Assignment 2

(100 pts is the perfect score - 20% of the course credit)

Deadline: Friday, November 10, 12:59 PM (around noon)

Submission Process: You are asked to upload a zip file on Canvas containing all necessary files of your submission. Only 1 student per team needs to upload a submission. *Aim for 2 students per team.*

You should create a top-level folder named with your NETID, (e.g., xyz007-abc001 for teams of 2 students or just xyz007 for students working independently). Include at least the following files by following this folder structure:

- xyz007-abc001/arm_1.py
- xyz007-abc001/arm_2.py
- xyz007-abc001/arm_3.py
- xyz007-abc001/arm_4.py
- xyz007-abc001/arm_5.py
- xyz007-abc001/arm_6.py (optional for PRM*)
- xyz007-abc001/rigid_body_1.py
- xyz007-abc001/rigid_body_2.py
- xyz007-abc001/rigid_body_3.py
- xyz007-abc001/rigid_body_4.py
- xyz007-abc001/rigid_body_5.py
- xyz007-abc001/car_1.py
- xyz007-abc001/car_2.py
- xyz007-abc001/report.pdf
- xyz007-abc001/videos/*.{mp4,gif}
- xyz007-abc001/others/* (if any, like .tex, .npy)

Put all videos as requested under xyz007-abc001/videos/ and put all other files under xyz007-abc001/others/ (for example, all .npy files should go into others). Please zip this xyz007-abc001 folder and submit the xyz007-abc001.zip on Canvas. You can also create a utils.py file that contains helper functions under the xyz007-abc001 folder. On the first page of the report's PDF indicate the name and Net ID of the students in the team (up to 2 students - aim for 2 students).
Failure to follow the rules will result in a lower grade (-10 pts) on the assignment. Late submissions will not be graded.
Extra Credit for LaTeX: You will receive 6% extra credit points if you submit your answers as a typeset PDF, i.e., one generated using LaTeX. For typeset reports in LaTeX, please also submit your .tex source file together with the report, e.g., by adding a report.tex file under the xyz007-abc001 folder. There will be a 3% bonus for electronically prepared answers (e.g., using MS Word, etc.) that are not typeset. Remember that these bonuses are computed as a percentage of your original

grade, i.e., if you were to receive 50 points and you have typeset your report using LaTeX, then you get a 3-point bonus. If you want to submit a handwritten report, scan it or take photos of it and submit the corresponding PDF as part of the zip file on Canvas. You will not receive any bonus in this case. For your reports, do not submit Word documents, raw ASCII text files, or hardcopies etc. If you choose to submit scanned handwritten answers and we cannot read them, you will not be awarded any points for the unreadable part of the solution.

Requirements: Standard Python libraries are allowed. Additionally, Numpy, Scipy and Matplotlib can be used. Please use the following script to build your Python environment under conda.

Setup python environment

```
#!/bin/bash
conda create -n cs460 python=3.10.12 numpy=1.25.2 matplotlib=3.7.2 scipy=1.11.2 -y
conda activate cs460
# run your code from here
```

This Python environment will be used for all assignments. We expect to run your code on ilab. If we cannot run your code in the Python environment as described above, then 0 credits will be awarded for the programming tasks. You can also not use conda, as long as your Python and library versions match.

1 Motion Planning for a 2-link Planar arm (40 pts + 20 Extra Credit)

The planar arm is defined as follows: The length between joint 1 and joint 2 should be 0.4m, and the length between joint 1 and joint 2 should be 0.25m. The width of the rectangles (body) should be 0.1m. The length of the body should be 0.3 and 0.15 respectively. The joints correspond to circles of radius 0.05m. The joint 1 is fixed at (1,1). There are no limits for both joints and they can rotate indefinitely in either direction. When the arm points to the right, it is at configuration $[\theta_1, \theta_2] = [0, 0]$. Counterclockwise rotation is in the positive direction.

Please read through the whole assignment before you start coding. For RRT and PRM, you need to make them work both for the planar arm and the 2D rigid body. You could implement one version of each planning algorithm for both systems, or you can implement separate instances per robot system, it is your choice. You can also re-use code from assignment 1, such as collision checking.

Please use the provided map to generate plots for your assignment. Your python scripts should be able to operate with different maps. Make sure that you test the correctness of your solutions for different maps, e.g., you can use the maps that you generated during the first assignment.

1.1 Sampling random collision-free configurations (5 pts)

Create an executable Python file called arm_1.py (you can have functions written in other helper files for reuse, and import these functions to this Python file to execute). We will run your code as:

```
python arm_1.py --map "arm_polygons.npy"
```

Your program should generate a plot with a random collision-free configuration of the robot arm in the workspace.

You are provided a map called "arm_polygons.npy" (a set of 2D convex polygons, vertices are stored in counter-clockwise order) in order to experiment. For your report randomly generate 5 collision-free configurations in this environment and include the corresponding visualizations in your report. Do so as well for one more environment from those that you generated as part of Assignment 1. Briefly explain your implementation.

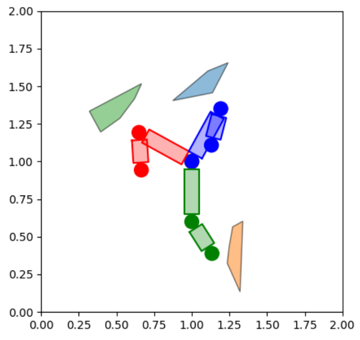

Figure 1: Example for 1.1, three random collision-free configurations.

1.2 Nearest neighbors with linear search approach (5 pts)

Create an executable Python file called arm_2.py. We will run your code as

```
python arm_2.py --target 0 0 -k 3 --configs "arm_configs.npy"
```

The two numbers after the "–target" parameter corresponding to the robot's configuration, i.e., the first and the second joint angle respectively. In the above example, both values are 0. The number after the "–k" parameter, i.e., 3 in the above example, corresponds to the number of nearest neighbors to be reported. The filename after the "–configs" parameter, i.e., "arm_configs.npy" in the above example, corresponds to a list of random configurations. Treat it as a 2D matrix, where the columns report the joint angles and the rows are different configurations.

Given this input, your program should generate a plot with the three nearest robot configurations to the target given the input list. In order to do so, it is ok to perform a naïve linear search approach, i.e., compute the distances between the target and all the configurations and remember the three nearest neighbors. Describe briefly your implementation in the report.

Make sure that you reason correctly regarding the topology of the robot's configuration space and define accordingly the distance metric between pairs of configurations. Discuss your implementation of distance between two configurations for this robot in your report.

For your visualization, plot the robot both at the target configuration and at its nearest neighbors. Visualize the arm at the target configuration using black color, then, for the 3 nearest neighbors configurations use red, green and blue in that order. If k is greater than 3, then use yellow color for the remaining neighbors. Include 5 example outputs in report.pdf (the environment does not matter here), including for the above example query (target=$[0, 0]$, $k = 3$ and for the list of your configurations in "arm_configs.npy"). For the different examples keep the list of configurations the same but change the target and the number of nearest neighbors.

Extra Credit: KD-Tree implementation and Comparison of Nearest Neighbor Methods (10 pts)

Implement a KD-Tree instead of linear search for extra points. Note that there is a KD-Tree function on "scipy" but it assumes an Euclidean topology. Your KD-tree implementation must handle the non-Euclidean topology of the 2-link planar arm. Describe your implementation in your report. Compare the runtime of the KD-tree and linear search implementation as the number of random configurations in the list increases (i.e., for 100s to 1000s of configurations) and as the number of nearest neighbors increases (i.e., from 3 to 10s of configurations). Report your statistics in your report and visualizations corresponding to the tests you executed. Make sure that the KD-tree implementation and the linear search are returning the same output.

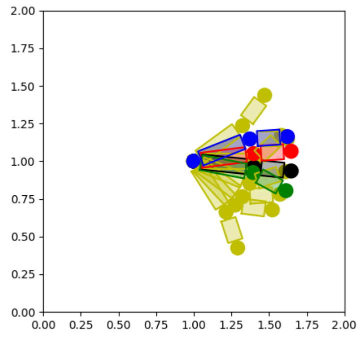

Figure 2: Example for 1.2, three nearest configurations from given configurations to the target configuration. The black is the target configuration, the first nearest is in red, the second is green, and the third is blue. Other possible configurations are in yellow.

1.3 Interpolation along the straight line in the C-space (5 pts)

Create an executable Python file called arm_3.py. We will run your code as:

```
python arm_3.py --start -0.5 -0.5 --goal 1.5 1.5
```

The input corresponds to a start joint configuration, i.e., $[0, 0]$ above, and goal configurations, i.e., $[1.5, 1.5]$ above. Each configuration indicates the value of the first and the second joint angle.

Your program should generate a plot that animates your robot arm moving from the start to the goal configuration step by step (depending on your chosen resolution). For this purpose, you should interpolate the two configurations along the straight line path between them in the configuration space. Make sure to again take care of the non-Euclidean topology. Pick the resolution of interpolation so that the animation of the planar arm moving from the start to the goal configuration looks smooth. You should provide at least 3 animations (gif or mp4 files) with your choice of resolution for different start and goal configurations (of your choice). See "arm_1.3_example.mp4" for an example. Save your videos as "arm_1.3-1.mp4", "arm_1.3-2.mp4" and "arm_1.3-3.mp4". Briefly describe your implementation in your report and the corresponding animations that resulted from it.

1.4 Implement RRT (10 pts)

Create an executable Python file called arm_4.py. We will run your code as

```
python arm_4.py --start 0 0 --goal -2.9 0 --map "arm_polygons.npy"
```

The input corresponds to a file that contains a map, i.e., "arm_polygons.npy" in the above example, as well as a start arm configuration, i.e., $[0, 0]$ above, and a goal arm configuration, i.e., $[1.5, 1.5]$. Each configuration indicates the value of the first and the second joint angle.

You are asked to implement the RRT algorithm in order to find a path from the given start configuration to the given goal configuration that avoids collisions with obstacles indicated in the map file. Your search should terminate after adding 1000 nodes to the graph (It's expected that the RRT cannot find the path given limited iterations in some cases). You can occasionally sample the goal configuration in order to assist RRT in finding a solution (e.g., 5% of the time).

Your program should generate two visualizations upon completion:

- An animation that shows the tree data structure generated by RRT growing over the search process in the arm's configuration space, i.e., each node of RRT corresponds to a point in the configuration space and edges correspond to straight line segments. Make sure that your animation properly treats the topology of the configuration space.
- An animation that shows the arm moving along the solution path inside the original workspace from the start configuration to the goal configuration. This path should not result in collisions with the environment.

In your report, explain your RRT implementation. Especially describe how you are deciding to add an edge in the tree. Also submit example tree growth animations and solution animations for at least 2 additional environments and 2 different starts and goals per environment. Make sure to include animations for the above example query in the "arm_polygons.npy" environment and save them as "arm_1.4-tree1.mp4" and "arm_1.4-solution1.mp4" respectively.

We provide “arm_1.4_example.mp4” as an example for the tree growth visualization. This example video does not consider the toric topology of the configuration space. In your implementation, however, special attention must be paid to points that are proximate to the upper and lower limits on the x or the y-axis. Specifically, when encountering such points, you should construct two line segments. We provide “arm_1.4_example2.mp4” as an example to control the robot arm following the path from the tree of the previous example.

1.5 Implement PRM (15 pts)

Create an executable Python file called arm_5.py. We will run your code as

```
python arm_5.py --start 0 0 --goal -2.9 0 --map "arm_polygons.npy"
```

The input corresponds to a file that contains a map, i.e., "arm_polygons.npy" in the above example, as well as a start arm configuration, i.e., $[0, 0]$ above, and a goal arm configuration, i.e., $[-2.9, 0]$. Each configuration indicates the value of the first and the second joint angle.

You are asked to implement the PRM algorithm in order to find a path from the given start configuration to the given goal configuration that avoids collisions with obstacles indicated in the map file. Your search should terminate after adding 1000 nodes to the graph. Use a constant number of neighbors, $k = 3$, for your implementation at this step. As part of your implementation for the PRM, you will also need to implement a discrete search algorithm, i.e., Dijkstra's or A*.

Your program should generate two visualizations upon completion:

- An animation that shows the roadmap data structure generated by PRM growing over the search process in the arm's configuration space, i.e., each node of PRM corresponds to a point in the configuration space and edges correspond to straight line segments. Make sure that your animation properly treats the topology of the configuration space.
- An animation that shows the arm moving along the solution path inside the original workspace from the start configuration to the goal configuration. This path should not be resulting in collisions with the environment.

In your report, explain your PRM implementation. Also submit example roadmap growth animations and solution animations for at least 2 additional environments and 2 different starts and goals per environment. Make sure to include animations for the above example query in the "arm_polygons.npy" environment and save them as "arm_1.5-roadmap1.mp4" and "arm_1.5-solution1.mp4" respectively.

Extra Credit: PRM* and Comparison of Planners (10 pts)

Change your PRM code so that it implements the PRM* approach and describe in your report how you changed the implementation and why. Provide visualizations of the roadmap construction and the solutions paths for PRM* over the same environment and start/goal query points.

Then, perform an experimental evaluation of the three planners (RRT, PRM and PRM*) in terms of their success rate, path quality and computation time given a fixed number of iterations. In particular, for the same environment and pair of starts and goals execute the methods multiple times, e.g., 10 times, and count the number of times that a solution was found or not after 500 iterations, the average solution quality upon completion of the algorithm and the average computation time required by each algorithm. Do that for 5 environments and 5 pairs of starts/goals. Include your statistics in your report.

2 Motion planning for a rigid body in 2D (25 pts)

This is a repetition of the previous section but instead of the 2-link planar arm, your robot is a rigid body that can rotate and translate in 2D. Create a 2D rigid body (car) with dimensions: $0.2 \times 0.1 m^2$. The reference point for the rectangle is its geometric center. The workspace limits are $[0, 2]$ for both x and y. When the long side of the rigid body points to the right, the robot is $\theta = 0$ in orientation. Counterclockwise rotation is in the positive direction.

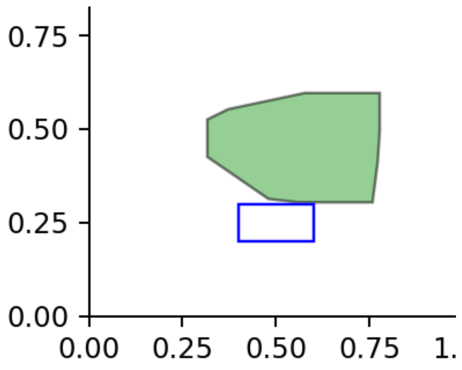

Figure 3: Example for 2D rigid body, the rectangle is at [0.5, 0.25, 0].

Repeat problems 1.1, 1.2, 1.3, 1.4, 1.5 but for the 2D rigid body. They will correspond to sections 2.1, 2.2, 2.3, 2.4, and 2.5 in your report. Ignore the extra credit questions for the rigid body in 2D. Save the required videos in your submission as well.

2.1 Sampling random collision-free configurations (5 pts)

Create an executable Python file called rigid_body_1.py (you can have functions written in other helper files for reuse, and import these functions to this Python file to execute). We will run your code as

```
python rigid_body_1.py --map "rigid_polygons.npy"
```

Your program should generate a plot with a random collision-free configuration of the robot arm in the workspace. Add to your report similar visualizations as the ones asked in Section 1.1.

2.2 Nearest neighbors with linear search approach (5 pts)

Create an executable Python file called rigid_body_2.py. We will run your code as

```
python rigid_body_2.py --target 1 1 0 --k 3 --configs "rigid_configs.npy"
```

Your program should generate a plot with the three nearest robot configurations to the target given the input list.

You need again to define a proper distance metric by integrating a Euclidean distance for translation (d_t) and a rotational distance (d_r) that properly reasons about the robot's orientation. To unify these two terms, use a weighted average as shown below, where $\alpha = 0.7$:

$$D = \alpha d_t + (1 - \alpha) d_r.$$

You are given a target as $[1, 1, 0]$ (x, y, theta), and a list of random configurations in "rigid_configs.npy" (2D matrix, the columns are x, y, θ).

2.3 Interpolation along the straight line in the C-space (5 pts)

Create an executable Python file called rigid_body_3.py. We will run your code as

```
python rigid_body_3.py --start 0.5 0.5 0 --goal 1.2 1.0 0.5
```

Your program should pop up a plot with animation that your robot moves from the given start configuration to the goal configuration step by step based on your chosen resolution.

2.4 Implement RRT (5 pts)

Create an executable Python file called rigid_body_4.py. We will run your code as

```
python rigid_body_4.py --start 0.5 0.25 0 --goal 1.75 1.5 0.5
                       --map "rigid_polygons.npy"
```

Your program should generate an animation that shows the planar body moving along the solution path inside the original workspace from the start configuration to the goal configuration in a collision free manner.

2.5 Implement PRM (5 pts)

Create an executable Python file called rigid_body_5.py. We will run your code as

```
python rigid_body_5.py --start 0.5 0.25 0 --goal 1.75 1.5 0.5
                       --map "rigid_polygons.npy"
```

Your program should generate an animation that shows the planar body moving along the solution path inside the original workspace from the start configuration to the goal configuration in a collision free manner.

3 Motion Planning for a first-order car (35 pts)

Create a planar car with dimensions: $0.2 \times 0.1m^2$. The workspace limits are $[0, 2]$ for both x and y. When the long side of the car points to the right, then the car is at $\theta = 0$ orientation. Counterclockwise rotation is in the positive direction.

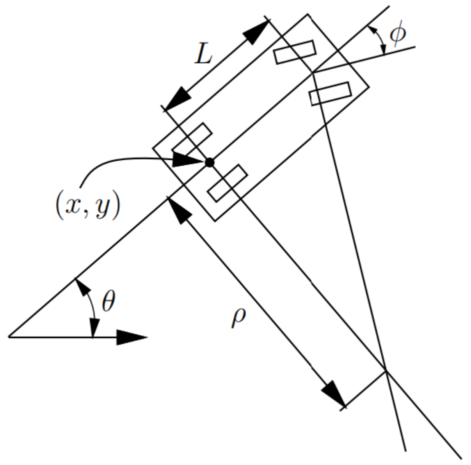

Figure 4: Example for car model, from `http://lavalle.pl/planning/ch14.pdf`.

3.1 Implement the dynamics model of the car, and use a keyboard to control it (5 pts)

Generate an animation that shows that you can move the car around following the model provided below. Check "car_example.mp4" for an example (you do not need to draw wheels, though it may be helpful for your debugging - this is just to emphasize that there is a steering wheel and the car does not rotate in place.). Save your video as "car_1.mp4" in your submission folder.

- $q(t)$ is the robot configuration at time t;
- u is the control input (where velocity and steering angle can change instantly);
- v is the car's velocity along its current heading direction ($-0.5 \leq v \leq 0.5$);
- ϕ is the steering angle ($-\pi/4 \leq \phi \leq \pi/4$);
- $L = 0.2$ is the wheelbase, i.e., the distance between the front and rear axles;
- dt is the time step you will pick in order to update the robot's configuration.

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}, \quad u = \begin{bmatrix} v \\ \phi \end{bmatrix} \tag{1}$$

$$\dot{q}(t) = \begin{bmatrix} v\cos(\theta(t)) \\ v\sin(\theta(t)) \\ \frac{v}{L}\tan(\phi) \end{bmatrix} \tag{2}$$

$$q(t + dt) = q(t) + \dot{q}(t)dt \tag{3}$$

3.2 Integration of dynamics for constant control (5 pts)

Create an executable Python file called car_1.py. We will run your code as

```
python car_1.py --control 0.5 0.2 --start 0.5 0.5 0
```

Your program should generate an animation of your robot moving in a free workspace given the input constant control, i.e., $v = 0.5$ and $\phi = 0.2$ in the above example, for 5 seconds.

In particular, given the above specified model, perform integration of the dynamics. Your software should be able to receive the start state of the robot and a constant control. It should then generate the sequence of states that the robot goes through over a duration of 5 seconds given the control remains constant. Provide animations of the corresponding integration process in the free environment for your choice of start states and constant controls. Pick a constant u, and show the corresponding circular arcs that your car is following. Save your video as "car_2.mp4".

3.3 Extend RRT to planning with dynamics (25 pts)

Create an executable Python file called car_2.py. We will run your code as

```
python car_2.py --start 0.5 0.25 0 --goal 1.75 1.5 0.0
                       --map "rigid_polygons.npy"
```

Your program should generate an animation of your robot moving from the given start to the given goal configuration without colliding with the obstacles specified in the given map file. As this program may never hit the goal exactly, you can stop the program once the car reaches the goal region **(the threshold for translation distance to goal should be less than** 0.1 **meters, and rotation should be less than** 0.5 **radians)**.

In order to find such a solution, extend your implementation of RRT so that it handles the given model of a first-order, non-holonomic car. In particular:

- As before, randomly sample configurations q_{rand} and identify the closest node q_{close} on the tree to the random sample q_{rand}. You can again occasionally sample the goal configuration in order to assist RRT in finding a solution (e.g., 5% of the time).
- Instead of trying to connect with an edge the closest node q_{close} to the random sample q_{rand}, in this version of RRT you sample a blossom b of random controls $\{u^1, \ldots, u^b\}$.
- For each of the b sampled controls u^i, you integrate forward in time the dynamics from the closest node q_{close} for the same duration ΔT. This results in b possible new states $\{q^1_{new}, \ldots, q^b_{new}\}$ that are reachable from q_{close}. Accept a new state only if the integration of the dynamics for time ΔT does not result in a collision.
- Check the distance between all the new states $\{q^1_{new}, \ldots, q^b_{new}\}$ to the random sample q_{rand} and identify which state $q^{closest}_{new}$ among them is the closest to q_{rand} given your distance metric in this configuration space.
- Then, add the edge from q_{close} to $q^{closest}_{new}$ and store the corresponding control on the edge.

If the above tree results in the generation of nodes in close vicinity to the given goal configuration, then we can trace the ancestors of the node close to the goal back to the root to identify the sequence of controls from the start configuration that can bring the robot close to the goal.
There are design choices you may need to make for balancing speed and path quality. For instance, you can experiment with the size of the blossom b. Make sure to check a value of $b = 1$ as well as higher numbers, e.g., $b = 3$ or $b = 5$ and study their effect in success rate, path quality and computational cost. Experiment with different design choices and submit a script that has performed the best for you. Describe in your report your experimentation process and the corresponding statistics. You may also need to execute the algorithm for a large number of iterations.

You will receive full credit if your solution can reliably find solutions from start to a goal configurations in a collision free manner while respecting the car's constraints. Submit a solution video for the "rigid_polygons.npy" environment titled as "car_3.mp4". Demonstrate in your submissions that you have tested your algorithm in additional environments.

Collusion, Plagiarism, etc.: Each team must prepare their solutions independently from others, i.e., without using common code, notes or worksheets with other students. You can discuss material about the class that relates to the assignment but you should not be provided the answers by other students and you must code your own solutions as well as prepare your own reports separately.

Furthermore, you must indicate at the end of your report any external sources you have used in the preparation of your solution. This includes both discussions with other students and online sources. Unless explicitly allowed by the assignment, do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or the university.

Online sources include the use of ChatGPT or other Large Language Models and Generative AI tools. While the use of such tools is allowed as supportive instruments for programming, the teams need to report the level of their use and the part of the code that was generated by them. Students have to be careful that blindly copying significant pieces of code from such tools - which falls under the definition of plagiarism - may result on submissions that do not accurately answer the assignment.

Failure to follow these rules may result in failure in the course.