Lab 2: Odometry

CSCI 3302: Introduction to Robotics

Goals

- Implement the forward kinematics of a differential wheel robot on the ePuck platform
- Experience and quantify sensor noise from wheel-slip, discretization, and timing violations
- Understand the concept of "loop closure"

Prerequisites

- A functional Webots development environment
- Lab 2 base environment folder [available on Canvas]

Submission Instructions

Each group must develop their own software implementation and turn in a lab report. **You are encouraged to engage with your lab partners for collaborative problem-solving**, but are expected to understand and be able to explain everything in your write-up. If your group does not finish the implementation by the end of the class, you may continue this lab on your own time as a complete group.

Please ensure exactly **one group member** submits the following in a .zip file to Canvas:

1) A PDF write-up answering each question from Part 4 (please put the number of each question next to your answers, rather than turning in your answers as an essay)
2) Your controller [.py file] for Lab 2.

Note – The questions in Parts 1-3 are to help with your understanding of the lab. You do not need to answer these in your lab report.

Overview

Odometry integrates the wheel speeds of your robot to calculate its 3-dimensional pose (x, y, θ) on the plane. To do this, you need to write down the forward kinematics of your robot. As your robot is non-holonomic, you need to calculate the *velocity* along the robot's x-axis and y-axis, as well as the rotational speed around its z-axis. You then need to transform these velocities into a global coordinate frame and integrate them to calculate the robot's pose. You will find out that you cannot do this without error as

the robot slips, time discretization introduces inaccuracies, and the robot's processor might not be fast enough to perform calculations while controlling its motors in real-time. (The latter is not an issue in Webots, but indeed a problem on small, miniature robots.)

Part 0: Loading Lab 2

To load the Lab 2 world, open Webots, go to File > Open World, navigate to where you extracted the CSCI3302_lab2 zip and select *CSCI3302_lab2.wbt* from the *worlds* directory.

Part 1: Measuring Wheel Speeds

1. Note the ePuck robot's pose at the Start Line on the floor, and record its x, z coordinates by looking at the "translation" field of the e-puck.

2. Make the ePuck go forward at full speed for a small fixed number of seconds (say 4). You might need to turn the robot around so that it can move in a long enough straight line.

3. Calculate the linear translation and estimate the speed in m/s.

4. Store the calculated speed value in the global variable EPUCK_MAX_WHEEL_SPEED. This will allow you to calculate the robot's speed in m/s without having to measure the wheel diameter. After you have completed this step, please comment out the portion that you used to calculate the speed if it's not needed for the next part.
   PERFORMANCE CHECK: Make sure you're getting a speed close to 0.13m/s.

Part 2: Odometry

1. Implement control code to cause the ePuck to follow the black line on the ground within the "line_follower" state. You should use +/- <motor>.getMaxVelocity() for motor velocity values or slower.
   a) If the center ground sensor detects the line, the robot should drive forward.
   b) If the left ground sensor detects the line, the robot should rotate *counterclockwise* in place.
   c) If the right ground sensor detects the line, the robot should rotate *clockwise* in place.
   d) Otherwise, if none of the ground sensors detect the line, rotate *counterclockwise* in place. (This will help the robot re-find the line when it overshoots on corners!)
   HINT: Keep track of the actual wheel speed that you set in a variable instead of sending it right to the motors.

2. Implement odometry code following Chapter 3 in the textbook (p. 67-74, Section 3.3.2). Calculate the actual distance traversed by multiplying the wheel speed with the time that the robot actually spent moving since the last odometry update. Ensure your pose is initialized (the vector [x, y, θ]) with (0,0,0) when the controller is initialized (e.g., before the "while robot.step…" loop). Use equation 3.41 to integrate your speeds into positions.
   Hint: The simulation time is fixed, there is no need to measure any time.

3. Print the robot's pose in the terminal window (don't change its format, otherwise the autograder will not work). What values do you expect to see when the robot arrives at the start line at the second (the third, the fourth) time? What actually happens?

Part 3: Line Following and Error Mitigation

1. Incorporate code to use the robot's ground sensors to identify the start line, as you did in Part 1.4.c. Implement a *loop closure* mechanism to use this information to prevent your odometry error from growing each lap.

   How could you exploit this information to reduce your error?
   HINT: If the ePuck has all 3 sensors below threshold for more than 0.1 seconds, that suggests you're passing over the start line! (Sometimes all three will momentarily pass under threshold on a sharp corner)

Part 4: Lab Report

Create a report that answers each of these questions:

1. What are the names of everyone in your lab group?
2. What happens (in terms of the robot's behavior) during the robot.step(TIME_STEP) statement?
3. What happens if your robot's time step is not exactly TIME_STEP long, but slightly varies?
4. What is the ePuck's average speed (in m/s) from Part 1?
5. In an ideal world, what should the ePuck's pose show each time it crosses the starting line?
6. How did you implement loop closure in your controller?
7. Roughly how much time did you spend programming this lab?
8. Does your implementation work as expected? If not, what problems do you encounter?