CS498 Intelligent Robotics, Spring 2020
Homework 2: Motion Planning for Vehicles and in Higher Dimensions

Due date: 2/20/2020

Instructions: This assignment is to be completed *individually*. You are allowed to discuss the problems with your classmates, but all work must be your own. You are not permitted to copy answers, take written notes, or look up online solutions.

All coding will be done on the Klamp't Jupyter system. Setup is as follows:

1. Login and open up a terminal under the `Run` tab.
2. If this is your first time, run `git clone https://github.com/krishauser/cs498ir_s2020.git; cs498ir_s2020/setup.sh`. If this is not your first time, run `cs498ir_s2020/update.sh` to get the updated homework code. Go ahead and close the terminal tab, and return to the file browser. The homework code is in `cs498ir_s2020/HW2`.
3. When you run the notebooks, check the Kernel menu to make sure you are using the `cs498ir-virtualenv` kernel.
4. Make sure to periodically backup your code to your local machine. To do this, select the notebook in the file browser and choose `Download`. (We don't expect to have the server die, but Engineering IT won't make any guarantees!)

You may implement your calculations manually using Klampt, Numpy, or the elementary operations found in the Python math module, according to your personal preference.

To submit, upload your written answers in the form of an image or PDF, and programming answers in the form of Jupyter notebook files (*.ipynb), onto the class http://learn.illinois.edu site.

Written Problem 1: Geometric primitive collision detection

A. Consider two lines on the plane. Line 1 passes through the points $a, b \in R^2$ and Line 2 passes through the points $c, d \in R^2$. Give a formula or procedure to calculate for the intersection between two lines. Your answer should report "no intersection" if the lines are parallel; no need to check if they are coincident. *Hint: the intersection point x must satisfy the two line equations $x = a + u(b - a)$ and $x = c + v(d - c)$ for some parameters $u, v \in R$.*

B. Consider the problem of checking collision between two segments on the plane. Segment 1 connects the points $a, b \in R^2$ and Segment 2 connects the points $c, d \in R^2$ (i.e., we want to check for collision between $\overline{ab}$ and $\overline{cd}$). Using your solution to part A, give a method to report whether $\overline{ab}$ and $\overline{cd}$ are colliding. *Hint: the intersection point of the two lines must lie within the*

range of the segment. Consider testing the values of u and v.

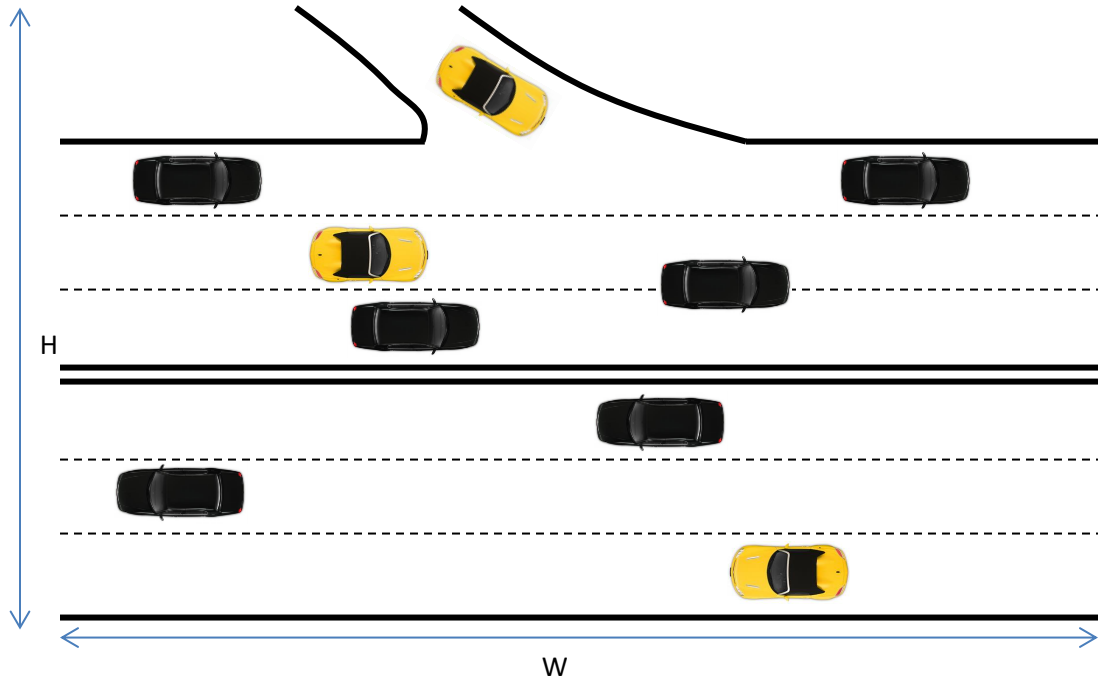
C. Describe an algorithm (in pseudocode) that uses your solution to part B to test whether two general polygons A=(a_1,…,a_m) and B=(b_1,…,b_n) are in collision. Here a polygon is described as an ordered list of points walking around the perimeter of the polygon. In other words, the edges of the polygon A are the segments S_A={$\overline{a_1a_2}$, $\overline{a_2a_3}$, …, $\overline{a_{m-1}a_m}$, $\overline{a_ma_1}$ }.

   Don't forget to handle the case where one polygon is contained completely within the other. You can use the fact that a point P lies within a polygon A if and only if a ray, whose source lies at P and whose direction is arbitrary, intersects A an odd number of times.

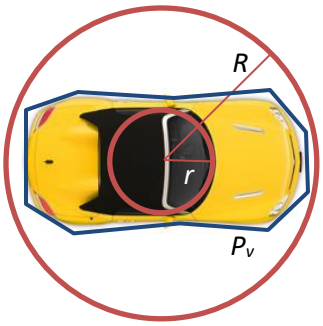
D. How many segment-segment collision tests does your algorithm perform, as a function of *m* and *n*?

Written Problem 2: Collision detection data structures

Imagine you are building a large scale traffic simulator for a section of congested highway, and you'd like to build a collision detector that can handle hundreds or thousands of simultaneous vehicles. Each vehicle V has a translation (x, y) and rotation θ, and its projection on the plane in local coordinates is given by a polygon P_V.



Suppose every vehicle's polygon has at most 10 vertices. It also fits within a circle of radius *R*, and contains a circle of radius *r* centered around the reference position (i.e., each car's local coordinate (0,0)).



A. If there are *N* vehicles, how many segment-segment collision tests might you need to run assuming a naïve all-pairs collision detection algorithm?

B. Using the inner radius, give an upper bound on the maximum number of vehicles that can fit on the highway section without colliding if the section has width *W* and height *H*.

C. Consider accelerating the collision computation using a spatial grid method, as described Robotic Systems Chapter 9.

Given a grid spacing *d*, the grid cell indexed by integers (i,j) contains all points (*x*,*y*) such that (i,j)=(round(*x*/*d*),round(*y*/*d*)), where the round() function rounds a real number to its nearest integer.

The grid-based collision detection algorithm is as follows:

1. Initialize a 2D grid with spacing *d* and sufficient numbers of entries to completely contain the highway section. Each cell contains an empty list of vehicles.
2. For each vehicle V at configuration (x,y,θ), repeat steps 3-6:
3. Find the grid cell (i,j) containing the origin of V by calculating (i,j)=(round(*x*/*d*),round(*y*/*d*)), where the round() function rounds a real number to its nearest integer.
4. Loop over all other cells that may contain vehicles whose geometry *might overlap* V's geometry. (You will need to figure out which cells this may be)
5. For each vehicle V_2 in those cells, test whether the geometry of V_2 overlaps the geometry of V at their respective configurations. For example, you may use your answer from 1.C. If a collision is found, report "V and V_2 are in collision".
6. Otherwise, add V to the list of vehicles in cell (i,j).

You are now considering the choice of the grid spacing *d*.

How small would you have to make *d* so that, in the usual case where nothing is colliding, you can be guaranteed that at most one vehicle is contained within each cell?

D. How large would you have to make d so that you only need to check the cell (round(x/d),round(y/d)) and its eight neighbors in step 2?

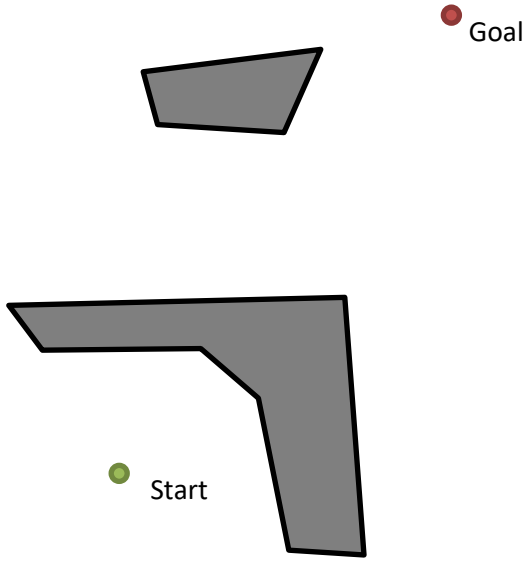
   In general, for a given d, at most how many cells would you have to check in step 2? (You may give a rough upper bound)

Written Problem 3: Bug Algorithm for a Point Robot

In this problem we will consider *bug algorithms*, which use local sensing and a small amount of computing to guide a robot, rather than global information about the shape of obstacles. Assume that the world is 2D, bounded, and contains some number of (not necessarily convex) polygonal obstacles, and that the robot is a point. The robot can sense where the target lies in the x-y plane, can detect when it hits an obstacle, and can also follow the boundary of obstacles.

A. Let "Bug A" perform the following steps:
   1. Proceed toward the goal.
   2. If an obstacle is hit, follow the obstacle's boundary toward the left, until you can head toward the goal again.
   3. Repeat until the goal is reached.

   First, draw the path traced out by Bug A when given the following problem:



B. Is there any arrangement of obstacles in which a path to the goal exists, but for which Bug A will not find one with a finite number of steps? If so, draw one. If not, give a brief explanation for why Bug A will always work.

C. Let Bug B perform the following steps:
   1. Proceed toward the goal.

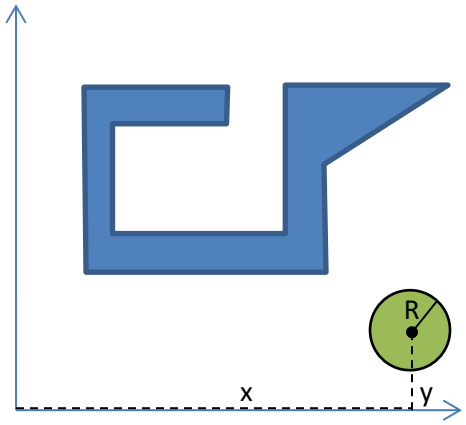
2. If an obstacle is encountered, circumnavigate it proceeding toward the left, and while doing so, remember the closest point to the goal.
3. Proceed to that closest point (again following the obstacle boundary) and then proceed toward the goal.

Draw the path traced out by Bug B when given the above problem.

D. Is there any arrangement of obstacles in which a path to the goal exists, but for which Bug B will not find one with a finite number of steps? If so, draw one. If not, given a brief explanation for why Bug B will always work.

Written Problem 4: Configuration space obstacles

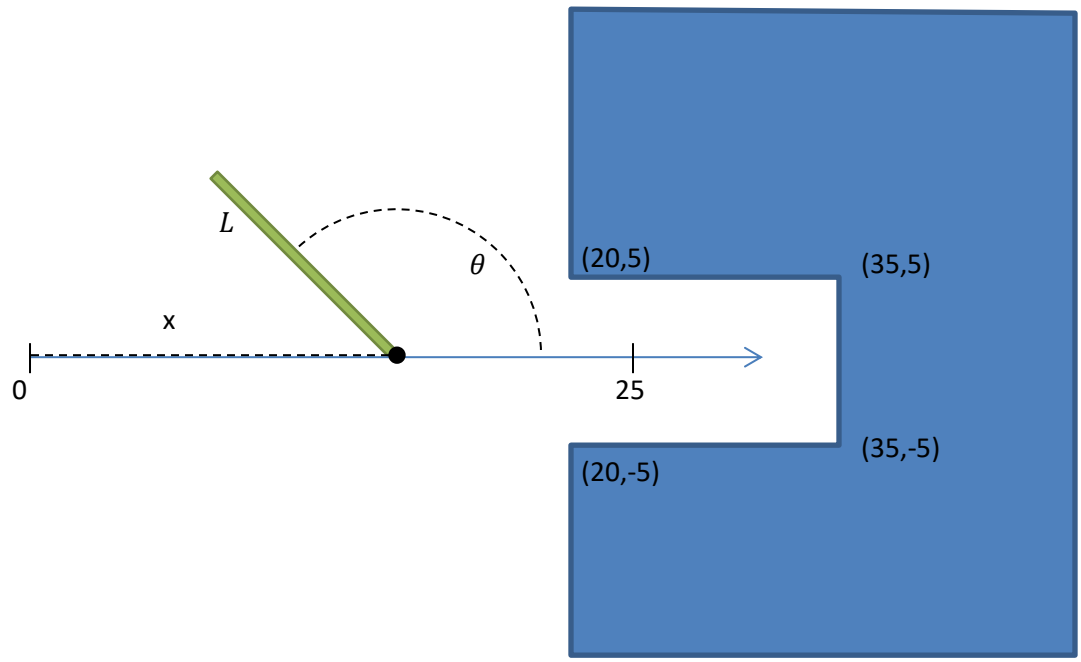
A. Suppose a planar, translating disk robot with radius R has a configuration (x,y) specifying the position of its center. Draw the C-space obstacle corresponding to the following blue workspace obstacle. Your drawing should capture all of the major features of the C-space obstacle.



B. Now suppose the robot can rotate about its center as well, like a Roomba. Its new configuration is (x, y, θ) and its configuration space is $R^2 \times SO(2)$. If you were to consider the θ dimension as a z dimension with range $[0,2\pi)$, what would be the shape of this configuration space obstacle?

C. Suppose a PR robot has a first joint that is allowed to slide along the x axis in the range [0,25] and it has a second joint that can rotate freely. A bar is attached to the second joint of length L=10. Draw the configuration space obstacle corresponding to the following blue obstacle. Your drawing should depict a region in (x, θ) space, and although it may not be totally precise, it

must capture all of the main features of the C-obstacle.



Programming Lab A: C-Obstacles

Lab 2a shows a differential drive robot model with some obstacles in its environment. Your assignment is to plot the configuration space obstacles (i.e., configuration space $q = (x, y, \theta)$) as a 3D grid, with the θ direction corresponding to the z axis.

For each configuration, collision check the robot's geometry against each of the obstacles given in the `obstacles`. The robot (a klampt.RigidObjectModel object) can be checked for collision with an obstacle (a klampt.Geometry3D object) by 1) setting the robot's transform by calling `robot.setTransform(R,t)` and then 2) using the boolean predicate `robot.geometry().collides(obstacle)`.

To help debug you may want to see where the robot is located at a particular rigid transform T=(R,t). To do so, you can call `robot.setTransform(R,t); kvis.update()`.

Your code should be placed in the last cell. For each workspace obstacle, add the C-obstacle points to the visualization using the `add_points(q_points,rgb_color)` function. Assign a different color to each C-obstacle.

Programming Lab B: Differential Drive Path Planning

Lab 2b asks you to implement a path planner for the same differential drive robot considered in Lab2a. You will also need to implement a routine for driving the robot's wheels to follow a desired path.

1. In the second cell, complete the implementation of a grid search in configuration space SE(2) (i.e., configuration space $q = (x, y, \theta)$) for the robot to reach a target position (x_G,y_G). The orientation of the goal is disregarded, and you will use an 8-connected grid in the x-y dimensions. In the search, perform collision checking at each grid vertex and along each edge. For edge collision checking, split the edge into 10 configurations and perform static collision checks at those configurations.
2. In the third cell, implement the procedure `dd_path_follow` that takes as input the configuration space path output by the grid search $q_1, \dots, q_n$, and outputs a piecewise constant sequence of robot wheel velocities $(v_{l1}, v_{r1}, \Delta t_1), \dots, (v_{lM}, v_{rM}, \Delta t_M)$ such that robot is driven along the desired path. What this means is that for time Δt_1, the left wheel is driven at velocity v_{l1} while the right wheel is driven at velocity v_{r1}. Then, for time Δt_2, the left wheel is driven at velocity v_{l2} while the right wheel is driven at velocity v_{r2}, etc.

   The values `dd_wheel_base` and `dd_wheel_radius` give the *L* and *r* parameters used in the slides.

   Check to make sure that your control generates the appropriate paths for the robot as shown in the visualizer (press the play button to show the animation).

Programming Lab C: Sampling-based Motion Planning

Lab 2c asks you to use a sampling-based planner for 4 disk robots to avoid collisions with one another. The joint configuration space of all four robots is 8-dimensional: q=(x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4). Each robot's geometry is a disk of radius R_i centered at (x_i,y_i). The workspace is the rectangle [x_{min},x_{max}]×[y_{min},y_{max}], and there are a set of circular and rectangular obstacles in the `obstacles` list.

1. Implement the `sample(), feasible(q), and visible(a,b)` methods of the `FourDiskCSpace` class. These are "hooks" used by the sampling based planner. The `sample` method should sample a configuration in this configuration space. The `feasible` method should check collisions between robots at the configuration *q* (returning false if robots collide with each other, the workspace boundary, or obstacles), while the `visible` method should check if a collision would occur if the straight line motion between feasible configurations *a* and *b* contains a collision. (Note: to verify that your code is working properly, your code should fail on the `barrier_obstacles` and `too_big_circle_obstacles` problems)
2. Test the planner types `'prm'`, `'rrt'`, `'prm'` + `shortcut=True`, and `'lazyrrg*'` on the three obstacle sets `empty_obstacles`, `narrow_passage_obstacles`, and `big_circle_obstacles`. The former two planner types are feasible (non-optimizing) planners, while the latter two are optimizing planners. Using the visualization, observe the paths generated. In your HW2 written answers, qualitatively describe the performance differences between each planner, including success rate, running time, path length, and other aspects of path quality (e.g., jerkiness, proximity to obstacles).

3. *Extra credit*: implement a class called `NDiskCSpace` that also accepts an arbitrary number of robots, defined by the parameter N. You should also define a function to sample random feasible start and goal configurations. Empirically test the performance of sampling-based planners as N increases, and devise some quantitative metrics for these performance qualities. Plot those metrics (e.g., using matplotlib) and include them with your HW2 written answers.