PATH FOLLOWING CONTROLLER FOR MOBILE ROBOTS

MEAM 520 LAB-02

Dinesh Jagai
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
dinesh97@seas.upenn.edu

Thomas Wang
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
ttwang@seas.upenn.edu

Monday July 6^{th} 2020

1 Concepts

2 Coding Assignment

1. **Setup**
   Only three changes were made to `mobile_sandbox.py` throughout the lab. Firstly, for section $3.1.d$, `robot_params['tracking-point-axle-offset']`, l, was set to 0.035. This was done in order to ensure $r = l$ and the robot was able to spin in place.
   Also, `mobile_sandbox.py` was edited to plot both the e_1 and e_2 errors vs time as required in section 3.1
   Lastly, the `initial_state` was modified to start at $(0, 1)$

2. **Path Following Controller**
   Inserted in appendix i, listing 6
   Firstly, e_1 and e_2 were calculated using the following equations from the lecture slides (lec12):

$$e_1(x) = \gamma(x_p, y_p) = \gamma(x + lcos(\theta), y + lsin(\theta))$$

$$e_2(x) = v^{des} + \sqrt{\dot{x}_P^2 + \dot{y}_P^2}$$

   Note that in order to get $\dot{x}_P$ & $_P$ we essentially multiplied the matrix $\begin{pmatrix} 1 & 0 & -lsin(\theta) \\ 1 & 0 & lcos(\theta) \end{pmatrix}$ by $\dot{x}$
   Now, to get $\dot{q}$, the following equation was utilized:

$$\begin{bmatrix} \Gamma J \\ \frac{r}{2}\mathbf{1}^T \end{bmatrix} \dot{q} = \begin{bmatrix} -k_p e_1 \\ v^{des} + k_i \int e_2 dt \end{bmatrix}$$

$$\implies \dot{q} = \begin{bmatrix} \Gamma J \\ \frac{r}{2}\mathbf{1}^T \end{bmatrix}^{-1} \begin{bmatrix} -k_p e_1 \\ v^{des} + k_i \int e_2 dt \end{bmatrix}$$

   note in the code m_1 was used to represent $\begin{bmatrix} \Gamma J \\ \frac{r}{2}\mathbf{1}^T \end{bmatrix}$ and m_2 was used to represent $\begin{bmatrix} -k_p e_1 \\ v^{des} + k_i \int e_2 dt \end{bmatrix}$
   Furthermore, $\int e_2 dt$ was found by keeping an intrinsic variable `integral_e2` initialized to 0 in the Controller class. The variable was updated using the formula `integral_e2 = integral_e2 + delta_t*e2` each time the update method was called. The essentially allowed us to get the sum of all the e_2 with time - i.e. the integral of e_2 with time.
   Lastly, k_p, k_i were both arbitrarily chosen to be 1 and updated continuously to get the correct paths.

3. **Path Definition**
   Inserted in appendix i, listing 7
   For the path class the methods `gamma` and `gamma_jacobian` were updated as follows. Gamma was changed depending on the problem, but it essentially gave the function that evaluated to 0 when robot is on the path. For example,the path $y = 0$ can be given by $\gamma = y$. While `gamma_jacobian` simply returned the partial derivative w.r.t x and y. Note that the γs and Jacobians generated were specific to the required path.

3 Simulation

3.1 Experiments to Conduct in Simulation

1. N.b. the initial state in mobile stand-box was set to $(0, 1)$ for the following paths

   (a) Line $y = 2x + 1$

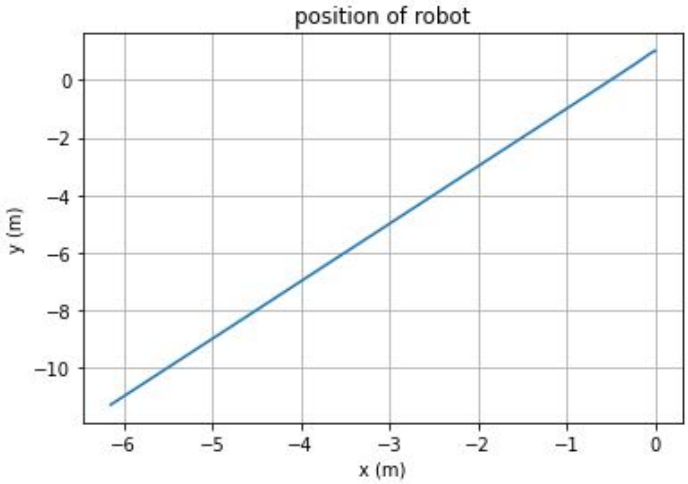

Figure 3.1.1.a: Figure Showing Plot Generated For The Given Path

code

Listing 1: mobile_path.py

```
import numpy as np


class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """

    def __init__(self, v):
        self.v_des = v
```

```
    def gamma(self, x, y):
        ### STUDENT CODE HERE ###
        # Test
#        gamma = y
        # 3.1.1a
        gamma = y -2*x - 1
        # 3.1.1b
        # gamma = y**2 + x**2 - 1
        # 3.1.1c
        # gamma = y - x**3 + x**2 + x - 1
        # 3.1.1d
#        r = 0.035
#        gamma = (y-1)**2 + x**2 - r**2
        return gamma

    def gamma_jacobian(self, x, y):
        # jacobian = np.array([0, 0])
        ### STUDENT CODE HERE ###
        # Test (For line y = 0 )
#        jacobian = np.array([0, 1])
        # 3.1.1a
        jacobian = np.array([-2, 1])
        # 3.1.1b
        # jacobian = np.array([2*x, 2*y])
        # 3.1.1c
        # jacobian = np.array([-3*(x**2) + 2*x + 1, 1])
        # 3.1.1d
#        jacobian = np.array([2*x, 2*(y-1)])

        return jacobian
```

(b) Line $y^2 + x^2 = 1^2$

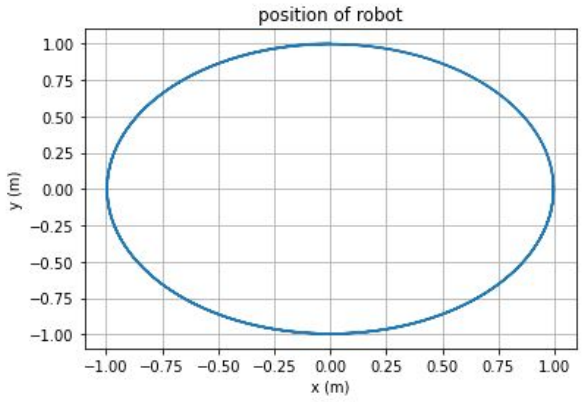

Figure 3.1.1.b: Figure Showing Plot Generated For The Given Path

code

Listing 2: mobile_path.py

```
import numpy as np
```

```
class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """

    def __init__(self, v):
        self.v_des = v

    def gamma(self, x, y):
        ### STUDENT CODE HERE ###
        # Test
#        gamma = y
        # 3.1.1a
#        gamma = y -2*x - 1
        # 3.1.1b
        gamma = y**2 + x**2 - 1
        # 3.1.1c
        # gamma = y - x**3 + x**2 + x - 1
        # 3.1.1d
#        r = 0.035
#        gamma = (y-1)**2 + x**2 - r**2
        return gamma

    def gamma_jacobian(self, x, y):
        # jacobian = np.array([0, 0])
        ### STUDENT CODE HERE ###
        # Test (For line y = 0 )
#        jacobian = np.array([0, 1])
        # 3.1.1a
#        jacobian = np.array([-2, 1])
        # 3.1.1b
        jacobian = np.array([2*x, 2*y])
        # 3.1.1c
        # jacobian = np.array([-3*(x**2) + 2*x + 1, 1])
        # 3.1.1d
#        jacobian = np.array([2*x, 2*(y-1)])

        return jacobian
```

(c) Line $y = x^3 - x^2 - x + 1$

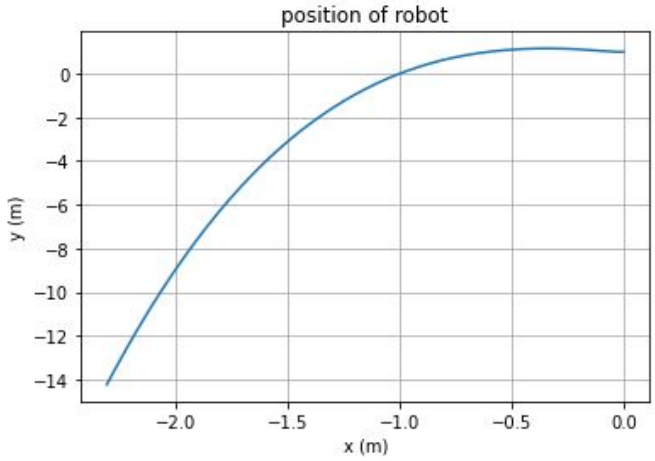

Figure 3.1.1.c: Figure Showing Plot Generated For The Given Path

code

Listing 3: mobile_path.py

```
import numpy as np


class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """

    def __init__(self, v):
        self.v_des = v

    def gamma(self, x, y):
        ### STUDENT CODE HERE ###
        # Test
#        gamma = y
        # 3.1.1a
#        gamma = y -2*x - 1
        # 3.1.1b
#        gamma = y**2 + x**2 - 1
```

```
        # 3.1.1c
        gamma = y - x**3 + x**2 + x - 1
        # 3.1.1d
#        r = 0.035
#        gamma = (y-1)**2 + x**2 - r**2
        return gamma

    def gamma_jacobian(self, x, y):
        # jacobian = np.array([0, 0])
        ### STUDENT CODE HERE ###
        # Test (For line y = 0 )
#        jacobian = np.array([0, 1])
        # 3.1.1a
#        jacobian = np.array([-2, 1])
        # 3.1.1b
#        jacobian = np.array([2*x, 2*y])
        # 3.1.1c
        jacobian = np.array([-3*(x**2) + 2*x + 1, 1])
        # 3.1.1d
#        jacobian = np.array([2*x, 2*(y-1)])

        return jacobian
```

(d) Line $x^2 + y^2 = (0.035)^2$

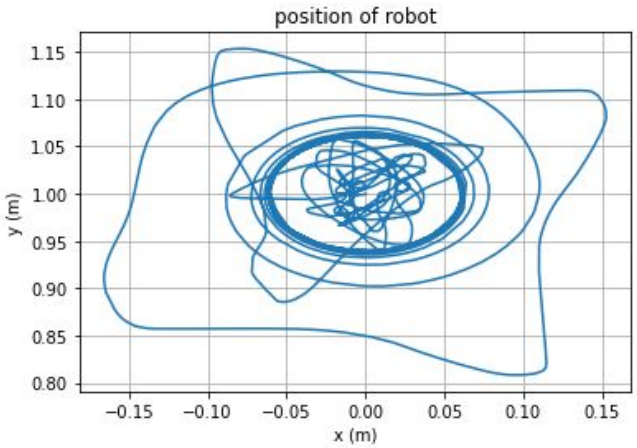

Figure 3.1.1.d: Figure Showing Plot Generated For The Given Path

code

Listing 4: mobile_path.py

```
import numpy as np


class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
```

```
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """

    def __init__(self, v):
        self.v_des = v

    def gamma(self, x, y):
        ### STUDENT CODE HERE ###
        # Test
#        gamma = y
        # 3.1.1a
#        gamma = y -2*x - 1
        # 3.1.1b
#        gamma = y**2 + x**2 - 1
        # 3.1.1c
#        gamma = y - x**3 + x**2 + x - 1
        # 3.1.1d
        r = 0.035
        gamma = (y-1)**2 + x**2 - r**2
        return gamma

    def gamma_jacobian(self, x, y):
        # jacobian = np.array([0, 0])
        ### STUDENT CODE HERE ###
        # Test (For line y = 0 )
#        jacobian = np.array([0, 1])
        # 3.1.1a
#        jacobian = np.array([-2, 1])
        # 3.1.1b
#        jacobian = np.array([2*x, 2*y])
        # 3.1.1c
#        jacobian = np.array([-3*(x**2) + 2*x + 1, 1])
        # 3.1.1d
        jacobian = np.array([2*x, 2*(y-1)])

        return jacobian
```

2. (a) using path : Line $y = 2x + 1$
with initial start configuration as $(2, 5)$ - note that this is point on the desired path

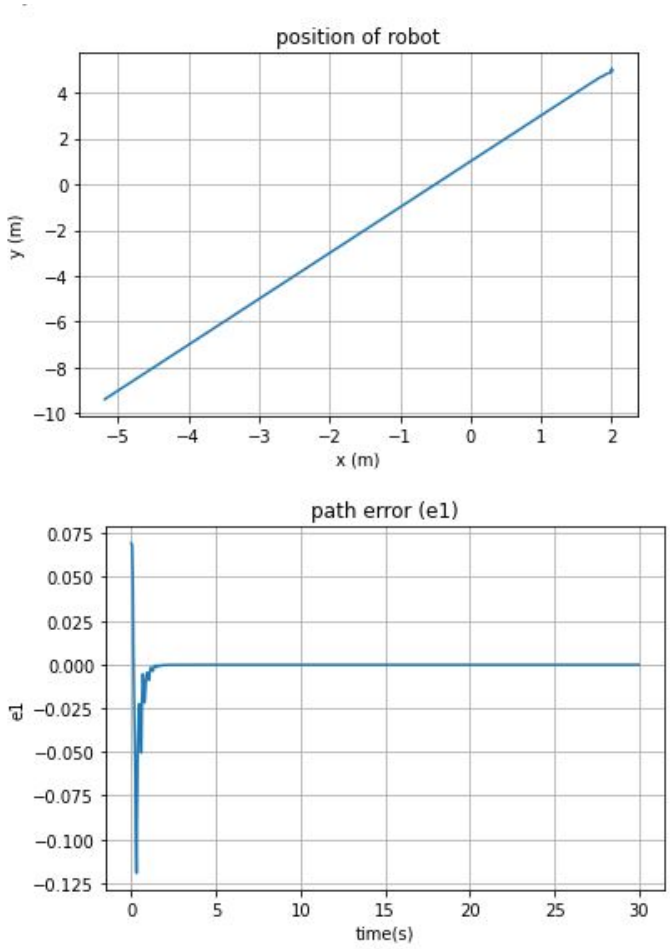

Figure 3.1.2.a.1: Figure Showing Path And Error Plots Generated For The Given Path

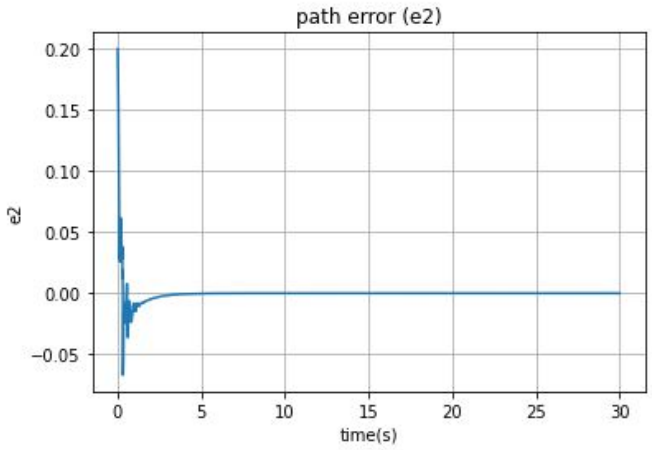

Figure 3.1.2.a.2: Figure Showing Path And Error Plots Generated For The Given Path

(b) using path : Line $y = 2x + 1$
with initial start configuration as $(1, 7)$ - note that this is point on near the desired path

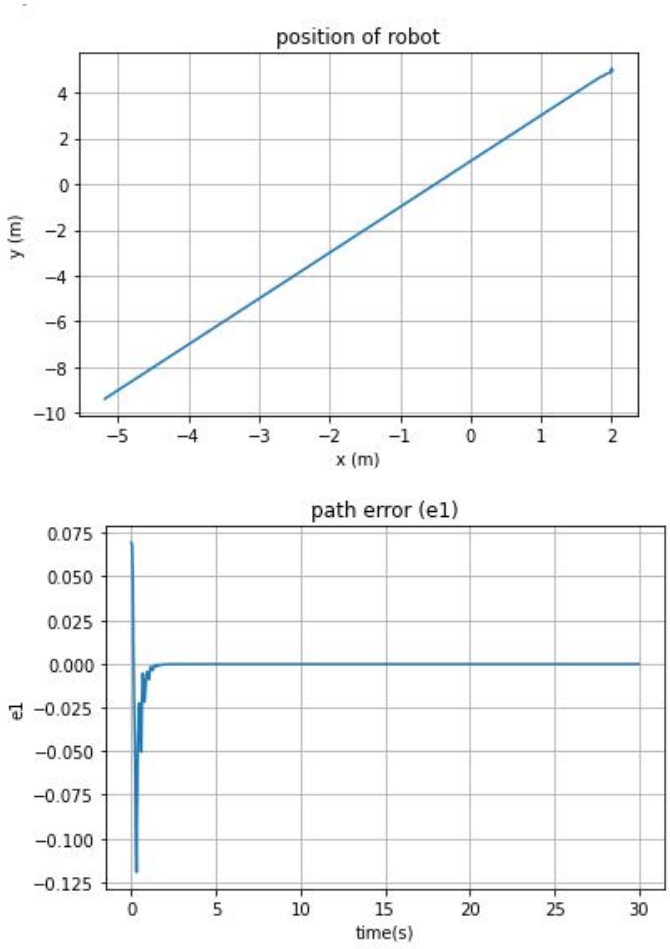

Figure 3.1.2.a.1: Figure Showing Path And Error Plots Generated For The Given Path

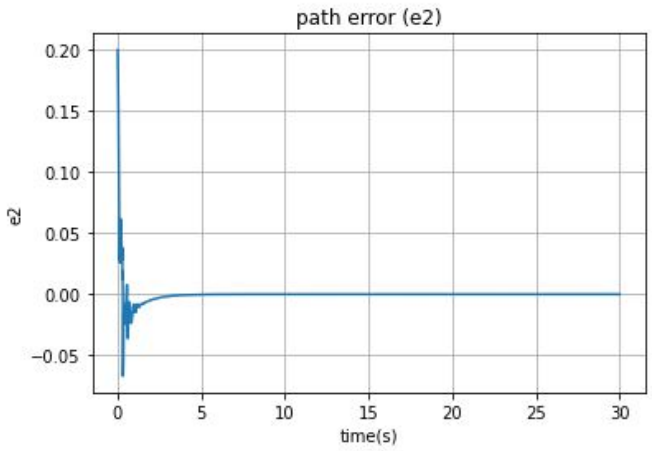

Figure 3.1.2.a.2: Figure Showing Path And Error Plots Generated For The Given Path

(c) using path : Line $y = 2x + 1$
with initial start configuration as $(1, 1000)$ - note that this is point on far the desired path

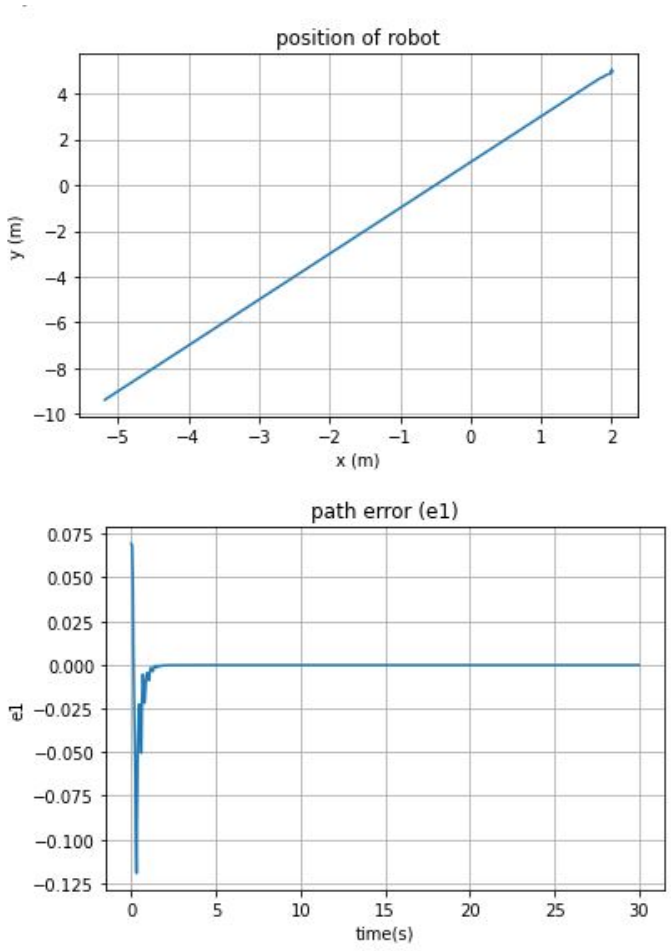

Figure 3.1.2.a.1: Figure Showing Path And Error Plots Generated For The Given Path

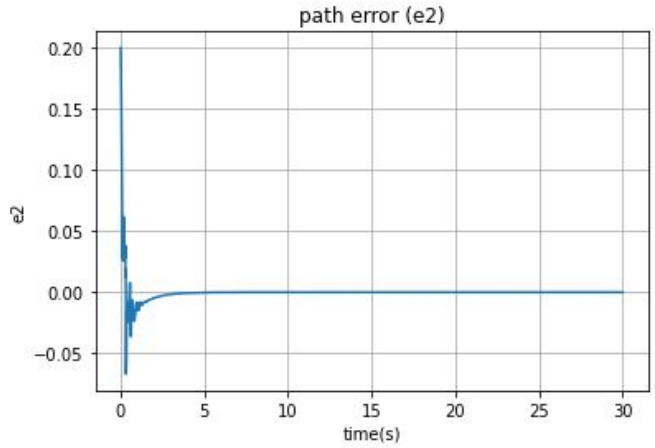

Figure 3.1.2.a.2: Figure Showing Path And Error Plots Generated For The Given Path

(d) using path : Circle $y^2 + x^2 = 1^2$
with initial start configuration as $(0, 0)$ - note that this is the center/middle of the circle

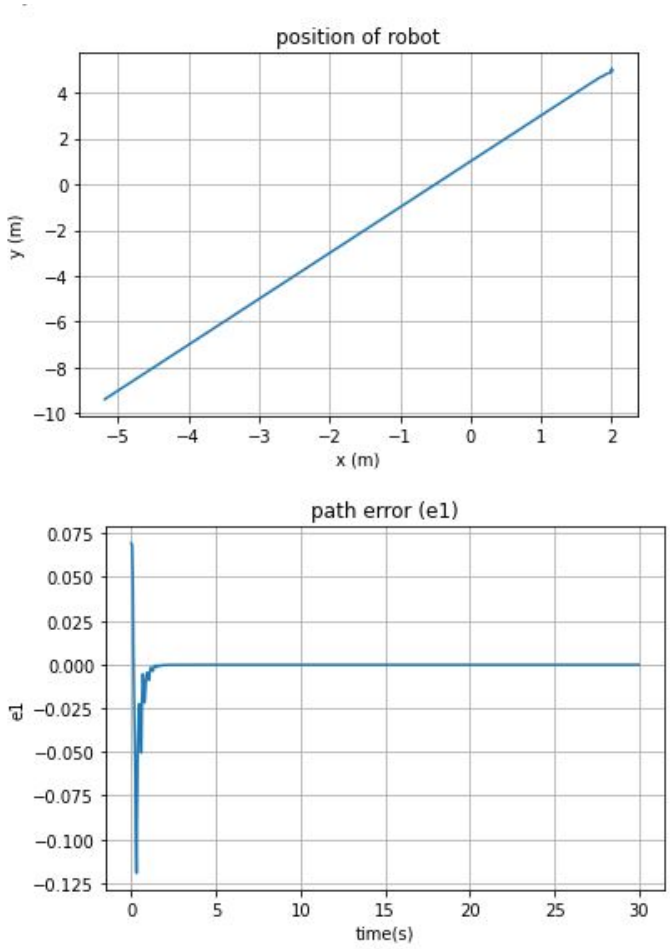

Figure 3.1.2.a.1: Figure Showing Path And Error Plots Generated For The Given Path

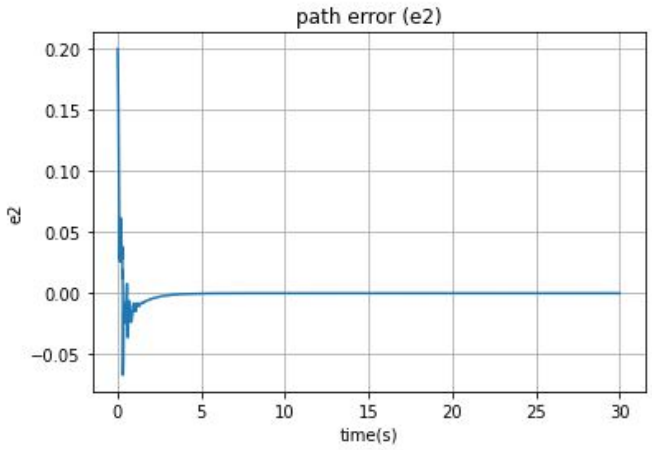

Figure 3.1.2.a.2: Figure Showing Path And Error Plots Generated For The Given Path

3. Constants held for the comparison :
   1) initial start configuration = $(0, 1)$
   2) using path : Line $y = 2x + 1$

   (a) Very small desired speed , v^{des} = 0.001

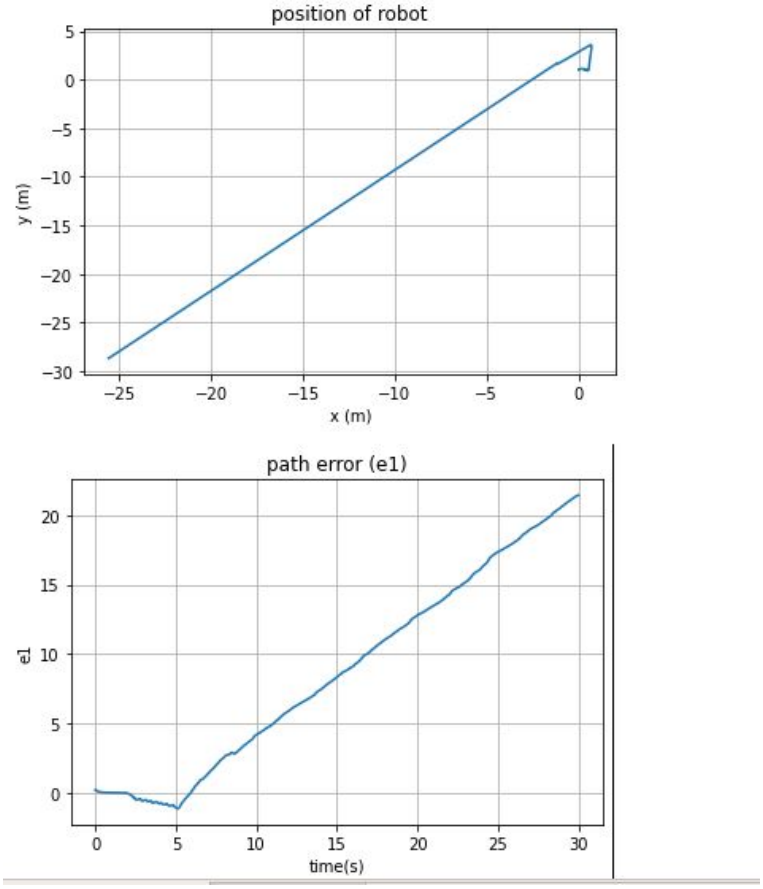

Figure 3.1.3.a.1: Figure Showing Path And Error Plots Generated For The Given Path

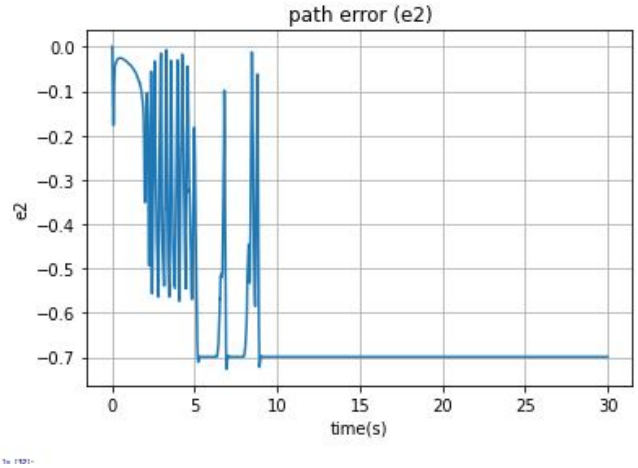

Figure 3.1.3.a.2: Figure Showing Path And Error Plots Generated For The Given Path

(b) normal desired speed , v^{des} = 0.2

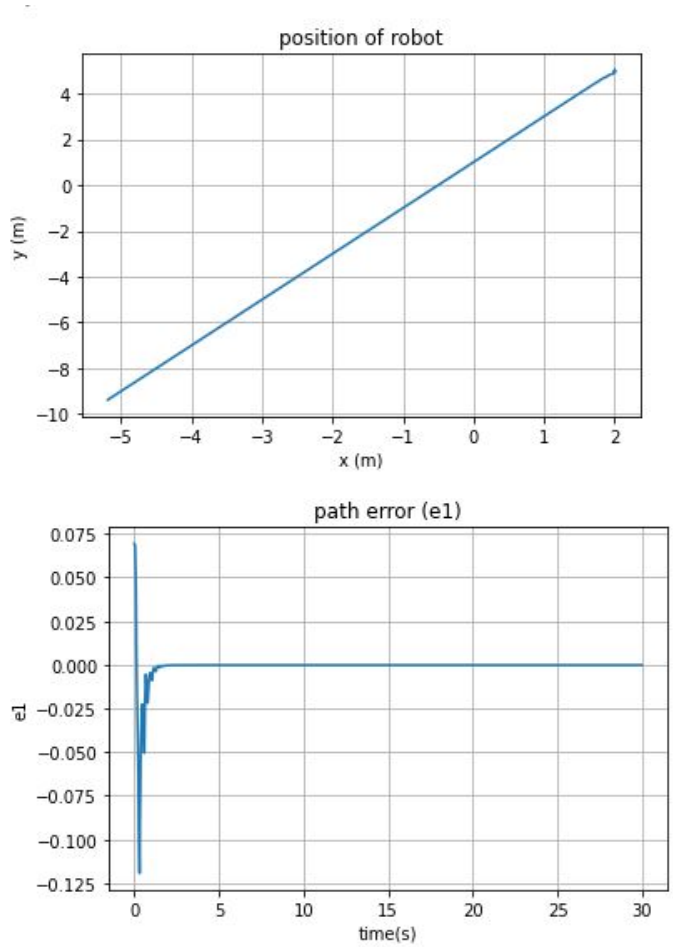

Figure 3.1.3.b.1: Figure Showing Path And Error Plots Generated For The Given Path

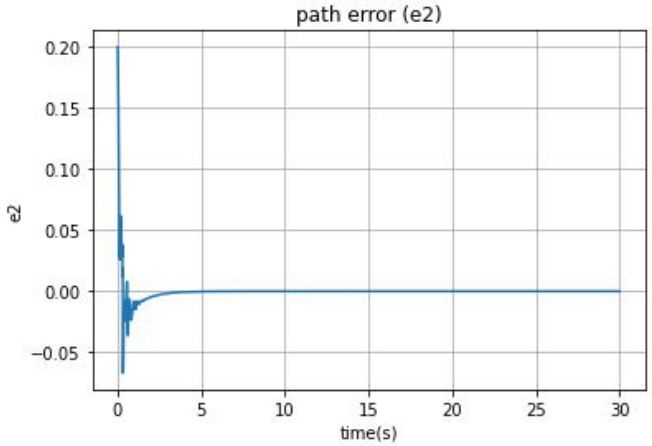

Figure 3.1.3.b.2: Figure Showing Path And Error Plots Generated For The Given Path

(c) high desired speed , v^{des} = 100

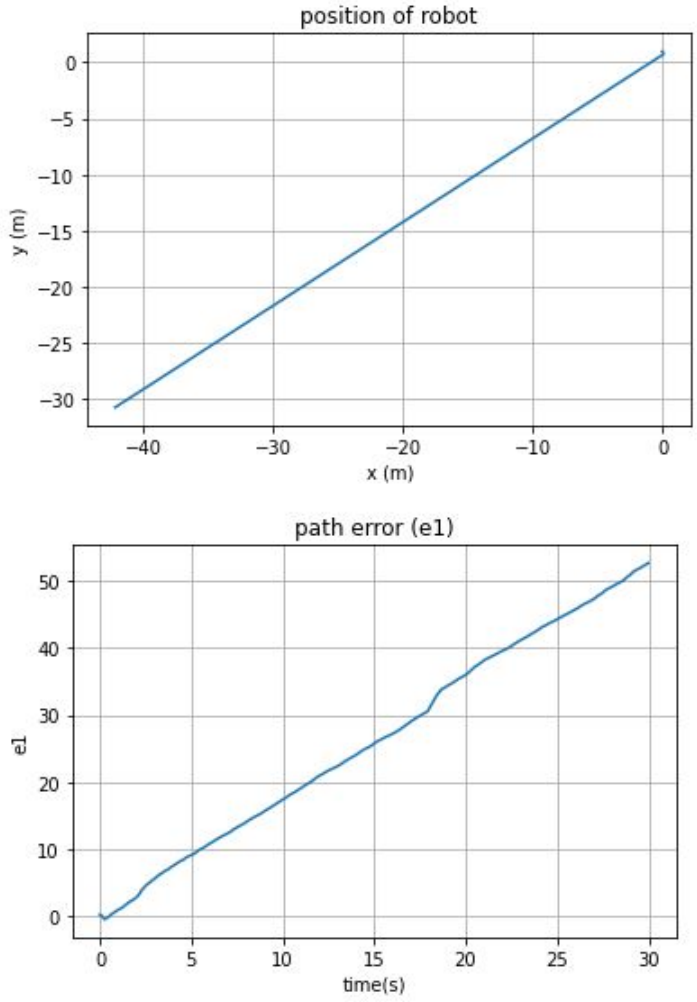

Figure 3.1.3.c.1: Figure Showing Path And Error Plots Generated For The Given Path

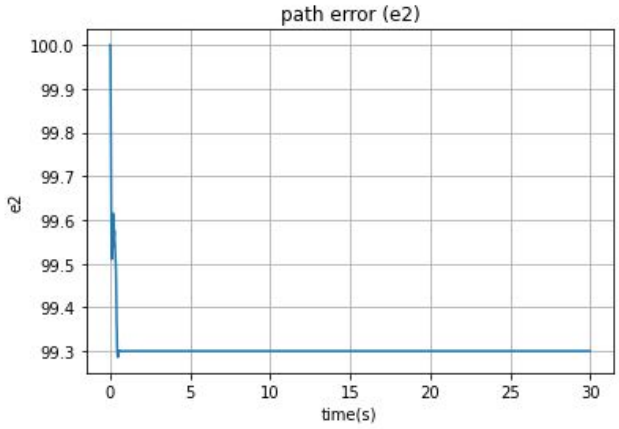

Figure 3.1.3.c.2: Figure Showing Path And Error Plots Generated For The Given Path

The higher the desired speed the higher the error but faster rate of convergence

4. Setting the tracking point axle offset increase errors in the robot path. After I increase the offset from 0.1 to 0.5, the path becomes slightly wrong. Then, I increase the offset to 1, and the error of path becomes even larger. On the other hand, if I decrease the offset to 0.01, there's also an increase in errors. So the tracking offset should be tuned to be a good parameter. If the parameter is large, the error is especially present on circular paths, but less for the straight line case. If the parameter is too small, both kinds of paths are thrown off.

3.2 Questions To Think About

1. The advantages of using this controller for a mobile robot are : It's easy to understand and code. When set up correctly, the robot can calculate and reach the desired path very quickly. It's also very accurate if given the right parameters. The disadvantages of using this controller for a mobile robot are : If the parameters are set correctly, it could easily fail to reach the path. Parameters work well for some paths but not other paths, making it difficult to tune parameters. Also, it doesn't consider wheel movements and physical limits of the robot.

   Quantities:

   (a) Rate at which the trajectory converges to the given trajectory. That is, how quickly is the robot able to converge to the expected path.

   (b) Does it get to the given trajectory (that is, does it find a solution).

   (c) Smoothness of the trajectory (the lack of outliers and erroneous behavior gives an crucial understand of the effectiveness of the controller).

   (d) Does the controller actually work

2. I started with k_p and k_i as 1 and 1, and it worked well. Since there are only two parameters here to tune, my strategy is to start by fixing one variable and altering the other one. I first attempted to increase the values dramatically, such as setting one or both as 10, and saw a large increase in errors. Since some paths such as the straight line one and the circle one are already quite good under (1,1), there wasn't much room for improvements and most cases only worsen the results. Then, I attempted to improve the case where gamma equals y. There's a slight improvement when $k_p = 2$ (or something similar) and $k_i = 1$, and when $k_p = 1, k_i = 2$. However, with these parameters, other paths, such as the circular path, become a lot worse. Different paths respond to different parameters, so once you understand the types of paths that the robot will traverse, the better you can choose parameters.

3. The parameters are important for the controller to reach the right path. I tuned the l value, which is the tracking point axle offset, and if the value is too large (1) or too small (0.01), the controller fail to reach the path. The effect also varies for different paths, such as the straight path is uneffected by large l but circular path is effected. Also, initial configuration makes a difference in if the path succeeds or not. Some paths are reachable when the initial configuration is $(0, 0)$, and others when it's $(0, 1)$. Also, k_p and k_i is significant, as when they are both 1, they can arrive at the test paths, but when they are really big or really small, the controller doesn't reach the paths. There are many possible conditions that would cause the controller to fail, and that's why parameter tuning is important.

4. When deploying the robot in the real world, it will experience friction (slide friction) when traveling. There is friction between the wheels and the ground (sliding friction), and there also might be friction within the robot's parts. Friction were not part of the calculation and might cause unexpected behaviours. Also, we assume the wheels roll well and correctly. There might be movements of the wheel that's infeasible in real world mechanics, such as two wheels rolling in very different directions or wheels turn in opposite directions. In addition, there might be unexpected obstacles on the road, such as oil on the floor causing the wheel to slide, and bumps of rocks that could trip the robot. Also, real world wheel acceleration limitations can cause the motion displayed in the real world to be different from that simulation. This is because the model is only a kinematic model and as such, with the right k_p and k_i parameters it can go off with a huge acceleration that will be impossible in the real world. Lastly, there are limits in the turning radius - the model is assumed to turn arbitrarily quickly but this is no possile in the real world.

5. When traveling through a path of given way points, the robot could think of each way point as a destination, and travel towards it. After it arrives, we can give it the next way point as the destination. Similarly, if it gets within a certain proximity to the destination, we can change its destination without it being exactly at the

way point. This way, the overall path would be slightly smoother. A different way is to interpolate a smooth curve given those way points, and simply make the robot follow that smooth curve. This smooth curve might require more mathematical calculations and sometimes might not be feasible, and in those situations we can make adjustments for inaccuracies. Overall, we hope to traverse through the way points in a path as smooth as possible.

4 Appendix

Listing 5: mobile_sandbox.py

```
from mobile_robot import MobileRobot
from mobile_controller import Controller
from mobile_path import Path
from time import sleep
import matplotlib.pyplot as plt
import numpy as np


if __name__=='__main__':

    # Instantiate robot
    initial_state = np.array([0, 0, np.pi])
    robot = MobileRobot(initial_state)

    # Create dictionary of robot parameters
    robot_params = robot.get_params()
    robot_params['tracking-point-axle-offset'] = .1

    # Instantiate controller
    controller = Controller(Path(0.2), robot_params, 100)

    # Set max time for simulation to run for
    t_max = 30 # seconds

    # Initialize data structures for plotting
    cycle = 0
    position = np.empty((t_max * controller.frequency, 2))
    error = np.empty((t_max * controller.frequency, 2))

    # Control loop to run until user interrupts it or time expires
    print("Setup complete - entering loop")
    while (cycle / controller.frequency < t_max):
        try:
            # get most recent state
            x, x_dot = robot.get_state()
            position[cycle, :] = x[0:2]

            # calculate new wheel speeds
            q_dot, e1, e2 = controller.update(x, x_dot)
            error[cycle, :] = [e1, e2]

            # set wheel speeds
            robot.set_wheel_velocity(q_dot)

            # repeat this loop after waiting appropriate amount
            sleep(1 / controller.frequency)

            # update data structures for plotting
            cycle += 1

        # using CNTRL-C will interrupt loop
        except KeyboardInterrupt:
            break

    # kill simulation
    robot.stop()

    # example code to create plots, modify as needed
    fig, ax = plt.subplots()
    t = np.arange(0, cycle) / controller.frequency
    ax.set(xlabel='x (m)', ylabel='y (m)', title='position of robot')
```

```
    ax.plot(position[:, 0], position[:, 1])
    ax.grid()
    plt.show()

#    E1 plot
    fig, ax = plt.subplots()
    t = np.arange(0, cycle) / controller.frequency
    ax.set(xlabel='time(s)', ylabel='e1', title='path error (e1)')
    ax.plot(t[:], error[:, 0])
    ax.grid()
    plt.show()

    #    E2 plot
    fig, ax = plt.subplots()
    t = np.arange(0, cycle) / controller.frequency
    ax.set(xlabel='time(s)', ylabel='e2', title='path error (e2)')
    ax.plot(t[:], error[:, 1])
    ax.grid()
    plt.show()
```

Listing 6: mobile_controller.py

```
import numpy as np


class Controller:
    """
    a class implementing path following controller for a differential
    drive robot

    ...

    Attributes
    ----------
    path :
        the desired path to be followed by the robot, an instantiation
        of the Path class
    robot_params :
        a dictionary of intrinsic robot parameters to be used by
        the controller
    frequency :
        the frequency that the controller loop runs at, measured in Hz

    Methods
    -------
    update(x, x_dot):
        calculate the wheel speeds needed to follow the desired path using
        the latest state information
    """

    def __init__(self, path, robot_params, frequency):
        '''
        initialize the Controller class

        Parameters:
            path :
                the desired path for the robot to follow
                an instance of the Path class
            robot_params:
                a dictionary of intrinsic robot parameters to be used by
                the controller
            frequency :
                the frequency that the controller loop runs at, measured in Hz
        '''

        # intrinsics
        self.path = path
        self.frequency = frequency # in Hz
        self.r = robot_params['wheel-radius']
        self.d = robot_params['axle-width']
        self.l = robot_params['tracking-point-axle-offset']
        self.integral_e2 = 0

        ### STUDENT CODE HERE ###

    def update(self, x, x_dot):
        '''
        calculate the wheel speeds needed to follow the desired path using
        the latest state information

        Parameters:
            x :
                numpy array of length 3 representing state of robot
                array elements correspond to [x, y, theta]
            x_dot :
```

```
            numpy array of length 3 representing derivative of robot state
            array elements correspond to [x_dot, y_dot, theta_dot]

        Return:
            q_dot :
                numpy array of length 2 representing desired wheel speeds
                array elements correspond to [phi_dot_right, phi_dot_left]
            e1 :
                a measure of position error, a scalar
            e2 :
                a measure of velocity error, a scalar
        '''

        q_dot = np.array([0, 0])
        e1 = 0
        e2 = 0
        ### STUDENT CODE HERE ###
        kp = 1
        ki = 1
        xp = x[0] + self.l*np.cos(x[-1])
        yp = x[1] + self.l*np.sin(x[-1])
        e1 = self.path.gamma(xp, yp)
        xp_dot = np.array([[1, 0, -self.l*np.sin(x[-1])], [0, 1, self.l*np.cos(x[-1])]])@(x_dot)
        e2 = self.path.v_des - np.sqrt(xp_dot[0]**2 + xp_dot[1]**2)
        delta_t = 1/(self.frequency)
        self.integral_e2 = self.integral_e2 + delta_t*e2
        m_2 = np.array([[-kp*e1], [self.path.v_des + ki*self.integral_e2]])
        jacobian = np.array([[(self.r*np.cos(x[-1])/2 + self.l*self.r*np.sin(x[-1])/self.d),
            (self.r*np.cos(x[-1])/2 - self.l*self.r*np.sin(x[-1])/self.d)], [
                        (self.r*np.sin(x[-1])/2 - self.l*self.r*np.cos(x[-1])/self.d),
                            (self.r*np.sin(x[-1])/2 + self.l*self.r*np.cos(x[-1])/self.d)]])
        #m_1 = np.array([[self.path.gamma_jacobian(xp, yp)@jacobian], [(self.r/2)*np.array([1,
            1])]])
        m_1 = np.vstack((self.path.gamma_jacobian(xp, yp)@jacobian,((self.r/2)*np.array([1, 1]))))
#       print(m_1)
#       print(((self.r/2)*np.array([1, 1])).shape)
#       print((self.path.gamma_jacobian(xp, yp)@jacobian).shape)
        q_dot = np.linalg.inv(m_1)@m_2
        return q_dot, e1, e2
```

Listing 7: mobile_path.py

```
import numpy as np


class Path:
    """
    a class defining a path for a mobile robot

    ...

    Attributes
    ----------
    v :
        desired forward velocity to follow the path with


    Methods
    ----------
    gamma :
        function with arguments x, y that evaluates to 0 when robot is
        on the path
    gamma_jacobian :
        function with arguments x, y that returns and evaluates the
        jacobian of the gamma function at the point (x, y)
    """

    def __init__(self, v):
        self.v_des = v

    def gamma(self, x, y):
        ### STUDENT CODE HERE ###
        # Test
        gamma = y
        # 3.1.1a
#        gamma = y -2*x - 1
        # 3.1.1b
        # gamma = y**2 + x**2 - 1
        # 3.1.1c
        # gamma = y - x**3 + x**2 + x - 1
        # 3.1.1d
#        r = 0.035
#        gamma = (y-1)**2 + x**2 - r**2
        return gamma

    def gamma_jacobian(self, x, y):
        # jacobian = np.array([0, 0])
        ### STUDENT CODE HERE ###
        # Test (For line y = 0 )
        jacobian = np.array([0, 1])
        # 3.1.1a
#        jacobian = np.array([-2, 1])
        # 3.1.1b
        # jacobian = np.array([2*x, 2*y])
        # 3.1.1c
        # jacobian = np.array([-3*(x**2) + 2*x + 1, 1])
        # 3.1.1d
#        jacobian = np.array([2*x, 2*(y-1)])

        return jacobian
```