

# **EECS 106B : Lab #3**

Spring 2018

**Yichi Zhang, Jingjun Liu, Jiarong Li**  
**27005775, 3033483513, 3033483511**

May 27, 2018

## Section 1

### Video

[https://www.youtube.com/watch?v=tk\\_1krAcWDA](https://www.youtube.com/watch?v=tk_1krAcWDA)

## Section 2

### Methods

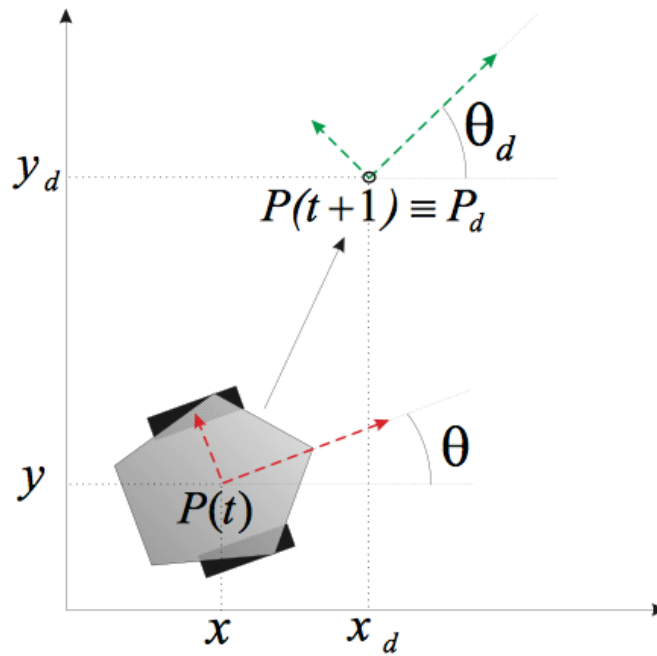
#### 1. Controller

Consider a virtual unicycle type robot governed by

$$\begin{cases} \dot{x}_r = v_r \cos(\theta_r) \\ \dot{y}_r = v_r \sin(\theta_r) \\ \dot{\theta}_r = \omega_r \end{cases}$$

where  $p_r(t) = (x_r(t), y_r(t))$  is the desired reference position at time  $t$ . The objective is to derive an outerloop control system responsible to generate the adequate desired linear and angular velocities  $(V_d, \omega_d)$  to force the position of the robot  $p = (x, y)$  to converge to the reference position  $p_r = (x_r, y_r)$ .

Figure 1: Parallel Parking Path



To achieve this, consider first the position error expressed in the body referential

$$e = R(p_r - p), \quad R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

Clearly, if  $e$  goes to zero then  $p$  converges to  $p_r$ , the control law is given by

$$\begin{bmatrix} v_d \\ \omega_d \end{bmatrix} = C \begin{bmatrix} v_r \\ \omega_r \end{bmatrix} - \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

where  $C = \text{diag}\{\cos(\theta_r - \theta), 1\}$ ,  $u_1 = -k_1(x_r - x)$ ,  $u_2 = k_2 v \sin(\theta_r - \theta)(y_r - y) - k_3(\theta_r - \theta)$  and  $k_i, i = 1, 2, 3$  are positive constant gains.

path	linear	arc
$\theta_r$	0	$\frac{s}{r}$
$x_r$	s	$r \sin(\theta)$
$y_r$	0	$r - r \cos(\theta)$

s is the total distance that the turtlebot has been traveled.

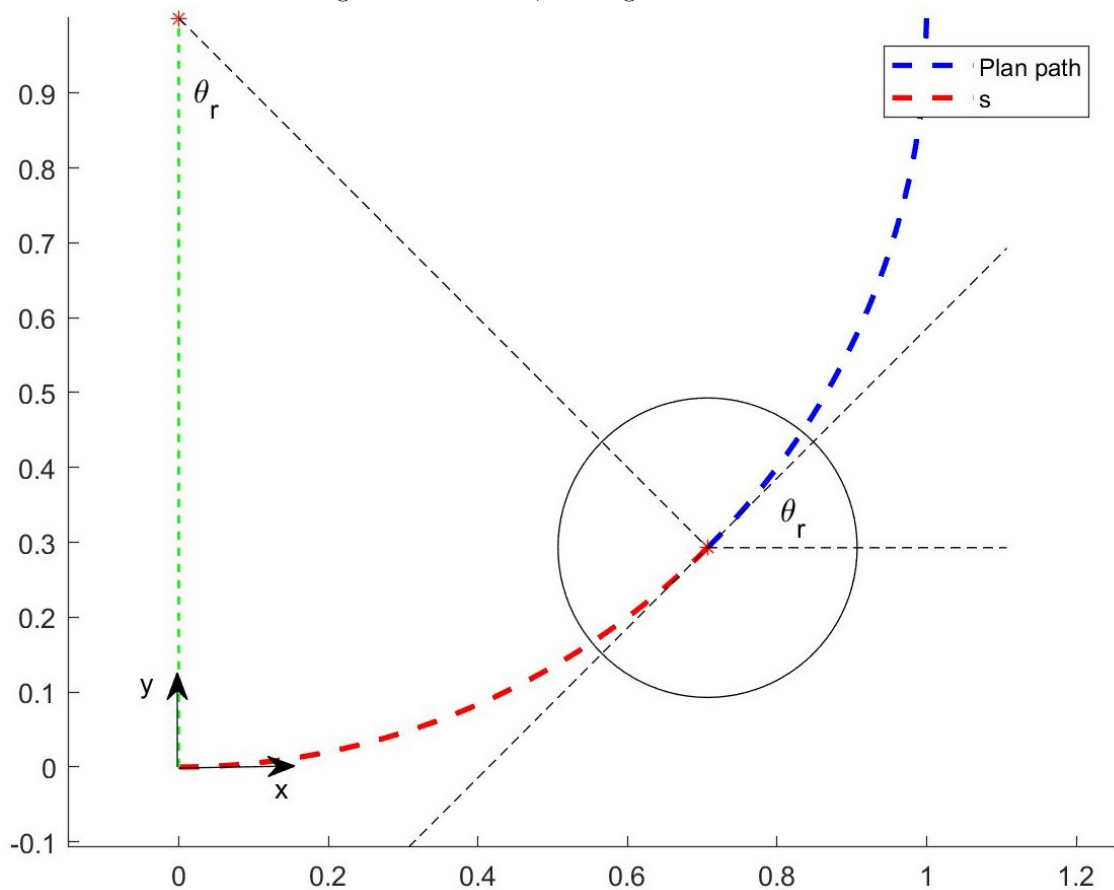
r is the radius of the arc circle.

$\theta, x, y$  are got from the current state function from turtlebot, which is referred to the initial position of the turtlebot, and we calculate and use the row pitch yaw from the return quaternion instead of using quaternion directly.

We also use two factor to define the direction of this control motion as follows.

- Left-turn(Orientation): which means whether we turn right or turn left, if it equals to one, it will turn left, otherwise it will turn right.
- Direction: which means whether we go forward or downward, if it equals to one, it will go forward, otherwise it will downward.

Figure 2: Arc-Path, turning left and forward



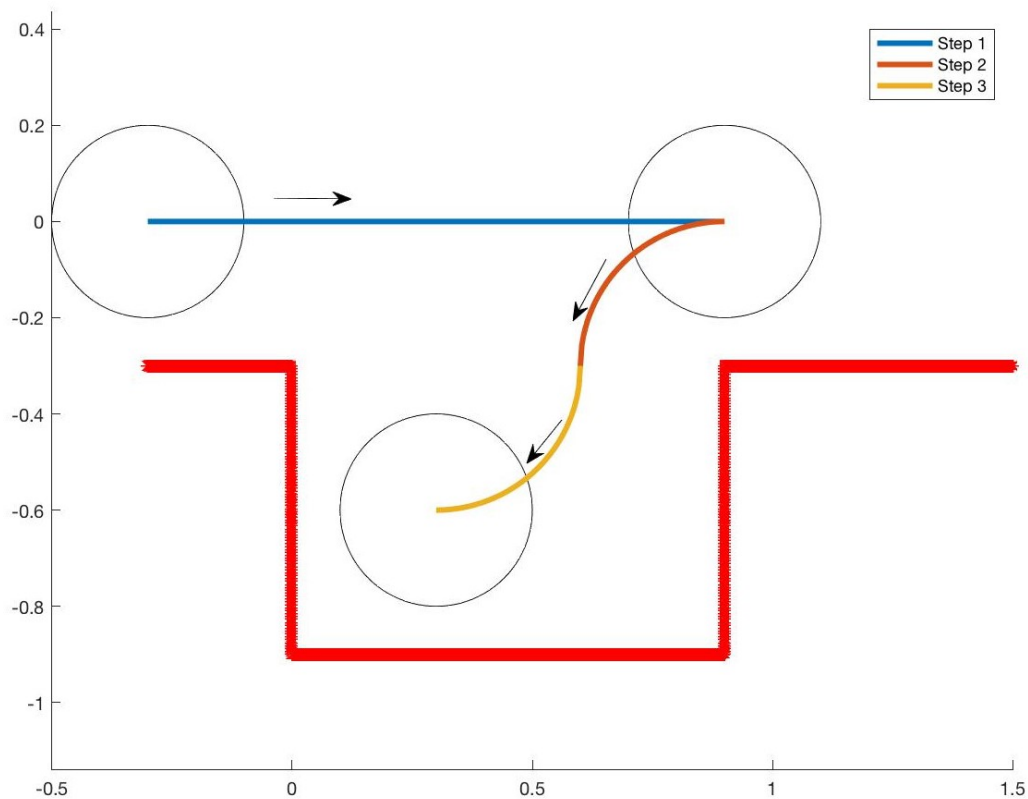
## Section 3

### Paths and Plan

#### Parallel Parking

The path of parallel parking is a combination of one linear path and two arc paths. In our case, the turtlebot firstly cover a linear path of 0.9m, then a reversing arc path with radius 0.3m, then another reversing arc path in different direction.

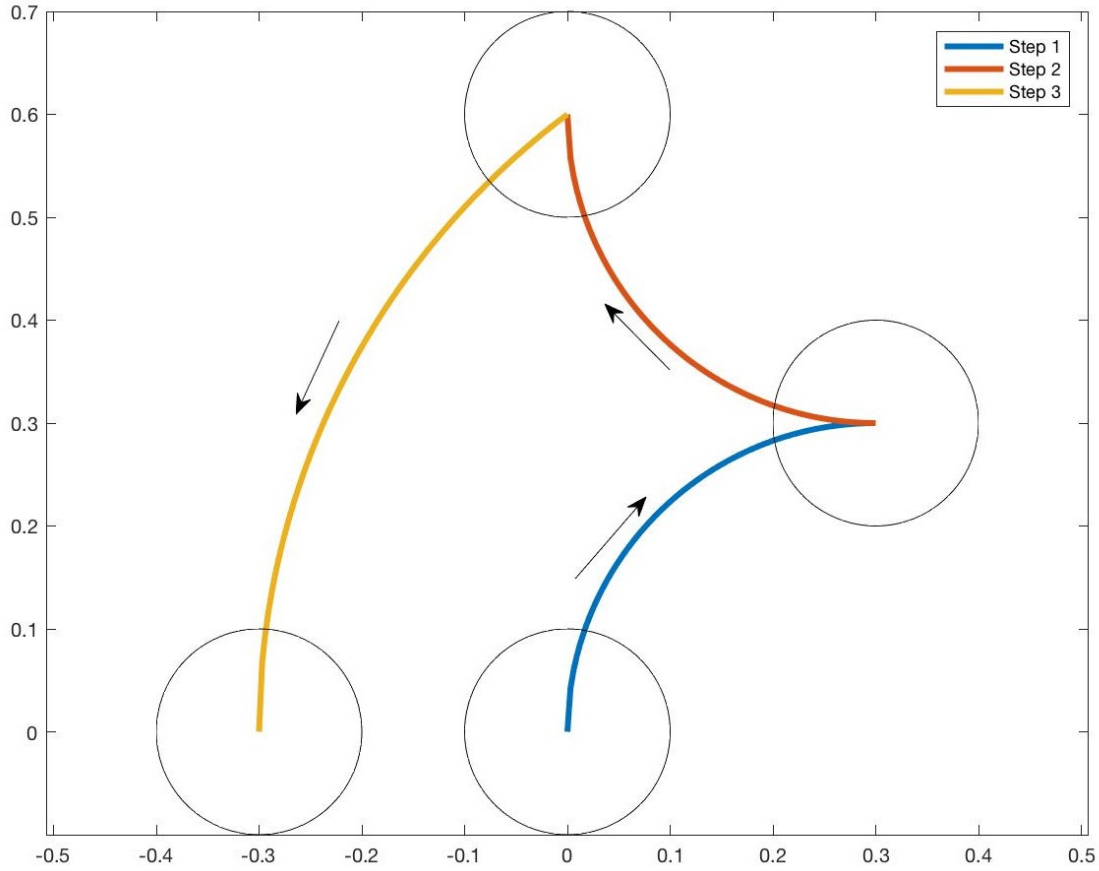
Figure 3: Parallel Parking Path



#### Three Point Turn

The path of three point turn is a combination of three arc path. Firstly, the turtlebot follow an arc path with radius 0.3m. Then a reversing arc path in a different direction, also 0.3m in radius. Lastly, an arc path with a larger radius.

Figure 4: Three Point Turn Path



### Trajectory-Based Obstacle Avoidance

In avoiding obstacle, we set the turtlebot to follow an arc path. After setting the obstacle's center  $O(x_0, y_0)$ , radius  $r$  and the total distance of moving  $d$ . Firstly, the turtlebot follows a linear path with length  $s1$ , then turn an angle  $\theta$ , then an arc path with radius  $R$ , turn back, lastly a linear path with length  $s2$ . Where :

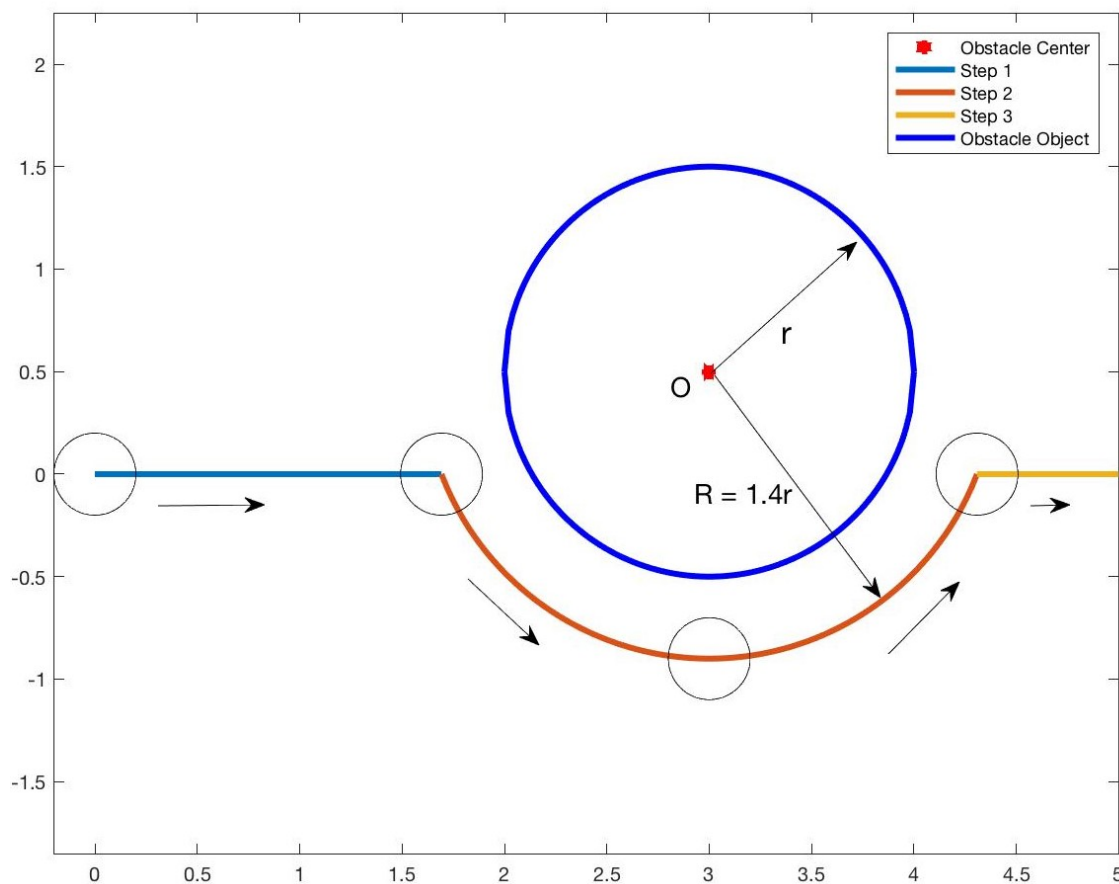
$$\theta = \arcsin\left(\frac{y_0}{r}\right)$$

$$R = 1.4 * r$$

$$s1 = x_0 - r * \cos(\theta)$$

$$s2 = d - s1 - 2 * r * \cos(\theta)$$

Figure 5: Trajectory-Based Obstacle Avoidance



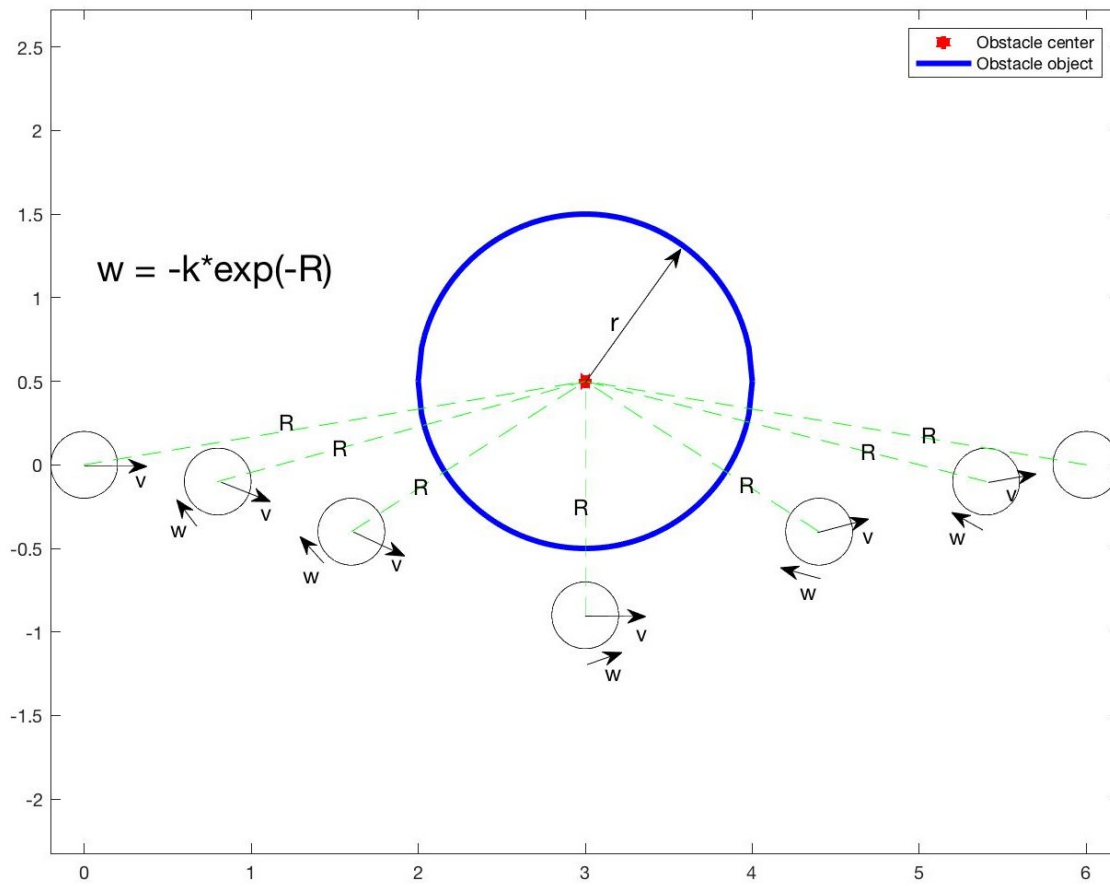
## Force-Based Obstacle Avoidance

In this path, we set a virtual force which gives the turtlebot an angular velocity  $\omega$ . Suppose  $R$  is the distance between the current position of turtlebot and the center of obstacle and  $k$  is a positive scalar. We choose:

$$\omega = -k * e^{-R}$$

Therefore, the angular velocity of turtlebot becomes bigger when approaching the obstacle, making it turn and avoid the obstacle. And by adjusting  $k$  we can make sure that the turtlebot would not hit the obstacle object. Also, we set turtlebot to turn a certain angle when it is parallel to the obstacle center point.

Figure 6: Force-Based Obstacle Avoidance





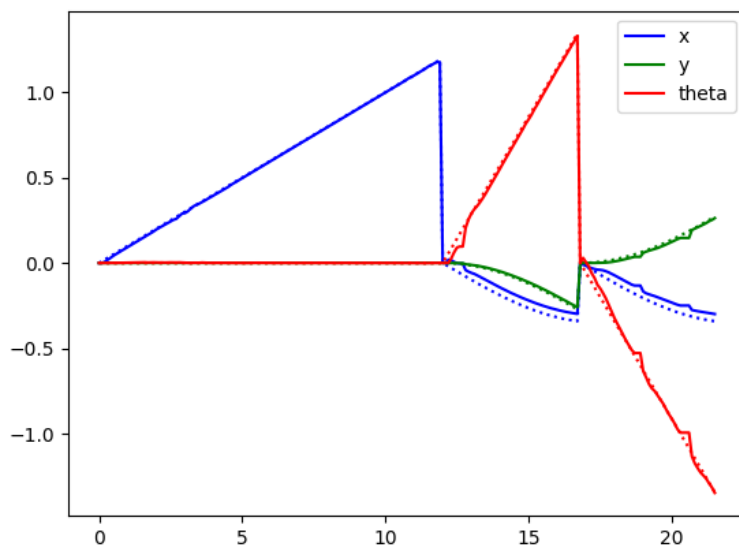
## Section 4

### Results and Theoretical Questions

#### Parallel Parking Result

In the parallel parking task, we predefined the reference position parametrized with time  $t$  and applied the trajectory tracking controller A to track the reference position. The resulting plot of current state of the turtlebot and the desired state is shown below in Figure 5. The dashed lines in the plot represent the desired reference position.

Figure 7: Parallel Parking State Plot



As we can see in the figure, basically the real states of the turtlebot follows the reference states very well, except for some small bubbling. That is probably because of the inaccuracy of the state calculation inside turtlebot itself. The turtlebot uses a wheel encoder to accumulate the length as well as the angle it has traveled. It is, however, obviously not accurate enough for state calculation, especially when we control the turtlebot to rotate a relatively small angle. The small bubbling in the figure, therefore, may be caused by the accumulation of such inaccuracies.

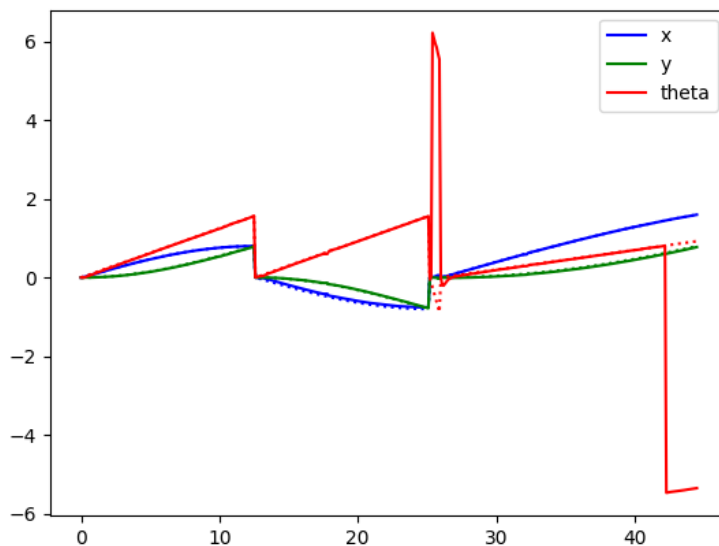
During the task, we found that the difficulty, or the little trick, was setting the direction of the velocity and angular velocity of the turtlebot. We had forward and backward movement, and left and right rotation. Together, we would four combinations. So we used two bits to encode that information and designed the time parametrized trajectory based on it. The reason we view it as a little trick is that, it is essentially not difficult but needs subtle and careful mathematic manipulations.

Finally, we successfully designed the arc path and linear path. The parallel parking path is composed of one linear path and two arc paths with angle  $\frac{\pi}{2}$ . No constraints were violated during the execution.

## Three Point Turning Result

In the three point turning task, we predefined the reference position parametrized with time  $t$  and applied the trajectory tracking controller A to track the reference position. The resulting plot of current state of the turtlebot and the desired state is shown below in Figure 6. The dashed lines in the plot represent the desired reference position.

Figure 8: Three Point Turing State Plot



As we can see in the figure, basically the real states of the turtle bot follows the reference states very well, except for the bumping at  $t=27$ . This is the plotting issue. The plot block in our code draws the current state of the turtle bot continuously, while the designed trajectory begins at zero every time the turtle bot executes a new path. When  $t=27$ , the turtle bot was given a new path, so the dashed line jumped to zero. But in real world we do not have that discrete change in state. The only thing we can see is that the controller tuned the orientation of the turtle bot quickly and soon followed the desired trajectory.

The difficulties we met in this task was similar to that we met in the previous task, and they were solved in the same way. Finally, we successfully designed the arc path and linear path. The parallel three point turning is composed of three arc paths. No constraints were violated during the execution.

## Obstacle Avoiding Result

In the obstacle avoiding task, we firstly predefined the reference position parametrized with time  $t$  and applied the trajectory tracking controller A to track the reference position as what we did before. Secondly we applied a proportional control to make the turtle bot avoid the obstacle, viewing the distance from the turtle bot to the center of the obstacle as a virtual force constraint. The resulting plot of current state of the turtle bot of method 1 and 2 are shown below in Figure 7 and 8, respectively.

Figure 9: Obstacle Avoiding State Plot

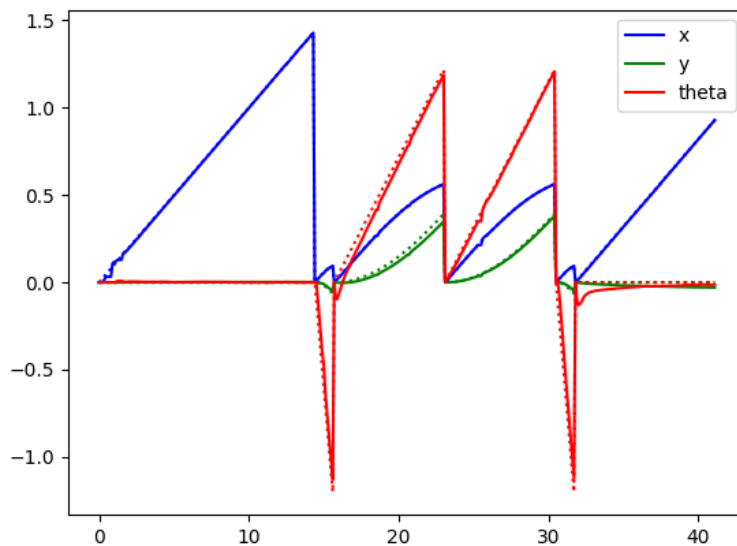
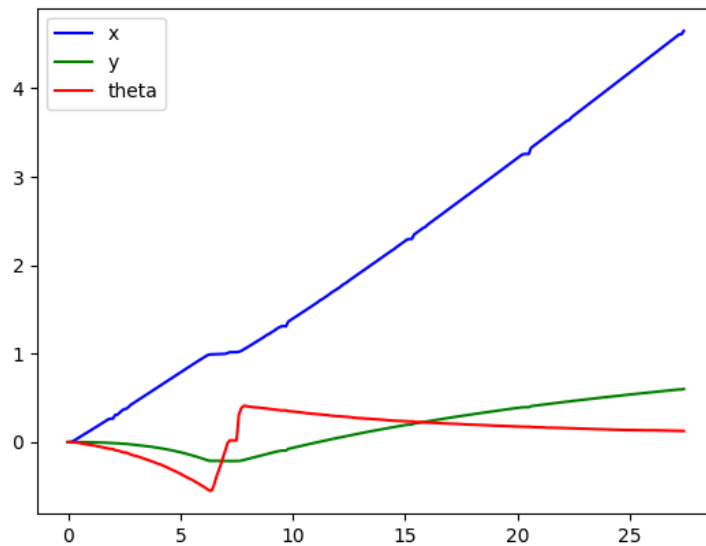


Figure 10: Obstacle Avoiding Virtual Force Constraint State Plot



As usual, the performance of trajectory tracking controller A is quite stable after we have tuned good parameters. The reasons for the bumping in figure 7 has been explained in previous sections. Here we focus on the new proportional controller taking the virtual force constraint into consideration. We notice that the curve of  $y$  coordinate roughly converge to 0 but has a bit offset, which means the turtle bot didn't quite return back to the original straight line that went across the obstacle. The downward curve of the green line in figure 8 indicates the distance the turtle bot moved in  $y$  direction in order to avoid the obstacle.

Compared to the result in figure 7, we can see that the performance of the proportional controller is not that good as the trajectory controller A. The main reason we guess is that we control the y direction movement according to the virtual force the turtle bot is feeling, but we have no guarantee that the positive and negative y direction movement are symmetric due to the non-ideal motor controls. We thus have no guarantee that the turtle bot is able to come back to the original straight line, as shown in figure 8. However, the speed of method 2 is faster than that of method 1, due to the easy execution.

In the equation 6.5 in MLS, the force is calculated by a changed version of lagrangian dynamic equation. The  $\lambda$  comes from the holonomic constraint. Here we regard the force as negatively related to the distance from the turtle bot to the center of the obstacle. To avoid the numerical issue, we use  $\exp()$  function here.

## Theoretical Questions

1.

In the previous task, we approximate the obstacles as single circles. But in practice, they are certainly not. Of course we have more efficient ways. Suppose that we can detect the contour of an obstacle. We can thus predefine the trajectory according to the contour and apply the trajectory tracking controller to follow the path. The controller is never restricted to shapes of the path. It will make the turtle bot avoid an arbitrary obstacle as long as we can write out the trajectory and tune good parameters for the controller.

2.

In the unicycle paper, the author provides controllers for both trajectory tracking and path following. The difference between these two controllers is that we define reference position in tracking and we hope the turtle bot to reach the reference position at every time point, while we design a geometric path in path following and we want the robot to converge to the geometric curve as soon as possible. For example, assume that we want the turtle bot to move from point A to point B, if we can detect the whole map that contains A and B, we probably want to use the path following controller. But in some cases we don't know the overall environment unless we are close to an obstacle. In these cases, we probably want to use trajectory tracking because it is more flexible. What's more, it is time related, we can control the speed that the turtle bot reaches the desired position.

## Section 5

### Bonus & difficulties

The lab document is great! It is clear and easy to understand. Thanks for all your efforts in it. Here is just some small advice.

1. We do not know to use row pitch yaw but use the quaternion directly, which make our rotation wrong. So the document might be better to specific this thing.
2. We set our turtlebot position according to Odom frame at first, and actually it should be related to base link frame.

## Section 6

### Code

Main file:

```

1  #!/usr/bin/env python
   import rospy
3  from geometry_msgs.msg import Twist, Vector3
   # from motion_path import *
5  from controllers import *
   from paths import *
7  from utils import *
   import tf
9  import tf.transformations as tfs

11
   cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
13

15 #####
   ##### Initialization #####
17 #####
   k = [5,3,3]
19 target_speed = 0.1
   obstacle = False
21 obstacle_center = vec(2.0, -0.0)
   obstacle_radius = 1
23
   parallel_parking_path = []
25 # Step 1 parallel parking
   direction = 1
27 s1 = 1.2
   parallel_parking_path.append(LinearPath(s1, direction, target_speed))
29 # Step 2 parallel parking
   direction = -1
31 left_turn = 1
   r2 = 0.35
33 angle2 = np.pi/2.3
   s2 = r2 * angle2
35 parallel_parking_path.append(ArcPath(r2, angle2, left_turn, target_speed, direction))
   # Step 3 parallel parking
37 direction = -1
   left_turn = -1
39 r3 = 0.35
   angle3 = np.pi/2.3
41 s3 = r3 * angle3
   parallel_parking_path.append(ArcPath(r3, angle3, left_turn, target_speed, direction))
43
   path = ChainPath(parallel_parking_path)
45
   # controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
47 # controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

49 def main():
   rospy.init_node('Lab3', anonymous=False)

51
   rospy.loginfo("To stop TurtleBot CTRL + C")
53   rospy.on_shutdown(shutdown)

55   # setting up the transform listener to find turtlebot position
   listener = tf.TransformListener()
57   from_frame = 'odom'

```

```

to_frame = 'base_link'
59 listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(5.0))
broadcaster = tf.TransformBroadcaster()

61
# this is so that each loop of the while loop takes the same amount of time. The
controller works better
63 # if you have this here
rate = rospy.Rate(10)

65
cnt = 0
67 t=0
times = []
69 actual_states = []
target_states = []
71 while(cnt<3):
    flag = parallel_parking_path[cnt].total_length
73    controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle,
obstacle_center, obstacle_radius)
    # getting the position of the
75    start_pos, start_rot = listener.lookupTransform(from_frame, to_frame, listener.
getLatestCommonTime(from_frame, to_frame))
    # 3x1 array, representing (x,y,theta) of robot starting state
77    start_state = np.array([start_pos[:2] + [GetYawFromQuat(start_rot)]]) .T
    g = np.linalg.pinv(rigid(start_state))

79
    s = 0
81    # path = parallel_parking_path
    # path = three_point_turn_path
83    # path = compute_obstacle_avoid_path(4.0, vec(2.0, -0.0), 0.5)
    while not rospy.is_shutdown() and s <= flag:
85        current_pos, current_rot = listener.lookupTransform(from_frame, to_frame,
listener.getLatestCommonTime(from_frame, to_frame))
        # 3x1 array, representing (x,y,theta) of current robot state
87        current_state_odom = np.array([current_pos[:2] + [1]]) .T
        # 3x1 array representing (x,y,theta) of current robot state, relative to
starting state. look at rigid method in utils.py
89        current_state = g.dot(current_state_odom)
        current_state[2][0] = GetYawFromQuat(current_rot) - start_state[2][0]

91

93    # for the plot at the end
    target_state = parallel_parking_path[cnt].target_state(s)
95    times.append(t * 10)
    actual_states.append(current_state)
97    target_states.append(target_state)

99    # I may have forgotten some parameters here
    move_cmd = controller.step_path(current_state, s)
101
    cmd_vel.publish(move_cmd)
103

    # I believe this should be the same as the ros rate time, so if you change that,
change it here too
105    s += target_speed / 10
    t += target_speed / 10
107    # this is governing how much each loop should run for. look up ROS rates if you
're interested
    rate.sleep()
109    cnt+=1

111 # Plots
times = np.array(times)

```

```

113     actual_states = np.array(actual_states)
114     target_states = np.array(target_states)
115
116
117     plt.figure()
118     colors = [ 'blue', 'green', 'red' ]
119     labels = [ 'x', 'y', 'theta' ]
120     for i in range(3):
121         plt.plot(times, actual_states[:,i], color=colors[i], ls='solid', label=labels[i])
122         plt.plot(times, target_states[:,i], color=colors[i], ls='dotted')
123     plt.legend()
124     plt.show()
125
126
127 def shutdown():
128     rospy.loginfo("Stopping TurtleBot")
129     cmd_vel.publish(Twist())
130     rospy.sleep(1)
131
132
133 if __name__ == '__main__':
134     main()
135     # try:
136     #     main()
137     # except e:
138     #     print e
139     #     rospy.loginfo("Lab3 node terminated.")

```

./code/main.py

controllers file:

```

1 # imports may be necessary
2 import numpy as np
3 import rospy
4 from geometry_msgs.msg import Twist, Vector3
5
6
7 class Controller():
8     def __init__(self, path, k, target_speed, obstacle, obstacle_center, obstacle_radius):
9         self.path = path
10        self.target_speed = target_speed
11        self.obstacle = obstacle
12        self.obstacle_center = obstacle_center
13        self.obstacle_radius = obstacle_radius
14        self.k = k
15        self.direction = path.direction
16
17    def step_path(self, current_state, s):
18        """
19        Takes the current state and final state and returns the twist command to reach the
20        target state
21        according to the path variable
22
23        Parameters
24        -----
25        current_state: :obj:'numpy.ndarray'
26            x , y , theta_z
27            twist representing the current state of the turtlebot. see utils.py
28        s: float
29            the path length the turtlebot should have travelled so far
30
31        Returns
32        """

```



```

31 :obj: 'geometry_msgs.msg.Twist'
32     Twist message to be sent to the turtlebot
33
34 """
35
36 # YOUR CODE HERE
37 if not self.obstacle:
38     vr = np.linalg.norm(self.path.target_velocity(s)[0:2]) # *np.sign(self.path.
target_velocity(s)[0])
39     u1 = - self.k[0]*(self.path.target_state(s)[0] - current_state[0][0])
    u2 = self.k[1]*vr*np.sinc(self.path.target_state(s)[2]-current_state[2][0])*(self.path
.target_state(s)[1]-current_state[1][0]) - self.k[2]*(self.path.target_state(s)[2]-
current_state[2][0])
41     C = np.array([ [np.cos(self.path.target_state(s)[2]-current_state[2][0]),0],[0,1] ])
    vdwd = C.dot(np.array([[vr],[self.path.target_velocity(s)[2]]]) - np.array([[u1],[u2
]])
43     vd = vdwd[0][0]
    wd = vdwd[1][0]
44
45     twist = Twist()
46     twist.linear.x = self.direction*abs(vd)
47     twist.linear.y = 0
48     twist.linear.z = 0
49     twist.angular.x = 0
50     twist.angular.y = 0
51     twist.angular.z = wd
52     return twist
53
54 # TURN = False
55
56 class ForceController(Controller):
57
58     def __init__(self, k, target_speed, obstacle, obstacle_center, obstacle_radius):
59
60         self.obstacle = obstacle
61         self.obstacle_center = obstacle_center
62         self.obstacle_radius = obstacle_radius
63         self.k = k
64         self.target_speed = target_speed
65         self.pre_alpha = 0
66
67     def step_path(self, current_state, s):
68         Ex = np.array([1],[0])
69         Ey = np.array([0],[1])
70         vx = self.target_speed*Ex
71         x = current_state[0][0]
72         y = current_state[1][0]
73         x0 = self.obstacle_center[0]
74         y0 = self.obstacle_center[1]
75         cur_angle = current_state[2][0]
76         r = np.array([y0 - y, x0 - x])
77         # maybe change
78         # r = np.array([1/(y0 - y), 1/(x0 - x)])
79         #####
80         theta = np.arctan((y0 - y)/(x0 - x))
81         if theta>0:
82             theta = theta
83         else:
84             theta = np.pi+theta
85         vth = self.k*np.linalg.norm(r)*(np.cos(theta)*Ex + np.sin(theta)*Ey)
86         vth = -self.k*np.linalg.norm(r)*(-np.sin(theta)*Ex + np.cos(theta)*Ey)
87         vthx = vth*np.cos(theta)

```

```

89     vinp = np.sqrt(np.linalg.norm(vthx)**2 + np.linalg.norm(vx)**2)
    alpha = np.arccos(np.transpose(vx + vth).dot(vinp)/np.linalg.norm(vx + vth)/np.linalg.
    norm(vinp))
91     # print('alpha', alpha)
    alpha = alpha[0][0]
93
94     # if self.pre_alpha == 0:
95     #     self.pre_alpha, dalpha = alpha, 0
96     # else:
97     #     self.pre_alpha, dalpha = alpha, alpha - self.pre_alpha
98
99     # if cur_angle < -np.pi/4 or cur_angle > np.pi/4:
100     #     dalpha = 0
101     #     dalpha *= 105
102     print(alpha)
103
104     error = np.exp(-abs(r[1])) # + np.exp(-abs(r[1]))*(x>x0)
105     if abs(x0-x) < 0.02 and abs(alpha) > 0.003:
106         dalpha = 1
107         x = 0
108     else:
109         dalpha = -self.k*error
110         x = 0.2
111     # else:
112     #     dalpha = self.k*error
113     #     x = 0.2
114
115     twist = Twist()
116     twist.linear.x = x
117     twist.linear.y = 0
118     twist.linear.z = 0
119     twist.angular.x = 0
120     twist.angular.y = 0
121     twist.angular.z = dalpha
122
123     # twist.linear.x = 0
124     # twist.angular.z = 1
125     # print('alpha', alpha)
126     print("twist", twist)
127     return twist

```

./code/controllers.py

paths function file:

```

1  import numpy as np
   import math
3  from math import sin, cos, asin, acos, atan2, sqrt
   from utils import *
5  from matplotlib import pyplot as plt
6
7  class MotionPath:
8      def target_state(self, s):
9          """
10             Target position of turtlebot given the path length s
11
12             Parameters
13             -----
14             s: float
15                 the path length the turtlebot should have travelled so far
16
17             Returns
18             -----

```

```

19         :obj: 'numpy.ndarray'
20             target position of turtlebot
21         """
22         raise NotImplementedError()
23
24     def target_velocity(self, s):
25         """
26         Target velocity of turtlebot given the path length s
27
28         Parameters
29         -----
30         s: float
31             the path length the turtlebot should have travelled so far
32
33         Returns
34         -----
35         :obj: 'numpy.ndarray'
36             target velocity of turtlebot
37         """
38         raise NotImplementedError()
39
40     @property
41     def total_length(self):
42         """ total path length
43         Returns
44         -----
45         float
46             total path length
47         """
48         raise NotImplementedError()
49
50     @property
51     def end_state(self):
52         """ Final state after completing the path
53         Returns
54         -----
55         :obj: 'numpy.ndarray'
56             Final state after completing the path
57         """
58         return self.target_state(self.total_length)
59
60 class ArcPath(MotionPath):
61     def __init__(self, radius, angle, left_turn, speed, direction): #, , left_turn):
62         """
63         Parameters
64         -----
65         radius: float
66             how big of a circle in meters
67         angle: float
68             how much of the circle do you want to complete (in radians).
69             Can be positive or negative
70         left_turn: bool
71             whether the turtlebot should turn left or right
72         """
73         self.radius = radius
74         self.angle = angle
75         self.total_length = radius*angle
76         self.direction = direction
77         self.speed = speed
78         self.left_turn = left_turn

```

```

81     def target_state(self, s):
82         """
83         Target position of turtlebot given the current path length s for Circular Arc Path
84
85         Parameters
86         -----
87         s: float
88             the path length the turtlebot should have travelled so far
89
90         Returns
91         -----
92         :obj: 'numpy.ndarray'
93             target position of turtlebot
94         """
95         # YOUR CODE HERE
96         r = self.radius
97         theta = self.left_turn*s/r
98         if self.left_turn>0:
99             x = self.direction*r*sin(theta)
100             y = self.direction*(r - r*cos(theta))
101         else:
102             x = -self.direction*r*sin(theta)
103             y = self.direction*(r*cos(theta) - r)
104
105         return np.array([x,y,theta])
106
107     def target_velocity(self, s):
108         """
109         Target velocity of turtlebot given the current path length s for Circular Arc Path
110
111         Parameters
112         -----
113         s: float
114             the path length the turtlebot should have travelled so far
115
116         Returns
117         -----
118         :obj: 'numpy.ndarray'
119             target velocity of turtlebot
120         """
121         # YOUR CODE HERE
122         r = self.radius
123         theta = self.left_turn*s/r
124         if self.left_turn>0:
125             vx = self.speed*cos(theta)
126             vy = self.speed*sin(theta)
127             w = self.left_turn*self.speed/r
128         else:
129             vx = self.speed*cos(theta)
130             vy = -self.speed*sin(theta)
131             w = self.left_turn*self.speed/r
132
133         return np.array([vx,vy,w])
134
135     @property
136     def total_length(self):
137         """ total length of the path
138
139         Returns
140         -----
141         float
142             total length of the path
143         """
144         # YOUR CODE HERE

```

```

143         return self.angle * self.radius
class LinearPath(MotionPath):
145     def __init__(self, length, direction, speed):
146         """
147         Parameters
148         -----
149         length: float
150             length of the path
151         """
152         self.total_length = length
153         self.direction = direction
154         self.speed = speed
155     def target_state(self, s):
156         """
157         Target position of turtlebot given the current path length s for Linear Path
158
159         Parameters
160         -----
161         s: float
162             the path length the turtlebot should have travelled so far
163
164         Returns
165         -----
166         :obj: 'numpy.ndarray'
167             target position of turtlebot
168             [x,y,theta_z]
169         """
170         # YOUR CODE HERE
171         return np.array([self.direction*s,0,0])
172
173     def target_velocity(self, s):
174         """
175         Target velocity of turtlebot given the current path length s for Linear Path
176
177         Parameters
178         -----
179         s: float
180             the path length the turtlebot should have travelled so far
181
182         Returns
183         -----
184         :obj: 'numpy.ndarray'
185             target velocity of turtlebot
186         """
187         # YOUR CODE HERE
188         return np.array([self.speed,0,0])
189
190     @property
191     def total_length(self):
192         """ total length of the path
193
194         Returns
195         -----
196         float
197             total length of the path
198         """
199         # YOUR CODE HERE
200         return self.total_length
201 class ChainPath(MotionPath):
202     def __init__(self, subpaths):
203         """
204         Parameters

```

```

205         subpaths: :obj:'list ' of :obj:'MotionPath'
207         list of paths which should be chained together
208     """
209     self.subpaths = subpaths
210     self.direction = self.subpaths[0].direction
211     self.left_turn = self.subpaths[1].left_turn
212
213 def target_state(self, s):
214     """
215     Target position of turtlebot given the current path length s for Chained Path
216
217     Parameters
218     -----
219     s: float
220         the path length the turtlebot should have travelled so far
221
222     Returns
223     -----
224     :obj:'numpy.ndarray'
225         target position of turtlebot
226     """
227     # YOUR CODE HERE
228     if( s <= self.subpaths[0].total_length ):
229         return self.subpaths[0].target_state(s)
230     elif( s <= self.subpaths[0].total_length+self.subpaths[1].total_length ):
231         # self.direction = self.subpaths[1].direction
232         # self.left_turn = self.subpaths[1].left_turn
233         return self.subpaths[1].target_state(s-self.subpaths[0].total_length)
234     else:
235         # self.direction = self.subpaths[2].direction
236         # self.left_turn = self.subpaths[2].left_turn
237         return self.subpaths[2].target_state(s-self.subpaths[0].total_length-self.
238 subpaths[1].total_length)
239
240 def target_velocity(self, s):
241     """
242     Target velocity of turtlebot given the current path length s for Chained Path
243
244     Parameters
245     -----
246     s: float
247         the path length the turtlebot should have travelled so far
248
249     Returns
250     -----
251     :obj:'numpy.ndarray'
252         target velocity of turtlebot
253     """
254     # YOUR CODE HERE
255     if( s <= self.subpaths[0].total_length ):
256         return self.subpaths[0].target_velocity(s)
257     elif( s <= self.subpaths[0].total_length+self.subpaths[1].total_length ):
258         # self.direction = self.subpaths[1].direction
259         # self.left_turn = self.subpaths[1].left_turn
260         return self.subpaths[1].target_velocity(s-self.subpaths[0].total_length)
261     else:
262         # self.direction = self.subpaths[2].direction
263         # self.left_turn = self.subpaths[2].left_turn
264         return self.subpaths[2].target_velocity(s-self.subpaths[0].total_length-self.
265 subpaths[1].total_length)

```

```

265     @property
266     def total_length(self):
267         """ total length of the path
268         Returns
269         -----
270         float
271             total length of the path
272         """
273         # YOUR CODE HERE
274         return self.subpaths[0].total_length+self.subpaths[1].total_length+self.subpaths[2].
total_length
275
276 def compute_obstacle_avoid_path(dist, obs_center, obs_radius):
277     # YOUR CODE HERE
278     target_speed = 0.1
279     direction = obs_center[1]/abs(obs_center[1])
280     R = 1.2*obs_radius
281     x_p = np.sqrt(R**2 - abs(obs_center[1])**2)
282     theta = np.arccos(abs(obs_center[1])/R)
283
284     obstacle_avoid_path = []
285     # Step 1 obstacle_avoid_path
286     s1 = obs_center[0] - x_p
287     obstacle_avoid_path.append(LinearPath(s1,1,target_speed))
288     # Step 2 obstacle_avoid_path
289     left_turn = -direction
290     r2 = 0.1
291     angle2 = theta
292     s2 = r2 * angle2
293     obstacle_avoid_path.append(ArcPath(r2, angle2, left_turn, target_speed, 1))
294     # Step 3 obstacle_avoid_path
295     left_turn = direction
296     r3 = R
297     angle3 = theta
298     s3 = r3 * angle3
299     obstacle_avoid_path.append(ArcPath(r3, angle3, left_turn, target_speed, 1))
300     # Step 4 obstacle_avoid_path
301     left_turn = direction
302     r4 = R
303     angle4 = theta
304     s4 = r4 * angle4
305     obstacle_avoid_path.append(ArcPath(r4, angle4, left_turn, target_speed, 1))
306     # Step 5 obstacle_avoid_path
307     left_turn = -direction
308     r5 = 0.1
309     angle5 = theta
310     s5 = r5 * angle5
311     obstacle_avoid_path.append(ArcPath(r5, angle5, left_turn, target_speed, 1))
312     # Step 6 obstacle_avoid_path
313     s6 = dist - 2*x_p - s1
314     obstacle_avoid_path.append(LinearPath(s6,1,target_speed))
315     return obstacle_avoid_path
316
317 def plot_path(path):
318     """
319     Plots on a 2D plane, the top down view of the path passed in
320
321     Parameters
322     -----
323     path: :obj: 'MotionPath'
324         Path to plot
325 
```

```

327     s = np.linspace(0, path.total_length, 1000, endpoint=False)
329     twists = np.array(list(path.target_state(si) for si in s))

331     plt.plot(twists[:,0], twists[:,1])
331     plt.show()

333 # YOUR CODE HERE
335 # parallel_parking_path = ChainPath([])

337 # YOUR CODE HERE
339 # three_point_turn_path = ChainPath([])

339 if __name__ == '__main__':
341     path = three_point_turn_path
341     # path = compute_obstacle_avoid_path()
341     print(path.end_state)
343     plot_path(path)

```

./code/paths.py

obs avoid function file:

```

1  #!/usr/bin/env python
   import rospy
3  from geometry_msgs.msg import Twist, Vector3
   # from motion_path import *
5  from controllers import *
   from paths import *
7  from utils import *
   import tf
9  import tf.transformations as tfs

11

13  cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)

15  #####
17  ##### Initialization #####
19  k = [5,3,3]
21  target_speed = 0.1
23  obstacle = False
25  obstacle_center = [2,-0.2]
27  obstacle_radius = 0.5
29  dist = 3.5
31  parallel_parking_path = compute_obstacle_avoid_path(dist, obstacle_center, obstacle_radius)
33  # controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
35  # controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

37  def main():
39      rospy.init_node('Lab3', anonymous=False)

39      rospy.loginfo("To stop TurtleBot CTRL + C")
39      rospy.on_shutdown(shutdown)

39      # setting up the transform listener to find turtlebot position
39      listener = tf.TransformListener()
39      from_frame = 'odom'
39      to_frame = 'base_link'
39      listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(5.0))
39      broadcaster = tf.TransformBroadcaster()

```



```

41 # this is so that each loop of the while loop takes the same amount of time. The
    controller works better
43 # if you have this here
    rate = rospy.Rate(10)

45 cnt = 0
    t=0
47 times = []
    actual_states = []
49 target_states = []
    while(cnt<6):
51         flag = parallel_parking_path[cnt].total_length
            controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle,
            obstacle_center, obstacle.radius)
53         # getting the position of the
            start_pos, start_rot = listener.lookupTransform(from_frame, to_frame, listener.
            getLatestCommonTime(from_frame, to_frame))
55         # 3x1 array, representing (x,y,theta) of robot starting state
            start_state = np.array([start_pos[:2] + [GetYawFromQuat(start_rot)]]).T
57         g = np.linalg.pinv(rigid(start_state))

59         s = 0
            # path = parallel_parking_path
61         # path = three_point_turn_path

63         while not rospy.is_shutdown() and s <= flag:
            current_pos, current_rot = listener.lookupTransform(from_frame, to_frame,
            listener.getLatestCommonTime(from_frame, to_frame))
65             # 3x1 array, representing (x,y,theta) of current robot state
                current_state_odom = np.array([current_pos[:2] + [1]]).T
67             # 3x1 array representing (x,y,theta) of current robot state, relative to
            starting state. look at rigid method in utils.py
                current_state = g.dot(current_state_odom)
69                 current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]

71
73             # for the plot at the end
                target_state = parallel_parking_path[cnt].target_state(s)
                times.append(t * 10)
75                 actual_states.append(current_state)
                target_states.append(target_state)

77
79             # I may have forgotten some parameters here
                move_cmd = controller.step_path(current_state, s)

81                 cmd_vel.publish(move_cmd)

83
85             # I believe this should be the same as the ros rate time, so if you change that,
            change it here too
                s += target_speed / 10
                t += target_speed / 10
            # this is governing how much each loop should run for. look up ROS rates if you
            're interested
87                 rate.sleep()
                cnt+=1

89
91 # Plots
    times = np.array(times)
    actual_states = np.array(actual_states)
93     target_states = np.array(target_states)

```

```

97     plt.figure()
    colors = ['blue', 'green', 'red']
99     labels = ['x', 'y', 'theta']
    for i in range(3):
101         plt.plot(times, actual_states[:,i], color=colors[i], ls='solid', label=labels[i])
        plt.plot(times, target_states[:,i], color=colors[i], ls='dotted')
103     plt.legend()
    plt.show()
105
107 def shutdown():
    rospy.loginfo("Stopping TurtleBot")
109     cmd_vel.publish(Twist())
    rospy.sleep(1)
111
112 if __name__ == '__main__':
113     main()
    # try:
115     #     main()
    # except e:
117     #     print e
    #     rospy.loginfo("Lab3 node terminated.")

```

./code/obs\_avoid.py

obs\_avoid.force function file:

```

#!/usr/bin/env python
2 import rospy
from geometry_msgs.msg import Twist, Vector3
4 # from motion_path import *
from controllers import *
6 from paths import *
from utils import *
8 import tf
import tf.transformations as tfs
10
12 cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
14
15 #####
16 ##### Initialization #####
17 #####
18 k = 0.2
target_speed = 0.1
20 obstacle = False
obstacle_center = [1,0.1]
22 obstacle_radius = 0.2
dist = 3.5
24
26 def main():
    rospy.init_node('Lab3', anonymous=False)
28
    rospy.loginfo("To stop TurtleBot CTRL + C")
30    rospy.on_shutdown(shutdown)
32
    # setting up the transform listener to find turtlebot position
    listener = tf.TransformListener()
34    from_frame = 'odom'
    to_frame = 'base_link'

```

```

36 listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(5.0))
   broadcaster = tf.TransformBroadcaster()
38
   # this is so that each loop of the while loop takes the same amount of time. The
   controller works better
40 # if you have this here
   rate = rospy.Rate(10)
42
   cnt = 0
44   t=0
   times = []
46   actual_states = []
   target_states = []
48   controller = ForceController(k, target_speed, obstacle, obstacle_center, obstacle_radius)
   # getting the position of the
50   start_pos, start_rot = listener.lookupTransform(from_frame, to_frame, listener.
   getLatestCommonTime(from_frame, to_frame))
   # 3x1 array, representing (x,y,theta) of robot starting state
52   start_state = np.array([start_pos[:2] + [GetyawFromQuat(start_rot)]]) .T
   g = np.linalg.pinv(rigid(start_state))
54
   s = 0
56   # path = parallel-parking-path
   # path = three-point-turn-path
58
   while not rospy.is_shutdown():
60       current_pos, current_rot = listener.lookupTransform(from_frame, to_frame, listener.
   getLatestCommonTime(from_frame, to_frame))
       # 3x1 array, representing (x,y,theta) of current robot state
62       current_state_odom = np.array([current_pos[:2] + [1]]) .T
       # 3x1 array representing (x,y,theta) of current robot state, relative to starting
       state. look at rigid method in utils.py
64       current_state = g.dot(current_state_odom)
       current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]
66
68       # for the plot at the end
       # target_state = parallel-parking-path[cnt].target_state(s)
70       times.append(t * 10)
       actual_states.append(current_state)
       # target_states.append(target_state)
72
74       # I may have forgotten some parameters here
       move_cmd = controller.step_path(current_state, s)
76
       cmd_vel.publish(move_cmd)
78
       # I believe this should be the same as the ros rate time, so if you change that,
       change it here too
80       s += target_speed / 10
       t += target_speed / 10
82       # this is governing how much each loop should run for. look up ROS rates if you're
       interested
       rate.sleep()
84
   # Plots
86   times = np.array(times)
   actual_states = np.array(actual_states)
88   # target_states = np.array(target_states)
90

```

```

92     plt.figure()
    colors = [ 'blue', 'green', 'red' ]
94     labels = [ 'x', 'y', 'theta' ]
    for i in range(3):
96         plt.plot(times, actual_states[:,i], color=colors[i], ls='solid', label=labels[i])
        # plt.plot(times, target_states[:,i], color=colors[i], ls='dotted')
98     plt.legend()
    plt.show()
100
102 def shutdown():
    rospy.loginfo("Stopping TurtleBot")
104     cmd_vel.publish(Twist())
    rospy.sleep(1)
106
107 if __name__ == '__main__':
108     main()
    # try:
109     #     main()
110     # except e:
111     #     print e
112     #     rospy.loginfo("Lab3 node terminated.")

```

./code/obs\_avoid\_force.py

three\_point function file:

```

1  #!/usr/bin/env python
    import rospy
3  from geometry_msgs.msg import Twist, Vector3
    # from motion_path import *
5  from controllers import *
    from paths import *
7  from utils import *
    import tf
9  import tf.transformations as tfs
11
12 cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
13
14 #####
15 ##### Initialization #####
16 #####
17 k = [5,3,3]
18 target_speed = 0.1
19 obstacle = False
21 obstacle_center = vec(2.0, -0.0)
    obstacle_radius = 1
23
24 parallel_parking_path = []
25 # Step 1 parallel parking
    direction = 1
27 left_turn = 1
    r1 = 0.8
29 angle1 = np.pi/2
    s1 = r1 * angle1
31 parallel_parking_path.append(ArcPath(r1, angle1, left_turn, target_speed, direction))
    # Step 2 parallel parking
33 direction = -1
    left_turn = 1
35 r2 = 0.8

```

```

angle2 = np.pi/2
37 s2 = r2 * angle2
parallel_parking_path.append(ArcPath(r2, angle2, left_turn, target_speed, direction))
39 # Step 3 parallel parking
direction = 1
41 left_turn = -1
r3 = 0.08
43 angle3 = np.arccos(3.0/5.0)
s3 = r3 * angle3
45 parallel_parking_path.append(ArcPath(r3, angle3, left_turn, target_speed, direction))
# Step 4 parallel parking
47 direction = 1
left_turn = 1
49 r4 = 5*r1/2
angle4 = np.arccos(3.0/5.0)
51 s4 = r4 * angle3
parallel_parking_path.append(ArcPath(r4, angle4, left_turn, target_speed, direction))
53
path = ChainPath(parallel_parking_path)
55
# controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
57 # controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

59 def main():
    rospy.init_node('Lab3', anonymous=False)

61
    rospy.loginfo("To stop TurtleBot CTRL + C")
63    rospy.on_shutdown(shutdown)

65    # setting up the transform listener to find turtlebot position
    listener = tf.TransformListener()
67    from_frame = 'odom'
    to_frame = 'base_link'
69    listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(5.0))
    broadcaster = tf.TransformBroadcaster()

71
    # this is so that each loop of the while loop takes the same amount of time. The
    controller works better
73    # if you have this here
    rate = rospy.Rate(10)

75
    cnt = 0
77    t=0
    times = []
79    actual_states = []
    target_states = []
81    while(cnt<4):
        flag = parallel_parking_path[cnt].total_length
83        controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle,
        obstacle_center, obstacle_radius)
        # getting the position of the
85        start_pos, start_rot = listener.lookupTransform(from_frame, to_frame, listener.
        getLatestCommonTime(from_frame, to_frame))
        # 3x1 array, representing (x,y,theta) of robot starting state
87        start_state = np.array([start_pos[:2] + [GetYawFromQuat(start_rot)]])
        g = np.linalg.pinv(rigid(start_state))

89
        s = 0
91        # path = parallel_parking_path
        # path = three_point_turn_path
93        # path = compute_obstacle_avoid_path(4.0, vec(2.0, -0.0), 0.5)
        while not rospy.is_shutdown() and s <= flag:

```

```

95         current_pos, current_rot = listener.lookupTransform(from_frame, to_frame,
listener.getLatestCommonTime(from_frame, to_frame))
# 3x1 array, representing (x,y,theta) of current robot state
97         current_state_odom = np.array([current_pos[:2] + [1]]).T
# 3x1 array representing (x,y,theta) of current robot state, relative to
starting state. look at rigid method in utils.py
99         current_state = g.dot(current_state_odom)
current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]
101
103         # for the plot at the end
target_state = parallel_parking_path[cnt].target_state(s)
105         times.append(t * 10)
actual_states.append(current_state)
107         target_states.append(target_state)
109
# I may have forgotten some parameters here
move_cmd = controller.step_path(current_state, s)
111
cmd_vel.publish(move_cmd)
113
# I believe this should be the same as the ros rate time, so if you change that,
change it here too
115         s += target_speed / 10
t += target_speed / 10
117         # this is governing how much each loop should run for. look up ROS rates if you
're interested
rate.sleep()
119         cnt+=1
121
# Plots
times = np.array(times)
123         actual_states = np.array(actual_states)
target_states = np.array(target_states)
125
127
plt.figure()
129         colors = ['blue', 'green', 'red']
labels = ['x', 'y', 'theta']
131         for i in range(3):
plt.plot(times, actual_states[:,i], color=colors[i], ls='solid', label=labels[i])
133         plt.plot(times, target_states[:,i], color=colors[i], ls='dotted')
plt.legend()
135         plt.show()
137
def shutdown():
139         rospy.loginfo("Stopping TurtleBot")
cmd_vel.publish(Twist())
141         rospy.sleep(1)
143
if __name__ == '__main__':
main()
145     # try:
# main()
147     # except e:
# print e
149     # rospy.loginfo("Lab3 node terminated.")

```

./code/three-point.py