Assignment 3: Motion Planner
CS 730/830, Spring 2025
Due **11pm on Mon Feb 10**

Overview

You will write a program to find robot motion trajectories in continuous space from a given start configuration to a goal configuration, avoiding obstacles. Your program should use the RRT algorithm; it can stop as soon as it finds its first feasible solution and it does not need to implement path smoothing. We will run on small problems so you don't need to bother implementing a fancy nearest-neighbor data structure.

Undergraduates can assume an omnidirectional point vehicle with infinite acceleration. In other words, the state of the robot is just $\langle x, y \rangle$ and it can change directions instantaneously. This means that new branches in the motion tree can extend directly to the sampled target position. Graduate students will also plan for a more realistic spaceship-like vehicle, discussed in the appendix. You should check for collisions every 0.05 units along the robot's path.

Because both the undergraduate and graduate vehicles are points, collision checking just means finding whether a point is inside an obstacle. The map representation is a simple grid, which will make this easy.

You should use a goal bias: your 'random target sample' should be the goal configuration with probability 0.05. The goal can be considered achieved when the robot is within 0.1 of it.

Input

Your program should read a map of the environment from standard input. While you will plan in continuous space, to simplify collision checking the map will be given using the same discrete format as in assignments 1 and 2: discrete 1×1 cells that are free or blocked:

```
4
3
____
__#_
___#
0.1
1.54
3.8
1.1
```

This example shows a world that is 4 units wide by 3 units high. The robot's starting state and goal specification will be given on additional lines. In this example, the robot starts at $\langle x = 0.1, y = 1.54 \rangle$ and needs to reach $\langle x = 3.8, y = 1.1 \rangle$, where $\langle 0, 0 \rangle$ is the bottom left corner. Graduate students can assume that the robot's initial angle and speed are 0, so the same initial state specification can be used.

Output

Your program should emit both a solution trajectory and a dump of the motion tree. The solution will be checked for correctness, the motion tree is just for visualization (aka debugging). For undergraduates, a solution trajectory is a list of $\langle x, y \rangle$ pairs, one per line, starting at the start state and ending at the goal. The list is preceded by the number of lines in the trajectory. For example:

```
4
0.1 0.1
2.96488988483 0.905801168404
0.935329301571 3.66633453507
0.1 3.9
```

For the graduate student vehicle, a solution trajectory is a list of $\langle \mathit{state}, \mathit{control} \rangle$ pairs, one per line, preceded by the number of lines. (The state and control are specified in more detail at the end of this handout.) Since the vehicle state is 4 numbers and a control is 2 numbers, this makes 6 numbers per line. You don't need to print the last line whose state is the goal.

The motion tree is a list of edges, one per line, preceded by the number of lines. The edges can appear in any order. For undergraduates, an edge is four numbers, the starting x and y and the ending x and y. For the graduate vehicle, each edge is 6 numbers, representing the starting state and the control to be applied.

Execution

Please use `make.sh` and `run.sh` scripts just like with assignments 1 and 2.

We will supply:

a few example benchmark problems.

`rrt-reference` a sample solution. Use the `-grad` flag for the grad student vehicle. (Use the `-left` flag for a version that can only turn left!)

`rrt-validator` runs a planner executable, validates its output, and optionally writes a PDF file showing the solution trajectory and motion tree. The validator takes two groups of arguments, separated by `--`. The first group are optional flags to control the validator's behavior: `-grad` specifies the grad student vehicle, `-noviz` deactivates the PDF output, `-o` *`filename`* controls where the PDF is written (default is `tree-viz.pdf`). None of these are automatically passed to the planner. The second group of arguments are the executable file to run and the arguments to pass to it. The validator expects a problem instance on standard input and will pass it to the planner's stdin.

Submission

Electronically submit your solution and write-up using the instructions on the course web page, including your source code as well as a transcript of your program running with the validator and solving the supplied example cases.

Your write-up should answer the following questions:

1. Describe any implementation choices you made that you felt were important. Clearly explain any aspects of your program that aren't working. Mention anything else that we should know when evaluating your work.
2. What suggestions do you have for improving this assignment in the future?

Evaluation

Grading is done with an automated script, so be sure you have one algorithm working before starting the next, so that we can give you some credit even if you don't implement both.

Rough guide to grading:

0 nothing

1 something but basically nothing

5 major bugs but something works

6 generates some kind of motion tree

8 Very slow or tiny bug but code looks nice.

10 Everything runs smoothly and correctly. The implementation is roughly on par with the reference solution.

Graduate Student Vehicle

Graduate students will also plan for a more realistic spaceship-like vehicle that has limited acceleration. The vehicle's state is $\langle x, y, \delta x, \delta y \rangle$, where δx and δy represent velocity (in map units per timestep). During each timestep, the spaceship experiences a constant acceleration, specified by $\langle \theta, s \rangle$, where θ represents an angle (in radians, counter-clockwise) relative to the ship's current direction of motion and s represents magnitude (in map units per timestep per timestep). To calculate the trajectory of the ship given its state and an acceleration, use simple numerical integration: divide each timestep into 20 slices, and at each slice, compute an update to the ship's state using vector addition:

$$\begin{aligned}
\delta t &= 1/20 \\
\theta_{total} &= \text{atan2}(\delta y, \delta x) + \theta \\
\delta x &\leftarrow \delta x + (\delta t \cdot s \cdot \cos(\theta_{total})) \\
\delta y &\leftarrow \delta y + (\delta t \cdot s \cdot \sin(\theta_{total})) \\
x &\leftarrow x + (\delta t \cdot \delta x) \\
y &\leftarrow y + (\delta t \cdot \delta y)
\end{aligned}$$

This will result in the ship tracing out curves, as the direction of acceleration θ_{total} is always computed relative to the current direction of motion as the ship moves. Check for a collision at the vehicle's location after each slice. Note that it's important that you and the validator agree closely on the vehicle's motion.

To steer towards a target location, generate 10 random controls and choose the one that gets the ship closest (without collisions). If all 10 controls collide, nothing is added to the motion tree. To generate a random control, choose a random angle and a magnitude (0 to 0.5 maps units per timestep per timestep).