CS593-ROB Assignment 1
Sampling-based Motion Planning

January 20, 2023

Part1: RRT and BiRRT in 3D environment

For the first part, you will implement Rapidly-exploring Random Tree(RRT) and Bidirectional Rapidly-exploring Random Tree(BiRRT) in 3D environments. The algorithms will be applied on a simplified 3-DOF UR5 robot arm to let it reach the target position.

Installation and the environment

Create a conda environment and simply run `pip install pybullet` to install the simulator. Verify your installation by running `python3 assignment1_part1_3d.py`.

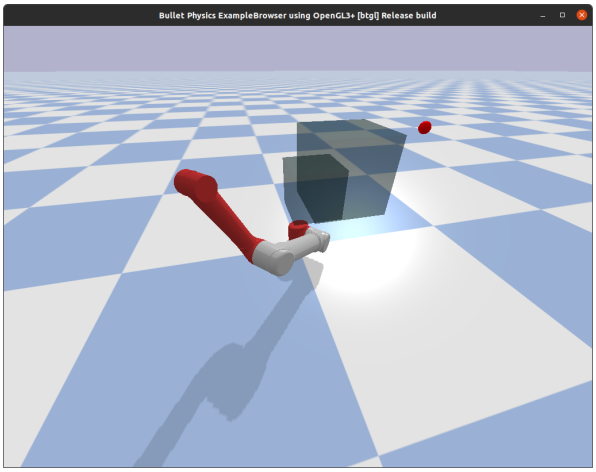

Figure 1: robot arm start configuration

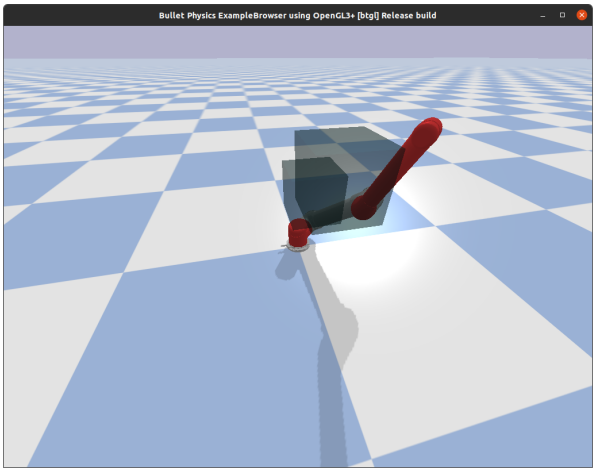

Figure 2: robot arm goal configuration

The robot used in this part is a simplified 3-DOF(first 4 links) UR5 robot arm manipulator. More details can be found in https://www.universal-robots.com/products/ur5-robot/ and the provided .urdf file in ./assets/ur5/ur5.urdf. The range of the three joints of the arm are bounded in [-2pi, 2pi], [-2pi, 2pi] and [-pi, pi], respectively.

Inside the simulation environments, aside from the robot, there are a ground plane and two obstacle blocks. The arm cannot collide with any of them during movement. Your task is to

implement the two algorithms stated above to let the end link of the arm reach the target position in a collision-free manner, see Figures 1 and 2.

Helper functions

- set_joint_positions(body, joints, values): use this function to set the arm to a specific pose defined by the values input
- collision_fn(conf): use this function to help check if there is collision between the arm defined by conf(configuration) and the environment(self collision + ground + obstacle blocks). If you are interested, give it a read to better understand how collision is detected.
- draw_sphere_marker(position, radius, color): use this function to draw a sphere in the environment to better track the robot's movement
- remove_marker(marker_id): use this to remove the marker from the environment

Question 1: RRT [2 pts]

Use the pseudo-code introduced in class to implement the RRT algorithm. The planning should be done in the configuration space while the collision checking should be done in workspace. We provide the start and goal configuration in the code. Note that we will use different start/goal configurations to grade your submission. You can use while True statement in the outer loop to make sure if will generate a valid path in one single run.
You need to implement the following part in the provided scaffolding code:

- RRT_Node class: use this class to define each RRT node in the RRT Tree structure. Implement the three member functions in this class, namely __init__(self, conf), set_parent(self, parent) and add_child(self, child).
- sample_conf(): this function should return a robot conf uniformly sampled from the robot's joint limits along with a boolean flag indicating whether the sampled conf is goal conf
- find_nearest(rand_node, node_list): return the nearest node of rand_node from the node_list, node_list should contain all nodes in the current RRT Tree
- steer_to(rand_node, nearest_node): check if there is a collision-free path between the two input nodes. Collision checking needs to be performed on each interpolated middle node between the two inputs. Use a step size of 0.05 as that is what we used in the grading system.
- RRT(): the main function that performs the RRT algorithm. This function should return an ordered list of robot confs that moves the robot arm from the start position to the goal position.

Run and test your implementation by `python3 assignment1_part1_3d.py`. You may need to write out all the intermediate nodes in your path(those generated by linear interpolation to check collision) to visually check the performance. However, the final path from the algorithm should not include them. Same thing apply for the BiRRT and BiRRT smoothing algorithms.

Question 2: BiRRT [2 pts]

Use the pseudo-code introduced in class to implement the BiRRT algorithm. Aside from the functions listed above, two additional functions are needed in this part

- steer_to_until(rand_node, nearest_node): compared with the steer_to function implemented in Question 1, this function returns the furthest collision-free intermediate node from nearest_node heading to rand_node.
- BiRRT(): the main function that performs the BiRRT algorithm. This function should return an ordered list of robot confs that moves the robot arm from the start position to the goal position.

Run and test your implementation by `python3 assignment1_part1_3d.py --birrt`.

Question 3: Path Smoothing [0.5 pt]

In this part, you should implement the path smoothing algorithm on the path found by BiRRT in function BiRRT_smoothing(). The pseudo-code is shown in the following snippet. Use $N = 100$ for this part. You should expect to stably find a final path only containing 3 nodes(including start and goal, i.e., only one intermediate node) in the given environment.

Algorithm 1 Path Smoothing

1: **for** $i = 1$ ***to*** $i = N$ **do**
2: randomly select non-adjacent $index_1$ and $index_2$ from the original path
3: check if there is a path between these two points by linear interpolation
4: if so, remove all intermediate points between them
5: **end for**

Run and test your implementation by `python3 assignment1_part1_3d.py --birrt --smoothing`.

Report [0.5 pt]

Summarize your findings regarding the pros and cons for each of these three motion planning algorithms.

Part2: RRT* in 2D enviroment

In the second part, you will implement the Rapidly-exploring Random Trees star (RRT*) method for three different robot systems, i.e., 2D point-mass, circular rigid body, and a rectangular rigid-body in a 2D environment. Algorithm 1 outlines a standard RRT* algorithm. However, if we remove the lines highlighted in red, it becomes a traditional RRT algorithm. The supplementary python code file provided with this assignment contains an implementation of the standard RRT algorithm. Please extend that python code to address the following questions.

Algorithm 2 RRTstar algorithm

1: $\mathcal{T} \leftarrow$ InitializeTree();
2: $\mathcal{T} \leftarrow$ InsertNode($z_{init}, \mathcal{T}$);
3: **for** $i = 1$ ***to*** $i = N$ **do**
4: $x_{rand} \leftarrow$ Sample(i)
5: $x_{nearest} \leftarrow$ Nearest($\mathcal{T}, x_{rand}$)
6: $r_{valid}, r_{cost} \leftarrow$ Steer($x_{nearest}, x_{rand}$);
7: **if** r_{valid} **then**
8: $X_{near} \leftarrow$ Near($\mathcal{T}, x_{rand}$);
9: $x_{min} \leftarrow$ ChooseParent($X_{near}, x_{nearest}, x_{rand}$);
10: $\mathcal{T} \leftarrow$ InsertNode($x_{min}, x_{rand}, \mathcal{T}$);
11: $\mathcal{T} \leftarrow$ Rewire($X_{near}, x_{rand}, x_{min}, \mathcal{T}$);
12: **end if**
13: **end for**

Question 4 [3 pts]

In this question, implement and add the following functions to turn the provided RRT code into RRTstar:

- Near (Algorithm 1, Line 8): Given a sample $x \in X$, the tree $T = (V, E)$ and the ball region $\mathfrak{B}_{x,r}$ of radius r centered at x, the set of near vertices is defined as: $\text{Near}(x, T, r) := \{v \in V : v \in \mathfrak{B}_{x,r}\} \mapsto X_{\text{near}} \subseteq V$. More specifically, $X_{\text{near}} = \{v \in V : d(x, v) \leq \gamma(logi/i)^{1/n}\}$ where i is the number of vertices, n represents the dimensions and γ is a constant.

- ChooseParent (Algorithm 1, Line 9): This procedure is used to search the list X_{near} for a state, x_{min} which provides the shortest, collision-free path from the initial state x_{init} to the random sample x_{rand}.

- Rewire (Algorithm 1, Line 11): Here, the algorithm examines each vertex $x' \in X_{\text{near}}$ lying inside the ball region centered at x_{rand}. If the cost of the path connecting x_{init} and x' through x_{rand} is less than the existing cost of reaching x' and if this path lies in obstacle free space x_{free}, then x_{rand} is made the parent of x'. If these conditions do not

hold true, no change is made to the tree and the algorithm moves on to check the next vertex. This process is iteratively performed for every vertex x' present in the X_{near}. *Note:* If a vertex x' is rewired, its cost must be updated to reflect the change; and if x' has descendants, they should also be updated. Your rewire() implementation should be sure to do this.

Once implemented, include in your report:

- The images of tree and final path found for both RRT and RRT*.
- The following table comparing RRT and RRT* algorithms. You can run the planning algorithm N times(N $\geq$ 30 to avoid bias) and get the average.

2D point-mass	**RRT**	**RRT***
path cost	-	-
computation time	-	-

Question 5 - Circular Rigid Body [1 pt]

In this question, extend both RRT and RRTstar algorithms to plan for a circular rigid body of radius 1 unit (Fig 1). **Hint** Think of adapting the collision checker and final trajectory

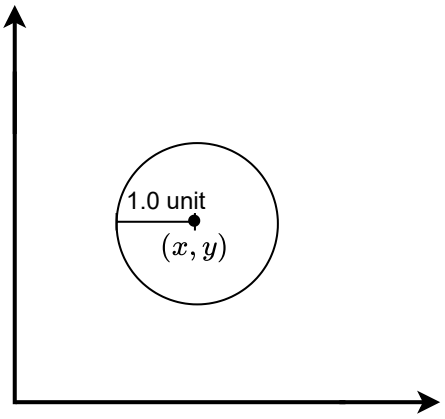

Figure 3: Circular Rigid Body of radius 1 unit.

visualizer. In the report include the following:

- The images of tree and final path found for both RRT and RRT*.
- The following table comparing RRT and RRT* algorithms.

Circular Rigid body	**RRT**	**RRT***
path cost	-	-
computation time	-	-

Question 6 - Rectangular Rigid Body [1 pt]

In this question, extend both RRT and RRTstar algorithms to plan for a rectangular rigid body of dimensions $L = 1.5, W = 3$ units. The rigid body has three dimensional configuration space, i.e., (x, y, θ), as shown in Fig 2.

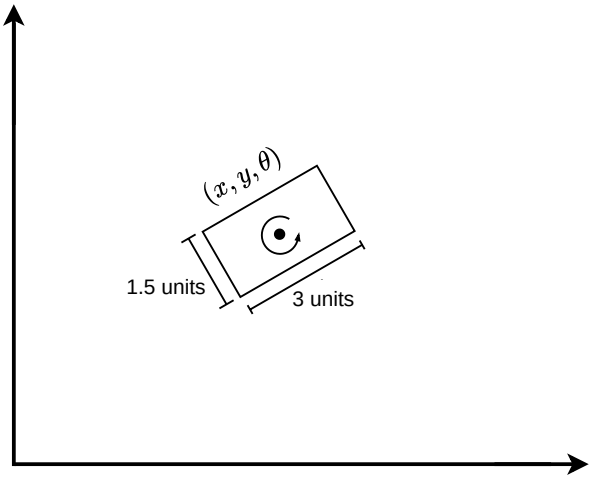

Figure 4: Rectangular rigid body.

Hint You will need to modify the random sampler to sample positional (x,y) and rotational (theta) coordinates. In addition, you will also need to adapt the collision checker and final trajectory visualizer. For collision-checking, consider robot and obstacles as boxes and check collision by finding the overlap between boxes. In the report include the following:

- The images of tree and final path found for both RRT and RRT*.
- The following table comparing RRT and RRT* algorithms.

Rectangular Rigid body	**RRT**	**RRT***
path cost	-	-
computation time	-	-

Using the provided code

The code provided in assign1.py contains a CLI and animations to visualize how the program is running. Run `assignment1_part2_2d.py` to watch a point-mass robot navigate its environment using RRT. The CLI contains several useful flags, including options to configure the algorithm used (RRT or RRT*), and the geometry of the robot (point mass, circle, or rectangle). Only the RRT/point mass combination is implemented; in the assignment, you'll add code for the other combinations.
Be sure to read over all the code, and understand what it is doing. For question 1, you will only have to implement 3 functions. For questions 5 and 6, you will have to make modifications in various places in the code. Feel free to make any changes you feel are

necessary to complete the assignment - we'll compare your submission against the original when grading.

Submission instructions

Your submission should consist of two items:

- Modified `assignment1_part1_3d.py` and `assignment1_part2_2d.py`.
- A PDF report. As discussed above, this report should contain the write-up about part1 3D and the results for part2 (path costs, computation times, and visualizations for both RRT and RRT*), using each of the robot geometries. (point mass, circle, and rectangular rigid body).

  Note: if you have any special explanations of how your code works, or instructions for running it, feel free to place them either in the report or in a separate README file.

Bundle all files into a zip or tarfile, and submit it via Brightspace. Name your submission "<first name_last name>.zip" or "<first name_last name>.tar.gz".