Motion Planning

Sandeep Chintada
A59015527
dchintada@ucsd.edu

***Abstract*—This project aims to compare the performance of search-based and sampling-based motion planning algorithms in 3D Euclidean space. It involves testing the algorithms in a range of challenging scenarios to assess their effectiveness in handling various types of obstacles.**

***Index Terms*—Motion Planning, A*, RRT, Deterministic Shortest Path**

I. Introduction

Path planning or motion planning is essential in robotics to determine the optimal path for achieving objectives or reaching target locations. It involves generating a sequence of safe and efficient actions to navigate the environment. For instance, self-driving cars use path planning to find the best route, while surgical robots rely on it to perform tasks safely and effectively.

In Project 1, we employed a dynamic programming approach to devise a path to the goal while adhering to the specified constraints. However, it became evident that as the constraints increased, the state space expanded exponentially, rendering the planning process intractable in complex environments. This was primarily due to the fact that the dynamic programming approach required computing the optimal path from every individual location to the goal.

To address this challenge, alternative approaches were formulated and the efficacy of search-based motion planning algorithms, such as A*, and sampling-based motion planning algorithms, like RRT was discovered. These algorithms offer a viable solution with significantly reduced computational complexity.

The A* algorithm stands out for its capability in efficiently exploring discrete state spaces by leveraging heuristic search techniques, evaluating potential paths and selecting the most promising ones based on a combination of actual cost and estimated cost-to-goal, making it particularly well suited for autonomous navigation and in other applications where a precise path must be planned, for example, in surgical robotics.

Sampling-based motion planning algorithms, such as RRT, on the other hand, are effective in high-dimensional configuration spaces with complex obstacles. RRT builds a tree like structure from samples and can rapidly explore uncharted regions of the configuration space, which is essential for navigation in complex and dynamic environments.

II. Problem Formulation

The objective of the project is to find the path that minimizes the total cost from the start location to the goal location given the environment constraints in a continuous 3D space. This can be formulated as a deterministic shortest path problem which is itself equivalent to a deterministic optimal control problem. In the deterministic shortest path formulation, the state space is represented by a graph where the nodes represent the state space where the edges represent the possible transitions and the associated costs.

A. Mathematical primitives

Consider a graph with a vertex set $\mathcal{V}$, edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ and edge weights $\mathcal{C} := \{c_{ij} \in \mathbb{R}\{\infty\} | (i,j) \in \mathcal{E}\}$, where c_{ij} represents the edge cost from node i to node j. In the given problem, nodes would be the locations on the 3D Euclidean space and edge costs would be the distance between any two nodes.

(a) A path is defined as a sequence $i_{1:q} = (i_i, i_2, \ldots, i_q)$ of nodes $i_k \in \mathcal{V}$

(b) Path length is defined as the sum of edge weights along the path:
$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$$

(c) All paths from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$ are defined mathematically as follows:
$$\mathcal{P}_{s,\tau} = \{i_{1:q} | i_k \in \mathcal{V}, i_1 = s \text{ and } i_q = \tau\}$$

(d) The objective as discussed earlier is to minimize the total cost from start node s to the goal node τ, i.e., the minimum path length of all the paths $\mathcal{P}_{s,\tau}$.
$$\text{dist}(s,\tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}} \qquad i^*_{1:q} = \arg \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J^{i_{1:q}}$$

(e) The assumption in all the steps is that there are no negative cycles in the graph i.e., $J^{i_{1:q}} \geq 0$ for all $i_{1:q} \in \mathcal{P}_{i,i}$ and all $i \in \mathcal{V}$. This is to avoid infinite loops within the graph when traversing.

In essence, we build a graph, assign weights to the graph and find the path that minimizes the cost from start to the goal. Now that we have defined our problem in mathematical terms, we can move on to the details of the algorithms.

III. Technical Approach

To address the deterministic shortest path problem outlined in the aforementioned formulation, we have implemented two distinct methods: a search-based approach utilizing the weighted A* algorithm, and a sampling-based approach employing the RRT(Rapidly Exploring Random Tree) algorithm. These methods differ primarily in how they construct the underlying graph representation. The weighted A* algorithm, belonging to the search-based category, constructs a graph by systematically exploring the state space. It utilizes a heuristic function, combined with a weighted cost evaluation, to guide the search towards the most promising paths. In contrast, the RRT algorithm, constructs the graph through a random sampling approach. It starts with an initial configuration and incrementally generates new nodes by sampling the state space. The algorithm connects these newly sampled nodes to the existing graph, gradually expanding it in a probabilistic manner and hope to reach the goal.

A. Collision Checking

We are given a 3D Euclidean space with a well-defined boundary and obstacles represented by Axis Aligned Bounding Boxes(AABBs). When the graph is constructed, we need to ensure that the new nodes do not lie within the obstacle or worse, the edge going through the obstacle. We implement a collision checking subroutine to prevent this from happening. We say that a line-segment is colliding with the cuboid if either of its vertices lie within the cuboid. To represent it mathematically, we perform a series of mathematical checks. The algorithm begins by assigning the start and end points of the line segment as P1 and P2, respectively. The minimum and maximum coordinates of the block, denoted as AABBMin and AABBMax, are extracted from the block data given to us.

(a) We check if the line segment lies entirely outside the block along each dimension. If the maximum x-coordinate of P1 and P2 is less than the minimum x-coordinate of the block, or if the minimum x-coordinate of P1 and P2 is greater than the maximum x-coordinate of the block, then there is no intersection. This check is repeated for the y-coordinate and z-coordinate as well.
(b) Next, we compute the direction vector of the line segment by subtracting the corresponding coordinates of P1 and P2 to help us find the potential intersection points.
(c) We then compute the parameter values for the intersection of the line segment with the boundaries of the block. This is done by calculating the ratios of the differences between the minimum or maximum coordinates of the block and the corresponding coordinates of P1, to the components of the direction vector.
(d) Then, we determine the largest value among the smallest intersection parameters (tEnter) and the smallest value among the largest intersection parameters (tExit). These values represent the entry and exit points of the line segment into and out of the block.
(e) If tEnter is greater than tExit or tExit is negative, it implies that the line segment either misses the block entirely or is already past it, resulting in no collision.
(f) If tEnter or tExit falls within the range [0, 1], it indicates that the line segment intersects the block. This condition is checked to confirm a collision.
(g) Lastly, if both tEnter and tExit are greater than 1, it implies that the entire line segment lies within the block, resulting in containment. This scenario is also considered a collision

B. Label Correction Algorithms

In project 1, we find out the shortest sequence of actions from the start to the goal by running the Dynamic Programming Algorithm (DPA) backwards from the goal. This algorithm gives the shortest paths from the all nodes to the goal and hence a large part of its computation is redundant. We can flip the DPA algorithm to begin from the start to give Forward DPA but it has the similar issues i.e., the algorithm computes the shortest path from the start to all other nodes. Consequently, we have another class of algorithms called label correcting(LC) algorithms that do not necessarily visit all nodes of the graph.

The label correcting algorithms employ a cost-to-arrive value function to prioritize the visited nodes. Each visited node $i \in \mathcal{V}$ is assigned a label g_i, representing the estimated optimal cost from the start node s to that specific node. Whenever a better path from the start node to a particular node is discovered, the labels of all its child nodes are updated accordingly. To facilitate this process, an OPEN set is maintained, containing the potential nodes along the optimal path to the goal. Nodes are systematically removed from the OPEN set to update their labels, repeating this iterative update until the goal is reached.

Dijkstra's algorithm is a specific version of the label correcting algorithms, where the node with the minimum label is sequentially removed from the OPEN list. On the other hand, the A* algorithm incorporates an additional heuristic into Dijkstra's algorithm, making admission to the OPEN list more stringent. This heuristic enhances the algorithm's efficiency by reducing the number of iterations required to reach the target node.

C. A algorithm*

As previously mentioned, the A* algorithm belongs to the category of label correcting algorithms and stands out due to its utilization of a heuristic function. This heuristic function, denoted as h_i, is created based on specialized knowledge of the problem at hand. The primary purpose of this function is to provide an informed estimation of the distance from the current node to the goal. For instance, if there is knowledge that the goal is situated on the right side, the heuristic function will guide the planner to avoid searching on the left side of the start node, resulting in a reduction in the number of iterations required.

Typically, the heuristic function involves a distance measure between each node and the goal, such as Euclidean, Manhattan, or Octile distances. However, certain constraints are imposed on the selection of the heuristic function to ensure the optimality of the A* solution. These restrictions are implemented to guarantee that the heuristic remains admissible and consistent, allowing the algorithm to find the optimal path.

The first restriction is admissibility, expressed mathematically as:

$$h_i \leq \text{dist}(i, \tau)$$

. his condition ensures that the heuristic's estimate is always less than or equal to the actual distance from any node i

to the goal τ. By satisfying this criterion, we guarantee that the estimated cost (label) never underestimates the true cost. Failing to meet this requirement could result in a suboptimal and inefficient solution, where a larger number of nodes would need to be expanded, and the computed paths might be longer than the actual shortest path.

The second restriction is consistency defined as:

$$h_i \leq c_{ij} + h_j$$

where j is a child of i. This means that the estimated cost from a node to its successor plus the cost of reaching that successor is less than or equal to the estimated cost from the current node to the goal. Additionally, there is a concept of ϵ-consistency wherein we place significant trust in our heuristic hoping for a faster convergence but it may lead to suboptimal paths.

By using an appropriate heuristic function, we can search large spaces and prioritize likely nodes that lead to the target. It can be shown that if a heuristic is admissible and consistent, we are guaranteed to find the optimal path from start to the goal. If we are using an inconsistent heuristic, we could still find the optimal path if we reopen the CLOSED states. In case of ϵ-consistent heuristics, it is guaranteed to return an ϵ-suboptimal path, i.e.,

$$\text{dist}(s, \tau) < g_\tau < \epsilon\text{dist}(s, \tau)$$

leading to a weighted-A* algorithm.

Now that we have defined all the primitives, the weighted-A* algorithm is as follows(A* is a special case when $\epsilon = 1$):

Algorithm 1 Weighted A* Algorithm

```
  Input: vertices $\mathcal{V}$, start $s \in \mathcal{V}$, goal $\tau \in V$, and costs $c_{ij}$
for $i, j \in \mathcal{V}$
0: OPEN ← {s} , CLOSED ← {}, $\epsilon \geq 1$
0: $g_s = 0, g_i = \infty, \forall i \in \mathcal{V} \neq s$
0: while $\tau \notin$ CLOSED do
0:   Remove $i$ with smallest $f_i = g_i + \epsilon h_i$ from OPEN
0:   Insert $i$ into CLOSED
0:   for $j \in Children(i)$ and $j \notin$ CLOSED do
0:     if $g_j > g_i + c_{ij}$ then
0:       $g_j \leftarrow g_i + c_{ij}$
0:       $Parent(j) \leftarrow i$
0:       if $j \in$ OPEN then
0:         Update priority of $j$
0:         if Otherwise then
0:           OPEN ← OPEN $\bigcup j$
```

We have previously defined the vertices $\mathcal{V}$ as points in a 3D Euclidean space, and g_i represents the label of the i^{th} vertex, which serves as the current estimation of the cost to arrive at that vertex. Within this framework, we utilize two lists, namely OPEN and CLOSED. Once a node enters the CLOSED list, its cost remains unchanged throughout the algorithm's execution.

The determination of which node enters the CLOSED list from the OPEN set is based on their f values, which are the sum of the current label and the weighted heuristic value. As we progress, the labels of nodes in the OPEN list undergo changes, and the most promising ones are selected for inclusion in the CLOSED list.

The termination of the algorithm occurs when the goal node is reached, identified by its presence in the CLOSED set. At this point, we can retrace the path back to the starting point. The time complexity of this algorithm is $\mathcal{O}(|\mathcal{V}|^2)$, while its memory complexity is $\mathcal{O}(|\mathcal{V}|)$ when utilizing priority queues for implementation.

D. A* implementation

In our implementation, each vertex is encapsulated as an object with various attributes, including parent, g-value, and h-value. During initialization, the g and h-values of each vertex are set to infinity by default. To represent the OPEN nodes, we employ a priority queue, where the keys correspond to the vertex locations, and the values are their associated h-values. This approach offers the advantage of updating priorities without the need to delete and reinsert elements into the queue. Additionally, we maintain a separate dictionary called 'Graph,' which serves as a mapping between location tuples and the corresponding vertex objects in Python

During the execution of our algorithm, we dynamically construct the graph rather than predefining it. This means that we simultaneously build and explore the graph. At each node, we have the ability to move in 26 different directions (27 - 1 combinations of (1,0)). Initially, we attempted to move equidistantly in all 26 directions, but this approach proved to be highly computationally expensive. To mitigate this, we adopted different step sizes for each direction. For instance, we moved 0.5 units straight, 0.5 multiplied by 1.4 units diagonally, and 0.5 multiplied by 1.7 units diagonally to another plane. In each direction, we move by a predefined step length, which corresponds to the chosen resolution.

To ensure collision-free paths, we examine whether the path from the current location to the next collides with any of the obstructing blocks using the previously discussed algorithm. If no collision is detected, we create a new node, add it to the graph, update its g-values, and append it to the OPEN set. Subsequently, we select the node with the lowest h-value from the priority queue, mark it as closed, and expand its children.

This process continues iteratively until we meet the exit criteria, which states that the distance between the goal and the agent should be less than the chosen step size. It's important to note that the code may not be organized in the exact order described above. Typically, we have a while loop that continues until the exit criteria are met, followed by the expansion of nodes. Once the exit criteria are satisfied, we traverse the path in reverse by following the parent attribute until we reach the goal.

E. RRT Algorithm

While A* algorithm systematically explores the graph by expanding nodes with the lowest h-values, this approach may prove to be inefficient for large environments with complex obstacles. In such cases, sampling-based path planning algorithms offer a solution. These algorithms construct a graph by randomly sampling points in the free configuration space to navigate around obstacles. Unlike A*, which guarantees resolution completeness, sampling-based algorithms achieve probabilistic completeness.

Sampling-based algorithms provide several advantages over A*. They tend to be faster and require less memory due to the sparsity of the graph they generate. By selectively sampling points, these algorithms can efficiently explore the configuration space and generate feasible paths without exhaustively searching every possible node. While A* guarantees optimality, sampling-based algorithms prioritize computational efficiency, making them suitable for scenarios where time and memory constraints are crucial.

In sampling-based motion planning, we define a set of primitive procedures as follows.

1) SAMPLE: returns iid samples from C(configuration space)
2) SAMPLEFREE: returns iid samples from Cfree (free configuration space)
3) NEAREST: given a graph $G = (V, E)$ with $V \subset C$ and a point x, returns a vertex v that is closest to x
$$\text{NEAREST}((V, E), x) = \arg\min_{v} \|x - v\|$$
4) NEAR: given a graph $G = (V, E)$ with $V \subset C$ and a point x, and $r > 0$, returns vertices in V that are at r-distance from x
$$\text{NEAR}((V, E), x, r) = \{v \in V |\ \|x - v\| < r\}$$
5) STEER$_\epsilon$: given points $x, y \in C$ and $\epsilon > 0$, returns a point $x \in C$ that minimizes $\|z - y\|$ while remaining within ϵ from x
$$\text{STEER}_\epsilon(x, y) = \arg \min_{z:\|z-x\|<\epsilon} \|z - y\|$$
6) COLLISIONFREE: given points $x, y \in C$, returns True if the line segment between x and y lies in Cfree and False otherwise.(Our collisioncheck function)

RRT is one of the many sampling based algorithms available and seemingly most popular. In this algorithm, we start at our initial configuration x_s and build the graph until the goal is part of the graph. The algorithm is as follows:

Algorithm 2 Algorithm 3: RRT

```
1: $V \leftarrow \{x_s\}$, $E \leftarrow \varnothing$, $n$
2: for $i = 1$ to $n$ do
3:     $x_{\text{rand}} \leftarrow$ SampleFree()
4:     $x_{\text{nearest}} \leftarrow$ Nearest($(V, E), x_{\text{rand}}$)
5:     $x_{\text{new}} \leftarrow$ Steer$_\epsilon(x_{\text{nearest}}, x_{\text{rand}})$
6:     if CollisionFree($x_{\text{nearest}}, x_{\text{new}}$) then
7:         $V \leftarrow V \cup \{x_{\text{new}}\}$
8:         $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$
9:         return $G = (V, E)$
10:    end if
11: end for=0
```

Here n is the no.of iteration we'd like to build the graph for. As there is no assurance of reaching the goal node, we introduce a predefined probability to sample the goal node and verify if a connection can be established within the existing graph. To further optimize the algorithm, we can implement enhancements such as bi-directional RRT and RRT connect.

In bi-directional RRT, we construct the graph simultaneously from both the start and goal nodes until they are successfully connected. This approach increases the likelihood of finding a feasible path by exploring the search space from multiple directions. RRT connect, on the other hand, is a variant of bi-directional RRT where attempts to connect the two trees are made at each iteration.

Upon obtaining a path, we employ path smoothing techniques to refine it and eliminate excessively rugged or irregular segments. Path smoothing enhances the quality of the generated path, making it more visually appealing and potentially more efficient to traverse.

F. RRT Implementation

To implement RRT, we utilize a Python library provided to us, leveraging its functionalities to construct the algorithm. We refer to the examples available in the library's repository and proceed to implement RRT within a custom class. Afterwards, we adjust the parameters according to our specific requirements.

For instance, we adapt the collision check resolution to 0.1, accommodating the complexity of our maps. We also set the goal sampling probability to 0.1, ensuring that the algorithm has a chance to sample the goal node during the construction of the graph. Additionally, we set the maximum number of samples to 50000, although it's worth noting that the algorithm typically converges around 15000 iterations.

In addition to these parameters, we fine-tune the Q variable. This variable encompasses a tuple specifying the number of points to be sampled at various distances. As we move farther away from the goal, we can sample points at a larger radius, while as we approach closer to the goal, we can reduce the sampling radius accordingly.

By calling the necessary functions and properly configuring the parameters, we can obtain the desired path output from the implementation of RRT. This allows us to generate a feasible path within the given environment.

IV. Results

Several experiments were conducted using A* with different step sizes. When the resolution was set to 0.5, four out of the total number of maps successfully reached the goal state with a final distance of 0.1, while all of the maps reached the goal with a final distance of 0.5. This outcome aligns with expectations since A* is known for its resolution completeness.

At a resolution of 0.5, the entire run for all the maps typically took around 10 to 15 seconds, and the results appeared satisfactory. However, as the resolution decreased, the computation time significantly increased. For instance, at a resolution of 0.3, the "Maze" map required approximately 150 seconds to complete, while the "Monza" and "Tower" maps took around 10 seconds each. At an even lower resolution of 0.1, the "Maze" map consumed more than 4 hours to finish, while the ”Monza” and ”Tower” maps each took approximately 10 minutes.

These findings indicate that as the resolution decreases, the computational demands of A* increase substantially. Maps with more intricate layouts, such as the ”Maze” map, tend to experience significantly longer computation times at lower resolutions compared to simpler maps like ”Monza” and ”Tower”.

At lower resolutions, the planned path generated by A* appears tightly constrained, resembling a skeleton graph that navigates from one obstacle edge to another in perfectly straight lines. As we increase the resolution, we observe a noticeable gap between the obstacles and the planned path, resulting in a suboptimal trajectory. However, it is important to note that the computational time required for higher resolutions is significantly greater.

In the accompanying figures, a comparison between the results obtained at different resolutions (0.2 and 0.5) and RRT is presented. A* demonstrates remarkably smooth paths, while RRT exhibits jagged trajectories due to the random nature of node selection, as depicted in the provided images. As a consequence of this randomness, the paths generated by RRT tend to be substantially suboptimal, with lengths reaching up to two or three times that of the optimal paths.

The computation time required for RRT is considerably less than that of A*, as expected, since RRT constructs a sparse graph. For instance, at a resolution of 0.5, A* took approximately 15 seconds to complete, whereas RRT only required a mere 3 seconds. However, it is worth mentioning that the success field of RRT may sometimes yield false results due to the potential limitations of the collision function implemented in the library, which may not be as robust as our own implementation

Another observation I made is the significant disparity in the number of nodes generated between A* and RRT when there is ample free space available. For instance, in the case of a single cube at a resolution of 0.5, A* produced approximately 580 nodes in the graph, whereas RRT only generated around 6 nodes. This trend was also noticeable in scenarios where a clear path existed from the start to the goal, although the difference between the two algorithms diminished as the environments became more complex. For example, in the case of the maze, A* at a resolution of 0.5 resulted in approximately 14,000 nodes in the graph, while RRT generated around 12,000 nodes in one run and 17,000 nodes in another run. It seems that RRT encounters challenges when the space between obstacles is minimal.

Interestingly, I noticed that increasing the epsilon value did not significantly impact the path lengths or quality. However, it had a notable effect on the number of nodes generated. For example, in the case of the cube, with an epsilon value of 1, there were 580 nodes, whereas with an epsilon value of 10, the number of nodes reduced to 190. Similarly, in the maze, the number of nodes decreased by approximately one-third, from 12,000 to 8,000, when the epsilon value was increased. However, the path length only increased by a small margin of 2 units.

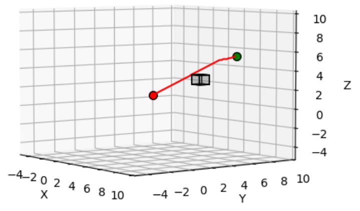

Fig. 1: Single Cube, res = 0.1

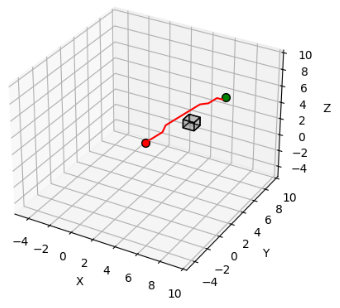

Fig. 2: Single Cube, res = 0.5

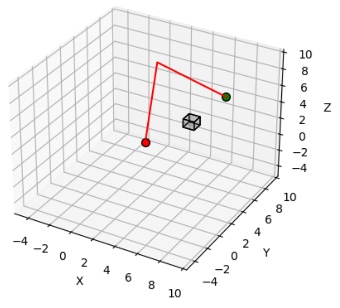

Fig. 3: Single Cube RRT

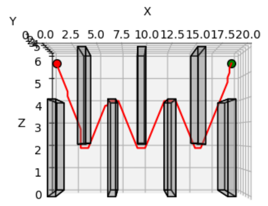

Fig. 4: flappybird, res = 0.2

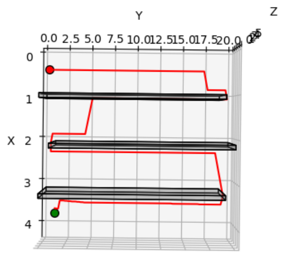

Fig. 7: monza, res = 0.2

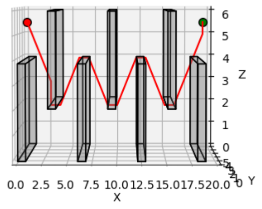

Fig. 5: flappybird, res = 0.5

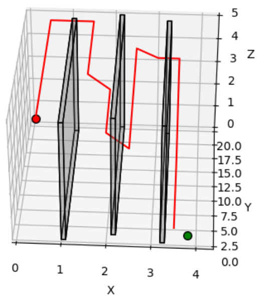

Fig. 8: monza, res = 0.5

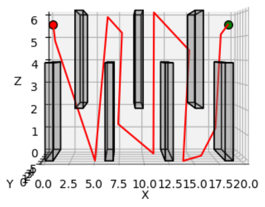

Fig. 6: flappy bird RRT

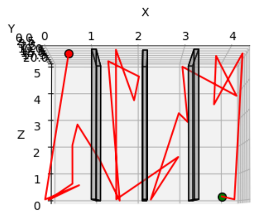

Fig. 9: monza RRT

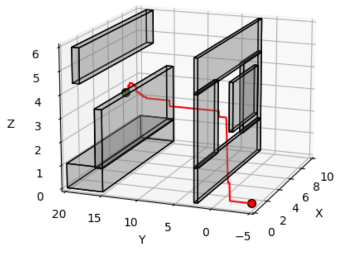

Fig. 10: window, res = 0.2

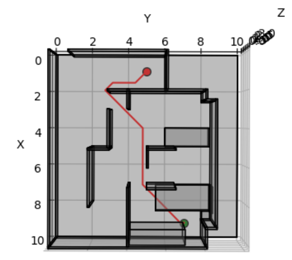

Fig. 13: room, res = 0.2

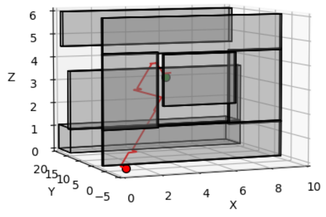

Fig. 11: window, res = 0.5

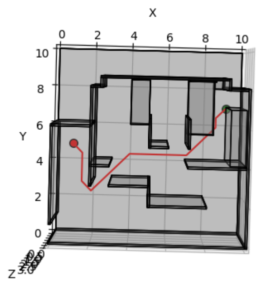

Fig. 14: room, res = 0.5

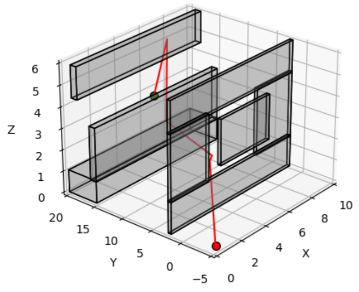

Fig. 12: window RRT

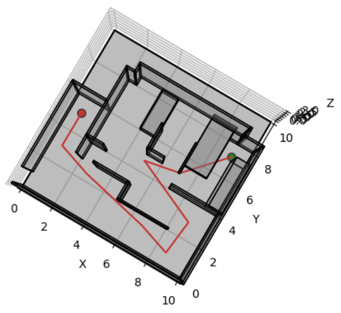

Fig. 15: room RRT

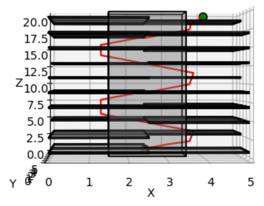

Fig. 16: tower, res = 0.2

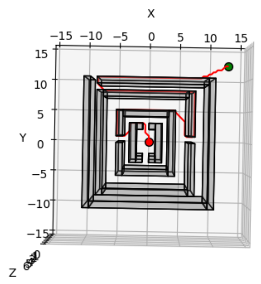

Fig. 19: maze, res = 0.3

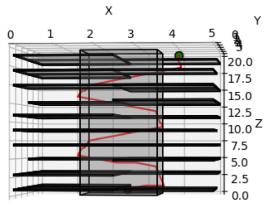

Fig. 17: tower, res = 0.5

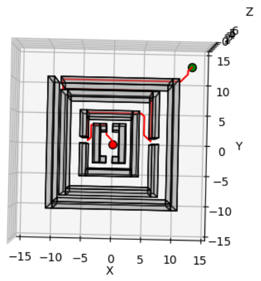

Fig. 20: maze, res = 0.5

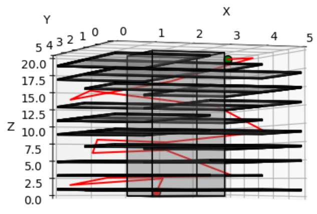

Fig. 18: tower RRT

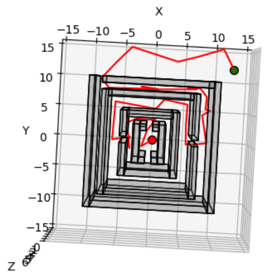

Fig. 21: maze RRT