COMP7308: Introduction to unmanned systems (2024 Fall)

Assignment 1: Simulation of Robot Dynamics

:

1 Introduction

In this project, you will start to implement the motion and related functions of a robot. The code for this project includes the following files, available on the Moodle.

Files you will edit and submit:

`/src/two_wheeled_robot.py`: The location where you will fill in portions. **Set variable `student_id` to your ID.** When you finish, you should submit **ONLY** this file without zipping or changing its name.

Files you should NOT edit:

`/src/two_wheeled_robot_qX.yaml`: Information of the simulation environment.

`/hint/`: Some reference results you can utilise to compare with yours.

`/test/student_grader.py`: Project autograder of student version. Pass it can make sure that you get **most of** the grades.

Evaluation

Your code will be autograded for technical correctness. Please **DO NOT** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Academic Dishonesty

We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. Instead, contact the TA if you are having trouble.

Prerequisite

To install the simulation package, please check Assignment 0. You should be familiar with `numpy` and we name it `np` in our project.

2 Question: Motion of two-wheeled robot (100 points)

The structure of a two-wheeled robot is shown in Figure 1. Its continuous motion model is:

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix} v\cos\theta \\ v\sin\theta \\ \omega_0 \end{pmatrix} = \begin{pmatrix} \frac{r\omega_1+r\omega_2}{2}\cos\theta \\ \frac{r\omega_1+r\omega_2}{2}\sin\theta \\ \frac{r\omega_1-r\omega_2}{2l} \end{pmatrix}.$$

Because simulation using robot is discrete, we need to **discretise** this motion model, and one popular way is as follows:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos(\omega_0 \Delta t) & -\sin(\omega_0 \Delta t) & 0 \\ \sin(\omega_0 \Delta t) & \cos(\omega_0 \Delta t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x - \mathrm{ICR}_x \\ y - \mathrm{ICR}_y \\ \theta \end{pmatrix} + \begin{pmatrix} \mathrm{ICR}_x \\ \mathrm{ICR}_y \\ \omega_0 \Delta t \end{pmatrix}. \tag{1}$$

Here $\texttt{state} = (x, y, \theta)^{\mathrm{T}}$ represents the state at current time t and $\texttt{next\_state} = (x', y', \theta')^{\mathrm{T}}$ means the state at next time $t + \Delta t$. The related parameters are defined below:

l: half of the robot length. Here $l = 0.25\,\mathrm{m}$.

r: robot's wheel radius. Here $r = 1\,\mathrm{m}$.

r_{ICR}: the rotation radius of the robot. $r_{\mathrm{ICR}} = \rho - l$.

ICR: the rotation centre of the robot, and its coordinate can be computed as $\mathrm{ICR} = \begin{pmatrix} x + r_{\mathrm{ICR}} \sin\theta \\ y - r_{\mathrm{ICR}} \cos\theta \end{pmatrix}$.

ω_1, ω_2: the rotation rate of the right and left wheels.

ω_0: the rotation rate. $\omega_0 = \dfrac{(\omega_1 - \omega_2) r}{2l}$.

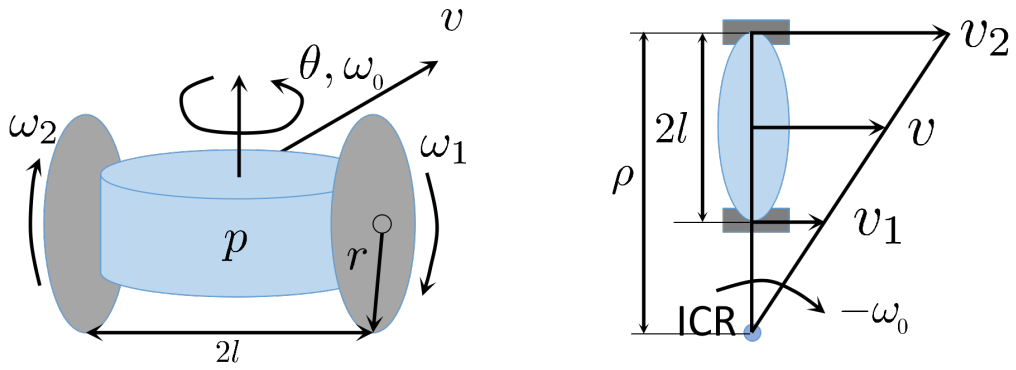

Figure 1: The model of a two-wheeled robot.

1. Robot dynamics (40 points)

Noticing that $(-\omega_0) = v_2/\rho = v/(\rho - l) = v_1/(\rho - 2l)$, $v_1 = \omega_1 r$, $v_2 = \omega_2 r$, find the expression of r_{ICR} related to (l, ω_1, ω_2). Think about all the cases that will trigger Python exceptions and the corresponding reasons. Under these cases, consider how robot will move and design the motion equations by yourself.

Then you can implement the motion model within the function `dynamics_without_noise()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q1`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q1 --graphics`, and compare it with the result `./hint/q1.gif` and `./hint/q1.png`..

Note:

1. Please check the comment within each function for variables' details and the requirement.
2. Make sure that the shape of each variable, either given by us or defined yourself, meets requirements. For `numpy` array `vel`, you can print it by `vel.shape` during debugging.

2. Robot dynamics with noise (40 points)
Here you should implement the motion model with some noise.

(a) (20 points) **After** the motion function, add the Gaussian addictive noise $\epsilon = \mathrm{N}(0, R)$ to the next state, i.e., next_state $\leftarrow$ next_state $+ \epsilon$.
Implement it within the function `dynamics_with_linear_noise()`.

(b) (20 points) **Before** the motion function, add the Gaussian noise $\epsilon^{\omega} = \mathrm{N}(0, R^{\omega})$ and $\epsilon_t^{\theta} = N(0, R^{\theta})$ to (ω_1, ω_2) and θ, respectively, i.e., $(\omega_1, \omega_2) \leftarrow (\omega_1, \omega_2) + \epsilon^{\omega}$ and $\theta \leftarrow \theta + \epsilon^{\theta}$. Here we suppose the noise for ω_1 and ω_2 is the same within each time step. Implement it within the function `dynamics_with_nonlinear_noise()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q2`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q2 --graphics`, and compare it with the result `./hint/q2.png`. You will see the banana-shape distribution of the final positions after the 100-time simulation.

Note:

1. The shapes of the noises are a bit different than those we discussed within class, because we only save the diagonal elements here.
2. The shape of the noise in (a) is different than that in (b). Please read the comments carefully within the code.
3. You can use the function `dynamics_without_noise()` to simplify your implementation.

3. Define control policy for specific task (20 points)

A two-wheeled robot starts at state $x = 0.5\,\mathrm{m}$, $y = 2.0\,\mathrm{m}$ and with heading $\theta = \frac{\pi}{2}$. It aims at moving to the goal state $x = 4.5\,\mathrm{m}$, $y = 2.0\,\mathrm{m}$, $\theta = \frac{\pi}{2}$ (all angles in radians, plotted in red, **goal heading direction should also be satisfied**). Find one solution $[(\omega_1^1, \omega_2^1, t^1), (\omega_1^2, \omega_2^2, t^2), ...]$ meets the following requirements **at the same time**, where ω_1^k, ω_2^k represent angular velocities of the right and left wheels, t^k means the duration for keeping these angular velocities:

1. (5 points) It contains the minimal number of steering commands, (ω_1, ω_2, t). We will not check its correction in `student_grader.py`.
2. (10 points) Under the first condition, the trajectory of the robot is the shortest (there are many trajectories satisfying this requirement). The accepted relative error of trajectory length is 5%.
3. (5 points) You should also return an answer of the shortest trajectory length calculated by hand. The accepted relative error is 1%. In `student_grader.py`, we will not check its correction, but it will be utilised to evaluate the correction of your solution. So make sure that this answer is right.
4. For the total controlling time $\sum_k t^k$, it should be less than 10s (but the time is sufficient). Considering that the simulation is a discrete process and the step time Δt here is 0.01 s, **the decimal of t^k should be less than two**, so that t^k can be the integral multiple of the step time Δt.
5. For the steering instructions ω_1^k, ω_2^k, the absolute value should not exceed $2\,\mathrm{rad/s}$. Please make the robot move slower for a more precise control. After the robot arrives at the goal (goal threshold: 5%), it will stop. We check the length of the trajectory only after the robot arrives at the goal point.

Finish you policy within the function `policy()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q3`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q3 --graphics`.

Note:

1. According to the discretised motion model, we know that given a fixed steering command, the robot is rotating around the ICR.
2. This question is very simple. Finding one feasible controlling instructions is enough. We do not need any derivations in your code.
3. One trick is making your instructions exist a bit longer to make sure that the robot can move over the goal point. Robot will automatically stop at the goal when the relative error meets requirements. The path after that will not take into account.