Assignment 3: Motion Planner
CS 730/830, Fall 2015

Electronic submission due at **11:30pm on Mon, Sept 21**

Hardcopy submission due at **start of class on Tue, Sept 22**

Overview

You will write a program using the RRT algorithm to find robot trajectories in continuous space from a given start configuration to a goal configuration, avoiding obstacles. Your program can stop as soon as it finds its first feasible solution and you do not need to implement path smoothing. We will run on small problems so you shouldn'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. You should check for collisions every 0.25 units along the robot's path.

Graduate students will plan for a more realistic spaceship-like vehicle that has limited acceleration. The vehicle's state is $\langle x, y, \theta, s \rangle$, where θ is the direction the spaceship is going and s is its speed (in map units per timestep). At each timestep, the spaceship can steer by accelerating relative to the way it is facing. For example, if it accelerates in the direction $\pi/2$ radians to the left of where it is facing, then it will go in a counter-clockwise circle. To choose a random control, choose an angle (either $-\pi$ to π or 0 to 2π, as you prefer) and a magnitude (0 to 0.5 maps units per timestep per timestep). The acceleration takes effect throughout the entire timestep. To compute the trajectory of the robot, you can use naive integration with mini-timesteps: at every mini-timestep, modify the position by the velocity, check for collision, and modify the velocity by the acceleration (scaled for the size of your mini-timesteps). You should be OK with 10 mini-timesteps per timestep. To 'steer' towards a sampled target configuration, try 10 random accelerations and take the one whose final resulting state is closest to the goal. The goal can be considered achieved when the robot is within distance 1.0.

Because both the undergraduate and graduate vehicles are points, collision checking just means finding whether the point is in an obstacle. The map representation, discussed below, will make this easy.

You should use a goal bias: your 'random target sample' should be the goal configuration with probability 0.05.

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 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 $\langle x = 03.8, y = 1.1 \rangle$, where $\langle 0, 0 \rangle$ is the bottom left corner. Graduate students can assume that the robot's initial velocity is 0, so the same initial state specification can be used.

No command-line arguments are necessary.

Output

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. For example:

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

For graduate students, a solution trajectory is a list of $\langle \mathit{state}, \mathit{control} \rangle$ pairs, one per line. Since the vehicle state is 4 numbers and a control is 2 numbers, this makes 6 numbers per line. No control needs to be listed on the last line with the goal state.

Execution

Please use `make.sh` and `run.sh` scripts just like with assignments 1 and 2.
We will supply:

`*.vw` a few example benchmark problems.

`rrt-reference` a sample solution using the undergraduate vehicle.

`rrt-grad-reference` a sample solution using the graduate student vehicle.

`rrt-validator` runs your program, validates its output, and displays the solution plan. The validator is written in Python 2.7 and the graphics depend on `pygame`, a Python package which is installed on agate (version 1.9.1). The validator takes as a command-line argument the pathname of a planner to run and expects a problem instance on standard input. It also takes flags `--undergrad` and `--grad` to determine which vehicle you are using, as in:

```
rrt-validator --undergrad run.sh < small-1.vw
```

To visualize the solution, add the `-v` flag.

The validator can also be run in a 'practice mode', which is helpful for debugging. In this mode, activated by the `-practice` flag, the output of your planner doesn't need to be a solution trajectory, but just a list of tree edges to visualize. For undergrads, each edge is a start and end state (so 4 numbers per line). For grads, each edge is a state and a control (so 6 numbers per line).

Submission

Electronically submit your solution using the instructions on the course web page, including your source code as well as a transcript of your program running with the validator on the supplied example problems.
Submit a brief write-up with your final hardcopy solution answering the following questions:

1. Describe any implementation choices you made that you felt were important. Mention anything else that we should know when evaluating your program.
2. What suggestions do you have for improving this assignment in the future?

Attach to the write-up a hardcopy of the transcript of your code solving the example cases using the validator, and a listing of your source code (2 pages per page, as with `a2ps -2`).

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 roughly on par with the reference solution.