Mobile Robotics
University of Ottawa

Assignment #4

Name: ______________________
Dr. W. Gueaieb

Total (150 points)

Instructions

- Individual work only. ABSOLUTELY no collaboration of any kind with anybody concerning any part of the assignment is allowed during the assignment period. Questions must be addressed to the professor during office hours (not by email).
- Source codes will neither be marked nor debugged. They will be executed to ensure proper functionality with different input/parameters. Only the program results are graded. It is the student's responsibility to make sure that their programs run as described.
- Source codes must be very well commented so that they are easy for the marker to understand when needed.
- You have to write your own programs. Copying a source code from the Internet or any other source is not acceptable and will be considered as plagiarism.
- All students must be aware of the rules and regulations posted on the course website, especially those on assignment submission and plagiarism.

Deliverables

- Submit your work online through the course website in exactly 6 files as follows:
  1. A pdf file containing your answer to question 1(a) and problem 2.
  2. The source code of each node in a separate file (with a proper file name and extension) and the launch file.
  3. A "00Readme.txt" text file explaining how to create the ROS package.
- Do NOT archive the files in a zip file.
- Do not forget to attach all the files BEFORE you click the "Submit" button.

Problems

(100pts) **1.** Bug algorithms are types of path planning algorithms which are based on the continuous alternation between two behaviors: (i) heading towards the goal in a straight line, and (ii) following an obstacle's boundary. We have already practiced moving a robot towards a target point in a straight line. In this assignment, you will implement the boundary following behavior based on a formal study of the problem, instead of the heuristic analyses often reported in some literature.
Consider the robotic setup detailed in Fig. 1. Let RF_W, RF_R, and RF_L, denote the world's (fixed) frame, the robot's frame, and the Lidar's frame. The x- and y-axes of the frames are colored in red and green, respectively. Assume that all coordinate systems are right-hand frames. For convenience, the robot's frame RF_R is often pinned halfway between the two active wheels with the x-axis pointing towards the front of the robot.

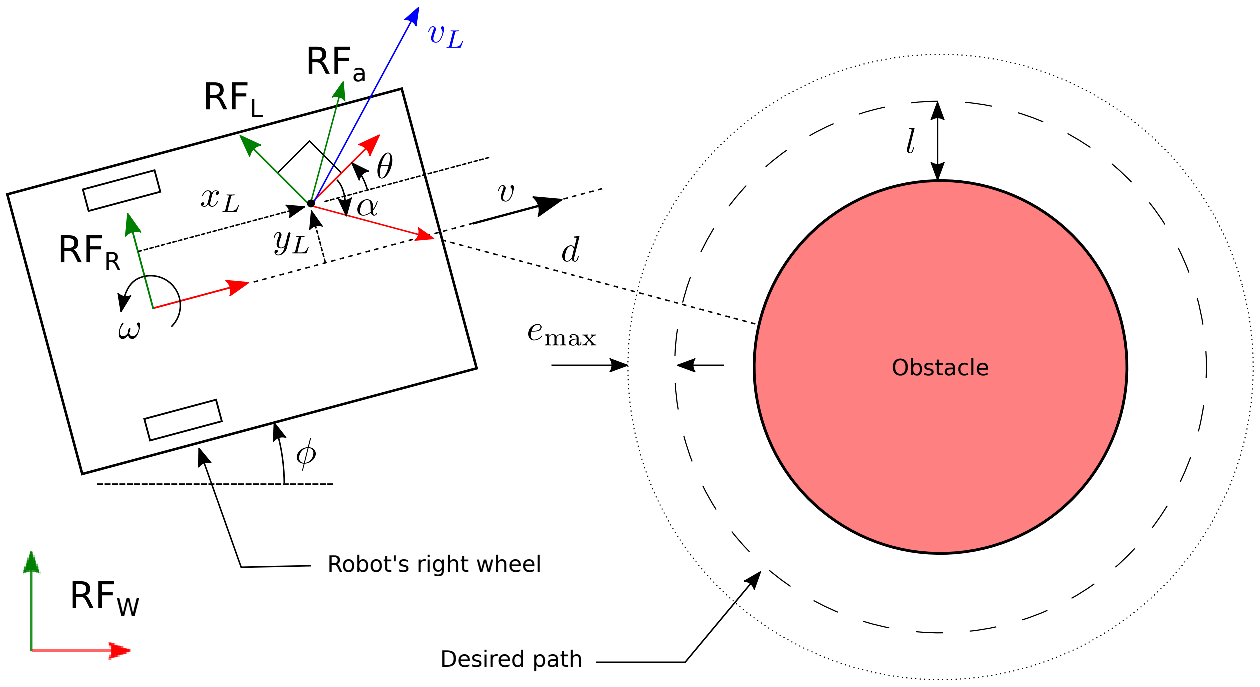

Figure 1: Robotic setup

In the following, we will use the notation ${}^{i}\cdot$ to denote the quantity $\cdot$, expressed in reference frame i. For example, the coordinates of o_L, the origin of RF_L, expressed in RF_R, is ${}^{R}\boldsymbol{o}_L = [x_L\ y_L]^T \in \mathbb{R}^2$.
Let ϕ be the robot's orientation with respect to the world coordinate system. It is also known as the robot's yaw. In other words, ϕ is the orientation of RF_R with respect to RF_W, which is the angle with which RF_W must rotate about its own z-axis to have the same orientation as RF_R. Another way to look at ϕ is that it is the directed angle from the x-axis of RF_W to the x-axis of RF_R. In the same way, define θ to be the orientation of the Lidar's frame relative to the robot's frame. Note that since all the frames are right-hand frames, their z-axes are pointing upward, and the positive direction of angles and of rotational motions is counter-clockwise.
Let d be the shortest distance between the Lidar and an obstacle in the workspace; and α be its correspondent angle (with respect to the Lidar's frame). To help with the subsequent analysis, we will consider an auxiliary frame RF_a pinned at the origin of the Lidar's frame RF_L while its x-axis is pointing towards the obstacle's closest point to the Lidar. Notice how α is also the orientation of RF_a relative to

RF$_\text{L}$.

The following is a description of the procedure to make a differential drive wheeled robot, such as the one in Fig. 1, follows an obstacle bounary of any shape while ideally leaving a safety distance l between the Lidar and the boundary. The problem boils down to determining the linear velocity vector $\boldsymbol{v}_L$ that has to be applied to the Lidar to execute this motion. It is important to observe that aligning $\boldsymbol{v}_L$ with the x-axis of RF$_\text{a}$ makes the Lidar head straight (in a radial direction) to the obstacle, while aligning $\boldsymbol{v}_L$ with the y-axis of RF$_\text{a}$ forces the Lidar to navigate parallel (in a tangential direction) to the obstacle's boundary. Therefore, deciding on a compromise between both directions, defines the coordinates of ${}^a\boldsymbol{v}_L$, and presumably drives the Lidar along the desired path.

Let $e = d - l$ be tracking error. Note that when $e > 0$, the Lidar is further than it should be from the obstacle's boundary, and when $e < 0$, it is closer than it should be. The objective is then to maintain e as close as possible to zero at all time. When $e = 0$, it makes sense to have

$${}^a\boldsymbol{v}_L = \begin{bmatrix} 0 \\ \gamma \end{bmatrix}$$

for some scalar $\gamma > 0$. Following the same logic, the Lidar must head straight to the obstacle, totally along the radial direction, if the error exceeds a certain user-defined threshold $e_\text{max} > 0$. In other words, when $|e| \geq e_\text{max}$, the velocity vector must be set as

$${}^a\boldsymbol{v}_L = \begin{bmatrix} \beta \\ 0 \end{bmatrix}$$

for some nonzero $\beta \in \mathbb{R}$. To implement this logic, let us define $\bar{e}$ as the normalized error. That is,

$$\bar{e} = \begin{cases} 1 & \text{, if } e \geq e_\text{max} \\ e/e_\text{max} & \text{, if } -e_\text{max} \leq e \leq e_\text{max} \\ -1 & \text{, if } e \leq -e_\text{max} \end{cases}$$

Therefore, defining ${}^a\boldsymbol{v}_L$ in the following manner, automatically satisfies all the above conditions, and so implements the desired robot behavior:

$${}^a\boldsymbol{v}_L = K_p \begin{bmatrix} \bar{e} \\ 1 - |\bar{e}| \end{bmatrix}$$

for some user-defined positive parameter K_p. Those who are familiar with control systems can probably see how this definition of ${}^a\boldsymbol{v}_L$ implements some sort of a P-controller to track the desired path. Of course, it would be more adequate to use a PID controller, but a P-controller is sufficient for this assignment.

Now that ${}^a\boldsymbol{v}_L$ is determined, we face another problem which is to determine the required twist to command the robot to generate the motion defined by ${}^a\boldsymbol{v}_L$? For planar robots, such as the one in hand, the commanded twist is defined by two components (as in the case of the ROS topic "cmd_vel"):

- $v \in \mathbb{R}$: the linear velocity of the robot (RF$_\text{R}$) along its own x-axis.
- $\omega \in \mathbb{R}$: the angular velocity of the robot (RF$_\text{R}$) around its own z-axis.

It can be proven that

$${}^W\boldsymbol{v}_L = \boldsymbol{T} \cdot \begin{bmatrix} v \\ \omega \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} v \\ \omega \end{bmatrix} = \boldsymbol{T}^{-1} \cdot {}^W\boldsymbol{v}_L$$

where "·" denotes the dot-product operator, and

$$\boldsymbol{T} = \begin{bmatrix} \cos\phi & -(x_L \sin\phi + y_L \cos\phi) \\ \sin\phi & x_L \cos\phi - y_L \sin\phi \end{bmatrix} \qquad \Rightarrow \boldsymbol{T}^{-1} = \begin{bmatrix} \cos\phi - \frac{y_L}{x_L} \sin\phi & \sin\phi + \frac{y_L}{x_L} \cos\phi \\ \frac{-1}{x_L} \sin\phi & \frac{1}{x_L} \cos\phi \end{bmatrix}$$

assuming that $x_L \neq 0$. Note that $\det(\boldsymbol{T}) = x_L$, which makes the matrix $\boldsymbol{T}$ invertible as long as $x_L \neq 0$. It is known that ${}^W\boldsymbol{v}_L = {}^W\boldsymbol{R}_a \cdot {}^a\boldsymbol{v}_L$, where

$${}^W\boldsymbol{R}_a = \boldsymbol{R}_z(\phi + \theta + \alpha) = \begin{bmatrix} \cos(\phi+\theta+\alpha) & -\sin(\phi+\theta+\alpha) \\ \sin(\phi+\theta+\alpha) & \cos(\phi+\theta+\alpha) \end{bmatrix}$$

Therefore,

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \boldsymbol{T}^{-1} \cdot {}^W\boldsymbol{R}_a \cdot {}^a\boldsymbol{v}_L = \begin{bmatrix} \cos(\theta+\alpha) + \frac{y_L}{x_L} \sin(\theta+\alpha) & -\sin(\theta+\alpha) + \frac{y_L}{x_L} \cos(\theta+\alpha) \\ \frac{1}{x_L} \sin(\theta+\alpha) & \frac{1}{x_L} \cos(\theta+\alpha) \end{bmatrix} \cdot {}^a\boldsymbol{v}_L \quad (1)$$

Note how the vector $[v\ \omega]^T$ is independent of ϕ, which is logical.

It is interesting to note that although many planar mobile robots, such as TurtleBot3 Burger, for instance, satisfy the conditions stated above, not all robots do. For example, Husky's Lidar frame does not satisfy these conditions since its z-axis is pointing downward. Hence, the above procedure, as is, may not work with Husky. It is possible to generalize the procedure to the most general case, but this is beyond the scope of this assignment.

In this exercise, you will create a ROS package to drive TurtleBot3 Burger to exhibit the obstacle following behavior described above. The ROS package consists of 3 nodes and a launch file, as described below. You will write a Python program for each node and the XML code for the launch file.

(a) (6 pts) Determine the values of x_L, y_L, and θ, corresponding to TurtleBot3 Burger. One way of doing so, is to use the TF display in Rviz. TurtleBot3 calls its world, robot, and Lidar frames, as `odom`, `base_link`, and `base_scan`, respectively. Another way, is to use the terminal command `rosrun tf tf_echo`. For instance, typing the following command at the command line, prints the relative position and orientation of `/base_link` with respect to `/odom`:
`rosrun tf tf_echo /odom /base_link`
Include your answer of this part in the pdf file.

(b) (32 pts) Node `sensed_object` uses the Lidar to calculate d and α, and publish them along with e, $\bar{e}$ and ${}^a\boldsymbol{v}_L$, over topic `sensed_object` at a rate of 10 Hz. If nothing is detected within the Lidar's nominal sensing range, then assign 'NaN' to these quantities. The messages sent over this topic are of type `Pose` which is defined in package `geometry_msgs`. Use the message's position variables `x`, `y` and `z`, to store d, e and $\bar{e}$, respectively (all in m); and the message's quaternion variables `x`, `y` and `z`, to store the x and y components of ${}^a\boldsymbol{v}_L$ (in m/s), and α (in rad), respectively.
Clearly define parameters $l = 1$ m, $e_{\max} = 0.3$ m and K_p, at the top of the code of this node so that the Corrector can test with different values. You may want to test with different values of K_p until you reach a satisfactory robot convergence behavior (in tracking its desired path). For a better modularity, it may be a good idea to define x_L, y_L and θ, as parameters at the top of this node's code as well.

(c) (16 pts) Node `drive_robot` drives the robot by implementing control law (1). It does so by publishing the robot's linear and angular velocities to the `/cmd_vel` topic with a frequency of 10 Hz. If no object is within the Lidar's sensing range, then the robot must stop. It should resume its motion when an object is sensed again.

(d) (20 pts) Node `data_reporter` reports the following signals, using the `loginfo` function, at a rate of 2 Hz: d, e, $\bar{e}$, α, ${}^{a}\boldsymbol{v}_L$, v and ω (one signal per line), where length, orientation, linear and angular speed signals are displayed in meters, degrees, m/s and °/s, respectively.

(e) (26 pts) A launch file which fires all the package nodes as well as `rqt_graph`, and spawns the robot in Gazebo. Node `data_reporter` is to be run automatically in a separate terminal. The launch file should also allow the following arguments to be overridden at the command line:

- x and y: the initial robot's position in the environment, with default values $(0, 0)$.
- `world_name`: the name of Gazebo's world file. The default value is `empty.world` in Gazebo's `worlds` directory.

The whole launch file must be terminated if node `navigate_robot` is killed.

Although the nodes can be executed from within the launch file, the user should be able to run them individually from the command line in any order (using the `rosrun` command). For simplicity, you may assume that the scene contains no more than one object. The object can be of any shape and may be displaced during the robot's motion.

TurtleBot3 Burger can be spawned in an empty environment in Gazebo by running the command:
`roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch`
Examining `turtlebot3_empty_world.launch` may help you creating your own launch file.
Normally, the system environment `TURTLEBOT3_MODEL` needs to be set to `burger` first, but this is already set in the provided docker image.

If needed, the robot's pose must be directly extracted from the `/odom` topic. The pose must not be calculated by integrating the robot's velocity over time (i.e., deadreckoning), as it leads to a growing cumulative error. The robot's orientation (yaw) can be calculated from a quaternion using the function `euler_from_quaternion`, as explained at:
https://www.theconstructsim.com/ros-qa-how-to-convert-quaternions-to-euler-angles/
http://docs.ros.org/jade/api/tf/html/python/transformations.html

(50pts) **2.** Consider a fixed-wing UAV (unmanned aerial vehicle) flying at a constant altitude with zero pitch angle. In this particular condition, its configuration at time t can be described by $[\,x(t)\;\; y(t)\;\; \psi(t)\;\; \phi(t)\,]^T$, where (x, y) are the Cartesian coordinates of the UAV's center of gravity, and ψ and ϕ are the roll and yaw angles, respectively (all relative to a base frame). The UAV's motion can be described by the following (continuous-time) kinematic model:

$$\dot{x}(t) = v(t)\,\cos\psi(t) \qquad \dot{y}(t) = v(t)\,\sin\psi(t) \qquad \dot{\psi}(t) = \frac{-g}{v(t)}\tan\phi(t) \qquad \dot{\phi}(t) = u_\phi(t)$$

where $g > 0$ is the gravitational acceleration; and the two scalars $v(t)$ and $u_\phi(t)$ are external control inputs. The input $v(t)$ represents the UAV's linear velocity along its frontal axis, as shown in Fig. 2. Assume that the UAV is equipped with a radio sensor that can measure the bearing angle between the UAV's main (frontal) axis and two stationary radio beacons (aka landmarks), placed at $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. For simplicity, assume that:

- The sensor is located at the UAV's center of gravity and shares the same reference frame as the UAV.
- The beacons are located on the plane of flight (i.e., the same horizontal plane as the UAV). In other words, the output of the sensor is calculated in the same way as seen in class.
- The sensor can distinguish between the two landmarks (e.g., by radio frequency).

- The sensor can always detect both landmarks.

What type of Kalman filter is adequate for estimating the configuration of the UAV? Justify your answer. Build the Kalman filter (no need to simulate it).

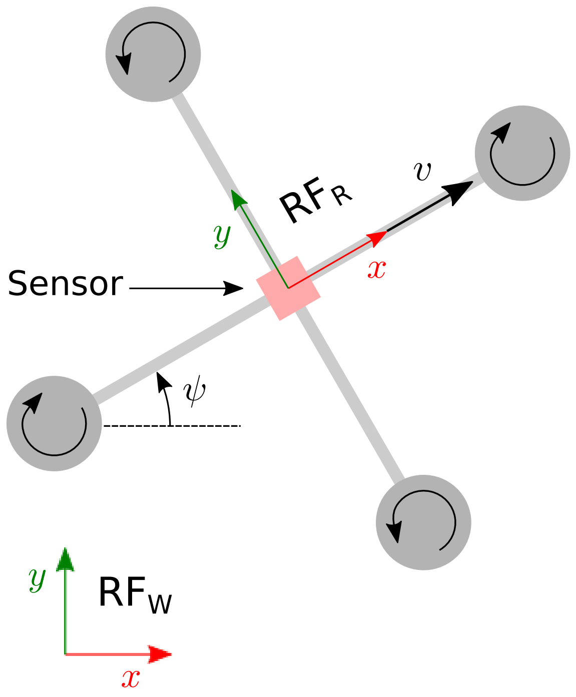

Figure 2: UAV top view