University of California, Riverside

EE/ME144/EE283A
Foundations of Robotics

Fall 2021

Lab 6 Report

Nov 17, 2021

Name	SID	Section	Group Number
Simraj Singh	862155747	Thursday	4
Shaheriar Malik	862154387	Thursday	4

1. Problem Statement

In Lab 6, we were tasked with successfully generating smooth trajectories using polynomial time scaling in simulation with the Turtlebot through Gazebo Simulator.

2. Design Idea

The main idea behind this lab was to implement time scaling of 3rd order polynomials and apply that for each segment of the trajectory in between waypoints. The waypoints were given to us in the code in the lab manual. We start off the move_to_point() function by determining boundary conditions for position and then velocity. Then we decompose the velocity in order to get the angle needed for the trajectory. After scaling the polynomial to a 3rd degree polynomial, we begin the login behind actually moving to the waypoints. We compute theta for each leg of the waypoint and compute the deviation between that theta and our current position. Using a PID controller, we send these updated values for velocity and angular z to the robot. In this case, we only set the Kp value, and the error is dt. Lastly, we update the previous waypoint and velocity to be the current, as the current values will be updated to the next. The previous waypoint and velocity values are initialized in the Turtlebot class outside these functions.

For the polynomial_time_scaling_3rd_order() function, we are given some position/velocity start and end values as well as the period, and use the matrix method introduced to us in the lab manual in order to compute the time scaling and return the corresponding vector. Finally, for the odom_callback() function, we get the position from the odometry topic and set the x, y, and theta values for position.

Using these two functions we are able to control the Turtlebot having it successfully reach the waypoints in a smooth trajectory.

```python
    def move_to_point(self, current_waypoint, next_waypoint):
        # generate polynomial trajectory and move to current_waypoint
        # determine boundary conditions for the position    -   boundary for x and y
        px_start = self.previous_waypoint[0]
        px_end = current_waypoint[0]

        py_start = self.previous_waypoint[1]
        py_end = current_waypoint[1]

        # determine boundary conditions for the velocity    -   velocity for vx and vy
        vx_start = self.previous_velocity[0]
        vy_start = self.previous_velocity[1]

        # dx = x1 - x0
        # decompose the velocity
        # vx = v ref * cos(angle)
        # vy = v ref * sin(angle)
        dx = next_waypoint[0] - current_waypoint[0]
        dy = next_waypoint[1] - current_waypoint[1]
        angle = atan2(dx,dy)

        vx_end = self.vel_ref*cos(angle)
        vy_end = self.vel_ref*sin(angle)

        T = 2

        # Tx = dx/(vx_end-vx_start)
        # Ty = dy/(vy_end-vy_start)
        # T = int(2*sqrt(dx**2 + dy**2)/0.3)

        ax = self.polynomial_time_scaling_3rd_order(px_start, vx_start, px_end, vx_end, T) #test diff T's
        ay = self.polynomial_time_scaling_3rd_order(py_start, vy_start, py_end, vy_end, T) #test diff T's

        # x(t) = a0 + (a1 * t) + (a2 * t**2) + (a3 * t**3)

        # v(t)= a1 + (2 * a2 * t) + (3 * a3 * t**2)

        ctrl = Controller()
        ctrl.setPD(5,0)
```

Figure 6.1.1: Code implemented for move_to_point function with comments

```
        for i in range(0,9*T):
            t = i*0.1
            vx_end = np.dot([3*(t**2), 2*t, 1, 0],ax)
            vy_end = np.dot([3*(t**2), 2*t, 1, 0],ay)
            # self.vel.linear.x = ax[2] + (2 * ax[1] * t) + (3 * ax[0] * t**2)
            # self.vel.linear.y = ay[2] + (2 * ay[1] * t) + (3 * ay[0] * t**2)
            #compute theta
            theta = atan2(vy_end,vx_end)
            dt = theta - self.pose.theta

            ctrl.setPoint(theta)
            if (dt > pi):
                self.vel.angular.z = ctrl.Kp*(dt-(2*pi))
            elif(dt < -pi):
                self.vel.angular.z = ctrl.Kp*(dt+(2*pi))
            else:
                self.vel.angular.z = ctrl.Kp*dt

            self.vel.linear.x = sqrt(vx_end**2 + vy_end**2)
            self.vel_pub.publish(self.vel)
            self.rate.sleep()
            #compute deviation bw theta and self.pose2D.theta   -   D_theta
            #update velocity
            # vel = Kp * delta theta (use high values of Kp)
            #use PID controller and use Vx_end and Vy_end to compute deviation and publish angular z
            #publish linear.x (magnitude of v) and angular.z (Kp * D_theta)
        self.previous_waypoint = current_waypoint
        self.previous_velocity = [vx_end,vy_end]
        pass
```

Figure 6.1.2: Rest of the code implementation for move_to_point function with comments

```
    def polynomial_time_scaling_3rd_order(self, p_start, v_start, p_end, v_end, T):
        # input: p,v: position and velocity of start/end point
        #        T: the desired time to complete this segment of trajectory (in second)
        # output: the coefficients of this polynomial
        x = np.array([p_start,p_end,v_start,v_end])
        M = np.array([[0,0,0,1],[T**3, T**2, T, 1],[0,0,1,0],[3*(T**2),2*T,1,0]])
        return np.dot(np.linalg.inv(M),x)
```

Figure 6.2: Code implemented for polynomial_time_scaling_3rd_order function with comments

```python
    def odom_callback(self, msg):
        # get pose = (x, y, theta) from odometry topic
        quarternion = [msg.pose.pose.orientation.x, msg.pose.pose.orientation.y,\
                    msg.pose.pose.orientation.z, msg.pose.pose.orientation.w]
        (roll, pitch, yaw) = tf.transformations.euler_from_quaternion(quarternion)
        self.pose.theta = yaw
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

        # logging once every 100 times (Gazebo runs at 1000Hz; we save it at 10Hz)
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])  # save trajectory
            rospy.loginfo("odom: x=" + str(self.pose.x) +\
                "; y=" + str(self.pose.y) + "; theta=" + str(yaw))

```

Figure 6.3: Code implemented for odom_callback function with comments

3. Results

Ultimately, we were able to successfully implement 3rd order polynomial time scaling in order to smoothen the trajectories of Turtlebot movement. By using test case values from the autograder, we were able to ensure our time scaling calculations were correct in comparison to the true values which were calculated using the matrix method discussed in lab. As seen by Figure 6.4, our trajectory plot matches that of the one we are supposed to emulate and does so with smooth movement as opposed to the sharp movement from Lab 3. We did not run into any troubles during this lab and learned quite a bit about how the robot moves.

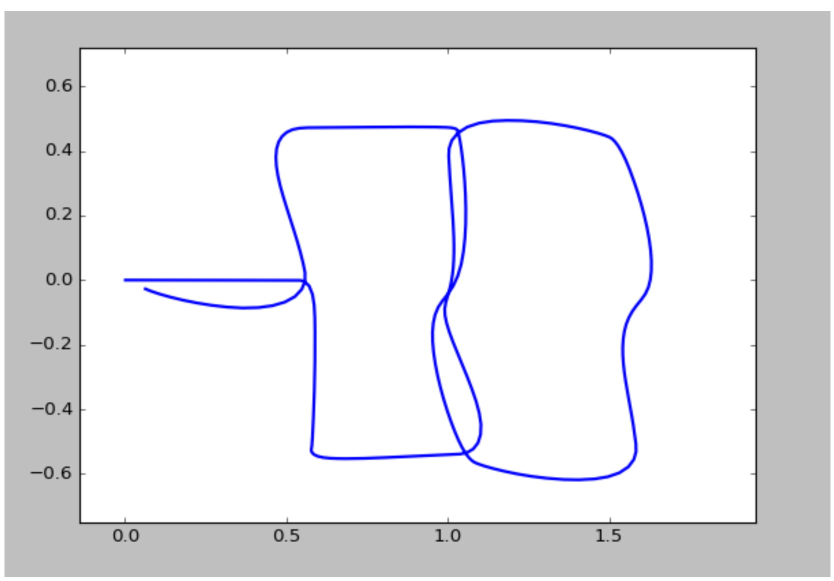

Figure 6.4: Visualization of the Turtlebot's trajectory with smooth movement

4. Appendix

How to run the code for lab 6:

- Launch Gazebo using:
    - `roslaunch ee144f21 gazebo.launch`
- Then run the test function using:
    - `roscd ee144f21/scripts`
    - `python trajectory_generation.py`
- To run the visualization script:
    - `roscd ee144f21/script`
    - `python visualizaiton.py`