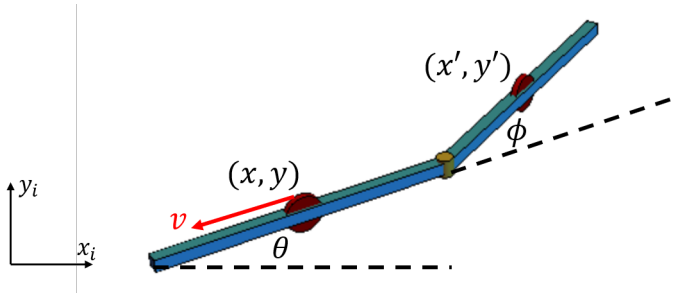
COMS W4733: Computational Aspects of Robotics

Homework 3

Due: March 11, 2019

Problem 1 (20 points)



We will investigate a simple wheeled *snake robot* comprising of two links of equal length L. The first link (left) can be driven forward like the unicycle, and the wheel on the center of the link has to satisfy both the no-slip and rolling constraints:

$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0$$
$$\dot{x}\cos\theta + \dot{y}\sin\theta = v$$

While the second link cannot be independently driven, we do have control over the relative joint velocity $\dot{\phi}$ between the two links. The second link also has a no-slip constraint centered at (x', y'); the angle θ' is the link's orientation relative to the inertial frame:

$$\dot{x}'\sin\theta' - \dot{y}'\cos\theta' = 0$$

(a) Find the coordinates $(x', y', \theta')^T$ in terms of the link length L and configuration variables x, y, θ, and ϕ. Then rewrite the second link's no-slip constraint in terms of these variables and their velocities.

(b) Write the robot's three constraints in Pfaffian form $\mathbf{A}^T(\mathbf{q})\dot{\mathbf{q}} = 0$, where the configuration velocities are $\dot{\mathbf{q}} = (\dot{x}, \dot{y}, \dot{\theta}, v, \dot{\phi})^T$. Then find the set of all allowed velocities. How many controllable degrees of freedom does the robot have, and what velocities do they correspond to?

(c) Solve for the robot's inverse kinematics. Given desired $\dot{x}(t)$ and $\dot{y}(t)$ trajectories, what are the functional forms of the input velocities? Does this robot have any singularities (configurations where the IK do not exist)? As a hint, you should see that the forward kinematics of $\dot{x}$ and $\dot{y}$ are very similar to the examples from class. These two impose requirements on θ and $\dot{\theta}$, the latter of which can be used to solve for the system's inputs.

Problem 2 (20 points)

Recall that the forward kinematics of the (rear-wheel drive) bicycle of length 1 are as follows:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \tan\phi & 0 \end{pmatrix} \begin{pmatrix} v \\ \dot{\phi} \end{pmatrix}$$

Due to the nature of the nonholonomic constraints it is not easy to predict how instantaneous velocity inputs lead to overall behavior along the entire trajectory. To do so we must simultaneously solve four ODEs (the three equations above plus one more for $\dot{\phi}(t)$). Some ways of numerically doing so would be to use `odeint` in Python, `ode45` in Matlab, or `NDSolve` in Mathematica.

(a) Suppose the bicycle travels at a constant velocity of $v = 1$ m/s while steering left and right periodically with $\dot{\phi}(t) = 0.3\sin(t)$. Numerically integrate the four ODEs for 20 seconds to solve for the workspace trajectories (set all initial configurations to 0). Provide a parametric $x(t)$-$y(t)$ plot and briefly explain why the bicycle moves in the manner you see. Also provide a plot verifying that $v = 1 = \sqrt{x(t)^2 + y(t)^2}$ by plotting the expression over the time interval.

(b) Repeat the solver with $\dot{\phi}(t) = 0.6\sin(t)$, and again provide a parametric plot and a brief explanation. How do the results differ from those of the previous input?

(c) What happens when you try to use the input $\dot{\phi}(t) = \sin(t)$? Explain.

Problem 3 (60 points)

You will be implementing the Bug 2 path planning algorithm for a Turtlebot in ROS. First set up the Arbotix python simulator by following the instructions in Chapter 6 of ROS By Example. To set up the Turtlebot in ROS, first run the command

```
sudo apt-get install ros-indigo-turtlebot-gazebo
```

You can then test your installation by running

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

You should see a Gazebo simulation window launch with the Turtlebot in an initialized environment. You'll be using different world models for this assignment, which you can launch by adding the world file option (be sure to use an absolute address for the world file):

```
roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=$PWD/bug2_0.world
```

We have provided several world files on which you can test your implementation. Make sure that your bug can successfully complete all worlds. After loading the new world in Gazebo, you can run your script to **move from the starting position at the origin to a goal position 10 meters away on the x-axis**. To start off, look at the provided `odom_out_and_back.py` script from Section 7.8 of ROS By Example, which simply moves the robot away and back toward its starting position. You will want to implement your Bug algorithm there.

You should use the `nav_msgs/Odometry` messages to update the position of the robot (refer to Section 7.8 of ROS By Example). To sense when obstacles are nearby, the Turtlebot has a laser scanner, the use of which is explained in Chapter 7 of Programming Robots with ROS.

Recall that the basic premise of Bug 2 is to move from a start position to a goal position along the m-line between the two positions. If the robot senses an obstacle, then it invokes a contour-following behavior until it reaches the m-line again, at which point it resumes following the line. This alternating behavior continues as long as there are obstacles or until the goal position is reached. One possible strategy that you can try out would be to repeatedly follow these steps (other strategies would be perfectly valid as well).

- Follow m-line (go forward) until obstacle encountered.
- If obstacle encountered:
  - Store hit point.
  - Turn left until obstacle is no longer detectable to the right of the robot.
  - Follow the obstacle according to the steps below until m-line or hit point reached. If hit point is reached, return failure.
    - Move forward a small distance.
    - If object not detected on the right, turn slightly right.
    - Else object is close on the right, so turn slightly left.

You will likely have to experiment with how much you translate or rotate in each step before re-checking the laser scanner for an obstacle. These example Bug 2 videos may be helpful.

What to submit: For your Bug implementation, provide a *brief* writeup of about a paragraph describing any difficulties or unexpected observations that you found. Also describe any solutions or workarounds that you felt worked well for these specific maps (including any that may not necessarily be useful for a generic Bug algorithm).

Also be sure to submit your Python implementation on Gradescope, **along with a zip file of recorded videos of your Gazebo window or computer screen while running Bug 2 on each map**. The videos will give us a reference while grading in case there are any discrepancies between results on your machine and ours when running your script.