CS 4610/5335: Robotic Science and Systems (Spring 2024) X. Zhi Tan
Northeastern University version: 1.1.0

HW 2: Motion Planning in CSpace

Please remember the following policies:

- Submissions should be made electronically via the Canvas. Please ensure that your solutions for both the written and/or programming parts are present and zipped into a single file.
- You are welcome to discuss the programming questions with other students in the class. However, you must understand and write all code yourself. Also, you must list all students (if any) with whom you discussed your solutions to the programming questions.
- Any use of generative AI must follow the class' generative AI policy.

Depending on the configuration of your computer, there are different ways to run the code in this assignment. We recommend using conda with the attached environment file.

Recommended Steps:

- Install the latest version of mini-conda from conda's website: conda.io
- Download `exercise2.zip` from Canvas. Unzip the file to your preferred directory (e.g., `/Users/Zhi/RSS/HW2`).
- Navigate to the folder using your terminal
- Create a conda environment using the following command: `conda env create --name "rss-hw2" --file=rss_env.yml`
- Open the folder using your favourite IDE and select `rss-hw2` as your intepreter.

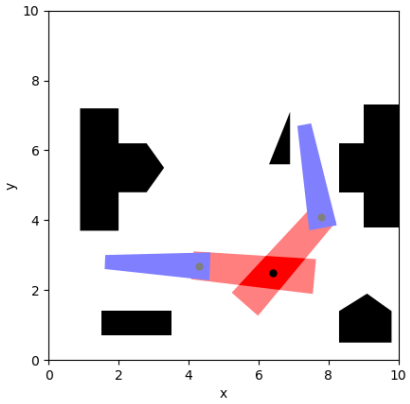
The following are common build or run errors that you might run into:

- `AttributeError: module 'pybullet' has no attribute 'performCollisionDetection'`
  You are likely using an older version of pybullet. Make sure the pybullet version is 3.2.6.
- `ModuleNotFoundError: No module named 'shapely'`
  You likely did not activate the conda environment. Make sure you run `conda activate rss-hw2` before running of the following code.

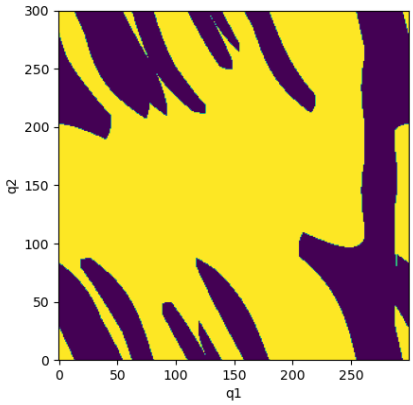
ChangeLog

- v1.1.0
  - [**Code - CSpace**] Fixed the display code for cspace in `hw2_cspace.py` such that the cspace matrix (generated by C2-C7) has q1 as the first dimension and q2 as the second dimension. We also corrected the corresponding code to display the path in C4. The path should now have the format: [[q1_0, q2_0], [q1_1, q2_1], ...] Thank you to your classmates for pointing this out.
  - [**Code - CSpace**] In `C2-C7`, we improved the function documentation by fixing typos, clarifying potential confusions, and adding examples.
  - [**Code - Motion**] Fixed an evaluation bug for M1 in `hw2_motion.py` where 0 was an invalid option when it should be. Thank you to your classmates for pointing this out.
  - [**Writeup**] In the writeup of C6, we clarified which figure we are referencing.

1 Configuration Space



Robot workspace

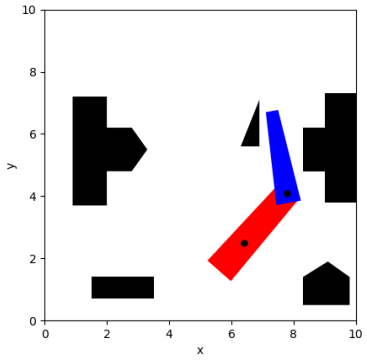


Robot configuration space

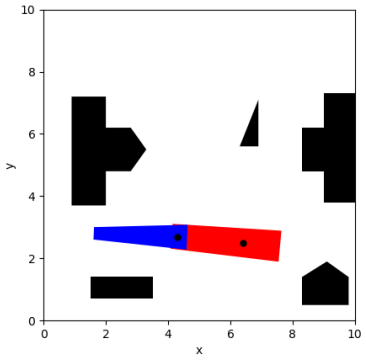
For this section, we will use the 2-DOF 2-link rotational planar robot shown above. The arm is attached to a table surface that we are viewing from a top-down perspective. There are obstacles on the table as shown by the shaded regions. The goal is move the arm from start configuration (1, purple) to the goal configuration (2, orange), without colliding into any obstacles. Assume that the illustrated workspace is 10 units wide and 10 units high; the origin of frame $\{0\}$ is at the bottom-left corner, with x_0 and y_0 axes as shown. The first link of the arm is attached to the table at $(6.4, 2.5)$, the origin of frame $\{1\}$. Link 2 is attached to link 1 at $(2.1, 0)$ with respect to frame $\{1\}$, the origin of frame $\{2\}$. The configuration of the arm is given by $q = (q_1, q_2)$, where q_1 is the angle between x_0 and x_1, and q_2 is the angle between x_1 and x_2. Both joints may rotate between 0 and 2π radians. For example, the start configuration (1, purple) is $q_{\text{start}} = (0.85, 0.9)$, and the goal configuration (2, orange) is $q_{\text{goal}} = (3.05, 0.05)$.

In the `cspace` folder, you will find:

- `hw2_cspace.py`: Main function. Run `python hw2_cspace -q <questionNum>` with an appropriate question number (1–7) to run the code for that question. Do not modify this file! (Feel free to actually make changes, but check that your code runs with an unmodified copy of `hw2_cspace.py`.)
- `C1.py` – `C7.py`: Code stubs that you will have to fill in for the respective questions.
- `q2poly.py` – Code stub for that you will need to fill in for C1.
- `helper_functions.py`: Helper functions for visualization.



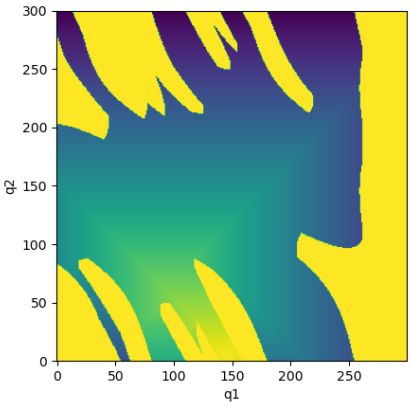
C1: Start configuration.



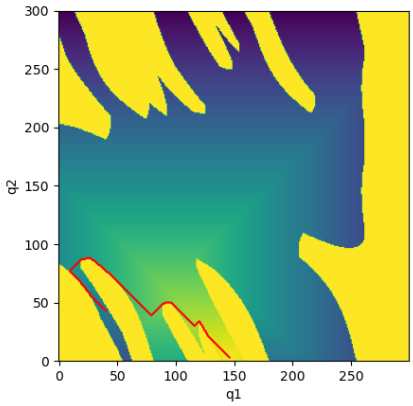
C1: Goal configuration.

C1. **2 points.** Plot the robot in the workspace, as shown above. The demonstration code in `C1.py` shows how to plot the robot at the zero configuration ($q = (0, 0)$). You will need to make appropriate transformations to both links' polygons and their pivot points (frame origins). Consider filling in `q2poly.py` first, which is a useful helper function for C1 and future questions. When you run the code `python hw2_cspace.py -q 1`, you should see the figures above.

C2. **2 points.** Convert the problem into configuration space by discretizing the configuration space, and checking for collisions at each discrete grid point. Using the specified grid for each axis given in `q_grid`, compute whether the configuration at each point is in collision or not, by intersecting the links' 2-D polygon with the obstacles' 2-D polygons. Assume that the robot is never in collision with itself. The resulting matrix should look similar to the configuration space diagram shown on the slide.
Hint: Checkout of the methods available for polygon constructed using the Shapely library.



C3: Distance transform from goal configuration.



C4: Path from start to goal.

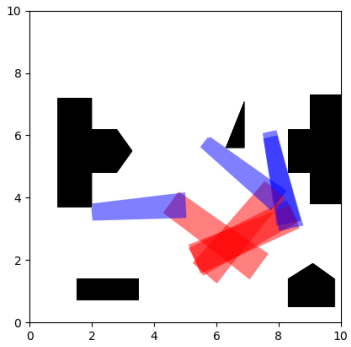
C3. **2 points.** Given a specified goal configuration and the configuration-space grid from C2, compute the distance transform from the grid point nearest to the goal.

C4. **2 points.** Using the distance transform from C3, find a path from the specified start configuration's closest grid point to the goal's grid point. Descend the distance transform in a greedy fashion, breaking ties with any strategy you wish. Diagonal neighbors are allowed.

C5. **1 point.** Convert the path in grid point indices, found in C4, into a path in configurations. Remember to include the actual start and goal configurations. This should trigger an animation where the robot moves from the start to the goal configuration.

C6. **2 points.** Unfortunately, since collisions have only been checked at discrete grid points, we cannot guarantee that the segments between those grid points are collision-free. In fact, the trajectory we found in our implementation of C5 contains four collisions, shown in the right above. These collisions can be detected by considering the *swept volume* between two configurations. The swept volume can be approximated by appropriate convex hulls of the robot links' 2-D polygons. Check if any segments of the trajectory you found in C5 are in collision, plot the violating swept volumes similar to the diagram below, and return the number of collisions. Depending on how you found your trajectory, it may not actually have any such swept-volume collisions!
Useful functions: `shapely.union_all` and `.convex_hull`

C7. **1 point.** Most of the collisions above were caused by planning a path that was too close to obstacles. One simple conservative way to avoid such collisions is to pad the obstacles by a small buffer zone. Pad the obstacles in configuration space by one grid cell (including diagonal neighbors), and verify through the animation that the resulting trajectory does not contain any swept-volume collisions.



C6: Swept-volume collisions along the path.

2 Sampling-based Motion Planning

In the `motion_planning` folder, you will find:

- `hw2_motion.py`: Main file - run `python hw2_motion.py -q <questionNum>` with an appropriate question number (0–6) to run the code for that question. Do not modify this file! (Feel free to actually make changes, but check that your code runs with an unmodified copy of `hw2_motion.py`.)
- `M0.py` – `M5.py`: Code stubs that you will have to fill in for the respective questions.
- `robot.py` – Defines the robot and its basic IK/FK/collision checking functionality.
- `\Robot` – the URDF and additional scripts for the robot.

In this part of the assignment, you will implement two sampling-based motion planning algorithms, the probabilistic roadmap (PRM) and the rapidly exploring random tree (RRT). A 4-DOF 2-link arm has been defined, together with a spherical obstacle. The cylinders depicting the arm's links should not collide with the obstacle.

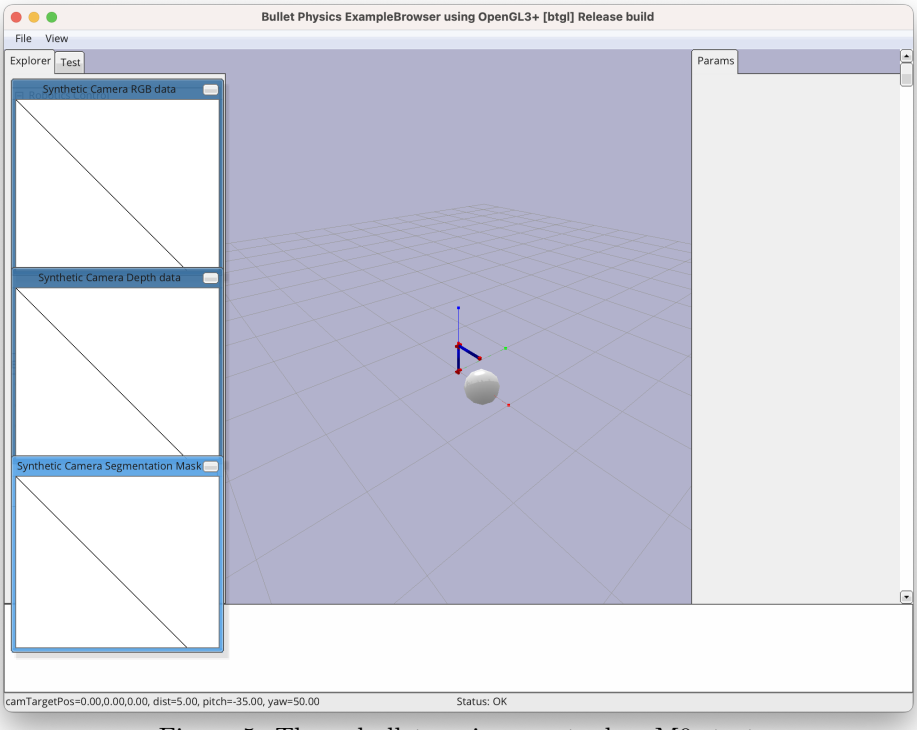


Figure 5: The pybullet environment when M0 starts

M0. **0 points.** Explore M0 and see how the robot works. The code will open pybullet and move to the joint angles given by M0. As you may notice, the robot ignores all collision.

M1. **1 points.** Given the provided joint limits, generate uniformly distributed samples from configuration space. The configurations should be within the joint limits, but they can be in collision. You would not be able to verify the correctness of this function until later questions.

M2. **2 points.** Implement the PRM algorithm to construct a roadmap for this environment. Your graph should contain `num_samples` collision-free configurations within joint limits. For each vertex, consider the nearest `num_neighbors` neighbors, and add an edge if the segment between them is collision-free. We will use `networkx.Graph` to store the roadmap. Note: we do collision checking by setting the robot joint in the simulator. This means you will see the robot "jump" between multiple positions as it does collision checking.

M3. **2 points.** Use the roadmap to find a collision-free path from the start configuration to the goal configuration. You will need to appropriate points to get "on" and "off" the roadmap from the start and goal respectively.

M4. **2 points.** Implement the RRT algorithm to find a collision-free path from the start configuration to the goal configuration. Choose appropriate hyperparameter values for the step size and frequency of sampling q_{goal}. Remember to traverse the constructed tree to recover the actual path.

M5. **2 points.** The path found by RRT likely has many unnecessary waypoints and motion segments in it. Smooth the path returned by the RRT by removing unnecessary waypoints. One way to accomplish this is to check non-consecutive waypoints in the path to see if they can be connected by a collision-free edge.

M6. **0 points.** If M4 and M5 have been implemented correctly, this question should require no further implementation. Watch your algorithm work on a more challenging motion planning problem. Can you make it even harder?

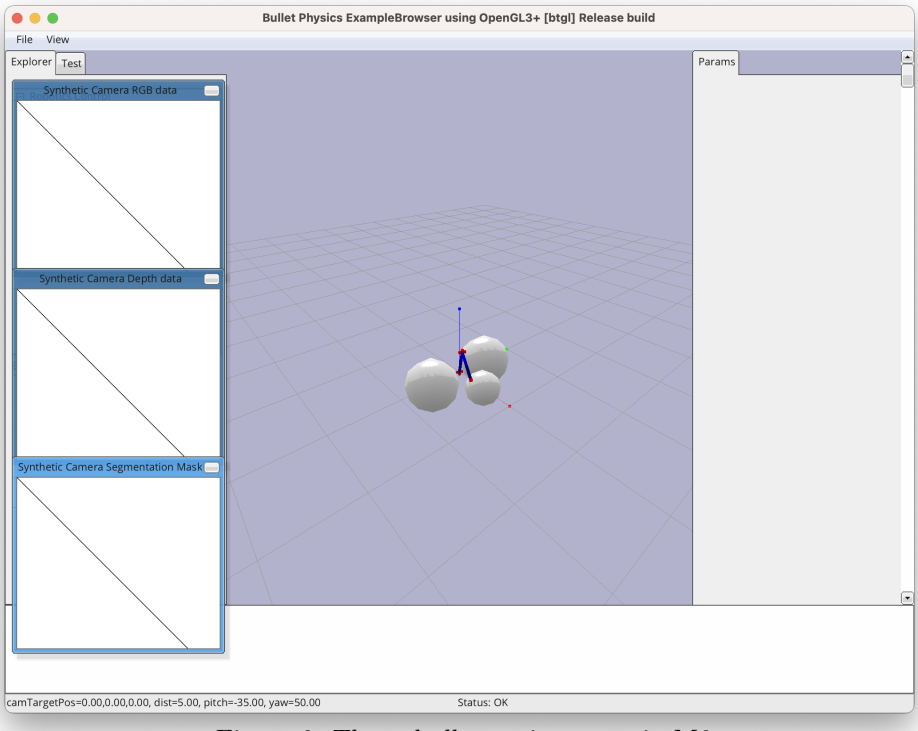


Figure 6: The pybullet environment in M6