RRT Motion Planning for Franka Emika 7 DoF Robot Arm

*Code: `GitHub`

Huaijing Hong Nikola Raicevic Subhadeep Chatterjee

***Abstract*—This project aims to develop a motion planner for a 7-DoF manipulator so its end-effector can move to a specific spot or follow a specific trajectory in 3D Cartesian space. We devised three scenarios and implemented two algorithms based on Rapidly Exploring Random Tree (RRT). In the end, we compared the planning time, obstacles hit, and goals reached between the three methods.**

***Index Terms*—Motion Planning, RRT-Based Algorithm, Robot Arm, Real-Time Planning**

I. Introduction

The motion planning problem has always been one of the most fundamental ones in the field of robotics, especially robot arms. It aims to find the trajectory that the robot arm end-effector can move along with, avoid obstacles and joint limits, and eventually reach the goal. However, it is extremely hard to achieve when a robot arm is placed in a complex environment, sometimes even with moving obstacles. Generally, the traditional motion planning algorithm can be divided into 3 categories: 1) Potential function-based algorithm. 2) Search-based algorithm. 3) Sampling-based algorithm.

A. Potential-function-Based Algorithm

This is the most basic motion planning algorithm, first introduced by Khatib in 1986 [1]. The overarching idea is to create a potential field (like gravity) by assigning a scalar value to each of the spots in the configuration space. For goals there will be attractive force fields with negative values and the obstacles will have repulsive force fields with positive values. The trajectory is found by following the potential field to a minimum. Even though it has the advantage of being able to do real-time planning, it is not that widely used due to it easily getting stuck in local minima and does not avoid obstacles.

B. Search-Based Algorithm

The most traditional and common seen are the searched-base algorithms, such as A* [2], Dijkstra [3], D* [4], etc. They are mostly done by first discretizing the free space and then running the planner which is defined under some rules and finding the trajectory. The essential of this type of planner is the way they define the planning rules. For example, the A* algorithm aims at finding the node with the largest value at every time step which is defined by value = heuristic - node cost. These types of methods are guaranteed to find the optimal path, but when it comes to high-DOF robots it is almost inapplicable due to the exponential growth of computational requirements.

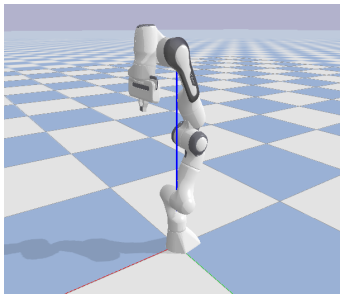


Fig. 1: Panda Robot Arm

C. Sam-Based Algorithm

Sampling-based algorithms, one of the most widely used ones with high efficiency, like Rapidly-exploring Random Trees (RRT) [5] and Probabilistic Roadmaps (PRM) [6]. These methods randomly sample points in the configuration space, making them more efficient for high-dimensional problems. RRT incrementally builds a tree from the start position toward the goal, while PRM builds a graph of random samples and connects them. Our project is based on RRT and the rest of this report will also focus on RRT.

II. Background

A. Problem formulation

Motion planning for robot arms can be wildly used in many scenarios in reality and the application is mostly used under manufacturing or construction. Therefore, the general idea of this project is to simulate the scene where a robot arm is doing welding on a construction site with given spots. For the robot arm, we chose the most wildly use model, Franka Emika Panda arm, a 7-DoF robot arm. Since we did not find the model that has a welding gun as its end-effector, the original version with a gripper is used instead. Since we are mostly focused on the kinematics they should generally be the same.

The simulation used is PyBullet, an open-source physics simulation platform written in C++, but we run the platform and our code in Python. Three different scenarios are also established as follows:

a) Ball-kicking: : There is a soccer ball that is surrounded by many obstacles and we want the arm can kick the soccer ball with its end-effector and avoid all the other obstacles. This environment is the simplest version of motion planning, it is used more like a testing environment.

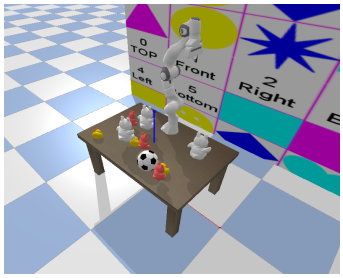


Fig. 2: The Ball-kicking environments

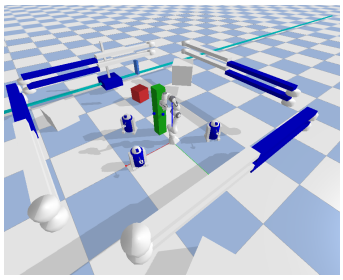
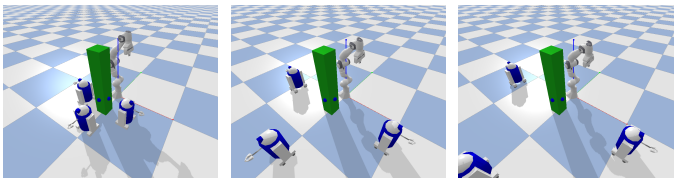


Fig. 3: The Static obstacles environment

b) Static Obstacles: : In this case, we establish a construction scenario with multiple static obstacles around it. Then there's a cuboid set in front of the arm and four goal points around the cuboid. The goal of this scenario is to touch the goal points with the end-effector 1 to 4 sequentially and stay at each point for 5 seconds.

c) Moving Obstacles: : In this case, a construction scenario similar to the second one but also more realistic is established. The robot arm is also facing a cuboid with the same goal, however, this time it is surrounded by multiple moving obstacles.



(a) (b) (c)

Fig. 4: The moving obstacles environment, has objects placed around the arm and moving back and forth:

III. Methods

A. Baseline Methods

In this project, RRT* [7] was picked as our baseline method, an extension of the Rapidly-exploring Random Trees (RRT) algorithm. Before we dive into RRT*, here is a brief introduction to RRT. In summary, the RRT is working through incrementally building a random tree rooted at the initial state and exploring the space by randomly sampling points and connecting them to the nearest point in the tree. It can be broken down following steps:

- Initialization: Start with an empty tree with the initial position as the root node.
- Sampling: Randomly sample a point in the configuration space.
- Find Nearest Node: Find the nearest node in the tree to the sampled point.
- Extend: Move from the nearest node toward the sampled point by a fixed step size, creating a new node.
- Check Collision: Verify if the path from the nearest node to the new node is collision-free.
- Add Node: If collision-free, add the new node to the tree and set its parent as the nearest node.
- Goal Check: If the new node is within a certain distance of the goal, terminate and return the path. If not start from the second bullet point, and run through everything until find the goal.

The pseudo-code of the RRT algorithm is as follows:

Algorithm 1 RRT Algorithm

$V \leftarrow \{x_s\}$; $E \leftarrow \emptyset$
for $i = 1 \ldots n$ **do**
  $x_{\text{rand}} \leftarrow \text{SampleFree}()$
  $x_{\text{nearest}} \leftarrow \text{Nearest}((V, E), x_{\text{rand}})$
  $x_{\text{new}} \leftarrow \text{Steer}_{\epsilon}(x_{\text{nearest}}, x_{\text{rand}})$
  if $\text{CollisionFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$
    $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$
return $G = (V, E)$

However, even though RRT can give us a trajectory in just a short period, it does not guarantee that the optimal trajectory will be found since it does not modify the tree once it has been established. This means that the resulting trajectory might and probably will mostly take some detour for the arm to go to the expected configuration. To solve this problem, RRT* is introduced, an improved version of RRT with an extra rewire step, which is used as the baseline method in this project.

It can be broken down into the following steps:

- Initialization: Start with an empty tree with the initial position as the root node. Also, now every node will have a cost, which here we can simplify as the step you took from start to node.
- Sampling: Randomly sample a point in the configuration space.
- Find Nearest Node: Find the nearest node in the tree to the sampled point.
- Extend: Move from the nearest node toward the sampled point by a fixed step size, creating a new node. Set the cost of the new node into the cost of the nearest node + the cost of the step it took from there to the new node
- Check Collision: Verify if the path from the nearest node to the new node is collision-free.

- Add Node: If collision-free, add the new node to the tree and set its parent node to the nearest node.
- Find neighbor: Given a constant radius R, find all the nodes on the tree within the vicinity R of the new node and make sure there's no collision in between.
- Rewire: For all the nodes inside of the neighbor, check whether its cost can be reduced if it reroutes through the new node. If it does, update its cost and set the parent node of that node to the new node.
- Terminate: Keep the algorithm running for given steps and only terminate after it runs out

The pseudo-code of the RRT* algorithm is as follows:

Algorithm 2 RRT* Algorithm

$V \leftarrow \{x_s\}$; $E \leftarrow \emptyset$
for $i = 1 \dots n$ **do**
  $x_{\text{rand}} \leftarrow \text{SampleFree}()$
  $x_{\text{nearest}} \leftarrow \text{Nearest}((V, E), x_{\text{rand}})$
  $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$
  if $\text{CollisionFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \text{Near}((V, E), x_{\text{new}}, \min\{r^*, \epsilon\})$
    $V \leftarrow V \cup \{x_{\text{new}}\}$
    $c_{\min} \leftarrow \text{Cost}(x_{\text{nearest}}) + \text{Cost}(\text{Line}(x_{\text{nearest}}, x_{\text{new}}))$
    for each $x_{\text{near}} \in X_{\text{near}}$ **do**
      if $\text{CollisionFree}(x_{\text{near}}, x_{\text{new}})$ **then**
        if $\text{Cost}(x_{\text{near}}) + \text{Cost}(\text{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
          $x_{\min} \leftarrow x_{\text{near}}$
          $c_{\min} \leftarrow \text{Cost}(x_{\text{near}}) + \text{Cost}(\text{Line}(x_{\text{near}}, x_{\text{new}}))$
        $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$
    for each $x_{\text{near}} \in X_{\text{near}}$ **do**
      if $\text{CollisionFree}(x_{\text{new}}, x_{\text{near}})$ **then**
        if $\text{Cost}(x_{\text{new}}) + \text{Cost}(\text{Line}(x_{\text{new}}, x_{\text{near}})) < \text{Cost}(x_{\text{near}})$ **then**
          $x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}})$
          $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$

B. Augmented Methods

Generally, both RRT and RRT* plans before the robot arm starts to execute can work pretty well with a static environment. However, in reality, most environments are not static, there are always going to moving obstacles. This means we have an algorithm that can modify the tree and trajectory timely as the environment changes. So the robot arm can execute certain tasks while avoiding the potential obstacles in its way or dodging the obstacles that are coming at it.

Therefore we proposed the Real-Time Rapidly-exploring Random Trees * (RT-RRT*) algorithm, inspired by [8]. It is a real-time RRT-based motion planner, which can cooperate with moving obstacles by adding an extra re-route step. It can be broken down into pre-planning and runtime-planning two parts.

For pre-planning, it just mostly follows what RRT* was doing. During Runtime, for every step, RT-RRT* will check whether there is an object within its vicinity. If so, it will abandon the current trajectory and start to replan for certain steps.

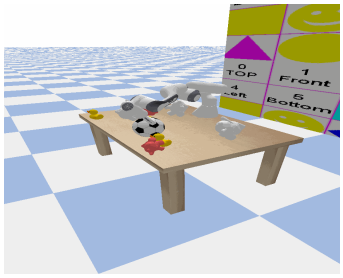


Fig. 5: Environement 1 testing

The pseudo-code of the RT-RRT* algorithm is as follows (Since the pre-plan part is the same as RRT, it is skipped here):

Algorithm 3 RT-RRT* Algorithm

$V \leftarrow \{x_s\}$; $E \leftarrow \emptyset$
$G = (V, E) \leftarrow RRT*$
$Traj \leftarrow G = (V, E)$
while $\|\texttt{end\_effector.pos} - \texttt{goal.pos}\| > \epsilon$ **do**
  if vicinity_check = True **then**
    $Traj = \emptyset$
    for $i = 1 \dots m$ **do**
      $RRT^*(G).RUN$
      return $G' = (V', E')$
REACH GOAL

IV. Results

Since the environment RRT and RRT* are designed for static environments, they were only doing test runs on environments 1 and 2. As for RTRRT*, since it was designed to cooperate with moving obstacles we only tested it in environment 3. The results are as follows:

A. Environment 1

In the pictures presented here, we can see both RRT, RRT*, and RT-RRT* can kick the target ball successfully without colliding with any obstacles around it. However, due to the simple setting, we are not able to see much difference between them.

Since RRT does not guarantee optimality, even under the simplest cases, it does not work every time. Sometimes the arm would go completely different direction, and take huge detours, even though it ends up finding the goal. There are also times that the arm is just stuck in one configuration and not moving at all. If it is given more iteration steps, things would go better, but it does not change much.

As for RRT*, it had a much better performance under the same iteration time due to its rewire step. The trajectory is much direct and it works most of the time.

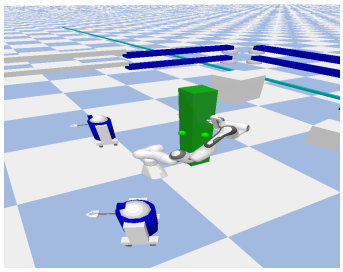


Fig. 6: Environment 2 testing

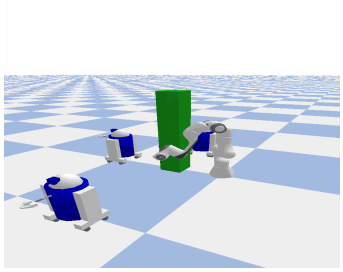


Fig. 7: Environment 3 testing

B. Environment 2

In this environment, the RRT is not working well, given it is a much more complex environment than the previous one. In most of the cases, it is only able to reach one of the 4 goals at most two. After it reached a goal, it would just stuck there. It cannot pull the arm away from the goal to re-plan for the next one after it reaches the previous one.

At the same time, RRT* demonstrates a much better performance. In the best case, it can reach all four goals consecutively. Most of the time, it can at least reach one to two. However, there is strong evidence to believe that our algorithm is applicable it just a lack of time and computation resources to do test runs.

C. Environment 3

Under the third environment, as expected, neither RRT nor RRT * worked, the path they had was generated before the run-time and they did not expect the moving obstacles.

As for RT-RRT*, we can see the sign of it detecting the obstacles that are coming and it starts to reroute. However, most of the time it would still collide into the obstacles. It is fair to believe that to successfully do real-time planning we need an algorithm that is lighter and quicker.

V. Discussion

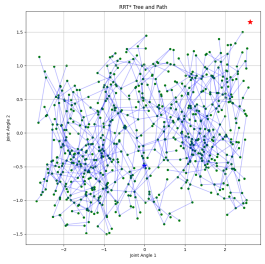
A. Inverse Kinematics

Since all the planning was done in a space composed of joint limits, while our goal is to reach the robot arm end-effector to a certain point in 3D Cartesian space, we need taken consider the transformation in between. At first, the goal in joint space was simply set by using PyBullet to generate goal points. However, it did not work well, because PyBullet was simply computing the goal point without considering the existence of the obstacles, which can lead to the goal joint configuration is actually in collision with obstacles.

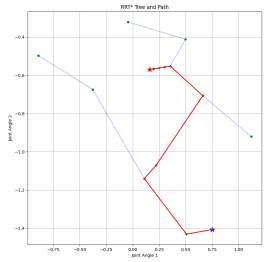
To solve this problem, we developed an algorithm based on damped least square optimization that can run through all the configurations when the robot arm end-effector is set at the goal point, filter out those in collision with obstacles, and find the optimal ones among those that are not. This set a solid foundation for our success.

B. Joints limit

Throughout the planning and execution, how the algorithm was sampled, the tree was built and the nodes were connected was plotted in joints space, as we can see in Fig. 5. One important piece of information that we can extract from the plot is that when the goal lies nears the joints limit, it is much harder for the algorithm to connect to it even with much more trial. From the plot we can see that, most of the sampling was done around the center, the more it goes outward, the fewer samples we have, which can be explained why our algorithm is not working sometimes. This also explains why the arm gets stuck easily when it is close to the obstacles.



(a) RRT* failed to find the path



(b) RRT* successfully found the path

Fig. 8: Comparison of RRT* algorithm in different scenarios.

To cooperate with this problem, we actually scale down the joints limit by around 0.9 to 0.85 which makes sure the goal is not set too close to the real joints limit. Therefore, we can say that if this scaling factor is properly set, we are able to get much better results.

VI. Conclusion

In this project, we successfully implemented and evaluated motion planning algorithms for a 7-DoF Franka Emika Panda robot arm in various scenarios. By using RRT, RRT*, and an augmented real-time variant, RT-RRT, we demonstrated their respective strengths and limitations across static and dynamic environments. Our results showed that RRT* improved trajectory efficiency over RRT through its rewiring step, while RT-RRT* offered a promising solution for real-time replanning in environments with moving obstacles, albeit with challenges in computational efficiency.

We identified key factors that affect performance, such as inverse kinematics computation, joint limits, and obstacle

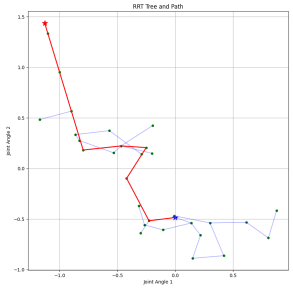


Fig. 9: Environment 1 with 1 goal, trial 1

handling. By scaling joint limits and optimizing goal configurations, we improved the reliability of planning in constrained spaces. However, achieving seamless real-time performance remains an open challenge that requires lightweight and faster algorithms.

Future work could focus on further optimizing RT-RRT* for dynamic environments and exploring hybrid approaches that combine sampling-based methods with learning-based techniques for enhanced adaptability and efficiency.

REFERENCES

[1] Khatib, Oussama. "Real-time obstacle avoidance for manipulators and mobile robots." The international journal of robotics research 5.1 (1986): 90-98.

[2] Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." IEEE transactions on Systems Science and Cybernetics 4.2 (1968): 100-107.

[3] Dijkstra, E.W. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 (1959).

[4] Stentz, Anthony. The D* algorithm for real-time planning of optimal traverses. Carnegie Mellon University, the Robotics Institute, 1994.

[5] LaValle, Steven. "Rapidly-exploring random trees: A new tool for path planning." Research Report 9811 (1998).

[6] Kavraki, Lydia E., et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." IEEE transactions on Robotics and Automation 12.4 (1996): 566-580.

[7] Karaman, Sertac, and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." The international journal of robotics research 30.7 (2011): 846-894.

[8] Naderi, Kourosh, Joose Rajamäki, and Perttu Hämäläinen. "RT-RRT* a real-time path planning algorithm based on RRT." Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games. 2015.

APPENDIX

All the visualization in joint space for different environments and algorithms.

A. *RRT Visualization*

... (truncated for brevity, but all captions updated in the same pattern) ...

B. *RTRRT* Visualization*

C. *RRT* Visualization*

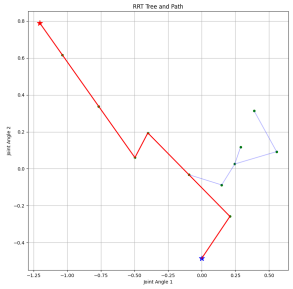


Fig. 10: Environment 1 with 2 goals, trial 1

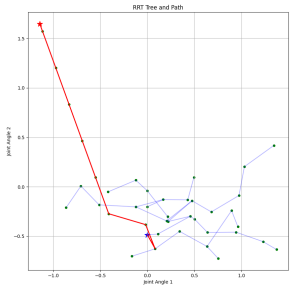


Fig. 11: Environment 1 with 3 goals, trial 1

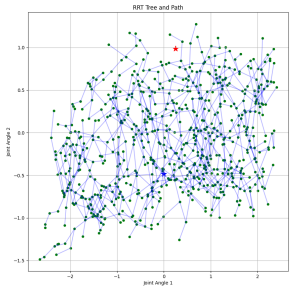


Fig. 12: Environment 3 with 1 goal, trial 1

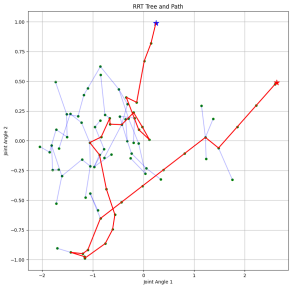


Fig. 13: Environment 3 with 1 goal, trial 2

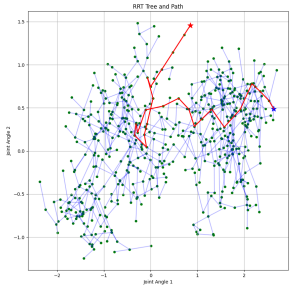


Fig. 14: Environment 3 with 1 goal, trial 3

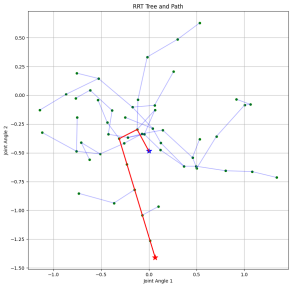


Fig. 15: Environment 3 with 2 goals, trial 1

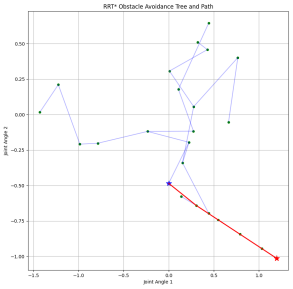


Fig. 16: Environment 1 with 1 goal, trial 1

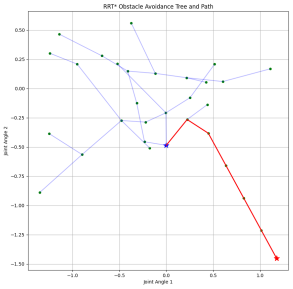


Fig. 17: Environment 1 with 2 goals, trial 1

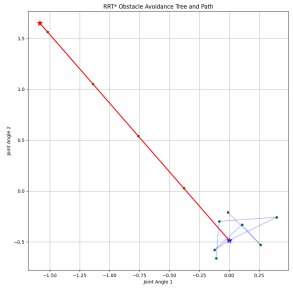


Fig. 18: Environment 1 with 3 goals, trial 1

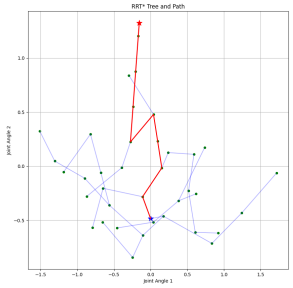


Fig. 19: Environment 1 with 1 goal, trial 1

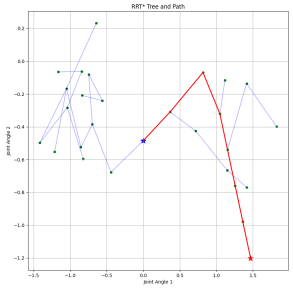


Fig. 20: Environment 1 with 2 goals, trial 1

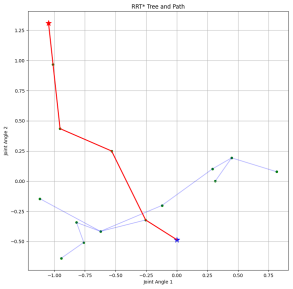


Fig. 21: Environment 1 with 3 goals, trial 1

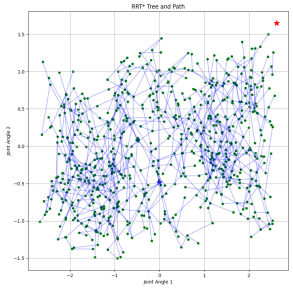


Fig. 22: Environment 3 with 1 goal, trial 1

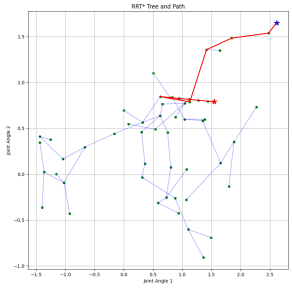


Fig. 23: Environment 3 with 1 goal, trial 2

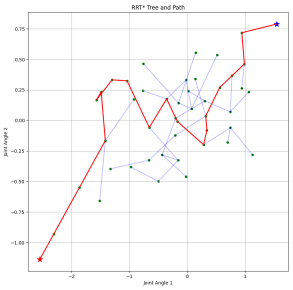


Fig. 24: Environment 3 with 1 goal, trial 3

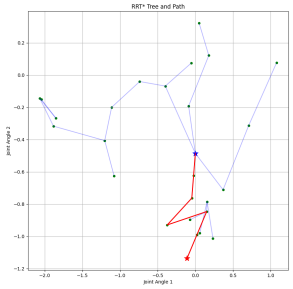


Fig. 25: Environment 3 with 2 goal, trial 1

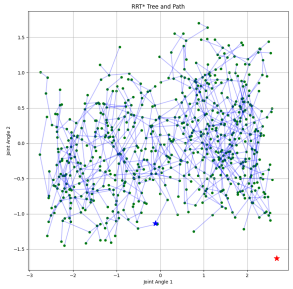


Fig. 26: Environment 3 with 2 goal, trial 2

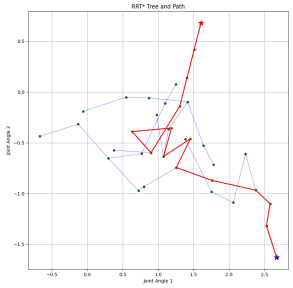


Fig. 27: Environment 3 with 2 goal, trial 3

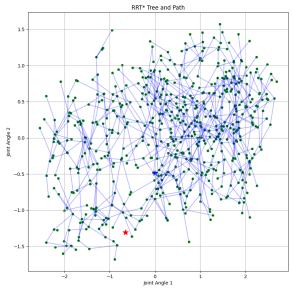


Fig. 28: Environment 3 with 3 goal, trial 1

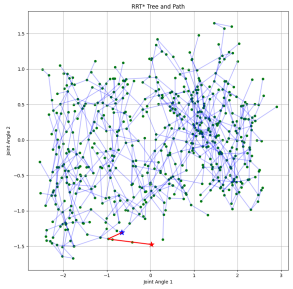


Fig. 29: Environment 3 with 3 goal, trial 2

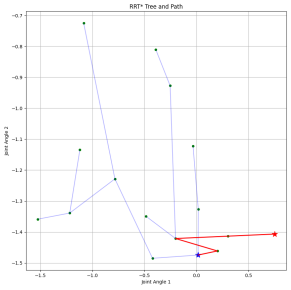


Fig. 30: Environment 3 with 3 goal, trial 3

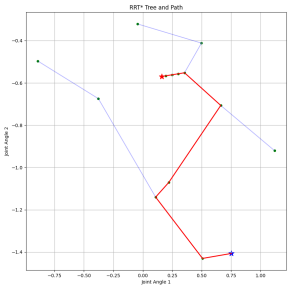


Fig. 31: Environment 3 with 3 goal, trial 4

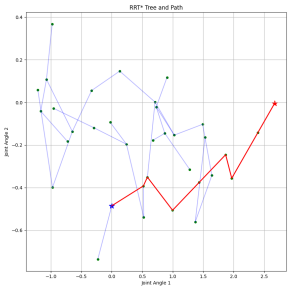


Fig. 32: Environment 4 with 1 goal, trial 1

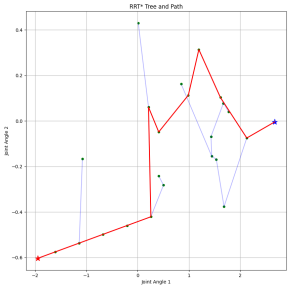


Fig. 33: Environment 4 with 1 goal, trial 2

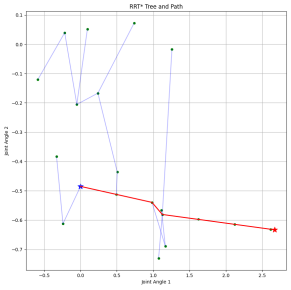


Fig. 34: Environment 4 with 2 goal, trial 1

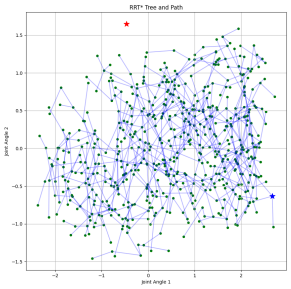


Fig. 35: Environment 4 with 2 goal, trial 2

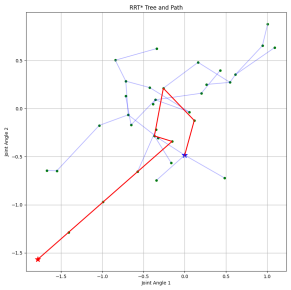


Fig. 36: Environment 4 with 3 goal, trial 1

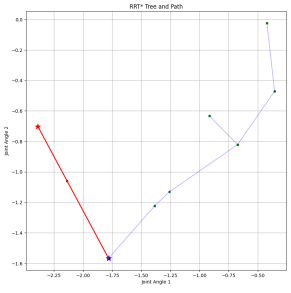


Fig. 37: Environment 4 with 3 goal, trial 2