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

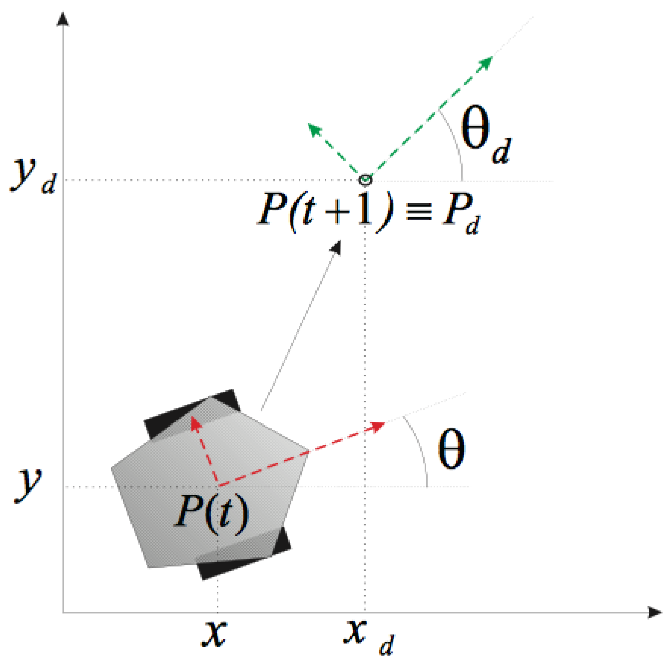
Section 2
Methods

1. Controller
   Consider a virtual unicycle type robot governed by

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

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

Section 2 continued on next page...

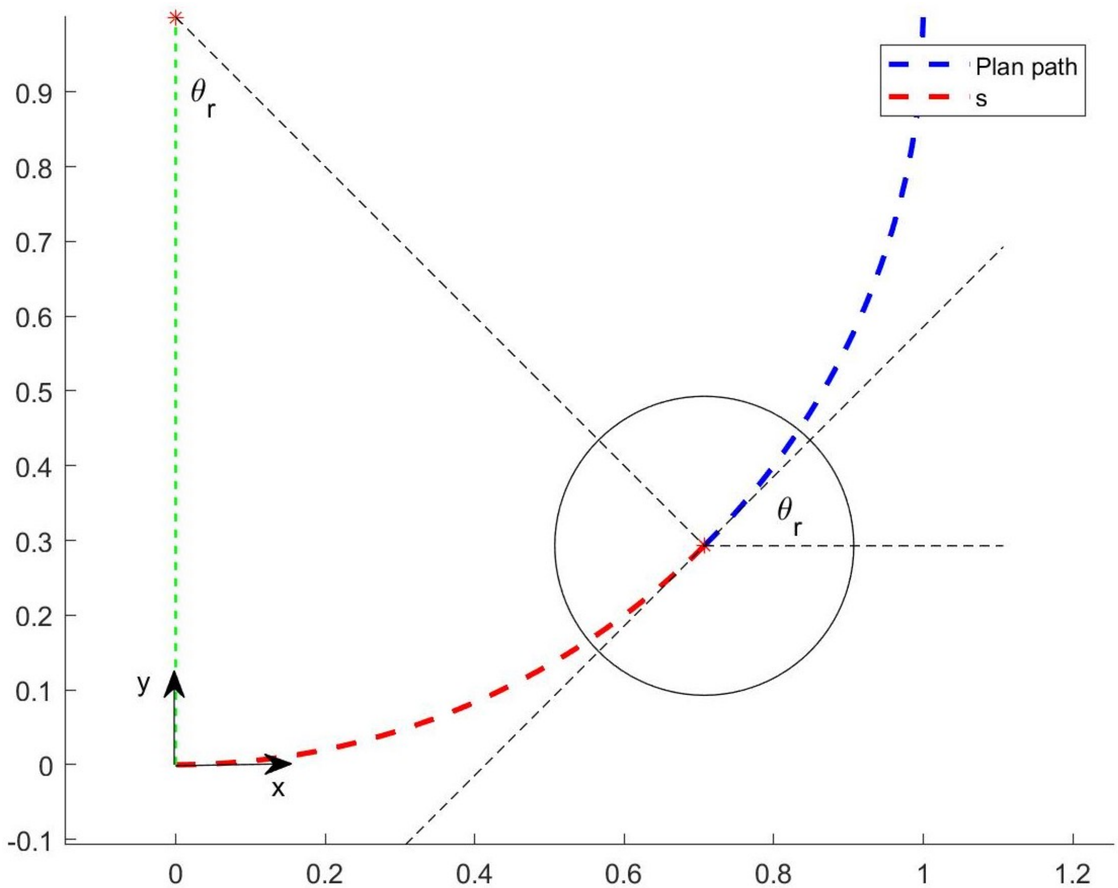
path	linear	arc
θ_r	0	$\frac{s}{r}$
x_r	s	$r\sin(\theta)$
y_r	0	$r - r\cos(\theta)$

s is the total distance that the turtlebot has been traveled.
r is the radius of the arc circle.
θ, 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.

(a) 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.

(b) 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



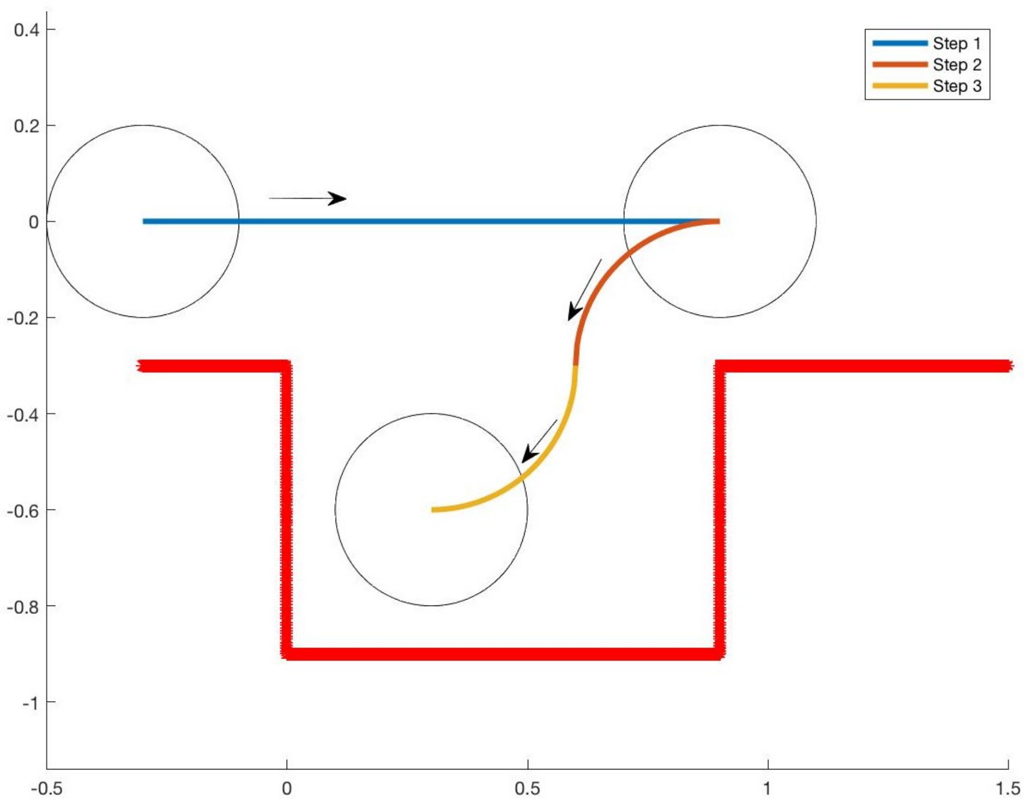
Section 3

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

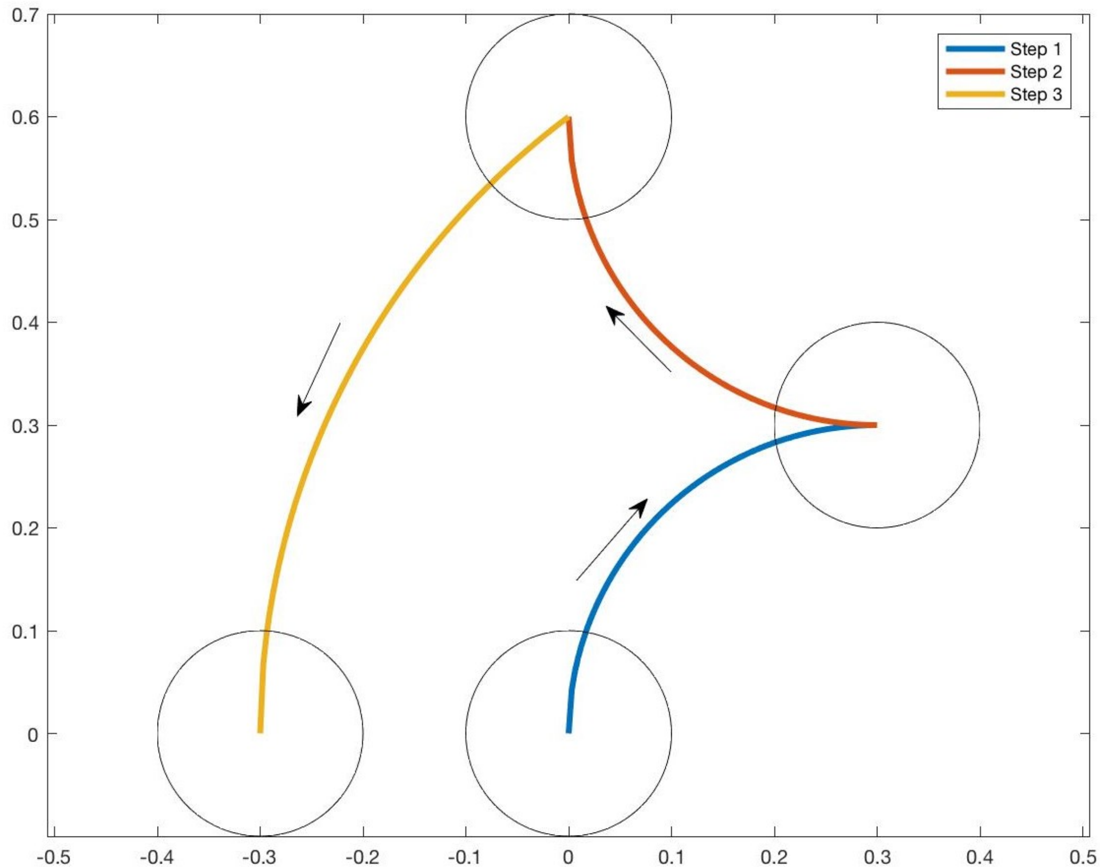


Three Point Turn

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.

Section 3 continued on next page. . .

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 $\boldsymbol{O}(x_0,y_0)$, radius $\boldsymbol{r}$ and the total distance of moving $\boldsymbol{d}$. Firstly, the turtlebot follows a linear path with length $\boldsymbol{s1}$, then turn an angle θ, then an arc path with radius $\boldsymbol{R}$, turn back, lastly a linear path with length $\boldsymbol{s2}$. Where :

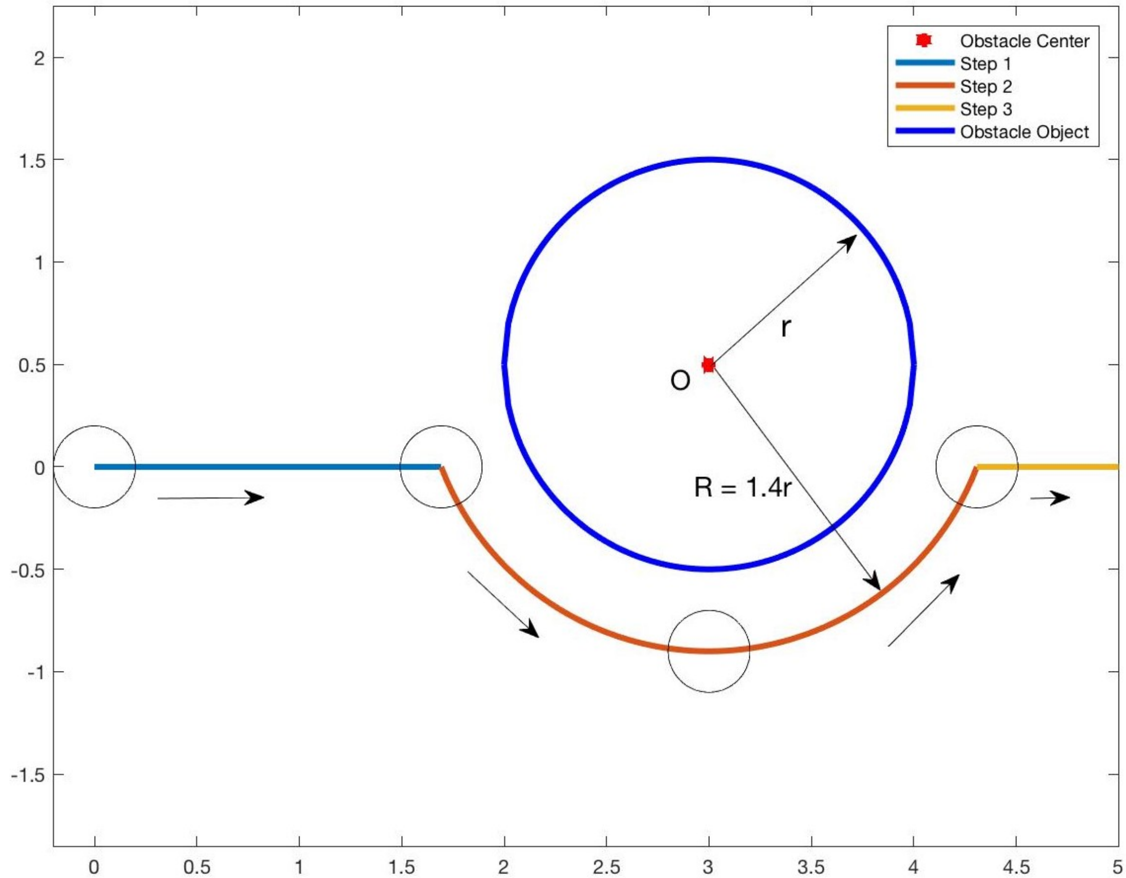
$$\theta = \arcsin(\frac{y_0}{r})$$

$$R = 1.4 * r$$

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

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

Figure 5: Trajectory-Based Obstacle Avoidance



Force-Based Obstacle Avoidance

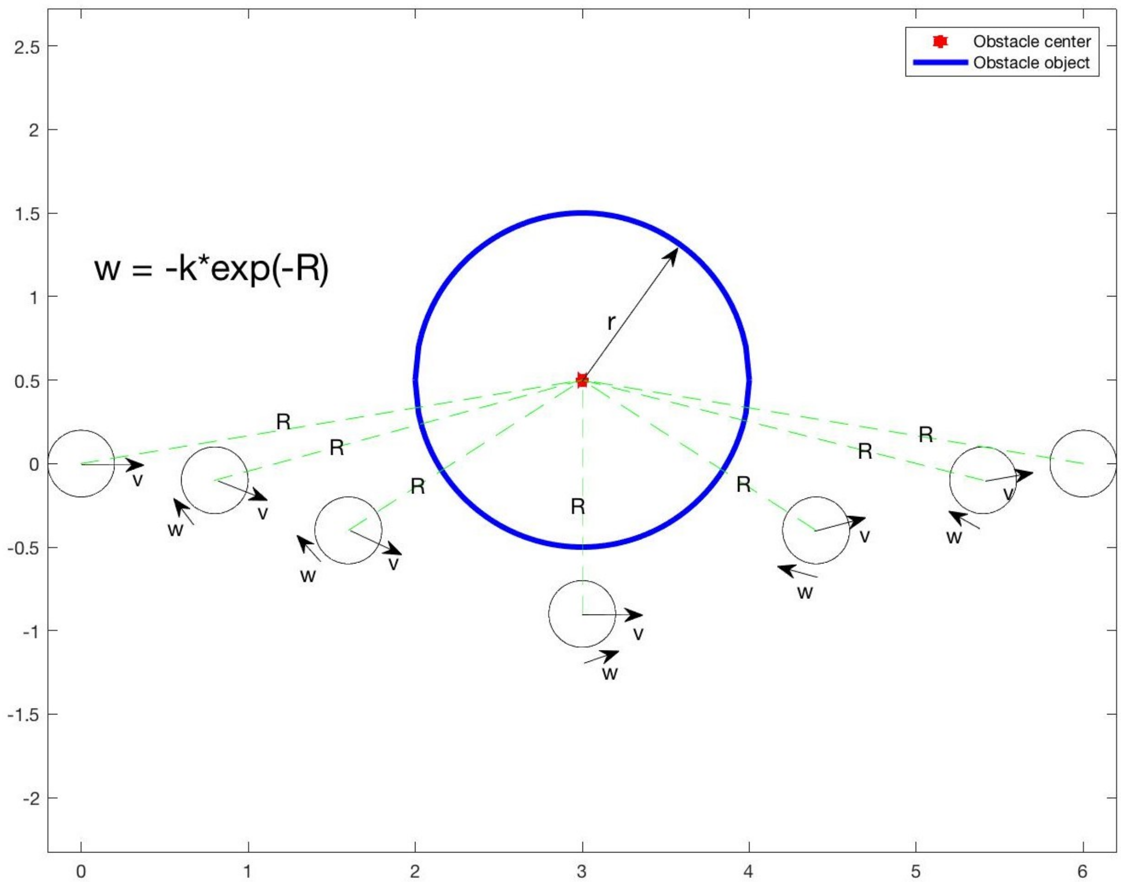
In this path, we set a virtual force which gives the turtlebot an angular velocity ω. Suppose R is the distance between the current position of turtlrbot and the center of obstacle and k is a positive scalar. We choose:

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

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.

Section 3 continued on next page. . .

Figure 6: Force-Based Obstacle Avoidance



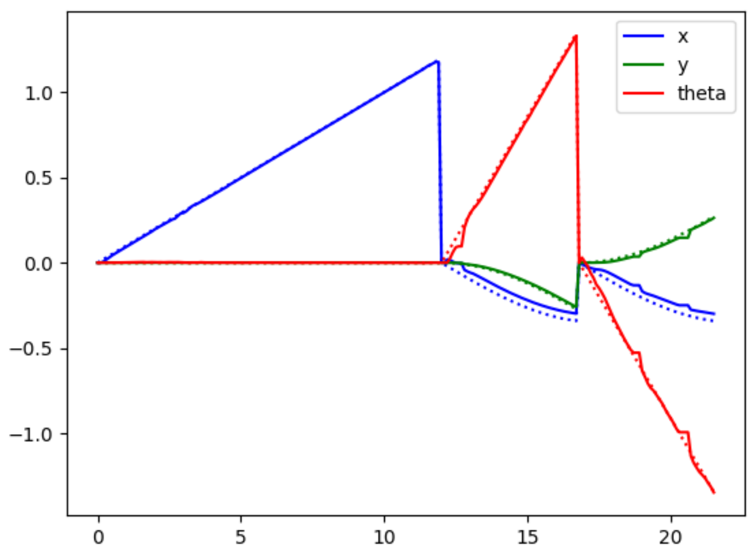
Section 4

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.

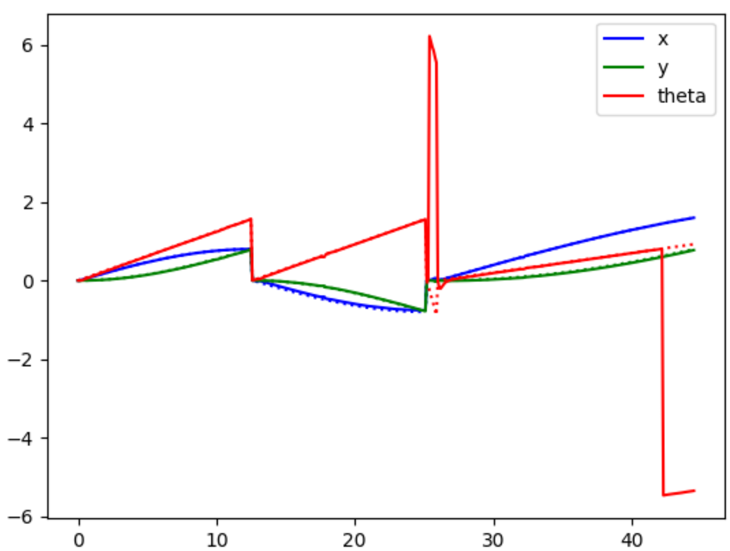
Finally, we successfully designed the arc path and linear path. The parallel parking path is composed of one linear path and two arc paths with angle $\frac{\pi}{2}$. No constraints were violated during the execution.

Section 4 continued on next page. . .

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

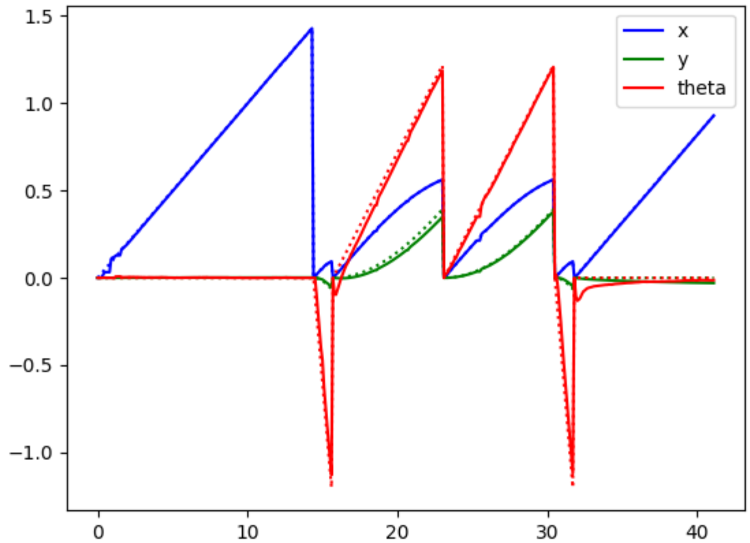
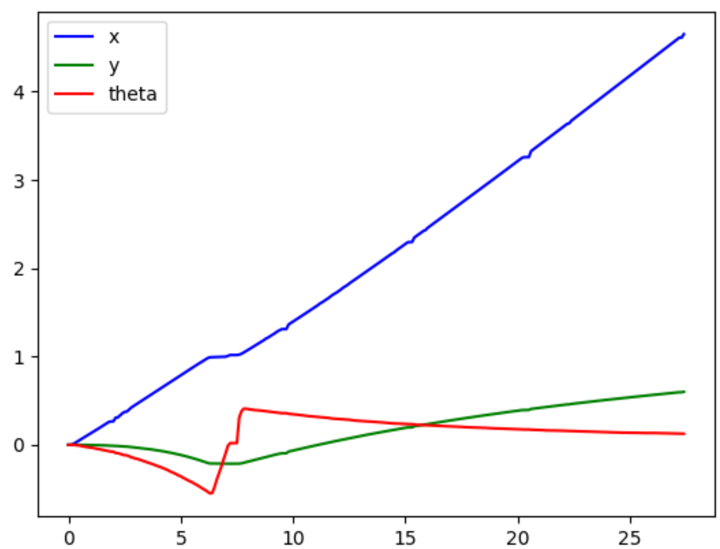


Figure 10: Obstacle Avoiding Virtual Force Constraint State Plot



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.

Section 4 continued on next page. . .

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 λ 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:

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


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


##############################################
############## Initialization ##############
##############################################
k = [5,3,3]
target_speed = 0.1
obstacle = False
obstacle_center = vec(2.0, -0.0)
obstacle_radius = 1

parallel_parking_path = []
# Step 1 parallel parking
direction = 1
s1 = 1.2
parallel_parking_path.append(LinearPath(s1, direction,target_speed))
# Step 2 parallel parking
direction = -1
left_turn = 1
r2 = 0.35
angle2 = np.pi/2.3
s2 = r2 * angle2
parallel_parking_path.append(ArcPath(r2, angle2, left_turn, target_speed, direction))
# Step 3 parallel parking
direction = -1
left_turn = -1
r3 = 0.35
angle3 = np.pi/2.3
s3 = r3 * angle3
parallel_parking_path.append(ArcPath(r3, angle3, left_turn, target_speed, direction))

path = ChainPath(parallel_parking_path)

# controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
# controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

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

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

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

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

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

    cnt = 0
    t=0
    times = []
    actual_states = []
    target_states = []
    while(cnt<3):
        flag = parallel_parking_path[cnt].total_length
        controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle, obstacle_center, obstacle_radius)
        # getting the position of the
        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
        start_state = np.array([start_pos[:2] + [GetyawFromQuat(start_rot)]]).T
        g = np.linalg.pinv(rigid(start_state))

        s = 0
        # path = parallel_parking_path
        # path = three_point_turn_path
        # path = compute_obstacle_avoid_path(4.0, vec(2.0, -0.0), 0.5)
        while not rospy.is_shutdown() and s <= flag:
            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
            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
            current_state = g.dot(current_state_odom)
            current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]


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

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

            cmd_vel.publish(move_cmd)

            # 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
            rate.sleep()
        cnt+=1

    # Plots
    times = np.array(times)
```

```
    actual_states = np.array(actual_states)
    target_states = np.array(target_states)



    plt.figure()
    colors = ['blue', 'green', 'red']
    labels = ['x', 'y', 'theta']
    for i in range(3):
        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')
    plt.legend()
    plt.show()


def shutdown():
    rospy.loginfo("Stopping TurtleBot")
    cmd_vel.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
    main()
    # try:
    #     main()
    # except e:
    #     print e
    #     rospy.loginfo("Lab3 node terminated.")
```

./code/main.py

controllers file:

```
# imports may be necessary
import numpy as np
import rospy
from geometry_msgs.msg import Twist, Vector3

class Controller():
  def __init__(self, path,k, target_speed, obstacle, obstacle_center, obstacle_radius):
    self.path = path
    self.target_speed = target_speed
    self.obstacle = obstacle
    self.obstacle_center = obstacle_center
    self.obstacle_radius = obstacle_radius
    self.k = k
    self.direction = path.direction


  def step_path(self, current_state, s):
    """
    Takes the current state and final state and returns the twist command to reach the
    target state
    according to the path variable

    Parameters
    ----------
    current_state: :obj:`numpy.ndarray`
      x , y , theta_z
      twist representing the current state of the turtlebot.  see utils.py
    s: float
      the path length the turtlebot should have travelled so far

    Returns
```

```
    -------
    :obj:`geometry_msgs.msg.Twist`
      Twist message to be sent to the turtlebot

    """
    # YOUR CODE HERE
    if not self.obstacle:
      vr = np.linalg.norm(self.path.target_velocity(s)[0:2])  # *np.sign(self.path.
    target_velocity(s)[0])
      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])
      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
    ]])
      vd = vdwd[0][0]
      wd = vdwd[1][0]

      twist = Twist()
      twist.linear.x = self.direction*abs(vd)
      twist.linear.y = 0
      twist.linear.z = 0
      twist.angular.x = 0
      twist.angular.y = 0
      twist.angular.z = wd
      return twist

# TURN = False

class ForceController(Controller):

  def __init__(self, k, target_speed, obstacle, obstacle_center, obstacle_radius):

    self.obstacle = obstacle
    self.obstacle_center = obstacle_center
    self.obstacle_radius = obstacle_radius
    self.k = k
    self.target_speed = target_speed
    self.pre_alpha = 0

  def step_path(self, current_state, s):
    Ex = np.array([[1],[0]])
    Ey = np.array([[0],[1]])
    vx = self.target_speed*Ex
    x = current_state[0][0]
    y = current_state[1][0]
    x0 = self.obstacle_center[0]
    y0 = self.obstacle_center[1]
    cur_angle = current_state[2][0]
    r = np.array([y0 - y, x0 - x])
    # maybe change
    # r = np.array([1/(y0 - y), 1/(x0 - x)])
    ######
    theta = np.arctan((y0 - y)/(x0 - x))
    if theta>0:
      theta = theta
    else:
      theta = np.pi+theta
    vth = self.k*np.linalg.norm(r)*(np.cos(theta)*Ex + np.sin(theta)*Ey)
    vth = -self.k*np.linalg.norm(r)*(-np.sin(theta)*Ex + np.cos(theta)*Ey)
    vthx = vth*np.cos(theta)
```

```
        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))
          # print('alpha', alpha)
        alpha = alpha[0][0]

        # if self.pre_alpha == 0:
        #   self.pre_alpha, dalpha = alpha, 0
        # else:
        #   self.pre_alpha, dalpha = alpha, alpha - self.pre_alpha

        # if cur_angle < -np.pi/4 or cur_angle > np.pi/4:
        #   dalpha = 0
          # dalpha *= 105
        print(alpha)

        error = np.exp(-abs(r[1])) # + np.exp(-abs(r[1]))*(x>x0)
        if abs(x0-x) < 0.02 and abs(alpha) > 0.003:
          dalpha = 1
          x = 0
        else:
          dalpha = -self.k*error
          x = 0.2
        # else:
        #   dalpha = self.k*error
        #   x = 0.2

        twist = Twist()
        twist.linear.x = x
        twist.linear.y = 0
        twist.linear.z = 0
        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = dalpha

        # twist.linear.x = 0
        # twist.angular.z = 1
        # print('alpha',alpha)
        print("twist", twist)
        return twist
```

./code/controllers.py

paths function file:

```
import numpy as np
import math
from math import sin, cos, asin, acos, atan2, sqrt
from utils import *
from matplotlib import pyplot as plt

class MotionPath:
    def target_state(self, s):
        """
        Target position of turtlebot given the path length s

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
```

```
        :obj:`numpy.ndarray`
            target position of turtlebot
        """
        raise NotImplementedError()

    def target_velocity(self, s):
        """
        Target velocity of turtlebot given the path length s

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target velocity of turtlebot
        """
        raise NotImplementedError()

    @property
    def total_length(self):
        """ total path length
        Returns
        -------
        float
            total path length
        """
        raise NotImplementedError()

    @property
    def end_state(self):
        """ Final state after completing the path
        Returns
        -------
        :obj:`numpy.ndarray`
            Final state after completing the path
        """
        return self.target_state(self.total_length)


class ArcPath(MotionPath):
    def __init__(self, radius, angle, left_turn, speed, direction): #, , left_turn):
        """
        Parameters
        ----------
        radius: float
            how big of a circle in meters
        angle: float
            how much of the circle do you want to complete (in radians).
            Can be positive or negative
        left_turn: bool
            whether the turtlebot should turn left or right
        """
        self.radius = radius
        self.angle = angle
        self.total_length = radius*angle
        self.direction = direction
        self.speed = speed
        self.left_turn = left_turn
```

```
    def target_state(self, s):
        """
        Target position of turtlebot given the current path length s for Circular Arc Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target position of turtlebot
        """
        # YOUR CODE HERE
        r = self.radius
        theta = self.left_turn*s/r
        if self.left_turn >0:
            x = self.direction*r*sin(theta)
            y = self.direction*(r - r*cos(theta))
        else:
            x = -self.direction*r*sin(theta)
            y = self.direction*(r*cos(theta) - r)

        return np.array([x,y,theta])

    def target_velocity(self, s):
        """
        Target velocity of turtlebot given the current path length s for Circular Arc Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target velocity of turtlebot
        """
        # YOUR CODE HERE
        r = self.radius
        theta = self.left_turn*s/r
        if self.left_turn >0:
            vx = self.speed*cos(theta)
            vy = self.speed*sin(theta)
            w = self.left_turn*self.speed/r
        else:
            vx = self.speed*cos(theta)
            vy = -self.speed*sin(theta)
            w = self.left_turn*self.speed/r

        return np.array([vx,vy,w])
    @property
    def total_length(self):
        """ total length of the path
        Returns
        -------
        float
            total length of the path
        """
        # YOUR CODE HERE
```

```
        return self.angle * self.radius
class LinearPath(MotionPath):
    def __init__(self, length, direction, speed):
        """
        Parameters
        ----------
        length: float
            length of the path
        """
        self.total_length = length
        self.direction = direction
        self.speed = speed
    def target_state(self, s):
        """
        Target position of turtlebot given the current path length s for Linear Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target position of turtlebot
            [x,y,theta_z]
        """
        # YOUR CODE HERE
        return np.array([self.direction*s,0,0])

    def target_velocity(self, s):
        """
        Target velocity of turtlebot given the current path length s for Linear Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target velocity of turtlebot
        """
        # YOUR CODE HERE
        return np.array([self.speed,0,0])

    @property
    def total_length(self):
        """ total length of the path
        Returns
        -------
        float
            total length of the path
        """
        # YOUR CODE HERE
        return self.total_length

class ChainPath(MotionPath):
    def __init__(self, subpaths):
        """
        Parameters
```

```
        ----------
        subpaths: :obj:`list` of :obj:`MotionPath`
            list of paths which should be chained together
        """
        self.subpaths = subpaths
        self.direction = self.subpaths[0].direction
        self.left_turn = self.subpaths[1].left_turn

    def target_state(self, s):
        """
        Target position of turtlebot given the current path length s for Chained Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target position of turtlebot
        """
        # YOUR CODE HERE
        if( s <= self.subpaths[0].total_length ):
            return self.subpaths[0].target_state(s)
        elif( s <= self.subpaths[0].total_length+self.subpaths[1].total_length ):
            # self.direction = self.subpaths[1].direction
            # self.left_turn = self.subpaths[1].left_turn
            return self.subpaths[1].target_state(s-self.subpaths[0].total_length)
        else:
            # self.direction = self.subpaths[2].direction
            # self.left_turn = self.subpaths[2].left_turn
            return self.subpaths[2].target_state(s-self.subpaths[0].total_length-self.
    subpaths[1].total_length)

    def target_velocity(self, s):
        """
        Target velocity of turtlebot given the current path length s for Chained Path

        Parameters
        ----------
        s: float
            the path length the turtlebot should have travelled so far

        Returns
        -------
        :obj:`numpy.ndarray`
            target velocity of turtlebot
        """
        # YOUR CODE HERE
        if( s <= self.subpaths[0].total_length ):
            return self.subpaths[0].target_velocity(s)
        elif( s <= self.subpaths[0].total_length+self.subpaths[1].total_length ):
            # self.direction = self.subpaths[1].direction
            # self.left_turn = self.subpaths[1].left_turn
            return self.subpaths[1].target_velocity(s-self.subpaths[0].total_length)
        else:
            # self.direction = self.subpaths[2].direction
            # self.left_turn = self.subpaths[2].left_turn
            return self.subpaths[2].target_velocity(s-self.subpaths[0].total_length-self.
    subpaths[1].total_length)
```

```
    @property
    def total_length(self):
        """ total length of the path
        Returns
        -------
        float
            total length of the path
        """
        # YOUR CODE HERE
        return self.subpaths[0].total_length+self.subpaths[1].total_length+self.subpaths[2].
    total_length

def compute_obstacle_avoid_path(dist, obs_center, obs_radius):
    # YOUR CODE HERE
    target_speed = 0.1
    direction = obs_center[1]/abs(obs_center[1])
    R = 1.2*obs_radius
    x_p = np.sqrt(R**2 - abs(obs_center[1])**2)
    theta = np.arccos(abs(obs_center[1])/R)

    obstacle_avoid_path = []
    # Step 1 obstacle_avoid_path
    s1 = obs_center[0] - x_p
    obstacle_avoid_path.append(LinearPath(s1,1,target_speed))
    # Step 2 obstacle_avoid_path
    left_turn = -direction
    r2 = 0.1
    angle2 = theta
    s2 = r2 * angle2
    obstacle_avoid_path.append(ArcPath(r2, angle2, left_turn, target_speed, 1))
    # Step 3 obstacle_avoid_path
    left_turn = direction
    r3 = R
    angle3 = theta
    s3 = r3 * angle3
    obstacle_avoid_path.append(ArcPath(r3, angle3, left_turn, target_speed, 1))
    # Step 4 obstacle_avoid_path
    left_turn = direction
    r4 = R
    angle4 = theta
    s4 = r4 * angle4
    obstacle_avoid_path.append(ArcPath(r4, angle4, left_turn, target_speed, 1))
    # Step 5 obstacle_avoid_path
    left_turn = -direction
    r5 = 0.1
    angle5 = theta
    s5 = r5 * angle5
    obstacle_avoid_path.append(ArcPath(r5, angle5, left_turn, target_speed, 1))
    # Step 6 obstacle_avoid_path
    s6 = dist - 2*x_p - s1
    obstacle_avoid_path.append(LinearPath(s6,1,target_speed))
    return obstacle_avoid_path


def plot_path(path):
    """
    Plots on a 2D plane, the top down view of the path passed in

    Parameters
    ----------
    path: :obj:`MotionPath`
        Path to plot
```

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

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

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

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

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

./code/paths.py

obs avoid function file:

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


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


#############################################
############## Initialization ##############
#############################################
k = [5,3,3]
target_speed = 0.1
obstacle = False
obstacle_center = [2,-0.2]
obstacle_radius = 0.5
dist = 3.5
parallel_parking_path = compute_obstacle_avoid_path(dist, obstacle_center, obstacle_radius)
# controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
# controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

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

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

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

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

    cnt = 0
    t=0
    times = []
    actual_states = []
    target_states = []
    while(cnt<6):
        flag = parallel_parking_path[cnt].total_length
        controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle, obstacle_center, obstacle_radius)
        # getting the position of the
        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
        start_state = np.array([start_pos[:2] + [GetyawFromQuat(start_rot)]]).T
        g = np.linalg.pinv(rigid(start_state))

        s = 0
        # path = parallel_parking_path
        # path = three_point_turn_path

        while not rospy.is_shutdown() and s <= flag:
            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
            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
            current_state = g.dot(current_state_odom)
            current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]


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

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

            cmd_vel.publish(move_cmd)

            # 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
            rate.sleep()
        cnt+=1

    # Plots
    times = np.array(times)
    actual_states = np.array(actual_states)
    target_states = np.array(target_states)

```

```
    plt.figure()
    colors = ['blue', 'green', 'red']
    labels = ['x', 'y', 'theta']
    for i in range(3):
        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')
    plt.legend()
    plt.show()


def shutdown():
    rospy.loginfo("Stopping TurtleBot")
    cmd_vel.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
    main()
    # try:
    #     main()
    # except e:
    #     print e
    #     rospy.loginfo("Lab3 node terminated.")
```

./code/obs_avoid.py

obs_avoid_force function file:

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


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


############################################
############## Initialization ##############
############################################
k = 0.2
target_speed = 0.1
obstacle = False
obstacle_center = [1,0.1]
obstacle_radius = 0.2
dist = 3.5


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

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

    # setting up the transform listener to find turtlebot position
    listener = tf.TransformListener()
    from_frame = 'odom'
    to_frame = 'base_link'
```

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

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

    cnt = 0
    t=0
    times = []
    actual_states = []
    target_states = []
    controller = ForceController(k, target_speed, obstacle, obstacle_center, obstacle_radius
    )
    # getting the position of the
    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
    start_state = np.array([start_pos[:2] + [GetyawFromQuat(start_rot)]]).T
    g = np.linalg.pinv(rigid(start_state))

    s = 0
    # path = parallel_parking_path
    # path = three_point_turn_path

    while not rospy.is_shutdown():
        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
        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
        current_state = g.dot(current_state_odom)
        current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]


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

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

        cmd_vel.publish(move_cmd)

        # 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
        rate.sleep()

    # Plots
    times = np.array(times)
    actual_states = np.array(actual_states)
    # target_states = np.array(target_states)

```

```
    plt.figure()
    colors = ['blue', 'green', 'red']
    labels = ['x', 'y', 'theta']
    for i in range(3):
        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')
    plt.legend()
    plt.show()


def shutdown():
    rospy.loginfo("Stopping TurtleBot")
    cmd_vel.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
    main()
    # try:
    #     main()
    # except e:
    #     print e
    #     rospy.loginfo("Lab3 node terminated.")
```

./code/obs_avoid_force.py

three_point function file:

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


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


############################################
############## Initialization ##############
############################################
k = [5,3,3]
target_speed = 0.1
obstacle = False
obstacle_center = vec(2.0, -0.0)
obstacle_radius = 1

parallel_parking_path = []
# Step 1 parallel parking
direction = 1
left_turn = 1
r1 = 0.8
angle1 = np.pi/2
s1 = r1 * angle1
parallel_parking_path.append(ArcPath(r1, angle1, left_turn, target_speed, direction))
# Step 2 parallel parking
direction = -1
left_turn = 1
r2 = 0.8
```

```
angle2 = np.pi/2
s2 = r2 * angle2
parallel_parking_path.append(ArcPath(r2, angle2, left_turn, target_speed, direction))
# Step 3 parallel parking
direction = 1
left_turn = -1
r3 = 0.08
angle3 = np.arccos(3.0/5.0)
s3 = r3 * angle3
parallel_parking_path.append(ArcPath(r3, angle3, left_turn, target_speed, direction))
# Step 4 parallel parking
direction = 1
left_turn = 1
r4 = 5*r1/2
angle4 = np.arccos(3.0/5.0)
s4 = r4 * angle3
parallel_parking_path.append(ArcPath(r4, angle4, left_turn, target_speed, direction))

path = ChainPath(parallel_parking_path)

# controller = Controller(path, k, target_speed, obstacle, obstacle_center, obstacle_radius)
# controller = Controller(path, target_speed, obstacle, obstacle_center, obstacle_radius)

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

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

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

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

    cnt = 0
    t=0
    times = []
    actual_states = []
    target_states = []
    while(cnt<4):
        flag = parallel_parking_path[cnt].total_length
        controller = Controller(parallel_parking_path[cnt], k, target_speed, obstacle,
    obstacle_center, obstacle_radius)
        # getting the position of the
        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
        start_state = np.array([start_pos[:2] + [GetyawFromQuat(start_rot)]]).T
        g = np.linalg.pinv(rigid(start_state))

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

```
            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
            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
            current_state = g.dot(current_state_odom)
            current_state[2][0] = GetyawFromQuat(current_rot) - start_state[2][0]


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

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

            cmd_vel.publish(move_cmd)

            # 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
            rate.sleep()
        cnt+=1

    # Plots
    times = np.array(times)
    actual_states = np.array(actual_states)
    target_states = np.array(target_states)



    plt.figure()
    colors = ['blue', 'green', 'red']
    labels = ['x', 'y', 'theta']
    for i in range(3):
        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')
    plt.legend()
    plt.show()


def shutdown():
    rospy.loginfo("Stopping TurtleBot")
    cmd_vel.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
    main()
    # try:
    #     main()
    # except e:
    #     print e
    #     rospy.loginfo("Lab3 node terminated.")
```

./code/three_point.py