HW 2—Sampling-based motion planning

1 General guideline

This homework will cover the topic of sampling-based motion planning. It includes two "dry" exercise (Sec. 2 and 3) and two "wet" programming exercise (Sec. 4 and 5). Writeups must be typed and 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.

2 Distribution of points in high-dimensional spaces (10 points)

In this exercise we will gain intuition on the behavior of how points are distributed in high-dimensional spaces. Throughout this exercise, think of the implication to sampling-based planners and especially asymptotically-optimal ones. You will need to use the formula for the volume of a d-dimensional unit ball (ball of radius one that resides in a d-dimensional Euclidean space). See `https://en.wikipedia.org/wiki/Volume_of_an_n-ball`.

Warmup (0) points

Please use your intuition only (and be honest ;-) to answer the following questions:

1. In a 2-dimensional unit ball (namely, a disk), how much of the volume is located at most 0.1 units from the surface (see Fig. 1)?
   (a) Roughly 1%
   (b) Roughly 10%
   (c) Roughly 20%
   (d) Roughly 60%

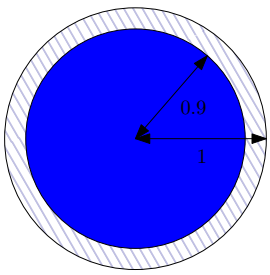


Figure 1: A 2-dimensional unit ball (blue) where the volume located at most 0.1 units from the surface is depicted in dashed lines.

2. In a 9-dimensional unit ball, how much of the volume is located at most 0.1 units from the surface?

   (a) Roughly 1%

   (b) Roughly 10%

   (c) Roughly 20%

   (d) Roughly 60%

Exercise (10 points)

Define the fraction of the volume that is ε distance from the surface of a d-dimensional unit ball as $\eta_d(\varepsilon)$.

Plot $\eta_d(\varepsilon)$ as a function of d for $\varepsilon = 0.2, 0.1, 0.01$ for $d = 2 \dots 10$. Namely, the x-axis should be d and the y-axis should be $\eta_d(\varepsilon)$.

Discuss the implications to reducing the connection radius required for a sampling-based algorithm to maintain asymptotic optimality.

3 Tethered robots (20 points)

We consider a 2D point robot translating amidst polygonal obstacles while being anchored by a tether to a given base point p_b. The robot may drive over the cable, which is a flexible and stretchable elastic band remaining taut at all times. We study the problem of constructing a data structure that allows to efficiently compute the shortest path of the robot between any two given points p_s, p_t while satisfying the constraint that the tether can extend to length at most L from the base.

1. (**2 points**) describe the structure of a path from a start to target configuration (robot location + tether description).

2. (**2 points**) Suggest an efficient way to encode the tether's description. Note that the robot can be at the same location but with completely different tether descriptions (e.g., circling once or twice around an obstacle) and what we need to define is the homotopy class of the tether.

3. (**6 points**) The *homotopy-augmented graph* G_h of a graph $G = (V, E)$ encodes for each vertex of G, all homotopy classes that can be used to reach the vertex using a tether of length L. Describe an approach to compute the homotopy-augmented graph of a given graph using a Dijkstra-like algorithm. Describe the nodes, how they are extended and when the algorithm terminates.

4. (**10 points**) Now we will now use the notion of a homotopy-augmented graph to efficiently answer queries solving the motion-planning problem for a point tethered robot.
   A query is given in the form of two points p_s, p_t and the h-invariant w_s describing the tethered placement at p_s. In order to compute a path (if one exists) between p_s and p_t with an original tether placement defined by w_s, we need to traverse the homotopy-augmented graph G_h^{vis} (the homotopy-augmented graph of the visibility graph).

- What new vertices do we need to add the homotopy-augmented graph G_h^{vis} to account for p_s and p_t?
- What new edges do we need to add the homotopy-augmented graph G_h^{vis} to account for p_s and p_t?
- How can we use the newly-constructed graph to efficiently solve our motion planning problem?

4 2D Robot Manipulator: Building Blocks and Basic Motion Planning

In this assignment, you will be tasked with implementing the PRM algorithm. Specifically, you will experience the challenges when working with a manipulator robot.

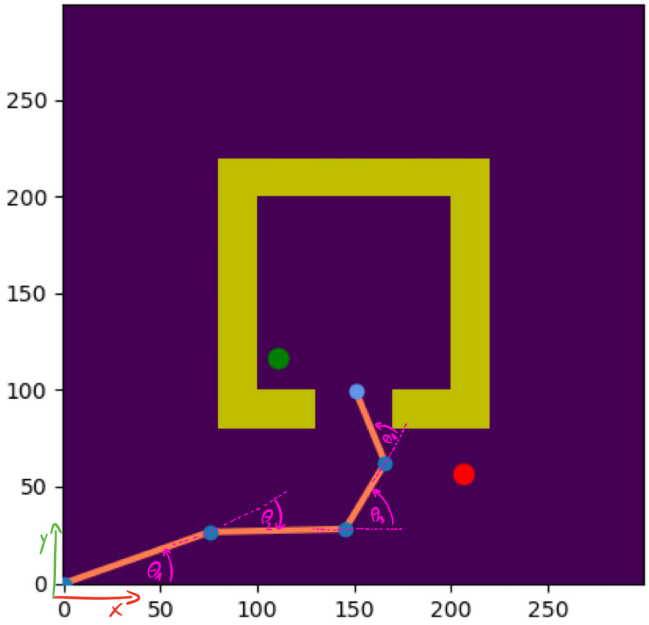
4.1 Code Overview

The starter code is written in Python and depends on numpy, matplotlib, imageio and shapely. We recommend you to 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.
- **twoD/building_blocks.py** - Robot management class, containing all functions related to the robot, including information on links and end-effector.
- **twoD/environment.py** - Environment management class, containing all functions related to the environment itself.
- **twoD/map_mp.json** - A map JSON file for a motion planning task. Contains map dimensions, (workspace) obstacles, and start and goal locations (in the C-space!).
- **twoD/map_ip.json** - A map JSON file for an inspection planning task. Contains map dimensions, (workspace) obstacles, (workspace) inspection points, and start location (in the C-space!).
- **twoD/prm.py** - A file where you will be implementing the PRM algorithm.



4.2 Building Blocks (10 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 `building_blocks.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 `building_blocks.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`).

4.3 PRM (30 points)

In this part, we will implement the PRM algorithm for the manipulator robot from the previous section. In order to do that please fill in the missing parts of `PRM.py` and `environment.py` according to the following instructions:

1. `gen_coords` - Your task is to generate n random samples called milestones. Make sure that the n configurations generated are collision free. **Hint**: use `building_blocks.py`.
2. `add_to_graph` - Given a configurations list, add the configurations to the PRM garph. In order to add the relevant edges, implement also the function `find_nearest_neighbour`. **Hint**: use nx.Graph and KDTree (you can read about them online).
3. `shortest_path` - Implement Dijkstra's algorithm to find the shortest path. Feel free to adjust the code you wrote on HW1.
4. `run_PRM` - Implement the PRM algorithm using the previously implemeted functions.
5. `run.py` - **Add** an example for computing a path and **visualize** it using the relevant function from `visualizer.py`. **Add** the image to your report.

Plots & discussion. Now that we have a working PRM implementation, we would like to study its properties. **create** the following two plots:

P1 Plot the cost of the path created by the PRM as a function of n, the number of generated nodes. Do this for the following values of k (the number of nearest neighbors to connect each node to): $\{5, 10, \log(n), 10\log(n), n/10\}$. Have all these plotted on the same figure. n should contain the following values $\{100, 200, 300, \ldots, 700\}$. The start and goal configurations should be $[0.78, -0.78, 0.0, 0.0]$ and $[0.8, 0.8, 0.3, 0.5]$, respectively.

P2 Plot a similar figure but here the x-axis should be the algorithm's runtime (using the same values for k and n).

Add the plots to your report and **discuss** the results in detail.

TIP: In order to shorten the time needed to create this graph you can (and should) add 100 new nodes each time and not calculate n new nodes each time. In addition we recommend to generate those nodes in advance (or on the first round) and then just add the correct number of nodes to the graph (and connect the edges of course).

5 3D Robot Manipulator: Building Blocks

In the next section and the following homeworks in this semester, we are going to use a 3D manipulator and get some hand-on experience with motion-planning algorithms and run their results on a real manipulator.

In this part we will develop the basic algorithmic building blocks required to implement a sampling-based motion-planning algorithm.

5.1 Software Infrastructure

In addition to the files in previous section, specifically relevant for this part are run.py and RRTTree.py, you can find the following files inside threeD directory: –

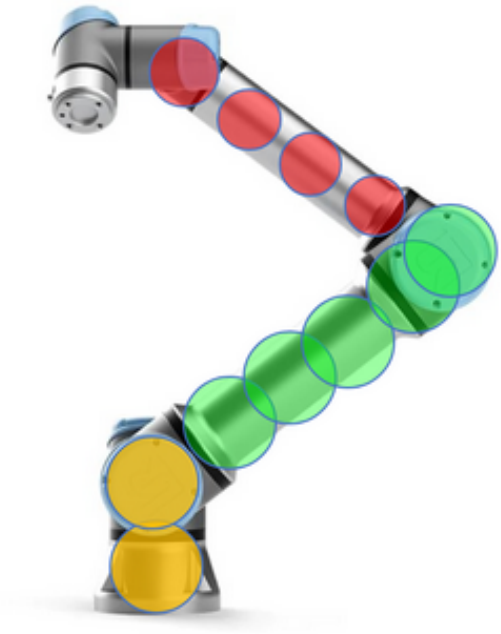
- **threeD/building_blocks.py** - includes the Building_Blocks class which implements basic functions used by the planner (e.g. sampling, collision detector, local planner).
- **threeD/environment.py** - includes the Environment class which defines the positions of the spheres that encapsulate the obstacles in the environment.
- **threeD/kinematics.py** - includes (i) `UR5e_PARAMS` class which defines the manipulator's geometry and (ii) `Transform` which implements the transformations from each link of the manipulator to the base_link.
- **threeD/inverse_kinematics.py** - includes methods that allow to obtain a configuration given the position and orientation of the end-effector.

5.2 Implement and Questions (30 points)

In this part we are going to fill in the missing parts in buildingBlocks3D.py:

1. **Sampling** - fill in the missing part of `sample()`. Specifically, each joint of the manipulator should be in the interval defined in `UR5e_PARAMS.mechanical_limits`. Recall that we often employ goal biasing (this is a planner's parameter named p_bias). Thus, with probability p_bias the function should return the goal configuration and with probability $(1 - p\_bias)$, the function should return a random configuration.
2. **Collision detection** - recall that a collision detector (CD) is used to determine whether a given configuration is valid or not. The collision detector `is_in_collision()` checks for two types of collisions: (i) internal collisions between different robot links and (ii) external collisions between the robot links and the obstacles. Here, we will use a simple approach where obstacles and robot links are each modeled as a collection of spheres. We say that a collision occurs when spheres of different bodies (i.e., two different links or a link and an obstacle) intersect. To this end, we need

to (i) decide how to model the robot and the obstacles as a collection of spheres (e.g., what radius should we use) and (ii) how to compute intersection between spheres (think about the sphere's radii and their respective centers). Note that in general, we are willing to have false positives but not false negatives. Namely, it is acceptable that the CD returns that a collision occurs even when it does not but it is unacceptable to return that no collision occurs when one does.



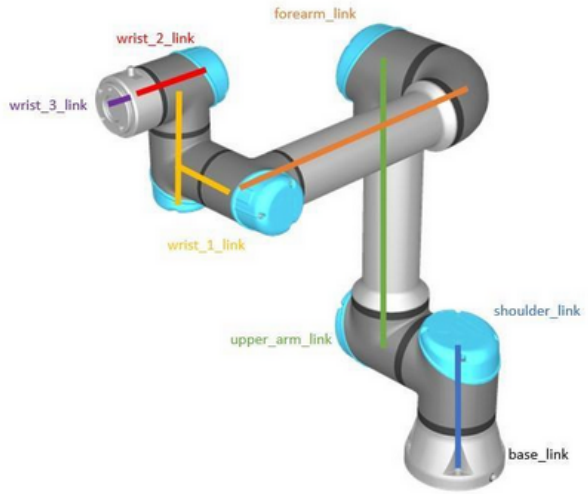
(a) Assume that a link is modeled as a cylinder with a radius r, and length 10r and that all spheres have equal radii. Furthermore, assume for simplicity that the center of the link is located along the x-axis with one endpoint at (0,0,0) and the other at (10r,0,0). For the following, give the location of each sphere together with their radius in order to ensure that there are no false negatives while minimizing the number of false positives.

   i. One sphere
   ii. Two spheres
   iii. Five spheres
   iv. Ten spheres

(b) **Discuss** in general terms the tradeoff and effect of the number of spheres and the chosen radius.

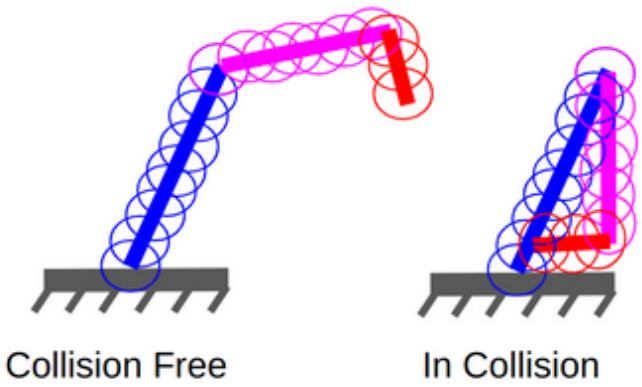
(c) The `inflation_factor` parameter provides a trade-off between the number of spheres and the accuracy. Set values $\{1, 3\}$ for `inflation_factor` in the `ur5e_params` class constructor and run the script `run.py` to visualize the configuration $[-0.694, -1.376, -2.212, -1.122, 1.570, -2.26]$[radians Add snapshots depicting the results. From now on, work with `inflation_factor` value of 1.

3. To determine the global coordinate system of each sphere we use transformations. These allow to map the location of a sphere, given in the robot's local coordinate frame to a global coordinate frame given a specific configuration. We define the coordinate system of the base_link (see Figure below) as a global coordinate system. With all sphere coordinates referenced to the common coordinate system we can check if a given configuration is valid.



Consider the following simple 2D example where we assume that each link can rotate freely relative to the previous link.

(a) Given a manipulator geometry, implement the function `is_in_collision()` in `building_blocks.py`. Here, the class `global_sphere_coords` is a dictionary with "keys" being the link name and "items" being a list of sphere coordinations along the link. **Provide** examples (i.e., a configuration and a snapshot) of (i) one configuration which is collision free and (ii) one configuration that is in collision.



(b) Extend the collision detector to consider obstacles and the floor. Here we treat the floor (i.e., $z\_cord = 0$) as an obstacle. set `env_idx` to be 1 in `Environment` constructor. **Provide** examples (i.e., a configuration and a snapshot) of (i) one configuration which is collision free and (ii) one configuration that is in collision with the obstacle.

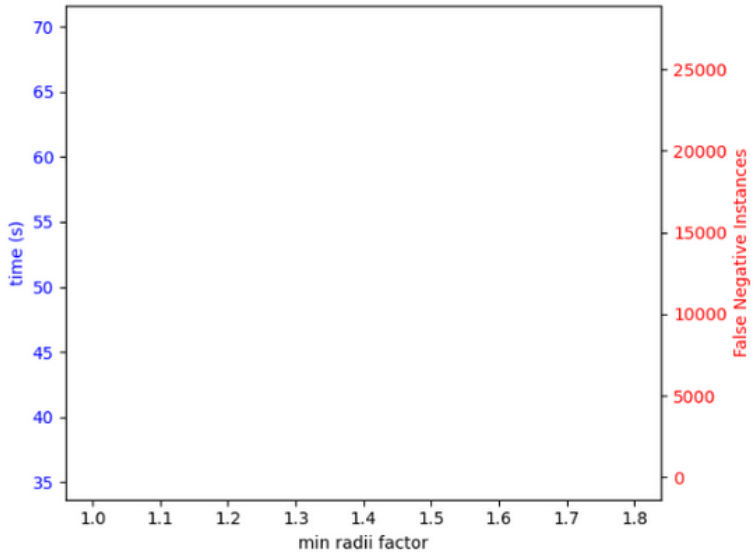
4. To determine if a transition between two configurations is valid we have to check for collisions in intermediate configurations. Implement the `local_planner()` function which takes two configurations q and q' and checks for collisions in intermediate configurations. The intermediate configurations are determined by the parameter `self.resolution`. Set the minimum value of configurations to check to be 2, in this case the local planner checks for collisions only for the provided configurations and not any intermediate ones. If a transition is valid the `local_planner()` returns `True`.

   Now, make sure `env_idx` is set to 1 in `Environment` class constructor and test the local planner with the following configurations:
   $conf_1$: [80, -72, 101, -120, -90, -10] [deg],
   $conf_2$: [20, -90, 90, -90, -90, -10][deg].
   convert to radians using the `numpy.rad2deg()` function.

   In this case, the manipulator can not transition from $conf_1$ to $conf_2$ without colliding the obstacle. Change the parameter self.resolution such that the local planner returns (i) `True` and (ii) `False`. **Report** the values you chose and how many configurations the local planner tested for each value. Finally, Set the minimum value of configurations to check to be 3.

5. In the file **random_samples_100k.npy**, you are given a set of $100,000$ random configurations. For each one, compute if it is in collision using the different inflation factors (e.g., an inflation factor of 1.5 inflates the minimal radius by 1.5). The inflation factors and template for the solution are provided in the file **exercise_5_graph.py**.

   Plot the overall computation time as a function of the radius used and the number of false-negatives as a function of the radius used in range of inflation factor [1.0, 1.8] in intervals of 0.1. Here, we treat the inflation factor = 1.0 as ground truth (the false negative for inflation factor = 1.0 is 0). For the other inflation factors values, we compare with respect to that ground truth.


70
65
60
55
50
45
40
35
time (s)
25000
20000
15000
10000
5000
0
False Negative Instances
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
min radii factor