Assignment 2

Sampling-Based Motion Planning: RRT and RRT*

Deadline: November 2, noon, i.e., 12pm
Perfect score: 100 points for CS460 and 120 points for CS560

Assignment Instructions:

Assignments should be completed **individually** by each student.

Submission Process: Submit your reports electronically as a PDF document through Sakai (`sakai.rutgers.edu`). For your reports, do not submit Word documents, raw text, or hardcopies etc. Make sure to generate and submit a PDF instead. On the first page of the PDF indicate your name, whether you are taking CS460 or CS560 and your Net ID. Failure to follow these rules will result in a lower grade in the assignment.

Late Submissions: No late submissions are allowed. You will be awarded 0 points for late assignments.

Extra Credit for LaTeX: You will receive 4% extra credit points if you submit your answers as a typeset PDF (i.e., using LaTeX). Resources on how to use LaTeX are available on the course's website. There will be a 2% bonus for electronically prepared answers (e.g., on MS Word, etc.) that are not typeset. Have in mind that these bonuses are computed as percentage of your original grade, i.e., if you were to receive 50 points and you have typesetted your report using LaTeX then you get 2 points bonus. If you want to submit a handwritten report, scan it and submit a PDF via Sakai. We will not accept hardcopies. If you choose to submit scanned handwritten answers and we are not able to read them, you will not be awarded any points for the part of the solution that is unreadable.

Precision: Try to be precise in your description and thorough in your evaluation. Have in mind that you are trying to convince skeptical evaluators (i.e., computer scientists...) that your answers are correct.

Collusion, Plagiarism, etc.: Each student must prepare their solutions independently from others, i.e., without using common code, notes or worksheets with other students. You can discuss material about the class that relates to the assignment but you should not be provided the answers by other students and you must write down your answers independently. Furthermore, you must indicate any external sources you have used in the preparation of your solution. This includes both online sources or discussions with other students. Unless explicitly allowed by the assignment, do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or the university (the standards are available through the course's website). Failure to follow these rules may result in failure in the course.

Course's website:
`https://robotics.cs.rutgers.edu/pracsys/courses/intro-to-computational-robotics/`

Assignment Description

This assignment is a programming assignment. You are asked to implement sampling-based motion planning algorithms, such as `RRT` and `RRT`* for some problems in 2D workspaces. Do not be overwhelmed by the number of pages this assignment includes. A lot of them are providing detailed specifications of the functions we expect you to implement and their behavior. Nevertheless, **do start early** since **later parts of the PA builds upon previous parts**, and it will get easier as you move along the assignment.

For the submission of this programming assignment, you should create a top-level folder named with your NETID, (e.g., xyz007). Then, for each problem in a given assignment, you should create a sub-folder for that problem named with the problem number. Also include a pdf file in your top folder (xyz007.pdf), which contains your report. In particular, for this programming assignment, which has four problems (where one of the problems is elective for CS460, and required for CS560), you should have the following folder structure:

xyz007/1/ (files for the 2D Geometric Motion Planning problem go here)
xyz007/2/ (files for the Car Motion Planning problem go here)
xyz007/3/ (files for the Kinematics Motion Planning go here)
xyz007/4/ (if you work on the extra problem)
xyz007/report.pdf

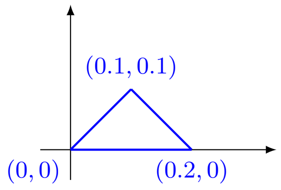
When you are ready to submit, remove any extra files (e.g., some python interpreter will create .pyc files) that are not required and zip the entire folder. The zip file should also be named with your NETID as xyz007.zip.

You are required to write your program adhering to Python 3.8 standards. Specifically, we will grade only using python 3.8. Beyond the default libraries supplied in the standard Python distribution, you may use ONLY numpy and matplotlib libraries for this MP assignment.

1 2D Geometric Motion Planning [50 points]

Robot. You will be implementing the Rapidly-exploring Random Tree (`RRT`) algorithm and the `RRT`* algorithm for a *convex polygonal translating robot*. That is, the robot will NOT rotate. The robot, in its *local coordinate system*, will be defined as a list of clockwise arranged points.

For instance, if the robot geometry is defined by the three points, $(0,0), (0.1,0.1), (0.2,0)$, ($(0,0)$ is the origin of the robot coordinate system, and will always be included in the point list) it would give rise to the triangle shown below.

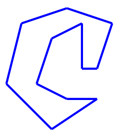


The robot's configuration in the (global) workspace will be specified using a single point $q = (q_x, q_y)$, which is the origin of the robot's local frame. That is, the robot specified above, will be occupying the space bounded by the points:

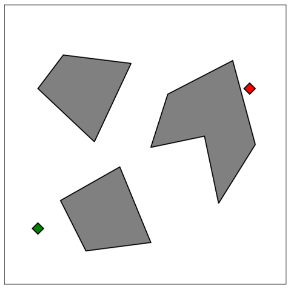
$(q_x, q_y), (q_x + 0.1, q_y + 0.1), (q_x + 0.2, q_y)$

Note that the robot is not fixed to be the triangle provided above. It may be any convex polygon. Your software should be able to handle any such polygon.

Environment/workspace. The robot resides in a 10×10 region with the lower left corner being $(0,0)$ and the upper right corner being $(10,10)$. There are multiple polygonal obstacles in the region. The obstacles may be non-convex, e.g., you may have obstacles that look like the non-convex polygon below.



The environment will contain a positive number of such obstacles. You may assume that two obstacles will not intersect with each other and no obstacle will intersect the boundary of the environment.



The problem. The robot and obstacles are specified in an input file (e.g., `robot_env_01.txt`). Each line in the file represents a list of clockwise arranged x-y coordinates, which define a polygon.The first line of the file defines the robot and the rest of the lines are the obstacles, one obstacle per line.

The planning problems are specified in another input file (e.g., `probs_01.txt`). Each line in the file represents the start and goal locations. An example could be

```
0.  0.  10.  10.
```

which represents the start location as $(0., 0.)$, and the goal location as $(10., 10.)$. This input filename contains the index of the environment (e.g., `probs_01.txt` for `robot_env_01`).

For your own testing, you can generate your own planning problems for each environment. After submission, however, we will use our planning problem files to test the correctness of your implementation.

The tasks. You need to create and implement Python scripts, and write a report by following the instructions below.

Problem 1 [30 points]. **Implementation of RRT for a Polygon Robot with Polygon Obstacles**.

1. **Implement a parser for the input file, and visualize the obstacles and the planning problem [5pt]**.

   You are to implement the function in file `file_parse.py`

   ```
   parse_problem(world_file, problem_file)
   ```

   The arguments to this function are the location of the world definition file and the problem definition file. This function should parse the definition files, and return a tuple

   ```
   (robot, obstacles, problems)
   ```

   Here `robot` is a list of 2d points representing the robot geometry in the local coordinate system; `obstacles` is a list of lists, where each list contains a collection of 2d points; `problems` is a list of start-goal pairs representing the planning problems. Specifically, it has the following structure as an example:

   $$[problem1 : [start : [x_s, y_s], goal : [x_g, y_g]]]$$

   Here "problem1" is used to indicate the first element in `problems` corresponds to the first problem, and there will be other problems in the list as well. The tags "problem1", "start" and "goal" are only used for annotation purpose, and they **WON'T** be included in the list.

   You also need to implement in file `visualizer.py`

   ```
   visualize_problem(robot, obstacles, start, goal)
   ```

   Given the parsed world definition and the start and goal of the planning problem, you need to show a visualization for the obstacles, the world boundaries, the start and goal states of the robot. To portrait the start and goal states, you just need to translate the robot polygon to the start and goal locations and show them with different colors.

2. **Implement a sampler to sample a random point in the configuration space, and visualize the robot at that point in the world [5pt]**.

   You are to implement the function in file `sampler.py`

   ```
   sample()
   ```

   There is no argument to this function, and the output should be a random tuple of a xy-coordinate that looks like

   ```
   (5.2,6.7).
   ```

   The sampling region is the world boundary $[0, 10] \times [0, 10]$. The sample DOESN'T have to be collision-free or within the configuration space boundary.

   You also need to implement the function in file `visualizer.py`

   ```
   visualize_points(points, robot, obstacles, start, goal).
   ```

The argument `points` is a list of tuples of xy-coordinates in the configuration space. An example of `points` may look like

```
[(5.2,6.7), (9.2,2.3)]
```

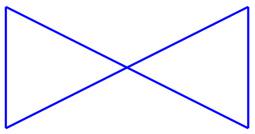
3. **Implement a collision checker for the robot and obstacles [5pt]**. You are to implement a basic collision checking routine that detects whether the robot, at a given global coordinate, would collide with **the boundary and obstacles** in the environment. For the boundary collision, **no parts of the robot geometry should be outside of the boundary.** The function should be inside the file `collision.py`. The function should have the following signature

   ```
   isCollisionFree(robot, point, obstacles)
   ```

   in which `robot` is the polygon (as a list of tuples) for the robot, `point` is where the robot is at, and `obstacles` is a list of obstacle polygons. All polygons are given as clockwise oriented points, e.g., a polygon will look like

   `[(`x_1`, `y_1`), (`x_2`, `y_2`), ...]`

   You may assume the boundary of the workspace as given in the beginning, i.e., the workspace is a 10×10 area. You do not need to consider self-intersecting obstacles like the one below. Your function should return either `True` or `False`.

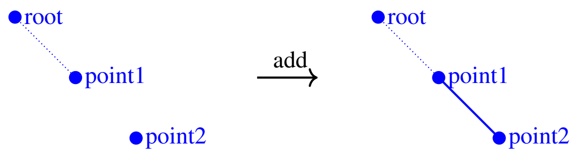


4. **Implement a data structure to represent the search tree of the planning process [5pt]**.

   You need to implement the class `Tree` in the file `tree.py`. For the class `Tree`, you have to provide the following functions:

   (a) `__init__(self, robot, obstacles, start, goal)`
   It takes the environment definition, and the start and goal as input, and initializes the tree with the first node holding value `start`.

   (b) `add(self, point1, point2)`
   `point1` and `point2` are two tuples of xy-coordinate. `point1` represents the coordinate of the parent node that you are inserting `point2` to. A visualization of the effect of this function is shown as follows:



   (c) `exists(self, point)`
   It takes an argument `point` which is a tuple of xy-coordinate, and checks whether the point is inside the tree.

   (d) `parent(self, point)`
   It takes an argument `point` which is a tuple of xy-coordinate, and returns its parent as a tuple of xy-coordinate.

   (e) `nearest(self, point)`
   It takes an argument `point` which is a tuple of xy-coordinate, and returns the nearest point (in terms of Euclidean distance) in the tree to the point. Notice that this is how we connect the abstract data structure with the geometric world.

   (f) `extend(self, point1, point2)`
   Here `point1` is a point already in the tree, and `point2` is the target point for extension from `point1`. It extends `point1` to `point2` until it hits **the last node that is collision-free**, and adds that node to the search tree. You can either implement a **discretized version** or a **complete version**. Please specify your choice in

the report. For the discretized version you can extend the line by taking a number of steps of a fixed size. The step size will thus be a parameter of your planning algorithm. For the complete version, you can get the exact intersection point between the path and the obstacle and use that as the extended point.

Please specify your implementations of each of the component and explain the parameters in the report. To test the correctness of the code, you should make sure that the functions can reflect the correct tree structure. For example, given a handcrafted tree, adding each node using the implemented function will result in the same tree structure, which can be tested by `exists` and `parent`.

[5 Extra Pts] Extra credits will be given to methods that use a more advanced method than linear search to find the nearest neighbor for the `nearest` function. Please explain the data structure and your implementation for the advanced nearest neighbor search method in your report to get the extra credit. You are **allowed** to use an external library for this purpose as long as you properly reference and devote a section in your report describing the algorithm it uses and its computational benefits relatively to the linear search. Provide a comparison of the speed up achieved by the use of the nearest neighbor data structure.

5. **Implement the RRT algorithm [5pt]**. You need to implement the function in file `rrt.py`.

   ```
   rrt(robot, obstacles, start, goal, iter_n)
   ```

   Here `iter_n` denotes the number of iterations for running RRT. The output of this function should be a list of tupled xy coordinates representing the path. An example of this could be

   ```
   [(0.0, 0.0), (0.0, 0.1),...].
   ```

   You need to make sure the path is feasible, i.e., it's collision free, and starts and ends at the desired locations represented in the definition file. You can add some simple variations, such as randomly selecting the goal for some probability.

   Please also explain your algorithm briefly, and clearly specify the parameters in your report. Also, report the number of iterations versus the success rate of the algorithm. Notice that since this is a randomized algorithm, you might need to run several times to collect data to measure the success rate.

6. **Transform the world into configuration space for visualization [5pt]**. Notice that before we have only visualized the planning scene. A better way to visualize the algorithm is to transform the world into the configuration space where the robot becomes a "point" robot. To achieve this, we have to first transform the obstacles into collision regions in the configuration space. Minkowski addition is one way to achieve this, and discretization can be another choice. Please implement the function in file `visualizer.py`

   ```
   visualize_path(robot, obstacles, path)
   ```

   to visualize the found path together with the planning scene. The path is a sequence of 2D points. Then implement

   ```
   visualize_configuration(robot, obstacles, start, goal)
   ```

   to visualize the planning problem in the configuration space. Then implement

   ```
   visualize_rrt(robot, obstacles, start, goal, iter_n)
   ```

   to visualize the rrt algorithm in the configuration space. You can either choose to use animation for visualizing the growth of the search tree, or just visualize the final state of the search tree. Notice that you should clearly show the collision regions to see how rrt can achieve obstacle-free paths.

   [5 Extra Pts] Extra credits will be given for an animated visualization of the found path.

   [5 Extra Pts] Extra credits will be given for an animated visualization of the search tree.

Problem 2 [20 points]. **Implementation of RRT* for a Polygonal Robot with Polygonal Obstacles**. Previously for RRT we only care about the feasibility of the planned path. For this problem, we are interested in planning for an optimal path as defined by the Euclidean distance. With the introduction of optimality, we need to modify parts of our previous code.

1. **Introducing cost to the search tree and Implement the Rewiring Function [5pt]**. To incorporate optimality during planning, we have to define "cost" and store this information in the search tree. Modify your code for `tree.py` so that it retains the correctness of the last problem, and supports this additional function that you need to implement:

   ```
   get_cost(self, point)
   ```

This method takes in a xy-coordinate tuple, which is the location of one of the nodes in the search tree, and returns the cost from start to that node. Here we measure the cost by the Euclidean path length. For example, if your tree contains a path from start:

```
(0,0) -- (0,1) -- (1,1)
```

The cost of point $(1, 1)$ should then become 2. Notice that you might need to add additional data fields in your tree structure to support it.

You need to implement the rewire function in the class `Tree` in the file `tree.py`. The function signature looks like:

```
rewire(self, point, r)
```

Here `point` is a tuple of xy-coordinate of a node in the tree. It should first find the nearest neighbors within distance r of the target point in the tree, and then rewire the edge as introduced in class.

2. **Implement the complete RRT* algorithm [10pt]**. You need to implement the function in file `rrt_star.py`

   ```
   rrt_star(robot, obstacles, start, goal, iter_n)
   ```

   which should reuse some of the functions that you implemented before. You can add helper functions if needed. The output of this function is a list of tupled xy coordinates representing the path, similar to the RRT function.

   Include a brief explanation of your algorithm and the newly implemented functions in your report. Also, report the success rate and path length versus iteration number as a graph in your report. Show a comparison with the RRT algorithm, and provide some analysis for the results. Does RRT* find paths with better quality than RRT?

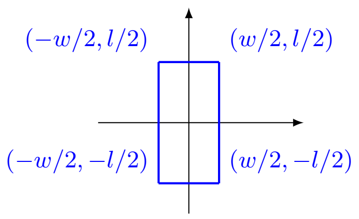
3. **Transform the world into configuration space for visualization [5pt]**. Similar to the RRT problem, you are to visualize the RRT* progress either by animation, or by visualizing the search tree of RRT*. You are to implement the function in file `visualizer.py`

   ```
   visualize_rrt_star(robot, obstacles, start, goal , iter_n)
   ```

   Clearly show in the report when the rewiring procedure takes place, and how it affects the search tree.

2 Geometric Motion Planning for a Car [30 points]

Robot. You will be implementing the Rapidly-exploring Random Tree (RRT) algorithm and RRT* algorithm for a car. That is, the robot can translate or rotate, and has the configuration given by pose (x, y, θ), where (x, y) represents the translation of the robot, and θ represents the rotational angle in the *counter-clockwise* direction. The robot will be defined given two parameters (w, l), where w represents the width of the car, and l represents the length of the car. At pose $(0, 0, 0)$, the robot can be visualized as



The robot geometry at pose (x, y, θ) contains points that are **first rotated** counter-clockwise by θ, and **then translated** by (x, y).

Environment/workspace. We use the same environment settings as before, where obstacles are represented by polygons.

The problem. The robot and obstacles are specified in an input file (e.g., `robot_env_01.txt`). The first line of the file defines the robot geometry giving the width w and the length l separated by one space. Then each of the remaining lines in the file represents a list of clockwise arranged x-y coordinates that defines a polygonal obstacle.

The planning problems are specified in another input file (e.g., `probs_01.txt`). Each line in the file represents the start and goal poses of the robot. An example could be

```
2.  2.  0.  7.  7.  1.57
```

which represents the start pose as $(2., 2., 0.)$, and the goal pose as $(7., 7., 1.57)$. Overall this input file contains a number of problems as defined in the environment indicated by the id (e.g., environment 01 as for `probs_01.txt`)

For testing on your own, you can generate the planning problems for each environment by generating the planning problem file. However, after submission, we will use our planning problem files to test the correctness of your implementations.

The tasks. You need to create and implement Python scripts, and write a report by following the instructions below.

Problem 1 [30 points]. **Implementation of RRT and RRT* for a Car with Polygon Obstacles**.

1. **Model the robot by separating the geometry and pose [5pt]**.

   Before we have introduced the robot model, which is a rectangular object represented by its width and length. Different from the translation problem we solved before, we have to consider also the rotation in this problem, and it would be easier to separately consider the geometry and pose of the robot, since we can then transform the robot geometry in the *local coordinate system* to the world. This principle is also generally adopted in motion planning of more complex robots such as robotic manipulators.

   In this problem, you are to implement a class `Robot` in file `robot.py` which defines a robot, and provides functions to transform a robot to the world. You need to implement the following functions:

   (a) `__init__(self, width, height)`
   This is the constructor for the robot object, and it assigns the width and height to the robot internal field `self.width` and `self.height` for later usage. It then resets the robot pose to $(0, 0, 0)$, and stores it internally at `self.translation` and `self.rotation`. Here `self.translation` is a collection (e.g., a tuple, a list, or an numpy array) of the xy coordinates, and `self.rotation` is a floating point number representing the angle in radians.

   (b) `set_pose(self, pose)`
   This function sets the pose of the object taking an input $pose$ which is a list/tuple of three numbers.

   (c) `transform(self)`
   This function transforms the robot by its pose specified by `self.translation` and `self.rotation`, and outputs a list of four points representing the transformed local points

   $$[(-w/2, -l/2), (-w/2, l/2), (w/2, l/2), (w/2, -l/2)]$$

   in the global coordinate system. Notice that the points should follow exactly as the order shown above.

2. **Implement a parser for the input file, and visualize the obstacles and the planning problem [5pt]**.

   You are to implement the function in file `file_parse.py`

   `parse_problem(world_file, problem_file)`

   The arguments to this function are the location of the world definition file and the problem definition file. This function should parse the definition files, and return a tuple

   `(robot, obstacles, problems)`

   Here `robot` is an object of class `Robot` storing the geometric information of the robot; `obstacles` is a list of lists, where each list contains a collection of 2d points; `problems` is a list of start-goal pairs representing the planning problems.

   You then need to implement in file `visualizer.py`

   `visualize_problem(robot, obstacles, start, goal)`

   Given the parsed world definition and the start and goal of the planning problem, you need to show a visualization for the obstacles, the world boundaries, the start and goal states of the robot. To portrait the start and goal states, you need to use the "transform" function that you implemented before to transform the robot local geometry to the global coordinate system. Remember to show the start and goal states in different colors.

3. **Extend the previous functions that you implemented for the polygonal geometric motion planning problem [5pt]**. You are to extend the functions that you implemented before to the car planning problem and implement in `sample.py`, `collision.py`, and `tree.py`. You need to extend the previous implemented functions to work in the new car planning problem. Notice that your function should now have the following specifications:

   (a) In `sample.py`,

   ```
   sample()
   ```

   Now you need to output a random tuple of the pose in the range:

   $$[0, 10] \times [0, 10] \times [-\pi, \pi]$$

   For uniform sampling of the angle in $[-\pi, \pi]$, a naive implementation by directly using uniform sampling in the window is fine. However, a correct way to uniform sampling takes more consideration of the topology, which we leave as an **OPTIONAL** practice :)

   ```
   visualize_points(points, robot, obstacles, start, goal).
   ```

   Your robot should be of class `Robot`. Your start and goal should be the poses of the car.

   (b) in `collision.py`,

   ```
   isCollisionFree(robot, point, obstacles)
   ```

   (c) in class `Tree` in `tree.py`,

   ```
   __init__(self, robot, obstacles, start, goal)
   add(self, point1, point2)
   exists(self, point)
   parent(self, point)
   nearest(self, point)
   extend(self, point1, point2)
   ```

   Notice that for the function `Tree.nearest(self.point)`, you are to use the distance:

   $$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + d(\theta_1, \theta_2)^2}$$

   where the distance between two angles $d(\theta_1, \theta_2)$ should be the wrapped distance in range $[-\pi, \pi]$. For example, for two angles $2\pi/3$ and $-2\pi/3$, the wrapped distance is $-2\pi/3$, instead of $4\pi/3$.

4. **Implement the complete RRT algorithm [5pt]**. You need to implement the function in file `rrt.py`

   ```
   rrt(robot, obstacles, start, goal, iter_n)
   ```

   for the car planning problem, and follow the same instruction as before to describe it in the report.

5. **Implement the complete RRT* algorithm [5pt]**. You need to implement the function in file `rrt_star.py`

   ```
   rrt_star(robot, obstacles, start, goal, iter_n)
   ```

   Similar as before, collect statistics of success rate and path quality, and compare with RRT algorithm in your report.

6. **Visualization of the found path [5pt].** Visualize the found path of the car by plotting how the car geometry changes along the trajectory. Implement the function in file `visualizer.py`

   ```
   visualize_path(robot, obstacles, path)
   ```

   where `path` is a list of robot states as 3D tuples.

   [5 Extra Pts] Provide an animation for the visualized path.

3 Kinematic Motion Planning for a Holonomic Car [20 points]

So far you have worked on the geometric motion planning problem, which only reasons about geometry, and doesn't consider velocity or even accelerations. However, in real life, robots have to respect physical constraints. In this section, you will start with this more complex but also more realistic problem by considering robot velocities.

Robot. Continuing on the previous car planning problem, we are going to use the same representation for the car geometry. To extend to the kinematics motion planning setting, we assume we have direct control over the velocities of the car using two variables: linear velocity v and the angular velocity ω for steering. (Note that here we assume you can suddenly change the velocities of the robot, which is not achievable in reality.) Given these two variables, and assuming the pose of the robot at time t is (x, y, θ), the velocity of the pose is given by:

$$\dot{x} = v\cos(\theta + \pi/2),\ \dot{y} = v\sin(\theta + \pi/2),\ \dot{\theta} = \omega.$$

Notice that we use $\theta + \pi/2$ to evaluate the linear velocities since we assume the direction of the car points upward when its configuration is $(0, 0, 0)$. This can be illustrated as:

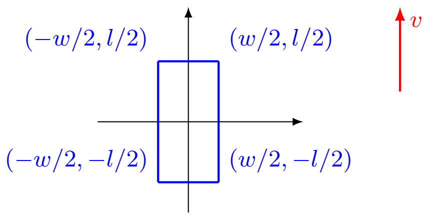


Figure 1: linear velocity at $(0, 0, 0)$

We can rewrite these by concatenating them into a single vector variable as the configuration state:

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}, u = \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix}$$

Then the dynamics equation can be written as:

$$\frac{\delta q}{\delta t}(t) = \dot{q}(t) = f(q(t), u(t), t) = \begin{bmatrix} v\cos(\theta + \pi/2) \\ v\sin(\theta + \pi/2) \\ \omega \end{bmatrix} = \begin{bmatrix} u(t)[0]\cos(q(t)[2] + \pi/2) \\ u(t)[0]\sin(q(t)[2] + \pi/2) \\ u(t)[1] \end{bmatrix} \tag{1}$$

Notice that the integral of the configuration velocity gives us the change of the configuration state:

$$\Delta q(t_1, t_2) = \int_{t_1}^{t_2} \dot{q}dt.$$

To numerically represent this in Python, here we use linear integration, which gives:

$$q(t + dt) = q(t) + \dot{q}(t)dt,$$

for some predefined value dt.

Then given an initial start configuration q, a sequence of controls $[u_0, u_1, \ldots, u_N]$, and a sequence of numbers representing how many dt are applied, we can obtain the configuration trajectory by using the above integration.

Environment/workspace. We use the same environment settings as before, where obstacles are represented by polygons.

The problem. We use the same problem definition scheme as before.

The tasks. You need to create and implement Python scripts and write a report by following the instructions below.

Problem 1 [20 points]. **Implementation of RRT for a Holonomic Car with Polygonal Obstacles**.

1. **Extend the planning problem to involve kinematics of the car [10pt]**.

   In this question, you are to extend the functions and classes that you have previously implemented for the above planning problem to work for the kinematics motion planning setting.

   Meanwhile, you need to add the following functions to take care of the kinematics:

(a) In the class `Robot` in `robot.py`, implement

```
kinematics(self, state, control)
propagate(self, state, controls, durations, dt)
```

(b) In the class `Tree` in `tree.py`, implement

```
extend(self, point, n1, n2, dt)
```

The specifications for them are the following:

```
kinematics(self, state, control)
```

should implement the kineamtics equation of the robot as follows:

$$\dot{q} = f(q, u)$$

by following equation (1), and return the computed configuration velocity $\dot{q}$. (ignore the time variable since it's unused in equation (1))

```
propagate(self, state, controls, durations, dt)
```

Given an initial state $q(0)$ by `state`, a sequence of controls

$$u_0, u_1, \ldots, u_N,$$

and a sequence

$$n_0, n_1, \ldots, n_N$$

indicating durations

$$n_0 \cdot dt, n_1 \cdot dt, \ldots, n_N \cdot dt,$$

return a sequence representing the state trajectory. The result trajectory should be a sequence of length

$$n_0 + n_1 + \cdots + n_N + 1$$

where adjacent state in the trajectory should have a duration difference of dt. Plus one since the sequence should contain the start state.

```
extend(self, point, n1, n2, dt)
```

implements a random shooting algorithm. The random shooting algorithm just randomly samples control input u, and a duration number $n \in [n_1, n_2)$, and then propagate the point $q(0)$ by them to obtain the end point $q(n \cdot dt)$. The control input u should be within some limit of your choice $u \in [u_{low}, u_{high}]$, which you should specify in the report. The choice for n_1 and n_2 is based on your design, and you should specify that in the report. If collision happens in the middle, it should stop at the last collision-free point as the end point. It then adds the endpoint to the tree.

Briefly explain how you extended the previous code to the kinematic motion planning setting. Also briefly explain how you implemented the three functions above.

2. **Extend the RRT algorithm to work for Kinematic Motion Planning [5pt]**. To extend the algorithm, modify the "extend" function by using the random shooting algorithm you just implemented. Briefly explain your algorithm in the report. Compute statistics of the algorithm, and show the results and analysis in the report as well. Does the success rate decrease?

3. **Visualize the found path [5pt]**. Provide a visualization for the path found by RRT by implementing in file `visualizer.py`:

```
visualize_trajectory(robot, obstacles, start, goal, trajectory)
```

This plots the car at each of the trajectory point.

[5 Extra Pts] Provide an animation of the result that respects the time. Does the visualization look more realistic than the path generated by geometric motion planning algorithm?

4 Kinodynamic Motion Planning for a Non-holonomic Car [Elective for CS460, Required for CS560] [20 points]

In the previous example of Kinematic Motion Planning task, you are assumed to have direct control over the velocities. As we have mentioned before, due to the Newton's Laws, sudden change to velocities is unrealistic. In this section, you will deal with the more realistic setting where you have direct control over the accelerations. The problem is often referred to as kinodynamic motion planning since you are dealing with kinodynamic constraints (accelerations, forces, etc.).

Robot. To extend to the kinodynamic motion planning setting, we assume we have direct control over the accelerations of the car using two variables: linear acceleration a_v and the steering angle acceleration a_θ. Given these two variables, and assuming the pose of the robot at time t is (x, y, θ), the acceleration of the pose is given by:

$$\ddot{x} = a_v \cos(\theta + \pi/2),\ \ddot{y} = a_v \sin(\theta + \pi/2),\ \ddot{\theta} = a_\theta.$$

Notice that we use $\theta + \pi/2$ to evaluate the linear acceleration since we assume the direction of the car points upward when its configuration is $(0, 0, 0)$, similar as in the previous problem. This can be illustrated as in Figure 2.

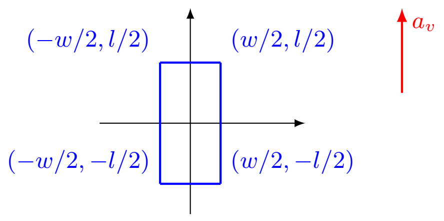


Figure 2: linear acceleration at $(0, 0, 0)$

We can rewrite the equations by concatenating them into a single vector variable as the configuration state, similar as before:

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \\ \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix}, u = \begin{bmatrix} a_v(t) \\ a_\theta(t) \end{bmatrix}$$

Then the dynamics equation can be written as:

$$\frac{\delta q}{\delta t}(t) = \dot{q}(t) = f(q(t), u(t), t) = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \\ \ddot{x}(t) \\ \ddot{y}(t) \\ \ddot{\theta}(t) \end{bmatrix} = \begin{bmatrix} q(t)[3] \\ q(t)[4] \\ q(t)[5] \\ u(t)[0]\cos(q(t)[2] + \pi/2) \\ u(t)[0]\sin(q(t)[2] + \pi/2) \\ u(t)[1] \end{bmatrix} \tag{2}$$

Notice that the integral of the configuration velocity gives us the change of the configuration state:

$$\Delta q(t_1, t_2) = \int_{t_1}^{t_2} \dot{q} dt.$$

To numerically represent this in Python, here we use linear integration, which gives:

$$q(t + dt) = q(t) + \dot{q}(t)dt,$$

for some predefined value dt.

Similar as before, given an initial start configuration q, a sequence of controls $[u_0, u_1, \ldots, u_N]$, and a sequence of numbers representing how many dt are applied, we can obtain the configuration trajectory by using the above integration.

Environment/workspace. We use the same environment settings as before, where obstacles are represented by polygons.

The problem. We use the same problem definition scheme as before.

The tasks. You need to create and implement Python scripts and write a report by following the instructions below.

Problem 1 [20 points]. **Implementation of RRT for a Non-holonomic Car with Polygonal Obstacles**.

1. **Extend the planning problem to involve dynamics of the car [10pt]**.

   In this question, you are to extend the functions and classes that you have previously implemented for the above planning problem to work for the kinodynamic motion planning setting.

   Meanwhile, you need to add the following functions to take care of the dynamics:

   (a) In the class `Robot` in `robot.py`, implement

   ```
   dynamics(self, state, control)
   propagate(self, state, controls, durations, dt)
   ```

   (b) In the class `Tree` in `tree.py`, implement

   ```
   extend(self, point, n1, n2, dt)
   ```

   The specifications for them are the following:

   ```
   dynamics(self, state, control)
   ```

   should implement the dynamics equation of the robot by following (2) and return the computed configuration velocity $\dot{q}$.

   ```
   propagate(self, state, controls, durations, dt)
   ```

   Given an initial state $q(0)$ by `state`, a sequence of controls

   $$u_0, u_1, \ldots, u_N,$$

   and a sequence

   $$n_0, n_1, \ldots, n_N$$

   indicating durations

   $$n_0 \cdot dt, n_1 \cdot dt, \ldots, n_N \cdot dt,$$

   return a sequence representing the state trajectory. The result trajectory should be a sequence of length

   $$n_0 + n_1 + \cdots + n_N + 1$$

   where adjacent state in the trajectory should have a duration difference of dt. Plus one since the sequence should contain the start state.

   ```
   extend(self, point, n1, n2, dt)
   ```

   implements a random shooting algorithm. The random shooting algorithm just randomly samples control input u, and a duration number $n \in [n_1, n_2)$, and then propagate the point $q(0)$ by them to obtain the end point $q(n \cdot dt)$. The control input u should be within some limit of your choice $u \in [u_{low}, u_{high}]$, which you should specify in the report. The choice for n_1 and n_2 is based on your design, and you should specify that in the report. If collision happens in the middle, it should stop at the last collision-free point as the end point. It then adds the endpoint to the tree.

   Briefly explain how you extended the previous code to the kinodynamic motion planning setting. Also briefly explain how you implemented the three functions above.

2. **Extend the RRT algorithm to work for Kinodynamic Motion Planning [5pt]**. To extend the algorithm, modify the "extend" function by using the random shooting algorithm you just implemented. Briefly explain your algorithm in the report. Compute statistics of the algorithm, and show the results and analysis in the report as well. Does the success rate decrease?

3. **Visualize the found path [5pt]**. Provide a visualization for the path found by RRT by implementing in file `visualizer.py`:

   ```
   visualize_trajectory(robot, obstacles, start, goal, trajectory)
   ```

   This plots the car at each of the trajectory point.

   [5 Extra Pts] Provide an animation of the result that respects time. Does the visualization look more realistic than the path generated by Kinematics Motion Planning?

Start Early!