Computer Exercise 3

This exercise will focus on motion planning. It is a pure Python exercise (no ROS this time).

The coding environment is going to be Jupyer Notebook with Python 3.6 (or higher), NumPy and Matplotlib. The easiest way to setup this environment is with the Anaconda installer. Note that you do not need the Ubuntu VM but can instead install it natively on Windows or Mac.

If this is your first time with Jupyter notebook, refer to this guide.

General instructions:

- Submission is in pairs only.
- Submission deadline: 1/7/2021, 23:59
- Questions related to the exercise should be posted in the "Discussion on Computer Exercises" forum on Moodle.
- You are required to submit your Jupyter notebook (.ipynb file) electronically in Moodle. The notebook should include your code and explanations (in markdown cells) as required.

Download the files attached to this exercise (*computer_exercise_3.ipynb*, *map1.txt*, *map2.txt*) and place the three of them at the same directory on your computer. Launch the Jupyter notebook server and open *computer_exercise_3.ipynb*.

Part I: Environment Definition

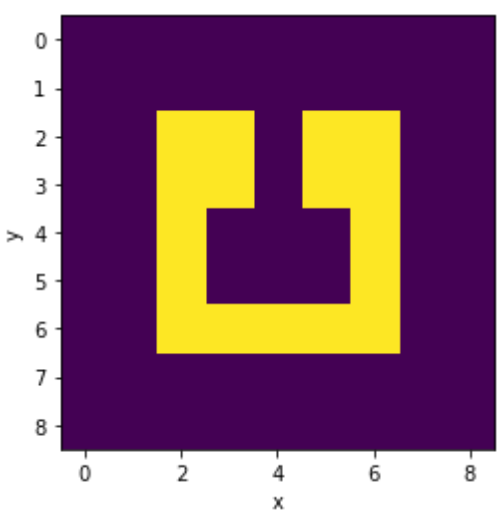
We want to solve the motion planning problem for a mobile planar point robot in a two-dimensional environment whose map is given. The configuration variables are the robot's pose variables x, y.

1. Explain why in this case no extra effort is needed in order to calculate the C-space. Add your answer to the Jupyter notebook.

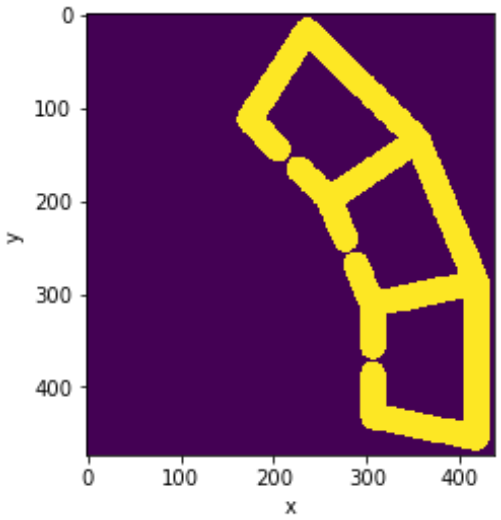
The configuration data structure is defined in the code as *Config*.

2. Complete the Python code line which calculates the Euclidean distance between the configurations (2, 3) and (4, 6).

You have been provided with two text files representing two different maps (*map1.txt* and *map2.txt*). Each *txt* file contains a binary matrix in which `1` denotes an occupied cell (obstacle) whereas `0` denotes a free cell. The cell (i, j) of the matrix represents the area [i-0.5..i+0.5; j-0.5..j+0.5] in the continuous space.



map1



map2

In the visualization above, purple represents the free spaces while yellow represents the obstacles. Note that map1 is significantly smaller than map2; use the former for development and testing of your implementation.

The *MapEnvironment* class implements the environment-specific functionalities. It is instantiated by passing the map file name as an argument, for example:

```
planning_env = MapEnvironment('map1.txt')
```

Note that as part of the *__init__* of the class, the map is shown.

3. Create a copy of *map1.txt* and name it *map3.txt*. Manipulate the new text file to create a new map at your will. Visualize the new map by instantiating *MapEnvironment*. Include the map visualization in the submitted notebook.

Certain functionalities are already implemented in *MapEnvironment*, others will be implemented by you.

4. Implement the function *compute_distance* (use the Euclidean distance as before). The function should return a *float* number. In the appropriate cell in the notebook, calculate the distance between two arbitrary configurations on map1.
5. Implement the function *check_edge_validity*. This function should return *True* if the straight line connecting two configurations is valid and *False* otherwise. You will have to sample points along the connecting line and use the function *check_state_validity* to check if these are valid. In addition, provide two examples on map1: in the two appropriate cells in the notebook, call this function with one valid edge and one invalid edge. Use *visualize=True* to visualize this edge on the map.

6. Run the notebook cell which uses *visualize_plan*. You will see the suggested plan visualized on the map. Slightly modify the plan at your will such that it still connects the same start and goal points but through another valid path. Visualize your new suggested plan.

Part II: RRT

In this part you will implement the RRT algorithm that will find a path between given start and goal points on the map.

The class *RRTTree* is a data structure which implements a tree. Use it for your implementation. Its functions are as follows:

- *get_root_id:* returns the ID of the root in the tree [*int*].
- *get_nearest_vertex*: returns the nearest state ID in the tree [*int*] and the state itself [*config*] (arguments: configuration [*Config*]).
- *add_vertex:* adds a state to the tree (arguments: configuration [*Config*]).
- *add_edge:* adds an edge in the tree (arguments: start state ID [*int*], end state id [*int*]).

Implement the core logic of RRT in the *RRTPlanner* class. The function that you need to implement is *plan*. Its mandatory arguments are the start configuration [*Config*] and the goal configuration [*Config*], but you can define additional optional arguments with default values. The function should return the planned path [*list* of *Config* elements]. If no path was found, return *None*. For better readability, it is recommended to define additional subroutines as class functions and call them from *plan*.

Your implementation should rely on the pseudocode and the implementation details that you have seen in class (slides 4-7). You are allowed to apply post-processing and goal biasing if you want. You are **not** allowed to implement RRT-connect, RRT* or any other variant of the vanilla RRT. As you have seen in class, RRT has several hyper-parameters that require certain tuning for improved performance (e.g., the number of iterations, the steering parameter, etc.); you are encouraged to play with them.

Evaluation and Competition

The function *evaluate* takes the planning environment and start and goal configurations. It runs your *RRTPlanner* and evaluates it. It returns three values:

- *invalidity_ratio*: the ratio of invalid points among the ones sampled along the planned path (the sampling is random and done uniformly).
- *total_cost*: the total length of the planned path
- *duration*: the run time (in seconds) of RRT.

In addition, this function visualizes the planned path on the map.

Your submitted code will be tested with *evaluate* for several values of start and goal and on both maps. **Do not change this function!** Any planned path with *invalidity_rate* below 0.05 is acceptable.

In addition, the performance of your implementation will compete against all others. Try to achieve shorter paths with shorter run duration. **2 bonus points for the final grade will be added for the winning team**.

Good luck!