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

University M'Hamed BOUGARA - Boumerdes

Institute of Electrical and Electronic Engineering

Project Report Presented in Partial Fulfillment of the Requirements of the Degree of:

'Master'

In Control Engineering

Feedback Motion Planning for Car-Like Robots

Presented by:

DERAR Iyed

KHOUMARI Malik

Supervisor:

Dr.R Geurnane

Academic year:2023/2024

Abstract

In this study, we address the problem of feedback motion planning for a car-like robot described by a kinematic model, navigating through obstacles. For this purpose, we designed two sampling-based algorithms equipped with the feedback property, a Modified Rapidly Exploring Random Tree Star (RRT*) algorithm and a Funnel-Graph algorithm. Both algorithms generate collision-free paths while accounting for non-holonomic constraints and uncertainties. These generated paths are then fed to the pure pursuit controller which handles the robot motion execution. Our approach is validated by testing the algorithms' ability to handle different uncertainties and adaptability to environmental changes, including a performance comparison between them. The results show the superiority of the funnel-graph algorithm across all tests, which makes it best suited for real-time applications.

Acknowledgement

Our strong interest in robotics led us to our supervisor, Dr. R.Geurnane, who truly lived up to our expectations. We would like to express our heartfelt thanks and deep appreciation to him. His guidance and advice helped us through every stage of our project, especially during the tough times when we realized that our robot implementation was very difficult and felt very frustrated.

Our heartfelt thanks are extended to everyone who contributed to the successful completion of this project.

We would also like to acknowledge each other for the hard work and dedication we both put into this project. Working together as a team made all the difference, and we supported each other through every challenge. Our combined efforts and collaboration were key to overcoming the obstacles we faced, and we are grateful for each other's commitment and perseverance.

Dedication

This project is dedicated to my beloved parents, who have been my source of inspiration and strength, who continually provide their moral, emotional and financial support.
To my big brother, sisters, relatives, friends and classmates who shared their words of advice and encouragement to finish this project.

KHOUMARI Malik

I would like to dedicate this work to the memory of my grandmother, She was always encouraging me to pursue the highest levels in studies, and life and to achieve the best version of myself, I wish she could see me graduate and still pursuing my dreams carrying her will with me. May Allah have mercy on her. I would like also to dedicate this work to my parents, Hopefully, I could repay them for the numerous sacrifices they made for my sake, also my little brother who is always my unwavering pillar.

DERAR Iyed

Contents

List of Figures

List of Tables

Introduction

Integrating motion planning and control techniques to enable physical vehicles to execute complex missions without human intervention is a core area of research in the field of autonomous vehicles. Motion planning involves determining a sequence of actions to move a robot from an initial state to a desired goal state while avoiding obstacles and satisfying various constraints. One of the most significant challenges confronting autonomous robots in motion planning is planning under uncertainty where most real-world robotic jobs involve motion uncertainty (caused by external disturbances) and poor state information (caused by partial and noisy measurements) where open-loop planning fails to account for these unpredictable events, which raises the need for feedback planning. As an important tool, feedback motion planning for mobile robots improves system performance by allowing the successful completion of the navigation task even in the presence of external disturbances.

In this report, the proposed robotic system is a vehicle whose kinematic model describes the mobility of a car. The state of the robot is represented by the position and orientation of its main body in the plane, velocity and wheel steering angle as inputs for motion control. Two sampling-based path planning algorithms are designed, a modified RRT* path planner and a funnel-based path planner, they will be dealt with separately. The role of the planner is to solve the obstacle avoidance problem and provide a series of motion goals to the controller. In this perspective, a pure pursuit controller is used, it deals with the basic issue of converting ideal plans into actual motion execution. The feedback law will be embedded as a part of the control, such that it intervenes whenever it is necessary. The designed algorithms are equipped with dynamic re-planning and goal-tracking properties, and their performance will be tested by introducing a measurement noise to the system.

A preview of the chapters is provided below:

- **Chapter 1** collects the definitions of all necessary informations required to understand the concept of path planning and motion planning. Starting from basic terminologies, passing through the geometric representation of the environment and the rigid body transformation description, ending by listing the most familiar path planners and path tracking controllers.

- **Chapter 2** delves into the concept of feedback motion planning, offering a thorough introduction to the foundational idea of feedback. Vector fields, and integral curves, which are essential for constructing navigation functions. The chapter then explores two different feedback motion planners, describing the functionality of each. The constructed algorithms are then explained in detail.

- **Chapter 3** serves as a display of the results obtained from simulating the system under normal and noisy conditions. Then, It evaluates the system's adaptability to environmental changes. The chapter concludes by introducing the goal-tracking feature specific to the Funnel-Graph algorithm in both static and dynamic environments. An extension describing our short experience with the implementation of the system is added.

Chapter 1

Theoretical background

The objective of this chapter is to provide a foundational understanding of key concepts and terminology essential for comprehending motion planning. Additionally, we will elucidate commonly utilized techniques and planners, while also establishing clear definitions for non-holonomic robots.

1.1 Basic definitions

Before embarking on the discussion of motion planning, it is crucial to familiarize oneself with the terminologies employed within this domain.

The World: Or the workspace denoted by $\mathcal{W}$, usually it is $\mathbb{R}^2$ or $\mathbb{R}^3$, it contains two kinds of entities, Obstacles and Robots[2].

Obstacles: Denoted by $\mathcal{O}_i$, they are sections of the world that are occupied[2].

Robots: Denoted by $\mathcal{A}_i$, geometrically modeled entities that can be manipulated using a specified motion plan[2].

The configuration space:

A configuration q represents a subset of the $\mathcal{C}$ space, an arbitrary object's configuration "state" is a specification of the position of every point in this object relative to a fixed reference frame[18]. (e.g in a 2D world, for a robot $\mathcal{A}$ that is capable of translation and rotation, each configuration $\mathcal{A}(q)$ has three parameters to be considered (x, y, θ)).

Denoted by $\mathcal{C}$, The configuration space of a robot $\mathcal{A}$ is the set $\mathcal{C}$, encompassing all possible configurations denoted by $\mathcal{A}(q)$ within the space $\mathcal{W}$. For a robot, $\mathcal{C}$ represents the set of all conceivable configurations, with the dimension determined by the degrees of freedom.

The configuration space comprises two distinct subspaces: $\mathcal{C}_{free}$, representing the set of collision-free configurations, and $\mathcal{C}_{Obs}$, denoting the set of configurations that lead to collisions with obstacles.

$$[\ C_{\text{obs}} = \{q \in C \mid A(q) \cap O_i \neq \emptyset\} \tag{1.1}$$

Path: A continuous mapping of states in the state space. A path is collision-free if each element of the path is an element of the free state space.

1.2 Geometric Modeling

To effectively address motion planning challenges, it's crucial to model robots and obstacles. These models represent a system of objects or bodies within a defined workspace $\mathcal{W}$.In most cases to model the object effectively two representations are used the most:

Boundary representation : as the name suggests it models a body by defining its boundaries so that every point inside is considered from the body.

Solid representation: This representation models the body by defining the set of all points belonging to the object's body.

1.2.1 Polygonal Modeling

In a two-dimensional world a polygon is a flat geometric shape consisting of at least three straight sides and angles. Polygons are widely used for modeling objects in a workspace. because of their straightforwardness and effectiveness in representing shapes and surfaces.

Convex polygons: a convex polygon is a polygon whose all vertices are pointing outward, and for any pair of points in a polygon X, all points along the line segment that connects them are contained in X [2].

A boundary representation of object O is defined by an m-sided convex polygon, where vertices and edges characterize its structure. Each vertex represents a corner of the polygon, while edges correspond to line segments connecting pairs of vertices. The polygon's outline can be precisely delineated by a sequence of m points, arranged in counterclockwise order.

To establish a solid representation of an object, we initially define a primitive H, which is a subset of $\mathcal{W}$ that is straightforward to represent and manipulate using a computer. with this concept we can represent complicated objects with a combination of primitives[2]For instance, we can designate a half-plane as a primitive, where all points lying on one side of a line are included in our object representation. As shown in figure 1.1 this approach allows us to define a polygon as the intersection of multiple half-planes.[2]

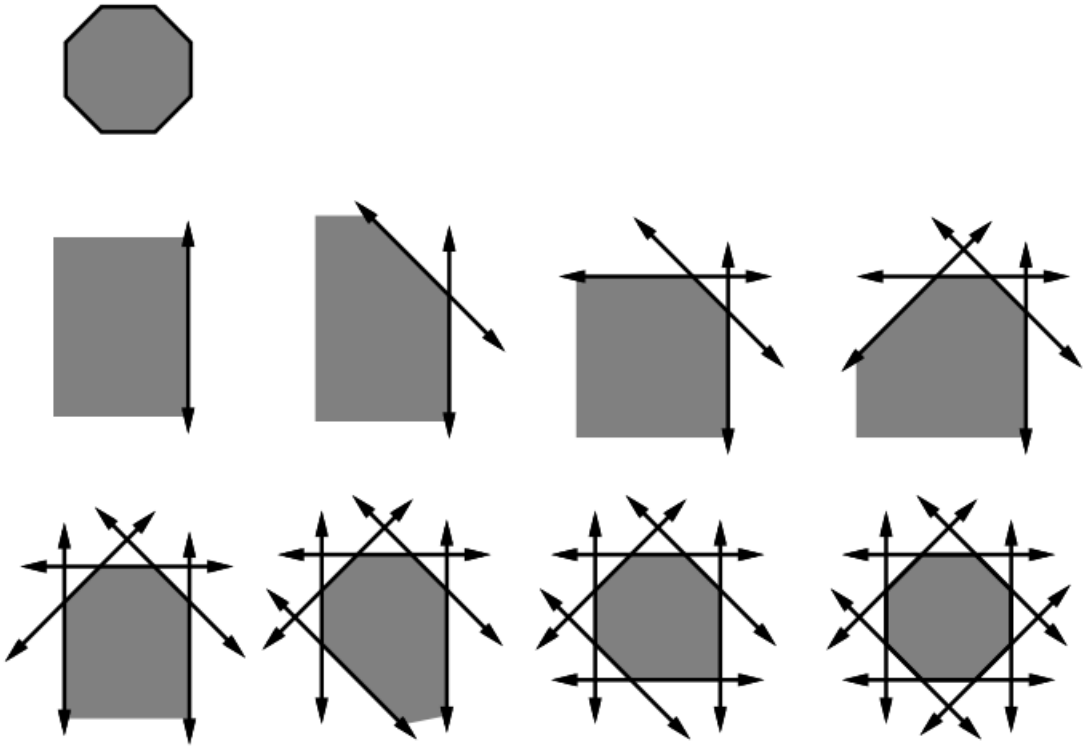


Figure 1.1: A polygon can be defined as the intersection of half planes

Non-convex polygons can be made by combining convex polygons. It's worth noting that the decomposition of non-convex polygons is not a unique process; there can be multiple ways to decompose them. One practical approach to decompose object is by applying the

vertical cell decomposition algorithm, which helps in breaking down complex shapes into simpler components.

1.2.2 Semi-algebraic models

Semi-algebraic sets are defined by means of first-order logic sentences whose predicates are $=, >, <$, whose variables are real numbers and whose terms are multivariate polynomials with rational coefficients [18], and we can consider these polynomials as semi-algebraic primitives for more complicated objects, for instance, we can represent a disk-shaped obstacle with $f = x^2 + y^2 - r$.

Overall, geometric modeling plays a pivotal role in motion planning, offering a structured approach to representing and analyzing the physical components of robotic systems. It enables robots to perceive and interact with their surroundings accurately, aiding in motion planning, simulation, and visualization of robot movements. Geometric models also facilitate collision detection, path optimization, and ensure the safety and efficiency of planning operations.

1.3 Rigid body transformations

A rigid-body transformation $h : \mathcal{A} \to \mathcal{W}$ is a function that maps each point in set $\mathcal{A}$ to set $\mathcal{W}$, satisfying two conditions:
1) It preserves the distance between any pair of points in $\mathcal{A}$.
2) It preserves the orientation of $\mathcal{A}$.

If robot $\mathcal{A}$ is defined by specifying individual points in $\mathbb{R}^2$, such as in a boundary representation of a polygon, then each point is straightforwardly transformed from a to $h(a) \in \mathcal{W}$[2]. The transformed robot by the transformation h is denoted $h(\mathcal{A})$ which indicates all the points occupied in $\mathcal{W}$ by the robot.

To proceed with transforming an object in robotics, it's crucial to define frames that describe the movement accurately. The world frame serves as a reference point, describing transformations relative to the workspace's coordinate system. It provides a global perspec-

tive for positioning objects and obstacles within the environment. On the other hand, the body frame is specific to the robot itself, offering a local coordinate system attached to the robot's structure. Figure 1.2 implicates how frame is instrumental in specifying the robot's movements, orientation, and interactions with its surroundings.

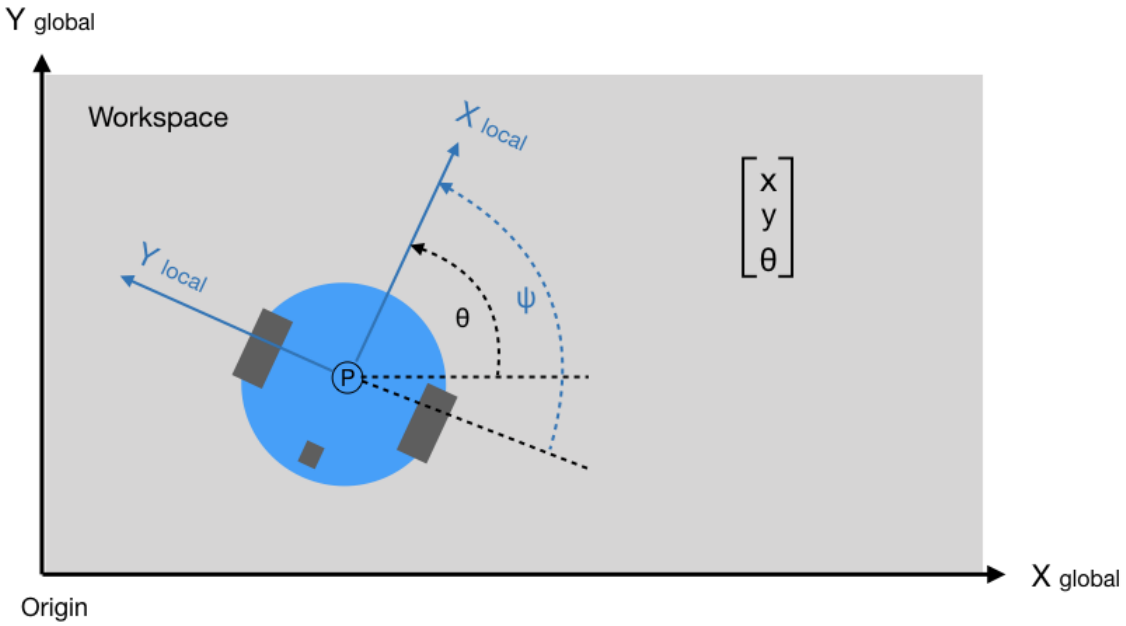


Figure 1.2: Definition of Frames in a 2D world

In a two-dimensional setting, the available transformations that can be applied to objects or entities include translation, which involves moving an object from one position to another without altering its orientation, and rotation, which involves turning or pivoting an object around a fixed point or axis or a combination of both translation and rotation.

Translation : In the context of robotics, a rigid robot $A \subseteq \mathbb{R}^2$ undergoes translation using two parameters, $x_t, y_t \in \mathbb{R}$, denoted as $q = (x_t, y_t)$. The transformation function h is defined as $h(x, y) = (x + x_t, y + y_t)$, where h translates the robot's position in the x and y directions by x_t and y_t units, respectively[2].

Rotation : The robot A can be rotated counterclockwise by an angle θ within the range $[0, 2\pi)$ using the transformation $(x, y) \rightarrow (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$ for every $(x, y) \in A$.

Combining translation and rotation: If a rotation by angle θ is performed, followed by a translation by x_t and y_t, this sequence of transformations can be used to position the robot in any desired location and orientation. However, it's important to note that translations and rotations do not commute, meaning the order in which they are applied matters.

If the operations are applied successively(Rotation then translation), each $(x, y) \in A$ is transformed to $(x\cos\theta - y\sin\theta + x_t, x\sin\theta + y\cos\theta + y_t)$.

1.4 Modeling the Configuration space

Obstacles in the workspace are tangible physical elements, whereas obstacles in the configuration space (C-space) signify configurations leading to collisions. Modeling C-space and specifying $\mathcal{C}_{Obs}$ are fundamental for generating collision-free paths during motion planning, optimizing robot movements, and ensuring adherence to safety constraints. This facilitates efficient and safe navigation in complex environments by identifying and avoiding potential collision scenarios.

Previous work on configuration space construction can be divided into two main methods: geometry-based and topology-based. Geometry-based approaches aim to compute the precise geometric representation of the configuration space, focusing on its shape and boundaries. In contrast, topology-based methods emphasize capturing the connectivity and paths within the configuration space. Both approaches offer valuable insights for motion planning and collision avoidance in robotics.

Geometry-based methods are usually limited to low-dimensional configuration spaces due to the combinatorial complexity involved in computing the boundary of C_{obs} for high-dimensional configuration spaces. Previous work in [1] has focused on the special case when objects A and B are rigid bodies only performing translational motion. Figure 1.3 shows one known geometrical method to construct C_{obs} is the Minkowski difference which is defined by:

$$A \ominus B = \{a - b \mid a \in A \text{ and } b \in B\}, \text{ where } A, B \subseteq \mathbb{R}^n \tag{1.2}$$

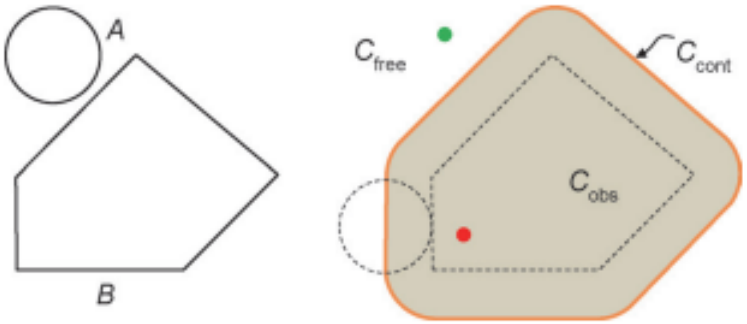


Figure 1.3: Geometrical interpretation of the Minkowski difference[1]

Topology-based methods, like sampling-based approaches shown in figure 1.4, focus on capturing the connectivity in the configuration space by generating random samples (milestones) in the free configuration space C_{free} and organizing them using a graph or tree structure. While they are faster than geometry-based methods in computing an approximate C-space representation, they may struggle with narrow passages and high-DOF robots, leading to slower performance in such scenarios[1].

Machine learning techniques have been integrated into topology-based methods to enhance the sampling process and optimize it, leading to significant advancements in navigating complex configuration spaces.

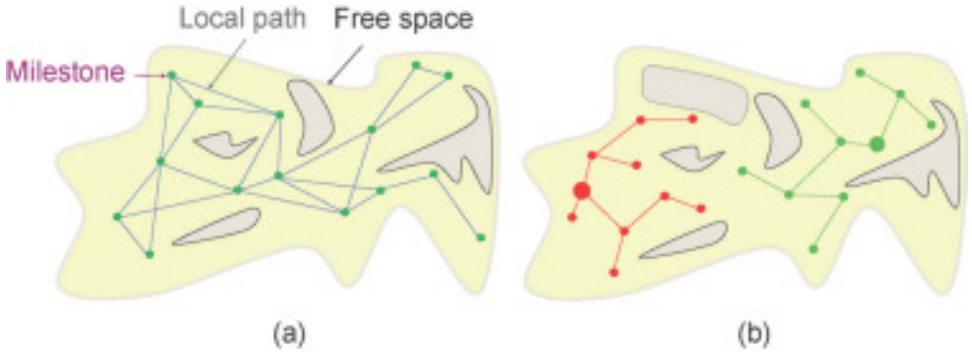


Figure 1.4: Sampling based method to construct the C-space[1]

1.5 Motion planning

The Piano Mover's problem shown in figure 1.5 is the basic motion planning problem in robotics, aimed at finding a continuous path for a robot from an initial configuration to a goal configuration while avoiding obstacles in its workspace. This problem is defined by several key components: the workspace W, which can be either two-dimensional or three-dimensional, an obstacle region O within the workspace, and a robot represented as either a rigid body or a collection of linked components. The configuration space C is partitioned into C_{obs} (occupied configurations) and C_{free} (free configurations), with the initial configuration q_{I} and goal configuration q_{G} specified as a query. The goal is to compute a continuous path τ from q_{I} to q_{G} in C_{free}, ensuring collision avoidance and efficient navigation for the robot.

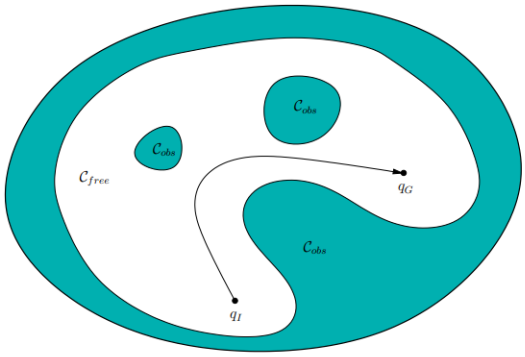


Figure 1.5: The basic motion planning problem.[2]

The main challenges in motion planning arise from the difficulty in directly computing the regions C_{obs} and C_{free}, as well as the high dimensionality of the configuration space (C-space). The computational complexity of motion planning problems, exemplified by the Piano Mover's problem, is significant.

1.5.1 Planning with potential Fields

An approach to motion planning draws inspiration from obstacle avoidance techniques. This method involves creating a differentiable real-valued function $\mathcal{U} : \mathbb{R}^m \to \mathbb{R}$, known as a potential function, which guides the motion of the moving object. The potential function

typically comprises an attractive component $\mathcal{U}_a(q)$, pulling the robot towards the goal, and a repulsive component $\mathcal{U}_r(q)$, pushing the robot away from obstacles[19].

While potential fields offer an intuitive framework for guiding robots towards goals and away from obstacles depicted in figure 1.6, their drawback lies in the lack of guarantee for the robot's convergence to the goal region. This is due to the potential for the robot to become trapped in local minima, hindering progress towards the ultimate objective. To address this challenge, random walks are often incorporated into the algorithmic framework, enabling the robot to explore alternative paths and escape local minima, thus enhancing the robustness and effectiveness of the motion planning process.

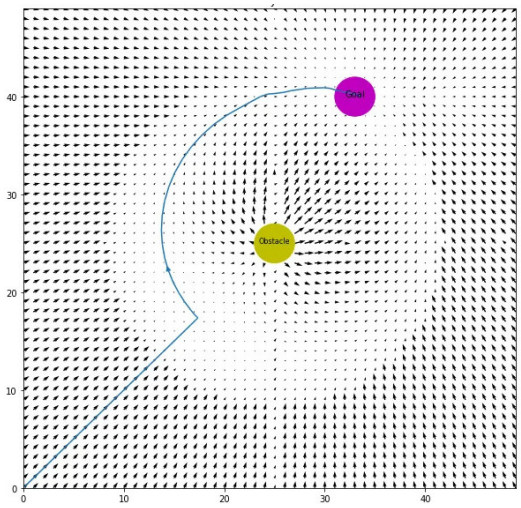


Figure 1.6: Solving motion planning query using potential field [3].

1.5.2 Grid-Based Planning

The earliest exact methods for computing navigation functions were designed for straightforward environments with obstacles of specific shapes. These methods faced limitations when applied to more complex scenarios. To address this challenge, approximate methods emerged, leveraging discretized spaces such as grids. While these approximate navigation functions provided a solution, they introduced exponential computational complexity as the number of dimensions in the state space increased. Referred to as numerical navigation functions, these approximations find utility in mobile robotics applications. Mobile robots, operating within lower-dimensional configuration spaces, benefit from the robust and efficient solving

of motion planning problems using these numerical navigation functions[20].The figure 1.7 bellow illustrates an example of A-Star algorithm in a discrete world.

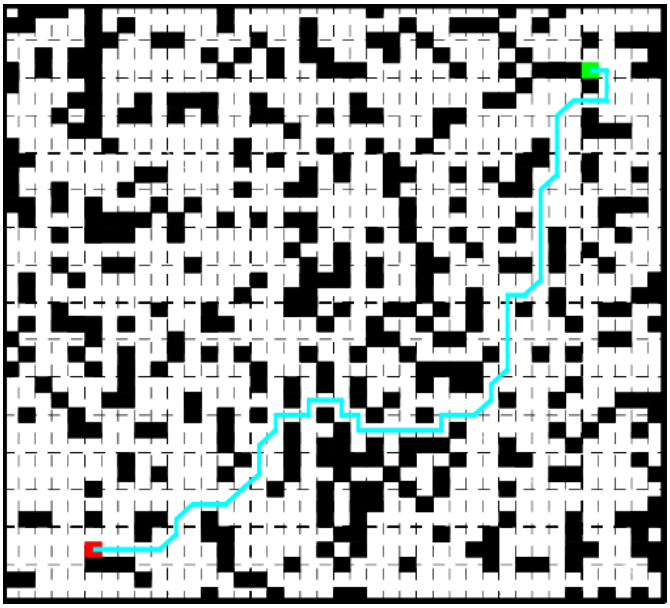


Figure 1.7: Path planned by A-Star for randomly generated world [4].

1.5.3 Sampling-Based Planning

Sampling-based planning is a general approach that has been shown to be successful in practice with many challenging problems. It avoids the exact geometric modeling of the C-space but it cannot provide the guarantees of a complete algorithm. Complete and exact algorithms are able to detect that no path can be found. Instead, sampling-based planning offers a lower level of completeness guarantee.[19]

Indeed, sampling-based motion planners can be categorized into two main types: Single-query planners and Multiple-query planners.

Multiple Query planners

The planners construct a roadmap as depicted bellow in figure 1.8, an undirected graph $\mathcal{G}$ that is precomputed once so as to map the connectivity properties of C-free. After this step, multiple queries in the same environment can be answered using only the constructed roadmap.

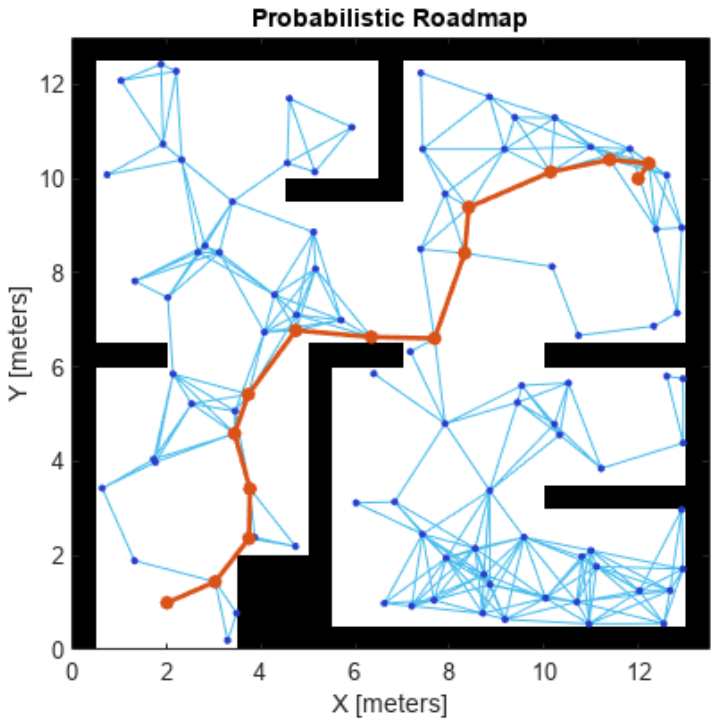


Figure 1.8: Planning with PRM planner [5].

Single Query planners

Planners in this category build a tree data structures on the fly given a planning query. They attempt to focus on exploring the part of the C-space that will lead to solving a specific query as fast as possible by biasing the tree toward the goal region, such as Rapid-Exploring Trees (RRT's) and a lot of its variants. However, there exists some other variants that are made to efficiently expand in the less explored regions in the C-space, by introducing a *voronoi bias* such as in Expansive Space Trees (EST's).

RRT

The Rapidly Exploring Random Tree (RRT) algorithm is a widely-used method in robotics motion planning. It iteratively builds a tree from an initial configuration towards random samples, efficiently exploring high-dimensional C-spaces and navigating complex environments. RRT's speed and adaptability make it a popular choice for tasks like path planning and obstacle avoidance.

Algorithm 1 RRT Algorithm

```
V ← q_init                                          ▷ Initialize vertices
E ← ∅                                               ▷ Initialize edges
for i = 1 to k do
    q_rand ← SampleRandomNode()
    q_near ← Nearest(T = (V, E), q_rand)
    q_new ← Steer(q_near, q_rand)
    if NoObstacle(q_near, q_new) then
        V ← V ∪ {q_new}                             ▷ Add new vertex
        E ← E ∪ {(q_near, q_new)}                   ▷ Add new edge
    end if
end for
return T = (V, E)                                   ▷ Return tree
```

SampleRandomNode() : This function returns a random configuration from the free C-space, there exists many ways for sampling but the most common is uniform sampling.

Nearest(`tree T,configuration q`**)** : This function takes a tree `T` and an arbitrary configuration `q` as inputs and returns the nearest configuration to the given one within the tree.

Steer`(configuration q1,configuration q2)` : This function takes two configurations as input and generates q_{new} that will be added to the tree by interpolating between q_{rand} and q_{near}.

NoObstacle`(configuration q1,configuration q2)` : The function checks if two configurations collide and return true if there is no collision, or false otherwise.

Over time, significant advancements have been made in sampling-based algorithms, incorporating the concept of cost to enable planning for optimal solutions. This evolution has led to algorithms like RRT* that optimize paths based on defined cost metrics, enhancing efficiency and performance in motion planning.

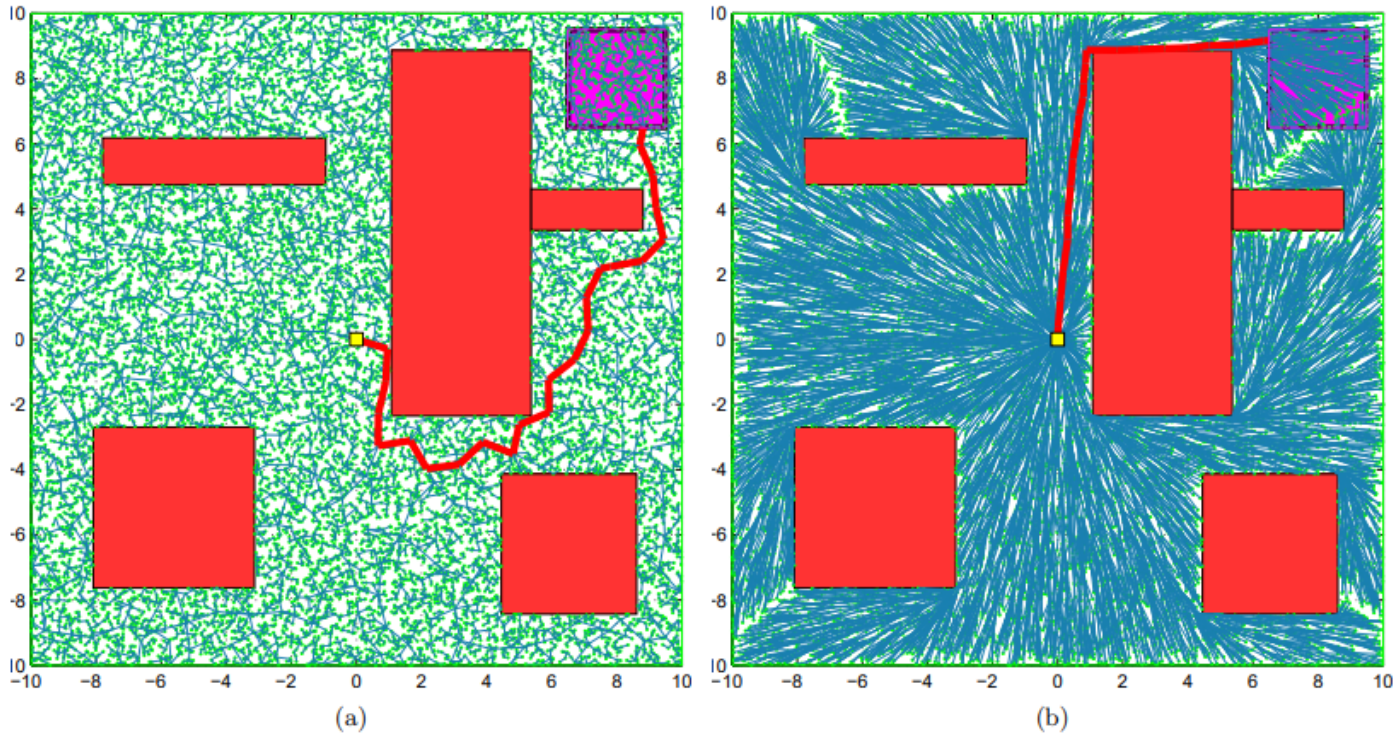


Figure 1.9: Performance Comparison of RRT and RRT* Algorithms in Simulation with Obstacles [6].

RRT algorithm (shown in (a)) with the RRT* algorithm (shown in (b)) using a simulation example with obstacles. Both algorithms were executed with the same sample sequence for 20,000 samples. The cost of the best path in RRT was 21.02, while in RRT* it was 14.51. This comparison provides a quantitative measure of the performance difference between the two algorithms in terms of path quality [6].

1.6 Car-like Robots

The car-like robot represents one of the simplest nonholonomic vehicles, showcasing fundamental characteristics and the challenging maneuverability in higher-dimensional systems. This mobile vehicle features two front wheels capable of parallel steering, while its two rear wheels remain fixed and parallel. Unlike omnidirectional robots, car-like robots cannot execute sideways movement or rotate in place, but can be conceptualized as a three-dimensional system, The motions of car-like robots are constrained by non-slipping and curvature limitations. Their minimal length paths are typically composed of a finite sequence of arcs of circles and straight-line segments. The car that is capable of moving forward and backward is called a Reeds-Shepp car [2] and can be modeled as the following :

$$\begin{cases} \dot{x} = u_1 \cos\theta \\ \dot{y} = u_1 \sin\theta \\ \dot{\theta} = u_1 u_2 \end{cases} \tag{1.3}$$

Where $u_1 \in \{-1, 1\}$ and $u_2 \in [-\tan\phi_{\max}, \tan\phi_{\max}]$ In this representation, $\phi_{\max}$ represents the maximum steering angle.

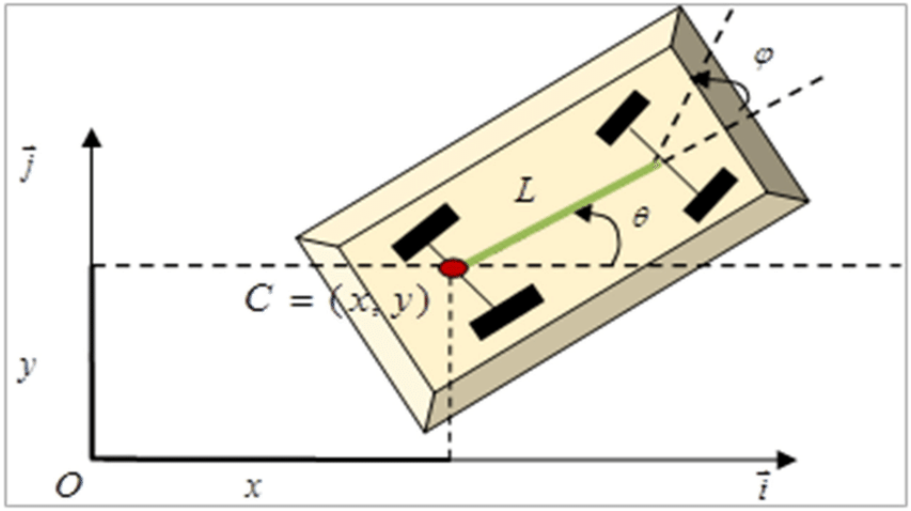


Figure 1.10: model of the car-like robot. [7].

1.7 Reeds-Shepp curves

Reeds-Shepp curves are a form of path used in robotics and vehicle navigation to find the shortest path between two points in a plane with curvature and movement direction constraints. These pathways are especially useful for car-like vehicles that have movement constraints, they consist of circular segments at the minimum turning radius, straight-line segments, and direction reversals, figure 1.11 below is an illustration of Reeds-Shepp curves.

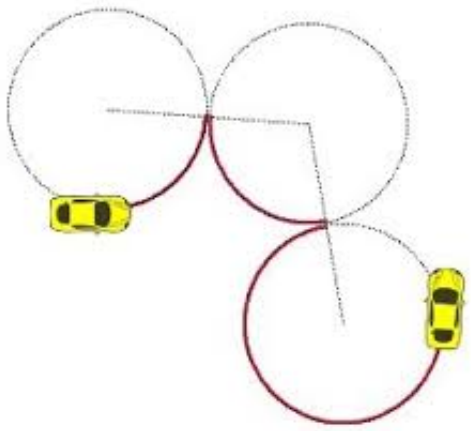


Figure 1.11: Reeds-Shepp curves

1.8 Geometric path tracking algorithms

In the area of automobile software, a path-tracking controller is an essential component for autonomous vehicles. It serves as the link between navigation systems and the vehicle's execution mechanisms. The primary objective of the path tracking algorithm is to generate actuator commands that minimize the lateral distance between the vehicle's center of mass and the geometric path estimated by the motion planner[21].

Before starting with the algorithms it is useful to understand the bicycle car model approach. Which is a common simplification used for path tracking. This model simplifies the four-wheel car by combining the two front wheels together and the two rear wheels together to form a two-wheeled model, like a bicycle(see figure 1.12).

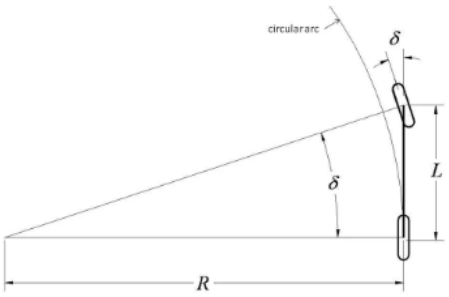


Figure 1.12: Bicycle model[8].

It is evident from the figure above

$$\tan(\delta) = \frac{L}{R} \tag{1.4}$$

Here R is the radius of the curvature and L is the wheelbase.

1.8.1 Pure pursuit controller

This technique calculates the required steering angle to keep a wheeled robot on a predefined path, Assuming that the linear velocity remains constant[22]. The task of the pure pursuit algorithm is to identify the curvature enabling a robot to move from its current location to some specified target location. The main objective of this algorithm is to select a destination point ahead of the robot at a predetermined distance on the path, as depicted in figure 1.13. Essentially, the robot is pursuing a point on its way lying in front in front of it while maintaining a given distance between them. This distance is known as the *lookahead distance* and represents the chord length of *the arc* relating the current location to the target position[23].

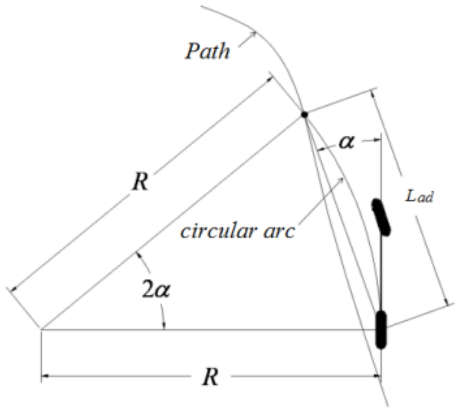


Figure 1.13: Geometric explanation of PP[9].

The set of equations relates the curvature to the lookahead distance. The algebra is straightforward and requires no further explanation[10].
the law of sine is satisfied and is given as:

$$\frac{L_{ad}}{\sin(2\alpha)} = \frac{R}{\sin(\frac{\pi}{2} - \alpha)} \tag{1.5}$$

$$\frac{L_{ad}}{2\sin(\alpha)\cos(\alpha)} = \frac{R}{\cos(\alpha)} \tag{1.6}$$

$$\frac{L_{ad}}{\sin(2\alpha)} = 2R \tag{1.7}$$

The formula of robot radius of rotation can be simplified as

$$R = \frac{1}{2}\frac{L_{ad}}{\sin(2\alpha)} \tag{1.8}$$

where R is the radius of the circle that the rear axle will travel along at the given steering angle, L_{ad} is the look ahead distance and α is the angle between the vehicle's heading vector and the look ahead vector. The curvature formula is as follows

$$\kappa = \frac{2\sin(\alpha)}{L_{ad}} \tag{1.9}$$

where κ is the curvature of the circular arc.

A geometric relationship between the turning radius and the curvature taken from 1.4 can be expressed as

$$\delta = \tan^{-1}(\kappa L) \tag{1.10}$$

where L is the distance between the front axle and rear axle (wheelbase).
The pure pursuit control law is given as

$$\delta = \tan^{-1}(\frac{2L\sin(\alpha)}{L_{ad}}) \tag{1.11}$$

1.8.2 Stanley controller

Unlike pure pursuit which measures distances from the rear axle position. Stanley controller uses position of the front axle. The Stanley method computes the steering command based on a non-linear control law that considers the cross-track error e_{fa} (the distance between the vehicle to the nearest path point (c_x, c_y)) (see figure1.14), as well as the heading error θ_e (the difference in orientation between the vehicle and the path)[10].

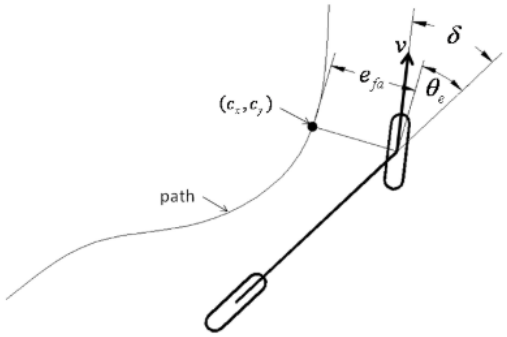


Figure 1.14: Stanley Algorithm Vehicle Model[10].

There are three intuitive steering laws of Stanley method,

1. Keeping the wheels aligned with the given path by setting $\delta(t) = \theta_e(t)$.
   $\theta_e = \theta - \theta_p$, where θ is the heading of the vehicle and θ_p is the heading of the path at (c_x, c_y).

2. Adjusting δ such that the intended trajectory intersects the path tangent from (c_x, c_y).The

steering angle can be corrected as follows,

$$\delta(t) = \tan^{-1}(\frac{ke_{fa}(t)}{v_x(t)}) \tag{1.12}$$

where k is a gain parameter, $v_x(t)$ is the velocity.

3. Obey the max steering angle bounds. That means $\delta(t) \in [\delta_{min}, \delta_{max}]$

   So we can arrive at

$$\delta(t) = \theta_e(t) + \tan^{-1}(\frac{ke_{fa}(t)}{v_x(t)}) \tag{1.13}$$

Chapter 2

Feedback Motion Planning

This chapter aims to explore feedback motion planning, where traditional planning algorithms are improved with feedback properties to help robots adjust their trajectories during execution. This chapter examines different techniques related to feedback motion planning, including artificial potential fields, cell decomposition, and sampling-based neighborhood graphs, after mentioning the key tools of navigation functions. Each of the provided techniques offer a clear understanding of how robots navigate through complex environments. Additionally, two prposed feedback motion planning algorithms are presented in detail: the modified RRT* algorithm and the funnels-graph-based approach.

2.1 Classification of feedback motion planning

In the realm of robotics, motion planning algorithms serve as the backbone for guiding robotic systems through their operational tasks. Ordinary planning algorithms focus on computing *open-loop* plans, which means overlooking any errors that might occur during execution of the plan, yet the plan will be executed as planned. Conversely, feedback motion planning algorithms introduce adaptability by incorporating real-time feedback from the environment leading to a *closed-loop* plan that responds to unpredictable events during execution, allowing robots to dynamically adjust their trajectories and attempting to follow the computed path as closely as possible.

Feedback is necessary due to the unpredictable nature of future states, In most problems involving physical world, this implies that a plan's activities should depend on informations obtained during execution. The book[2] mentioned two ways to model uncertainty in the predictability of future states:

Explicit feedback : involves developing models that clearly account for probable ways in which the actual future state can diverge from the planned future state, its purpose lies in directly considering uncertainty during execution and incorporating it into the planning model. Suppose we are planning a robot's path, and we know that external disturbances (such as wind or friction) can affect the robot's actual trajectory. An explicit feedback approach would explicitly model these disturbances and adjust the planned path accordingly.

Implicit feedback : presupposes that the model of state transitions suggests that no uncertainty exists. However, a feedback plan is created to ensure that it knows which action to take in case it encounters an unexpected state during execution. This technique provides a safety net by having an emergency plan even when the system's behavior is assumed to be deterministic. Imagine a self-driving car following a planned route. Implicit feedback would create a backup plan for unexpected situations even though the primary plan assumes deterministic behavior. This approach is adopted in this project.

Implicit feedback is chosen to address uncertainty due to the proven effectiveness of the control approach in various applications throughout the past century. Additionally, explicit uncertainty modeling necessitates stronger requirements for creating reliable models, which can complicate algorithm design and raise the risk of modeling errors.

2.2 Vector fields and integral curves

To study feedback motion plans across continuous state spaces, including configuration spaces, we must first establish a vector field and then integrate the vector field from an initial point to obtain the trajectory. A vector field is appropriate for describing a feedback plan across a continuous state space.

Vector spaces Before defining a *vector field*, it is helpful to be precise about what is meant by a *vector space*, a *vector space* is a fundamental algebraic structure constructed from a non-empty set V whose elements are called *vectors*, a *field* F whose elements are called *scalars*, two binary operations *vector addition* and *scalar multiplication* satisfying a set of axioms, The vector space is denoted $V(F)$ meaning the vector space V over the field F. We are concerned mostly with the real space $\mathbb{R}^n(\mathbb{R})$.

Vector fields A *vector field* as indicated in figure 2.1, is an assignment of a vector to each point in a space, most commonly Euclidean space $\mathbb{R}^n$. A *vector field* on a plane can be visualized as a collection of arrows with given magnitudes and directions[11].

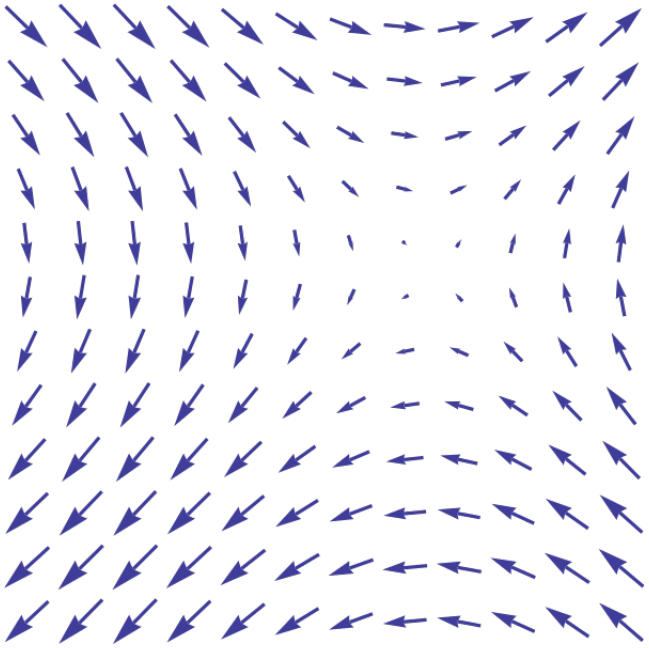


Figure 2.1: A portion of a vector field. [11].

Vector fields serve as a representation of a feedback plan across a continuous state space, offering guidance on the robot's movement at every point within its configuration space. More precisely, at any specific configuration, the vector field assigns a vector that signifies the direction and magnitude of motion for the robot.

Integral curves An *integral curve* is a trajectory that follows the direction provided by a vector field. *The integral curves* guarantee collision-free paths because they follow the

vector field, which considers obstacle avoidance. An example of an integral curve is shown in figure 2.2.

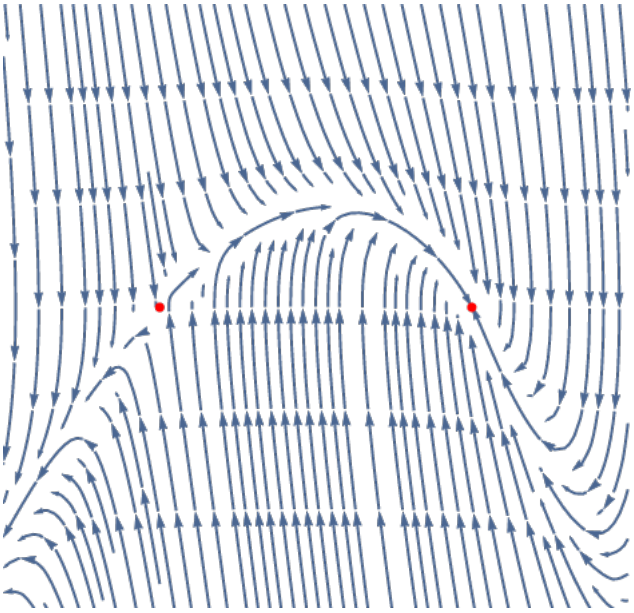


Figure 2.2: Integral curves of a vector field. [12].

In essence, we've unpacked the key tools for navigating robots in complex environments. Robots are able to find secure paths following continuous guidance from vector fields, which assign direction and velocity to the robot at every state. Following these vectors leads to collision-free paths, as the integral curves, representing the robot's trajectory, are guaranteed to avoid obstacles. This paves the way for further exploration of how the feedback motion planning algorithms work.

2.3 Common feedback motion planning techniques

In this section, we introduce some common techniques for constructing smooth feedback laws over configuration spaces for robot navigation.

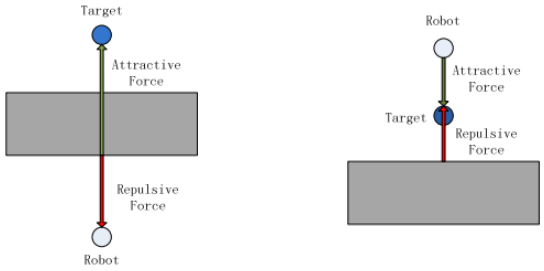
Artificial potential field The APF method's foundational idea declares the robot as a point moving through an abstract force field. The abstract force field has two parts: an attractive field generated by the target point, which directs the robot towards the target;

and an obstacle-repellent field. The repulsive field is a synthesis of the repulsive forces exerted by these obstacles, and it pushes the robot away from them. Consequently, the potential function (eq.2.1) represents the APF of the robot, which is the combined effect of the attractive and repulsive fields. The robot controls its movement towards the target point by following the direction of the APF. By employing the APF method, the robot can find a collision-free path by searching for a route along the decline direction of the potential function.

$$U(q) = U(q)_{att} + U(q)_{rep} \tag{2.1}$$

Where: $U(q)$ is the artificial potential field. $U(q)_{att}$ is the attractive field. $U(q)_{rep}$ is the repulsive field.

Despite the effectiveness of the traditional APF method in planning smooth paths, it is plagued by significant issues. In situations where the attractive and repulsive forces are nearly equal and collinear but in opposite directions during the movement towards the target, the robot experiences a potential force of zero. This leads to the robot becoming trapped in local minima and experiencing oscillations (figure 2.3a). Additionally, when the target is positioned very close to obstacles, the robot is unable to reach its intended destination (figure 2.3b).



(a) (b)

Figure 2.3: Local minima. [13].

Cell decomposition This method employs breaking the 2D environment into simple cells, building local controllers on the portion of state space existence assigned to each cell, and then combining these local controllers[24]. Moreover, the strategy of this algorithm is as follows:

1. Firstly, the environment is decomposed into convex cells. Then a discrete plan is computed over the cell connectivity graph.

2. Each cell and each face separating adjacent cells should be defined by a local controller (vector fields).

3. A global controller needs to be constructed by interpolating between vector fields defined in each cell.

The example in figure 2.4 illustrates the cell decomposition algorithm.

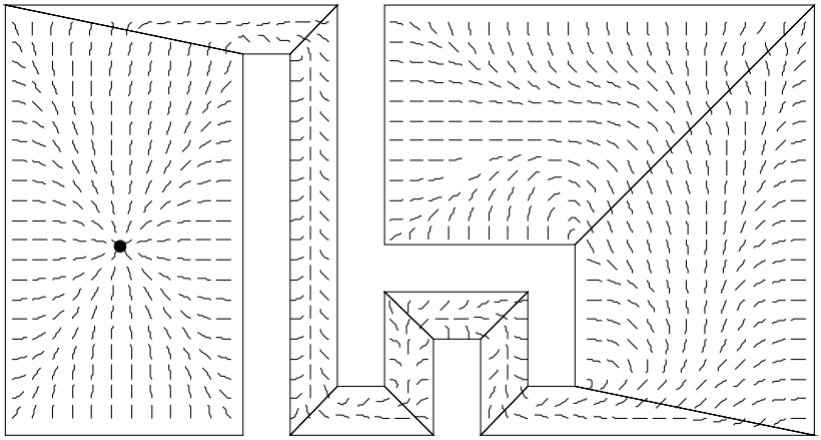
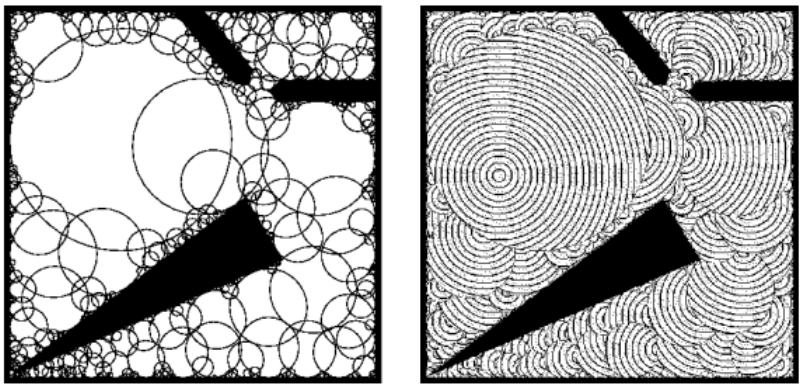


Figure 2.4: A smooth vector field generated by the cell decomposition algorithm [14].

Sampling-based neighborhood graph This algorithm proposes a sampling-based approach for generating feedback motion strategies as illustrated in figure2.5[15].

The SNG construction algorithm incrementally places new neighborhoods in the configuration space using the distance information provided by existing collision detection algorithms. In other words, The key idea of this algorithm is to fill the collision-free subset of the configuration space with overlapping neighborhoods such as balls. The free-space filling termination condition suggests the probability that a specified portion of the space is covered. The task succeeding the neighborhoods construction is to compute a *navigation function* which is a potential function that has only one *local minimum*, which is at the goal configuration. Finally, the feedback motion strategy is executed using the constructed navigation function.



(a) Coverage of $\mathcal{C}_{free}$ (b) A navigation function defined over it

Figure 2.5: The configuration space is covered by overlapping neighborhoods on which navigation functions are defined[15].

According to the implementation done by [15], the Sampling-Based Neighborhood Graph algorithm provides two important advantages. Firstly, it is well-suited for problems involving up to six degrees of freedom (DOFs). Secondly, even on a standard personal computer (PC), the SNG can be constructed within seconds, in addition to that feedback motion strategies can be executed with a response time of several microseconds.

This section explored three prominent techniques for constructing smooth feedback laws for robot navigation: artificial potential fields, cell decomposition, and sampling-based neighborhood graphs.

- **APF** offer a simple and adaptable approach for dynamic environments but suffers from local minima.
- **Cell decomposition** simplifies obstacle representation but may struggle in high-dimensional spaces.
- **SNG** provides efficient planning and fast execution, particularly for robots with up to six degrees of freedom.

2.4 Proposed feedback motion planning algorithms

Having provided the theoretical basis for feedback motion planning, we can now focus on the practical part of the project. Specifically, this section reveals the two algorithms employed in enabling our robotic device to navigate well. Initially, we will consider each algorithm exhaustively before providing a detailed explanation of its pseudo-code as well as implementation details relating to the same. Consequently, we shall then grasp how these algorithms work.

2.4.1 Feedback motion planning based on modified RRT* algorithm

In this section, we describe one of the algorithms we developed in our project, a feedback motion planning approach built upon a modified RRT* (Rapidly exploring Random Tree Star) path planner. This approach addresses the challenge of navigating a car-like robot in an obstacle environment with potential noise or disturbances.
The algorithm operates in a two-step process:

1. Optimized Path Generation:

    - A modified RRT* algorithm is employed to generate a collision-free path. This modified RRT* prioritizes finding a path with minimum distance along with ensuring safety and smooth navigation.
    - The path is constructed from a tree rooted at the goal configuration, considering the robot's non-holonomic constraints.

2. Real-Time Adaptation with Feedback:

    - A pure pursuit controller takes over, utilizing the generated path to compute the optimal velocity and steering angle commands for the robot.
    - When noise is introduced to the robot, the feedback mechanism activates, using the robot's current position to select a new path from the modified RRT* gener-

ated tree. This new path reestablishes the connection between the robot and the original goal, ensuring it remains on course.

By combining the strengths of a modified RRT* for efficient path planning with real-time feedback control, this algorithm offers a robust and adaptable solution for navigating car-like robots in the presence of noise.

Path planner working principle:

Algorithm 2 ModifiedRRT* Path Planning

Inputs: GoalState, InitialState, MaxIterations, MaxNodes
Outputs: Tree, Path (if found)
$T \leftarrow$ InitializeTree(GoalState) ▷ Initialize the tree with the goal state
PathFound $\leftarrow$ **false** ▷ Pre-loop setup
Iterations $\leftarrow 0$
while Iterations $<$ MaxIterations and $|T| <$ MaxNodes **do**
    RandomState $\leftarrow$ SampleFromSpace() ▷ Sample a state uniformly
    T $\leftarrow$ ExtendTree(RandomState, T)
    if StartReached **then**
        PathFound $\leftarrow$ **true**
    end if
    if Max nodes reached **then**
        break
    end if
    Iterations $\leftarrow$ Iterations + 1
end while

The following term need to be defined first:

- Collision Detector: is an object that checks if a given state (position and orientation) of the robot doesn't collide with any obstacles in the map and that it adheres to the robot's constraints. Essentially, ensures that the robot remains within permissible and safe configurations throughout its path.

Algorithm 3 ExtendTree Algorithm

Inputs: Tree, State
Outputs: Tree
$x_{\text{nearest}} \leftarrow \text{Nearest}(Tree, x_{\text{state}})$ ▷ Identify the closest node
$d \leftarrow |x_{\text{nearest}} - x_{\text{state}}|$ ▷ Compute the distance
if $d \leq \epsilon$ **then**
    return $Tree$
end if
if $d > \delta_{\max}$ **then**
    $x_{\text{new}} \leftarrow x_{\text{nearest}} + \frac{\delta_{\max}}{d}(x_{\text{state}} - x_{\text{nearest}})$ ▷ Interpolate new state
else
    $x_{\text{new}} \leftarrow x_{\text{state}}$
end if
$c_{\text{new}} \leftarrow \text{CalculateCost}(x_{\text{new}}, x_{\text{nearest}})$ ▷ Calculate the new cost
if !CheckReedsSheppPath$(x_{\text{nearest}}, x_{\text{new}})$ **then**
    return $Tree$
else
    $\mathcal{N}(x_{\text{new}}) \leftarrow \text{NearbyNodes}(x_{\text{new}})$ ▷ Find nearby nodes
end if
for each $x_{\text{near}} \in \mathcal{N}(x_{\text{new}})$ **do**
    $c_{\text{tentative}} \leftarrow c(x_{\text{near}}) + \text{cost}(x_{\text{near}}, x_{\text{new}})$ ▷ Calculate tentative cost
    if $c_{\text{tentative}} < c_{\text{new}}$ **and** CheckReedsSheppPath$(x_{\text{near}}, x_{\text{new}})$ **then**
        $c_{\text{new}} \leftarrow c_{\text{tentative}}$
        $x_{\min} \leftarrow x_{\text{near}}$
    end if
end for
Insert$(x_{\text{new}}, Tree, x_{\min})$ ▷ Insert the new node into the tree
if NewNodeCount$(Tree) \geq N_{\max}$ **then**
    return $Tree$
end if
if NewNodeReachedStart(x_{new}) **then**
    return $Tree$
end if
return $Tree$ ▷ Node successfully added

- **ModifiedRRT*** algorithm:
  - Initializes a tree with the goal state.
  - Iterates to add new nodes, aiming to find a path from the goal state to the start.
  - Samples random states and attempts to extend the tree.
  - Checks if the start state is reached.
  - Ultimately extracts and returns the tree and the path if found.
- **ExtendTree**:
  - Identifies the nearest existing node.
  - Computes the distance to the sampled state.
  - For larger distances, interpolates a new state.
  - Ensure the path conforms to non-holonomic constraints and is collision-free.
  - If the path is valid we try to optimize the tree around the new node.

Motion planning process :

Algorithm 4 Modified RRT* navigation function

Inputs: Tree, Path (if found)
if path_exists **then**
    controller.waypoints ← path
else
    path ← assign_new_path(robot_initial_condition)
    controller.waypoints ← path
end if
while distance > goal_radius **do**
    $[u_1, u_2]$ ← controller.CalculateControls(current_state)
    current_state ← execute_action(u_1, u_2)
    distance ← distance_to_goal(current_state)
    error ← CalculateError(current_state)
    if error > threshold **then**
        path ← assign_new_path(current_state)
        controller.waypoints ← path
    end if
end while

- If there is a path, assign it directly; otherwise, assign one from the generated tree.
- While the goal is not reached:
  - Compute actions and execute them.
  - If the error is greater than the threshold, assign another path.

2.4.2 Funnels-graph based feedback motion planning algorithm

In this section, the second feedback motion algorithm proposed in our project is introduced. As the name suggests, the path-planning process in this algorithm depends on a graph data structure to represent a network of funnels ensuring the discovery of a feasible path between the start and goal configurations and an efficient coverage of the free state space which enables the robot to navigate a collision-free path to the goal configuration from any start configuration. The figure 2.6 below is an illustration of a funnel graph:

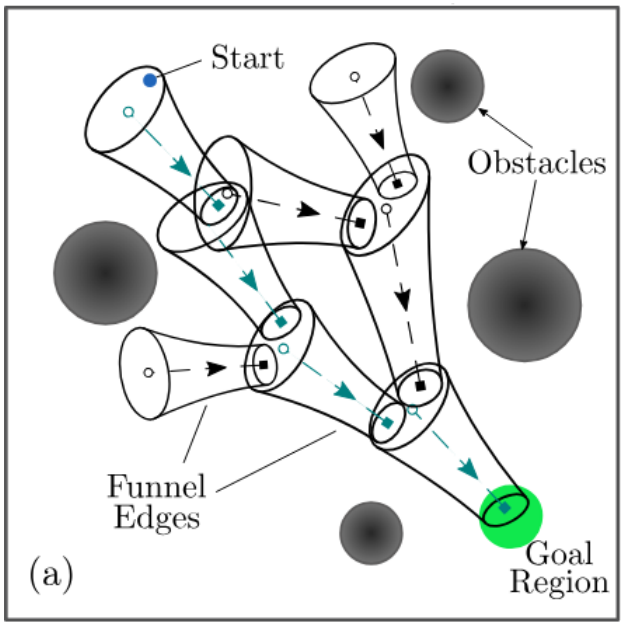


Figure 2.6: Example of a funnel-graph[16].

Path planner working principle:

Algorithm 5 Funnels_Path_Planning

Inputs: goal state, initial state, coverage threshold
Outputs: Graph , Path
Cfree ← `estimateCfree()` ▷ Estimate the obstacle-free space
G ← `init_graph(goal_state)` ▷ Initialize the graph with the goal state
covered_area ← `estimate_Coverage(goal_state)`
while True **do**
    new_funnel ← `CreateFunnel(G)` ▷ Generate a new state
    if new_funnel is Empty **then**
        `continue` ▷ Skip to the next iteration
    end if
    covered_area ← `estimate_Coverage(new_funnel)` ▷ Estimate the covered area
    if `is_start_reached(new_funnel)` **then**
        solution_flag ← True
    end if
    if solution_flag **and** (covered_area ≥ coverage_threshold) **then**
        Path ← `set_path()` ▷ Set the final path
        `break` ▷ Exit the while loop
    end if
end while

Algorithm 6 CreateFunnel

Inputs: Graph
Outputs: Node
state ← SampleRandomState()
if CheckIfInObstacle(state) **then**
    return Empty
end if
neighborhood ← EstimateObstacleFreeRadius(state)
potential ← $1/(\text{neighborhood} + \epsilon)$
if $U[0,1] \geq e^{-\text{potential}}$ **then**
    return Empty
end if
if CheckIfInNeighborhood(Existing_Funnels, neighborhood, state) = true **then**
    return Empty
end if
if CheckIfIntersects(Existing_Funnels, neighborhood, state) = false **then**
    return Empty
end if
G ← AddNode(state,neighborhood)

- The **Funnel Graph** Algorithm:
  - Estimates the available obstacle free area
  - Initialize the graph and estimates the initial coverage
  - Create Funnels and estimate the coverage
  - Set the path whenever the start state and the desired coverage are both reached by running a Dijkstra algorithm on the graph.
- **Create Funnel** :
  - Sample a random state within tha limits of the map.
  - Check if the state is not colliding with the obstacles.
  - Estimate the obstacle free neighborhood of the state.
  - Sample rejection to bias the sampling toward large areas in the state space to minimize the number of Funnels.
  - Ensure that the sample is not in the neighborhood of the existing Funnels
  - Ensure that the sample's neighborhood is at least intersecting with at least an existing Funnels to guide the expansion in the environment.
  - Add the new node (state, neighborhood) as a new funnel in the graph, with edges based on the distances between the centers of intersecting neighborhoods.

Coverage criteria: In the context of the described algorithm, the coverage criteria is fundamental for evaluating how effectively the algorithm explores the map. The coverage is assessed by determining the area of the free state space then whenever a new node is added we re-evaluate the coverage until the threshold is met, this criteria was implemented by creating an empty map that has the same dimensions of the first one and whenever a new node is added to the graph, all the space that falls within the obstacle-free radius of the node is set to occupied then we estimate the newly occupied area as a ratio from the free state space.

Motion planning process description:

Algorithm 7 Robot navigation function

```
1: Inputs: Graph, Path
2: path←localise_and_set(intial_state)
3: controller.Waypoints ← path                    ▷ Pass reference path to the controller.
4: while distance > goal_radius do
5:     [u1, u2] ← controller.CalculateControls(current_state)
6:     current_state ← execute_action(u1, u2)
7:     distance ← distance_to_goal(current_state)
8:     if not is_in_path_funnels(current_state) then
9:         path←localise_and_set(current_state)    ▷ Define new path to the goal state
10:        controller.Waypoints ←path
11:    end if
12: end while
```

1. Set the path corresponding to the initial condition

2. While the goal is not reached:

   - Compute actions and execute them.
   - If the Robot is out of the path funnels assign a new path.

Chapter 3

Simulation and Results

3.1 Introduction

The purpose of this chapter is to present the simulation results of the feedback motion planning algorithms introduced in the previous chapter, it also evaluates the performance of two distinct methods: the Tree-Based Method and the Funnel-Graph-Based Method. The evaluation is based on different scenarios designed to test the robustness, adaptability, and efficiency of each method.

3.2 Mathematical model of the car-like robot

The car-like robot is modeled using a kinematic bicycle model, which captures the essential dynamics required for motion planning. This model is chosen for its simplicity and ability to represent the motion of a wide range of wheeled robots. The state of the robot is described by (x, y) for position, θ for orientation, ϕ for steering angle, v for velocity, and L is the length of the robot's wheelbase.

$$\begin{cases} \dot{x} = v\cos(\theta) \\ \dot{y} = v\sin(\theta) \\ \dot{\theta} = \frac{v}{L}\tan(\phi) \end{cases} \tag{3.1}$$

Where the control inputs to the robot are the linear velocity v and the steering angle ϕ. The intuitive linear velocity and steering angle control inputs make it easy to translate desired

trajectories into control commands, The robot is assumed to have a wheel base of 0.4 meters and a maximum steering angle of 45 degrees, which leads to a minimum turning radius of 0.8 meters according to the formula derived in [17]. 3.1 is an illustrative figure.

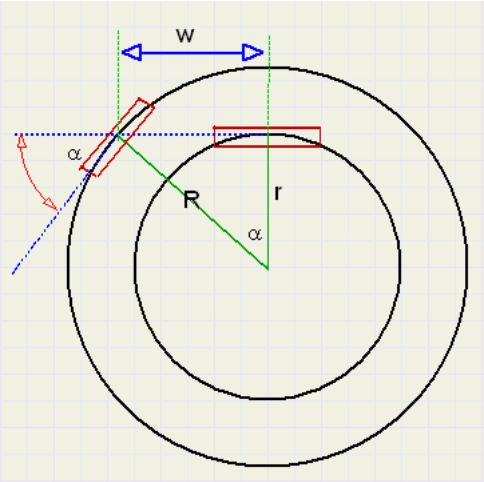


Figure 3.1: Turning Radius Calculation.[17]

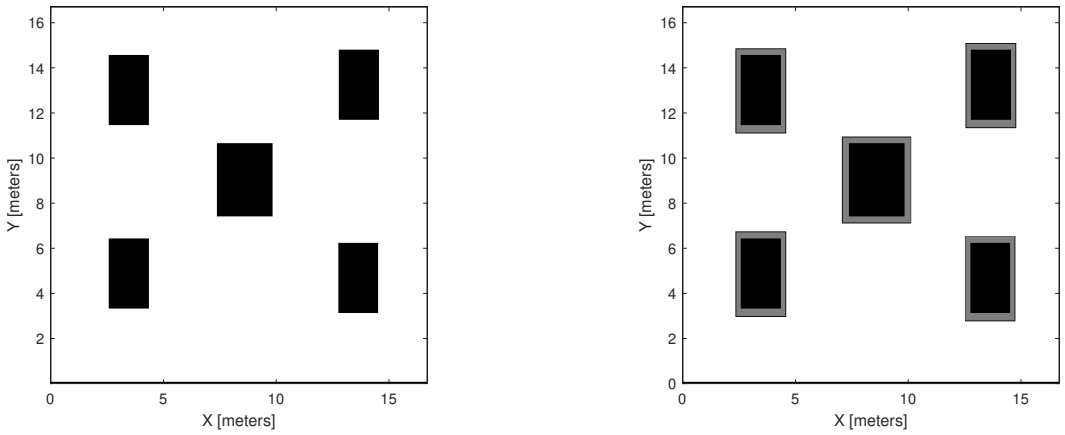
$$R = \frac{L}{\sin(\alpha)} \tag{3.2}$$

3.3 Map Construction

Before executing our algorithm, it is crucial to ensure that the environment is accurately modeled to guarantee the algorithm's effective performance. We will model the environment as a 2D occupancy grid, where each grid cell contains the occupancy probability of that region of the map, represented as values ranging from 0 to 1. This probabilistic representation allows us to handle uncertainties in the environment, enabling more robust motion planning and navigation.

To construct a reliable map, two main factors must be considered. First, the map resolution needs to be carefully chosen to balance detail and computational efficiency. Second, since both of our motion planning methods rely on sampling-based algorithms, which do not inherently account for the robot's size during sampling, it is essential to prevent sampling near obstacles to avoid invalid configurations. This can be achieved by effectively inflating

the obstacle regions, thereby increasing their size in the occupancy grid to ensure a safe buffer zone around them. This approach helps maintain valid and feasible paths during the planning process. In our implementation, we modeled the environment as a square map with dimensions of 16.7 by 16.7 meters with 30 grid cell per meter. Initially, we placed fixed obstacles throughout the map to represent the environment's static features, as indicated by figure 3.2a. To ensure safe navigation and valid path planning, we then added a buffer region around each obstacle, and the passages in the environment are always larger than the robot size see figure 3.2b.



(a) Original Map (b) Inflated Obstacles

Figure 3.2: Comparison of the map with original and inflated obstacles.

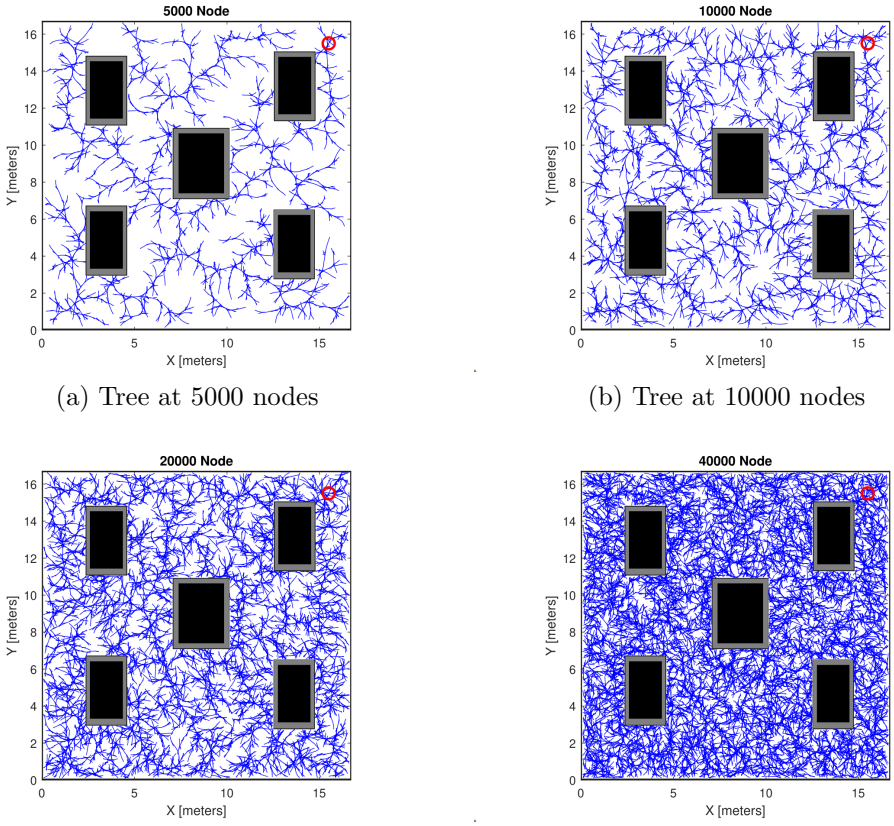
During this chapter, we assumed a fixed map for all experiments to ensure consistency and comparability of the results. This fixed map allowed us to standardize the testing environment, enabling a clear evaluation of our motion planning algorithms under controlled conditions.

3.4 Methods overview

Two sampling-based motion planning algorithms are developed mainly to address the problem of guiding a car-like robot from any starting position to a goal position through an obstacle environment. These algorithms generate geometric paths, which are then fed to

the pure pursuit controller to produce the appropriate control commands. Modified RRT* and funnel-based algorithms are built upon tree and graph data structures respectively. It is important to note that the tree is generated only once for a given goal configuration. In other words, the tree or graph is regenerated only when the goal configuration changes, contrariwise the graph algorithm does not require regeneration when the goal is change. The feedback plan is embedded as a part of both algorithms' control loops. The role of feedback is to ensure that the robot always converges to the goal state despite the presence of noise and regardless of the state of the robot. Both algorithms are equipped with the adaptation to environmental changes feature during execution, whereas the goal-tracking feature is added only to the funnels-based algorithm. Each of these features will be discussed in detail throughout this chapter.

- Figure 3.3 illustrates the tree generated by the modified RRT* algorithm. As described in Section 2.4.1, the tree generation process begins at the goal configuration (the red circle in the figures) with the aim of exploring the obstacle-free space. Notably, the tree continues to grow even after reaching the starting position. When possible, the tree performs optimization by re-evaluating and refining its paths to reduce the overall cost or improve efficiency. This involves checking for shorter paths or less complex trajectories between nodes and adjusting connections to achieve a more optimal solution. The figure shows also that this algorithm accounts for the non-holonomic constraints of a car-like robot when constructing the edges of the tree for different number of nodes. Since a good coverage of the free state space is needed, a large number of nodes is needed to ensure that there is always a path leading to the goal region.



(a) Tree at 5000 nodes

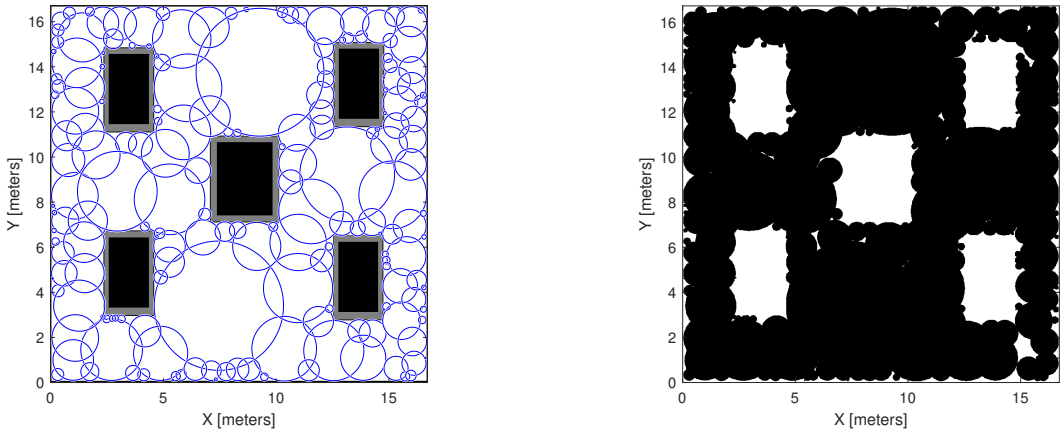
(b) Tree at 10000 nodes

(c) Tree at 20000 nodes

(d) Tree at 40000 nodes

Figure 3.3: Tree expansions at different node counts.

- The funnels-graph-based algorithm follows the same expansion pattern as the first algorithm. This algorithm holds strong coverage criteria that guides the graph expansion process along with a sampling bias that forces the sampling away from the obstacles to ensure the coverage of the free state space with an optimum number of nodes. The coverage criteria works by creating a duplicate of the original map, and each time a funnel is generated, it is simultaneously converted into an occupied space. A dedicated function continuously estimates the obstacle-free area until the coverage criteria are met. Figure 3.4 illustrates an experiment where a coverage criteria of 96 percent is used.



(a) Funnel-graph (b) Coverage estimation map

Figure 3.4: Visualization of the funnel-graph based Method.

3.5 Static planning with static goal

General description In this scenario, we test the fundamental task of guiding the robot from the start to the goal position using the two developed algorithms. This scenario is designed to evaluate the basic capability of each algorithm to successfully navigate the robot in a static environment with obstacles from any valid initial state.

This scenario is divided into three parts for each algorithm. The first part addresses the navigation task without any noise or uncertainty introduced during motion execution. In contrast, the second and the third parts incorporate actuation noise and measurement uncertainty respectively into the system, investigating the immunity of the feedback plan to different unpredictabilities. A detailed discussion of the un-predictabilities implementation and its impact will be provided in its appropriate part.

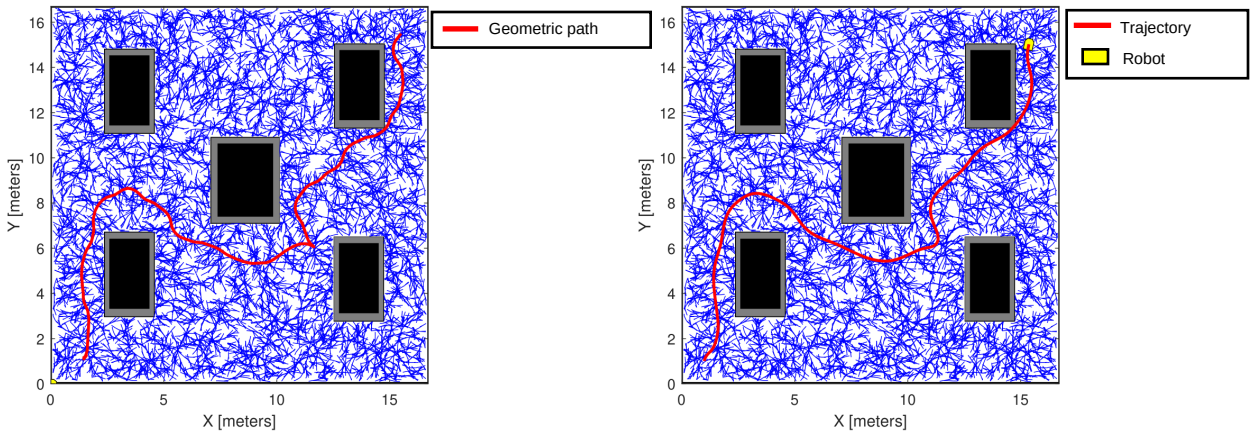
Some key parameters will be standardized for both algorithms to ensure consistency, including a goal region centered at [15.5, 15.5] with a radius of 0.5 meters and controller parameters such as a linear velocity of 1.5 meters per second, a maximum steering angle of $\frac{\pi}{4}$, and a look-ahead distance of 1 meter.

3.5.1 Planning with deterministic, time-invariant model

The experiment of part 1 will involve testing each algorithm under the same three configurations by fixing the goal position and varying the starting positions. To facilitate a fair comparison between the two algorithms, certain parameters will be recorded and presented in a table.

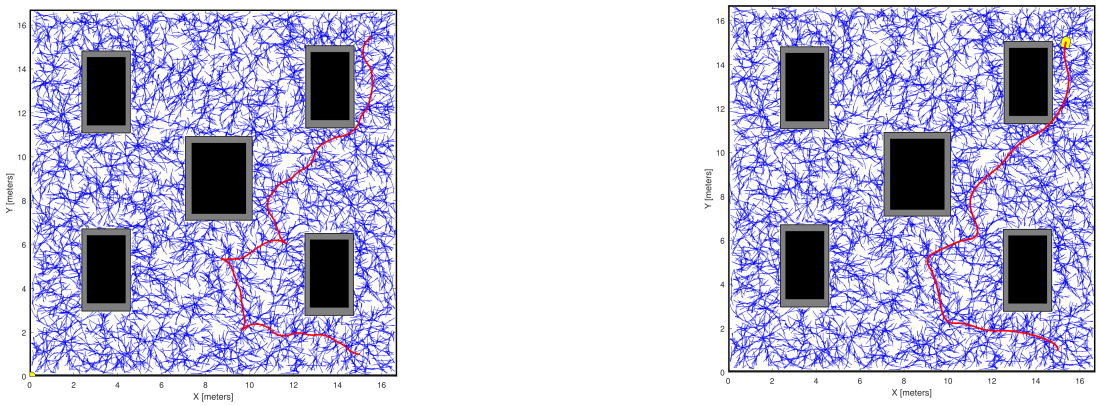
Modified RRT* algorithm

The figures 3.5, 3.6, and 3.7 provided bellow offer a visualization of the robot guiding task for various samples. The ones on the left depict the absolute geometric path represented by a set of waypoints, while the figures on the right illustrate how the robot (yellow polygon) moves over time to follow the specified geometric path.



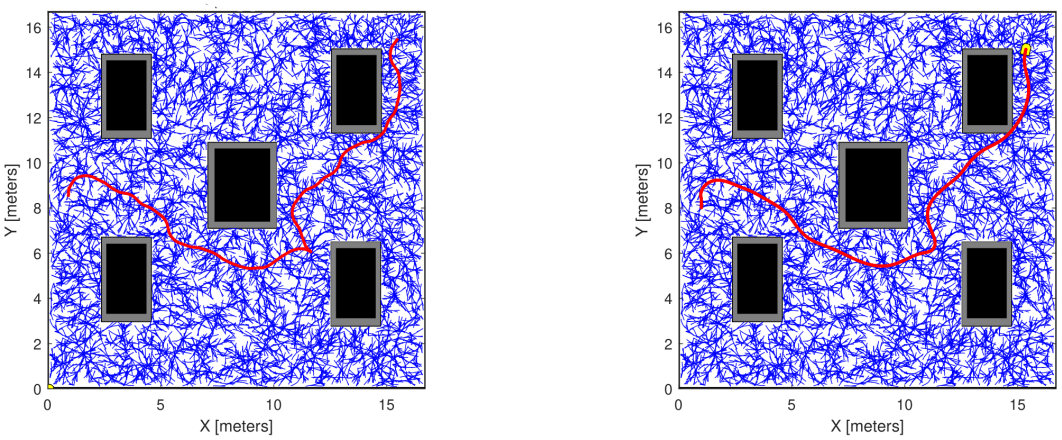
(a) Geometric path (b) Robot trajectory

Figure 3.5: Starting position [1 1 0]



(a) Geometric path (b) Robot trajectory

Figure 3.6: Starting position [15 1 pi/2]

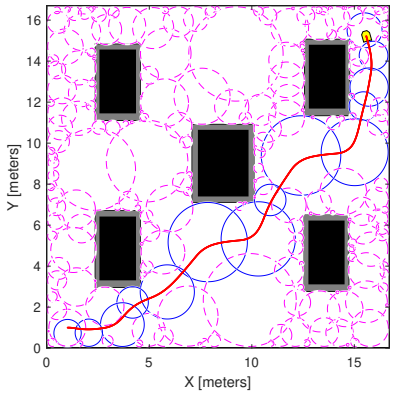


(a) Geometric path (b) Robot trajectory

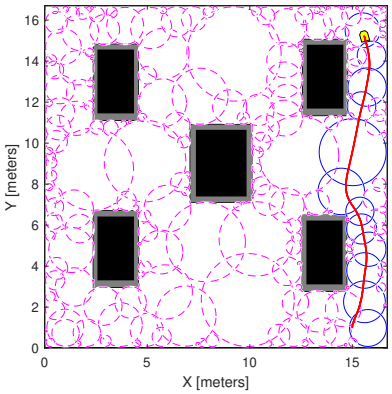
Figure 3.7: Starting position [1 8 pi/2]

Funnels-graph algorithm

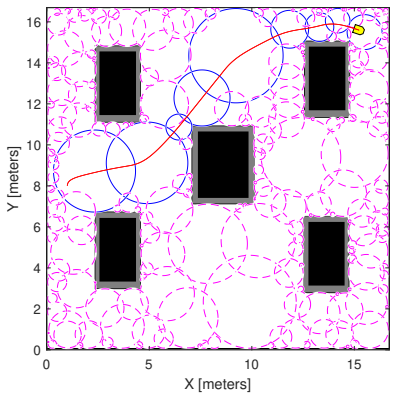
Figures 3.8a, 3.8b, 3.8c display the motion execution paths for different initial states. The graph funnels are depicted with magenta dashed lines, while the path funnels are shown in blue.

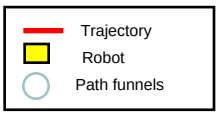


(a) Starting position (1 1 0)



(b) Starting position (15 1 pi/2)





(c) Starting position (1 8 pi/2)

Figure 3.8: Funnel-graph motion execution test samples

Table 3.2 summarizes the performance metrics of two algorithms, Modified RRT* and Funnel-graph, in a basic navigation task under noise-free conditions. The metrics recorded include planning time, number of generated nodes, path length, and execution time for three different starting positions.

Table 3.1: Experiment's recorded parameters

	Modified RRT* algorithm		Funnel-graph algorithm	
Planning time(s)	340		29.63	
Number of nodes	30000		221	
Start position	Path length(m)	Execution time(s)	Path length(m)	Execution time(s)
[1 1 0]	28.33	19	23.35	15.58
[15 1 pi/2]	21.63	14.43	14.64	9
[1 8 pi/2]	23.28	15.52	17.16	11.45

For the same execution time as the funnel graph algorithm here is the results of Modified RRT* algorithm :

Table 3.2: Experiment's recorded parameters

	Modified RRT* algorithm	
Number of nodes	5117	
Start position	Path length(m)	Execution time(s)
[1 1 0]	34.33	25
[15 1 pi/2]	25.98	19.76
[1 8 pi/2]	27.36	18.52

Discussion

1. Planning time: The Funnel-graph algorithm significantly outperforms the Modified RRT* algorithm in terms of planning time, requiring only 29.63 seconds compared to 340 seconds for the Modified RRT* algorithm.

2. Number of nodes: The Funnel-graph algorithm also generates far fewer nodes (221) than the Modified RRT* algorithm (30,000) and for the same execution time it gave less accurate results and less nodes which is not helpful for handling noise in next sections.

3. The Funnel-graph algorithm consistently outperforms the Modified RRT* algorithm in both path length and execution time for all three starting positions.

Interpretation of results

- **Efficiency and Performance**: The Funnel-graph algorithm repeatedly overtakes the Modified RRT* algorithm across all metrics. The shorter planning time and fewer nodes indicate a more efficient algorithm, due to the use of funnels, which effectively cover a larger portion of the environment at each sample compared to the Modified RRT* algorithm, highlighting a superiority in both time and space complexity.

- **Path Optimization**: The Funnel-graph algorithm's shorter path lengths suggest better optimization and a more direct route to the goal. This efficiency translates into faster execution times, which are crucial for real-time applications.

- **Algorithm Robustness**: both algorithms demonstrate good robustness as they successfully converge to the goal from all three initial states. This indicates that both algorithms can continuously find a feasible path to the target, regardless of the starting position. Making them reliable and effective in the navigation task, opposing to the open loop case where one plan is valid only for one configuration and the re-planning is needed whenever the goal state is changed

In summary, the Funnel-graph algorithm demonstrates significant advantages over the Modified RRT* algorithm in terms of planning efficiency, path optimization, and execution speed under noise-free conditions.

3.5.2 Planning with actuation noise

As mentioned in the general description, this section will present a simulation involving the presence of actuation noise during motion execution. The noise will be introduced as a deviation in the steering angle command as a ramp for a specified amount of time.

Modified RRT* algorithm

To incorporate the feedback plan during motion execution, we need to have a proper measure to know whether the robot is following the assigned path. Since the path is an array of waypoints, the continuous evaluation of the projection on the path represents a good measure to estimate the error since it represents the closest point to the path. This can be done by iterating through the waypoints and finding the projection of the robot's current position onto the line segments connecting the waypoints.

Given:

- $\mathbf{p}_i = (x_i, y_i)$ and $\mathbf{p}_{i+1} = (x_{i+1}, y_{i+1})$ are consecutive waypoints.
- $\mathbf{p}_r = (x_r, y_r)$ is the robot's current position.

For each segment $\mathbf{p}_i \rightarrow \mathbf{p}_{i+1}$:

1.We Define the Vectors:

$$\mathbf{v}_{i,i+1} = \mathbf{p}_{i+1} - \mathbf{p}_i = \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix} \tag{3.3}$$

$$\mathbf{v}_{r,i+1} = \mathbf{p}_{i+1} - \mathbf{p}_r = \begin{pmatrix} x_{i+1} - x_r \\ y_{i+1} - y_r \end{pmatrix} \tag{3.4}$$

2. We Compute the Projection Scalar:

$$\alpha = \frac{\mathbf{v}_{i,i+1}^\top \mathbf{v}_{r,i+1}}{\mathbf{v}_{i,i+1}^\top \mathbf{v}_{i,i+1}} \tag{3.5}$$

Where $\mathbf{v}_{i,i+1}^\top \mathbf{v}_{r,i+1}$ and $\mathbf{v}_{i,i+1}^\top \mathbf{v}_{i,i+1}$ denote the dot products.

3. Determine the Closest Point:

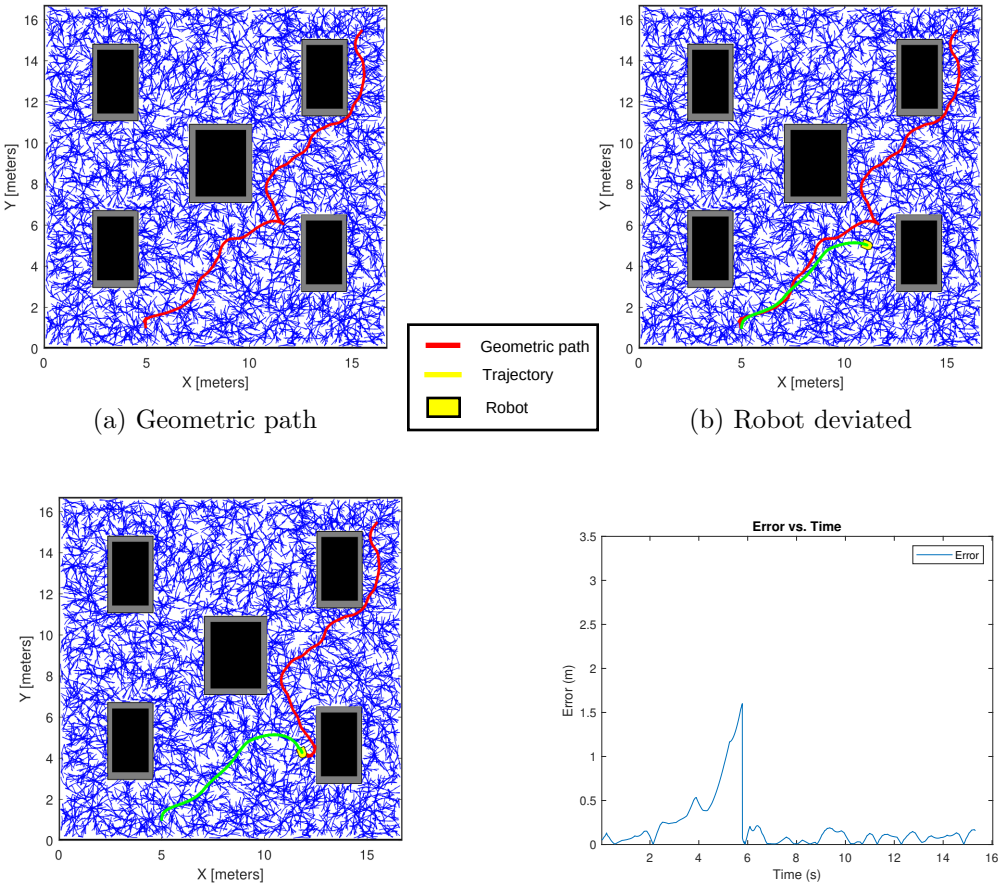
$$\mathbf{p}_{\text{closest}} = \begin{cases} \mathbf{p}_i & \text{if } \alpha \leq 0 \\ \mathbf{p}_{i+1} & \text{if } \alpha \geq 1 \\ \alpha \mathbf{p}_i + (1-\alpha)\mathbf{p}_{i+1} & \text{otherwise} \end{cases} \tag{3.6}$$

4. Calculate the Distance:

$$d = \|\mathbf{p}_r - \mathbf{p}_{\text{closest}}\| \tag{3.7}$$

By iterating through the way-points and calculating the distance d for each segment, we can continuously evaluate the projection of the robot's position onto the path and estimate the error by taking the minimum distance, this error will be compared to a threshold to know when assigning a new path is necessary, in this experiment the threshold will be set to 1.5 meters.

The geometric path is illustrated in Figure 3.9a. During motion execution, the robot initially follows the pre-planned path (green highlighted path), but then deviates as a result of the actuation noise, as shown in Figure 3.11c. Figure 3.11b depicts the newly allocated path designed to guide the robot back to the goal position. Finally, the graph in figure 3.9d records the error between the geometric path and the robot's trajectory.



(a) Geometric path

(b) Robot deviated

(c) New path assigned

(d) Error variation over time

Figure 3.9: Dynamically adjusting the robot's path upon deviation.

Funnels-graph algorithm

Actuation noise is introduced to the system for the configuration shown in Figure 3.8a. When the robot deviates from the original path funnels, the feedback plan is activated, selecting an alternative path from the graph, which connects the robot's current state to the goal state. The noise region is indicated by the black rectangle in Figure 3.10.

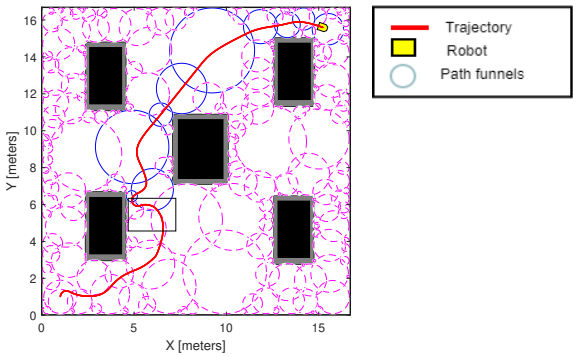
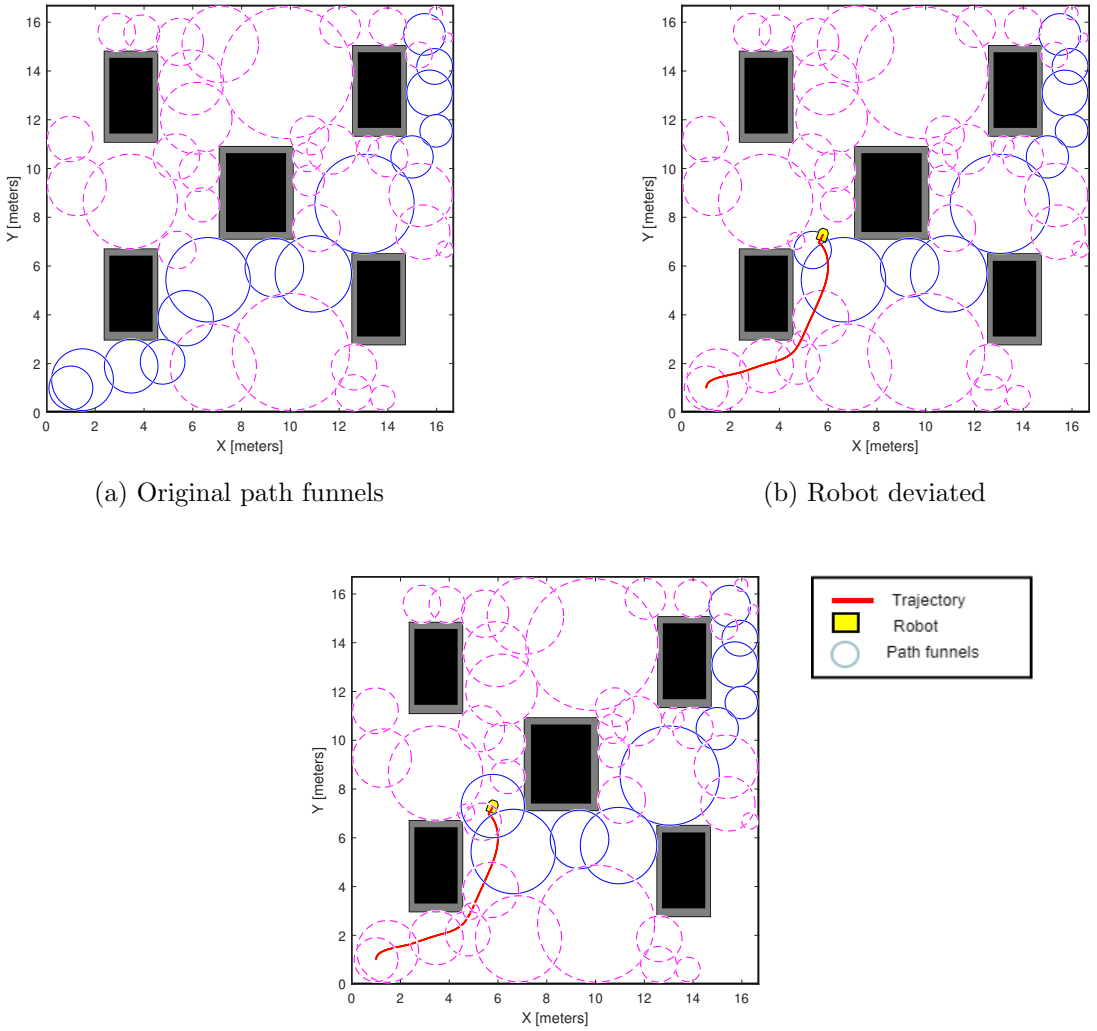


Figure 3.10: Funnel-graph algorithm adapts to actuation noise.

Another case to be considered is when the robot deviates into an unexplored area. The feedback plan expertly handles this situation by creating a new funnel in the unexplored region, integrating it into the existing graph, and forming edges with the neighboring funnels. this case is illustrated in the figure 3.11, where a map that is not fully covered is used for demonstration purposes.



(a) Original path funnels

(b) Robot deviated

(c) New funnel created

Figure 3.11: Exploring new areas by Funnel Creation.

Discussion

1. When applying a ramp noise to the steering angle the robot deviates gradually from the original path. The feedback plan has to react accordingly to ensure the convergence of the robot to the goal, the two algorithms have different feedback mechanisms. The modified RRT* algorithm reacts when the error threshold is exceeded by finding the nearest node within the tree to the actual position and tracing it back to the goal since all nodes converge there. Meanwhile, the funnel-graph algorithm reacts when the

robot deviates from the path funnels. It locates the robot and assigns a new optimized path to follow.

2. For this experiment both algorithms handled the case efficiently and their response time was within fractions of the second.

3.5.3 Planning with feedback noise

Next, the feedback plan is tested again by the measurement uncertainty. This uncertainty is plugged into the system by applying a Gaussian distribution noise $\mathcal{N}(\mu, \sigma^2)$ that simulates a sensor behaviour. The sensor operates at a frequency of 33Hz, referring to the rate at which measurements are taken. The figures shared for both algorithms represent a test sample that has *zero* mean μ, 0.3 and 1.5 variance σ^2 for position and heading angle measurements respectively.
The noisy measurements are indicated by red dots along both sides of the path in each algorithm.

Modified RRT* algorithm

Figure 3.12a illustrates the robot's trajectory starting at [4,1,0]. The same configuration has a path length of 20 meters under noise-free conditions. Figures 3.12b 3.12d 3.12d shows the simulated sensor measurements to the current position. Meanwhile, Figure 3.13 shows the projection error for the same path, using a threshold distance of 0.5 meters.

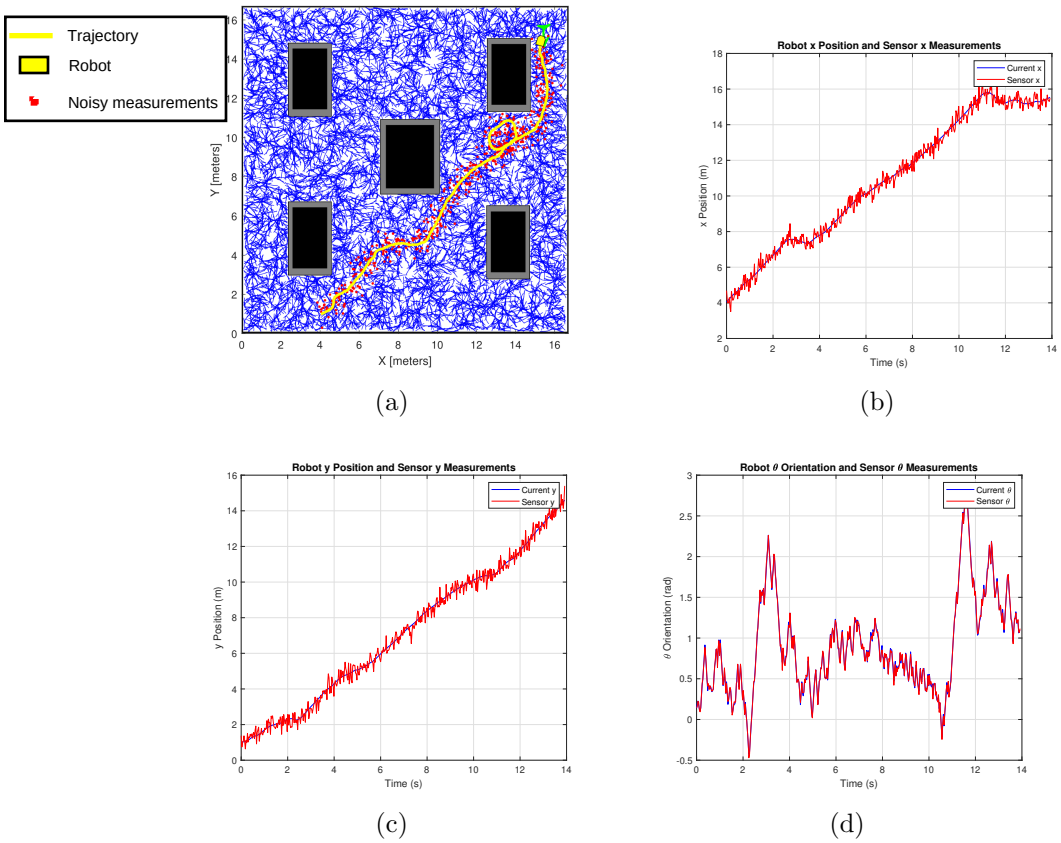


Figure 3.12: (a) Motion execution with Gaussian noise, (b) X coordinates sensor measurements, (c) Y coordinates sensor measurements, and (d) Heading angle sensor measurements.

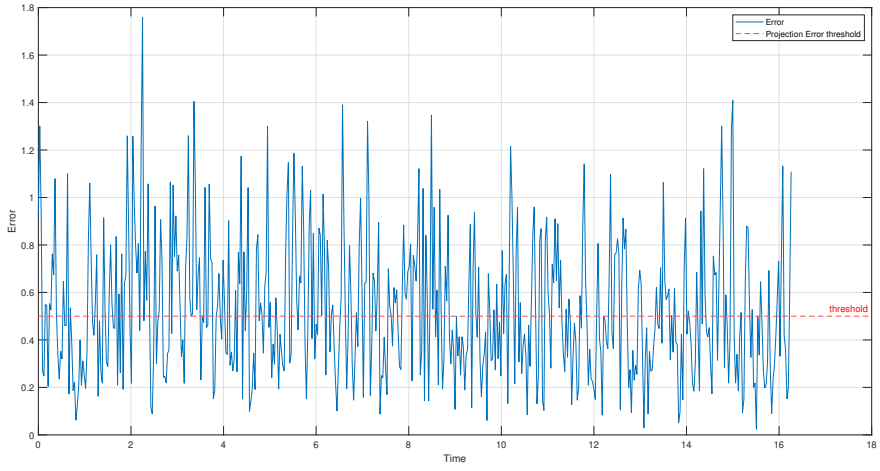


Figure 3.13: Projection error along the path.

Table 3.3 presents the performance metrics of the modified RRT* algorithm under different levels of position variance and projection error. The columns represent different position variances (σ_p^2) and the rows represent various projection error values in meters. For each combination, the table lists the path length (in meters) and the number of times the feedback plan was activated to adjust the path. "Diverges" indicates scenarios where the algorithm was unable to converge to the goal state.

Table 3.3: Performance Metrics under Varying Measurement

Position variance (σ_p^2)	0.3		0.4		0.6		0.8
Projection error threshold(m)	path length(m)	path adjustments	path length(m)	path adjustments	path length(m)	path adjustments	
0.3	22.62	380	Diverges		Diverges		Diverges
0.5	24.55	215	Diverges		Diverges		
0.8	20.87	64	20.86	190	Diverges		
1	21.05	26	21.27	80	28.23	261	
1.5	20.65	1	20.46	13	21.26	80	

Funnel-Graph algorithm

The path assigned in figure 3.14 starts at [1 1 0] and has a path length of 23 meters under noise-free conditions. Table 3.4 indicates how increasing the position variance affects both the path length and the number of adjustments required.

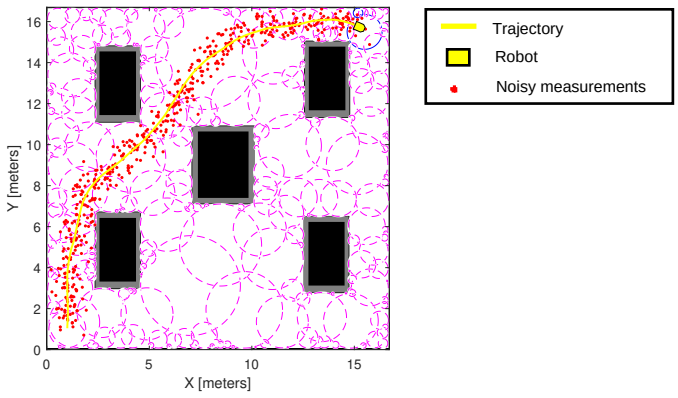


Figure 3.14: Motion execution with Gaussian noise.

Table 3.4: Path length and adjustments under different position variances

Position variance (σ_p^2)	0.3	0.4	0.6	0.8
Path length	21.95	22.00	21.77	21.94
Path adjustments	6	24	49	97

Discussion

The first table records the performance of the Modified RRT* algorithm under varying levels of measurement noise, represented as position variance (σ_p^2) to different projection errors. Whereas, the second table evaluates the Funnel-graph algorithm under similar noise conditions.

1. Path adjustments:

    - Modified RRT*: Required a high number of path adjustments, particularly at lower variances and projection errors. At 0.4m and 0.6m variances, the algorithm often diverges, showing its limitations in handling noise.
    - Funnel-graph: The number of path adjustments increases with the increase of noise levels, but the algorithm continues to function without divergence. This indicates its capacity to adapt and correct paths even under significant noise.

2. Convergence:

    - Modified RRT*: Struggles with convergence at higher noise levels (0.8m). For variances of 0.4m and 0.6m, the algorithm frequently fails to find a valid path, especially with lower projection errors, making it less reliable in noisy environments.
    - Funnel-graph: Maintains convergence across all tested noise levels, demonstrating superior robustness and reliability over uncertain environments.

Interpretation of results

- **Robustness to Noise**: The Funnel-graph algorithm demonstrates superior robustness to noise compared to the Modified RRT* algorithm. The Modified RRT* algorithm

frequently diverges at higher noise levels, limiting its applicability in noisy environments. In contrast, the Funnel-graph algorithm maintains consistent path lengths and adapts effectively to increasing noise levels, ensuring reliable navigation by adjusting the path as needed.

Limitations

This section summarizes the limitations identified across all three parts of the first scenario, highlighting the specific constraints for each algorithm:

- Modified RRT*:
  - **Scalability**: The Modified RRT* algorithm's efficiency decreases as the complexity of the task increases due to the need for a large number of nodes and extensive planning time.
  - **Sensitivity to Noise**: It shows significant sensitivity to measurement noise, frequently diverging at higher noise levels, which restricts its reliability in real-world scenarios with high sensor inaccuracies.
  - **Computational cost**: Due to its iterative nature and extensive node expansion, the algorithm becomes computationally intensive.
- Funnel-graph:
  - **Struggling in Narrow Passages**: The funnel-graph algorithm struggles in narrow passages, where small radius path funnels are required, making it spend more time navigating through these areas, especially in noisy environments.

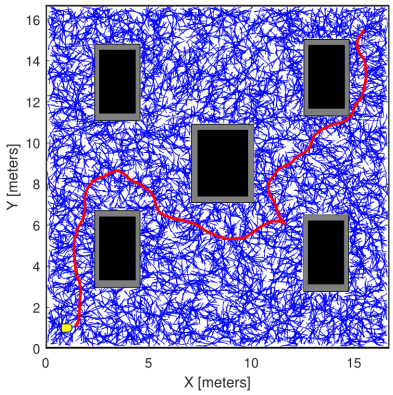
Both the modified RRT* and funnel-graph algorithms exhibit unique strengths and limitations. The modified RRT* algorithm, while providing a nearly optimized path in terms of distance, suffers from increased computational complexity because of the large number of nodes needed to cover the space. On the other hand, the funnel-graph algorithm, with its efficient coverage of the environment and quicker planning times, struggles with narrow passages.

3.6 Dynamic planning with static goal

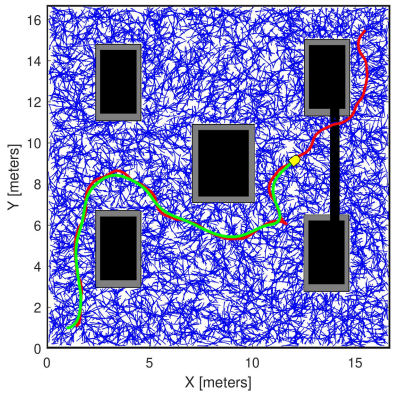
General description Another challenge involves visualizing the system's reaction to environmental changes. The plan entails introducing an obstacle that blocks the robot's pre-planned path during motion execution. The robot must then find a new path to the goal position. This scenario is applied to both algorithms.

Modified RRT* algorithm

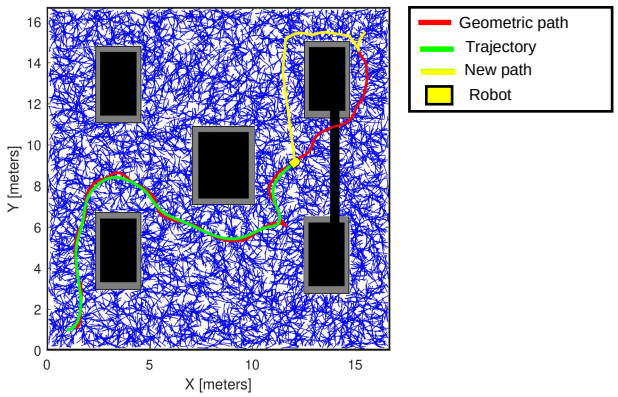
As it may appear from figure 3.15 that the map is more dense than the map used in the previous scenario, as the number of nodes has increased to 40000. That is mainly due to the complexity of the task and the need for a more optimal path.
Figure 3.15a shows an obstacle-free path connecting the start and the goal positions. Initially, the robot follows the pre-planned path, as indicated by the green trajectory (3.15b), Suddenly, an obstacle intercepts the robot's path. The algorithm then seeks a valid path for the robot in the changed environment. It begins by sorting the tree nodes based on their Euclidean distance to the current state. Next, it iterates through these sorted nodes, checking the feasibility of moving from the current state to each node. If a valid motion is found, it traces a potential path back to the tree root, ensuring that the entire path is valid between each pair of nodes. Upon identifying a valid path, the algorithm selects and returns it taking an average of 34 seconds variable depending on the complexity of situation and the tree's expansion. The resulting path (highlighted in yellow) is displayed in Figure 3.15c.



(a) Geometric path before obstacle appears.



(b) The obstacle appeared.

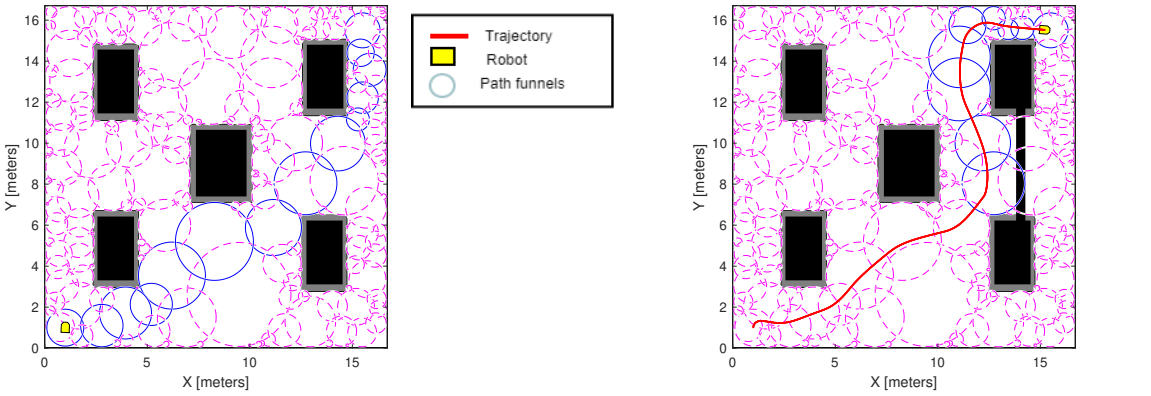


(c) Robot finds a new geometric path.

Figure 3.15: Robot Navigation Adaptation to environment change by modified RRT* star algorithm.

Funnels-graph algorithm

The figure 3.16 bellow illustrates the adaptation process in the funnel-graph algorithm when an unexpected obstacle blocks the planned path. In such cases, the feedback mechanism updates the cost of the graph edges blocked by the obstacle to infinity. This effectively removes those edges from the path choices, forcing the algorithm to find an alternative route around the obstacle. By performing a search for the shortest path possible path with the new adapted graph taking about 0.6 seconds which is quite the same for all experiments.



(a) Path-funnels before obstacle appears.

(b) Robot finds a new path-funnels after an obstacle appeared.

Figure 3.16: Robot Navigation Adaptation to environment change by funnel-graph algorithm.

Discussion

In the face of unexpected obstacles, the Modified RRT* and Funnel-graph algorithms exhibit distinct differences in adaptability, in terms of the employed mechanism and the time spent to find an alternative path. The Modified RRT* algorithm takes an average of 34 seconds to find a new path due to the computational cost reflected by its adaptability complex process, which has to sort close neighborhoods and iterate through their respective path. In contrast, the Funnel-Graph algorithm quickly adapts to changes, since it only updates the edges relative to the covered area and then starts looking for a feasible path in approximately 0.6 seconds across various experiments. This rapid response highlights the Funnel-Graph algorithm's superior efficiency in dynamically updating paths in real-time scenarios.
While both algorithms are robust and capable of finding alternative paths, the Funnel-Graph algorithm stands out for its speed and efficiency in adapting to environmental changes.

3.7 Extensions of the funnel-graph algorithm

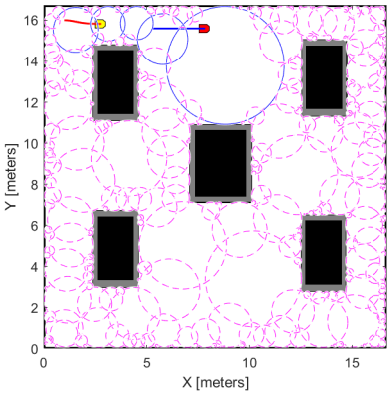
The funnel graph algorithm has proven to be an efficient feedback motion planning strategy, consistently delivering satisfying results across various experiments and scenarios. One notable feature of this algorithm is its remarkable flexibility, which enables it to be applied

to a wide range of useful applications, such as goal tracking. This section explores the algorithm's capability in both static and dynamic environments, demonstrating its versatility and effectiveness in addressing complex motion planning challenges.

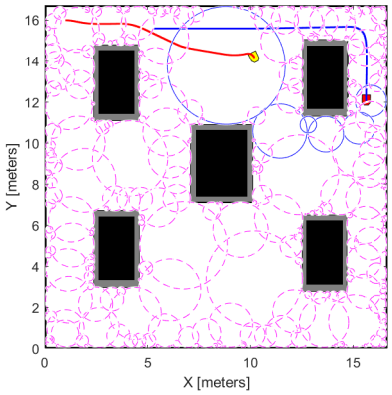
3.7.1 Static planning with dynamic goal

This experiment shows how the funnel-graph algorithm handles the case of goal tracking by simulating the Leader-Follower problem. Two robots are in the same environment one of them (Leader) has a predefined valid path to be followed then stops at a certain final state, the second robot (Follower) tries to catch up with it continuously finding the shortest path to reach the first one, the Leader robot is assumed to have higher speed than the Follower to showcase that the follower is indeed tracking and adapting to the Leader robot position not just following the same path.

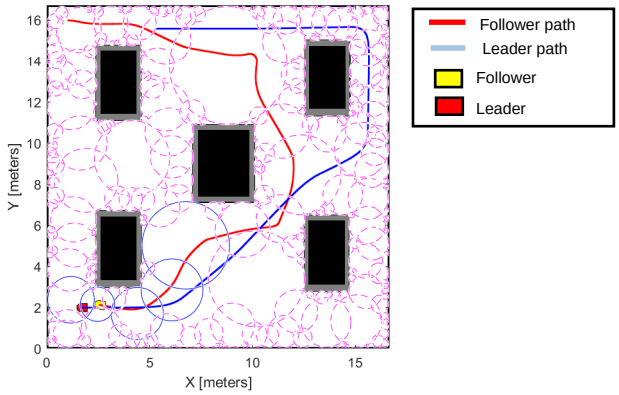
As shown in the figure 3.17 the Follower robot (Yellow with red trace) is always trying to reach the Leader(Red with blue trace) robot (and adapting to it's current position (Assuming that both robots positions is always known),by continuously checking for their positions within funnel and leveraging the flexibility of the graph data structure we can always set the shortest obstacle-free path within the graph that connects the follower to the leader robot. Since the algorithm is efficient both computation-wise and space-wise, it guarantees real-time adaptation within 0.6 seconds. This efficiency is critical for scenarios requiring quick response times and high precision, such as multi-robot coordination, search and rescue operations, and any situation demanding precise and continuous following behavior.



(a) Initial state for both robots.



(b) The follower found a shorter path to reach the leader.



(c) The follower successfully reached the leader.

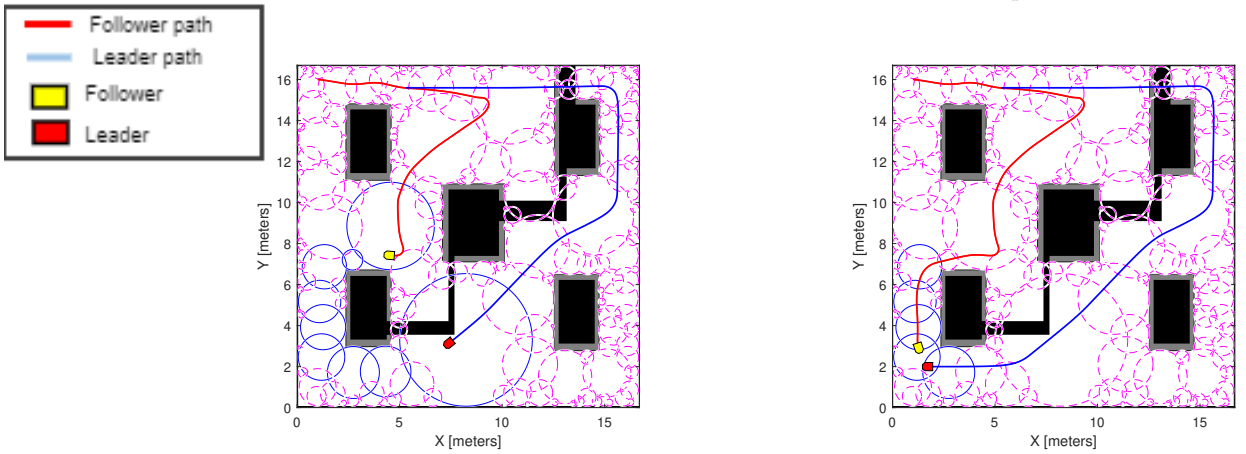
Figure 3.17: Leader-Follower in a static environment.

3.7.2 Dynamic planning with dynamic goal

Building on the static environment scenario, we now look at the dynamic environment case. Here, the environment changes a lot with new blockages showing up. The Follower robot has to keep adjusting to these changes, finding new paths in real-time to go around the new obstacles while staying close to the Leader. This dynamic case checks if the algorithm can deal with these fast changes and keep the Follower tracking the Leader even with the changing environment.

Figure 3.18 illustrates the dynamic adaptation in a Leader-Follower scenario. In figure 3.18a, both robots are starting. Figure 3.18b shows the first adaptation, where the follower robot

finds a shorter path to reach the leader, demonstrating its ability to react to changes in the environment. In figure 3.18c, the follower makes a second adaptation, further adjusting its path to stay on course toward the leader as the environment continues to change. Finally, figure 3.18d depicts the successful completion of the task. The adaptation process is the same as in figure 3.6 where the the costs of the edges are updated pushing the robot to pick a path with a lower cost.



(a) Initial State: Both robots started moving.

(b) First Adaptation: Follower finds an alternative path to the leader.

(c) Second Adaptation: Follower adjusts its path again to reach the leader.

(d) Goal Achieved: Follower successfully reaches the leader.

Figure 3.18: Leader-Follower in a dynamic environment.

3.8 Implementation

In this section, we attempted to put the theoretical concepts into practice. We focused on implementing feedback motion planning algorithms and integrating them into a robotic system. We described the system architecture, the hardware and software components used, and the methods we employed to achieve real-time path planning and adaptation. Despite our efforts, we faced significant challenges during the implementation process and did not succeed in achieving the desired outcomes.

3.8.1 Overview

The goal of this implementation is to incorporate feedback motion planning into a real robotic system (see figure 3.19) to navigate and adapt in real time to noise and uncertainties in modeling and measurements.

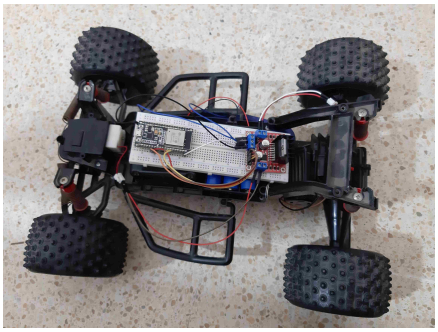


Figure 3.19: Wheeled Mobile Robot.

This robot utilizes RC car chassis for its base, equipped with two DC motors for maneuverability. One motor, positioned at the front, acts as the steering mechanism. The rear motor is for driving the robot forward and backward.
The robot has the following dimensions :

Table 3.5: Robot Specifications

Specifications	Value
Length	40 cm
Width	38 cm
Wheel Radius	5 cm
Maximum Steering Angle	45 degrees

3.8.2 Hardware

ESP-Wroom-32: The ESP32 is a versatile microcontroller, featuring a compact size, low power consumption, and a wide array of GPIO pins. It can interface with various sensors, actuators, and motor controllers due to its multiple connectivity options, such as Wi-Fi, Bluetooth, and serial communication protocols. In this project, the ESP32, presented in figure 3.20, is used. It boasts a powerful dual-core Xtensa LX6 processor running at high clock speeds. This processor configuration offers notable advantages, including high-speed data transmission and robust performance capabilities.

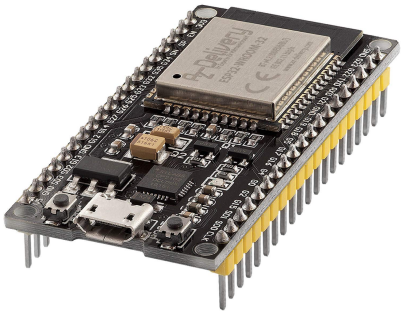


Figure 3.20: ESP32 microcontroller.

L298N Motor driver: The L298 motor driver in figure 3.21 is a robust and efficient H-Bridge motor driver that can control the direction and speed of two DC motors or a single stepper motor. It features high current handling capacity and thermal protection, making it suitable for a wide range of applications. The L298 can handle up to 2A per channel and up to 46V.

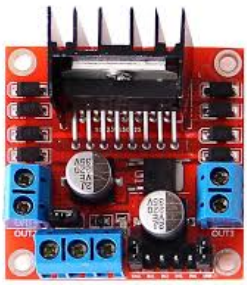


Figure 3.21: Motor driver.

3.8.3 System Hierarchy

:

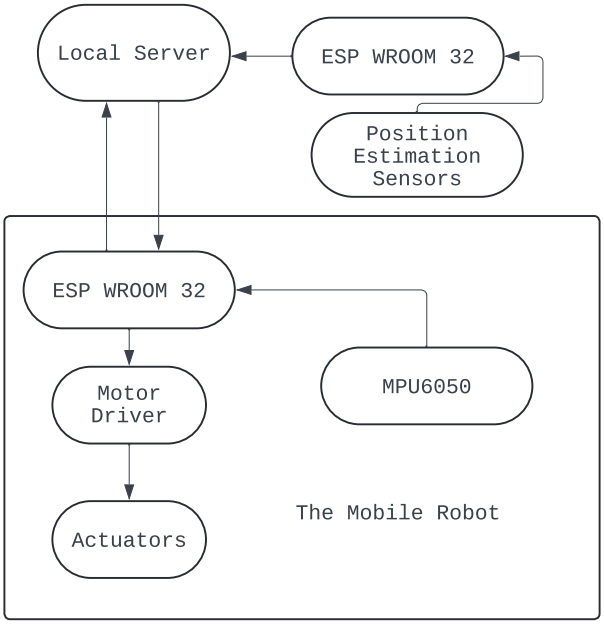


Figure 3.22: System Hierarchy.

This diagram in figure 3.22 represents our attempt to realize a robotic system, where it outlines the primary components and their interconnections. The local server is setup on a PC and it used to generate and save the Feedback plan to use it in the online navigation task, The server initiates the communication with the robot sending the commands as

motor voltages along with execution time for each command and receiving the orientation information. Meanwhile, another microcontroller is used to estimate the coordinates of the robot together with the orientation, we should get a good estimation of robot pose which is then fed back to the server to determine the next action.

We were able to establish two-way communication between the local server and microcontroller, also send predefined commands via TCP IP. Our initial plan was to employ the triangulation method using fixed access points in the environment and since the ESP32 is capable of determining the access point signal strength in dbm, this measure is converted to distance with the formula mentioned in [25] to convert dbm to meters.

$$d = 10^{\left(\frac{\text{RSSI}-A}{10 \cdot n}\right)}$$

Where A is the RSSI value in dBm measured at 1 meters and n is the rate at which the signal attenuates with distance. but research results showed the method were inaccurate (the error was over 6 meters). Another approach we tried was to estimate the time of flight of the ultrasonic waves but the the range and measurement angle of the HC-SR04 ultrasonic sensors were unstable, this issue raised the need for more accurate way to estimate the position as ESP32 Ultra-Wide Band which proven reliable for indoor position estimation in [26] or a ranging sensor as LiDAR but due to their unavailability in the local market, high cost and late delivery date if ordered from global markets, we were not able to complete the implementation within the due date.

General Conclusion

Throughout this report, we explored the concept of feedback motion planning for autonomous car-like robots by developing two sampling-based motion planners: the Modified RRT* algorithm and the Funnel-graph algorithm. These planners were designed with a feedback property to ensure the robot's convergence to the goal position, even in the presence of noise. Both algorithms were tested under identical noise conditions and environmental changes to compare their performance in the navigation task. Additionally, the Funnel-graph algorithm was subjected to an extra test to evaluate its goal-tracking capability in a dynamic environment.

The obtained results were satisfactory concerning the funnel-graph algorithm's ability to guide the robot in noisy conditions and its rapid adaptability. In contrast to the Modified RRT* algorithm, which failed to converge at certain noise levels and exhibited a significantly longer adaptation time, making it less suitable for real-time applications.

Future work should focus on enhancing the robustness of the Modified RRT* to noise and improving distance optimization For the Funnel-graph algorithm. Additionally, implementing these algorithms on a real robot and validating their performance in real-world conditions will be an important step in future work.

In conclusion, this study contributes valuable insights into the performance of feedback motion planning algorithms under various conditions, guiding future advancements in autonomous robot navigation.

Bibliography

[1] J. Pan and D. Manocha, "Efficient configuration space construction and optimization for motion planning," *Engineering Sciences Press*, 2015.

[2] S. M.LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[3] A. Anuragi, "Local path planning using virtual potential field in python." [Online]. Available: https://medium.com/nerd-for-tech/local-path-planning-using-virtual-potential-field-in-python-ec0998f490af

[4] G. Sasi Kumar, B. Shravan, H. Gole, P. Barve, and L. Ravikumar, "Path planning algorithms: A comparative study," 12 2011.

[5] "Probabilistic roadmaps (prm)," accessed: 2024-03-24.

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *International Journal of Robotic Research - IJRR*, 2011.

[7] B. Aicha, A. BENALIA, and F. Boudjema, "Robust output trajectory tracking of car-like robot mobile," *Journal of Electrical Systems*, vol. 12, pp. 541–555, 09 2016.

[8] S. Kundu, "Understanding geometric path tracking algorithms: Stanley controller," 2020, accessed: 2024-06-10. [Online]. Available: https://medium.com/roboquest/understanding-geometric-path-tracking-algorithms-stanley-controller-25da17bcc219

[9] Y.-H. Chen, Y. Shan, L. Chen, K. Huang, and D. Cao, "Optimization of pure pursuit controller based on pid controller and low-pass filter," *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:54463850

[10] J. M. Snider, "Automatic steering methods for autonomous automobile path tracking," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:17512121

[11] Wikipedia contributors, "Vector field — Wikipedia, the free encyclopedia," 2024, [Online; accessed 2024-05-10]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Vector_field&oldid=1223971094

[12] C. D. W. Pieter-Jan De Smet, "Extended solutions for the biadjoint scalar field," *Physics Letters B*, 2017- 11 -17.

[13] H. A. G Li, A Yamashita and Y. Tamura, "An efficient improved artificial potential field based regression search method for robot path planning," *Mechatronics and Automation*, 2012.

[14] S. R. Lindemann and S. M. LaValle, "Smoothly blending vector fields for global robot navigation," *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference 2005*, December 2005.

[15] S. M. L. Libo Yang, "The sampling-based neighborhood graph: An approach to computing and executing feedback motion strategies," *IEEE Transactions on Robotics and Automation*, June 2004.

[16] M. K. M. Jaffar and M. W. Otte, "Pip-x: Online feedback motion planning/replanning in dynamic environments using invariant funnels," *ArXiv*, 2022.

[17] D. E. Dirkse. (2024) Turning radius calculation. Accessed: 2024-05-27. [Online]. Available: https://davdata.nl/math/turning_radius.html

[18] J.-C. Latombe, *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[19] L. E. Kavraki, S. M. L. I. B. Siciliano, and e. O. Khatib, *Springer Handbook of Robotics.* Springer-Verlag, 2008.

[20] O. Brock, J. Kuffner, J. X. I. B. Siciliano, and e. O. Khatib, *Springer Handbook of Robotics.* Springer-Verlag, 2008.

[21] J. hyuk Yu, "Implementation of path tracking algorithms and trajectory optimization based on the extended kalman filter," *Journal of Advanced Robotics*, 2021.

[22] S. Xiang. (2024) Basic pure pursuit — purdue sigbots wiki. Accessed: 2024-05-22. [Online]. Available: https://wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit

[23] R. Coulter, "Implementation of the pure pursuit path tracking algorithm," *The Robotics Institute, Carnegie Mellon University, Pittsburgh*, January 1992.

[24] S. R. Lindemann and S. M. LaVall, "Smooth feedback for car-like vehicles in polygonal environments," *2007 IEEE International Conference on Robotics and Automation*, April 2007.

[25] T. A. Ebru Alp, Tamer Dag, "Indoor positioning system by using triangulation algorithm," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 2016.

[26] V. S. T. Ton Nhat Nam Ho and N. T. M. Nguyen, "Design an indoor positioning system using esp32 ultra-wide band module," *Springer Nature Switzerland*, 2023.