People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of the Requirements for the Degree of

MASTER

In **Control**

Option: **Control**

Title:

Kinodynamic Planning For an omnidirectional mobile robot

Presented by:

HALITIM Hiba

Supervisor:

Mr. GUERNANE Reda

Registration Number:……/2020

INTRODUCTION

Motion planning is the field of computer science that aims at developing algorithmic techniques allowing the automatic computation of trajectories for a mechanical system. The nature of such a system will vary according to the fields of application. The field of application of this work being robotics, the system is here a robot. We define a *robot* as mechanical agent controlled by a computer program. The modern meaning of the word seems to come from the English translation of the 1920 play *R.U.R.* (Rossum's Universal Robots), by Karel Capek, from Czech, *robotnik* (slave) derived from *robota* (forced labor). As for the term *robotics* (the science of robots), it was first used by Isaac Asimov in the 1941 short story *Liar!* The main feature of a robot is its ability to interact with its environment through motion. Being able to efficiently plan its movements is therefore a fundamental component of any robotic system.

More specifically we will focus here on holomonic robotics, which means we will deal with robots with no differential constraints.

The classic motion planning problem consists in computing a series of motions that brings the system from a given initial configuration to a desired final configuration without generating collisions with its environment, most of the time known in advance. Usual methods typically explore the system's configuration space regardless of its dynamics. By construction the thrust force that allows a quadrotor to fly is tangential to its attitude which implies that not every motion can be performed. This could, for example, be compared to the fact that a car cannot move sideways. Furthermore, the magnitude of this thrust force is limited by the physical capabilities of the engines operating the propellers. Therefore the same applies to the linear acceleration of the robot. For all these reasons, not only position and orientation must be planned, higher derivatives must be planned also if the motion is to be executed. When this is the case we talk of *kinodynamic motion planning*.

In this project we will simulate a three-wheeled omnidirectional robot and use it to demonstrate the task of motion planning under its kinematic and dynamic constraints.

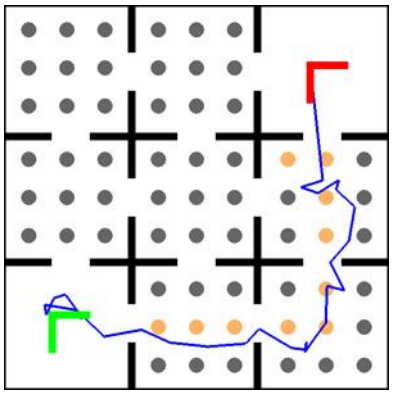
This report is organized as follows:

Chapter1: introduces the motion planning problem with some basic concepts and proposed solutions.

Chapter 2: concerns the kinodynamic planning using the basic RRT and anytime motion planning using the RRT* Algorithm.

Chapter 3: presents the kinematic and the dynamic models of the omnidirectional robot.

Chapter 4: This chapter illustrates the proposed RRT-Algorithms and the simulation with MATLAB.



CHAPTER 1

INTRODUCTION TO MOTION PLANNING

1.1 Introduction :

Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles in the environment and satisfies the dynamic constraints, such as joint limits or torque limits. Motion planning is one of the most active subfields of robotics, and it is the subject of entire books. This chapter begins with a brief overview of motion planning. This is followed by foundational material including configuration space obstacles and graph search. We conclude with summaries of several different planning methods.

1.2 Overview of Motion Planning:

A key concept in motion planning is configuration space, or C-space for short. Every point in the C-space C corresponds to a unique configuration q of the robot, and every configuration of the robot can be represented as a point in C-space. For example, the configuration of a robot arm with n joints can be represented as a list of n joint positions, $q = (\theta_1, \ldots., \theta_n)$. The free C-space C_{free} consists of the configurations where the robot neither penetrates an obstacle nor violates a joint limit.

The control inputs available to drive the robot are written as an m-vector $u \in U \in R^m$.

The notation $q(x)$ indicates the configuration q corresponding to the state, and

$$X_{free} = \{x \mid q(x) \in C_{free}\}$$

The equations of motion of the robot are written as:

$$\dot{x} = f(x, u)$$

1.2.1 Types of Motion Planning Problems :

With the definitions above, a fairly broad specification of the motion planning problem is the following:

Given an initial state $x(0) = x_{start}$ and a desired final state x_{goal} , find a time T and a set of controls $u: [0, T] \rightarrow U$ such that $x(T) = x_{goal}$ and $q(x(t)) \in C_{free}$ for all $t \in [0, T]$. There are many variations of the basic problem; some are discussed below.

1.2.1.1 Path planning versus motion planning :

The path planning problem is a subproblem of the general motion planning problem. Path planning is the purely geometric problem of finding a collision-free path $q(s); s \in [0,1]$, from a start configuration $q(0) = q_{start}$ to a goal configuration $q(1) = q_{goal}$, without concern for the dynamics, the duration of motion, or constraints on the motion or on the control inputs. It is assumed that the path returned by the path planner can be time scaled to create a feasible trajectory. This problem is sometimes called the piano mover's problem, emphasizing the focus on the geometry of cluttered spaces. [1]

1.2.1.2 Holomonic versus Nonholomonic :

Control inputs $m = n$ versus $m < n$. If there are fewer control inputs m than degrees of freedom n, then the robot is incapable of following many paths, even if they are collision-free.

For example, a car has n = 3 (the position and orientation of the chassis in the plane) but m=2 (forward -backward motion and steering); it cannot slide directly sideways into a parking space; we call the car a nonholomonic robot, while the omnidirectional robot is a holomonic system, it can slide sideways.

1.2.1.3 Online versus offline :

A motion planning problem requiring an immediate result, perhaps because obstacles appear, disappear, or move unpredictably, calls for a fast, online, planner. If the environment is static then a slower offline planner may suffice.

1.2.1.4 Optimal versus satisficing:

In addition to reaching the goal state, we might want the motion planner to minimize (or approximately minimize) a cost J.

1.2.1.5 Exact versus approximate :

We may be satisfied with a final state $x(T)$ that is sufficiently close to x_{goal} ,

e.g $\|x(T) - x_{goal}\| < \epsilon$.

1.2.1.6 With or without obstacles:

The motion planning problem can be challenging even in the absence of obstacles, particularly if m < n or optimality is desired.

1.2.2 Properties of Motion Planners :

Planners must conform to the properties of the motion planning problem as outlined above. In addition, planners can be distinguished by the following properties:

1.2.2.1 Multiple-query versus single-query planning :

If the robot is being asked to solve a number of motion planning problems in an unchanging environment, it may be worth spending the time building a data structure that accurately representsC_{free}. This data structure can then be searched to solve multiple planning queries efficiently. Single-query planners solve each new problem from scratch.

1.2.2.2 ‘’Anytime’’ planning:

An anytime planner is one that continues to look for better solutions after a first solution is found. The planner can be stopped at any time, for example when a specified time limit has passed, and the best solution returned.

1.2.2.3 Completeness:

A motion planner is said to be **complete** if it is guaranteed to find a solution in a finite time if one exists, and to report failure if there is no feasible motion plan. A weaker concept is **resolution completeness**. A planner is resolution complete if it is guaranteed to find a solution if one exists at the resolution of a discretized representation of the problem, such as the resolution of a grid representation of C_{free}. Finally, a planner is **probabilistically complete** if the probability of finding a solution, if one exists, tends to 1 as the planning time goes to infinity.

1.2.2.4 Computational complexity :

The computational complexity refers to characterizations of the amount of time the planner takes to run or the amount of memory it requires. These are measured in terms of the description of the planning problem, such as the dimension of the C-space or the number of vertices in the representation of the robot and obstacles. For example, the time for a planner to run may be exponential in *n*, the dimension of the C-space. The computational complexity may be expressed in terms of the average case or the worst case. Some planning algorithms lend themselves easily to computational complexity analysis, while others do not. [1]

1.3 Foundations:

Before discussing motion planning algorithms, we establish concepts used in many of them: configuration space obstacles, collision detection, graphs, and graph search.

1.3.1 Configuration Space Obstacles :

The workspace obstacles partition the configuration space C into two sets, the free space C_{free} and the obstacle space Cobs, where $C = C_{free} \cup C_{obs}$. Joint limits are treated as obstacles in the configuration space.

1.3.2 Distance to Obstacles and Collision Detection :

Given a C-obstacle B and a configuration q, let $d(q, B)$ be the distance between the robot and the obstacle, where:

$d(q, B) > 0$ (No contact with the obstacle)

$d(q, B) = 0$ (Contact)

$d(q, B) < 0$ (Penetration)

The distance could be defined as the Euclidean distance between the two closest points of the robot and the obstacle, respectively.

A distance-measurement algorithm is one that determines $d(q, B)$. A Collision-detection routine determines whether $d(q, B_i) \leq 0$ for any C-obstacle B_i. A collision-detection routine returns a binary result and may or may not utilize a distance-measurement algorithm at its core.

1.3.3 Graphs and Trees :

Many motion planners explicitly or implicitly represent the C-space or state space as a **graph**. A graph consists of a collection of nodes $\boldsymbol{N}$ and a collection of edges $\boldsymbol{\varepsilon}$, where each edge e connects two nodes. In motion planning, a node typically represents a configuration or state while an edge between nodes $n_1$and n_2 indicates the ability to move from n_1 to n_2 without penetrating an obstacle or violating other constraints.

A graph can be either **directed** or **undirected**. In an undirected graph, each edge is bidirectional: if the robot can travel from n_1 to n_2 then it can also travel from n_2 to n_1. In a directed graph, or **digraph** for short, each edge allows travel in only one direction. The same two nodes can have two edges between them, allowing travel in opposite directions.

Graphs can also be **weighted** or **unweighted.** In a weighted graph, each edge has a positive cost associated with traversing it. In an unweighted graph each edge has the same cost.

A tree is a digraph in which there are no cycles and each node has at most one parent node. A tree has one root node with no parents and a number of leaf nodes with no child. A digraph, undirected graph, and tree are illustrated in Figure 1.1.

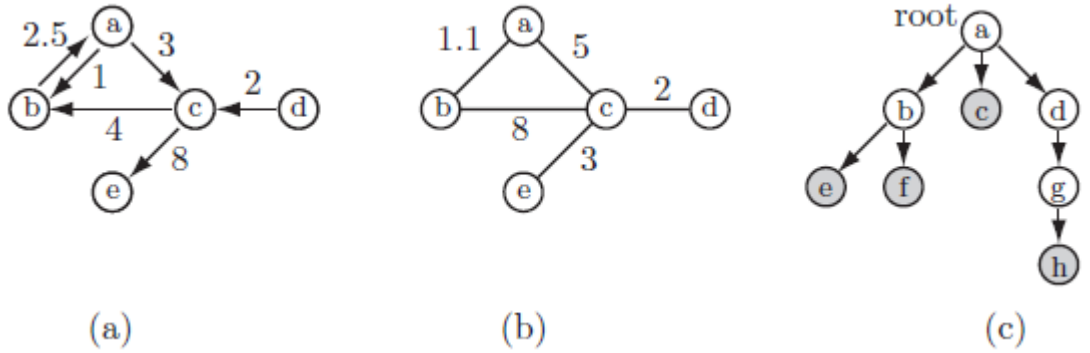


Figure 1.1: (a) A weighted dgraph (b) A weighted undirected graph (c) A tree (The leaves are shaded grey) [1]

1.3.4 Graph search:

Once the free space is represented as a graph, a motion plan can be found by searching the graph for a path from the start to the goal. One of the most powerful and popular graph search algorithms is A^* (pronounced ''A star'') search.

1.3.4.1 A^* search :

A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals. What A* Search Algorithm does is that at each step it picks the node according to a value '**f**' which is a parameter equal to the sum of two other parameters '**g**' and '**h**'. At each step it picks the node having the lowest '**f**', and processes that node/cell. [10]

1.3.4.2 Other search methods :

- **Dijkstra's algorithm.** If the heuristic cost-to-go is always estimated as zero then A^* always explores from the OPEN node that has been reached with minimum past cost. This variant is called Dijkstra's algorithm, which preceded A^* historically. Dijkstra's algorithm is also guaranteed to find a minimum-cost path but on many problems it runs more slowly than A^* owing to the lack of a heuristic look-ahead function to help guide the search.
- **Breadth-first search.** If each edge in ε has the same cost, Dijkstra's algorithm reduces to breadth-first search. All nodes one edge away from the start node are considered first, then all nodes two edges away, etc. The first solution found is therefore a minimum-cost path.

- **Suboptimal A^* search.** If the heuristic cost-to-go is overestimated by multiplying the optimistic heuristic by a constant factor $\alpha > 1$, the A^* search will be biased to explore from nodes closer to the goal rather than nodes with a low past cost. This may cause a solution to be found more quickly but, unlike the case of an optimistic cost-to-go heuristic, the solution will not be guaranteed to be optimal. One possibility is to run A^* with an inated cost-to-go to find an initial solution, then rerun the search with progressively smaller values of α until the time allotted for the search has expired or a solution is found with $\alpha = 1$.

1.4 Motion Planning Methods:

There is no single planner applicable to all motion planning problems. Below is an overview of some of the many motion planners available.

1.4.1 Complete methods :

Complete path planners rely on an exact representation of the free C-space C_{free}. These techniques tend to be mathematically and algorithmically sophisticated, and impractical for many real systems, so we do not delve into them in detail.

1.4.2 Grid methods:

These methods discretize C_{free} into a grid and search the grid for a motion from q_{start} to a grid point in the goal region. These methods are relatively easy to implement and can return optimal solutions but, for a fixed resolution, the memory required to store the grid, and the time to search it, grow exponentially with the number of dimensions of the space. This limits the approach to low-dimensional problems.

1.4.3 Virtual potential fields:

Virtual potential fields create forces on the robot that pull it toward the goal and push it away from obstacles. The approach is relatively easy to implement, even for high-degree-of freedom systems, and fast to evaluate, often allowing online implementation. The drawback is local minima in the potential function: the robot may get stuck in configurations where the attractive and repulsive forces cancel but the robot is not at the goal state.

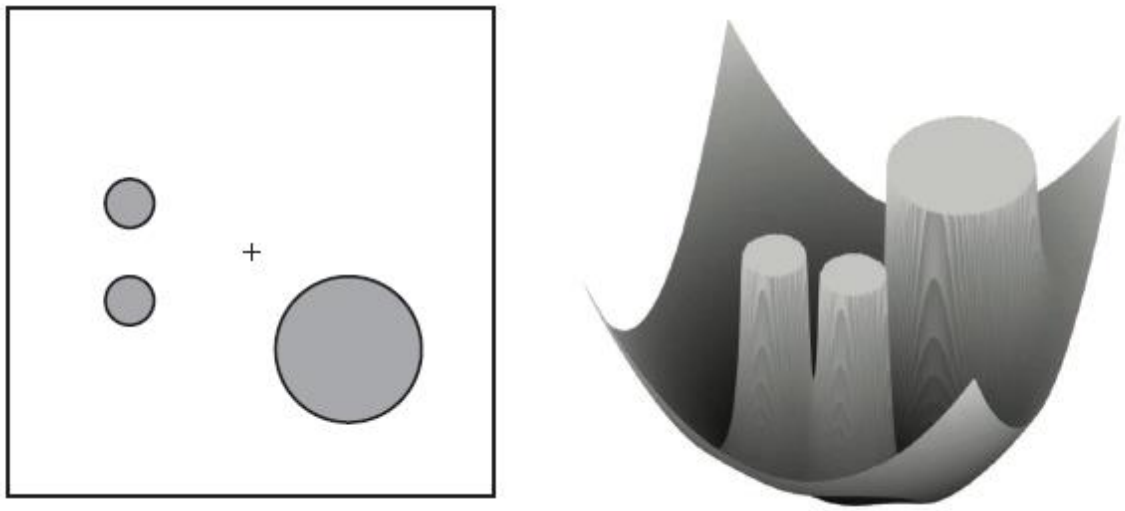


Figure 1.2: (left) Three obstacles and a goal point, marked with a +, in R^2 (right) the potential function

1.4.4 Sampling-Based methods :

A generic sampling method relies on a random or deterministic function to choose a sample from the C-space or state space; a function to evaluate whether the sample is in X_{free} ; a function to determine the ‘’closest’’ previous free-space sample; and a local planner to try to connect to, or move toward, the new sample from the previous sample. This process builds up a graph or tree representing feasible motions of the robot. Sampling methods are easy to implement, tend to be probabilistically complete, and can even solve high-degree-of freedom motion planning problems. The solutions tend to be satisficing, not optimal, and it can be difficult to characterize the computational complexity.

Two major classes of sampling methods are rapidly exploring random trees (RRTs) and probabilistic roadmaps (PRMs). The former uses a tree representation for single-query planning in either C-space or state space, while PRMs are primarily C-space planners that create a roadmap graph for multiple-query planning.

1.4.4.1 *Rapidly-exploring Random Tree (RRT):*

RRTs were introduced as a single-query planning algorithm that efficiently covers the space between x_{init} and x_{goal}. Rapidly-exploring Random Trees is an incremental method to quickly explore the whole state space. The principle is simple. Starting from an arbitrary state in the state space, the algorithm selects a random state within the free portion of the space. Subsequently, it searches for the closest state in the current tree using an arbitrary metric.

Furthermore, the algorithm uses a local planner to generate a new node toward the random state. If the path between the nearest node and the new node is feasible (do not collide with obstacles), the algorithm will extend the tree between them. Otherwise, the Algorithm will discard the new state. Figure 1.3 shows an RRT evolution. [2]

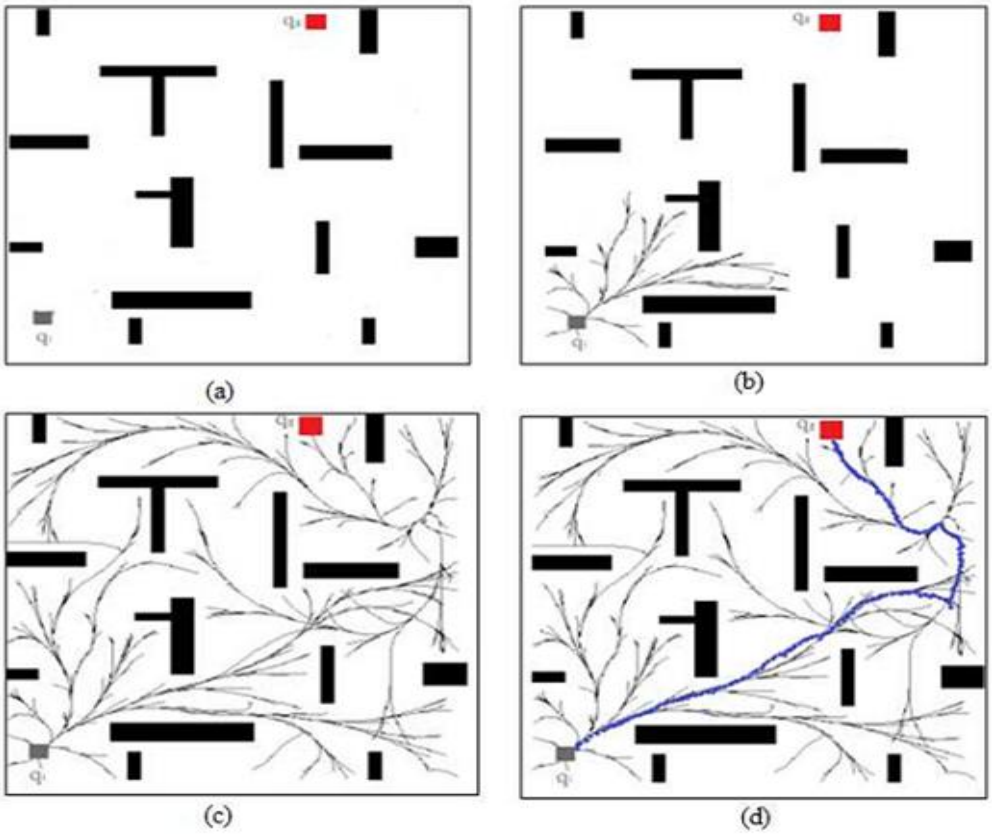


Figure 1.3: RRT evolution [2]

1.4.4.2 Probabilistic Road Map (PRM):

This method is divided into two different phases:

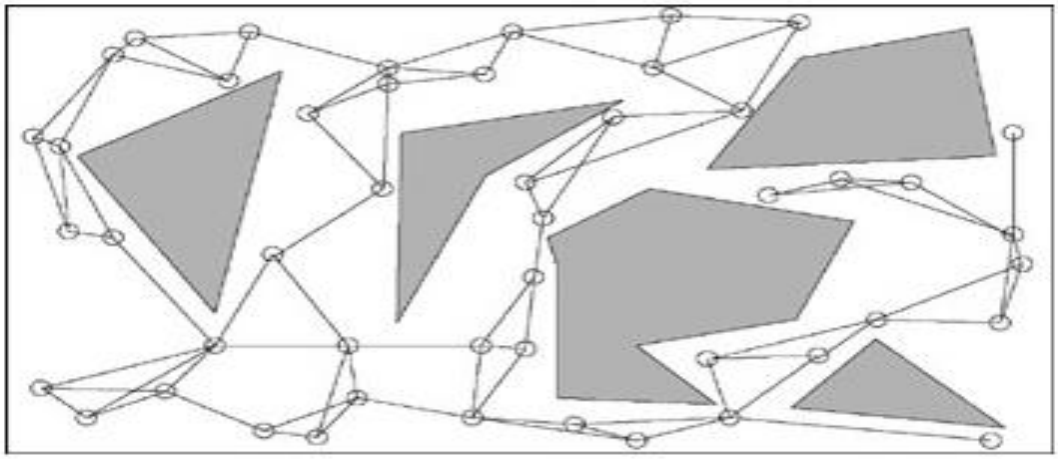
a) Preprocessing Phase:

The preprocessing phase generates a roadmap based on the configuration space (C) and the obstacles. Initially, the graph $G = (V, E)$ is empty - the initial and the goal state are not considered in this phase -. Then, repeatedly, a configuration is sampled from C**.** For the moment, assume that the sampling is done according to a uniform random distribution on ***C.*** If the configuration is collision-free, it is added to the roadmap. The process is repeated until ***n*** collision-free configurations have been sampled. For every node $q \in V$, a set N_q of k closest neighbors to the configuration q according to some metric *distance* is chosen from V.

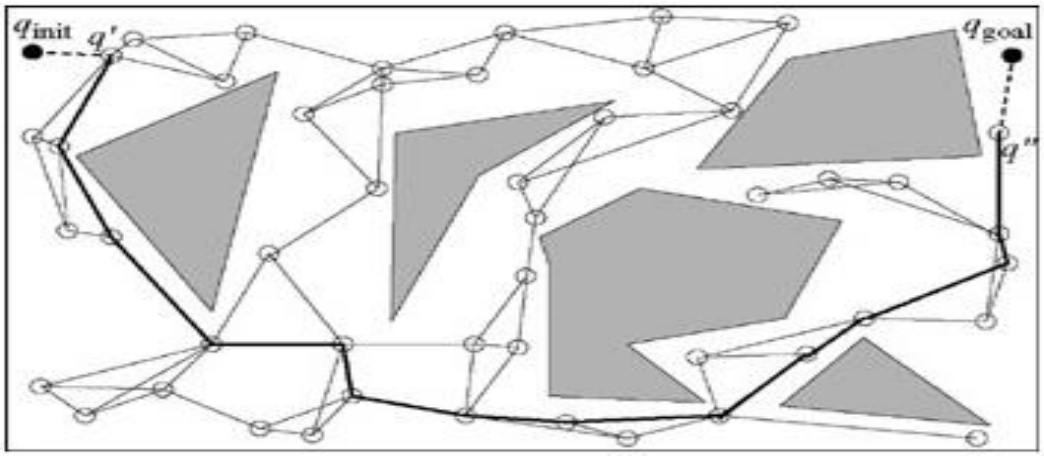
The local planner (Δ) is called to connect q to each node $q' \in N_q$. Whenever Δ succeeds in computing a feasible path between q and q', the edge (q, q') is added to the roadmap. Figure 1.4.a shows a roadmap constructed for a point robot in a two-dimensional Euclidean workspace, where Δ is a straight-line planner and the gray areas are obstacles. The empty circles correspond to the nodes of the roadmap. The straight lines between circles correspond to edges. The number of k closest neighbors for the construction of the roadmap is three. The degree of a node can be greater than three since it may be included in the closest neighbor list of many nodes.

b) Query Phase:

If a roadmap was created, queries can be executed which ask for a path between the initial configuration q_{init} and the goal configuration q_{goal}. Since the roadmap uses only random configurations, it is nearly impossible that any random configuration q is equal to q_{init} or q_{goal}. Hence, a path between q_{init} and the closest state in the roadmap $q'_{\text{init}} \in V$ and further between q_{goal} and $q'_{\text{goal}} \in V$ has to be computed. If these paths were found, a graph search algorithm can be used to find a feasible path. The Figure 1.4.b shows how to solve a query with the roadmap. [2]



(a)

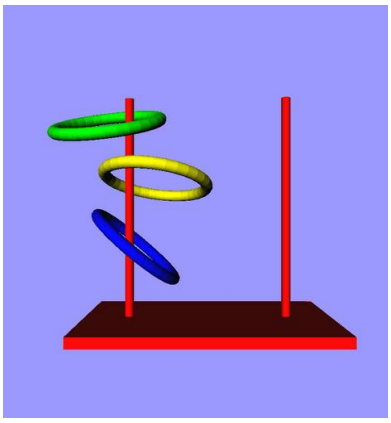


(b)

Figure1.4: PRM. (a) The Preprocessing Phase. (b) The query phase.

1.5 Conclusion:

In this chapter we have presented the importance of motion planning, then we have explained different types of motion planning problems and the properties of motion planners along with the different motion planning methods.



CHAPTER 2

KINODYNAMIC PLANNING

2.1 Introduction:

There is a strong need for a general-purpose, efficient planning technique that determines control inputs to drive a robot from an initial configuration and velocity to a goal configuration and velocity while obeying physically-based dynamical models and avoiding obstacles in the robot's environment. In other words, a fundamental task is to design a feasible open-loop trajectory that satisfies both global obstacle constraints and local dynamical constraints. We use the word kinodynamic planning to refer to such problems. The solution introduced in this work is a direct trajectory planning method (there is no path planning stage).

2.2 Problem Formulation: Path Planning in the State Space:

We formulate the kinodynamic planning problem as path planning in a state space that has dynamic constraints.

2.2.1 The robot state (X):

Let X denote the state space, in which the state $x \in X$ is defined as $x = (q, \dot{q})$; where each configuration $q \in C$ represents a transformation that is applied to a geometric model of the robot.

Figure 2.1 shows the state space representation of an omnidirectional robot [6]

$$x = (q, \dot{q}) = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$$

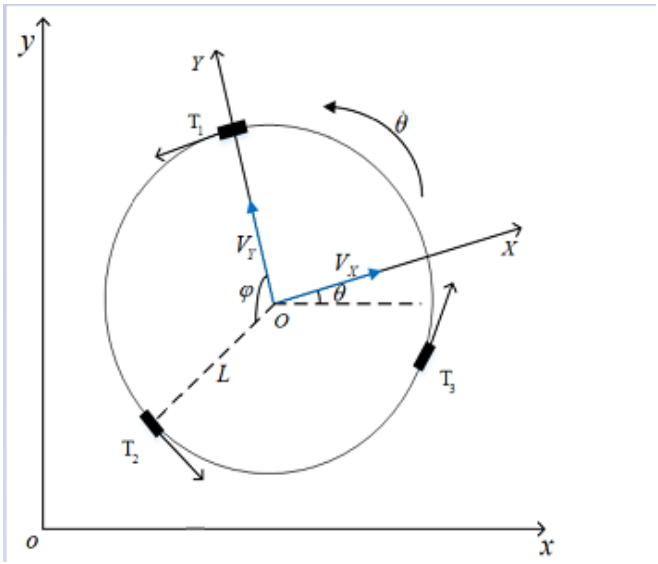


Figure 2.1: omnidirectional robot in state space

2.2.2 Dynamic constraints :

Working in state space allows the planner to incorporate dynamic constraints on path

Examples: maximum velocity, minimum turning radius

The constraints can be expressed as:

$$\dot{x} = f(x, u) \qquad (1)$$

In which $\in U$, and U is the set of allowable controls or inputs.

2.2.3 Obstacles in the state space:

The path planning problem involves finding a continuous path that maps into

$C_{free} = C - C_{obs}$.

For planning in X ;

$X_{free} = X - (X_{obs} + X_{ric})$

Where X_{ric} is the region of Imminent Collision Where the robot's actuators cannot prevent a collision with an obstacle.

2.2.4 A solution trajectory:

Given the holomonic robot A with dynamic constraints expressed by equation (1), find an input $u : [0, T] \rightarrow U$ which results in a collision-free trajectory that starts at x_{init} and ends at x_{goal} .

2.3 A planner based on Rapidly-Exploring Random tree:

2.3.1 The basic RRT Algorithm:

The RRT algorithm searches for a collision-free motion from an initial state x_{start} to a goal set X_{goal}. The basic RRT grows a single tree from x_{start} as outlined in Algorithm 2.1.

Algorithm 2.1 *The basic RRT algorithm*

1: initialize search tree T with x_{start}

2: ***while*** *T is less than the maximum tree size* ***do***

3: x_{samp} *sample from X*

4: $x_{nearest}$ *nearest node in T to* x_{samp}

5: employ a local planner to find a motion from $x_{nearest}$ *to* x_{new} *in the direction of* x_{samp}

6: ***if*** *the motion is collision-free* ***then***

7: add x_{new} *to T with an edge from* $x_{nearest}$ *to* x_{new}

8: ***if*** x_{new} *is in* X_{goal} ***then***

9: ***return SUCCESS*** *and the motion to* x_{new}

10: ***end if***

11: ***end if***

12: ***end while***

13: ***return FAILURE***

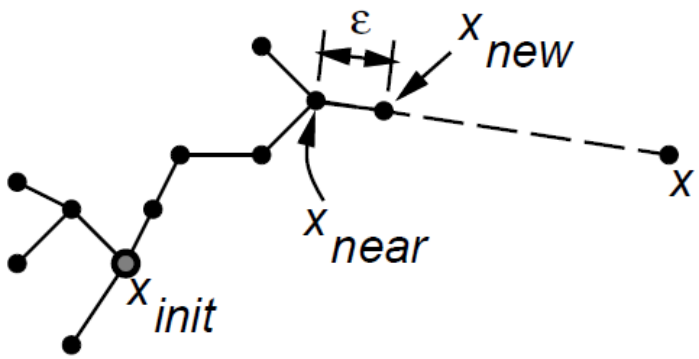


Figure 2.2: The EXTEND operation of the tree

In a typical implementation for a kinematic problem (where x is simply q), the sampler in line 3 chooses x_{samp} randomly from an almost-uniform distribution over X, with a slight bias toward states in X. The closest node $x_{nearest}$ in the search tree T (line 4) is the node minimizing the Euclidean distance to x_{samp}. The state x_{new} (line 5) is chosen as the state with small distance d from $x_{neasrest}$ on the straight line to x_{samp}. Because d is small, a very simple local planner, e.g., one that returns a straight-line motion, will often find a motion connecting $x_{neasrest}$ to x_{new}. If the motion is collision-free, the new state $x_{new}$is added to the search tree T. The net effect is that the nearly uniformly distributed samples ‘’pull’’ the tree toward them, causing the tree to rapidly explore X_{free}. An example of the effect of this pulling action on exploration is shown in Figure 2.3.

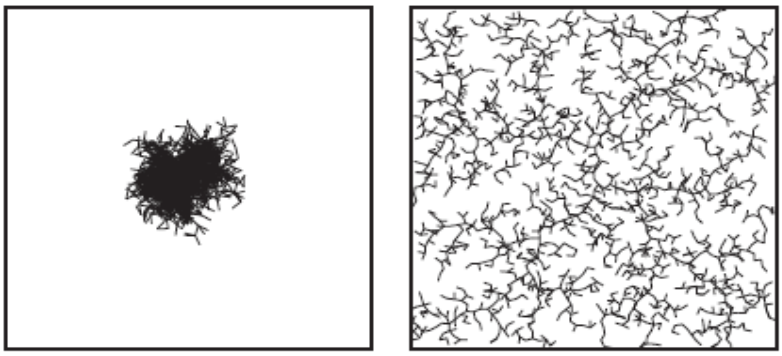


Figure 2.3: (Left) A Naive random tree (Right) A Rapidly-Exploring Random tree. Both trees have 2000 nodes.

The basic algorithm leaves the programmer with many choices: how to sample from X (line 3), how to define the ‘’nearest’’ node in T (line 4), and how to plan the motion to make progress toward x_{samp} (line 5).

2.3.1.1 Line 3: The Sampler

For dynamic systems, a uniform distribution over the state space can be defined as the cross product of a uniform distribution over C-space and a uniform distribution over a bounded velocity set.

2.3.1.2 Line 4: Defining the Nearest Node

Finding the ‘’nearest’’ node depends on a definition of distance on X. For an unconstrained kinematic robot on$C = R^n$, a natural choice for the distance between two points is simply the Euclidean distance. For other spaces, the choice is less obvious.

The closest node $x_{nearest}$ should perhaps be defined as the one that can reach x_{samp} the fastest.

A simple choice of a distance measure from x to x_{samp} is the weighted sum of the distances along the different components of $x_{samp} - x$. If more is known about the set of states that the robot can reach from a state x in limited time, this information can be used in determining the nearest node.

2.3.1.3 Line 5: The Local Planner

The job of the local planner is to find a motion from $x_{nearest}$ to some point x_{new} which is closer to x_{samp}. The planner should be simple and it should run quickly.

Discretized controls planner. For dynamic systems, the controls can be discretized into a discrete set $\{u_1, u_2, u_3, \dots\}$, as in the grid methods with motion constraints. Each control is integrated from $x_{nearest}$ for a fixed time Δt using $\dot{x} = f(x, u)$. Among the new states reached without collision, the state that is closest to x_{samp} is chosen as x_{new} .

2.3.2 Bidirectional RRT:

The bidirectional RRT grows two trees: one ''forward'' from x_{start} and one ''backward'' from x_{goal} . The algorithm alternates between growing the forward tree and growing the backward tree, and every so often it attempts to connect the two trees by choosing x_{samp} from the other tree. The advantage of this approach is that a single goal state x_{goal} can be reached exactly, rather than just a goal set X_{goal} . Another advantage is that, in many environments, the two trees are likely to find each other much more quickly than a single ''forward'' tree will find a goal set. [3]

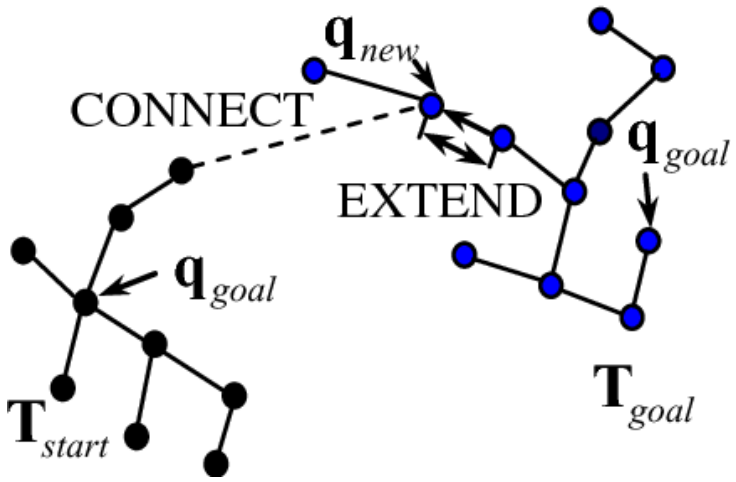


Figure 2.4: Bidirectional RRT Operation

2.3.2.1 Basic RRT-Connect:

RRT_CONNECT *(x_{init}, x_{goal}) {*

Ta.init(x_{init}); Tb.init(x_{goal});

for *k= 1 to K do*

x_{rand} = RANDOM_CONFIG();

if not *(EXTEND(Ta, x_{rand}) = Trapped) then*

if *(EXTEND(Tb, x_{new}) = Reached) then*

Return *PATH(Ta, Tb);*

SWAP*(Ta, Tb);*

Return *Failure;*

}

2.3.3 Anytime Motion Planning using the RRT*

The Rapidly-exploring Random Tree (RRT) algorithm, based on incremental sampling, efficiently computes motion plans. Although the RRT algorithm quickly produces candidate feasible solutions, it tends to converge to a solution that is far from optimal. Practical applications favor "anytime" algorithms that quickly identify an initial feasible plan, then, given more computation time available during plan execution, improve the plan toward an optimal solution. This section describes an anytime algorithm based on the RRT* which (like the RRT) finds an initial feasible solution quickly, but (unlike the RRT) almost surely converges to an optimal solution. It is asymptotically optimal. We present two key extensions to the RRT*, committed trajectories and branch-and-bound tree adaptation, that together enable the algorithm to make more efficient use of computation time online, resulting in an anytime algorithm for real-time implementation.

The RRT* algorithm is a variation on the single-tree RRT that continually rewires the search tree to ensure that it always encodes the shortest path from x_{start} to each node in the tree.

2.3.3.1 RRT* Algorithm :

The RRT* essentially behaves identically to the RRT. The RRT* starts with an empty tree and adds a single node corresponding to the initial state. It then builds and refines the tree through a set of N iterations (lines 3–11).

Like the RRT, the RRT* incrementally builds the tree by sampling a random state z_{rand} from the obstacle-free space (line 4) and solving for a trajectory x_{new} that extends the closest node in the tree $z_{nearest}$ toward the sample (lines 5–6). If this trajectory does not collide with obstacles (line 7), the standard RRT inserts the new node z_{new} into the tree with $z_{nearest}$ as its parent and continues with the next iteration. It is here that the operation of the RRT* differs. Rather than choosing the nearest node as the parent, the RRT* considers all nodes in a neighborhood of z_{new} (line 8) and evaluates the cost of choosing each as the parent. This process (Alg. 2) evaluates the total cost as the additive combination of the cost associated with reaching the potential parent node and the cost of the trajectory to z_{new}. The

node that yields the lowest cost becomes the parent as the new node is added to the tree (Alg. 1, line 10).

The ReWire procedure described in Alg. 3 then checks each node z_{near} in the vicinity of z_{new} to see whether reaching z_{near} via z_{new} would achieve lower cost than doing so view its current parent (Alg. 3, line 3). When this connection reduces the total cost associated with z_{near}, the algorithm modifies ("rewires") the tree to make z_{new} the parent of z_{near} (line 4). The RRT*then continues with the next iteration.

Algorithm 1: $\mathcal{T} = (V, E) \leftarrow \text{RRT}^{\star}(z_{\text{init}})$

```
1  T ← InitializeTree();
2  T ← InsertNode(∅, z_init, T);
3  for i = 1 to i = N do
4      z_rand ← Sample(i);
5      z_nearest ← Nearest(T, z_rand);
6      (x_new, u_new, T_new) ← Steer(z_nearest, z_rand);
7      if ObstacleFree(x_new) then
8          Z_near ← Near(T, z_new, |V|);
9          z_min ← ChooseParent(Z_near, z_nearest, z_new, x_new);
10         T ← InsertNode(z_min, z_new, T);
11         T ← ReWire(T, Z_near, z_min, z_new);
12 return T
```

Algorithm 2: $z_{\min} \leftarrow \texttt{ChooseParent}(Z_{\text{near}}, z_{\text{nearest}}, x_{\text{new}})$

$z_{\min} \leftarrow z_{\text{nearest}}$;
$c_{\min} \leftarrow \texttt{Cost}(z_{\text{nearest}}) + c(x_{\text{new}})$;
for $z_{near} \in Z_{near}$ **do**
    $(x', u', T') \leftarrow \texttt{Steer}(z_{\text{near}}, z_{\text{new}})$;
    if $\texttt{ObstacleFree}(x')$ *and* $x'(T') = z_{new}$ **then**
        $c' = \texttt{Cost}(z_{\text{near}}) + c(x')$;
        if $c' < \texttt{Cost}(z_{new})$ *and* $c' < c_{min}$ **then**
            $z_{\min} \leftarrow z_{\text{near}}$;
            $c_{\min} \leftarrow c'$;
return $z_{\min}$

Algorithm 3: $\mathcal{T} \leftarrow \texttt{ReWire}(\mathcal{T}, Z_{\text{near}}, z_{\min}, z_{\text{new}})$

for $z_{near} \in Z_{near} \setminus \{z_{min}\}$ **do**
    $(x', u', T') \leftarrow \texttt{Steer}(z_{\text{new}}, z_{\text{near}})$;
    if $\texttt{ObstacleFree}(x')$ *and* $x'(T') = z_{near}$ *and*
    $\texttt{Cost}(z_{new}) + c(x') < \texttt{Cost}(z_{near})$ **then**
        $\mathcal{T} \leftarrow \texttt{ReConnect}(z_{\text{new}}, z_{\text{near}}, \mathcal{T})$;
return $\mathcal{T}$

Unlike the RRT, the solution provided by RRT^* approaches the optimal solution as the number of sample nodes increases. Like the RRT, the RRT^* algorithm is probabilistically complete. Figure 2.5 demonstrates the rewiring behavior of RRT^* compared to that of RRT for a simple example in $C = R^2$.

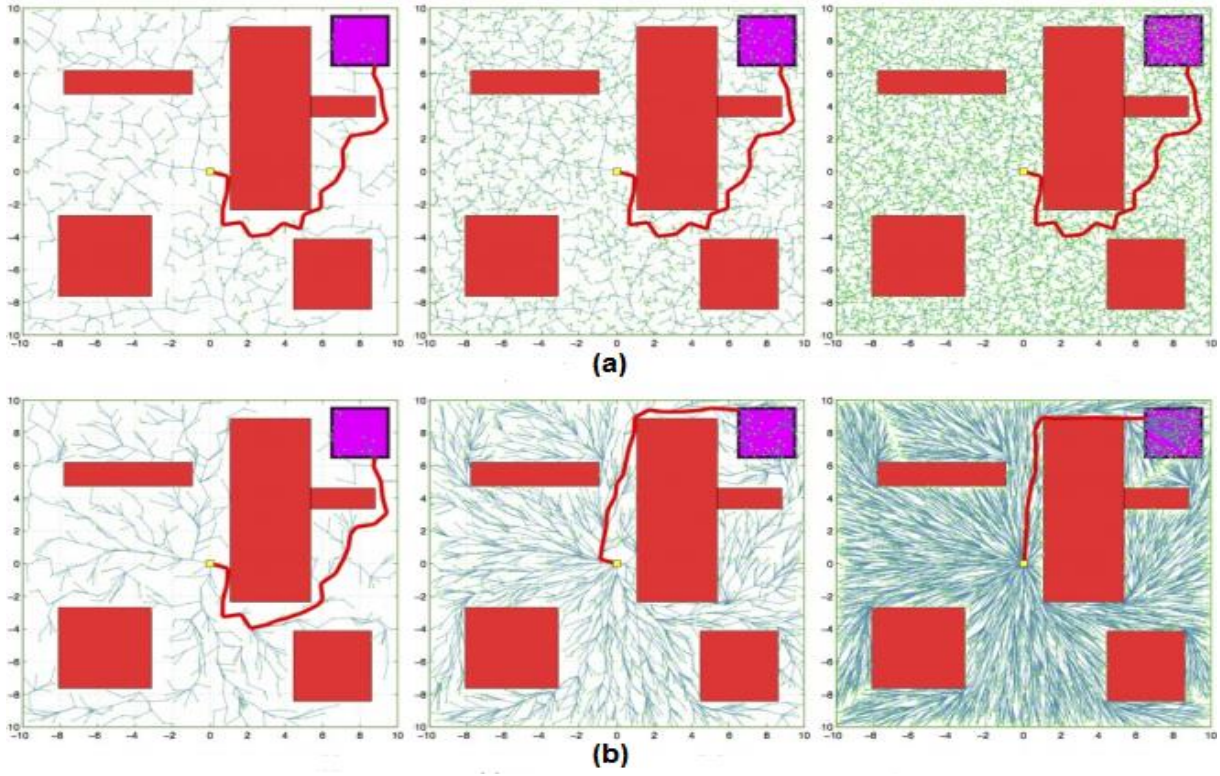
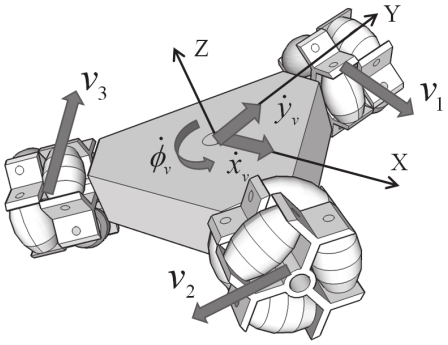


Figure2.5: RRT* VS Basic RRT. (a) Basic RRT. (b) RRT*.

2.4 Conclusion:

In this chapter, we have illustrated the Kinodynamic motion planning problem and presented the RRT algorithm as simple and practical algorithms used to solve the problem. We have presented also our implemented RRT^*algorithm that aims to improve the solution.



CHAPTER 3

KINODYNAMIC MODELLING OF THE ROBOT

3.1 Introduction :

This chapter provides a description of the kinematic and the dynamic models of the omnidirectional mobile robot, which are essential in the kinodynamic planning.

3.2 The omnidirectional wheeled mobile robot :

In recent decades, omnidirectional mobile robot (OMR) has attracted increasing attention and investigation from the research communities. One of advantages of OMR using omnidirectional wheels is that it does not have nonholonomic constraint which exists in differentially driven mobile robot. With the input of the rotating speed of each omnidirectional wheel, the mobile robot can easily move wherever the user wants. We present in this section two different models of the OMR; the first one is a simple kinematic model which is used in path planning problems. The second one is a dynamic model which we are going to use in our study. [4]

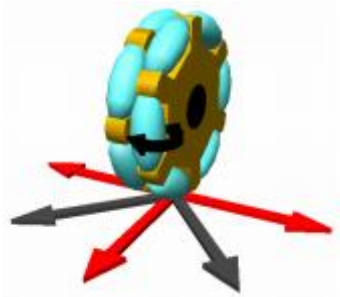


Figure 3.1: The Omni-directional wheel

3.3 Modelling:

Next we are going to describe the three-wheeled omnidirectional robot which has three omni-wheels, each is 120° apart.

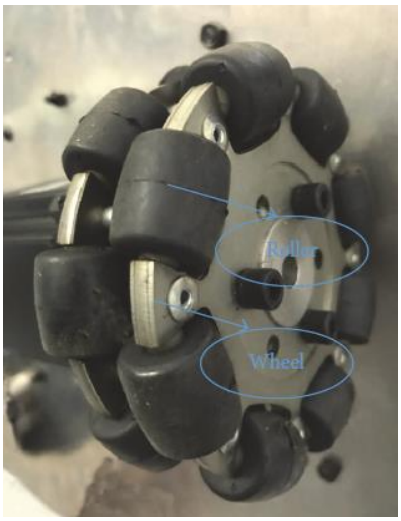


Figure 3.2: The structure of i^{th} omni-wheel [9]

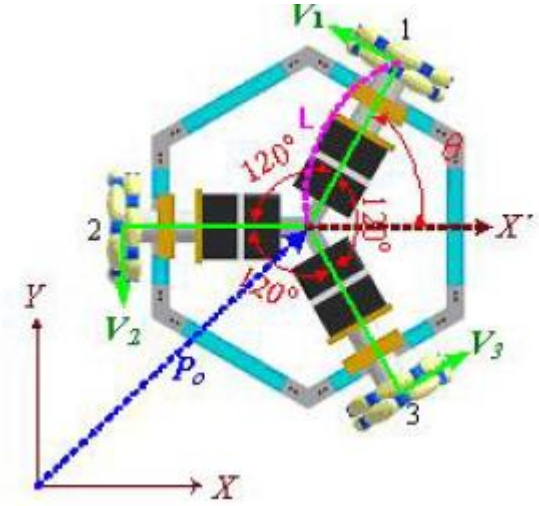


Figure 3.3: The geometry of the OMR [9]

3.3.1 The Robot kinematics:

This section aims to describe the kinematic model of this kind of robot. Figure 3.2 depicts its kinematic diagram that is used to find the kinematic model of the robot, where θ denotes the vehicle orientation. Before doing so, one considers the following rotation matrix. [8]

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Where θ is the rotation angle which is positive in the counterclockwise direction. As can be seen in Figure3.2, P_0 denotes the position of the center of mass with respect to the world frame, i.e., $P_0 = [x \quad y]^T$.The position $[x \quad y]^T$ of each wheel can be given with respect to the center of mass of the robot, i.e., for i=1,2,3.

$$P_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = R(\theta) \cdot L \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Where the parameter L is the distance from each wheel to the center of mass. Hence, we have the following three vectors:

$$P_1 = R(0) \cdot L \begin{bmatrix} 1 \\ 0 \end{bmatrix} = L \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad ; \quad P_2 = R\left(\tfrac{2\pi}{3}\right) \cdot L \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \tfrac{L}{2} \begin{bmatrix} -1 \\ \sqrt{3} \end{bmatrix} \quad ;$$

$$P_3 = R\left(\tfrac{4\pi}{3}\right) \cdot L \begin{bmatrix} 1 \\ 0 \end{bmatrix} = -\tfrac{L}{2} \begin{bmatrix} 1 \\ \sqrt{3} \end{bmatrix}$$

The normal unit vectors D_i of each wheel, representing the translational direction, are described as follows:

$$D_i = \frac{1}{L} R(\theta) P_i \quad ; \quad i = 1,2,3$$

Which yields:

$$D_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad D_2 = -\tfrac{1}{2} \begin{bmatrix} \sqrt{3} \\ 1 \end{bmatrix} \qquad D_2 = \tfrac{1}{2} \begin{bmatrix} \sqrt{3} \\ -1 \end{bmatrix}$$

The position and velocity of each wheel with respect to the world frame are then expressed by, for i=1, 2, 3

$$r_i = P_0 + R(\theta + \tfrac{2\pi}{3}(i-1))P_i \quad ; \quad v_i = \dot{P}_0 + \dot{R}(\theta + \tfrac{2\pi}{3}(i-1))P_i$$

Then the translational velocity V_i of each wheel is obtained from :

$$V_i = v_i^T \left(R\left(\theta + \frac{2\pi}{3}(i-1)\right) D_i\right)$$

Thus, one obtains:

$$V_1 = -\sin\theta \cdot \dot{x} + \cos\theta \cdot \dot{y} + L \cdot \dot{\theta}$$

$$V_2 = -\sin(\frac{\pi}{3} - \theta) \cdot \dot{x} - \cos(\frac{\pi}{3} - \theta) \cdot \dot{y} + L \cdot \dot{\theta}$$

$$V_3 = \sin(\frac{\pi}{3} + \theta) \cdot \dot{x} + \cos(\frac{\pi}{3} + \theta) \cdot \dot{y} + L \cdot \dot{\theta}$$

Which can be rewritten in vector-matrix form:

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} R\omega_1 \\ R\omega_2 \\ R\omega_3 \end{bmatrix} = P(\theta) \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$

Where R is the wheel's radius and:

$$P(\theta) = \begin{bmatrix} -\sin\theta & \cos\theta & L \\ -\sin(\frac{\pi}{3} - \theta) & -\cos(\frac{\pi}{3} - \theta) & L \\ \sin(\frac{\pi}{3} + \theta) & \cos(\frac{\pi}{3} + \theta) & L \end{bmatrix}$$

And ω_i , i=1, 2, 3, denotes the angular velocity of each wheel, respectively. Notice that the matrix P (θ) is always nonsingular for any θ, and

$$P^{-1}(\theta) = \begin{bmatrix} -\frac{2}{3}\sin\theta & -\frac{2}{3}\sin(\frac{\pi}{3} - \theta) & \frac{2}{3}\sin(\frac{\pi}{3} + \theta) \\ \frac{2}{3}\cos\theta & -\frac{2}{3}\cos(\frac{\pi}{3} - \theta) & -\frac{2}{3}\cos(\frac{\pi}{3} + \theta) \\ \frac{1}{3L} & \frac{1}{3L} & \frac{1}{3L} \end{bmatrix}$$

Thus; we can derive the forward kinematics as:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \frac{R}{3}\begin{bmatrix}-2\sin\theta & -2\sin(\frac{\pi}{3}-\theta) & 2\sin(\frac{\pi}{3}+\theta)\\ 2\cos\theta & -2\cos(\frac{\pi}{3}-\theta) & -2\cos(\frac{\pi}{3}+\theta)\\ \frac{1}{L} & \frac{1}{L} & \frac{1}{L}\end{bmatrix}\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix}$$

And the inverse kinematics can be expressed as:

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix} = \frac{3}{R}\ P(\theta)\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}$$

3.3.2 The Robot Dynamics:

The OMR dynamic model is described by:

$$M(q)\ddot{q}\ + C(q,\dot{q})\dot{q} = B(q)\tau$$

Where $q = [x \quad y \quad \theta]^T$ is the configuration vector, $\dot{q}$ is the velocity vector, $\ddot{q}$ is the acceleration vector, and $\tau \in \mathbb{R}^3$ is the control input which includes the torques applied to the wheels. $M(q) \in \mathbb{R}^{3\times3}$ is the inertia matrix, $C(q,\dot{q})\dot{q} \in \mathbb{R}^3$ is the Coriolis and centrifugal force vector, and $B(q) \in \mathbb{R}^{3\times3}$ is the input matrix. These are given as: [9]

$$\tau = \begin{bmatrix}T_1\\ T_2\\ T_3\end{bmatrix}$$

$$M(q) = \begin{bmatrix}m+\frac{3J}{2R^2} & 0 & 0\\ 0 & m+\frac{3J}{2R^2} & 0\\ 0 & 0 & I_Z+\frac{3JL^2}{R^2}\end{bmatrix}$$

$$C(q,\dot{q})\begin{bmatrix}0 & \frac{3J}{2R^2}\dot{\theta} & 0\\ -\frac{3J}{2R^2}\dot{\theta} & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$$

$$B = \begin{bmatrix} -\frac{1}{2R}(\sin\theta - \sqrt{3}\cos\theta) & \frac{1}{R}\sin\theta & -\frac{1}{2R}(\sin\theta + \sqrt{3}\cos\theta) \\ \frac{1}{2R}(\cos\theta + \sqrt{3}\sin\theta) & -\frac{1}{R}\cos\theta & \frac{1}{2R}(\cos\theta - \sqrt{3}\sin\theta) \\ \frac{L}{R} & \frac{L}{R} & \frac{L}{R} \end{bmatrix}$$

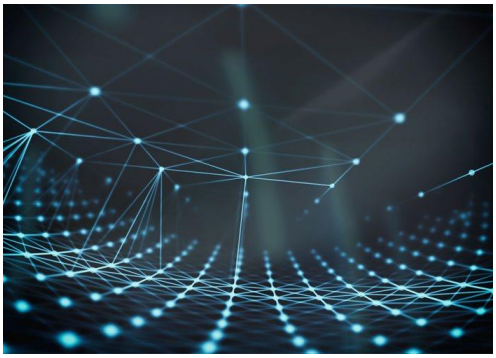
The system parameters correspond to those of a physical prototype and are listed in Table1.

Table 1 : parameters of the omnidirectional mobile robot

parameter	description	Value	units
J	Wheel's inertia	5.82E-4	Kg.m²
I_Z	Mobile's inertia	0.0127	Kg.m²
m	Mass	11.83	Kg
L	Wheel's radius	0.0625	m
R	Distance to the wheels	0.287	m

3.4 Conclusion:

In this chapter we have presented the kinematic and dynamic description of our omnidirectional robot .This kinodynamic model will be used to simulate and test the performance of the kinodynamic planning algorithm. The next chapter will show the obtained results.



CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction:

In this chapter, we present our work, in which we use the kinodynamic model of the robot presented in chapter 3 and the RRT algorithms presented in chapter 2 to do the simulation and plan the motion of the omnidirectional robot in a known map.

4.2 MATLAB simulation:

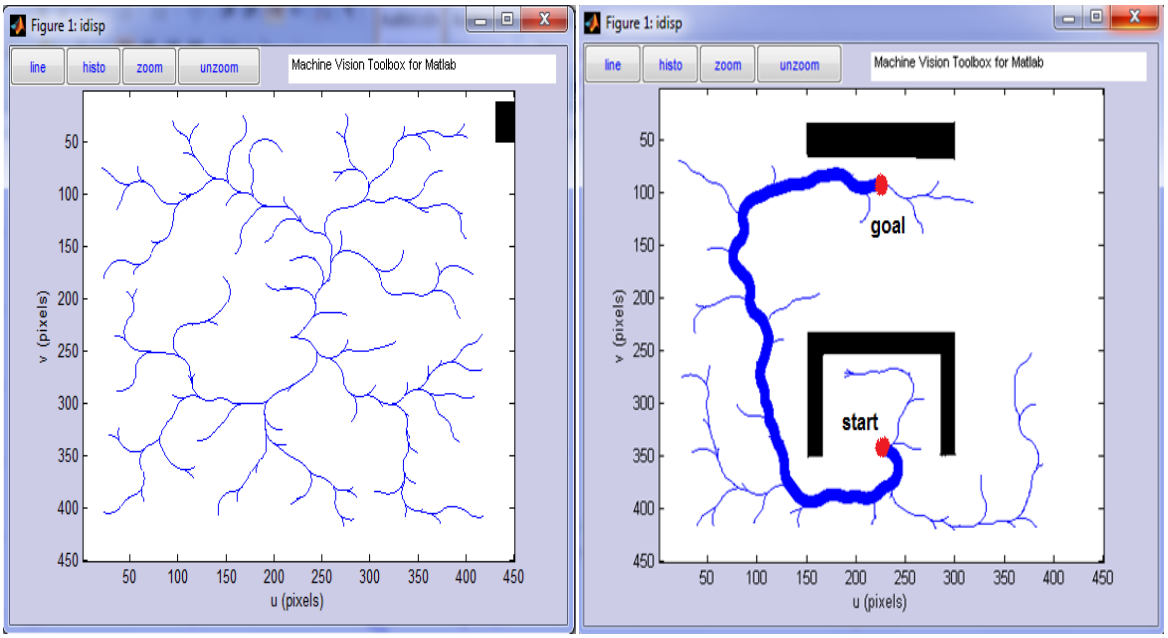
In this simulation we assume that all obstacles are stationary and the environment is well known.

4.2.1 Trajectory planning with the Basic RRT:

Figure 4.1 shows a simulation of the Basic RRT and its main feature.

Figure (a) shows our implemented Kinodynamic RRT exploring a completely free environment.

Figure (b) shows the advantage of using RRT for planning. It explores the whole environment (hence the name *Rapidly-Exploring*) to find a trajectory toward the desired goal (it is not greedy to the goal).



(a) (b)

Figure 4.1: The basic RRT. (a) The Exploring RRT. (b) The Planning RRT.

4.2.1.1 Complications of Rapidly Exploring in The State Space:

When exploring in X for a kinodynamic planning problem, several complications immediately occur:

i) The dimension is typically high

ii) The tree must stay in X_{free}

iii) Drift and other dynamic constraints can yield undesired motions and biases

iv) There is no natural metric on *X* for selecting ‘’nearest’’ neighbors

For the first complication, approximate nearest neighbor techniques can be employed to help improve the performance.

The second complication can make it harder to wander through narrow passages, much like in the case of probabilistic roadmaps.

The third complication can be partly overcome by choosing an action that brings the velocity component of x as close as possible toward the random sample.

The fourth complication might lead to the selection of one metric over another for particular kinodynamic planning problems, if one would like to optimize performance.

In theory, there exists a perfect metric (or pseudo-metric due to asymmetry) that would overcome all of these complications if it were easily computable. This is the optimal cost (for any criterion,such as time, energy,etc) to get from one state to another. Unfortunately, computing the ideal metric is as hard as solving the original planning problem. In general, we try to overcome these additional complications while introducing as few heuristics as possible. This enables the planner to be applied with minor adaptation to a broad class of problems.

4.2.1.2 RRTs and Veronoi Bias:

The RRT exploration strategy has an interesting property: it is characterized by Voronoi bias. At each iteration, the probability that a node is selected is proportional to the volume of its Voronoi region; hence, search is biased toward those nodes with the largest Voronoi regions (representing unexplored regions of the configuration space). This causes RRTs to rapidly explore. Alternatively, RRTs can be seen as attempting to decrease dispersion. The random sample then becomes an estimate of the center of the largest empty ball, and its distance to its nearest neighbor an estimate of the dispersion. The RRT then grows toward that sample, attempting to decrease the dispersion. [7]

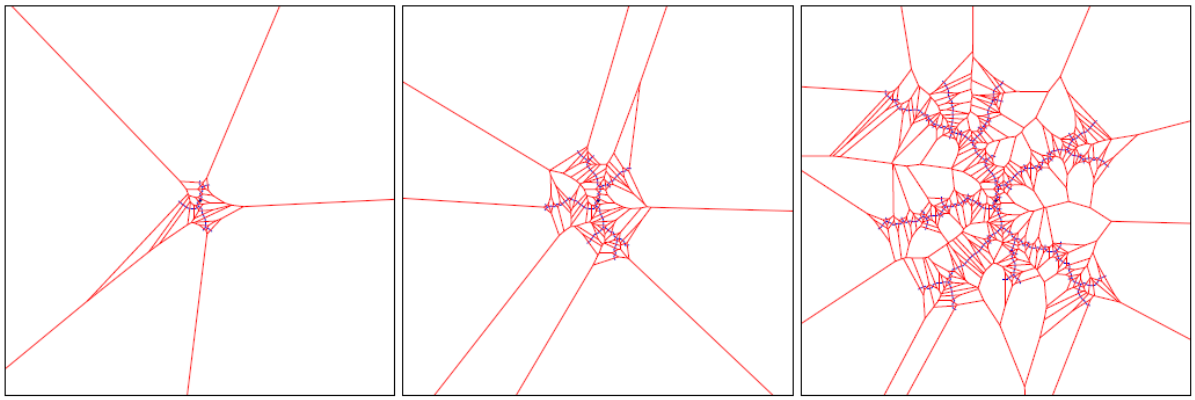
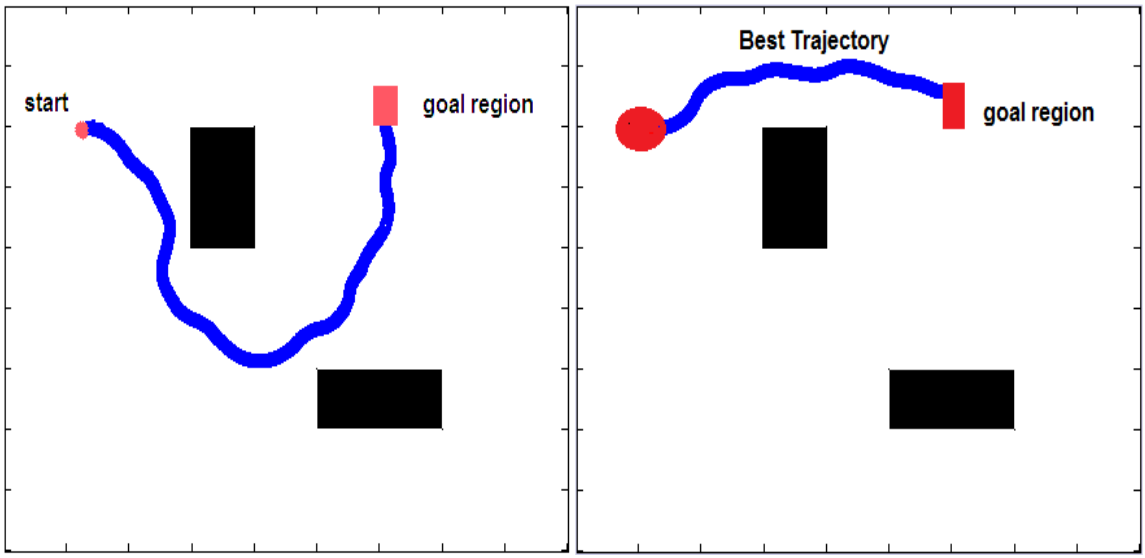


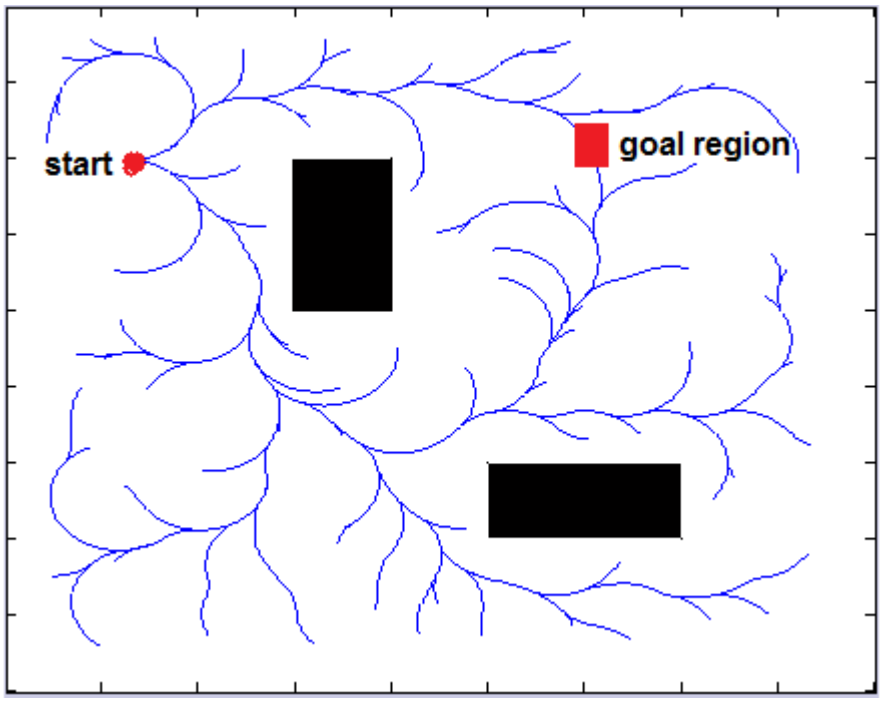
Figure 4.2: The RRT contains Veronoi bias which causes rapid exploration

4.2.2 Trajectory planning with RRT*:

In this section, we are going to present the simulation results of the RRT* and also a comparison between the basic RRT and RRT*.



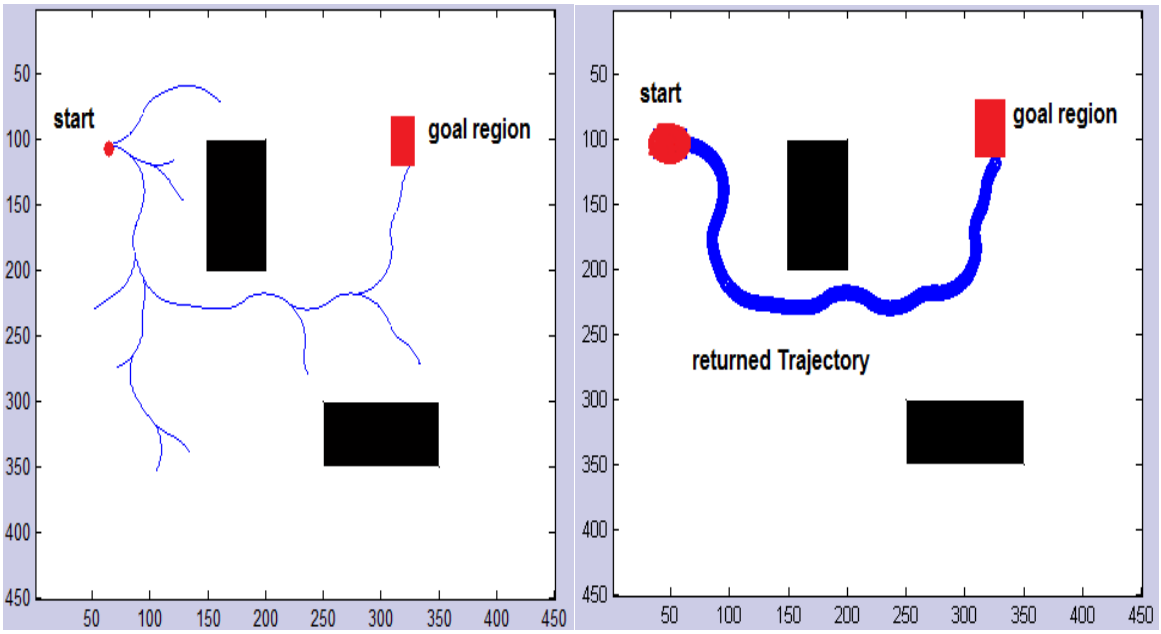
(a) (b)



(c)

Figure 4.3: RRT* simulation. (a) First found trajectory. (b) Best found trajectory.

(c) RRT*.



(a) (b)

Figure 4.4: The basic RRT simulation. (a) The Tree. (b) The returned trajectory.

Figure 4.3 shows that the anytime RRT* uses the whole given planning time in order to find the less costly trajectory while the basic RRT (Figure 4.4) breaks the planning loop to return the first found trajectory.

4.2.3 Extensions For Anytime Motion Planning:

This section describes how to exploit the anytime nature of the RRT* algorithm to achieve an online motion planning algorithm that significantly improves path quality during path execution, i.e. as the robot is moving toward its goal. These extensions are inspired by techniques for real-time kinodynamic planning.

4.2.3.1 Commited Trajectory

Given a motion plan $x : [0, T] \rightarrow X_{free}$ generated by the RRT* algorithm, the robot starts to execute an initial portion of $x : [0, t_{com}]$ until a given commit time t_{com}. We refer to this initial path as the committed trajectory. Once the robot starts executing the committed trajectory, the RRT* algorithm deletes each of its branches and declares the end of the committed trajectory $x(t_{com})$ to be the new tree root. This effectively shields the committed trajectory from any further modification. As the robot proceeds along the committed trajectory, the RRT* algorithm continues to improve the motion plan within the new (i.e., uncommitted) tree of trajectories. Once the robot reaches the end of the committed trajectory, the procedure restarts, using the initial portion of what is currently the best path in the RRT* tree to define a new committed trajectory. The iterative phase repeats until the robot reaches the goal region.

4.2.3.2 Branch-and-bound algorithm:

In addition to considering a committed trajectory, we also employ a branch-and-bound technique to more efficiently build the tree. Branch-and-bound is used within many domains in optimization and artificial intelligence. Most notably, the approach we present in this section shares certain aspects with the A* graph search algorithm and its variants, which are widely used in robotics applications [10].

1) Cost-to-go functions: Before providing the details of the branch-and-bound algorithm, let us first define a cost-to-go function as follows. For an arbitrary state $z \in X_{free}$, let

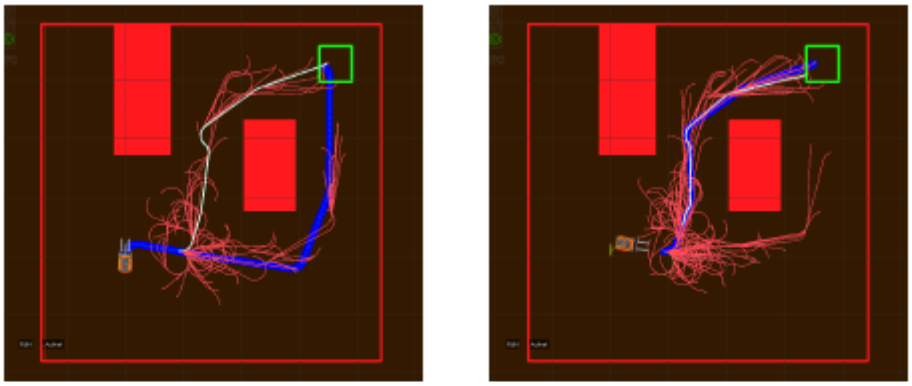
C_z^* be the cost of the optimal path that starts at z and reaches the goal region, X_{goal} ,. A cost-to-go function CostToGo(z) associates each $z \in X_{free}$ with a real number between 0 and C_z^*. Essentially, CostToGo(z) provides a lower-bound on the optimal cost to reach the goal from z. There are many ways to define a cost-to-go function, the most trivial being CostToGo(z) = 0 for all $z \in X_{free}$.

In this paper, we use the Euclidean distance between z and X_{goal} (neglecting obstacles) divided by the maximum speed of the vehicle as a cost-to-go function.

2) Branch-and-bound algorithm: In the context of the RRT and RRT*, the branch-and-bound algorithm works as follows. Let T = (V,E) be a tree and $z \in V$ be a vertex in T . Cost(z) denotes the cost of the unique path that starts from the root node and reaches z through the edges of T . Let z_{min} be the node that lies in the goal region and has the lowest-cost trajectory that reaches X_{goal} along the edges of T . The cost of the unique trajectory that starts from the root and reaches z_{min} gives an upper bound on cost. Let V' denote the set of nodes z for which the cost to get to z, plus the lower-bound on the optimal cost-to-go, is more than the upper-bound c_u, i.e., $V' = \{z \in V | Cost(z) + CostToGo(z) \geq Cost(z_{min})\}$. The branch-and-bound algorithm keeps track of all such nodes and periodically deletes them from the tree.

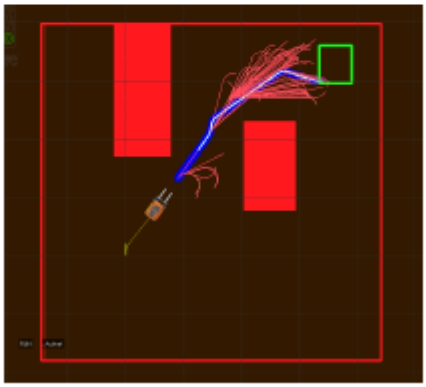
4.2.4 Performance Analysis:

Figure 4.5 depicts the result of two independent runs of the RRT* in the simulation environment. In the first, the RRT* initially finds a trajectory that takes the vehicle along a relatively high cost path to the right of the obstacle (Figure 4.1(a), in blue). As the vehicle begins to execute the plan, however, tree rewiring reveals a shorter, lower-cost route between the obstacles (Figure 4.5(b)). Meanwhile, the second run demonstrates the benefit of branch-and-bound and online refinement as the algorithm improves the current path (Figure 4.5(c)) into a more direct path to the goal (Figure 4.5(d)).

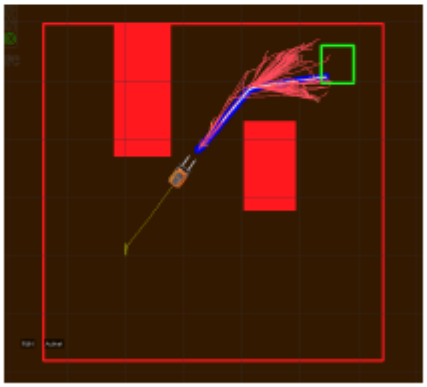


(a) RRT* run 1

(b) RRT* run 1



(c) RRT* run 2



(d) RRT* run 2

Figure 4.5: The RRT* tree at two points during the execution of two different simulation runs. In the first run, (a) the planner initially finds the longer path to the right of the obstacle but, as a result of the online refinement, (b) the RRT* correctly chooses the lower cost path between the obstacles. The results of the second run demonstrate typical behavior of the RRT*, which refines (c) an initial path into (d) a more direct path to the goal.

We compare the paths executed by the RRT* with those that result from a standard RRT-based planner. Figure 4.6 shows two different runs of the RRT at different points of execution. The re-planning together with branch-and-bound enable the RRT to refine an existing solution as demonstrated by the removal of unnecessary loops in the path. In contrast to the RRT* algorithm, however, these improvements tend to be local in nature and do not provide the significant modifications to the structure of the tree necessary to achieve lower cost solutions. Consequently, the free space bias of the RRT limits the extent to which the planner is able to refine paths. This effect is evident in the result of the first run as the RRT gets “stuck” with a tree that favors longer paths to the right of the obstacle (Figure 4.6(a)) and converges to a sub-optimal path (Figure 4.6(b)). As is evident in Figure 4.7(a), the RRT

frequently produces trajectories that are unnecessarily long due either to the selection of over-long routes, or to oscillations in otherwise direct paths.

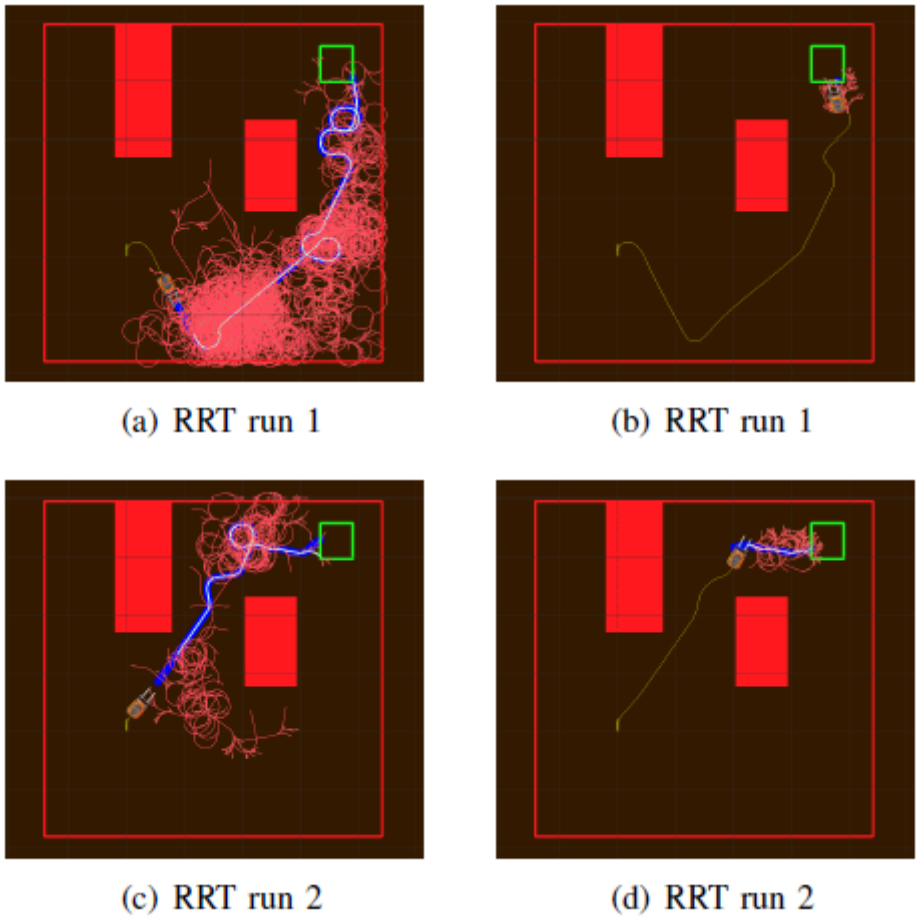


Figure 4.6 : Two simulation runs with the RRT motion planner. (a,b) The first run demonstrates a common failure of the RRT, which effectively gets stuck after constructing a tree biased toward the longer route to the goal. While the RRT does refine the path (b), it converges to a high-cost solution. (c) During the initial period of the second run, the RRT identifies a feasible path to the goal that includes a loop maneuver. The planner continues to search for an improved trajectory and, with the assistance of branch-and-bound, (d) discovers a shorter loop-free path that the vehicle then executes.

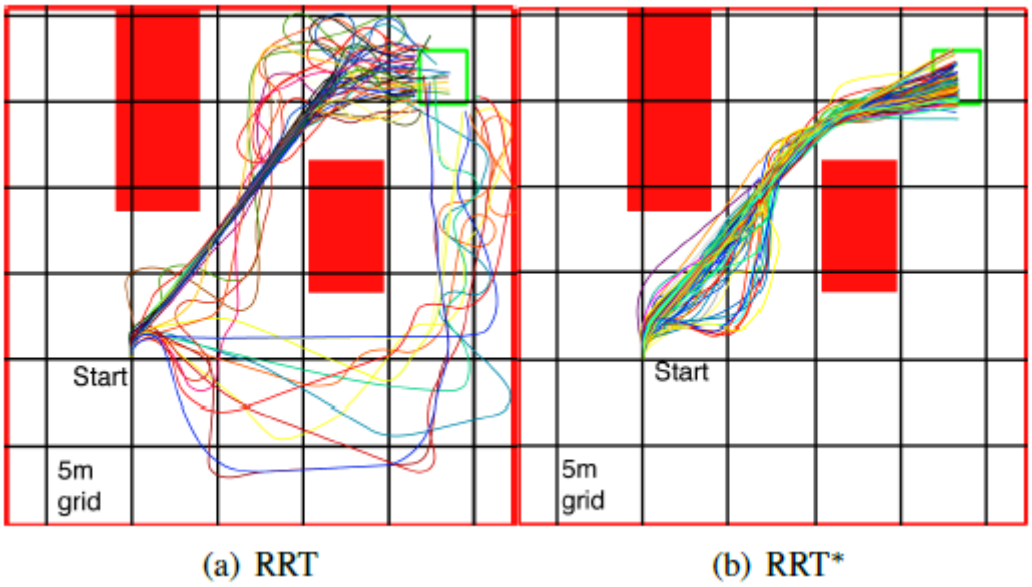
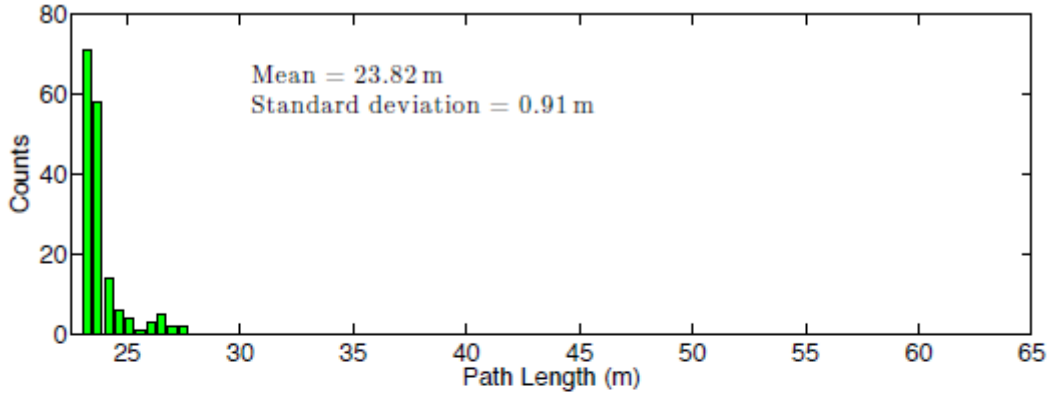


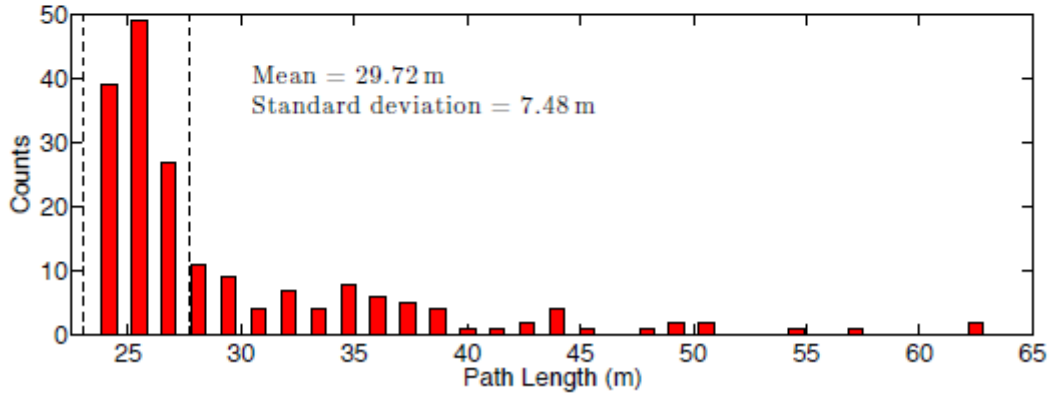
Figure 4.7: Vehicle paths traversed for (a) 65 simulations of the RRT and (b) 140 simulations with our RRT* planner.

Figure 4.7(b) depicts the final paths for the RRT* simulations. In each case, the algorithm correctly identifies the route between the two obstacles as providing shorter paths to the goal. Occasionally, the RRT* yields an initial solution that steers the vehicle away from the goal. As the vehicle executes the path, the RRT* rewires the structure of the tree to discover a more direct path. This refinement continues while the vehicle executes the committed portion of the trajectory. The result is loop-free paths that tend to be more direct than those of the RRT.

The online formulation of the RRT* algorithm exploits the execution period to modify the tree structure as it converges to the optimal path. This convergence is evident in the distribution over the length of the executed simulation trajectories (Figure 4.8(a)) that exhibits a mean length (cost) of 23.82m and a standard deviation of 0.91m for the set of 166 simulations. For comparison, Figure 4.8(b) presents the corresponding distribution for the RRT planner. The mean path length for the 191 RRT simulations is 29.72m while the standard deviation is 7.48 m. The significantly larger variance results from the RRT getting "stuck" refining a tree with suboptimal structure. The anytime RRT*, on the other hand, opportunistically takes advantage of the available execution time to converge to a near-optimal path.



(a) RRT*



(b) RRT

Figure 4.8: Histogram plots of the executed path length for simulations of (a) the RRT* and (b) the RRT. The vertical dashed lines in (b) depict the range of path lengths that result from the RRT* planner.

4.2.5 Motion Planning for The OMR :

As we have seen in chapter 3, the dynamic model of the OMR has the following main parameters:

M = 11.83 kg L = 0.0625 m R = 0.287 m I_z = 0.0127 kg.m² J = 5.82 E-4 kg.m²

We used the above parameters to demonstrate the performance of the RRT*.We conducted a series of tests with both the RRT* anytime algorithm as well as the RRT-based planner. The vehicle operated in a 20m by 20m packed gravel environment consisting of five obstacles (Figure 4.9). The task was to navigate from a starting position in one corner to a 1.6m goal region in the opposite corner while avoiding the obstacles. We manually specified the location of the obstacles. In each experiment, planning started immediately prior to the controller tracking the committed trajectory.

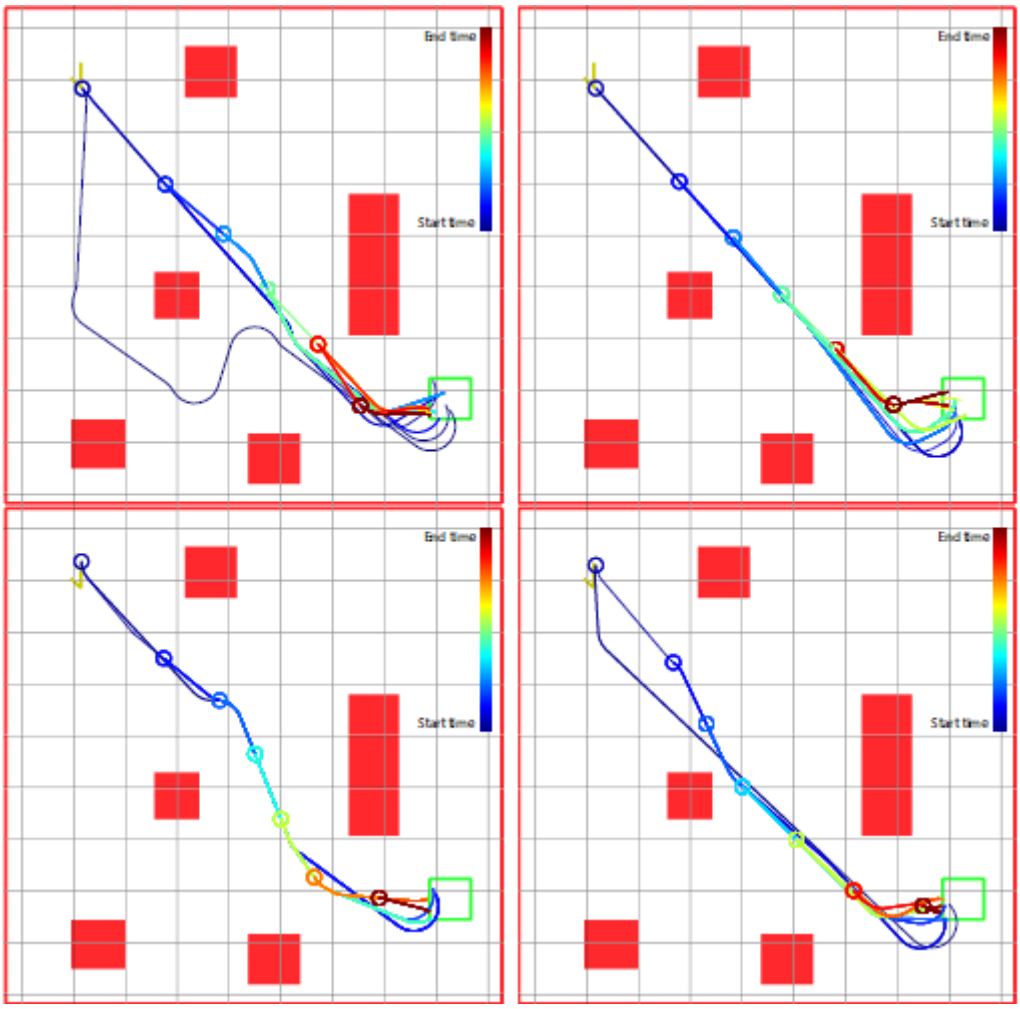


Figure 4.9: Four runs of the anytime RRT* on the OMR . Starting in the upper left, the OMR was tasked with driving to the goal region while avoiding obstacles. The trajectories indicate the optimal path as estimated by the RRT* at different points in time during the execution and are false-colored by time. Circles denote the initial position for each path.

Figure 4.9 presents the result of four different tests with the RRT* anytime motion planner. The plots depict the best trajectory as maintained by the RRT* at different points during the plan execution (false-colored by time). In the scenario represented in the upper left, the RRT* initially identifies a sub-optimal path that goes around an obstacle but, as the vehicle begins to execute the path, the planner correctly refines the solution to a shorter trajectory. As the vehicle proceeds along the committed trajectory, the planner continues to rewire the tree as evident in the improvements near the end of the execution when the paths more directly approach the goal.

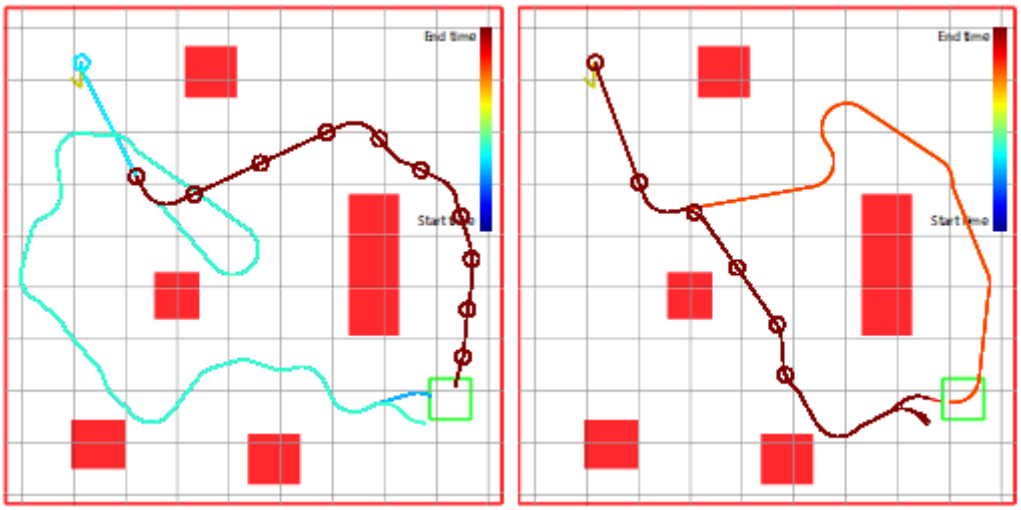


Figure 4.10: Plans generated by the anytime planner using the standard RRT.

For comparison, Figure 4.10 presents the resulting paths for the anytime planner utilizing the standard RRT. In the scenario depicted on the left, the RRT initially finds a looping trajectory that goes wide to the left but, after moving a few meters, discovers a shorter path that takes the vehicle wide to the right. At this point, the structure of the tree biases the RRT toward refinements that improve the trajectory only locally. In the second test, the RRT revises the initial trajectory that unnecessarily goes to the right of the obstacle and discovers a shorter, yet sub-optimal path to the goal.

4.3 Conclusion:

In this chapter, we have presented and discussed the results of our entire work . Incremental sampling-based motion planners have been used successfully to plan trajectories for OMR with restricted dynamics operating in the presence of obstacles. We analysed and compared the performance of Anytime-RRT and Anytime-RRT* in terms of optimality and convergence to the optimal path .

CONCLUSION

In this report, we applied a kinodynamic planning for a three-wheeled omnidirectional robot in a known environment with static obstacles. We used the RRT algorithm to plan a trajectory from a start state to a final state in the state space using the kinematic and dynamic models of the robot. The appeal of incremental planners such as the RRT stems, in part, from their efficiency at identifying feasible motion plans and their intuitive implementation. However, the feasible solutions produced by the RRT tend to be far from optimal. This report described an anytime motion planning algorithm that uses the RRT* to solve for and improve solutions to the motion planning problem in an online fashion. We described methods that enable the planner to asymptotically converge to the optimal solution online, during trajectory execution. We used Matlab simulation to evaluate convergence of the anytime RRT* algorithm, and compared it to a standard RRT-based motion planner. We further demonstrated the algorithm's performance while planning trajectories for the OMR .

BIBLIOGRAPHY

[1] Kevin M. Lynch and Frank C. Par.''modern robotics'' , May 3, 2017

[2] BELKHIR Malek and KEDDAD Adel ,'' Kinodynamic Motion Planning For Car-Like Robots'', final year project report , 2012

[3] Steven.M.LaValle and James J. Kuffner Jr .''Randomised Kinodynamic Planning '', 1996

[4] Weihao Li. ''Motion Planning for Omnidirectional Wheeled Mobile Robot by Potential Field Method '', 12 March 2017

[5] T.A Baede . ''Motion Control of an Omnidirectional Robot '', DCT 2006.084, Eindhoven, September 18th, 2006.

[6] Aisha Walcott, Nathan Ickes,Stanislav Funiak.''kinodynamic path planning'',October 31, 2001.

[7] Stephen R.Lindemann and Steven M.La Valle.''incrementally Reducing Dispersion by Increasing Voronoi Bias in RRTs'', 2002

[8] Miguel G. Villarreal-Cervantes,''Motion Control Design for an Omnidirectional Mobile Robot Subject to Velocity Constraints'',15 February 2015.

[9] Vjekoslav Damić . '' dynamic analysis of an omnidirectional mobile robot''. 17th International Research/Expert Conference TMT 2013, Istanbul, Turkey, 10-11 September 2013

[10] Baijayanta Roy. ''A* Search Algorithm '', towardsdatascience.com, Sept 29, 2019

[11] IJCSNS International Journal of Computer Science and Network Security, VOL.16 No.10, 2016.

Appendix

1. RRT MATLAB code:

```
clearvars
close all
x_max = 1000;
y_max = 1000;
obstacle = [500,150,200,200];
EPS = 20;
numNodes = 3000;
q_start.coord = [0 0];
q_start.cost = 0;
q_start.parent = 0;
q_goal.coord = [999 999];
q_goal.cost = 0;
nodes(1) = q_start;
figure(1)
axis([0 x_max 0 y_max])
rectangle('Position',obstacle,'FaceColor',[0 .5 .5])
hold on
for i = 1:1:numNodes
q_rand = [floor(rand(1)*x_max) floor(rand(1)*y_max)];
plot(q_rand(1), q_rand(2), 'x', 'Color', [0 0.4470 0.7410])
% Break if goal node is already reached
for j = 1:1:length(nodes)
if nodes(j).coord == q_goal.coord
break
end
end
% Pick the closest node from existing list to branch out from
ndist = [];
for j = 1:1:length(nodes)
n = nodes(j);
tmp = dist(n.coord, q_rand);
ndist = [ndist tmp];
end
[val, idx] = min(ndist);
q_near = nodes(idx);
q_new.coord = steer(q_rand, q_near.coord, val, EPS);
if noCollision(q_rand, q_near.coord, obstacle)
line([q_near.coord(1), q_new.coord(1)], [q_near.coord(2), q_new.coord(2)], 'Color',
'k', 'LineWidth', 2);
drawnow
hold on
```

42.RRT* MATLAB code :

```
clearvars
close all
x_max = 1000;
y_max = 1000;
obstacle = [500,150,200,200];
EPS = 20;
numNodes = 3000;
q_start.coord = [0 0];
q_start.cost = 0;
q_start.parent = 0;
q_goal.coord = [999 999];
q_goal.cost = 0;
nodes(1) = q_start;
figure(1)
axis([0 x_max 0 y_max])
rectangle('Position',obstacle,'FaceColor',[0 .5 .5])
hold on
for i = 1:1:numNodes
q_rand = [floor(rand(1)*x_max) floor(rand(1)*y_max)];
plot(q_rand(1), q_rand(2), 'x', 'Color', [0 0.4470 0.7410])
% Break if goal node is already reached
for j = 1:1:length(nodes)
if nodes(j).coord == q_goal.coord
break
end
end
% Pick the closest node from existing list to branch out from
ndist = [];
for j = 1:1:length(nodes)
n = nodes(j);
tmp = dist(n.coord, q_rand);
ndist = [ndist tmp];
end
[val, idx] = min(ndist);
q_near = nodes(idx);
q_new.coord = steer(q_rand, q_near.coord, val, EPS);
if noCollision(q_rand, q_near.coord, obstacle)
line([q_near.coord(1), q_new.coord(1)], [q_near.coord(2), q_new.coord(2)], 'Color', 'k',
'LineWidth', 2);
drawnow
hold on
q_new.cost = dist(q_new.coord, q_near.coord) + q_near.cost;
% Within a radius of r, find all existing nodes
q_nearest = [];
r = 60;
neighbor_count = 1;
for j = 1:1:length(nodes)
```

```
if noCollision(nodes(j).coord, q_new.coord, obstacle) && dist(nodes(j).coord,
q_new.coord) <= r
q_nearest(neighbor_count).coord = nodes(j).coord;
q_nearest(neighbor_count).cost = nodes(j).cost;
neighbor_count = neighbor_count+1;
end
end
% Initialize cost to currently known value
q_min = q_near;
C_min = q_new.cost;
% Iterate through all nearest neighbors to find alternate lower
% cost paths
for k = 1:1:length(q_nearest)
if noCollision(q_nearest(k).coord, q_new.coord, obstacle) && q_nearest(k).cost +
dist(q_nearest(k).coord, q_new.coord) < C_min
q_min = q_nearest(k);
C_min = q_nearest(k).cost + dist(q_nearest(k).coord, q_new.coord);
line([q_min.coord(1), q_new.coord(1)], [q_min.coord(2), q_new.coord(2)], 'Color', 'g');
hold on
end
end
% Update parent to least cost-from node
for j = 1:1:length(nodes)
if nodes(j).coord == q_min.coord
q_new.parent = j;
end
end
% Append to nodes
nodes = [nodes q_new];
end
end
D = [];
for j = 1:1:length(nodes)
tmpdist = dist(nodes(j).coord, q_goal.coord);
D = [D tmpdist];
end
% Search backwards from goal to start to find the optimal least cost path
[val, idx] = min(D);
q_final = nodes(idx);
q_goal.parent = idx;
q_end = q_goal;
nodes = [nodes q_goal];
while q_end.parent ~= 0
start = q_end.parent;
43. line([q_end.coord(1), nodes(start).coord(1)], [q_end.coord(2), nodes(start).coord(2)],
    'Color', 'r', 'LineWidth', 2);
44. hold on
45. q_end = nodes(start);
46. end
```

DEDICATION

To my dear ones, the living and the dead

ACKNOWLEDGEMENTS

First of all, praise to Allah, lord of the worlds, who, only him has helped me and guided me through all the steps to finishing this work.

Secondly, I am thankful to my parents and friends and the supervisor for their Financial and emotional support throughout my whole life.

ABSTRACT

This project is about kinodynamic planning for a three-wheeled omnidirectional mobile robot in a static environment. First we present the kinematics and dynamics of the robot, then we use those models to plan the trajectory of the robot in the state space using the RRT then the anytime RRT sampling-based algorithms. The algorithms are then implemented and tested on MATLAB.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE:

CHAPTER TWO:

CHAPTER THREE:

CHAPTER FOUR: