Lab 2: Odometry

CSCI 3302: Introduction to Robotics

Report due Tuesday 02/10/25 @ 11:59pm

Gavin: all highlighted items have already been completed.
Underligned items are just to underscore certain facts.
If you have any notes please also leave them in this doc, or in the google doc.

The goals of this lab are to

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

You need:

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

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 at least **one group member** submits the following 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 a robot's wheel speeds to estimate its 3-dimensional pose (x, y, θ) on a 2D plane. In this lab, you will derive the forward kinematics equations for your non-holonomic robot to calculate the velocity along its local x-axis and y-axis, as well as the rotational speed around its z-axis. You will then transform these velocities into a global coordinate frame and integrate them to calculate the robot's pose.
Note that odometry is prone to accumulating errors due to factors such as wheel slippage, inaccuracies introduced by time discretization, and latency in the robot's processor during real-time motor control.

Part 0: Loading the Environment

To load the Lab 2 world:
Open **Webots**.
Navigate to **File > Open World**.
Locate the extracted `S25_CSCI3302_Lab2` folder.
Enter the `Worlds` directory and select `CSCI3302_lab2.wbt`.

Part 1: Measuring Wheel Speeds

Our goal is to calculate the e-puck's speed within the XZ-plane. Before coding the controller, please modify the robot's translation field in the world file by reducing the z-coordinate to a smaller value; this provides sufficient distance for the robot to reach full speed before crossing the "Start Line." You are now ready to implement the controller logic.

1. **State Initialization:** Define a state variable at the top of your controller script and initialize it to`"speed_measurement"`.
2. **Speed Measurement Logic:** Within the `"speed_measurement"` state, implement the following behaviors:
    1. **Drive Forward:** Command the e-puck to move forward at max speed.
    2. **"Start Line" Detection:** Monitor the ground sensors to identify the exact moment the robot reaches "Start Line".
    3. **Stop:** Once the line is detected, stop the motors and update the state to `" line_follower"`.
    4. **Calculate Speed:** Calculate the linear translation distance and divide it by the elapsed time to estimate the robot's speed in m/s.

    Hint: The robot's `getTime()` function will be useful here.

3. Print the Elapsed time

4. **Store the Constant:** Save the calculated speed in the global variable `EPUCK_MAX_WHEEL_SPEED`. This allows you to utilize speed in m/s for future calculations without measuring wheel diameter.

Please verify that your calculated speed is approximately **0.13 m/s**.

Part 2: Odometry Implementation

Within the `"line_follower"` state, implement the control logic required for the e-puck to navigate the black line. Use `+/- <motor>.getMaxVelocity()` for setting motor velocities.

A. Line Following Logic

Implement the following behaviors based on ground sensors detection:
Center Sensor detects line, Drive forward.
Left Sensor detects line: Rotate counter-clockwise in place (set motors to opposite directions).
Right Sensor detects line: Rotate clockwise in place (set motors to opposite directions).
No Sensors Detect Line: Rotate counter-clockwise in place. (This behavior helps the robot re-acquire the line if it overshoots a corner).
Hint: Track the intended wheel speed in a variable rather than writing values directly to the motors immediately; this assists with the odometry calculations below.

B. Odometry Calculation

Implement odometry by following **Section 3.3.2** of the textbook.
Initialization: Ensure your pose vector `[x, y, theta]` is initialized to `(0,0,0)` when the controller starts (e.g., before the `while robot.step...` loop).
Calculation: Calculate the actual distance traversed by multiplying the wheel speed by the time the robot spent moving since the last odometry update.
Integration: Use the textbook subsection **"From Forward Kinematics to Odometry" in page 73** to integrate your speeds into global positions.
Hint: Because the simulation time step is fixed, you do not need to measure delta time dynamically. Read the comments “Hints for update_odometry” to have more details on how to calculate the delta time.

C. Output and Analysis

Print the robot’s pose to the terminal window. **Do not change the output format**, as this is required for the autograder.
Analysis: Observe the reported pose when the robot returns to the start line on the second, third, and fourth laps. What values do you expect to see versus what actually happens?

Part 3: Loop Closure

Implement a **loop closure** mechanism using this data to prevent odometry error from growing with each lap.

1. Use the ground sensor to detect the timing the robot passes “Start Line”.
   Hint: If all three sensors read below the threshold for more than **0.1 seconds**, the robot is likely passing over the start line. (Note: A duration check is necessary because all three sensors may momentarily dip below the threshold during sharp corners).
2. Once it passed “Start Line”, reset the robot’s pose `[x, y, theta] to (0,0,0).`

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 varies slightly?
4. What is the ePuck's average speed (in m/s) when covering the 10cm distance 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?