2.12: Introduction to Robotics
Lab 4: Mobile Robot Path Tracking using Odometry *

Spring 2023

Instructions:

1. **When your team is done with each task, call a TA to do your check-off.**
2. **Turn in your answers to the questions along with the screenshots of the robot following the trajectory with and without slippage in a single pdf file. No need to turn in your code.**

1 Introduction

Today's goals are to:

1. Estimate robot's change in pose $(\Delta x, \Delta y, \Delta\theta)$ using the encoders in the motors of the wheels.
2. Move the robot using delta pose $(\Delta x, \Delta y, \Delta\theta)$ commands.

To do so, we will apply the wheeled-robot kinematics from lecture. In this handout, we will denote the mobile robot platform as robot.

2 Setting up the code

Open a terminal window (Ctrl+Alt+T for Windows/Linux users and Ctrl+Opt+T for Mac users) and enter the following commands.

```
cd ~                                # note: make sure we are at home folder
git clone https://github.com/mit212/mobile_robot_2023.git
```

*

1. Version 1 - 2016: Peter Yu, Ryan Fish and Kamal Youcef-Toumi
2. Version 2 - 2017: Yingnan Cui and Kamal Youcef-Toumi
3. Version 3 - 2019: Jerry Ng
4. Version 4 - 2023: Joseph Ntaimo, Kentaro Barhydt, Ravi Tejwani and Kamal Youcef-Toumi

3 Instructions

1. Open the project with PlatformIO IDE in VScode.
2. Select `env:robot` before uploading the code.
3. To run the code on the joystick, joystick.cpp needs to be in the joystick folder, and make sure to select `env:joystick` before uploading the code.
4. For wireless communication:
   (a) Get the MAC address of the robot and put it in the joystick
      - Drag all the files in the `src/robot` folder out into `src/`
      - Drag `get_mac.cpp` into the robot folder
      - Set the environment to robot
      - Upload
      - Open Serial monitor
      - Use the address printed in the serial monitor to fill in the broadcast address in line 22 of the `src/joystick/joystick.cpp` file:

   (b) Get the MAC address of the joystick and put it in the robot (similar process to that above)
      - Drag all 5 robot files back into the robot folder
      - Drag all the files in the `src/joystick` folder out into `src/`
      - Drag `get_mac.cpp` into the joystick folder
      - Set the environment to joystick
      - Upload
      - Open Serial monitor
      - Use the address printed in the serial monitor to fill in the broadcast address in line 16 of the `src/robot/wireless.cpp` file:

4 Code modifications for the tasks

In this lab, you will need to complete five tasks (described in the section below). These are the functions that you need to modify following functions in `src/robot/pathPlanner.cpp`

1. setDesiredVel() : Convert the desired velocity in linear velocity and curvature, k, to angular velocities to send to the motor.
2. getSetPointTrajectory() : Change the velocity and k curvature setpoints to make the robot follow a trajectory.
3. updateRobotPose() : Use the odometry readings to update the current position of the robot relative to the world frame.

Few things to consider:

- Task 1 and 2: This is in the old lab that maps directly to filling out updateRobotPose(). Once this is done correctly, you should be able to leave the drivetrain off and push the robot around and get odometry readings that make sense. If you want to update the way the data is printed, change printOdometry() on line 162.
- Task 3 and 5: Robot trajectory tracking : This will be filling out getSetPointTrajectory(). You will need to actually set vel and k based on how much of the path they have traversed and/or how much theta has changed. Make sure to run setDesiredVel(vel,k) at the end of the function. Otherwise the velocity won't actually change.
- When running the robot, all of the data will be sent over to the remote and printed over the serial port if the remote's broadcast address was correctly put into the robot's `wireless.cpp` file. This should make it much easier to debug because students don't need to be directly connected to the robot as it's moving.
- If students want to get the robot working with the joystick again, all they have to do is comment getSetPointTrajectory() in the main loop() and switch to getSetPointJoystick(), where they have to figure out how to maps the `joyData.joyX` and `joyData.joyY` variables from their initial range of 0-1024 to the robot's desiredVelocity range of -30-30 rad/s.
- When running the robot, place it on the ground, turn on the drivetrain switch, then press reset on the microcontroller to make it move.

5 Task 1 & 2: Mobile Robot Odometry

Odometry is to estimate the change in robot pose over time using changes in encoder values. First, let's define some variables and constants:

- (x, y, θ): estimated robot pose using odometry relative to the world frame (starting frame),
- ϕ_R, ϕ_L: the right and left wheel net rotation (positive sign is in the robot forward direction),
- $b = 0.225$m, $r = 0.037$m, and $a = 0.3$m: robot dimensions as shown in Figure 1.

Motor 1 drives right wheel, and motor 2 drives left wheel.

To find the odometry, encoder values need to go through 2 conversions:

1. encoder count change to wheel position change;
2. wheel position change to robot pose change.

For conversion 1, you will implement it in `calc_dPhis(self, leftEnc, rightEnc)`. For conversion 2, you will implement it in `updatePose(self, dPhiL, dPhiR, Pose))`. Recall from the lectures that we can first compute $\dot{\theta}$ using the following expression.

$$\dot{\theta} = \frac{r}{2b}\left(\dot{\phi}_R - \dot{\phi}_L\right). \tag{1}$$

However, in our robot platform, we can only measure ϕ's at discrete timestamps. Thus we rewrite (1) as (2):

$$\Delta\theta = \frac{r}{2b}\left(\Delta\phi_R - \Delta\phi_L\right). \tag{2}$$

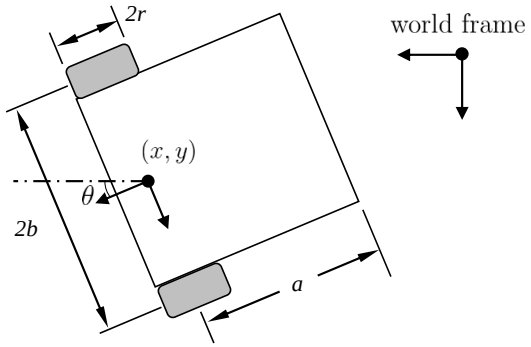

Figure 1: Definition of the robot dimensions and robot frame. Robot forward direction matches the x axis of the robot frame.

To estimate current robot orientation $\theta(t)$ we use:

$$\theta(t) = \theta(t - \mathrm{d}t) + \Delta\theta. \tag{3}$$

Now we can use $\theta(t)$ to compute the position of the robot.

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} x(t - \mathrm{d}t) \\ y(t - \mathrm{d}t) \end{bmatrix} + \frac{r}{2} \begin{bmatrix} \cos\theta(t) & \cos\theta(t) \\ \sin\theta(t) & \sin\theta(t) \end{bmatrix} \begin{bmatrix} \Delta\phi_R(t) \\ \Delta\phi_L(t) \end{bmatrix} \tag{4}$$

All relevant variables have been defined in `updatePose(self, dPhiL, dPhiR, Pose))` and are listed below. You do not need to define your own variable. The following variables are of `float` type.

Question 1 *Is this approximation accurate? What is a way to improve this approximation?*

- `X`, `Y`, `Th`: $x(t)$, $y(t)$, $\theta(t)$,
- `dX`, `dY`, `dTh`: $\dot{x}(t)$, $\dot{y}(t)$, $\dot{\theta}(t)$,
- `pathDistance`: keeps track the total distance traveled by your robot.

6 Task 3 & 4: Robot Trajectory Tracking

The task is to program a navigation policy to follow a U-shaped trajectory and stop at the goal position. An illustration of the track is shown in Figure 3.

To do so, we'll implement a policy in `callback(self, msg)` and a function `PathPlanner(self, robotVel, K)` in `planner.py`.

Here are three steps to complete this task:

1. Determine what stage of the track your robot is in. We suggest to write an `if/else` statement in `callback(self, msg)` function in `planner.py`. The U trajectory can be divided into 3 stages: 1) straight line, 2) semi-circle, and 3) straight line again.
2. Specify a robot velocity and motion curvature for each stage.

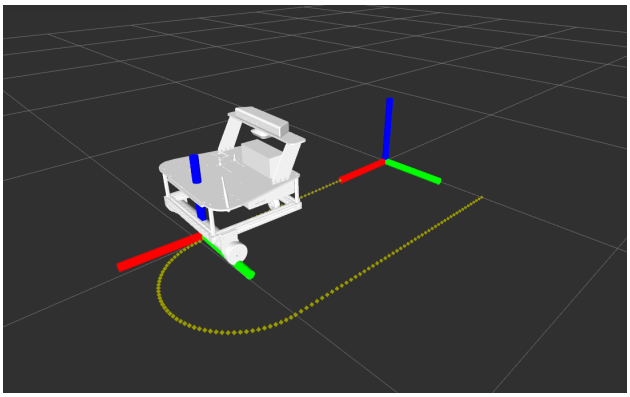

Figure 2: An example view of rviz showing the current robot coordinate frame, map (world) frame, and the U trajectory that we will be implement in the next task. The red, green, and blue bars of the frame correspond to x, y, and z axes respectively. The robot frame is defined at the center of the two motors and projected to the ground, with x axis pointing forward, y pointing to the left, and z pointing up. Initially, robot frame starts at the map frame.

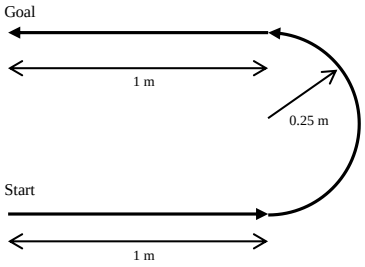

Figure 3: The U Trajectory.

3. Compute the desired R/L wheel velocities from a specific motion velocity and curvature. A function `PathPlanner(self, robotVel, K)` for this purpose has been declared for you in `planner.py`.

You need to complete this function using the following two equations.

$$\kappa = \frac{\dot{\phi}_R - \dot{\phi}_L}{b(\dot{\phi}_R + \dot{\phi}_L)}, \tag{5}$$

where κ is the curvature, the inverse of the radius of a circle, as shown in Figure 4.

$$\texttt{robotVel} = |(\dot{x}, \dot{y})| = r\frac{(\dot{\phi}_R + \dot{\phi}_L)}{2}. \tag{6}$$

Note that `desiredWV_L`$=r \cdot \dot{\phi}_L$ and `desiredWV_R`$=r \cdot \dot{\phi}_R$.

Question 2 *Given the curvature κ and the maximum wheel speed $\dot{\phi}_{max}$, what is the maximum* `robotVel` *that you can drive your robot at?*

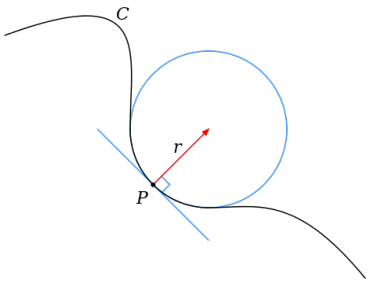

Figure 4: Curvature of a circle is defined to be the reciprocal of the radius. Image source: Wikipedia.

7 Task 5: The effect of slippage

You'll notice that the robot has been perfectly travelling along the desired trajectories. Would you expect this behaviour in the real world? A simple model for wheel slip has been introduced in `simulator.py`. Itroduce the slip by changing self.slip to TRUE. Re-run the code for the U-shaped trajectory in Task 4. What changed? Does the robot accurately follow the desired trajectory? Can you think of other effects that influence tracking?

References