Lab 2: Mobile Robot Path Tracking using Odometry
2.12: Introduction to Robotics
Fall 2016

Peter Yu, Ryan Fish, and Kamal Youcef-Toumi

Instructions:

1. **When your team is done with each task, call a TA to do your check-off.**
2. **After the lab, zip your files and make a backup using online storage/flash drive.**
3. **No need to turn in your code or answers to the questions in the handout.**

1 Introduction

In the previous lab we learned how to control the speed of the robot wheels. Today's goal is 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. The lessons you will learn from this lab are very crucial to navigating your robot on the final project arena. In this handout, we will denote the mobile robot platform as robot.

On the implementation side, this lab utilizes two essential software development tools: git [1] and ROS (Robot Operating System) [2]. Commands to use them are provided and we won't dive into details in this lab. You are encouraged to browse the referenced websites to learn about them. We will work with ROS more in the next lab. Also, this lab uses the object-oriented programming to structure the Arduino code. If you are not familiar with this, please read these two tutorials [3, 4].

2 Setting up the code

Open a terminal (Ctrl-Alt-T), and enter the following commands without the leading $.

```
$ cd ~                              # note: make sure we are at home folder
$ git clone https://github.com/mit212/me212lab2.git
$ cd me212lab2/catkin_ws
$ catkin_make                       # let ROS know where is our package
$ rebash                            # reload bash resource file, ~/.bashrc
```

From now on, we will denote the path to the `me212lab2` folder, `~/me212lab2`, as `LAB2`. Remember that `~` is an alias of user's home folder in Ubuntu, which is `/home/robot/` in our case.

2.1 Folders and files

The `me212lab2` folder contains all the files required for this lab. The overall structure follows the ROS catkin build system [5] and is shown below. Unimportant files and folders are ignored for brevity. However, don't delete them because they are still required for the package to work properly.

- `catkin_ws` : ROS catkin Workspace
  - `src` : Source space, where ROS packages are located
    - `me212_robot` : A folder containing a ROS package for the 2.12 moving platform

The folder hierarchy helps organize a large project that contains several *packages*. Package is a term for units of files related to one idea. The files can be code for libraries and programs, as well as config files. Now let's focus on the content inside package `me212_robot`, which is for our robot platform. Below some important files are listed.

- `src/controller` :
  - `controller.ino` : a controller in Arduino.
  - `helper.h` : helper for the controller. This file includes declaration of constants, variables and class `EncoderMeasurement`, `RobotPose`, `PIController`, `SerialComm`, and `PathPlanner`.
  - `helper.cpp` : implementation of the above classes.
- `scripts/me212_robot.py` : read odometry from Arduino and publish them to ROS network on PC.
- `launch/viz.launch` : a ROS launch file that launches tools to visualize the above messages.

In this lab, you only need to modify `helper.cpp` and `controller.ino`, and refer to `helper.h` for declarations.

2.2 High level code structure

In `controller.ino`, the main `loop` function has the following sections:

1. Obtain and convert encoder measurement.
2. **Compute robot odometry.**
3. Send robot odometry through serial port.
4. **Compute desired wheel velocity with motion planner.**
5. Command desired velocity with PI controller.

Item 1 and 5 were done in Lab 1, and example code for them are provided as `class EncoderMeasurement`, and `class PIController`. You should read their implementation in `helper.cpp`. You may change the gain of the controller and make sure that positive direction of encoder and motor corresponds to forward wheel motion. Item 3 is also provided in `class SerialComm`. Item 2 and 4 are what you are going to complete today.

3 Task 1: 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.

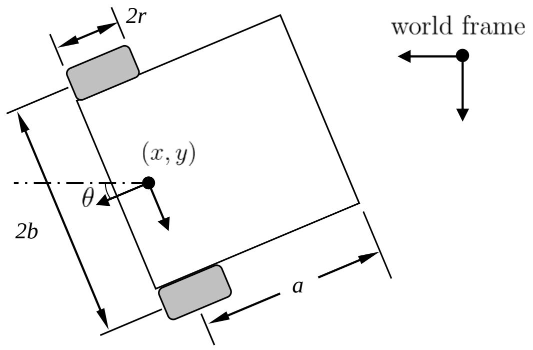

Figure 1: Definition of the robot dimensions and robot frame. Robot forward direction matches the x axis of the robot frame.

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.

Conversion 1 was done in Lab 1, and the code is provided in `class EncoderMeasurement`. However, make sure to check which motor model (26 or 53) your robot has, and configure it correctly when creating an `EncoderMeasurement` object in `controller.ino`. For conversion 2, you will implement it in `class RobotPose`. 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):

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

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 `class RobotPose` and are listed below. You do not need to define your own variable. The following variables are of `float` type.

- `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.

In order to test your system, the wheel velocities are set to some constants in section 4 of `controller.ino`.

```
pathPlanner.desiredWV_R = 0.2;
pathPlanner.desiredWV_L = 0.2;
```

Use the ROS visualization tool, `rviz`, to visualize your robot pose from odometry. Open a terminal (Ctrl-Alt-T), and enter the following commands without the leading $.

```
$ roslaunch me212_robot viz.launch
```

This session should stay open.

Open another terminal; you can use Ctrl-Shift-O to split the existing terminal horizontally, or use Ctrl-Shift-T to open a terminal in a new tab. This helps your terminals to stay organized. And enter the following commands.

```
$ rosrun me212_robot me212_robot.py
```

This will connect to Arduino through serial communication and restart the Arduino program. The program can be stopped by Ctrl-C. Note that every initiation of serial communication will restart the Arduino program. Then you should see the robot moving in `rviz` as in Figure 2.

4 Task 2: 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 `PathPlanner::navigateTrajU(const RobotPose& robotPose)`. 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 on `robotPose.pathDistance`. 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.
3. Compute the desired R/L wheel velocities from a specific motion velocity and curvature. It is up to you to ensure that the resulting motor velocities do not exceed the maximum speed of the motors. A function for this purpose has been declared for you:

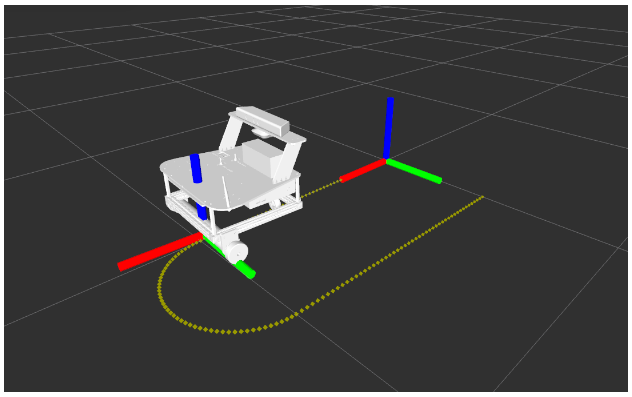

Figure 2: An example view of rviz showing the current robot coordinate frame, map (world) frame, and the U trajectory that we will be implement in the next task. The red, green, and blue bars of the frame correspond to x, y, and z axes respectively. The robot frame is defined at the center of the two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

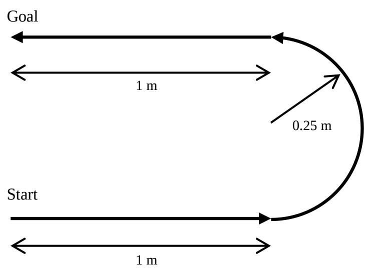

Figure 3: The U Trajectory.

```
//Input:  desired robot velocity robotVel and curvature K
//Output: update PathPlanner::desiredWV_L and desiredWV_R
void PathPlanner::updateDesiredV(float robotVel, float K);
```

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}`$=r \cdot \dot{\phi}_{\{L/R\}}$, and that you need to change `usePathPlanner` to `true` in `controller.ino` to test your navigation policy. For testing, if you just want to do dry run, make sure the robot wheels are off the table. If you want it to move on the floor, make sure there is no external wires connected to the robot. Plug the HDMI cable to the onboard screen so that you can still operate it. Hold the robot by the 80/20 frame to move it.

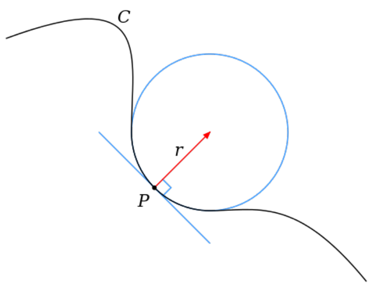

Figure 4: Curvature of a circle is defined to be the reciprocal of the radius. Image source: Wikipedia.

Question 1 *Given the curvature* κ *and the maximum wheel speed* `maxMV`*, what is the maximum* `robotVel` *that you can drive your robot at?*

5 Task 3: More Complex Trajectories

You have implemented simple continuous trajectory control but more complex tasks require various combinations of straight and curved motion. Try implementing more challenging trajectories such as a figure 8 or a three-point turn (Figure 5). You could have a new navigation policy for a figure 8 implemented in function `PathPlanner::navigateTraj8(const RobotPose& robotPose)`.

Question 2 *Can your robot start and end at the same pose for a figure 8 trajectory?*

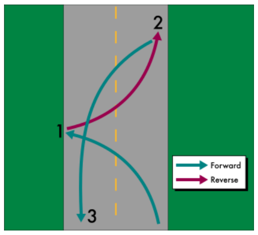

Figure 5: A three-point turn is for turning a vehicle around in a narrow space. Image source: Wikipedia.

References

[1] Website of Git. [Online]. Available: https://git-scm.com/

[2] Website of ROS. [Online]. Available: http://www.ros.org/about

[3] Tutorial of C++ classes. [Online]. Available: http://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

[4] Tutorial of C++ class member function. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm

[5] Website of ROS/catkin. [Online]. Available: http://wiki.ros.org/catkin/conceptual_overview