Homework 3

ME 416 - Prof. Tron

2024-03-21

The main goal of this homework is to analytically derive and implement in ROS the concepts we saw in class about rigid body transformations and modeling the motion of robots. In particular, you will see first hand the effect of modeling errors and calibration of the model.

Review of the dynamical model for ROSBot

As mentioned in class, the state of our robot can be represented with a 2-D pose (R, T) (expressed as an Euclidean transformation from the body to the world reference frames), which can be parametrized with a minimum number of three variables: x and y for the translation, and θ for the 2-D rotation angle. These variables can be combined in the *state*

$$z = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}. \tag{1}$$

From class, we saw that the motion of the robot can be modeled with the following dynamical system:

$$A\big(\theta(t)\big) = \frac{k}{2}\begin{bmatrix} \cos\big(\theta(t)\big) & \cos\big(\theta(t)\big) \\ \sin\big(\theta(t)\big) & \sin\big(\theta(t)\big) \\ -\frac{1}{d} & \frac{1}{d} \end{bmatrix}, \; u = \begin{bmatrix} u_{\mathrm{LW}} \\ u_{\mathrm{RW}} \end{bmatrix}, \tag{2}$$

$$\dot z(t) = A\big(\theta(t)\big)u \tag{3}$$

where $u_{\mathrm{LW}}, u_{\mathrm{RW}}$ are, respectively, the left and right commanded motor speeds (between -1 and 1, in our implementation), $A(\theta)$ is a matrix in $\mathbb{R}^{3\times 2}$, and k, d are two coefficients that combine physical characteristics of the motors and the robot.

Problem 1: Improved `twist_to_speeds`

Goal Update `twist_to_speeds`( ) from HW1 with a kinematically correct model, and also see first-hand how hard it is to have a robot to consistently reproduce a given motion without the use of feedback control.

From class, we have seen that the linear velocity for a differential drive robot is always aligned with the robot's body-fixed x-axis (which, in world coordinates, is represented by the vector $\begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$). Moreover, from the definition of θ, the magnitude of the angular velocity is simply given by $\dot{\theta}$. Letting `speed_linear` and `speed_angular` denote the magnitude of the linear and angular velocities, respectively (see also 1), we have that:

$$\dot{z} = \begin{bmatrix} \texttt{speed\_linear} \cdot \cos(\theta) \\ \texttt{speed\_linear} \cdot \sin(\theta) \\ \texttt{speed\_angular} \end{bmatrix}. \tag{4}$$

Question report 1.1. Combine (3) and (4), and obtain an expression for the left and right motor speeds, u_{LW} and u_{RW}, as a function of `speed_linear`, `speed_angular`, and the system parameters k, d. Describe, at an intuitive level, what is the effect of changing k and d on the linear/turning speed (consider all four possible combinations of increasing or decreasing each of the two variables simultaneously).

Question code 1.1. Modify the function `twist_to_speeds`( ) in `motor_command_model.py` from Homework 1 with the expressions obtained in the previous question. Use the function `model_parameters`( ) from a previous in-class assignment to obtain the values of k, d when you need them. **Note:** Make sure to add instructions to limit each of the output speeds between `-1.0` and `1.0`, independently from any input.

1) Make sure that there are no loose screws, that the motors and wheels are solidly attached, that the wheels are straight, and that the battery is placed to achieve an approximately uniform weight distribution on the two wheels.

2) From the previous homework, when you use `key_op.py` and drive straight, the vehicle should go straight; if not, please refer to the previous homework.

3) Draw or tape a square with a side length of about 2 ft on a floor surface where the robot does not slip easily; **avoid carpeted surfaces or surfaces that are uneven**; it might be useful to place the robot on a large piece of paper or cardboard to improve friction.

Question video 1.1. As mentioned in class, the constants k and d in the model are lumped parameters that summarize the physical characteristics and dimensions of the robot. In practice, we will tune the constants k and d in the `model_parameters`( ) experimentally by using the `scripted_op.py` node from Homework 2 and following the steps below.

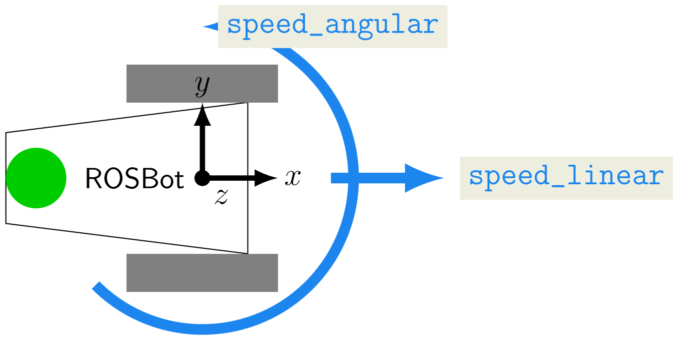

Figure 1: Axes on ROSBot to define linear and angular velocities.

1) Tune k: let the robot complete a few segments: if most of the length of most of the segments is less (respectively, more) than 1 ft, increase (respectively, decrease) k; note that not all the segments might be the same, this is normal.

2) Tune d: let the robot complete a few turns: if most of the turns are less (respectively, more) than $90°$, increase (respectively, decrease) d; note that not all the turns might be the same, that is normal.

Note that, if you change k again, this will affect the angular speed as well, so you will have to adjust d correspondingly (see the relation you derived in Question report 1.1). After tuning, submit a video showing the robot performing at least one complete loop with four turns. Showing less turns will not give you full credits for this question.

Note: do not worry if you cannot track the square precisely (errors up to around $\frac{1}{3}$ of a square per segment are fine): you will find that it is hard to get repeateable motions with our hardware, and this is normal and should be expected, unless you have high quality components. The practical solution to this is to use feedback control, which we will consider in the last assignment. For this assignment, you will be graded on the effort you show rather than the final trajectory.

Question report 1.2. Report *all* the values you tried for k and d.

Problem 2: Utility class `StampedMsgRegister`

Goal Learn how to look up information about specific classes using IPython and docstrings. Make a class that can save a previously received message, and compute the delay between subsequent messages.

This class will be used in a couple of future homework assignments.

Question report 2.1. Open `ipython3` in a terminal. Import `rclpy`, then type `from rclpy.node import Node`, `rclpy.init()`, `Node('a')`, and finally `rclpy.time.Time.seconds_nanoseconds?` (including the question mark[1]); this will show the docstring for that method. Include a screenshot of all the output, and explain what the method `seconds_nanoseconds`(` `) does for an object of type `rclpy.time.Time`.

Question provided 2.1. A utility function that computes the difference between two message stamps.

File name: `me416_utilities.py`

[1]This object cannot be accessed before creating a `Node`, hence the sequence of commands.

Function name: `stamp_difference`
Description: Transform the message stamps into Time objects, take the difference to get a Duration, and then convert the Duration in seconds as a float
Input arguments

- `stamp2` (type `builtin_interfaces.msg.Time`), `stamp1` (type `builtin_interfaces.msg.T` two message stamps (i.e., the filed `stamp` from the field `header` of a stamped message). Note that the first argument (`stamp2`) is assumed to be newer than the second argument (`stamp1`).

Output arguments

- `difference` (type `float`): the difference between the two time stamps expressed in seconds.

Question code 2.1. Implement the following class, which simplifies the process of computing the delay between two stamped messages.

File name: `robot_model.py`
Class name: `StampedMsgRegister`
Description: Computes the delay, in seconds, between two ROS messages.

Method name: `__init__`
Description: Initialize the internal variables `msg_previous` to `None`.

Method name: `replace_and_compute_delay`
Description: Given a new stamped message as input, computes the delay (in seconds) between the time stamp of this message and the value in `time_previous`, and then replaces the internal copy of the previous message with the current message.
Input arguments

- `msg`: any stamped ROS message (i.e., that has the `header` attribute).

Return arguments

- `time_delay`: the difference between the stamp time of `msg` and `msg_previous`. If `msg_previous` is `None`, return `None` instead. Note: you should use Question provided 2.1 in this function.
- `msg_previous`: a copy of `msg` from the previous call.

Note: before returning, the function replaces `msg_previous` with `msg`. **However, you should still return the message from the previous call (this means you will need a temporary variable for this).**

> Method name: `previous_stamp`
>
> Description: Returns the time stamp of `msg_previous`.
>
> Input arguments
>
> - No input arguments.
>
> Return arguments
>
> - `stamp` (type `rclpy.time.Time`): Return `msg_previous.header.stamp`, or `None` if `msg_previous` is `None`.

Question **report** **2.2.** Discuss the logic you used to compute `time_delay`, incorporating code snippets as necessary.

The idea behind the `StampedMsgRegister` class is that, after creating an object, say, `mdelay=StampedMsgRegister()`, you can invoke `mdelay.replace_and_compute_delay(msg)` in a callback to get the delay between two messages received at two different times by that callback. The first time the callback is called by ROS, you will obtain `None` (since there was no previous message). But the second time, you will get the delay (in seconds) between the second and first calls, plus the value of the old message.

Hint for question report 1.1: The expression you obtained should not depend on θ; if it does, it is wrong. For the dependency on k and d, see what happens when both are multiplied by the same amount.