MEAM 520

Lab 3: Trajectory Planning

Xinlong Zheng / Xiaozhou Zhang

xinlongz@seas.upenn.edu

xzzhang@seas.upenn.edu

Method

1.1 Task overview

The task is asked to write a program. take a struct map, start and end locations, $[x \quad y \quad z]$, expressed in the base frame, and return the path, an $N \times 5$ array containing the sequence of joint variable values along the trajectory that we send to the Lynx robot. In this case, we manage to use RRTs (Rapidly-exploring Random Trees) algorithm to accomplish the trajectory planning.

There is a MABLAB code *findpath.m* accomplishing the task and plotting the path listed in the Appendix 1 and attached to the file.

The program overall flowchart is shown as Figure 1-1.

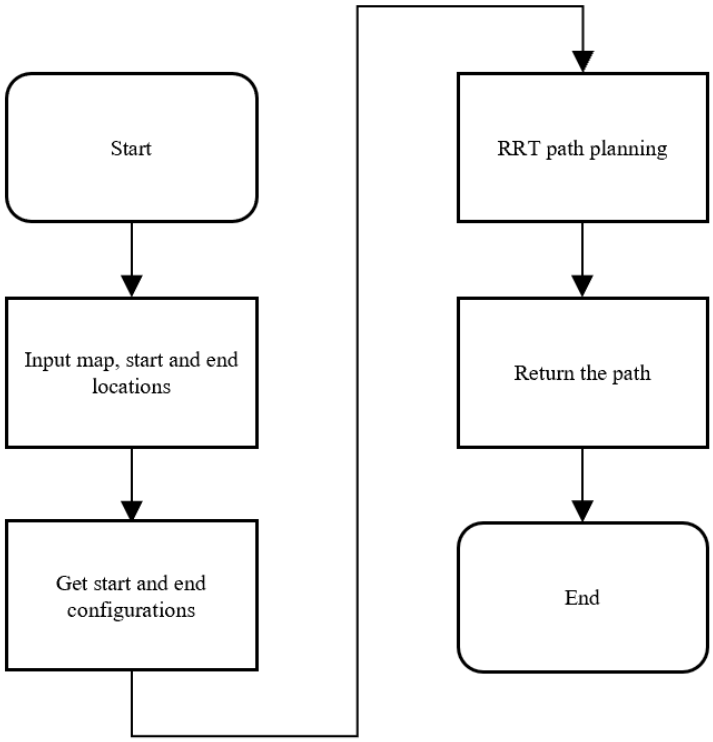

Figure 1-1 Overall flowchart for accomplishing the task

1.2 Get configurations with given locations

As we plan to implement RRTs algorithm in this task, we need to build the roadmap using sampling in the collision-free configuration space. Therefore, we need to know the configurations of the start and end positions.

The configuration can be obtained using inverse kinematics. Therefore, we need to generate a random rotation matrix from the base frame to the end- effector's frame, in order to provide a homogenous transformation matrix to input in the inverse kinematics.

As what was stated in the Lab2, there are some orientation which the end-effector can not reach due to the lack of 1 degree of freedom. Then we need to make the random rotation matrix R satisfy two conditions: the matrix is special orthogonal ($R \in SO(3)$), and it passes the mathematical check ($r_{13}y = r_{23}x$).

We treat the rotation matrix R as three basic rotations about the current x_e, y_e, and z_e axes, by the angle of θ_1, θ_2, and θ_3, respectively, as R_1, R_2, and R_3. The equation is shown as

$$R = R_1R_2R_3 = \begin{bmatrix}1 & 0 & 0\\ 0 & cos\theta_1 & -sin\theta_1\\ 0 & sin\theta_1 & cos\theta_1\end{bmatrix}\begin{bmatrix}cos\theta_2 & 0 & sin\theta_2\\ 0 & 1 & 0\\ -sin\theta_2 & 0 & cos\theta_2\end{bmatrix}\begin{bmatrix}cos\theta_3 & -sin\theta_3 & 0\\ sin\theta_3 & cos\theta_3 & 0\\ 0 & 0 & 1\end{bmatrix}$$

The θ_1 is fixed by a given location $o = [x \quad y \quad z]$ as $\theta_1 = atan2(y/x)$ because the wrist lacks 1 degree of freedom. Then we generate the value of θ_2 and θ_3 randomly, and take them back to the above equations to get a rotation matrix which satisfies the prerequisite. The homogenous transformation matrix is now obtained as

$$T = \begin{bmatrix}R & o\\ 0 & 1\end{bmatrix}$$

Now we can use inverse kinematics to get the configurations of the start and end positions.

The pseudocode of this part is shown as followed.

```
## Algorithm GetConfigurationFromPosition
Input o = [x, y, z]^T
if o in the boundary
    While true
        θ1 = atan2(y/x)
        θ2 = random value in the joint5's limits
        θ3 = random value
        R = R_{x,θ1} R_{y,θ2} R_{z,θ3}
        T = [R  o; 0  1]
        [q_IK, is_possible] = IK_lynx_sol(T)
        if is_possible
            if q_IK in the joint limits && Q_free
                q = q_IK
                break
            end if
        end if
    end while
else
    return error
end if
```

When it comes to determine whether the configuration lies in the collision-free configuration space, we need to check the collisions with the end-effector and the robot body, which will be stated in detail in the section 2.4.

There is a MABLAB code *getq.m* listed in the Appendix 1 and attached to the file.

1.3 RRT path planning

The map in this task is in five dimension thus creating a complete and feasible roadmap of the collision-free configuration space is relatively hard and time consuming. Since we do not need to run the task many times, we manage to plan trajectory using Rapidly-exploring Random Trees (RRTs) building the roadmap incrementally for single-query search.

We first generated uniform random configuration nodes in the collision-free configuration space and add the node of start configuration in the tree. As each node is sampled, a connection is attempted between it and the nearest node in the tree. If the connection is feasible (passes entirely through free space and obeys any constraints), and the end-effector does not travel a long distance (we set the step as 100 mm), we will add this new node to the tree. If the connection between the new node and the node of end configuration is also feasible and the end-effector does not travel longer than 100 mm, we will add the node of end configuration to the tree and the path is now found.

The pseudocode of this part is shown as followed.

```
## Algorithm BulidRRT
T = (q_start, ∅)
p_goal = position vector of end − effector by q_goal in W
For i = 1 to n_iter
        q = random configuration in Q_free
        q_a = closest node in T
        p = position vector of end − effector by q in W
        p_a = position vector of end − effector by q_a in W
        if NOT collide(q_a, q)
                if distance(p_a, p) < 100mm
                        Add (q_a, q) to T
                end if
        end if
        if NOT collide(q, q_goal)
                if distance(p, p_goal) < 100mm
                        Add (q, q_goal) to T
                        break
                end if
        end if
end for
```

When it comes to generate nodes in the collision-free configuration space and judge whether the connection is feasible, we need to check the collisions with the end-effector and the robot body, which will be stated in detail in the section 2.4.

1.4 Detect collisions

(1) Add the base support to map as an obstacle
When the robot is moving, we are trying to avoid the obstacles in the workspace. The main difference between the simulated workspace and the physical workspace is that the arm may collide with the base support, so we treat the base support as an obstacle and add it to the map, described as $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$, which is $[-60, -60, -150, 60, 60, 10]$ after the measurement.

(2) Check if the generated configuration lies in the collision-free configuration space
When we generate the node along with the start and end configuration, we need to make sure that they lie with in the collision-free configuration space. That is to check for the collision with the end-effector and the rest of the robot.
In this given task, after measuring the set-up maps, we find that the collision can only occur on the section of link 3, link 4, link 5 and end-effector. Thus, we make the geometry representation of the robot as Figure 1-2, in which the section of link 3 is represented as a cylinder. Link 4, link 5 and end-effector is represented as a cuboid, as what is shown as Figure 1-3, in which d_3 is the length of the diameter and d_e is the length of the diagonal.

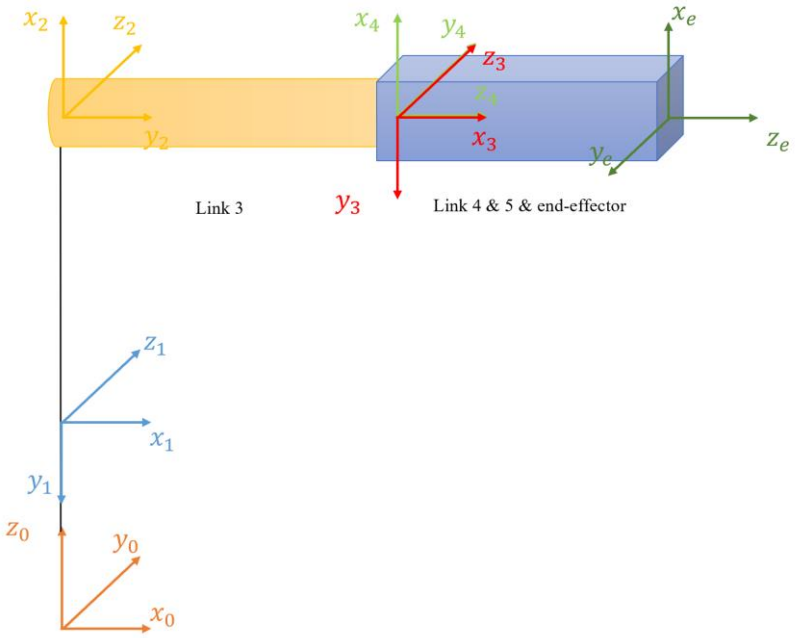

Figure 1-2 Geometric representation of the robot

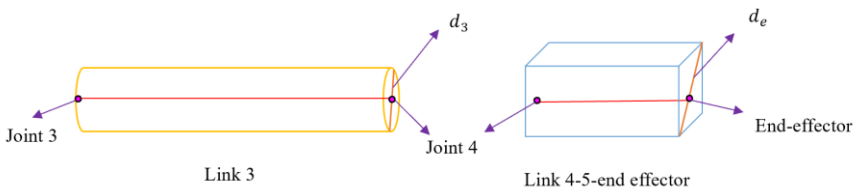

Figure 1-3 Geometric representation of link 3 and link 4-5-end effector

We inflate the block from extreme coordinates $[x^i_{\min}, y^i_{\min}, z^i_{min}, x^i_{\max}, y^i_{\max}, z^i_{\max}]$, to $[x^i_{\min} - \frac{d}{2}, y^i_{\min} - \frac{d}{2}, z^i_{\min} - \frac{d}{2}, x^i_{\max} + \frac{d}{2}, y^i_{\max} + \frac{d}{2}, z^i_{\max} + \frac{d}{2}]$, in which d is d_3 for checking link 3 and d_e for checking link 4-5-end effector. Thus, we turn the problem into detecting collisions between a line segment (red lines in Figure 1-3) and a rectangular prism (inflated blocks).
With a given configuration, we can determine the position of joint 3, joint 4 and end-effector using forward kinematics. Using the provided code *detectcollision.p*, taking the blocks, the position of joint 3 and joint 4, besides the position of joint 4 and end-effector to judge whether these two section is collided with obstacles. If the result is no, we consider the generated node lies in the collision-free configuration space.
The pseudocode of this part is shown as followed.

```
## Algorithm CheckQfree
generate node q
get position of joint 3, 4 and end effector
if NOT detectcollision (joint 3, 4, block) && (joint 4, e, block)
        q in Qfree
end if
```

(3) Check the collision in the movement between waypoints
Consider that the robot is moving from configuration A to configuration B, shown as Figure 1-4.

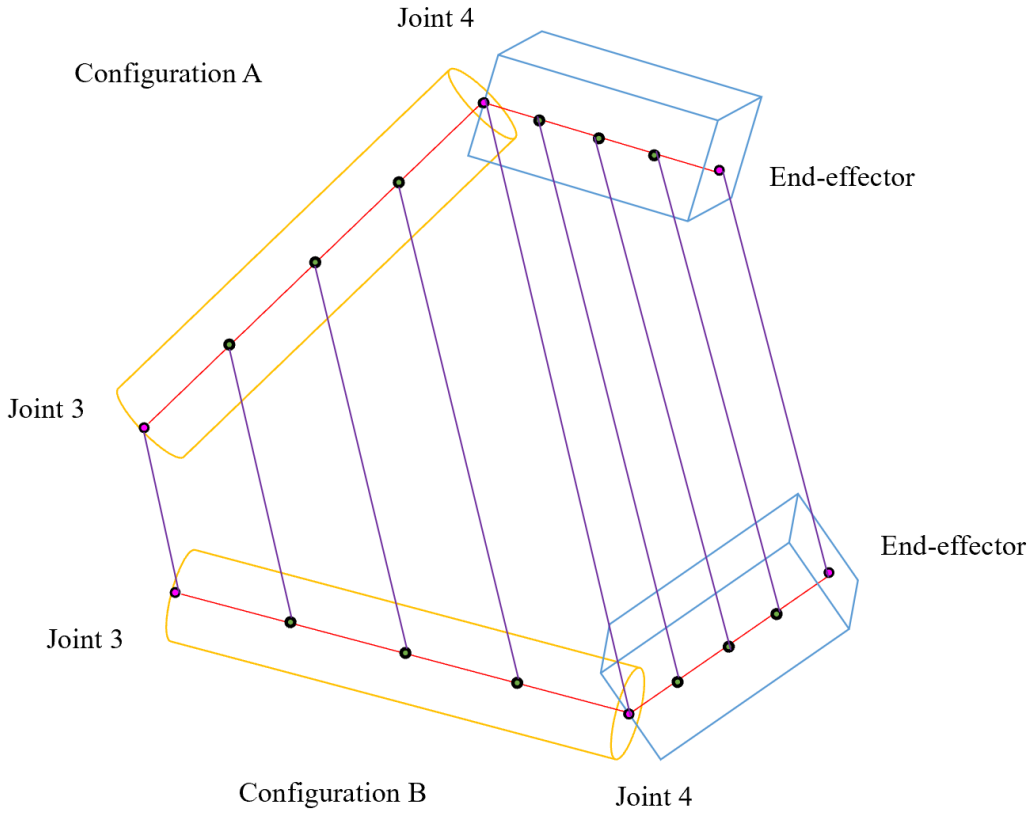

Figure 1-4 Robot moving between waypoints

We sample the line representation of robot (red lines in Figure 1-4) at 5 discretization. Since we set the step of the end-effector relatively small (limited to move within 100 mm a time), we may consider the trajectory between the sampled point as line segment. We turn the problem into detecting collisions between a line segment (purple lines in Figure 1-4) and a rectangular prism (inflated blocks). Using the same strategy in section (2), if the result is no, we consider the connection between these two nodes as feasible.
The pseudocode of this part is shown as followed.

```
## Algorithm CheckConnectionFeasible
inflate the block
node q_a, q_b
get positions of sampled points
if NOT detectcollision (sampled points_a, sampled points_b, block)
        connection is feasible
end if
```

Evaluation

2.1 Implement the planner in map_1

We try to plan for a relatively simple trajectory in this test.

In this test, we manage to move the robot from the middle of the obstacles to the below of the obstacles, as [180.1,0,222.25] to [220,0, −20]. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-1. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

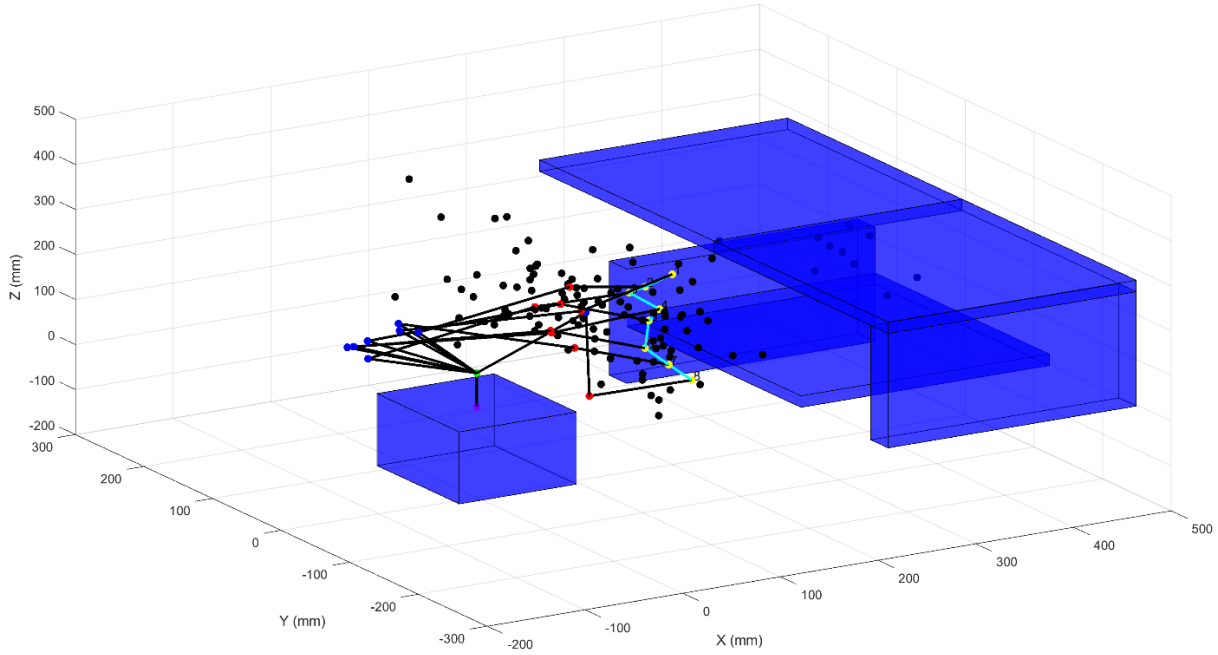

Figure 2-1 Trajectory of test 1

Click the Run and Time button in MATLAB, the result is shown as Table 2-1.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	216.950 *s*	2.170 *s*	100	100	100%

Table 2-1 Planner behavior on test 1

Because the robot in station 4 is down during our experiment, we are unable to implement the planner on the physical map. There is a video (Video 1) showing the simulated trajectory of test 1 plotting process listed in Appendix 2.

2.2 Implement the planner in simulated map_2

We try to test the planner when it's goal is near the base support.

In this test, we manage to move the robot from the above-left of the obstacles to the below of the obstacles, as [292.1,0,305.25] to [292.1,0,0]. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-2. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

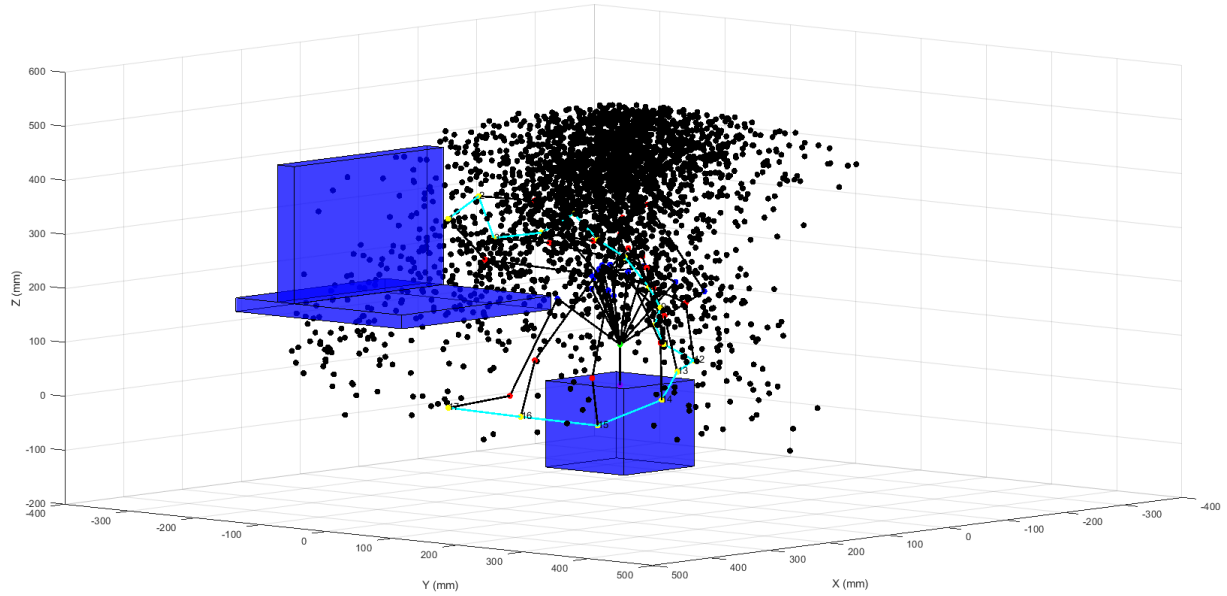

Figure 2-2 Trajectory of test 2

Click the Run and Time button in MATLAB, the result is shown as Table 2-2.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	1754.247 *s*	17.542 *s*	100	100	100%

Table 2-2 Planner behavior on test 2

Write a code, *physicalImplement.m*, to implement the planner in the physical map. There is a video (Video 2) showing the trajectory in Figure 2-2 implemented in the physical map_2 listed in Appendix 2.

2.3 Implement the planner in simulated map_3

We set three tests in map_3, and try to test the planner's performance when it comes to some narrow paths.

In the first test, we manage to move the robot from the middle of the obstacles to the below-left of the obstacles, as [292,0,222.25] to [300, −150,20]. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-3. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

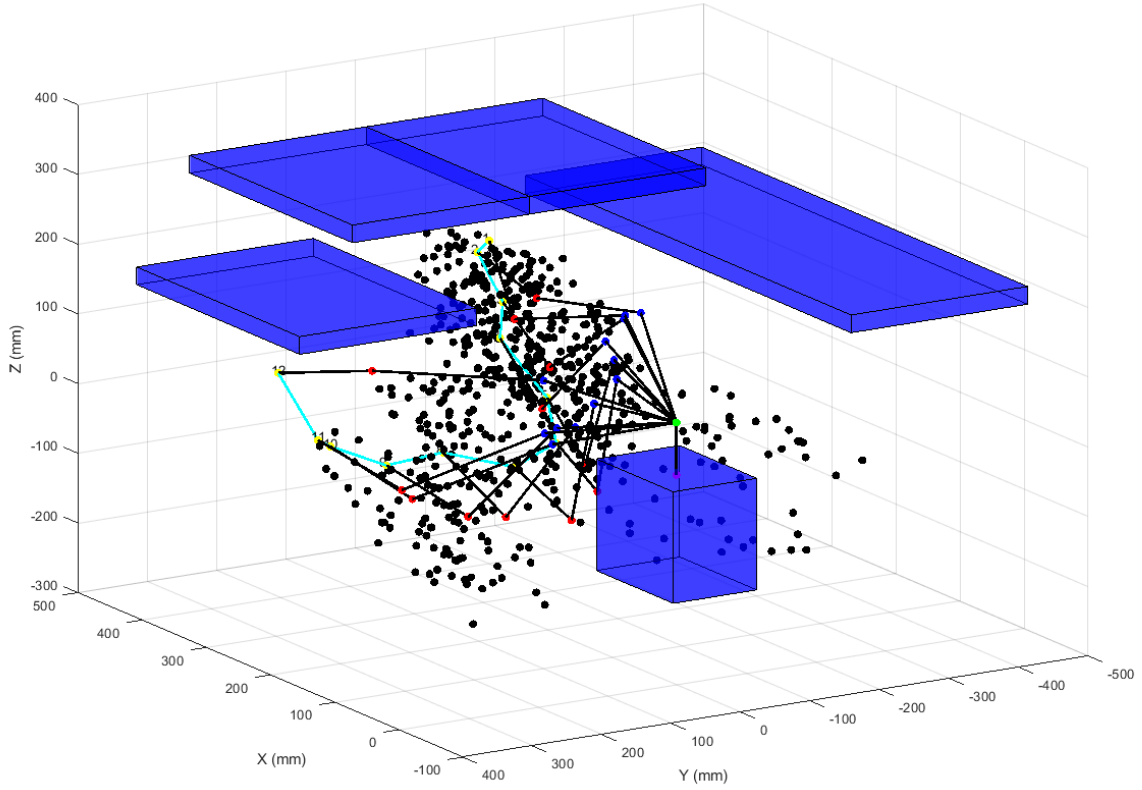

Figure 2-3 Trajectory of test 3.1

Click the Run and Time button in MATLAB, the result is shown as Table 2-3.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	537.394 *s*	5.374 *s*	100	100	100%

Table 2-3 Planner behavior on test 3.1

There is a video (Video 3) showing the trajectory in Figure 2-3 implemented in the physical map_3 listed in Appendix 2.

In the second test, we manage to move the robot from the middle of the obstacles to the above-right of the obstacles, as $[292,0,222.25]$ to $[100,-300,350]$. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-4. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

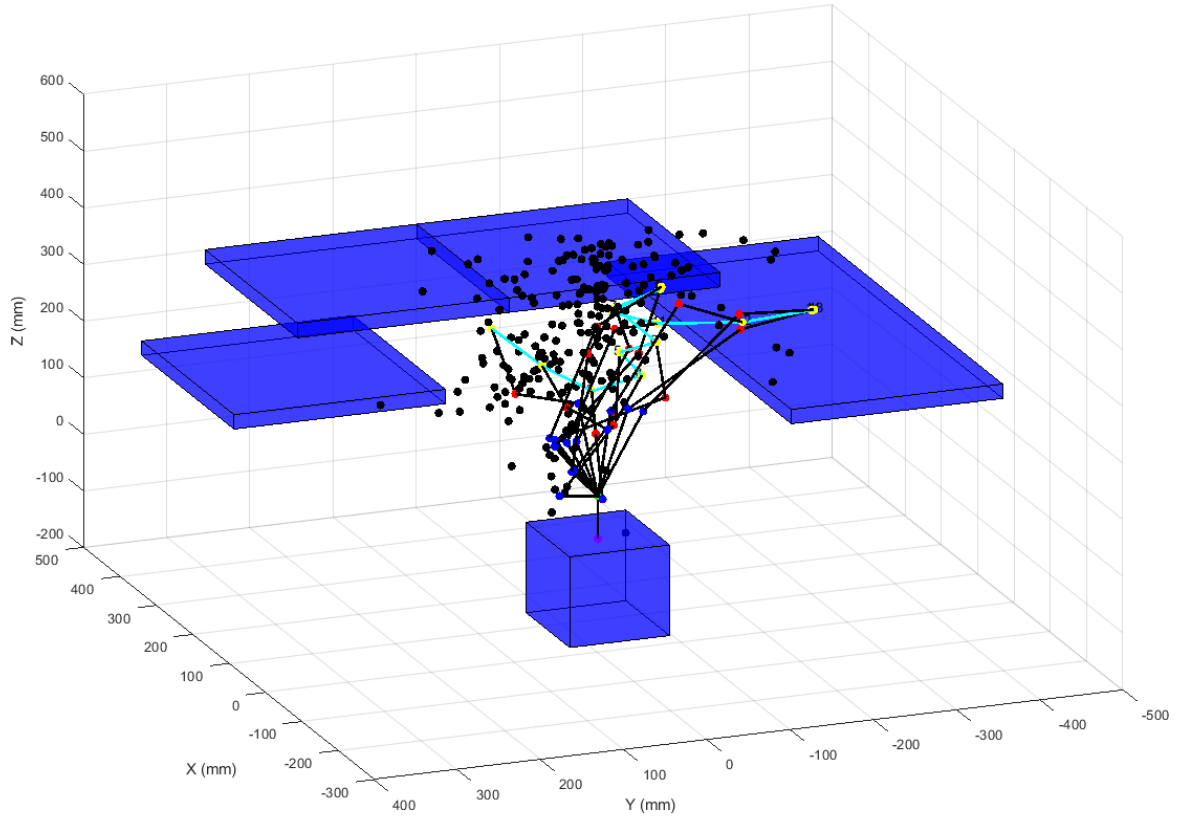

Figure 2-4 Trajectory of test 3.2

Click the Run and Time button in MATLAB, the result is shown as Table 2-4.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	4933.211 *s*	49.332 *s*	100	69	69%

Table 2-4 Planner behavior on test 3.2

There is a video (Video 4) showing the trajectory in Figure 2-4 implemented in the physical map_3 listed in Appendix 2.

In the third test, we manage to move the robot from the middle of the obstacles to the below-right of the obstacles, as [292,0,222.25] to [100, −300,100]. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-5. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

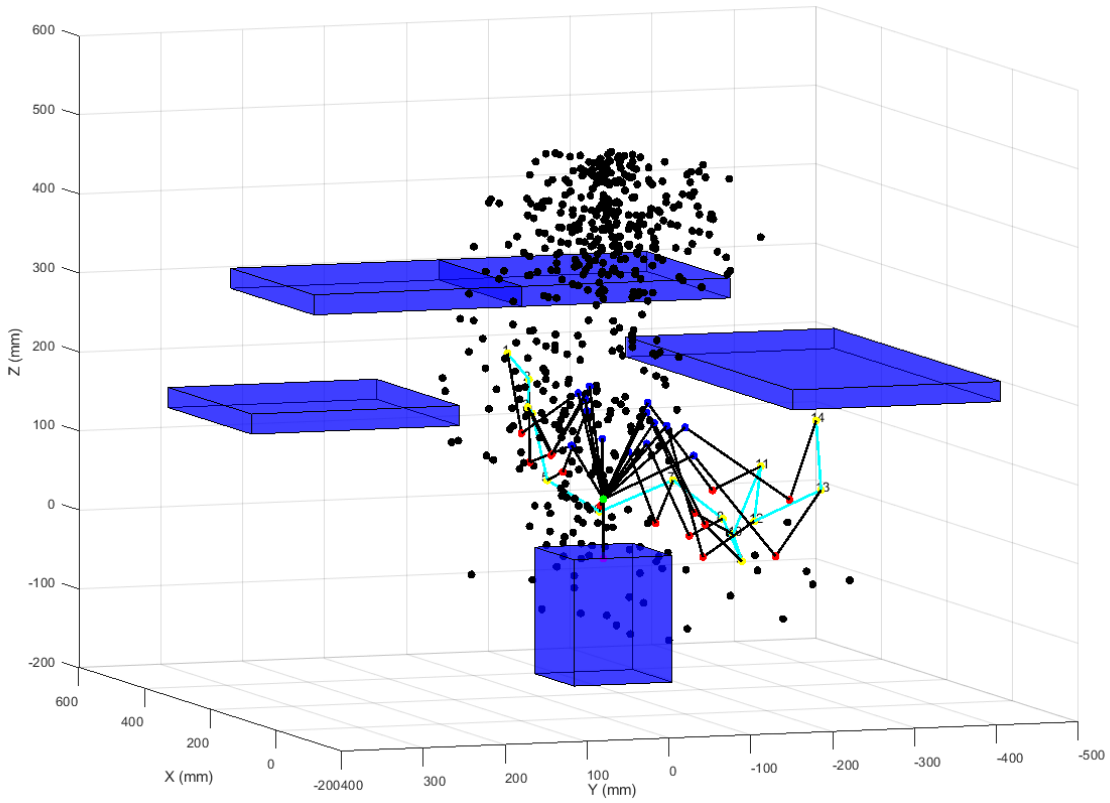

Figure 2-5 Trajectory of test 3.3

Click the Run and Time button in MATLAB, the result is shown as Table 2-5.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	1211.191 *s*	12.112 *s*	100	100	100%

Table 2-5 Planner behavior on test 3.3

There is a video (Video 5) showing the trajectory in Figure 2-5 implemented in the physical map_3 listed in Appendix 2.

2.4 Implement the planner in simulated map_4

We try to test the planner's performance when both the start and end configurations are near singular configuration.

In this test, we manage to move the robot from the middle of the obstacles to the left of the obstacles, as [292.1,0,280] to [300, −150,300]. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-6. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

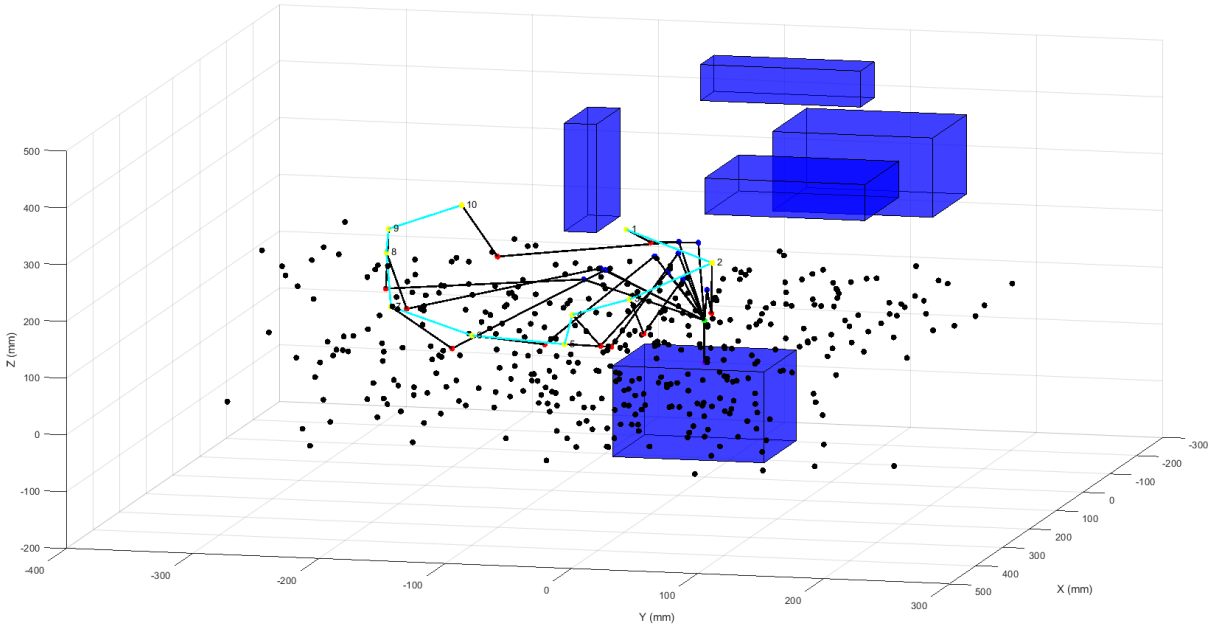

Figure 2-6 Trajectory of test 4

Click the Run and Time button in MATLAB, the result is shown as Table 2-6.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	278.126 *s*	2.781 *s*	100	100	100%

Table 2-6 Planner behavior on test 4

There is a video (Video 6) showing the trajectory in Figure 2-6 implemented in the physical map_3 listed in Appendix 2.

2.5 Implement the planner in simulated map_5

We create our own map and try to plan for a relative long-distance trajectory.

In this test, we manage to move the robot from the below-right of the obstacles to the above-left of the obstacles, as $[300, -100, 200]$ to $[300, 150, 300]$. Write a code to call the *findpath.m* 100 times to see the successful rate and the time complexity. One outputted trajectory of our planner is shown as Figure 2-7. The black points are the nodes which we discard. The light blue lines show the trajectory of the end positions.

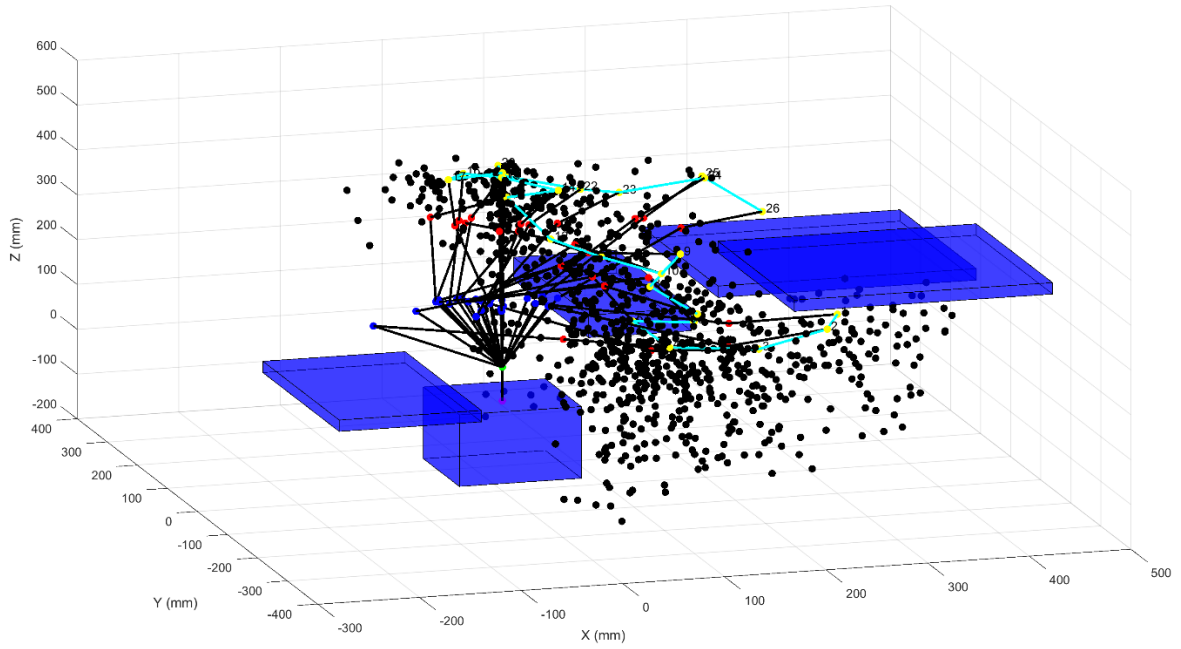

Figure 2-7 Trajectory of test 5

Click the Run and Time button in MATLAB, the result is shown as Table 2-7.

Function	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
findpath	2766.332 *s*	27.663 *s*	100	100	100%

Table 2-7 Planner behavior on test 5

2.6 Summery

The planner's performance in the previous 7 test are shown as Table 2-8.

Test	Total Time	Average Time	Number of Experiment	Number of Success	Successful Rate
1	216.950 *s*	2.170 *s*	100	100	100%
2	1754.247 *s*	17.542 *s*	100	100	100%

3.1	537.394 s	5.374 s	100	100	100%
3.2	4933.211 s	49.332 s	100	69	69%
3.3	1211.191 s	12.112 s	100	100	100%
4	278.126 s	2.781 s	100	100	100%
5	2766.332 s	27.663 s	100	100	100%

Table 2-8 Planner behavior in given tests

(1) Successful rate

In terms of successful rate, our planner performs well in 6 of 7 test. In test 3.3, the path between the start configuration and end configuration is too narrow, the total number of waypoints we sampled may be not enough for accomplishing the task.

(2) Running time

In terms of running time, we can see that the easier the task, the less time will be cost. The short trajectory will cost less time than the long one, and the wide path will cost less time than the narrow one. When the planner having trouble finding the path (test3.3) , the time would increase largely since it has to try all the nodes sampled in the collision-free configuration space.

(3) The difference among multiple run

Of course, the path is not the same over multiple run, because our sampling is random every time. However, when the test is relatively easy, the differences between each path may varies a lot while the harder test would produce relatively minor differences between each path. This is shown as Figure 2-8 and Figure 2-9.

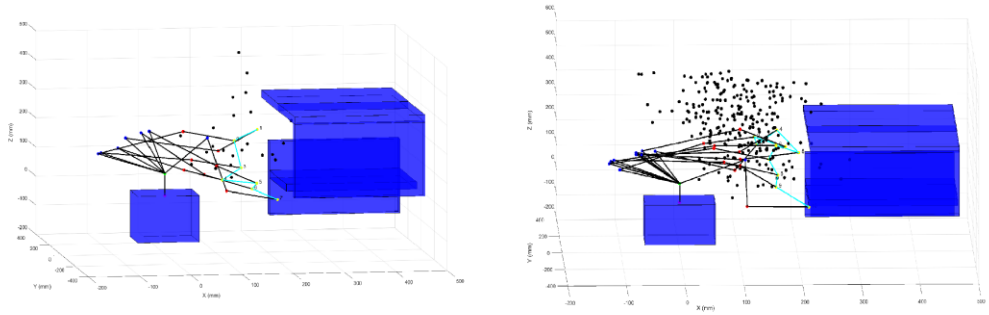

Figure 2-9 Two paths of test 1

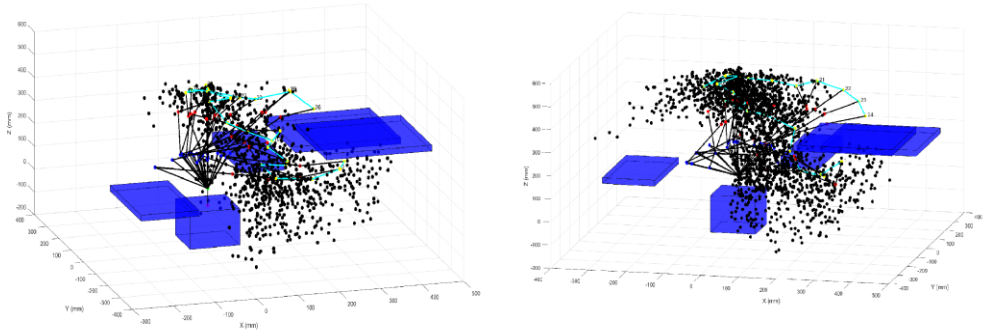

Figure 2-9 Two paths of test 5

Analysis

3.1 The environments our planner prefers

Our planner works well when the free space between the start and end position is broad, where there are little obstacles. Each point along the path in the workspace has a lot of feasible configurations which won't collided with the obstacles.

Our planner works bad when the free space between the start and end position is narrow, where there are lots of obstacles. Each point along the path in the workspace has a few feasible configurations which won't collide with the obstacles.

The distance between the start and end position is not directly relative to the planner's preference. A task with short distance and the start and end position is near the obstacle is much harder than the task with long distance but the start and end position is far from the obstacle.

When the start and end position require singular configurations, the task also become hard for our planner because the node is sampled at random and singular configurations would have less possibilities to be sampled around compared to multiple configurations.

3.2 Difference between expected and experimental performance

(1) When the path is feasible in the simulated map, it may not work in the physical map

As what is stated in Lab1 and Lab2, the robot is not an exact rigid body, and there are some system errors (i.e. error of the servo and the existence of gravity) and random errors (i.e. the measurement error) which affect the accuracy of physical robot. Thus, it may not reach the exact simulated points and may collide with obstacles if some points on the path are very close to obstacles, especially when the point is a little bit above the obstacles.
Because we use linear interpolation to travel between the way points, and linear interpolation only consider the position of the way points thus may need a huge acceleration in start and end. Because our robot is actually flexible, the robot may not stop and the exact position due to the existence of inertia, and may cause vibrations which affect the accuracy and collide with the obstacles.
The map constructed in the simulation map is not exactly the same to the real one, we have to modify them based on our measurement and there is always error in measurement.

(2) When there is a path in the physical map, it may not be found by our planner

The geometry representation of the robot is relatively rough, and we try to be safe as we inflating the obstacle boxes with a relatively big value. So there might be some path for a given task in the physical map, but our planner cannot find.

3.3 Improvement we would like to make

First, we would like to make compensation for the errors in physical experiment, as what is stated in our Lab2 report.

Second, we would like to make our trajectory smooth by using polynomial interpolations between the waypoints.

Third, we would like to improve the efficiency and performance of our planner. It could be done in various ways. We would like to improve the sample efficiency by sampling the node with bias in the configuration space. We would like to add the reward of any nodes to find the more direct path.

Appendix 1 Code Overview

1. *findpath.m*	find path
2. *calculateFK_sol.m*	predefined compute forward kinematics
3. *detectcollision.p*	predefined detect collision function
4. *getq.m*	get configuration from position
5. *IK_lynx_sol.m*	predefined compute inverse kinematics
6. *loadmap.p*	predefined lynx function
7. *plotmap.p*	predefined lynx function
8. *lynxInitializeHardware.m*	predefined lynx function
9. *lynxServo.m*	predefined lynx function
10. *lynxServoSim.m*	predefined lynx function
11. *lynxStart.m*	predefined lynx function
12. *lynxVelocityPhysical.m*	predefined lynx function
13. *lynxServoPhysical.m*	predefined lynx function
14. *plotLynx.m*	plot simulated configuration
15. *physicalImplement.m*	run the result of findpath in physical map
16. *runme.m*	run the code to call findpath.m
17. *findpath1.m*	findpath.m without plotting (for time counting)
18. *time.m*	count the time of findpath1

Appendix 2 Video Overview

1 https://youtu.be/tUrJ066Ic8k	test 1
2 https://youtu.be/JOYOIkAtk-w	test 2
3 https://youtu.be/bYjC6-OBAuM	test 3-1
4 https://youtu.be/PFqipa7mFDo	test 3-2
5 https://youtu.be/piN-5Dir-gI	test 3-3
6 https://youtu.be/w6JZppDhtXk	test 4