In the name of God

Shahid Beheshti University of Tehran

Faculty of Computer Science and Engineering

Mobile Robot Kinematics and Control

Fundamentals of Robotics Course

Shiva Zeymaran

Yalda Yarandi

Winter 2021

1. Overview

This computer assignment contains three parts about kinematic and control in mobile robots that we will discuss the implemented code for each component in this report entirely.

2. Description

2.1. Forward kinematic

In this part, the code contains three functions that calculate the result of forward kinematic. These three functions are named “calculate_local”, “change_coordinates”, and “forward_cal”. Let’s dive into their functionalities:

- **calcualte_local**

  This function gets the wheels' radius, the length of the shaft between two wheels, and each wheel's angular velocity in Degree/Second.

  The return value is the robot's linear and angular velocity in the robot's frame.

  To see more details for this function's functionality and the formulas, look at slide#22 of course lectures.

  Note that for changing Degree into Radian, needed calculations performed.

- **change_coordinates**

  This function shared between forward and inverse kinematic, in the forward kinematic case, gets the result matrix from the last function, theta in Degree,

and a boolean variable named "toGlobal" (to distinguish between forward and inverse calculations) as input parameters.

The return value is again the robot's linear and angular velocity but this time in the Inertial Frame.

- **forward_cal**

  Anyone who needs to calculate forward kinematic for any robot must call this primary function.

  All that we need is to call both previous functions consecutively. But the important thing is to call "change_coordinates" with the boolean value of "True"; This causes us to get used to this function in a "direct " way, and the result will be velocities in the Inertial Frame.

The velocity of the robot is calculated up to here. Now, it's time to move the robot in the Inertial Frame and see the results. "forward_res" and "forward_plot" functions calculate the place of the robot at each time and plot the result:

- **forward_res**

  This function calls "forward_cal", t_range times; Which t_range is a variable that describes the time we want to capture our robot's movement.

  Note that x0 and y0 are two variables that show the robot's initial position in the Inertial Frame (It can be passed as a parameter to this function, too).

  At the end of this function, the "forward_plot" is called to plot the results.

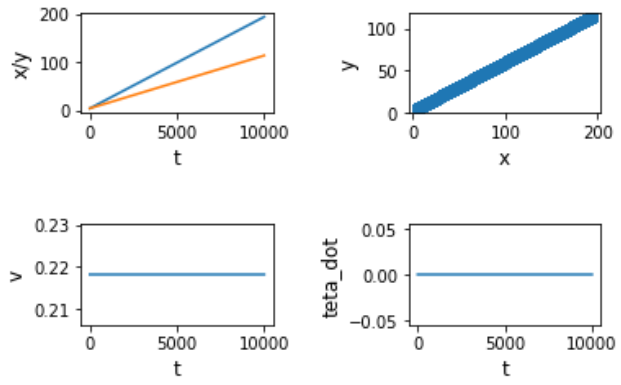
- **forward_plot**

  This function plots four diagrams according to the things that the question mentioned. We will explain each chart in more detail soon.

Tests that be done to check the functionality of the code are as following:

1. Equal and same direction angular velocity for both wheels

   In this case, we expected the robot to move in a straight line and without turning in either direction. Therefore, we test it with a robot with two wheels with a radius of 2.5 and a shaft size of 20. We assume that its wheels' angular velocity is 5 and 5 with angles of 30 Degree with the Inertial Frame. Here are the plot results:



   The first plot has two lines: the blue one is for x_t, and the orange one indicates y_t. Both of them increase linearly in time.

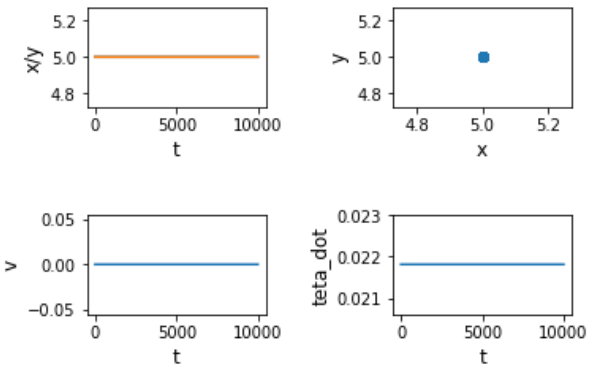
   The second plot indicates the path that the robot moves that is moving in a straight line correctly in this case.

The third plot indicates the changes in linear velocity during the time. It is just the change of velocity toward the x-axis because the linear velocity toward the y-axis in the robot's coordinate is equal to Zero. This velocity is a constant value all time since the robot's movement is not accelerated.

The last plot indicates the changes in angular velocity in time. It is constant the whole time since the robot does not turn in any direction.

2. Equal angular velocity for both wheels and different directions

   In this case, we expected the robot to be fixed in its position. Here is the result:



As it's clear from the first diagram, the robot's x and y coordinates are fixed during the time, and it is equal to the first position of the robot.

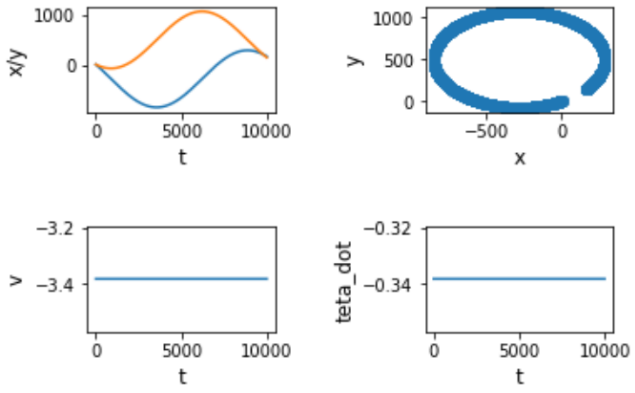
The second diagram indicates that the robot is correctly fixed in its place, and the robot's path is just a dot!

The third diagram indicates that the linear velocity is zero during the time and the robot hasn't any movements.

The last one indicates a little angular velocity that I think it's because of some errors and some natural causes.

3. Zero angular velocity for one wheel and non-zero for the other

   In this case, we expected the robot to just turn around in its place. Here is the result:



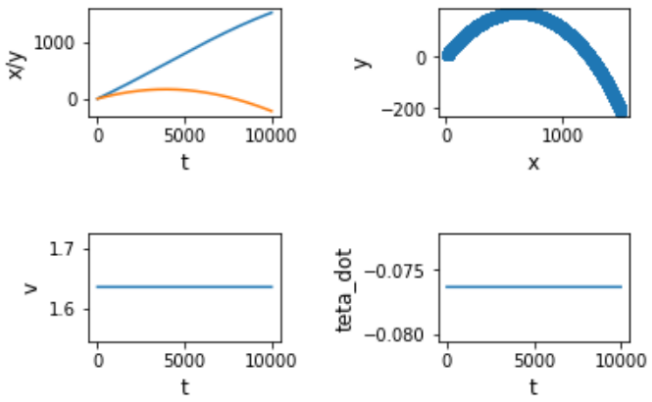
Following the first diagram, the robot's x coordinate decreases until around 3000 seconds and then, it increases (Like the first test, the blue line indicates the changes of x per time and the orange line indicates the changes of y). In contrast, the robot's y coordinate increases and then it decreases, you can see these changes in the second diagram. Indeed, when one of the wheels is fixed

and the other one has speed, the robot's middle point turns around the wheel that is fixed, so you can see a circle in the diagram.

The third one, the same as the first test, is just the change of velocity toward the x-axis. This velocity of the last two diagrams are a constant value (upwards of -3.4 and -0.34) all times since the robot's movement is not accelerated.

4. Larger angular velocity for one wheel than the other

   As the two wheels' velocity are in the same direction, we expected that the robot would go forward and also, as one of the wheels has a bit more speed, we expected a small deviation. Here is the result:



   In the first plot, the blue line shows the forward-going (in the direction of the x-axis) and the orange line indicates the changes of the y-axis which has been created because of the speed difference.

   In the second plot, you can trace the robot that is the same as we expected.

Just like the previous examples, the velocity diagrams are linear since the robot's movement is not accelerated.

2.2. Inverse kinematic

In this part, again, the code contains three functions that calculate the result of inverse kinematic. These three functions are named "change_coordinates", "calculate_velocity", and "inverse_cal". Let's dive into their functionalities:

- **change_coordinates**

  This function is described before in section 2.1. The only difference is in the way of using this function that will describe very soon.

- **calculate_velocity**

  This function gets a matrix that results from "change_coordinates", radius, and the shaft size as input parameters.

  This function will perform exactly in a reverse way of "calculate_local" in section 2.1. After solving the equations, the result value will be the wheels' angular velocity in Radian/Second.

- **inverse_cal**

  Anyone who needs to calculate inverse kinematic for any robot must call this primary function.

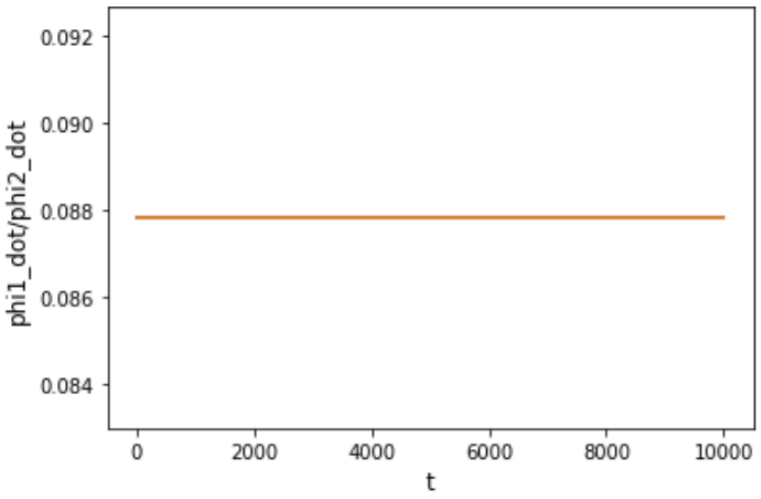
  All that we need is to call both previous functions consecutively. But the important thing is to call "change_coordinates" with the boolean value of

"False"; This causes us to get used to this function in an "inverse" way, and the result will be velocities in the robot's frame.

Tests that be done to check the functionality of the code are as following:

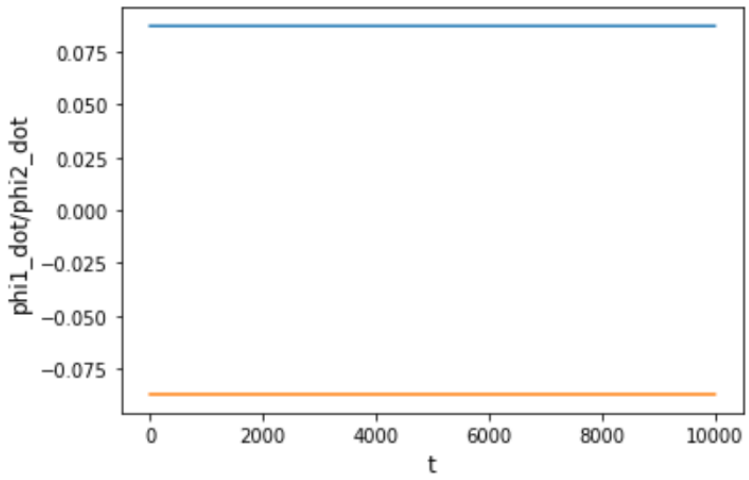
1. Linear movement with constant velocity to the straight forward

   In a nutshell, the velocity of the wheels should be the same as each other to go the robot straight. Also, the velocity is fixed because the robot's movement is not accelerated. So, as you can see in the diagram, the velocity of both of them is roughly 0.088 radian/s.



2. Circular in-place movement (without changing position)

   As the below results, to make the robot turn around its place, you should give the wheels the same amount of speed in opposite directions.



2.3. Control

In this part, there are two functions. These functions are named “velocity_cal”, and “controller”. Let’s dive into their functionalities:

- **velocity_cal**

  This function takes the robot's initial position and its destination and calculates the amount of velocity its needs. More precisely, it takes the initial x, y and theta and the destination's x and y and gets v and w (linear and angular velocity). By slide#41 of the third-course lecture, we can calculate alpha, beta and rho from those inputs. Furthermore, we had to choose numbers for k_p, k_a and k_b that satisfied the rules at slide#44. Then we can calculate v and w from alpha, beta, rho, k_p, k_a and k_b.

- **controller**

  This is the main method of part 3. It takes the robot's initial position, its destination and also, the wheels' radius and the length of the shaft between them, and moves the robot to reach the destination. Also, this function takes two empty lists to put the robot's steps on them. First of all, the method calculates the needed velocity by calling the "velocity_cal" method.

  This method returns the velocities in the robot's coordinate so then, we took it into the main coordinate by calling the "change_coordinates" method. Then, the velocity of each wheel is calculated by calling "inverse_cal". So now, we have the needed velocity to give it to the wheels. Then, we should call the "forward_cal" function to calculate the speed of the robot and reach the next position. Then we push this new position into those two lists and call this method again with new initial states.

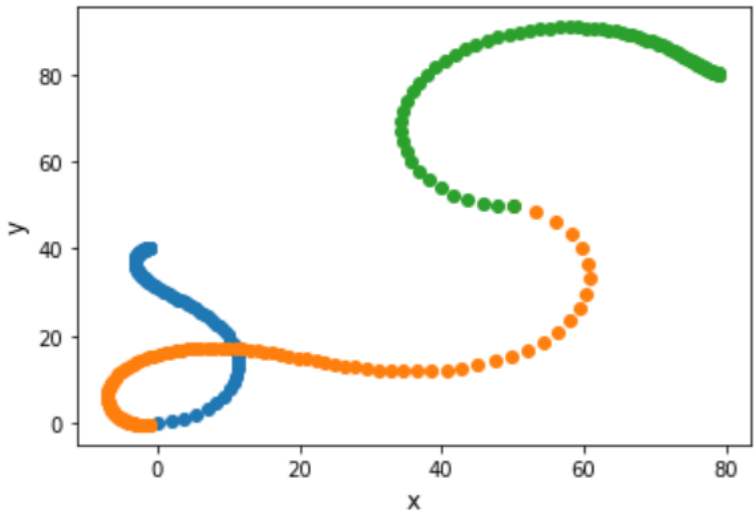
  This function is a recursive function that calls itself until the initial position becomes around the destination. To prevent some errors, we have taken a threshold for the distance between initial and destination's x and y.

Tests that be done to check the functionality of the code are as following:

1. We have tested the controller with three samples.

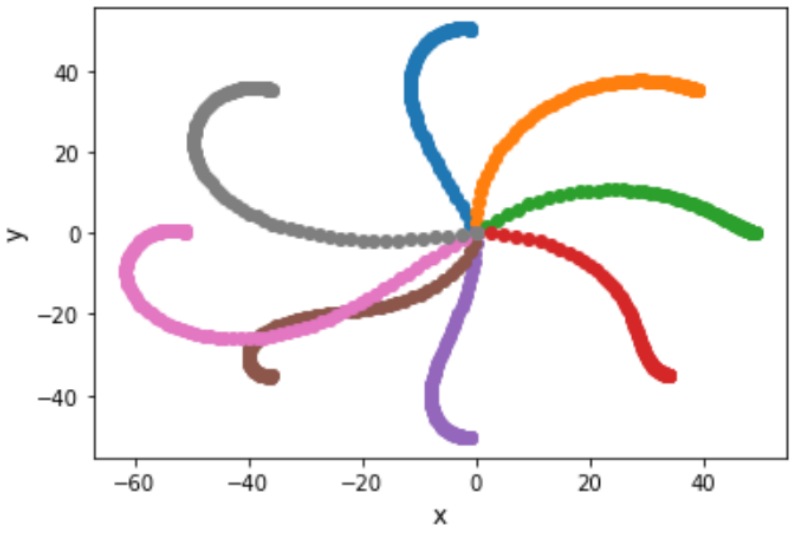
   For example, the orange plot indicates the robot's movement from point $(50, 50)$ with -25 degree for theta, to point $(0, 0)$. At first, the robot moves with large steps and as it gets closer to the destination, it takes shorter steps that

have been shown by the orange points. Indeed, as can be seen, the first points are far from each other and the last ones are quite close together. Also, since the robot wants to be with theta 0 degree at the destination, there is a great deviation in its movement.



2. Starting point in the middle and different destinations

   In this case, we gave $(0, 0)$ as the starting point to 8 robots with different destinations. Here is the result:



3. The destination point is in the middle and the starting points are different

   In this case, we gave $(0, 0)$ as the destinations to 8 robots with different starting points. Here is the result: