side = *Occupied*

Left Side = *Free*
Front Side = *Free*

Left Side = *Free*
Front Side = *Occupied*

For the complex world :

Setting up the parameter for the PD controller:

class PDController:

```python
def __init__(self, P=1.0, D=0.1, set_point=0.0):
    self.Kp = P
    self.Kd = D
    self.set_point = set_point  # reference (desired value)
    self.previous_error = 0

def update(self, current_value):
    # calculate P_term and D_term
    #  e = r - y
    error = self.set_point - current_value
    P_term = self.Kp * error
    D_term = self.Kd * (error - self.previous_error)
    self.previous_error = error
    return P_term + D_term
```

```python
def setPoint(self, set_point):
    self.set_point = set_point
    self.previous_error = 0

def setPD(self, P=0.0, D=0.0):
    self.Kp = P
    self.Kd = D
```

STEP 2

Next, setting up all the parameters for the Publishers and Suscribers, and the TransformListerner for the robot

After that, setting up the state machine for the robot facing in which way/ directions, and the goal tolerance

Continue on, setting up the PDController() function to initialize the pd control for the robot

```python
class Turtlebot():
    def __init__(self, goal_x, goal_y, csv_file):
        # Data to be taken from Launch File. Don't Edit
        self.goal_x = goal_x
        self.goal_y = goal_y
        self.csv_file = csv_file

        # Initialize subscribers and publishers
        self.lidar_sub = rospy.Subscriber('/scan', LaserScan, self.lidar_callback)
        self.vel_pub = rospy.Publisher("/mobile_base/commands/velocity", Twist,
queue_size=10)
        self.odom_sub = rospy.Subscriber("odom", Odometry, self.odom_callback)
        self.rate = rospy.Rate(10)
        self.listener = tf.TransformListener()

        # Initialize state variables
        self.left = "O"
        self.front = "F"
        self.wall = "S"
        self.state = "forward"
        self.current_facing = 0  # straight
```

```
    self.logging_counter = 0
    self.left_min_dist = 100
    self.forward_min_dist = 999
    self.pose = Pose2D()
    self.goal_tolerance = 0.3
    self.trajectory = list()
    self.angular_threshold = 0.003
    self.pd_control = PDController()
    self.pd_control.setPD(1.0, 0.1)
```

Initialize the angle for the left state and the right state

Initialize the state of the robot is facing (directions)

```
self.control_list = [pi/2, pi]
    self.control_list2 = [pi/2, 0]
    # Define Finite-State Machine matrix by NumPy
    self.state_transition_matrix = np.array([
        # Left Side: Free
        [   # Front side: free  -> Left
            [1],
            # Front side: Occupied  -> Right
            [2]
        ],
        # Left Side: Occupied
        [   # Front side: free  -> Forward
            [0],
            # Front side: Occupied  -> Right
            [2]
        ]
    ])

    # Define state and condition encoding
    self.state_encoding = {'forward': 0, 'left': 1, 'right': 2}
    self.state_decoding = {0: 'forward', 1: 'left', 2: 'right'}
    self.condition_encoding = {'F': 0, 'O': 1}     # F: free; O: Occupied
```

```
self.current_facing_decoding = {0: "straight", -1: "small_left", -2: "medium_left", -3:
"large_left",
                                1: "small_right", 2: "medium_right", 3: "large_right"}
```

Break out:

Intialize the angle

```
self.control_list = [pi/2, pi]
```

```
self.control_list2 = [pi/2, 0]
```

self.control_list = [pi/2, pi] for the left state

self.control_list2 = [pi/2, 0] for the right state

Initialize the state of the robot is facing:

```
# Define state and condition encoding
```

```
self.state_encoding = {'forward': 0, 'left': 1, 'right': 2}
self.state_decoding = {0: 'forward', 1: 'left', 2: 'right'}
self.condition_encoding = {'F': 0, 'O': 1}      # F: free; O: Occupied
self.current_facing_decoding = {0: "straight", -1: "small_left", -2: "medium_left", -3:
"large_left",
                                1: "small_right", 2: "medium_right", 3: "large_right"}
```

STEP 4:

```
def run(self):
    # Don't edit anything
    while not rospy.is_shutdown():
        self.update_state()  # Update the robot's state based on sensor readings
        # Publish velocity commands based on the current state
        self.publish_velocity()
        # Sleep to maintain the loop rate
        self.rate.sleep()
```

Publish velocity commands based on the current state

STEP 5:

```python
def update_state(self):
    # State machine to update the robot's state based on sensor readings
    current_state_encoding = self.state_encoding[self.state]
    left_cond_encoding = self.condition_encoding[self.left]
    front_cond_encoding = self.condition_encoding[self.front]

    # Write code to encode current state and conditions
    # Get the new state from the state transition matrix
    new_state_encoded = self.state_transition_matrix[left_cond_encoding,
front_cond_encoding][0]

    # Decode the new state
    self.state = self.state_decoding[new_state_encoded]
```

Update the state function

```python
def publish_velocity(self):
    vel = Twist()
    # Publish velocity commands based on the current facing direction
    # Fill in the velocities, keep the values small
    # Keep editing values in the given range till the robot moves well.

    # Velocity values are good in the range of (0.01 to 0.2)
    # Angular Velocities are good in the range of (-0.08 to 0.08)
    if self.current_facing_decoding[self.current_facing] == "straight":  # 0
        vel.linear.x = 0.2
        vel.angular.z = 0.0
    if self.current_facing_decoding[self.current_facing] == "small_left":  # -1
        vel.linear.x = 0.1
        vel.angular.z = 0.03 #0.03
    if self.current_facing_decoding[self.current_facing] == "medium_left":  # -2
        vel.linear.x = 0.1
        vel.angular.z = 0.05  #0.05
    if self.current_facing_decoding[self.current_facing] == "small_right":  # 1
        vel.linear.x = 0.1
        vel.angular.z = -0.03 #0.03
    if self.current_facing_decoding[self.current_facing] == "medium_right":  # 2
        vel.linear.x = 0.1
```

```python
        vel.angular.z = -0.05 #-0.05
print(self.left + ' ' + self.front + ' ' + self.wall + ' ' +
    str(round(self.left_min_dist, 3)) + ' ' + str(round(self.forward_min_dist, 3)) +
    ' ' + self.state + ' ' + str(self.current_facing))

if self.state == "left":
    '''if len(self.control_list) < 1:
        rospy.signal_shutdown('Received shutdown message')'''

    for i in range(5):
        vel.linear.x = 0.2
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep()

    # use PD controller to control the angle: from Lab 5
    pd_control=PDController()
    pd_control.setPoint(self.control_list[0])
    while abs(self.pose.theta -self.control_list[0]) > self.angular_threshold:
        adjust_pos = pd_control.update(self.pose.theta)
        vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
        self.vel_pub.publish(vel)
        self.rate.sleep()
    rospy.loginfo("done with turning left")
    self.control_list.pop(0)

    # move forward a bit: from lab 4
    for i in range(5):
        vel.linear.x = 0.2
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep()

elif self.state == "right":

    if len(self.control_list2) < 1:
        rospy.signal_shutdown('Received shutdown message')
```

```python
        # use PD controller to control the angle: from Lab 5
        pd_control=PDController()
        pd_control.setPoint(self.control_list2[0])
        while abs(self.pose.theta -self.control_list2[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
            self.vel_pub.publish(vel)
            self.rate.sleep()
        rospy.loginfo("done with turning right")
        self.control_list2.pop(0)

        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep

    else:
        self.vel_pub.publish(vel)



def lidar_callback(self, data):
    self.left_min_dist = 100

    # Update the forward distance with the distance directly in front of the robot
    if str(data.ranges[0]) == "inf":
        self.forward_min_dist = 999
    else:
        self.forward_min_dist = data.ranges[0]
```

The main goal of this publish_velocity() function is to define the state of the robot.

```python
        if self.current_facing_decoding[self.current_facing] == "straight":  # 0
```

```python
        vel.linear.x = 0.2
        vel.angular.z = 0.0
```

```python
if self.current_facing_decoding[self.current_facing] == "small_left": # -1
    vel.linear.x = 0.1
    vel.angular.z = 0.03 #0.03
if self.current_facing_decoding[self.current_facing] == "medium_left": # -2
    vel.linear.x = 0.1
    vel.angular.z = 0.05 #0.05
if self.current_facing_decoding[self.current_facing] == "small_right": # 1
    vel.linear.x = 0.1
    vel.angular.z = -0.03 #0.03
if self.current_facing_decoding[self.current_facing] == "medium_right": # 2
    vel.linear.x = 0.1
    vel.angular.z = -0.05 #-0.05
print(self.left + ' ' + self.front + ' ' + self.wall + ' ' +
    str(round(self.left_min_dist, 3)) + ' ' + str(round(self.forward_min_dist, 3)) +
    ' ' + self.state + ' ' + str(self.current_facing))
```

This state machine above will initialize all the forward movements and the angular movement of the robot.

When the state = left:

```python
if self.state == "left":
```

```python
'''if len(self.control_list) < 1:
    rospy.signal_shutdown('Received shutdown message')'''

for i in range(5):
    vel.linear.x = 0.2
    vel.angular.z = 0.0
    self.vel_pub.publish(vel)
    self.rate.sleep()

# use PD controller to control the angle: from Lab 5
pd_control=PDController()
pd_control.setPoint(self.control_list[0])
while abs(self.pose.theta -self.control_list[0]) > self.angular_threshold:
    adjust_pos = pd_control.update(self.pose.theta)
    vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
    self.vel_pub.publish(vel)
    self.rate.sleep()
rospy.loginfo("done with turning left")
```

```python
        self.control_list.pop(0)

        # move forward a bit: from lab 4
        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep()
```

In the left state, first the robot will move forward a bit,

```python
        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep()
```

Next :

```python
        # use PD controller to control the angle: from Lab 5
        pd_control=PDController()
        pd_control.setPoint(self.control_list[0])
        while abs(self.pose.theta -self.control_list[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
            self.vel_pub.publish(vel)
            self.rate.sleep()
        rospy.loginfo("done with turning left")
        self.control_list.pop(0)
```

Based on this code above , the goal is rotating based on the desire rotaion on the left state. The robot will compare its current angle, self.pose.thetha, with the desire angle rotation, self.control_list[0]. If the range difference is greater than the threshold (0.003), then it keeps rotating based on the tunned PD controller . If the range difference is smaller than the threshold, the robot will stop.

After that , it will pop the first angle so the next angle rotation will be ready to be executed.

<mark>**When the state = right:**</mark>

The process is very similar to the left state

```python
elif self.state == "right":
```

```python
    if len(self.control_list2) < 1:
        rospy.signal_shutdown('Received shutdown message')




    # use PD controller to control the angle: from Lab 5
    pd_control=PDController()
    pd_control.setPoint(self.control_list2[0])
    while abs(self.pose.theta -self.control_list2[0]) > self.angular_threshold:
        adjust_pos = pd_control.update(self.pose.theta)
        vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
        self.vel_pub.publish(vel)
        self.rate.sleep()
    rospy.loginfo("done with turning right")
    self.control_list2.pop(0)

    for i in range(5):
        vel.linear.x = 0.2
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep

else:
    self.vel_pub.publish(vel)
```

```python
    # use PD controller to control the angle: from Lab 5
    pd_control=PDController()
    pd_control.setPoint(self.control_list2[0])
```

```python
        while abs(self.pose.theta -self.control_list2[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
            self.vel_pub.publish(vel)
            self.rate.sleep()
        rospy.loginfo("done with turning right")
        self.control_list2.pop(0)
```

Based on this code above , the goal is rotating based on the desire rotaion on the right state. The robot will compare its current angle, self.pose.thetha, with the desire angle rotation, self.control_list[0]. If the range difference is greater than the threshold (0.003), then it keeps rotating based on the tunned PD controller . If the range difference is smaller than the threshold, the robot will stop.

After that , it will pop the first angle so the next angle rotation will be ready to be executed.

```python
for i in range(5):
        vel.linear.x = 0.2
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep

    else:
        self.vel_pub.publish(vel)
```

Continue moving forward

If the state is not left, or right, just publish the velocity of the robot.

For the LIDAR CALLBACK function():

```python
def lidar_callback(self, data):
    self.left_min_dist = 100

    # Update the forward distance with the distance directly in front of the robot
    if str(data.ranges[0]) == "inf":
        self.forward_min_dist = 999
```

```python
        else:
            self.forward_min_dist = data.ranges[0]

        # transform the lidar points frame /rplidar_link from to another frame: from lab 3 and
lab 2
        #listener = tf.TransformListener()
        (trans, rot) = self.listener.lookupTransform('/base_link', '/rplidar_link', rospy.Time(0))



        # Process the LIDAR data and transform the points to the robot's coordinate frame
(another frame you specified)
        for i in range(len(data.ranges)):
            # get the left side lidar data
            if i * data.angle_increment < 1.59 and i * data.angle_increment > 1.55:
                if str(data.ranges[i]) == "inf":
                    dist = 9999
                else:
                    dist = data.ranges[i]
                (x, y) = self.calculate_position_of_range(dist, i, data.angle_increment,
data.angle_min)

                transformed_point = np.dot(tf.transformations.quaternion_matrix(rot),
np.array([x, y, 0, 1]))

                left_dist = transformed_point[1]
                if left_dist < self.left_min_dist:
                    self.left_min_dist = left_dist

        # Update left and forward state
        if self.left_min_dist < 0.5:
            self.left = "O"
        else:
            self.left = "F"
        if self.forward_min_dist < 0.5:
            self.front = "O"
        else:
            self.front = "F"

        # Set wall state
```

```
    if self.left_min_dist < 0.3:
        self.wall = "C"
    else:
        self.wall = "S"
```

Breakout the code explanation:

```
def lidar_callback(self, data):
    self.left_min_dist = 100

    # Update the forward distance with the distance directly in front of the robot
    if str(data.ranges[0]) == "inf":
        self.forward_min_dist = 999
    else:
        self.forward_min_dist = data.ranges[0]

    # transform the lidar points frame /rplidar_link from to another frame: from lab 3 and
lab 2
    #listener = tf.TransformListener()
    (trans, rot) = self.listener.lookupTransform('/base_link', '/rplidar_link', rospy.Time(0))
```

Initialize the transformation and rotation for the lidar

If the robot distance of the lidar is at infinity, so set the forward direction equals to 999

If not, then start the forward_min_dist

Next, processing the LIDAR data and transform the points to the robot's coordinate frame

If the data angle increment is < 1.59 or >1.55:

-> the range of the data is infinity

```
    for i in range(len(data.ranges)):
        # get the left side lidar data
        if i * data.angle_increment < 1.59 and i * data.angle_increment > 1.55:
            if str(data.ranges[i]) == "inf":
                dist = 9999
            else:
                dist = data.ranges[i]
```

```
        (x, y) = self.calculate_position_of_range(dist, i, data.angle_increment,
data.angle_min)

        transformed_point = np.dot(tf.transformations.quaternion_matrix(rot),
np.array([x, y, 0, 1]))

        left_dist = transformed_point[1]
        if left_dist < self.left_min_dist:
            self.left_min_dist = left_dist
```

For the segment code above, call the calculate_position_of_range()

The calculate_position_of_range() function is:

```python
def calculate_position_of_range(self, range, index, angle_increment,
angle_min):
```

```
    # Calculate the position of a range measurement in the LIDAR frame
    angle = index * angle_increment + angle_min
    x = range * np.cos(angle)
    y = range * np.sin(angle)
    return x, y
```

The `range` is the distance measurement from the LIDAR sensor

`np.cos(angle)` calculates the cosine of the angle, and when this value is multiplied by the range, it gives the x-coordinate. Similar to the y-coordinate

 4-dimensional for the transform_point

  if left_dist < self.left_min_dist:

       keep the minimum distance as the left_min_dist

Update left and forward state

```python
    if self.left_min_dist < 0.5:
        self.left = "O"
    else:
        self.left = "F"
    if self.forward_min_dist < 0.5:
        self.front = "O"
    else:
        self.front = "F"

    # Set wall state
    if self.left_min_dist < 0.3:
        self.wall = "C"
    else:
        self.wall = "S"
```

For the LEFT

Update current_facing direction

if the robot is too close, take the medium_right facing; if it is too far, take the medium_left facing

For this small_right and small_left are used to fine-tune the navigation.

```python
    # The basic idea is:
    # if the robot is too close, take the medium_right facing; if it is too far, take the
medium_left facing
    # For this small_right and small_left are used to fine-tune the navigation.
    if self.left_min_dist < 0.3:
        self.current_facing = 1
    elif self.left_min_dist < 0.25:
        self.current_facing = 2
    elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 2) :
        self.current_facing = 1
    elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 1) :
        self.current_facing = 2
    elif (self.left_min_dist > 0.35) and (self.forward_min_dist > 2) :
        self.current_facing = -1
```

```python
    elif (self.left_min_dist > 0.4)  :
        self.current_facing = -2
    else:
        self.current_facing = 0
```

For the ODOM_CALLBACK function():

```python
        def odom_callback(self, data):
```

```python
    # get the position of the robot
    pose = data.pose.pose
    self.pose.x = pose.position.x
    self.pose.y = pose.position.y
    quaternion = (
        pose.orientation.x,
        pose.orientation.y,
        pose.orientation.z,
        pose.orientation.w
    )
    euler = tf.transformations.euler_from_quaternion(quaternion)
    self.pose.theta = euler[2]

    # Append pose data to the trajectory
    self.logging_counter += 1
    if self.logging_counter == 100:
        self.logging_counter = 0
        self.trajectory.append([self.pose.x, self.pose.y])
        rospy.loginfo("Save point")

    # If the goal is reached, stop the robot and save the trajectory
    if sqrt((self.pose.x - self.goal_x)**2 + (self.pose.y - self.goal_y)**2) < self.goal_tolerance:
        rospy.loginfo("Goal reached")
        self.stop()
        self.save_trajectory()
        rospy.signal_shutdown('Goal reached')
```

Breakout the code segment:

```python
    pose = data.pose.pose
```

```
self.pose.x = pose.position.x
self.pose.y = pose.position.y
quaternion = (
    pose.orientation.x,
    pose.orientation.y,
    pose.orientation.z,
    pose.orientation.w
)
euler = tf.transformations.euler_from_quaternion(quaternion)
self.pose.theta = euler[2]
```

Get the position of x,y for ROS of odometry

-> rospy.Subscriber("odom", Odometry, self.odom_callback)

```
quaternion = (
    pose.orientation.x,
    pose.orientation.y,
    pose.orientation.z,
    pose.orientation.w
)
euler = tf.transformations.euler_from_quaternion(quaternion)
self.pose.theta = euler[2]
```

For the rotation of x,y,z,w

Next,

Append pose data to the trajectory

```
self.logging_counter += 1
if self.logging_counter == 100:
    self.logging_counter = 0
    self.trajectory.append([self.pose.x, self.pose.y])
    rospy.loginfo("Save point")
```

Next,

If the goal is reached, stop the robot and save the trajectory

To calculate the goal distance, the formula is

$$x = \sqrt{((self.pose.x - self.goal\_x)^{**}2 + (self.pose.y - self.goal\_y)^{**}2)}$$

If x< self_goal.toleracne (0.003) then the goal is reached

```python
if sqrt((self.pose.x - self.goal_x)**2 + (self.pose.y - self.goal_y)**2) < self.goal_tolerance:
    rospy.loginfo("Goal reached")
    self.stop()
    self.save_trajectory()
    rospy.signal_shutdown('Goal reached')
```

Last step for the main function():

```python
def main(args):
```

```python
# Initialize ROS Node
rospy.init_node('left_wall_follower', anonymous=True)
# Get parameters from the launch file
goal_x = rospy.get_param('goal_x')
goal_y = rospy.get_param('goal_y')
csv_file = rospy.get_param('csv_file')

# Create an instance of the Turtlebot class
robot = Turtlebot(goal_x, goal_y, csv_file)
try:
    rospy.spin()
except KeyboardInterrupt:
    print("Shutting down")

if __name__ == '__main__':
    try:
        main(sys.argv)
    except rospy.ROSInterruptException:
        rospy.loginfo("Action terminated.")
```

FOR THE MORE COMPLEX WORLD:

The step is exactly the same as the complex world. The only thing we modified is adding the angle angle, pi, into the self arrayas the following below:

self.control_list = [pi/2, pi,pi/2) References

That's all for this lab.

The external source is only: [Welcome to UCR EE106 — UCR EE106 Spring 2024 0.1 documentation (ucr-ee106.readthedocs.io)](#) and the help from TA

3) Scripts

For the complex world:

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

import roslib
roslib.load_manifest('ee106s24')
import rospy
import sys
import tf
import numpy as np
from math import pi, sqrt
from std_msgs.msg import String
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist, Pose2D
from nav_msgs.msg import Odometry
import os

class PDController:
    def __init__(self, P=1.0, D=0.1, set_point=0.0):
        self.Kp = P
        self.Kd = D
        self.set_point = set_point  # reference (desired value)
        self.previous_error = 0

    def update(self, current_value):
        # calculate P_term and D_term
        #  e = r - y
        error = self.set_point - current_value
        P_term = self.Kp * error
        D_term = self.Kd * (error - self.previous_error)
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
```

```python
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D

class Turtlebot():
    def __init__(self, goal_x, goal_y, csv_file):
        # Data to be taken from Launch File. Don't Edit
        self.goal_x = goal_x
        self.goal_y = goal_y
        self.csv_file = csv_file

        # Initialize subscribers and publishers
        self.lidar_sub = rospy.Subscriber('/scan', LaserScan, self.lidar_callback)
        self.vel_pub = rospy.Publisher("/mobile_base/commands/velocity", Twist,
queue_size=10)
        self.odom_sub = rospy.Subscriber("odom", Odometry, self.odom_callback)
        self.rate = rospy.Rate(10)
        self.listener = tf.TransformListener()

        # Initialize state variables
        self.left = "O"
        self.front = "F"
        self.wall = "S"
        self.state = "forward"
        self.current_facing = 0  # straight
        self.logging_counter = 0
        self.left_min_dist = 100
        self.forward_min_dist = 999
        self.pose = Pose2D()
        self.goal_tolerance = 0.3
        self.trajectory = list()
        self.angular_threshold = 0.003
        self.pd_control = PDController()
        self.pd_control.setPD(1.0, 0.1)

        # For the complex world map
        self.control_list = [pi/2, pi]
```

```python
        self.control_list2 = [pi/2, 0]
        # Define Finite-State Machine matrix by NumPy
        self.state_transition_matrix = np.array([
            # Left Side: Free
            [   # Front side: free  -> Left
                [1],
                # Front side: Occupied  -> Right
                [2]
            ],
            # Left Side: Occupied
            [   # Front side: free  -> Forward
                [0],
                # Front side: Occupied  -> Right
                [2]
            ]
        ])

        # Define state and condition encoding
        self.state_encoding = {'forward': 0, 'left': 1, 'right': 2}
        self.state_decoding = {0: 'forward', 1: 'left', 2: 'right'}
        self.condition_encoding = {'F': 0, 'O': 1}     # F: free; O: Occupied
        self.current_facing_decoding = {0: "straight", -1: "small_left", -2: "medium_left", -3:
"large_left",
                        1: "small_right", 2: "medium_right", 3: "large_right"}
        self.run()

    def run(self):
        # Don't edit anything
        while not rospy.is_shutdown():
            self.update_state()  # Update the robot's state based on sensor readings
            # Publish velocity commands based on the current state
            self.publish_velocity()
            # Sleep to maintain the loop rate
            self.rate.sleep()

    def update_state(self):
        # State machine to update the robot's state based on sensor readings
        current_state_encoding = self.state_encoding[self.state]
        left_cond_encoding = self.condition_encoding[self.left]
        front_cond_encoding = self.condition_encoding[self.front]
```

```python
        # Write code to encode current state and conditions
        # Get the new state from the state transition matrix
        new_state_encoded = self.state_transition_matrix[left_cond_encoding,
front_cond_encoding][0]

        # Decode the new state
        self.state = self.state_decoding[new_state_encoded]

    def publish_velocity(self):
        vel = Twist()
        # Publish velocity commands based on the current facing direction
        # Fill in the velocities, keep the values small
        # Keep editing values in the given range till the robot moves well.

        # Velocity values are good in the range of (0.01 to 0.2)
        # Angular Velocities are good in the range of (-0.08 to 0.08)
        if self.current_facing_decoding[self.current_facing] == "straight":  # 0
            vel.linear.x = 0.2
            vel.angular.z = 0.0
        if self.current_facing_decoding[self.current_facing] == "small_left":  # -1
            vel.linear.x = 0.1
            vel.angular.z = 0.03 #0.03
        if self.current_facing_decoding[self.current_facing] == "medium_left":  # -2
            vel.linear.x = 0.1
            vel.angular.z = 0.05  #0.05
        if self.current_facing_decoding[self.current_facing] == "small_right":  # 1
            vel.linear.x = 0.1
            vel.angular.z = -0.03 #0.03
        if self.current_facing_decoding[self.current_facing] == "medium_right":  # 2
            vel.linear.x = 0.1
            vel.angular.z = -0.05 #-0.05
        print(self.left + ' ' + self.front + ' ' + self.wall + ' ' +
            str(round(self.left_min_dist, 3)) + ' ' + str(round(self.forward_min_dist, 3)) +
            ' ' + self.state + ' ' + str(self.current_facing))

        if self.state == "left":
            '''if len(self.control_list) < 1:
                rospy.signal_shutdown('Received shutdown message')'''
```

```python
        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep()

        # use PD controller to control the angle: from Lab 5
        pd_control=PDController()
        pd_control.setPoint(self.control_list[0])
        while abs(self.pose.theta -self.control_list[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
            self.vel_pub.publish(vel)
            self.rate.sleep()
        rospy.loginfo("done with turning left")
        self.control_list.pop(0)

        # move forward a bit: from lab 4
        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep()

    elif self.state == "right":

        if len(self.control_list2) < 1:
            rospy.signal_shutdown('Received shutdown message')




        # use PD controller to control the angle: from Lab 5
        pd_control=PDController()
        pd_control.setPoint(self.control_list2[0])
        while abs(self.pose.theta -self.control_list2[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.8))
            self.vel_pub.publish(vel)
```

```python
            self.rate.sleep()
        rospy.loginfo("done with turning right")
        self.control_list2.pop(0)

        for i in range(5):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep

    else:
        self.vel_pub.publish(vel)




def lidar_callback(self, data):
    self.left_min_dist = 100

    # Update the forward distance with the distance directly in front of the robot
    if str(data.ranges[0]) == "inf":
        self.forward_min_dist = 999
    else:
        self.forward_min_dist = data.ranges[0]

    # transform the lidar points frame /rplidar_link from to another frame: from lab 3 and
lab 2
    #listener = tf.TransformListener()
    (trans, rot) = self.listener.lookupTransform('/base_link', '/rplidar_link', rospy.Time(0))



    # Process the LIDAR data and transform the points to the robot's coordinate frame
(another frame you specified)
    for i in range(len(data.ranges)):
        # get the left side lidar data
        if i * data.angle_increment < 1.59 and i * data.angle_increment > 1.55:
            if str(data.ranges[i]) == "inf":
                dist = 9999
            else:
```

```python
            dist = data.ranges[i]
            (x, y) = self.calculate_position_of_range(dist, i, data.angle_increment,
data.angle_min)

            transformed_point = np.dot(tf.transformations.quaternion_matrix(rot),
np.array([x, y, 0, 1]))

            left_dist = transformed_point[1]
            if left_dist < self.left_min_dist:
                self.left_min_dist = left_dist

        # Update left and forward state
        if self.left_min_dist < 0.5:
            self.left = "O"
        else:
            self.left = "F"
        if self.forward_min_dist < 0.5:
            self.front = "O"
        else:
            self.front = "F"

        # Set wall state
        if self.left_min_dist < 0.3:
            self.wall = "C"
        else:
            self.wall = "S"

        # Update current_facing direction
        # The basic idea is:
        #  if the robot is too close, take the medium_right facing; if it is too far, take the
medium_left facing
        #  For this small_right and small_left are used to fine-tune the navigation.
        if self.left_min_dist < 0.3:
            self.current_facing = 1
        elif self.left_min_dist < 0.25:
            self.current_facing = 2
        elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 2) :
            self.current_facing = 1
        elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 1) :
            self.current_facing = 2
```

```python
        elif (self.left_min_dist > 0.35) and (self.forward_min_dist > 2) :
            self.current_facing = -1
        elif (self.left_min_dist > 0.4)  :
            self.current_facing = -2
        else:
            self.current_facing = 0

    def odom_callback(self, data):
        # get the position of the robot
        pose = data.pose.pose
        self.pose.x = pose.position.x
        self.pose.y = pose.position.y
        quaternion = (
            pose.orientation.x,
            pose.orientation.y,
            pose.orientation.z,
            pose.orientation.w
        )
        euler = tf.transformations.euler_from_quaternion(quaternion)
        self.pose.theta = euler[2]

        # Append pose data to the trajectory
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])
            rospy.loginfo("Save point")

        # If the goal is reached, stop the robot and save the trajectory
        if sqrt((self.pose.x - self.goal_x)**2 + (self.pose.y - self.goal_y)**2) < self.goal_tolerance:
            rospy.loginfo("Goal reached")
            self.stop()
            self.save_trajectory()
            rospy.signal_shutdown('Goal reached')

    def stop(self):
        # Stop the robot
        vel = Twist()
        vel.linear.x = 0.0
        vel.angular.z = 0.0
```

```python
        self.vel_pub.publish(vel)

    def save_trajectory(self):
        # Save the trajectory to a CSV file
        '''with open(self.csv_file, 'w') as f:
            for pose in self.trajectory:
                f.write(f"{pose[0]},{pose[1]}\n")'''
        np.savetxt(self.csv_file, np.array(self.trajectory), fmt='%f', delimiter=',')

    def calculate_position_of_range(self, range, index, angle_increment, angle_min):
        # Calculate the position of a range measurement in the LIDAR frame
        angle = index * angle_increment + angle_min
        x = range * np.cos(angle)
        y = range * np.sin(angle)
        return x, y


'''if __name__ == '__main__':
    try:
        # Initialize the ROS node
        rospy.init_node('turtlebot_navigation', anonymous=True)

        # Get parameters from the launch file
        goal_x = rospy.get_param("~goal_x", 5.0)
        goal_y = rospy.get_param("~goal_y", 5.0)
        csv_file = rospy.get_param("~csv_file", "/home/desktop/trajectory.csv")

        # Create the Turtlebot object and start the navigation
        turtlebot = Turtlebot(goal_x, goal_y, csv_file)
    except rospy.ROSInterruptException:
        pass'''
def main(args):
    # Initialize ROS Node
    rospy.init_node('left_wall_follower', anonymous=True)
    # Get parameters from the launch file
    goal_x = rospy.get_param('goal_x')
    goal_y = rospy.get_param('goal_y')
    csv_file = rospy.get_param('csv_file')

    # Create an instance of the Turtlebot class
    robot = Turtlebot(goal_x, goal_y, csv_file)
```

```python
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    try:
        main(sys.argv)
    except rospy.ROSInterruptException:
        rospy.loginfo("Action terminated.")
```

MORE COMEPLEX WORLD:

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

import roslib
roslib.load_manifest('ee106s24')
import rospy
import sys
import tf
import numpy as np
from math import pi, sqrt
from std_msgs.msg import String
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist, Pose2D
from nav_msgs.msg import Odometry
import os

class PDController:
    def __init__(self, P=1.0, D=0.1, set_point=0.0):
        self.Kp = P
        self.Kd = D
        self.set_point = set_point  # reference (desired value)
        self.previous_error = 0

    def update(self, current_value):
```

```python
        # calculate P_term and D_term
        # e = r - y
        error = self.set_point - current_value
        P_term = self.Kp * error
        D_term = self.Kd * (error - self.previous_error)
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D

class Turtlebot():
    def __init__(self, goal_x, goal_y, csv_file):
        # Data to be taken from Launch File. Don't Edit
        self.goal_x = goal_x
        self.goal_y = goal_y
        self.csv_file = csv_file

        # Initialize subscribers and publishers
        self.lidar_sub = rospy.Subscriber('/scan', LaserScan, self.lidar_callback)
        self.vel_pub = rospy.Publisher("/mobile_base/commands/velocity", Twist, queue_size=10)
        self.odom_sub = rospy.Subscriber("odom", Odometry, self.odom_callback)
        self.rate = rospy.Rate(10)
        self.listener = tf.TransformListener()

        # Initialize state variables
        self.left = "O"
        self.front = "F"
        self.wall = "S"
        self.state = "forward"
        self.current_facing = 0   # straight
        self.logging_counter = 0
        self.left_min_dist = 100
        self.forward_min_dist = 999
        self.pose = Pose2D()
```

```python
        self.goal_tolerance = 0.3
        self.trajectory = list()
        self.angular_threshold = 0.003
        self.pd_control = PDController()
        self.pd_control.setPD(1.0, 0.1)

        # For the complex world map
        self.control_list = [pi/2, pi,pi/2]
        self.control_list2 = [pi/2, 0]
        # Define Finite-State Machine matrix by NumPy
        self.state_transition_matrix = np.array([
            # Left Side: Free
            [  # Front side: free  -> Left
                [1],
                # Front side: Occupied  -> Right
                [2]
            ],
            # Left Side: Occupied
            [  # Front side: free  -> Forward
                [0],
                # Front side: Occupied  -> Right
                [2]
            ]
        ])

        # Define state and condition encoding
        self.state_encoding = {'forward': 0, 'left': 1, 'right': 2}
        self.state_decoding = {0: 'forward', 1: 'left', 2: 'right'}
        self.condition_encoding = {'F': 0, 'O': 1}     # F: free; O: Occupied
        self.current_facing_decoding = {0: "straight", -1: "small_left", -2: "medium_left", -3:
"large_left",
                            1: "small_right", 2: "medium_right", 3: "large_right"}
        self.run()

    def run(self):
        # Don't edit anything
        while not rospy.is_shutdown():
            self.update_state()  # Update the robot's state based on sensor readings
            # Publish velocity commands based on the current state
            self.publish_velocity()
```

```python
        # Sleep to maintain the loop rate
        self.rate.sleep()

    def update_state(self):
        # State machine to update the robot's state based on sensor readings
        current_state_encoding = self.state_encoding[self.state]
        left_cond_encoding = self.condition_encoding[self.left]
        front_cond_encoding = self.condition_encoding[self.front]

        # Write code to encode current state and conditions
        # Get the new state from the state transition matrix
        new_state_encoded = self.state_transition_matrix[left_cond_encoding,
front_cond_encoding][0]

        # Decode the new state
        self.state = self.state_decoding[new_state_encoded]

    def publish_velocity(self):
        vel = Twist()
        # Publish velocity commands based on the current facing direction
        # Fill in the velocities, keep the values small
        # Keep editing values in the given range till the robot moves well.

        # Velocity values are good in the range of (0.01 to 0.2)
        # Angular Velocities are good in the range of (-0.08 to 0.08)
        if self.current_facing_decoding[self.current_facing] == "straight":  # 0
            vel.linear.x = 0.2
            vel.angular.z = 0.0
        if self.current_facing_decoding[self.current_facing] == "small_left":  # -1
            vel.linear.x = 0.1
            vel.angular.z = 0.03 #0.03
        if self.current_facing_decoding[self.current_facing] == "medium_left":  # -2
            vel.linear.x = 0.1
            vel.angular.z = 0.05  #0.05
        if self.current_facing_decoding[self.current_facing] == "small_right":  # 1
            vel.linear.x = 0.1
            vel.angular.z = -0.03 #0.03
        if self.current_facing_decoding[self.current_facing] == "medium_right":  # 2
            vel.linear.x = 0.1
            vel.angular.z = -0.05 #-0.05
```

```python
print(self.left + ' ' + self.front + ' ' + self.wall + ' ' +
    str(round(self.left_min_dist, 3)) + ' ' + str(round(self.forward_min_dist, 3)) +
    ' ' + self.state + ' ' + str(self.current_facing))

if self.state == "left":
    if len(self.control_list) < 1:
        rospy.signal_shutdown('Received shutdown message')



    for i in range(5):
        vel.linear.x = 0.15
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep()

    # use PD controller to control the angle: from Lab 5
    pd_control=PDController()
    pd_control.setPoint(self.control_list[0])
    while abs(self.pose.theta - self.control_list[0]) > self.angular_threshold:
        adjust_pos = pd_control.update(self.pose.theta)
        vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.9))
        self.vel_pub.publish(vel)
        self.rate.sleep()
    rospy.loginfo("done with turning left")
    self.control_list.pop(0)

    # move forward a bit: from lab 4
    for i in range(15):
        vel.linear.x = 0.2
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)
        self.rate.sleep()

elif self.state == "right":

    if len(self.control_list2) < 1:
        rospy.signal_shutdown('Received shutdown message')
```

```python
        # use PD controller to control the angle: from Lab 5
        pd_control=PDController()
        pd_control.setPoint(self.control_list2[0])
        while abs(self.pose.theta -self.control_list2[0]) > self.angular_threshold:
            adjust_pos = pd_control.update(self.pose.theta)
            vel.angular.z = max(-0.8,min(adjust_pos*2.5,0.9))
            self.vel_pub.publish(vel)
            self.rate.sleep()
        rospy.loginfo("done with turning right")
        self.control_list2.pop(0)

        for i in range(6):
            vel.linear.x = 0.2
            vel.angular.z = 0.0
            self.vel_pub.publish(vel)
            self.rate.sleep

    else:
        self.vel_pub.publish(vel)




def lidar_callback(self, data):
    self.left_min_dist = 100

    # Update the forward distance with the distance directly in front of the robot
    if str(data.ranges[0]) == "inf":
        self.forward_min_dist = 999
    else:
        self.forward_min_dist = data.ranges[0]

    # transform the lidar points frame /rplidar_link from to another frame: from lab 3 and lab 2
    #listener = tf.TransformListener()
    (trans, rot) = self.listener.lookupTransform('/base_link', '/rplidar_link', rospy.Time(0))
```

```python
    # Process the LIDAR data and transform the points to the robot's coordinate frame (another
frame you specified)
    for i in range(len(data.ranges)):
        # get the left side lidar data
        if i * data.angle_increment < 1.59 and i * data.angle_increment > 1.55:
            if str(data.ranges[i]) == "inf":
                dist = 9999
            else:
                dist = data.ranges[i]
            (x, y) = self.calculate_position_of_range(dist, i, data.angle_increment, data.angle_min)

            transformed_point = np.dot(tf.transformations.quaternion_matrix(rot), np.array([x, y, 0,
1]))

            left_dist = transformed_point[1]
            if left_dist < self.left_min_dist:
                self.left_min_dist = left_dist

    # Update left and forward state
    if self.left_min_dist < 0.5:
        self.left = "O"
    else:
        self.left = "F"
    if self.forward_min_dist < 0.5:
        self.front = "O"
    else:
        self.front = "F"

    # Set wall state
    if self.left_min_dist < 0.3:
        self.wall = "C"
    else:
        self.wall = "S"

    # Update current_facing direction
    # The basic idea is:
    # if the robot is too close, take the medium_right facing; if it is too far, take the medium_left
facing
```

```python
        #  For this small_right and small_left are used to fine-tune the navigation.
        if self.left_min_dist < 0.3:
            self.current_facing = 1
        elif self.left_min_dist < 0.25:
            self.current_facing = 2
        elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 2) :
            self.current_facing = 1
        elif (self.left_min_dist > 0.35) and (self.forward_min_dist < 1.5) :
            self.current_facing = 2
        elif (self.left_min_dist > 0.3) and (self.forward_min_dist > 2) :
            self.current_facing = -1
        elif (self.left_min_dist > 0.35)  :
            self.current_facing = -2
        else:
            self.current_facing = 0


    def odom_callback(self, data):
        # get the position of the robot
        pose = data.pose.pose
        self.pose.x = pose.position.x
        self.pose.y = pose.position.y
        quaternion = (
            pose.orientation.x,
            pose.orientation.y,
            pose.orientation.z,
            pose.orientation.w
        )
        euler = tf.transformations.euler_from_quaternion(quaternion)
        self.pose.theta = euler[2]

        # Append pose data to the trajectory
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])


        # If the goal is reached, stop the robot and save the trajectory
        if sqrt((self.pose.x - self.goal_x)**2 + (self.pose.y - self.goal_y)**2) < self.goal_tolerance:
            rospy.loginfo("Goal reached")
```

```python
        self.stop()
        self.save_trajectory()
        rospy.signal_shutdown('Goal reached')

    def stop(self):
        # Stop the robot
        vel = Twist()
        vel.linear.x = 0.0
        vel.angular.z = 0.0
        self.vel_pub.publish(vel)

    def save_trajectory(self):
        # Save the trajectory to a CSV file
        '''with open(self.csv_file, 'w') as f:
            for pose in self.trajectory:
                f.write(f"{pose[0]},{pose[1]}\n")'''
        np.savetxt(self.csv_file, np.array(self.trajectory), fmt='%f', delimiter=',')

    def calculate_position_of_range(self, range, index, angle_increment, angle_min):
        # Calculate the position of a range measurement in the LIDAR frame
        angle = index * angle_increment + angle_min
        x = range * np.cos(angle)
        y = range * np.sin(angle)
        return x, y

'''if __name__ == '__main__':
    try:
        # Initialize the ROS node
        rospy.init_node('turtlebot_navigation', anonymous=True)

        # Get parameters from the launch file
        goal_x = rospy.get_param("~goal_x", 5.0)
        goal_y = rospy.get_param("~goal_y", 5.0)
        csv_file = rospy.get_param("~csv_file", "/home/desktop/trajectory.csv")

        # Create the Turtlebot object and start the navigation
        turtlebot = Turtlebot(goal_x, goal_y, csv_file)
    except rospy.ROSInterruptException:
        pass'''
def main(args):
```

```python
    # Initialize ROS Node
    rospy.init_node('left_wall_follower', anonymous=True)
    # Get parameters from the launch file
    goal_x = rospy.get_param('goal_x')
    goal_y = rospy.get_param('goal_y')
    csv_file = rospy.get_param('csv_file')

    # Create an instance of the Turtlebot class
    robot = Turtlebot(goal_x, goal_y, csv_file)
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")


if __name__ == '__main__':
    try:
        main(sys.argv)
    except rospy.ROSInterruptException:
        rospy.loginfo("Action terminated.")
```