Introduction to Robotics
Assignment 1

Course lead: Dr. Amanda Prorok

Michaelmas Term, 2021-2022

1 Assignment 1 (100 points)

For this assignment, you will hand in a zip file named `A1_CRSID.zip` that contains your report (PDF file) as well as your code. For this set of exercises, you will review concepts from *(1)* motion control and dynamical systems, and *(2)* perception and localization.

1.1 Dynamical Systems and Motion Control

Exercise 1 (30 points + 5 bonus points)
In this exercise, you will implement your very own kinematic simulator. This simulator will simulate a differential-drive wheeled robot translating with constant forwards velocity of $u = 0.25$ [m/s], and steering with a rotational velocity $\omega = \cos(t)$ [rad/s]. Your work in this exercise will illustrate some of the issues one encounters in simulating the real world, and will demonstrate why we need good robotic simulators.

Open the file `~/catkin_ws/src/exercises/part1/python/kinematics.py`. To run the simulator, open a terminal, navigate to `~/catkin_ws/src/exercises/part1/python` and type `python kinematics.py --help`.

(a) **[5 points]** Write down the differential equations that correspond to the motion of the differential-wheeled robot using u for the forwards velocity and ω for the rotational velocity.

(b) **[5 points]** The current simulator includes all the necessary code to display the robot's progress. However, if you run it using `python kinematics.py --dt 0.1 --animate`, you will notice that nothing happens: the forward kinematics are missing. To fix this, implement Euler's method in the `euler` function. This function should return the next pose of your robot at time $t + \mathrm{dt}$ knowing the current time t (i.e., the step-size is dt). Briefly explain the concept of your solution.

(c) **[5 points]** Run the simulator with `python kinematics.py --animate --dt=1.,.5,.1,.01`. This generates a plot, which you should include in your answer. Describe what happens. Explain the advantages / disadvantages of choosing a large vs. a small step-size.

(d) **[5 points]** Euler is a first-order method, while classical Runge-Kutta (RK4) is a fourth-order method. Implement RK4 in the `rk4` function. Briefly explain the concept of your solution.

(e) **[5 points]** Run the simulator with `python kinematics.py --method=rk4 --dt=1.,.5,.1,.01`. This generates a plot, which you should include in your answer. Describe the differences between RK4 and Euler, and why these differences exist. Explain under which conditions it is preferable to use Euler.

(f) **[5 points]** Instead of setting $\omega = \cos(t)$, use $\omega = \cos(\lfloor t \rfloor)$ where $\lfloor \cdot \rfloor$ represents the floor operation. This simulates a perception-action loop running at 1Hz. Run the simulator with Euler and RK4 with different step-sizes. Include your plots, explain what happens, and why.

(g) **[5 bonus points]** So far, we have used a *fixed* step-size. What might be the issue with this, for a given perception-action loop duration? If you know in advance that the fixed perception-action loop takes 1 second, can you develop an adaptive solution, to avoid some of the issues exacerbated in question (f)? Make a copy of the `kinematics.py` file as `kinematics_bonus.py` and implement a fix for both Euler and RK4. Explain the concept of your solution.

Hand in your solution `kinematics.py` (and `kinematics_bonus.py` if you did the bonus question (g)) as well as your answers to (a)-(f). Do not exceed 200 words per answer.

Exercise 2 (26 points + 0 bonus points)

In this exercise, we are going to implement two obstacle avoidance controllers using ROS and Gazebo. The main file is located at `~/catkin_ws/src/exercises/part1/ros/obstacle_avoidance.py`.

(a) **[8 points]** Implement a Braitenberg controller in the `braitenberg` function. The controller should make use of a smooth function between control inputs and sensory observations, and should not use `if-else` statements. To run your controller, you will need to start Gazebo first with `roslaunch exercises gazebo_simple.launch` and then run your controller with `roslaunch exercises obstacle_avoidance.launch mode:=braitenberg`. You do not need to restart Gazebo when relaunching your controller, you can revert to the original state by using the "Edit→Reset World" menu. Once you are happy with your controller, let it run for roughly 20-30 seconds, pause Gazebo, and plot your robot's trajectory with the `plot_trajectory.py` script in the same folder. Include this figure with your report, and explain the concept of your solution.

(b) **[8 points]** Implement a rule-based controller (i.e., with `if-else` statements) in the `rule_based` function. To run your controller, type:
`roslaunch exercises obstacle_avoidance.launch mode:=rule_based`. Again, plot your robot's trajectory and explain the concept of your solution.

(c) **[5 points]** For both controllers, modify the environment (adding obstacles within Gazebo) while the robot is moving around. Compare the performance of the two controllers (e.g., with respect to robustness).

(d) **[5 points]** Extend your answer to the previous question in the case where the robot's sensors are perfect (i.e., they can sense the environment without added noise). What are the advantages / disadvantages of the two approaches in both cases?

Hand in your solution `obstacle_avoidance.py` along with the resulting trajectories as well as your answers to (a)-(d). Do not exceed 200 words per answer.

1.2 Localization

Exercise 3 (44 points + 0 bonus points)

In this exercise, we are going to implement a particle filter. The main file is located at `~/catkin_ws/src/exercises/part1/ros/localization.py`. We are going to assume that we have a map and know the location of all obstacles. In particular, we will use the same map as in the previous exercise, which contains 4 walls that delimit a 4×4 [m^2] area centered at $[0, 0]$, and a cylinder at position $[0.3, 0.2]$ of radius 0.3 [m].

(a) **[3 points]** Implement a Braitenberg controller in the `braitenberg` function. You can copy-paste the code that you developed in the previous exercise (you can also implement a new controller as long as it avoids obstacles). Start Gazebo with:
`roslaunch exercises gazebo_simple.launch` and run your controller with:
`roslaunch exercises localization.launch`.
In the report, include screenshots of your setup.

(b) **[5 points]** Implement the function `Particle.__init__` which initializes a particle pose randomly in the arena and sets its weight (or importance factor) to 1. Briefly explain the concept of your solution.

(c) **[8 points]** Implement `Particle.move` which applies the motion model to a particle. Start Gazebo with `roslaunch exercises gazebo_simple.launch` and then run your controller with `roslaunch exercises localization.launch`. Start the simulation by hitting the play button. This is not very informative as we do not see our particles. To that end, start RViz with `roslaunch exercises rviz.launch`, you should see the point cloud from the laser-range finder as well as the robot. We are going to add a visualization for our particles. To do so, click the "Add" button and create a visualization "by topic". From the topic list, add the `/particles` topic as a `PointCloud` (if you are curious about how we export the particle positions, read the function `main` in `localization.py`). You can change additional visualization options from the left pane. In particular, we recommend disabling the "Use rainbow" option and setting the "Min Color" to white. See Figure 1 for details. Now you should be able to see particles roaming around our environment. Pause Gazebo, reset the world ("Edit→Reset World"). Do not close neither Gazebo nor RViz.
Explain the concept of your solution and include images of your setup.

(d) **[8 points]** Implement the function `Particle.compute_weight` which applies the observation model. Restart your script with `roslaunch exercises localization.launch` and hit the play button in Gazebo. The particles should now aggregate and create clusters. Improve your particle filter by tuning your code. Beware of *particle collapse*. Explain the concept of your solution.

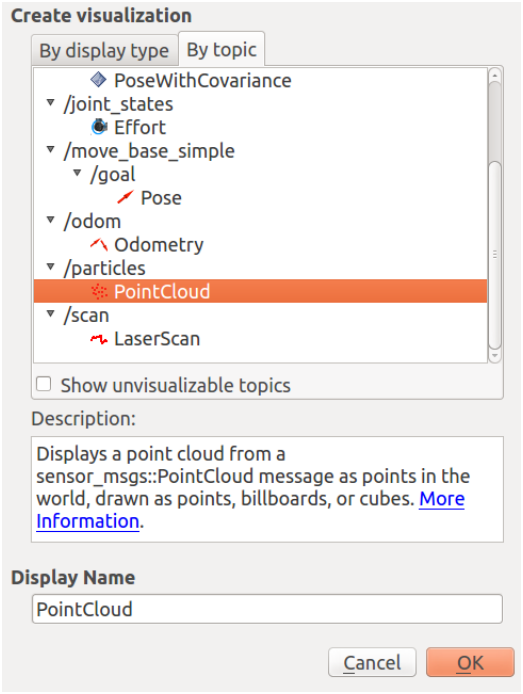
(e) **[5 points]** Restart the experiment a few times. Does the localization always succeed? When it succeeds, does it converge quickly to the correct position? Explain the reasons for unsuccessful localization or slow convergence.

(f) **[5 points]** Try to *kidnap* the robot by pausing the simulation, moving the robot manually. Is the localization able to succeed? Explain which parameters to tune to improve the robustness of the particle filter.

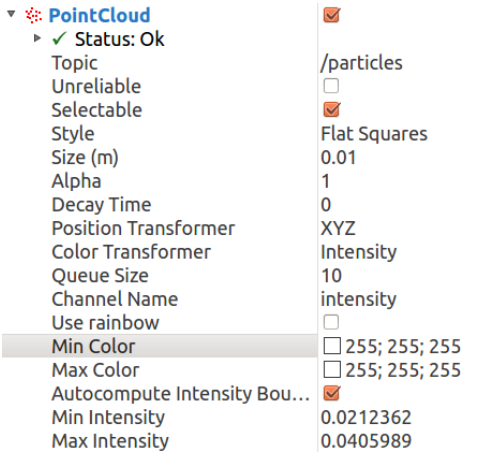
(g) **[5 points]** Once you are happy with your particle filter performance, restart the simulation and let Gazebo run for a few minutes. Stop or pause Gazebo and use the `plot_trajectory.py` script in the same folder to plot the trajectory and localization error. Include both figures with your report, and comment on the performance.

(h) **[5 points]** Referring to the insights made in this exercise, describe how an Extended Kalman Filter can improve over a particle filter, and what its key advantages / disadvantages are.

Hand in your solution `localization.py` as well as your answers to (a)-(h). Do not exceed 200 words per answer.



(a) Select the `/particles` topic with "Add" button.



(b) Update the color scheme for the point cloud.

Figure 1: Setup in RViz