HW 3—Sampling-based inspection planning

1 General guideline

This homework will cover the topic of inspection planning which is a "wet" programming exercise (2). Writeups must be submitted as a PDF. LaTeX is preferred, but other typesetting methods are acceptable. Code for the programming component must be submitted in a zip archive. Plots generated as part of the programming component should be included in the writeup. The homework can be submitted in pairs.

2 Robot Manipulator: Motion Planning and Inspection Planning

In the previous assignment, you were asked to implement the RRT algorithm for a point robot. Here, you will experiment with the challenges when working with a manipulator robot, and will be required to complete both motion-planning and inspection-planning tasks.

2.1 Code Overview

The starter code is written in Python and depends on numpy, matplotlib, imageio and shapely. We advise that you work with virtual environments for python. If any of the packages are missing in your python environment, please use the relevant command:

```
pip install numpy
pip install matplotlib
pip install imageio
pip install Shapely
```

You are provided with the following files:

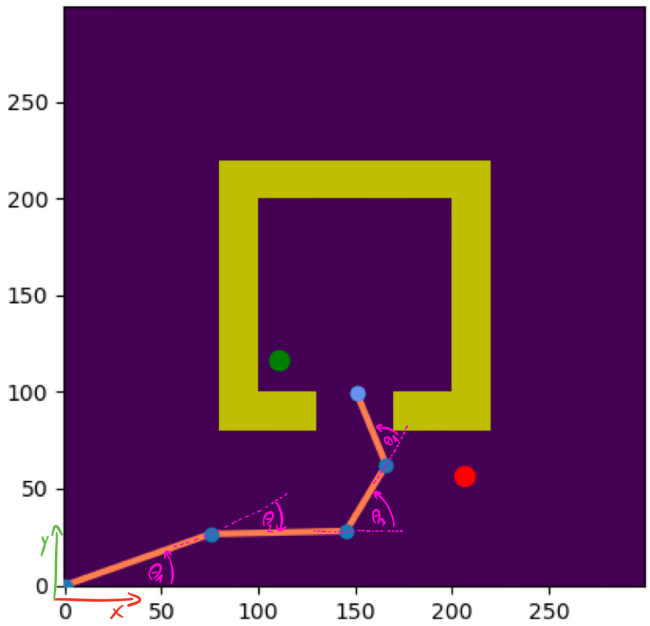
- **run.py** - Contains the main function. Note the command-line arguments that you can provide.
- **Robot.py** - Robot management class, containing all functions related to the robot, including information on links and end-effector.
- **MapEnvironment.py** - Environment management class, containing all functions related to the environment itself.
- **map_mp.json** - An environment (map) file for a motion-planning task. Contains the map's dimensions, (workspace) obstacles, and start and goal locations (in the C-space !).

- **map_ip.json** - An environment (map) file for an inspection-planning task. Contains the map's dimensions, (workspace) obstacles, (workspace) inspection points, and start location (in the C-space !).
- **RRTMotionPlanner.py** - RRT planner for motion planning. Logic to be filled by you.
- **RRTInspectionPlanner.py** - RRT planner for inspection planning. Logic to be filled by you.
- **RRTTree.py** - Management class for the RRT tree for both motion and inspection planning.

The following are examples of how to run the code for motion planning/inspection planning tasks:

```
python run.py –map map_mp.json –task mp –ext_mode E2 –goal_prob 0.05
python run.py –map map_ip.json –task ip –ext_mode E2 –coverage 0.5
```

2.2 Robot Modelling (20 points)



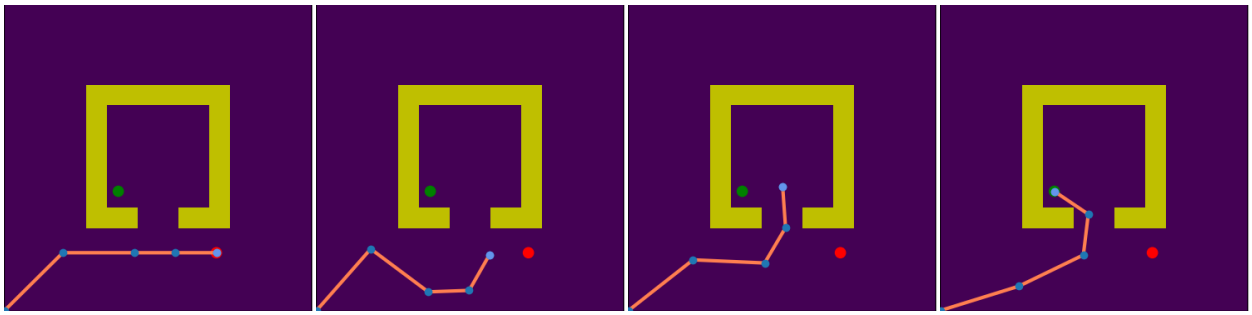
In this exercise, we will work with a manipulator robot. The robot has four degrees of freedom with four links of fixed size (defined in `Robot.py`). You will start by implementing a few features that are necessary to operate a robotic manipulator. In your writeup, **detail your approach** for implementing each of the following functions in `Robot.py`:

1. `compute_distance` - Your task is to compute the Euclidean distance between the two given configurations, and return the result.

2. `compute_forward_kinematics` - Given a configuration (where angles are in radians and ranged in $[-\pi, \pi]$), your task is to compute the position of each link, including the end-effector (and excluding the origin), and return it. Use the notation in the figure to understand the geometric contribution of each angle to the location of the link. Make sure that the output of this function is a numpy array of the shape $(4, 2)$, where the row i represents the i'th link, and contains the x-axis and y-axis coordinates w.r.t. the environment's origin ($[[x_1, y_1], \ldots, [x_4, y_4]]$). **Clarification:** The last row should represent the end-effector, so make sure you receive the start and goal locations that are in the figure.

3. `validate_robot` - For a given set of all robot's links locations (including the origin), your task is to validate that there are no self-collisions of the robot with itself. Return `True` for no self-collisions (**Notice!** you are not required to check for collisions with obstacles or with the environment's boundaries! This is already implemented in the `MapEnvironment.py`).

2.3 Motion Planning (30 points)



In the previous assignment, you implemented the Rapidly-Exploring Random Tree (RRT) for a point robot in a 2D world. Here, you will start by implementing the same algorithm for the robotic manipulator. In the file `RRTMotionPlanner.py`, implement the following functions:

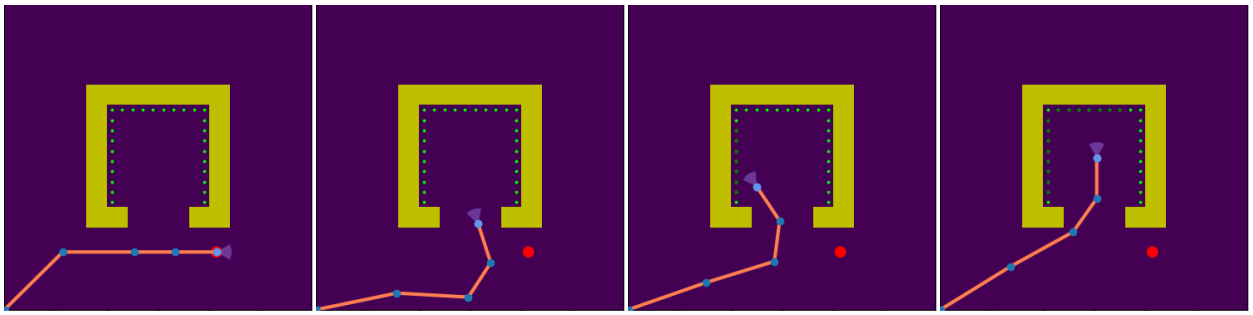
1. `extend` - With the same instructions as before - $E1$ for extending all the way untill the sampled point and $E2$ for extending to the sampled point only by a step-size η).

2. `compute_cost` - Compute the cost of the given path, as the sum of the distances of all steps.

3. `plan` - The body of your planning algorithm with goal biasing. Make sure that the output is a numpy array of the calculated path in the configuration space (should be a shape of $(N, 4)$ for N configurations, including start and goal configurations).

Report the performance of your algorithm (cost of the path and execution time) for goal biasing of 5% and 20% averaged over 10 executions for each, and attach the visualizations created by `visualize_plan`.

2.4 Inspection Planning (50 points)



In inspection planning, we are require to inspect a set of points of interest (POI) and not reach a predefined goal. In this exercise you are required to adapt your RRT implementation to handle the inspection task (this is slightly easier than the IRIS algorithm taught in class). A vertex v_i in the RRT tree, will now also include the subset of POI that were seen so far when following the path from the root of the RRT tree to the configuration associated with the vertex and set $I_i = \{p_1, \ldots, p_i\}$ to be this subset of POI. Namely, if in the (vanilla) motion-planning problem we defined a vertex as $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}]$, it will now be $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}, I_i]$. Moreover, if the edge (v_{i-1}, v_i) exists, then the set of POI inspected must be $I_i = I_{i-1} \cup S(v_i)$, where $S(v_i)$ is the set of inspection points that the robot can see at step vertex v_i.

Next, we define the *coverage* of a vertex v_i as $\frac{|I_i|}{|I_{\max}|}$, where $I_{\max}$ is the set of all POI. Namely, a vertex with coverage of 1.0 corresponds to a path along which all POI have been inspected.

To this end, the stopping condition of your planner will be reaching not a goal vertex but a vertex with a desired coverage (use the argument `--coverage` as shown in Sec. 2.1) and return the path to it. Implement the following in `RRTInspectionPlanning.py`:

1. `extend` and `compute_cost` - These can stay the same as in the motion-planning task.
2. `compute_union_of_points` (in `MapEnvironment.py`) - Compute the union of two sets of inspection points.
3. `plan` - The body of your planning algorithm. Make sure that the output is a numpy array of the calculated path in the configuration space (should be a shape of $(N, 4)$ for N configurations, including start and end configurations).

Report your performance (cost of the path and execution time) for coverage of 0.5 and 0.75, averaged over 10 executions for each, and attach the visualizations created by `visualize_plan`.

Discuss the changes you were required to do in order to compute inspection planning. Notice that for higher coverage the search may take a few minutes, so think about how you can improve your search!

Hint: In motion planning we biased the sampling towards a goal vertex. If there is no goal, how can we bias the sampling to improve coverage?

Good luck!