Vers une planification robuste et sûre pour les systèmes autonomes

Romain Pepy

N° D'ORDRE 9371

THÈSE DE DOCTORAT

SPECIALITE : PHYSIQUE

Ecole Doctorale « Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

Présentée par : Romain Pepy

Sujet :

Vers une planification robuste et sûre pour les systèmes autonomes

Soutenue le 4 février 2009 devant les membres du jury :

M. Tarek HAMEL	Rapporteur
M. Luc JAULIN	Président
M. Michel KIEFFER	Co-encadrant
M. Hugues MOUNIER	Examinateur
M. Philippe POIGNET	Rapporteur
M. Eric WALTER	Directeur de thèse

Vers une planification robuste et sûre pour les systèmes autonomes

Le chemin le plus court d'un point à un autre c'est de ne pas y aller.
P. Geluck

Remerciements

Merci à mon directeur de thèse Eric Walter et à mon co-encadrant Michel Kieffer de m'avoir accordé leur confiance en acceptant d'encadrer mes travaux de thèse.

Merci aux deux rapporteurs, Tarek Hamel et Philippe Poignet, d'avoir accepté de se plonger dans mon manuscrit.

Merci à Luc Jaulin d'avoir présidé le jury.

Merci à Alain Lambert et Hugues Mounier d'avoir participé à l'encadrement de ma thèse et d'avoir pris part au jury.

Merci à tous les collègues rencontrés durant ces trois années, et plus particulièrement à Stéphane et Eric pour ces interminables pauses café.

Merci à Marianne et à ma famille d'avoir toujours été à mes côtés.

Table des matières

Notations

Valeurs ponctuelles

x	:	scalaire
$\mathbf{x}$	:	vecteur colonne
$\mathbf{X}$	:	matrice
$\mathbf{I}$	:	matrice identité

Ensembles

$\emptyset$	:	ensemble vide
$\mathbb{N}$	:	ensemble des nombres entiers naturels
$\mathbb{Z}$	:	ensemble des nombres entiers
$\mathbb{R}$	:	ensemble des nombres réels
$\mathbb{X}$	:	espace d'état
$\mathbb{X}_{\text{free}}$	:	sous-ensemble de l'espace d'état où l'état du système peut se trouver
$\mathbb{X}_{\text{obs}} = \mathbb{X} \setminus \mathbb{X}_{\text{free}}$	:	sous-ensemble de l'espace d'état où l'état du système ne peut pas se trouver

Intervalles

$[x] = [\underline{x}, \overline{x}]$	:	intervalle
$[\mathbf{x}] = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$	:	vecteur d'intervalles
$[\mathbf{X}] = [\underline{\mathbf{X}}, \overline{\mathbf{X}}]$	:	matrice d'intervalles
$\text{lb}([x])$	:	borne inférieure $\underline{x}$ de $[x]$
$\text{ub}([x])$	:	borne supérieure $\overline{x}$ de $[x]$
$\text{mid}([x])$	:	centre de l'intervalle $[x]$
$w([x])$	:	longueur de l'intervalle $[x]$

Structures de données

G	:	graphe
$\mathbb{S}$	:	ensemble des sommets d'un graphe
$\mathbb{A}$	:	ensemble des arêtes d'un graphe
$\mathbb{V}$	:	ensemble des nœuds déjà visités d'un graphe
Vor(.)	:	région de Voronoï
V(.)	:	diagramme de Voronoï

Algorithmes

Les algorithmes sont écrits en pseudo-code. Les arguments des fonctions sont précédés de **in :** si ce sont des paramètres nécessaires à l'exécution de l'algorithme. Ils sont précédés de **inout :** si ce

sont des arguments dont la valeur sera modifiée par l'algorithme. Ils sont précédés de **out :** si ce sont des arguments dont la valeur est retournée par l'algorithme.

Table des figures

Liste des algorithmes

Introduction

Un système physique peut être décrit de façon univoque par un ensemble de variables minimal. Ces variables sont associées à des grandeurs physiques qui caractérisent l'état de ce système à un moment donné. Elles peuvent être regroupées dans un vecteur d'état dont la dimension correspond au nombre de degrés de liberté du système.

Pour se rendre d'un état initial à un autre état, le système doit en général passer par des états intermédiaires. La recherche de ces états intermédiaires permettant de faire passer un système d'un état initial à l'état voulu s'appelle la *planification de trajectoires*. La trajectoire est ici comprise comme la succession d'états du système permettant de rejoindre l'état cible. Le passage d'un état à un autre le long de cette trajectoire est possible en utilisant la suite ordonnée d'actions à effectuer contenue dans le *plan*.

Le plan renvoyé par le *planificateur* doit prendre en compte différentes contraintes. Certaines sont liées au modèle régissant les changements d'état du système. Le passage d'un état à un autre peut s'avérer impossible pour un système du fait de sa construction (bras articulé ne pouvant dépasser un certain angle). D'autres contraintes sont liées à l'environnement dans lequel le système évolue, qui peut rendre impossible certains mouvements en principe autorisés (objet encombrant l'environnement) ou à des contraintes de sécurité (température de fonctionnement trop élevée, vitesse de rotation trop importante).

Ces contraintes permettent un partitionnement de l'*espace d'état* contenant l'ensemble des états que pourraient prendre le système s'il n'était soumis à aucune contrainte. L'espace d'état est ainsi divisé en deux sous-ensembles : les états autorisés par les contraintes, et les états interdits. La trajectoire doit donc être générée de telle façon qu'elle ne contienne aucun état interdit.

De nombreux outils existent pour résoudre ces problèmes de planification sous contraintes. On peut les regrouper en deux classes principales. Les plus anciens planificateurs utilisent une discrétisation préalable de l'espace d'état. Les plus récents, les planificateurs à échantillonnage, permettent une exploration plus efficace. Ces planificateurs sont utilisés dans de nombreux domaines d'application, comme la chimie, la biologie, la robotique, l'automatique ou encore l'intelligence artificielle.

Notons que l'état du système à modifier n'est pas forcément parfaitement connu. Outre le fait que le modèle du système puisse être défini de façon imparfaite, on doit parfois utiliser des observations pour estimer l'état probable du système. Ces informations obtenues à l'aide de capteurs (thermomètre, centrale inertielle, photodiode...) engendrent une connaissance imparfaite de l'état du système. L'accumulation de ces imprécisions peut, même si le système suit à la lettre un plan prédéfini, entraîner le système dans des états inattendus qui peuvent lui être interdits avec des conséquences éventuellement désastreuses. Le planificateur doit donc prendre en compte ces imperfections

lors de la génération de la trajectoire afin que ces situations soient anticipées. On parle alors de *planification sous incertitudes*. Des planificateurs sous incertitude existent mais utilisent fréquemment des techniques supposant une discrétisation de l'environnement ou des représentations irréalistes de l'incertitude sur l'état du système.

La contribution majeure de nos travaux sera d'apporter une réponse au problème de planification de trajectoires en présence d'incertitudes en associant une technique de planification moderne, permettant une exploration rapide de l'espace d'état à des méthodes de localisation permettant de caractériser l'incertitude sur l'état du système à un instant donné.

Les exemples d'application que nous traiterons sont inspirées de la robotique mobile et plus particulièrement de la planification de trajectoires pour un véhicule de type voiture, mais nous avons tenu à commencer par une présentation aussi générale que possible car les techniques dont nous parlerons ne sont pas limitées à ce seul contexte.

On peut distinguer trois étapes lorsque l'on souhaite déplacer un système d'un état à un autre. Tout d'abord, le système doit connaître, au moins partiellement, son environnement. Cette phase de cartographie est nécessaire afin de pouvoir situer les zones interdites. Dans le cas de la robotique, ces zones sont liées à la présence d'obstacles, de murs, de piétons ou bien d'autres véhicules à éviter par exemple. Une fois l'environnement immédiat connu, la phase de planification de trajectoire va définir les changements d'états successifs à opérer afin que le système puisse se rendre dans l'état souhaité. Cette phase doit prendre en compte les contraintes liées à la dynamique du système ainsi que les informations sur les obstacles identifiable sur la carte de l'environnement. La troisième partie consiste à suivre la trajectoire en appliquant les commandes successives du plan au système muni des asservissements appropriés.

Ces trois parties ne sont pas nécessairement indépendantes les unes des autres. Il est en effet courant que la cartographie se fasse au fur et à mesure des déplacements. Il est aussi possible dans un environnement inconnu que la cartographie nécessite de faire bouger le système : les trois étapes sont alors concurrentes.

Nous nous sommes efforcés de décrire, dans une première partie allant du chapitre 1 à 4, le problème de planification sous incertitudes ainsi que les solutions que nous proposons dans un cadre le plus général possible. La seconde partie, du chapitre 5 au chapitre 7, est quant à elle dédiée aux applications de cette méthodologie en robotique.

Le chapitre 1 exposera le problème générique de la planification de trajectoires. Nous verrons différents problèmes qui peuvent se poser en fonction du contexte considéré, et présenterons divers planificateurs permettant de résoudre ces problèmes.

Le chapitre 2 sera consacré à la présentation et à la formalisation du problème de planification de trajectoires en présence d'incertitudes. Nous y dresserons un bref état de l'art.

Une première technique de planification tenant compte de l'incertitude pour un système quelconque sera présentée dans le chapitre 3. Ce planificateur, nommé Kalman-RRT utilise l'algorithme de planification RRT pour explorer l'espace d'état étendu. Cet espace d'état étendu représente de façon probabiliste, à l'aide de fonctions gaussiennes, la probabilité que le système a de se trouver dans un état donné. L'évolution des incertitudes se calcule en utilisant la phase de prédiction d'un filtre de Kalman étendu. La phase de correction de ce filtre est également utilisée afin de réduire l'incertitude sur l'état du système lorsque cela est possible.

Le chapitre 4 proposera une nouvelle approche radicalement différente de la résolution de ce problème. Les problèmes induits par l'utilisation d'un modèle probabiliste pour représenter les états seront en partie résolus grâce à l'utilisation d'ensembles pour représenter tous les états que peut prendre le système à un instant donné. L'algorithme résultant, toujours fondé sur le RRT, sera décliné en deux versions : la première nommée Set-RRT se veut la plus générale possible dans le choix de l'ensemble de référence. La seconde version, nommée Box-RRT, est une mise en œuvre pratique du Set-RRT pour les vecteurs d'intervalles.

Le chapitre 5 est une introduction au problème de planification de trajectoires dans le contexte de la robotique. Nous y verrons les contraintes spécifiques aux véhicules et notamment la contrainte de non-holonomie.

Les chapitres 6 et 7 proposeront une mise en œuvre des algorithmes de planification Kalman-RRT et Box-RRT présentés respectivement dans les chapitres 3 et 4 dans le cadre de la robotique. Nous montrerons ainsi les capacités de nos planificateurs à résoudre des problèmes concrets de recherche de trajectoires permettant d'éviter les collisions en présence d'incertitudes.

Ces travaux ont donné lieu à :

- Cinq articles dans des conférences avec comité de lecture :
  - R. Pepy et A. Lambert. Safe path planning in an uncertain-configuration space using RRT. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5376–5381, Beijing, China, 2006,
  - R. Pepy, A. Lambert, et H. Mounier. Path planning using dynamic vehicle model. In *Proc. IEEE International Conference on Information and Communication Technologies : from Theory to Applications*, pages 781–786, Damascus, Syria, 2006,
  - R. Pepy, A. Lambert, et H. Mounier. Reducing navigation errors by planning with realistic vehicle model. In *Proc. IEEE Intelligent Vehicle Symposium*, pages 300–307, Tokyo, Japan, 2006,
  - R. Pepy. Collision test in a Gaussian-configuration space : a geometrical point of view. In *Proc. IEEE International Conference on Advanced Robotics*, pages 1063–1068, Jeju, Korea, 2007,
  - R. Pepy, M. Kieffer et E. Walter. Reliable Robust Path Planner. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1655–1660, Nice, France, 2008.
- Un article de revue :
  - R. Pepy, M. Kieffer, et E. Walter. Reliable Robust Path Planning. *International Journal of Applied Mathematics and Computer Science, à paraître en 2009.*

Première partie

Planification de trajectoires

Chapitre 1

Planification de trajectoires

Sommaire

Cette partie introduit tout d'abord le concept de planification de trajectoires. Elle présente différentes façons de résoudre ces problèmes de recherche de trajectoires en fonction de leurs caractéristiques et dresse ainsi un état de l'art des planificateurs existants.

1.1 Planification de trajectoires : domaines d'application

Planifier, c'est dans notre contexte trouver une suite de commandes à appliquer à un système pour qu'il arrive à l'état final désiré. Ainsi, partant d'un état initial fixé, on cherche à le faire évoluer jusqu'à l'état désiré en respectant les contraintes qui rendent la solution acceptable. Bien que les types de problèmes soient les mêmes, le vocabulaire dépend des domaines d'application considérés et les méthodes de résolution utilisées diffèrent elles aussi suivant le contexte. Il sera intéresssant de voir dans quelle mesure les méthodes classiques dans un domaine sont transposables à d'autres.

Les paragraphes qui suivent n'épuisent bien sûr pas la liste des domaines dans lesquels la planification de trajectoire joue un rôle important.

1.1.1 Robotique

Dans le cadre de la robotique, et plus particulièrement des systèmes autonomes, on a affaire à un système mécanique pourvu de capteurs que l'on souhaite faire changer d'état. On dispose pour ce faire d'un certain nombre d'entrées, les commandes, qui permettent d'agir sur sa configuration actuelle. Le planification doit, dans ce contexte, trouver une suite de commandes réalisables à appliquer afin que le robot arrive à l'état final désiré.

Rappelons que l'état est un vecteur dont chaque composante est associée à un degré de liberté du système. Il renferme toutes les informations permettant de calculer l'évolution du système en fonction des entrées qui lui seront appliquées.

La planification de trajectoires pour un véhicule relève de ce domaine d'application. On cherche alors à déplacer le véhicule partant d'une position et d'une orientation initiales connues vers un autre état se situant dans une place de parking par exemple. Le vecteur d'état permettra d'identifier par exemple la position du véhicule dans un plan cartésien ainsi que son orientation. Le problème est alors de trouver les commandes à appliquer au véhicule (accélération, freinage, orientation du volant) afin de rejoindre la place de parking sans heurt.

Les problèmes en robotique peuvent revêtir un caractère beaucoup plus industriel. C'est le cas, par exemple, des bras articulés dans les chaînes de montage. Ces bras, qui peuvent s'articuler dans de nombreuses directions, doivent enchaîner des mouvements compliqués sans heurter leur environnement qui peut être également fort complexe.

1.1.2 Intelligence artificielle

En intelligence artificielle, planifier peut s'apparenter à la résolution de puzzles. On essaie par exemple, dans un jeu vidéo, de déplacer un personnage d'un endroit A de la carte vers un endroit B, choisi par l'utilisateur. Cette fonctionnalité, appelée *PathFinding*, est souvent critiquée par les utilisateurs de jeux de stratégie : les personnages ayant souvent tendance à faire des tours et des détours pour traverser la carte. Les valeurs stockées dans le vecteur d'état ne sont plus continues comme pouvait l'être l'angle du volant dans un véhicule ; ce sont ici des valeurs discrètes identifiant la case où se trouve le personnage. Sur la figure 1.1, un chemin est planifié afin qu'un personnage situé sur la case D1 puisse se déplacer sans heurter l'environnement (cases noires) en A2.

Les puzzles tels le cube de Rubik ou des casses-têtes chinois en trois dimensions pourraient être appréhendés dans un espace d'état continu, mais c'est tout naturellement que l'on arrive à discrétiser les différents états du système considéré. Ainsi, l'angle de rotation de chacune des faces du cube de

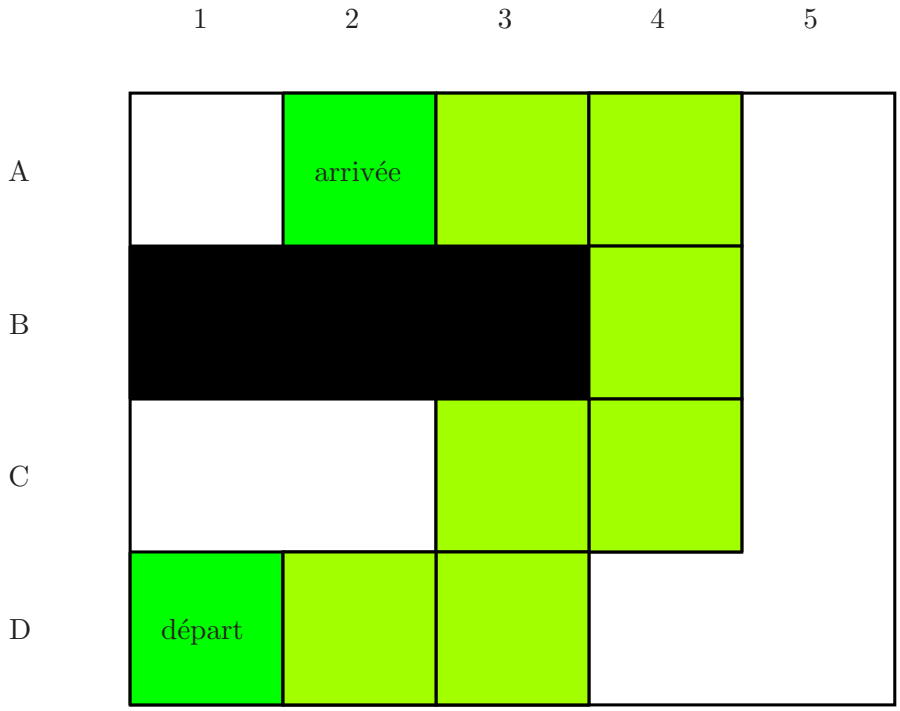


FIG. 1.1 – Recherche d'un chemin entre les cases D1 et A2

Rubik pourrait faire office de variable d'état. Mais les seules positions qui font sens sont celles où l'objet prend la forme d'un cube, l'ensemble des angles que peut prendre une face ne contient donc que quatre éléments. Il en est de même pour les jeux vidéos. Les positions de chacun des personnages sont discrètes même si les nouveaux jeux tendent à avoir des pas de discrétisation très faibles afin de laisser une plus grande liberté aux joueurs. Dans ce cadre, le terme *robot* est souvent remplacé par le terme *agent*. Les plans sont donc dans ces cas précis une suite de cases contiguës sur lesquelles le personnage peut se déplacer, ou une suite de configurations accessibles par des rotations du cube de Rubik.

1.1.2.1 Exemple

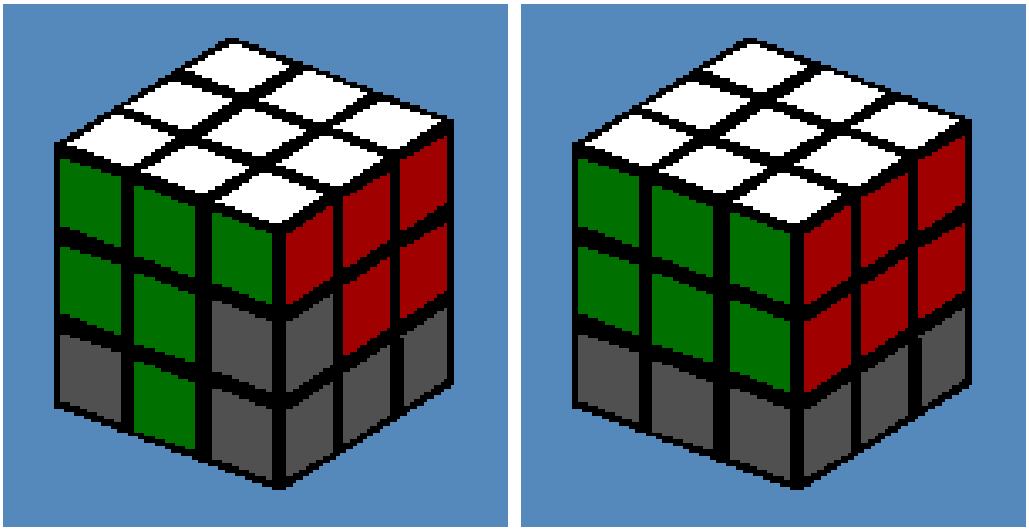
La résolution d'un cube de Rubik se fait en général couche après couche. Une fois la première couche réalisée ainsi qu'une partie de la seconde, supposons la configuration du cube illustrée par la figure 1.2(a). Afin de finir la deuxième couche et d'arriver dans la configuration de la figure 1.2(b), il suffit d'exécuter le plan suivant :

$$D'R'DRDFD'F',$$

où chaque lettre correspond à une face du cube (Down, Right, Front) à faire tourner d'un quart de tour dans le sens des aiguilles d'une montre (ou dans le sens inverse si cette lettre est suivie d'une apostrophe).

1.1.3 Biologie et chimie

Les algorithmes de planification sont également utilisés dans de multiples domaines de l'ingénierie et notamment en biologie et en chimie pour la synthèse de médicaments [1–3]. Les molécules telles que la caféine et l'ibuprofène sont petites et flexibles et se logent dans les cavités de protéines nettement plus grosses. L'ancrage des médicaments dans leurs protéines cibles est important puisque cette capacité à se loger dans la protéine influencera sur leurs performances. Des tests doivent donc être



(a) Etat initial du cube (b) Etat final du cube

FIG. 1.2 – Trouver une suite de rotations élémentaires pour passer le cube de Rubik de l'état décrit par la figure de gauche à celui décrit par la figure de droite, c'est planifier une trajectoire.

réalisés afin de savoir si un médicament nouvellement synthétisé va pouvoir pénétrer facilement la protéine qu'il doit cibler.

Une fois que des modèles géométriques sont appliqués aux molécules, la résolution se rapproche de problèmes d'assemblage tels que l'insertion d'une pièce mécanique dans un moteur sur une chaîne de montage.

1.2 L'espace d'état

Nous avons vu dans les exemples précédents que le vecteur $\mathbf{x}$ identifiant l'état d'un système varie en fonction du problème à résoudre. Sa taille est égale au nombre de degrés de liberté du système considéré. Les composantes du vecteur peuvent différer par leur caractère dénombrable ou non. L'*espace des états* (aussi appelé espace des configurations) est un ensemble $\mathbb{X}$ contenant l'ensemble des $\mathbf{x}$ respectant les contraintes définies par ou pour le système.

Reprenons l'exemple du jeu vidéo où le personnage se déplace sur une grille de dimension 100×100. L'espace d'état sera l'ensemble discret

$$\mathbb{X} = \{\{1, 2, \ldots, 100\} \times \{1, 2, \ldots, 100\}\}. \tag{1.1}$$

Outre l'aspect discret ou continu que revêt l'espace d'état, il peut contenir des états interdits (existence d'obstacles par exemple). L'espace $\mathbb{X}$ doit donc prendre en compte les contraintes dues à l'environnement du système. On parle alors d'états atteignables par le système, compte-tenu de son environnement.

L'ensemble $\mathbb{X}$ est ainsi partitionné en deux sous-ensembles :

- $\mathbb{X}_{\text{free}}$ contient l'ensemble des états atteignables par le système compte-tenu des contraintes extérieures.
- $\mathbb{X}_{\text{obs}}$ contient l'ensemble des états qui ne sont pas atteignables à cause des contraintes extérieurs.

Les relations $\mathbb{X} = \mathbb{X}_{\text{free}} \cup \mathbb{X}_{\text{obs}}$ et $\mathbb{X}_{\text{free}} = \mathbb{X} \setminus \mathbb{X}_{\text{obs}}$ sont ainsi toujours vérifiées.

L'existence de $\mathbb{X}_{\text{obs}}$ complique nettement la recherche de solutions puisque prendre en compte le modèle du système ne suffit plus. Il devient nécessaire de se soucier aussi de l'environnement dans

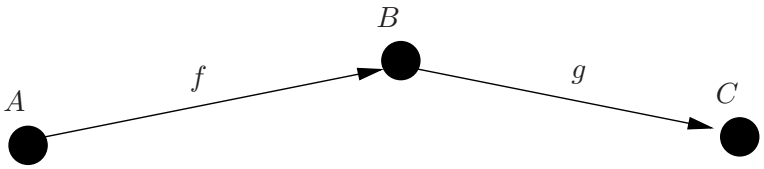


FIG. 1.3 – Exemple d'arbre de changement d'état

lequel le système évolue. Deux approches sont classiquement utilisées pour chercher des solutions dans $\mathbb{X}_{\text{free}}$ et non plus dans $\mathbb{X}$. La première consiste à calculer a priori, avant la phase de résolution du problème, l'ensemble des configurations atteignables $\mathbb{X}_{\text{free}}$. Une fois cet ensemble défini, l'algorithme de planification restreindra ses recherches aux états lui appartenant. La seconde approche utilise un test de collision en ligne durant la phase de résolution du problème. Ce test permet de dire si un état $\mathbf{x}$ de $\mathbb{X}$ appartient au sous-ensemble atteignable $\mathbb{X}_{\text{free}}$ ou au contraire à $\mathbb{X}_{\text{obs}}$. L'utilisation de l'une ou l'autre de ces approches dépendra du problème à résoudre et du type de planificateur utilisé.

1.3 Exemples de planificateurs

On distingue deux grandes familles de planificateurs, suivant que l'espace d'état est discret ou continu.

1.3.1 Cas discret

La résolution d'un problème de planification discret comporte deux étapes.

Dans la première, on construit un graphe représentant les différents états possibles de l'objet. Chaque état correspond à un nœud du graphe. Si un état B peut être atteint à partir d'un autre état A à la suite d'un déplacement élémentaire, alors ces deux états sont reliés par une arête orientée de A à B. Ainsi, si le mouvement élémentaire f permet de passer de l'état A à l'état B et si g permet de passer de B à C, on aura l'arbre de changement d'état représenté sur la figure 1.3.

Dans la seconde étape, on parcourt ce graphe afin de trouver une méthode de passage de l'état initial vers l'état à rejoindre qui soit optimale en un sens à définir.

Si la liste des commandes pour passer de A à C est évidente dans l'exemple trivial de la figure 1.3, la planification peut bien sûr devenir beaucoup plus difficile si le graphe se complexifie. Il est donc nécessaire d'utiliser des algorithmes de parcours de graphe efficaces.

Ainsi, la planification dans un espace discret se résume à la construction d'un graphe puis à la recherche d'un chemin optimal dans celui-ci. Le critère d'optimalité peut être la longueur du chemin par exemple. Les algorithmes trouvant ce genre de chemin ne manquent pas et sont utilisés dans de nombreux domaines comme les réseaux de télécommunications lorsqu'il faut par exemple acheminer des données entre deux nœuds.

Voyons maintenant quelques exemples d'algorithmes de recherche dans un graphe.

1.3.1.1 Algorithme de Dijkstra

Cet algorithme [4,5] permet de trouver un chemin optimal en coût dans un arbre connexe (c'est-à-dire un graphe où il existe un chemin entre chaque paire de nœuds), et où le poids associé à chacune des arêtes est positif ou nul. Il est utilisé dans de nombreux domaines tels que les réseaux informatiques où il est connu dans l'algorithme de routage OSPF (*Open Shortest Path First*).

Algorithme 1 DijkstraInit(**in** : G, $\mathbf{x}_{\text{init}}$)

1: **PourTout** $\mathbf{x} \in \mathbb{S}$ **faire**
2: $\quad$ d($\mathbf{x}$)$\leftarrow \infty$
3: $\quad$ antécédent[$\mathbf{x}$]$\leftarrow$indéfini
4: **FinPour**
5: d($\mathbf{x}_{\text{init}}$)$\leftarrow 0$

Le principe de cet algorithme est de parcourir l'ensemble des nœuds de l'arbre en partant du nœud de départ n_{init} et en associant initialement une distance infinie entre chacun des autres nœuds du graphe et n_{init}. Au cours du parcours, qui se fait en explorant successivement tous les nœuds voisins du nœud étudié, si le coût réel à partir de n_{init} est inférieur au coût actuellement enregistré, ce coût est modifié et n_{init} devient le père de ce nœud voisin.

Notations La recherche de chemins s'effectue dans le graphe $G = (\mathbb{S}, \mathbb{A})$, où $\mathbb{S}$ représente l'ensemble des sommets du graphe et $\mathbb{A}$ l'ensemble des arêtes de ce graphe. Si l'on adapte cet algorithme au problème de planification :

- $\mathbb{S}$ représente l'ensemble des états possibles de l'objet, un sommet est donc équivalent à une valeur de l'état.
- Chaque arête $\mathbf{a}_{\text{i}} \in \mathbb{A}$ contient à la fois le coût du lien (à définir en fonction du critère à optimiser) ainsi que le déplacement (ou la commande) qui permet de passer de l'état situé à l'origine de l'arête vers l'état situé à sa destination.

L'état initial est noté $\mathbf{x}_{\text{init}}$, l'état que l'on veut rejoindre est $\mathbf{x}_{\text{goal}}$. On définit un ensemble $\mathbb{Q}$ qui va contenir les sommets du graphe qui restent à visiter. La fonction `Coût`($\mathbf{x}_{\text{i1}}$,$\mathbf{x}_{\text{i2}}$) renvoie le coût pour suivre l'arête de $\mathbf{x}_{\text{i1}}$ à $\mathbf{x}_{\text{i2}}$. La fonction d($\mathbf{x}_{\text{i}}$) renvoie le coût pour aller de $\mathbf{x}_{\text{init}}$ à $\mathbf{x}_{\text{i}}$.

Initialisation Avant même de pouvoir rechercher un chemin dans le graphe G, il faut initialiser chacun des coûts associés aux arêtes de G. C'est le rôle de l'algorithme 1. À chacun des sommets du graphe, il associe un coût infini et un prédécesseur indéfini car on ne connaît pour le moment aucun chemin entre un sommet quelconque $\mathbf{x}$ et le sommet initial $\mathbf{x}_{\text{init}}$. On affecte un coût nul pour rester en $\mathbf{x}_{\text{init}}$.

Fonction principale Une fois accomplie l'initialisation de l'ensemble des coûts, la recherche du chemin de moindre coût peut commencer (algorithme 2). L'ensemble des sommets restant à visiter $\mathbb{Q}$ est initialisé à $\mathbb{S} \setminus \{\mathbf{x}_{\text{init}}\}$, puisqu'il faut visiter tous les nœuds sauf le nœud de départ.

Tous les sommets contenus dans $\mathbb{Q}$ sont à parcourir (ligne 3 de l'algorithme 2). L'algorithme ne se terminera que lorsque tous les sommets ont été visités. On choisit alors à la ligne 4 de l'algorithme 2, grâce à la fonction `DijkstraMin` (algorithme 3), le nœud de $\mathbb{Q}$ le plus proche de $\mathbf{x}_{\text{init}}$, noté $\mathbf{x}_1$. Une fois ce nœud choisi, on l'enlève de $\mathbb{Q}$ puisqu'il va être visité. On teste alors, pour chaque voisin $\mathbf{x}_2$ (ligne 6) de $\mathbf{x}_1$ si le coût de $\mathbf{x}_{\text{init}}$ à $\mathbf{x}_1$ auquel on ajoute celui de l'arête ($\mathbf{x}_1$, $\mathbf{x}_2$) est plus faible que le coût d($\mathbf{x}_2$) déjà enregistré de $\mathbf{x}_{\text{init}}$ à $\mathbf{x}_2$. Si c'est le cas, on en conclut qu'il est avantageux de passer par $\mathbf{x}_1$ pour rejoindre $\mathbf{x}_2$ et l'on met à jour les enregistrements de coût (ligne 8) et d'antécédent (ligne 9) pour $\mathbf{x}_2$.

La fonction principale `Dijkstra` renvoie le tableau des antécédents pour chacun des nœuds de $\mathbb{S}$. Il ne reste plus qu'à les parcourir grâce à la fonction `DijkstraReturnPath` (algorithme 4) en partant du nœud de destination (ligne 1 de l'algorithme 4) et en remontant d'antécédent en antécédent jusqu'à arriver au nœud de départ, le seul n'ayant pas d'antécédent défini (lignes 2 à 5 de l'algorithme 4).

Algorithme 2 Dijkstra(**in** : G, $\mathbf{x}_{\text{init}}$, $\mathbf{x}_{\text{goal}}$, **out** : chemin[])

```
1: DijkstraInit(G, x_init)
2: Q = S \ {x_init}
3: TantQue Q ≠ ∅ faire
4:    x_1 ← DijkstraMin(Q)
5:    Q ← Q \ {x_1}
6:    PourTout Voisin x_2 de x_1 faire
7:       Si d(x_1)+Coût(x_1, x_2)<d(x_2) alors
8:          d(x_2) ← d(x_1)+Coût(x_1, x_2)
9:          antécédent[x_2]← x_1
10:      FinSi
11:   FinPour
12: FinTantQue
13: Retourner DijkstraReturnPath(x_goal, antécédent[])
```

Algorithme 3 DijkstraMin(**in** : $\mathbb{Q}$, **out** : $\mathbf{x}$)

```
1: d_min ← ∞
2: x ← 0
3: PourTout x_i ∈ Q faire
4:    Si d(x_i)< d_min alors
5:       x ← x_i
6:       d_min ← d(x_i)
7:    FinSi
8: FinPour
9: Retourner x
```

Algorithme 4 DijkstraReturnPath(**in** : $\mathbf{x}_{\text{goal}}$, antécédent[],**out** : chemin[]

```
1: chemin[].vider()
2: cible ← x_goal
3: TantQue antécédent[cible] est défini faire
4:    Insérer cible au début de chemin[]
5:    cible ← antécédent[cible]
6: FinTantQue
7: Retourner chemin[]
```

1.3.1.2 Algorithme A*

La principale différence entre l'algorithme A* et l'algorithme de Dijkstra est l'utilisation d'une heuristique visant une obtention rapide d'un résultat acceptable. Cet algorithme est donc à privilégier dans les applications privilégiant un temps de calcul faible par rapport à la garantie de l'optimalité.

Ces caractéristiques en font l'algorithme privilégié en recherche de chemin dans le domaine de l'intelligence artificielle, et notamment celui majoritairement utilisé dans les jeux vidéo.

L'algorithme A* utilise une heuristique à définir pour estimer le coût à payer pour rejoindre l'état final désiré à partir d'un nœud quelconque du graphe. Cette estimation permet d'explorer le graphe en commençant par les nœuds les plus prometteurs.

Notations Nous utilisons la notation suivante :
- $\mathbb{V}$ est l'ensemble des nœuds déjà visités.
- La fonction $g(\mathbf{x}_\text{i})$ renvoie le coût réel du nœud de départ $\mathbf{x}_\text{init}$ à un nœud quelconque $\mathbf{x}_\text{i}$ du graphe.
- La fonction $h(\mathbf{x}_\text{i})$ renvoie une estimée du coût d'un nœud quelconque $\mathbf{x}_\text{i}$ du graphe au nœud d'arrivée $\mathbf{x}_\text{goal}$.
- La fonction $f(\mathbf{x}_\text{i}) = g(\mathbf{x}_\text{i}) + h(\mathbf{x}_\text{i})$ estime donc le coût de $\mathbf{x}_\text{init}$ à $\mathbf{x}_\text{goal}$ lorsque l'on passe par $\mathbf{x}_\text{i}$.

Initialisation L'initialisation de l'algorithme A* (algorithme 5) s'effectue aux lignes 1 et 2. L'ensemble des nœuds déjà visités est vide et l'ensemble des nœuds à visiter contient uniquement le nœud initial $\mathbf{x}_\text{init}$.

Fonction principale L'algorithme 5 s'exécute tant que la liste $\mathbb{Q}$ des nœuds à visiter n'est pas vide (ligne 3) ou bien lorsqu'un chemin est trouvé (ligne 7).

Le parcours des nœuds à visiter se fait dans l'ordre inverse des coûts estimés du chemin. Cette fonction `A*ShortestPath` est appelée à la ligne 4 de l'algorithme 5 et est décrite par l'algorithme 6. Elle parcourt l'ensemble des nœuds contenus dans $\mathbb{Q}$ et renvoie celui qui génère le coût estimé par la fonction f le plus faible.

Aux lignes 5 et 6 de l'algorithme 5, le nœud choisi est inséré dans $\mathbb{V}$ puisqu'il va être visité et retiré de l'ensemble à visiter $\mathbb{Q}$.

Si ce nœud est le nœud d'arrivée $\mathbf{x}_\text{goal}$, on peut arrêter l'exécution de l'algorithme puisqu'un chemin a été trouvé. Il suffit alors de parcourir les nœuds d'antécédent en antécédent en partant de $\mathbf{x}_\text{goal}$ pour obtenir le chemin.

Si au contraire ce nœud n'est pas $\mathbf{x}_\text{goal}$, on met à jour les coûts pour chacun de ses successeurs au sein du graphe. Ces successeurs sont contenus dans l'ensemble $\mathbb{T}$ (ligne 15). Si la distance permettant de rejoindre un des successeurs $\mathbf{t} \in \mathbb{T}$ en passant $\mathbf{x}$ est plus faible que la distance déjà stockée (ligne 17), alors ces distances sont mises à jour (lignes 18 et 19) et l'antécédent de $\mathbf{t}$ devient $\mathbf{x}$ (ligne 20). De plus, si le nœud $\mathbf{t}$ ne fait pas encore partie de l'ensemble des nœuds à visiter $\mathbb{Q}$ (ligne 21), il y est ajouté (ligne 22).

L'algorithme continue ainsi jusqu'à ce qu'un chemin soit trouvé ou que $\mathbb{Q}$ ne contienne plus de nœud à visiter. Dans ce cas l'algorithme indique qu'aucune solution n'a été trouvée (ligne 27).

1.3.2 Cas continu

Nous avons présenté dans la partie précédente des planificateurs permettant de résoudre des problèmes de planification discrets. Une fois l'arbre représentant les états construit, il suffit de parcourir cet arbre pour obtenir des chemins.

Algorithme 5 A*(**in** : G, $\mathbf{x}_{\text{init}}$, $\mathbf{x}_{\text{init}}$, **out** : chemin[])

```
1: 𝕍 ← ∅
2: ℚ ← x_init
3: TantQue ℚ ≠ ∅ faire
4:    x ← A*ShortestPath(ℚ)
5:    𝕍 ← ℚ ∪ {x}
6:    ℚ ← 𝕍 \ {x}
7:    Si x = x_goal alors
8:       cible ← x_goal
9:       TantQue Antécédent(cible) est défini faire
10:         Insérer cible au début de chemin[]
11:         cible ← Antécédent(cible)
12:      FinTantQue
13:      Retourner chemin[]
14:   FinSi
15:   𝕋 ← (Successeurs(x) ∉ ℝ)
16:   PourTout t ∈ 𝕋 faire
17:      Si t ∉ ℚ ou g(t) > g(x)+Coût(x,t) alors
18:         g(t) ← g(x)+Coût(x,t)
19:         f(t) ← g(t)+h(t)
20:         Antécédent(t) ← x
21:         Si t ∉ ℚ alors
22:            ℚ ← ℚ ∪ {t}
23:         FinSi
24:      FinSi
25:   FinPour
26: FinTantQue
27: Retourner NoSolution
```

Algorithme 6 A*ShortestPath(**in** : $\mathbb{Q}$, **out** : $\mathbf{x}$)

```
1: d_min ← ∞
2: x ← 0
3: PourTout x_i ∈ ℚ faire
4:    Si f(x_i)< d_min alors
5:       x ← x_i
6:       d_min ← f(x_i)
7:    FinSi
8: FinPour
9: Retourner x
```

Ce type de planificateur ne peut résoudre les problèmes liés à un espace d'état continu, pour lesquels la difficulté vient autant de la construction de l'arbre des états que de son parcours.

1.3.2.1 Extension des algorithmes de recherche discrète

La première idée qui vient à l'esprit est de procéder par extension des algorithmes de recherche de chemins que nous venons de présenter pour le cas discret. La phase de parcours de l'arbre ne varie pas, c'est la phase de construction du graphe qui évolue.

Utilisation d'une grille régulière La première façon de discrétiser l'espace est d'utiliser une grille régulière que l'on pose en quelque sorte sur l'environnement de départ. Le principe de ce genre de planificateur est simple. Il se décompose en quatre phases :

- Dans une première étape, illustrée par la figure 1.4(a), l'espace des états est discrétisé en utilisant une grille. Chaque point de croisement de la grille représente une valeur prise par l'état.
- Dans une deuxième étape, illustrée par la figure 1.4(b), les états $\mathbf{x}_{\text{init}}$ et $\mathbf{x}_{\text{goal}}$ sont chacun reliés par une arête à l'un des nœuds de l'arbre les plus proches.
- Dans une troisième étape, illustrée par la figure 1.4(c), les nœuds et arêtes du graphe interdits par les contraintes, c'est-à-dire ceux appartenant à $\mathbb{X}_{\text{obs}}$ sont supprimés.
- Dans une quatrième et dernière étape, illustrée par la figure 1.4(d), un algorithme de recherche de chemins discret, comme ceux que nous avons présentés précédemment, est utilisé afin de trouver un chemin entre $\mathbf{x}_{\text{init}}$ et $\mathbf{x}_{\text{goal}}$ dans ce graphe.

Dans ce genre de planificateur, le nombre de voisins est au maximum de quatre pour une planification dans le plan. Le nombre des chemins possibles est alors très réduit et dépend fortement de la résolution de la grille. Le choix de cette résolution peut d'ailleurs s'avérer délicat. Si la résolution est trop faible (grille trop grossière), des chemins qui existent dans l'espace des configurations continu ne seront pas trouvés. Ainsi, le passage d'une porte par un robot peut s'avérer impossible comme illustré sur la figure 1.5.

Sur la figure 1.5(d), on arrive à une configuration où le graphe n'est plus connecté, et la recherche du chemin entre $\mathbf{x}_{\text{init}}$ et $\mathbf{x}_{\text{goal}}$ devient impossible.

Pour résoudre ce problème de façon partielle, on peut diminuer le pas de la grille chaque fois que la taille de grille actuelle ne permet pas de trouver un chemin. L'algorithme bouclera infiniment si aucun chemin n'existe. Il faut donc spécifier un pas minimum au dessous duquel on considère qu'aucun chemin ne peut être trouvé.

1.3.2.2 Probabilistic Roadmap Planner

Le *Probabilistic Roadmap Planner* [6–10], ou PRM, est une extension naturelle du planificateur à grille régulière qui permet de dépasser les problèmes dûs à la rigidité de la grille et à sa résolution. Le principe et les étapes sont identiques à ceux du planificateur utilisant une grille régulière. La différence fondamentale vient de la forme de la grille qui perd son côté rigide et est construite de façon incrémentale :

- Un point est choisi aléatoirement dans l'environnement.
- On utilise un planificateur local afin de joindre ce point avec l'un des nœuds du graphe existant.
- Une fois que $\mathbf{x}_{\text{init}}$ et $\mathbf{x}_{\text{goal}}$ appartiennent au graphe, la phase de recherche de chemin peut commencer.

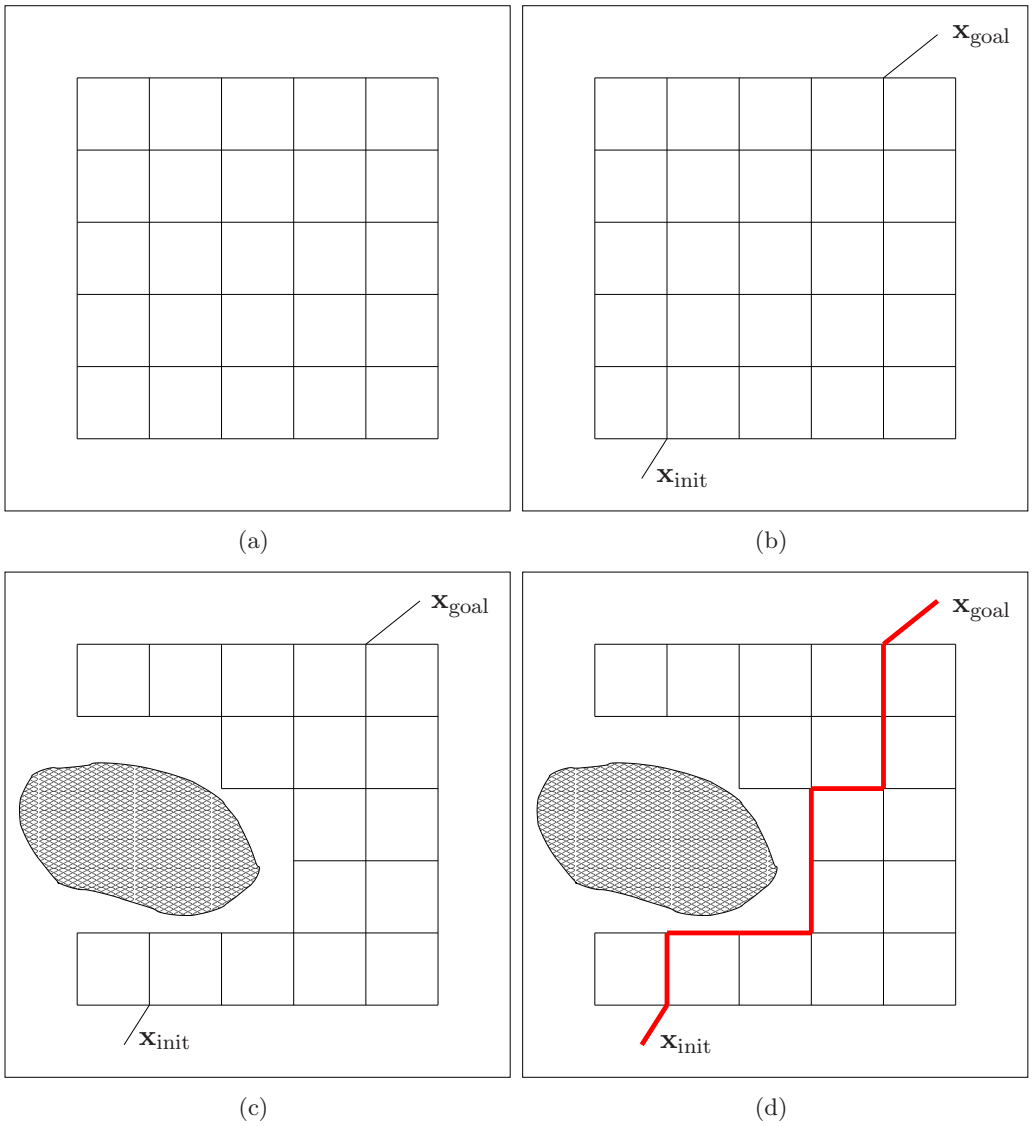


FIG. 1.4 – Les étapes d'un planificateur de type grille

1.3.3 Champs de potentiels

Les méthodes utilisant les champs de potentiels [11, 12] sont historiquement les premières à avoir cherché à résoudre les problèmes de planification en s'éloignant des techniques de résolution discrètes. Cette méthode utilise cependant une grille à haute résolution qui servira à placer les nœuds du chemin.

Le principe s'inspire du mouvement d'une particule chargée attirée par le point d'arrivée et repoussée par les obstacles. Il faut donc définir une fonction de potentiel p qui dépend du problème à résoudre. Cette fonction peut être considérée comme une pseudo-métrique qui caractérise la distance entre un état quelconque et l'état d'arrivée. La plupart du temps, cette fonction se décompose en deux termes :

- un terme d'attractivité qui caractérise la distance entre un état et l'arrivée,
- un terme de répulsivité qui caractérise la distance entre cet état et les obstacles de l'environnement.

La construction d'une telle fonction de potentiel est difficile et largement heuristique [13].

La construction du chemin comporte deux phases à chaque itération.

- On choisit comme origine de la nouvelle arête le dernier nœud ajouté au chemin.

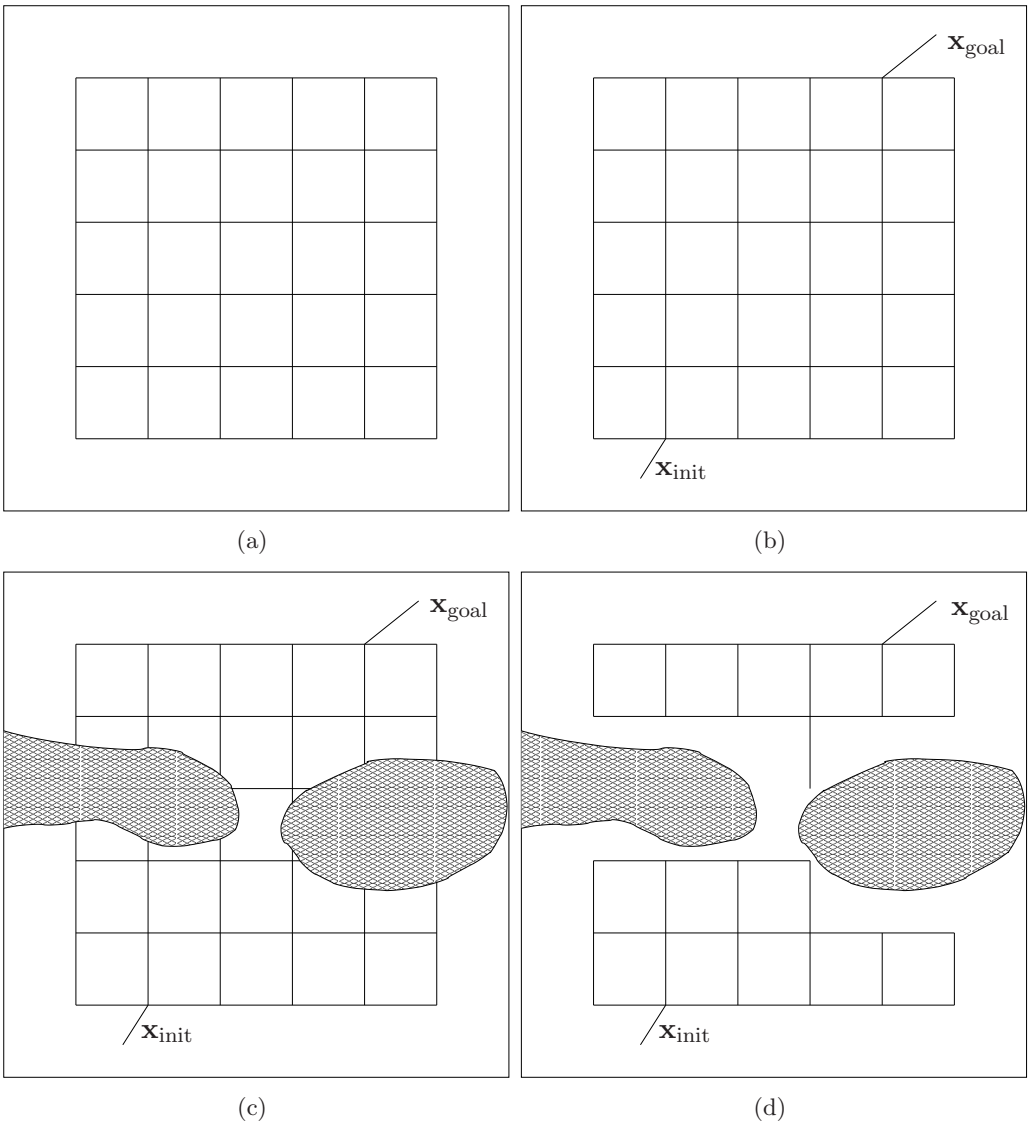


FIG. 1.5 – Une résolution trop faible peut empêcher la découverte d'un chemin, qui pourtant existe, par un planificateur de type grille

– On choisit ensuite la destination de cette arête afin que celle-ci minimise la fonction de potentiel p. La destination de cette arête doit se situer sur la grille de discrétisation à proximité de son nœud d'origine (la valeur du pas de discrétisation est ajoutée ou enlevée de chacune de ses coordonnées).

Ainsi, à chaque étape, le chemin s'allonge en tentant de minimiser la fonction de potentiel. La distance entre le point courant et l'arrivée se réduit à chaque étape tout en assurant que ce point reste éloigné des obstacles de l'environnement.

Il arrive parfois que la fonction p ne puisse plus être réduite à cause de l'existence d'un minimum local. Dans ce cas, une direction de déplacement est choisie au hasard pendant un nombre d'itérations spécifié (cette période de choix de la direction au hasard est appelée une marche aléatoire) ou jusqu'à ce que g soit réduite. Il n'y a bien sûr aucune garantie que cette démarche permette de trouver un chemin globalement optimal au sens de la fonction de potentiel choisie.

La méthode des champs de potentiel s'avère ingénieuse et permet de résoudre des problèmes dans des espaces d'état de grande dimension. Cependant, l'utilisation d'une marche aléatoire peut rendre la recherche longue et générer des chemins inutilement longs et biscornus pas toujours adaptés

aux problèmes à résoudre. De plus, définir la fonction de potentiel implique de régler de nombreux paramètres afin de rendre les heuristiques efficaces. L'algorithme s'avère donc très difficile à mettre en œuvre.

1.3.4 Fil d'Ariane

L'algorithme du fil d'Ariane [14, 15] cherche à étendre un arbre G dans $\mathbb{X}$ pour explorer le plus de territoire possible à chaque itération. Il enchaîne des phases d'exploration et de recherche :

- En phase d'exploration, l'algorithme n'a pas pour ambition de rejoindre l'arrivée : un nouveau nœud $\mathbf{x}_{\text{new}}$ est choisi aléatoirement pour étendre l'arbre à partir d'un nœud $\mathbf{x}$. Ce nouveau nœud est choisi de façon à ce qu'il puisse être facilement connecté à $\mathbf{x}$ et qu'il soit le plus loin possible de tous les autres nœuds de G.
- En phase de recherche, l'algorithme cherche ensuite à rejoindre l'arrivée en étendant l'arbre depuis $\mathbf{x}_{\text{new}}$.

Contrairement à la méthode utilisant les champs de potentiel qui s'efforçait uniquement de s'approcher de l'arrivée à chaque itération, on cherche ici également à explorer l'espace d'état. Ainsi, le problème des minima locaux ne se pose plus (ou en tout cas pas avec la même acuité).

Chaque phase d'exploration implique la résolution d'un problème d'optimisation. On cherche en effet à trouver un nœud qui puisse à la fois se connecter rapidement et être le plus éloigné possible des autres nœuds de l'arbre. Un algorithme génétique est utilisé pour résoudre ce problème dans [15] mais s'avère difficile à mettre en œuvre à cause des nombreux paramètres à définir. Rien ne garanti, là aussi, que la solution trouvée sera optimale.

1.3.5 Planificateur à expansion

Ce planificateur [16, 17] adopte la même philosophie que le fil d'Ariane, en essayant de générer des nœuds qui explorent l'espace d'état. Le planificateur choisit un nœud $\mathbf{x}$ comme origine d'une extension avec une probabilité inversement proportionnelle au nombre de nœuds de G se situant dans son voisinage. Ainsi, les nœuds les plus isolés sont privilégiés. Dans une deuxième phase, un nœud $\mathbf{x}_{\text{new}}$ est créé au hasard dans le voisinage de $\mathbf{x}$. L'algorithme choisit alors d'insérer $\mathbf{x}_{\text{new}}$ dans le graphe G avec une probabilité inversement proportionnelle au nombre de nœuds situés dans son voisinage.

Ce planificateur permet de résoudre de nombreux problèmes pour peu que la façon de qualifier le voisinage d'un nœud soit adaptée au problème. Les performances se dégradent dans des environnements sinueux ou en forme de labyrinthe.

1.3.6 Marche aléatoire

Il est également possible de trouver un chemin en utilisant uniquement une marche aléatoire [18]. A chaque itération, le nœud choisi comme origine de l'expansion de l'arbre est le dernier ajouté. La commande, en terme de direction et d'amplitude est choisie suivant une loi normale multidimensionnelle paramétrée par le nombre de succès consécutifs des dernières tentatives : plus le nombre de succès est important et plus l'amplitude du mouvement le sera. Dans [19], la marche aléatoire est utilisée conjointement à la méthode des potentiels.

Cette méthode s'avère traverser difficilement les longs couloirs étroits.

1.3.7 Rapidly-exploring Random Trees

Le *Rapidly-exploring Random Trees* [20–25], ou RRT, est une méthode incrémentale de construction d'arbre qui a pour but d'explorer rapidement et uniformément l'espace d'état. Cet arbre peut

Algorithme 7 RRT(**in** : $K \in \mathbb{N}$, $\mathbf{x}_{\text{init}} \in \mathbb{X}_{\text{free}}$, $\mathbb{X}_{\text{goal}} \subset \mathbb{X}_{\text{free}}$, $\Delta t \in \mathbb{R}^+$, **out** : G)

1: G.init($\mathbf{x}_{\text{init}}$)
2: $i = 0$
3: **Répéter**
4: $\quad \mathbf{x}_{\text{rand}} \leftarrow$ RandomState($\mathbb{X}_{\text{free}}$)
5: $\quad \mathbf{x}_{\text{new}} \leftarrow$ RRTExtend(G, $\mathbf{x}_{\text{rand}}$)
6: **Jusqu'à** i++> K **ou** ($\mathbf{x}_{\text{new}} \neq$**null et** $\mathbf{x}_{\text{new}} \in \mathbb{X}_{\text{goal}}$)
7: **Retourner** G

Algorithme 8 RRTExtend(**in** : G, $\mathbf{x}_{\text{rand}}$, **out** : $\mathbf{x}_{\text{new}}$)

1: $\mathbf{x}_{\text{near}} \leftarrow$ NearestNeighbor(G, $\mathbf{x}_{\text{rand}}$)
2: $\mathbf{u} \leftarrow$ SelectInput($\mathbf{x}_{\text{rand}}$, $\mathbf{x}_{\text{near}}$)
3: $\mathbf{x}_{\text{new}} \leftarrow$ NewState($\mathbf{x}_{\text{near}}$, $\mathbf{u}$, Δt)
4: **Si** CollisionFreePath($\mathbf{x}_{\text{near}}$, $\mathbf{x}_{\text{new}}$, $\mathbf{u}$, Δt) **alors**
5: $\quad G$.AddNode($\mathbf{x}_{\text{new}}$)
6: $\quad G$.AddEdge($\mathbf{x}_{\text{near}}$, $\mathbf{x}_{\text{new}}$, $\mathbf{u}$)
7: $\quad$ **Retourner** $\mathbf{x}_{\text{new}}$
8: **FinSi**
9: **Retourner null**

être utilisé comme planificateur lorsque le nœud racine est associé à l'état de départ $\mathbf{x}_{\text{init}}$. L'état d'arrivée $\mathbf{x}_{\text{goal}}$ est alors atteint si un des nœuds de l'arbre lui correspond. Ce planificateur est à classer dans les planificateurs dits incrémentaux ou à diffusion.

La génération d'un arbre de type RRT suit un principe de génération de points aléatoires et de tentatives successives de rapprochement des feuilles de l'arbre de ce point. Le but n'est pas d'aller en ce point mais de s'en rapprocher. On évite ainsi l'utilisation d'un planificateur local au sein même du planificateur global pour trouver une trajectoire entre deux états successifs. Le point tiré aléatoirement est donc là uniquement pour fournir une orientation à l'agrandissement du graphe et ne sera pas inclus directement dans l'arbre.

Comme nous l'avons vu précédemment, planifier une trajectoire peut être vu comme la recherche dans un espace $\mathbb{X}$ d'une trajectoire continue entre $\mathbf{x}_{\text{init}}$ et $\mathbf{x}_{\text{goal}}$. Cette trajectoire devra être contenue dans $\mathbb{X}_{\text{free}}$, l'ensemble des configurations admissibles compte-tenu des contraintes imposées par l'environnement. On doit donc nécessairement avoir $\mathbf{x}_{\text{init}} \in \mathbb{X}_{\text{free}}$ et $\mathbf{x}_{\text{goal}} \in \mathbb{X}_{\text{free}}$ dès l'initialisation de l'algorithme.

Une équation d'état de la forme $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$ est également utilisée afin de relier les évolutions de l'état aux choix faits pour la commande du système $\mathbf{u}$. Le vecteur $\mathbf{u}$ est sélectionné parmi un ensemble $\mathbb{U}$ d'entrées possibles. Ainsi, en intégrant $\mathbf{f}$ sur un intervalle de temps Δt donné à $\mathbf{u}$ constant, le nouvel état $\mathbf{x}_{\text{new}}$ peut être obtenu.

Par exemple, pour un mobile holonome pouvant se diriger dans toutes les directions du plan sans contrainte on prendra :

$$\mathbf{f}(\mathbf{x}, \mathbf{u}) = \mathbf{u}, \tag{1.2}$$

avec $\mathbf{x} \in \mathbb{R}^2$ et $\mathbf{u} \in \mathbb{R}^2$.

1.3.7.1 Algorithme

Observons maintenant plus en détail l'algorithme RRT [20] de construction de cet arbre (algorithmes 7 et 8) afin de mieux en comprendre le fonctionnement et ainsi d'identifier quelques propriétés intéressantes.

L'algorithme de génération de l'arbre RRT (algorithme 7) prend cinq paramètres en entrée :
- K représente le nombre de nœuds à ajouter au graphe G après la création du point initial $\mathbf{x}_{\text{init}}$.
- $\mathbf{x}_{\text{init}}$ est le point de départ de l'expansion de l'arbre G. Il sera utilisé comme racine de cet arbre et doit appartenir à $\mathbb{X}_{\text{free}}$.
- $\mathbb{X}_{\text{goal}}$ l'ensemble d'arrivée, qui doit être un sous-ensemble de $\mathbb{X}_{\text{free}}$. $\mathbb{X}_{\text{goal}}$ est défini comme un ensemble englobant $\mathbf{x}_{\text{goal}}$ puisque la probabilité de rejoindre $\mathbf{x}_{\text{goal}}$ exactement est nulle.
- Δt est l'intervalle de temps sur lequel sera intégrée la fonction de changement d'état $\mathbf{f}$ (avec une entrée $\mathbf{u}$ fixée).

L'algorithme 7 doit également connaître l'environnement afin de savoir grâce à un test de collision si un état appartient à $\mathbb{X}_{\text{free}}$ ou à $\mathbb{X}_{\text{obs}}$. Il commence par l'ajout de l'état initial $\mathbf{x}_{\text{init}}$ à l'arbre G. Il sera donc la racine de cet arbre. A la ligne 3, démarre la boucle d'ajout des nœuds dans l'arbre. Cette boucle ne s'arrêtera à la ligne 6 que si le nombre de nœuds générés atteint K ou bien si l'un des nœuds générés a atteint la zone d'arrivée $\mathbb{X}_{\text{goal}}$. Dans cette boucle, un état aléatoire est tout d'abord choisi dans l'ensemble $\mathbb{X}_{\text{free}}$ à la ligne 4. C'est le rôle de la fonction `RandomState`. Ensuite, la fonction `RRTExtend` est exécutée, c'est elle qui va générer de nouveaux nœuds et éventuellement les ajouter à l'arbre G.

La fonction `RRTExtend` (Algorithme 8) prend deux paramètres en entrée : le graphe G et le nœud $\mathbf{x}_{\text{rand}}$ tiré aléatoirement. Elle renvoie le nœud nouvellement généré.
- À la ligne 1, l'état $\mathbf{x}_{\text{near}}$ appartenant à G et étant le plus proche de $\mathbf{x}_{\text{rand}}$ est renvoyé par la fonction `NearestNeighbor`. Cette fonction est à définir en fonction du problème à résoudre.
- À la ligne 2, la fonction `SelectInput` renvoie la commande $\mathbf{u}$ qui sera appliquée à $\mathbf{x}_{\text{near}}$. Le choix de cette commande peut être aléatoire ou bien correspondre au minimum d'un critère donné. Par exemple, on pourrait choisir la commande $\mathbf{u}$ qui, une fois appliquée à l'objet situé en $\mathbf{x}_{\text{near}}$ pendant Δt lui permettrait de s'approcher au plus près de $\mathbf{x}_{\text{rand}}$.
- À la ligne 3, la fonction `NewState` utilise l'état choisi $\mathbf{x}_{\text{near}}$ ainsi que la commande choisie $\mathbf{u}$ afin d'intégrer l'équation d'état sur l'intervalle de temps Δt.
- On regarde à la ligne 4 si la trajectoire élémentaire entre les deux états $\mathbf{x}_{\text{near}}$ et $\mathbf{x}_{\text{new}}$ est entièrement contenue dans l'ensemble $\mathbb{X}_{\text{free}}$. C'est ce qui est fait dans la fonction `CollisionFreePath`.
- Si cette trajectoire est sûre, le nœud $\mathbf{x}_{\text{new}}$ ainsi que l'arête contenant la commande $\mathbf{u}$ sont ajoutés au graphe G.
- Finalement, à la ligne 8, le nouveau nœud généré est renvoyé.

La figure 1.6 illustre le fonctionnement du RRT dans un environnement en 2D. L'arbre explore de façon uniforme l'espace d'état jusqu'à ce qu'il rejoigne un état contenu dans $\mathbb{X}_{\text{goal}}$.

Examinons maintenant les principales propriétés du RRT.

L'arbre s'étend vers les zones inexplorées de $\mathbb{X}_{\text{free}}$ Considérons le cas où $\mathbb{X}_{\text{free}}$ est borné et convexe et réalisons plusieurs itérations de l'algorithme dans le cas d'un objet se déplaçant sur un plan.

La figure 1.7 montre que l'arbre se developpe suivant quatre branches principales, chacune se dirigeant vers l'un des coins. Au bout de 100 générations de nœuds, les quatre branches commencent à s'étendre. On les voit nettement mieux sur le deuxième schéma où elles se sont bien différenciées et explorent chacune un coin de $\mathbb{X}_{\text{free}}$. Finalement, avec 6000 nœuds générés, il est très difficile de distinguer les branches tant l'arbre devient dense. L'exploration de l'espace des configurations en est alors à un stade très avancé.

Comme il a été dit précédemment, la façon dont $\mathbf{x}_{\text{rand}}$ est généré va influencer les directions d'expansion des branches de l'arbre. On peut également considérer cette expansion en regardant les diagrammes de Voronoï [21] associés à chaque nœud du graphe.

Définition Diagramme de Voronoï.

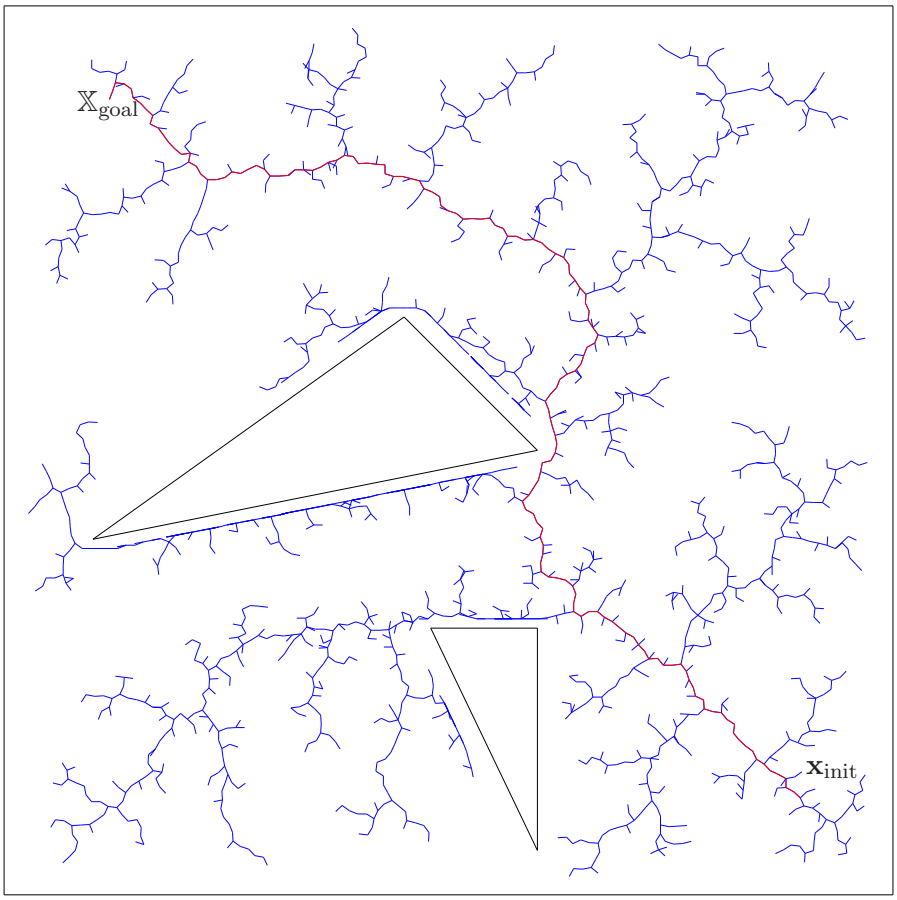


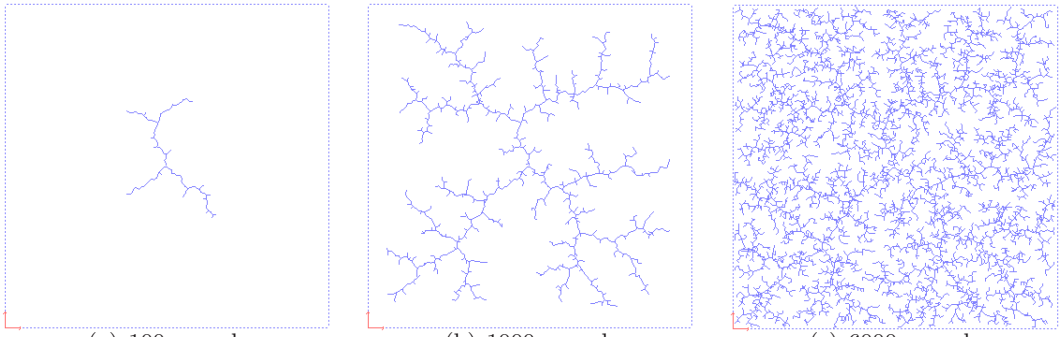
FIG. 1.6 – Trajectoire planifiée par un RRT

> Soit $\mathbb{S}$ un ensemble de n points (ou sites) de l'espace euclidien en dimension d. Pour chaque point s de $\mathbb{S}$, la région de Voronoï $\text{Vor}(s)$ de s est l'ensemble des points de l'espace qui sont plus proches de s que de tous les autres points de $\mathbb{S}$. Le diagramme de Voronoï $V(\mathbb{S})$ est la décomposition de l'espace formée par les régions de Voronoï des points de $\mathbb{S}$ (figure 1.8).

Si $\mathbb{S}$ est l'ensemble des nœuds de l'arbre générés par RRT, on s'aperçoit que les régions de Voronoï de taille importante se situent à la périphérie de l'arbre. Or, le tirage des points fait que les nœuds à la périphérie de l'arbre sont plus souvent choisis en tant que $\mathbf{x}_{\text{near}}$ pour l'expansion. Ces nœuds correspondant à ceux ayant une grande région de Voronoï, on en conclut qu'un RRT est construit en cassant les grandes régions de Voronoï à chaque étape. Cette remarque n'implique bien sûr pas que la région la plus grande soit cassée à chaque itération. C'est en effet un constat fait en moyenne lors de la construction de l'arbre.

Il est éclairant de voir comment réagit un RRT lorsqu'il se déploie sur un disque (figure 1.9). Les résultats sont conformes aux attentes. On voit ainsi les branches de l'arbre se développer sans direction préférentielle vers les bords du disque.

Le graphe généré par RRT est toujours connecté puisque chaque nouveau nœud est connecté au graphe pré-existant. Le fait que les coordonnées du nouveau nœud soient calculées en prenant en compte l'état $\mathbf{x}_{\text{near}}$ et une entrée $\mathbf{u} \in \mathbb{U}$ implique que $\mathbf{x}_{\text{new}}$ sera connecté à $\mathbf{x}_{\text{near}}$ (lui-même faisant partie de l'arbre) par l'intermédiaire de $\mathbf{u}$, et donc à l'ensemble du graphe.



(a) 100 noeuds (b) 1000 noeuds (c) 6000 noeuds

FIG. 1.7 – Génération d'un arbre aléatoire

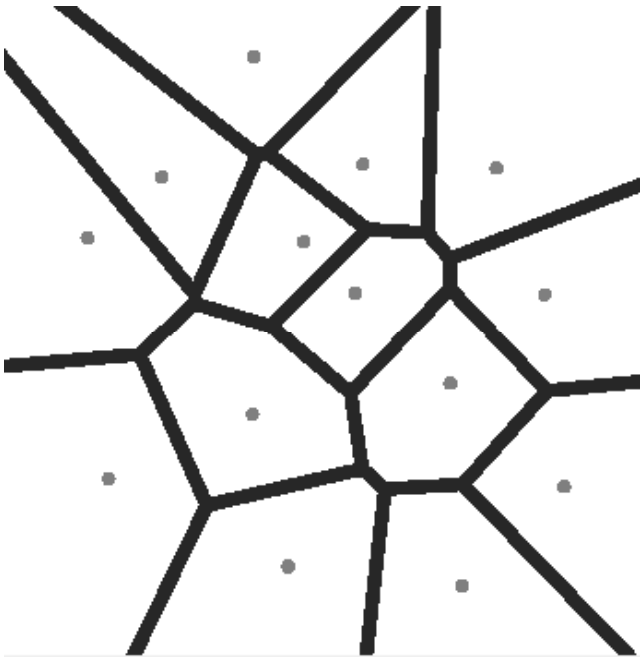


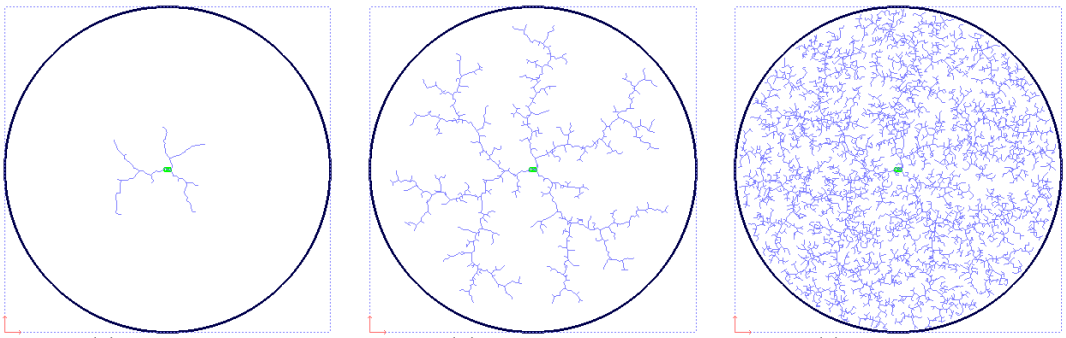
FIG. 1.8 – Exemple de régions de Voronoï

L'algorithme est complet probabilistiquement Kuffner a prouvé [23] que la probabilité de trouver une trajectoire tend vers 1 quand le nombre de nœuds tend vers l'infini si l'ensemble des trajectoires résolvant le problème est de mesure non nulle.

Depuis sa création en 1998, le RRT a connu de nombreuses évolutions qui permettent dans certains cas d'améliorer ses performances dans la recherche de trajectoires. Nous présentons ici une liste non-exhaustive d'améliorations.

Dual-RRT Le Dual-RRT utilise deux arbres aléatoires pour accélérer l'exploration. Un des arbres a comme racine le point de départ, l'autre le point d'arrivée. On considère qu'une trajectoire est trouvée lorsque les deux arbres se rejoignent ou, de façon plus réaliste, quand ils atteignent des états très proches.

RRT-*Goalbias* Comme RRT parcourt aléatoirement tout $\mathbb{X}$ il va se retrouver à un moment ou à un autre (si les contraintes au niveau de l'environnement le permettent) près de $\mathbb{X}_{\text{goal}}$. Toutefois,



(a) 100 noeuds (b) 1000 noeuds (c) 6000 noeuds

FIG. 1.9 – Déploiement d'un RRT sur un disque

le temps mis pour trouver une trajectoire peut s'avérer relativement long si l'on n'influence pas le planificateur afin qu'il se dirige vers l'arrivée.

Ceci suggère que les performances pourraient être améliorées si l'on arrivait à diriger les branches de l'arbre vers la destination. Pour cela, LaValle et Kuffner proposent une variante du RRT nommée RRT-*Goalbias* [21]. La modification se situe au niveau de la fonction de tirage du point aléatoire $\mathbf{x}_{\text{rand}}$. Au lieu de tirer des points sur tout l'espace, la fonction va renvoyer $\mathbb{X}_{\text{goal}}$ avec une probabilité p et tirer des points ailleurs avec une probabilité de $(1-p)$. Même avec p faible (≈ 0.05), RRT-*Goalbias* converge fréquemment vers la destination plus rapidement que le RRT classique. Cependant, si p est trop grand, le planificateur aura tendance à se diriger, quelqu'en soit le coût, vers $\mathbb{X}_{\text{goal}}$ et ainsi rester bloqué. Mentionnons ici encore que rien ne peut être prouvé quant au comportement en temps fini sur un problème donné.

Les figures 1.10 et 1.11 illustrent le fonctionnement de RRT-*Goalbias* pour deux probabilités p différentes. Sur la figure 1.10, $p = 0.1$. Le gain en temps de calcul et en nombre de nœuds générés est important par rapport au RRT classique. Lorsque $p = 0.5$, le planificateur tend à aller en ligne droite malgré la présence du mur. Il n'a donc d'autre choix que de le longer. Durant cette phase, le planificateur tente toujours de rejoindre $\mathbb{X}_{\text{goal}}$ en ligne droite malgré le mur. Ainsi de nombreux nœuds générés sont supprimés puisqu'ils ne peuvent passer le test de collision. Les performances en terme de nombres de nœuds générés pour trouver la trajectoire sont donc moins bonnes que lorsque $p = 0.1$.

RRT-*Goalzoom* RRT-*Goalzoom* [21] est une variante de RRT-*Goalbias* permettant de rajouter de la souplesse dans la sélection de la commande. Au lieu de choisir $\mathbf{x}_{\text{rand}} \in \mathbb{X}_{\text{goal}}$ avec une probabilité p, RRT-*Goalzoom* va tirer les points dans un disque circulaire contenant $\mathbb{X}_{\text{goal}}$ et inclus dans $\mathbb{X}$ avec une probabilité p et dans le reste de $\mathbb{X}$ avec une probabilité $(1-p)$. Le disque est construit de la façon suivante :

- son centre est celui de $\mathbb{X}_{\text{goal}}$, noté $\mathbf{x}_{\text{goal}}$,
- son rayon est $\min_i d(\mathbf{x}_i, \mathbf{x}_{\text{goal}})$ (c'est-à-dire la distance qui sépare $\mathbf{x}_{\text{goal}}$ du nœud du graphe qui lui est le plus proche).

Au fur et à mesure de l'avancement de l'algorithme, le rayon du cercle dans lequel sont tirés les points va donc diminuer, le disque tendant finalement vers $\mathbf{x}_{\text{goal}}$.

Les figures 1.12 et 1.13 illustrent le fonctionnement de RRT-*Goalzoom* pour $p = 0.1$ pour la figure 1.12 et $p = 0.5$ pour la figure 1.13.

La trajectoire générée par RRT-*Goalzoom* se révèle moins directe que celle générée par RRT-*Goalbias* mais ceci peut faciliter la rechercher d'une solution, notamment dans des environnements

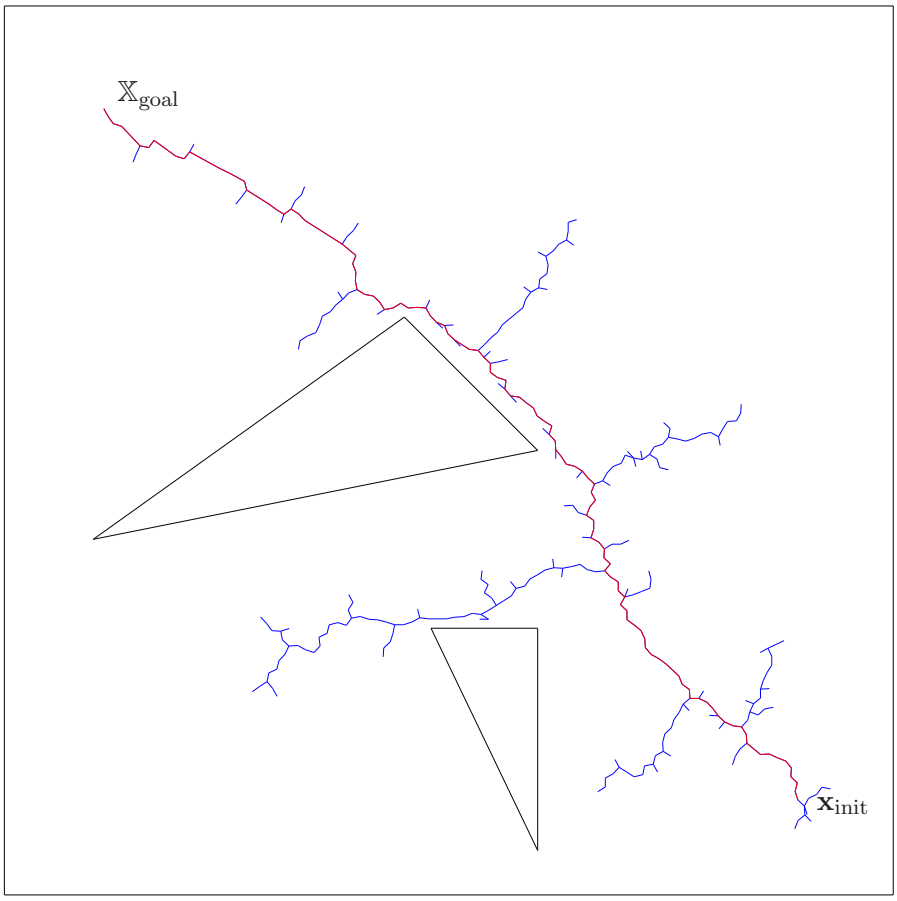


FIG. 1.10 – Trajectoire planifiée par un RRT-*Goalbias* avec $p = 0.1$

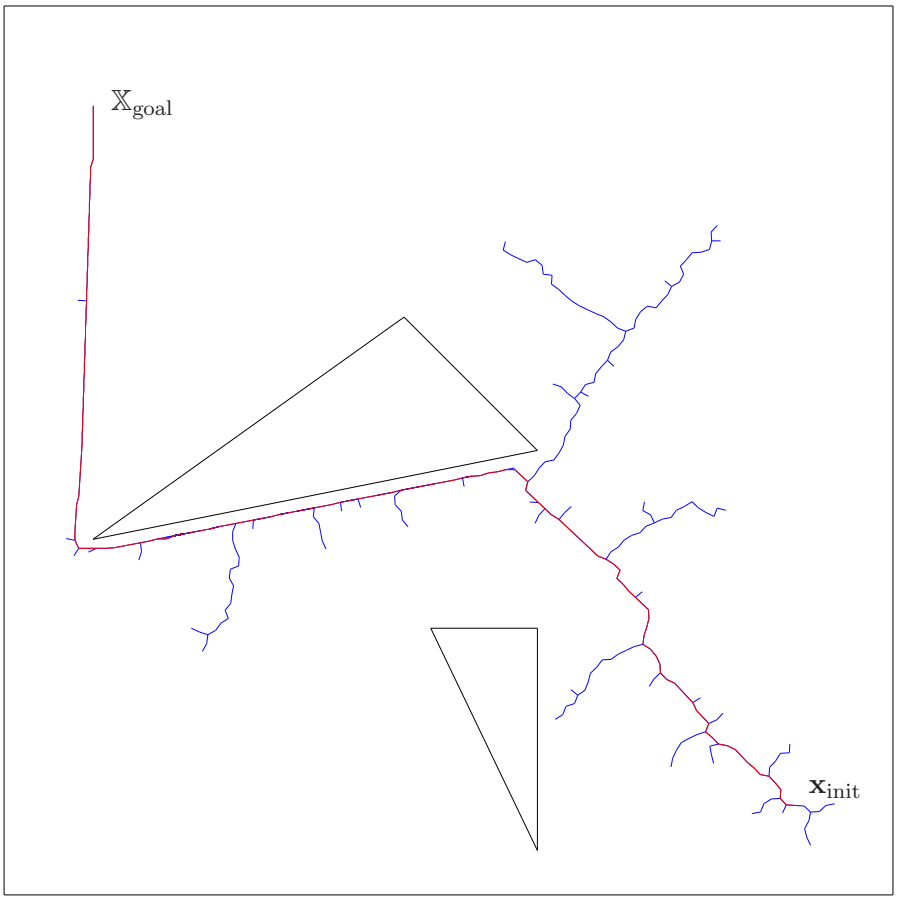


FIG. 1.11 – Trajectoire planifiée par un RRT-*Goalbias* avec $p = 0.5$

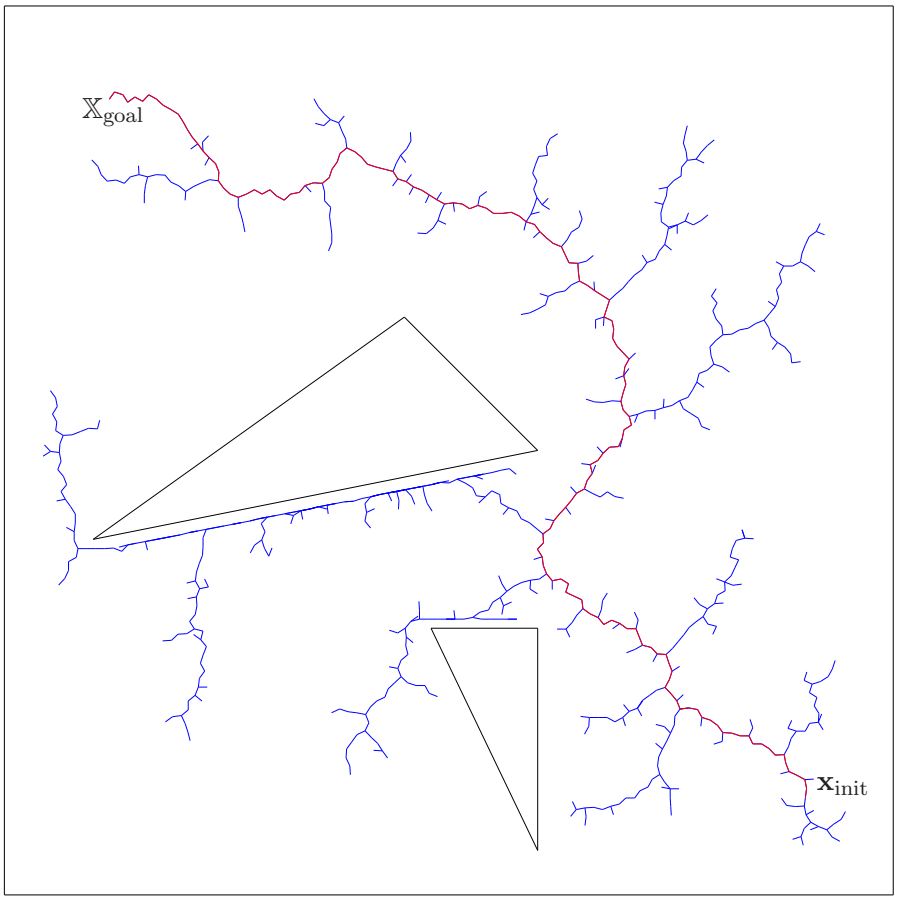


FIG. 1.12 – Trajectoire planifiée par un RRT-*Goalzoom* avec $p = 0.1$

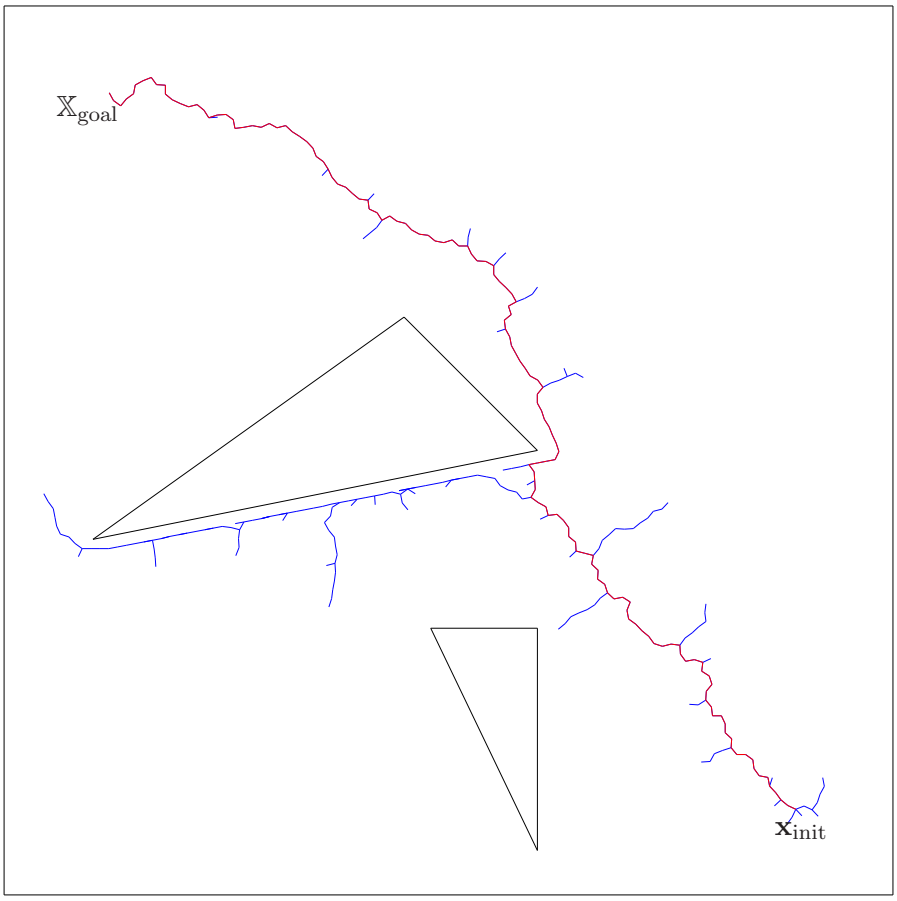


FIG. 1.13 – Trajectoire planifiée par un RRT-*Goalzoom* avec $p = 0.5$

contenants de nombreux obstacles, là encore sans aucune garantie.

Connect La primitive `Connect` peut remplacer la primitive `Extend` utilisée dans le RRT. Rappelons que lorsque `Extend` est utilisée, on calcule un nouvel état $\mathbf{x}_{\text{new}}$ avec `NewState` et si la trajectoire élémentaire entre $\mathbf{x}_{\text{near}}$ et $\mathbf{x}_{\text{new}}$ appartient à $\mathbb{X}_{\text{free}}$, $\mathbf{x}_{\text{new}}$ est ajouté à l'arbre. Une fois ce nœud ajouté, l'algorithme reprend depuis le début.

La primitive `Connect` itère au contraire la primitive `Extend` jusqu'à ce que le nœud créé ne fasse plus partie de $\mathbb{X}_{\text{free}}$ ou que le temps imparti à cette commande soit dépassé (on peut par exemple imposer de ne pas appliquer la commande pendant plus de 10 Δt). Ainsi, la commande choisie par la fonction `SelectInput` est appliquée au système jusqu'à ce qu'il rencontre un obstacle ou l'arrivée. Un exemple du fonctionnement du RRT utilisant la primitive `Connect` est visible sur la figure 1.14. Sur cet exemple, nous voyons que les segments de droite sont plus longs, et l'arrivée dans $\mathbb{X}_{\text{goal}}$ plus rapide.

Cette primitive permet une résolution plus rapide des problèmes de planification puisque les étapes les plus coûteuses de l'algorithme, qui sont les étapes de recherche du plus proche voisin ainsi que la recherche de la commande, ne sont plus exécutées à la création de chaque nouveau nœud. Cette amélioration est tout de même à relativiser. Le gain est certes important pour les systèmes holonomes, dans ce cas, le mouvement se fait en ligne droite. Ainsi, en utilisant `Connect`, on va tenter de se rapprocher du point d'arrivée en ligne droite jusqu'à ce qu'un obstacle empêche le passage. Ainsi, si aucun obstacle ne gêne le passage et qu'il existe une ligne droite entre $\mathbf{x}_{\text{near}}$ et $\mathbf{x}_{\text{goal}}$, la trajectoire est trouvée en une seule itération de `Connect`.

Au contraire, pour les systèmes non-holonomes, le gain peut être nul. Par exemple, si nous nous plaçons dans le cadre de la robotique et que la commande unique est l'angle de courbure, il n'est pas rare de voir le planificateur faire parcourir au véhicule un cercle. Ce cercle est bien sûr utile puisqu'il permet d'explorer l'espace d'état, cependant, l'efficacité en terme de temps de calcul est nettement plus discutable. Ainsi, on ajoute un grand nombre de points sans avoir à calculer le plus proche voisin ni la commande à appliquer. Les nœuds correspondants seront ajoutés à l'arbre ce qui fait que le calcul du prochain plus proche voisin deviendra nettement plus coûteux. De plus, le déplacement en arc de cercles ne favorise pas forcément l'exploration de grandes zones.

La primitive `Connect` peut être couplée aux modifications du tirage d'état aléatoire comme *Goalbias*. Sur les figures 1.15 et 1.16 sont générées des trajectoires utilisant la primitive `Connect` avec $p = 0.1$ et $p = 0.5$ respectivement dans *Goalbias*. Ces trajectoires ont l'avantage d'être rapides à générer (peu de nœuds générés) et courtes grâce au changement de la primitive de tirage aléatoire.

1.4 Conclusion

Ce chapitre a présenté de façon générique le problème de planification ainsi que quelques méthodes de résolution de ce problème. Toutes ces méthodes considèrent un modèle déterministe du système et de son environnement. Les différentes incertitudes de modèle ainsi que celles liées à l'environnement ne sont pas prises en compte.

Nous présenterons dans les chapitres suivants la problématique de la planification de trajectoires en présence d'incertitudes ainsi que nos contributions dans ce contexte.

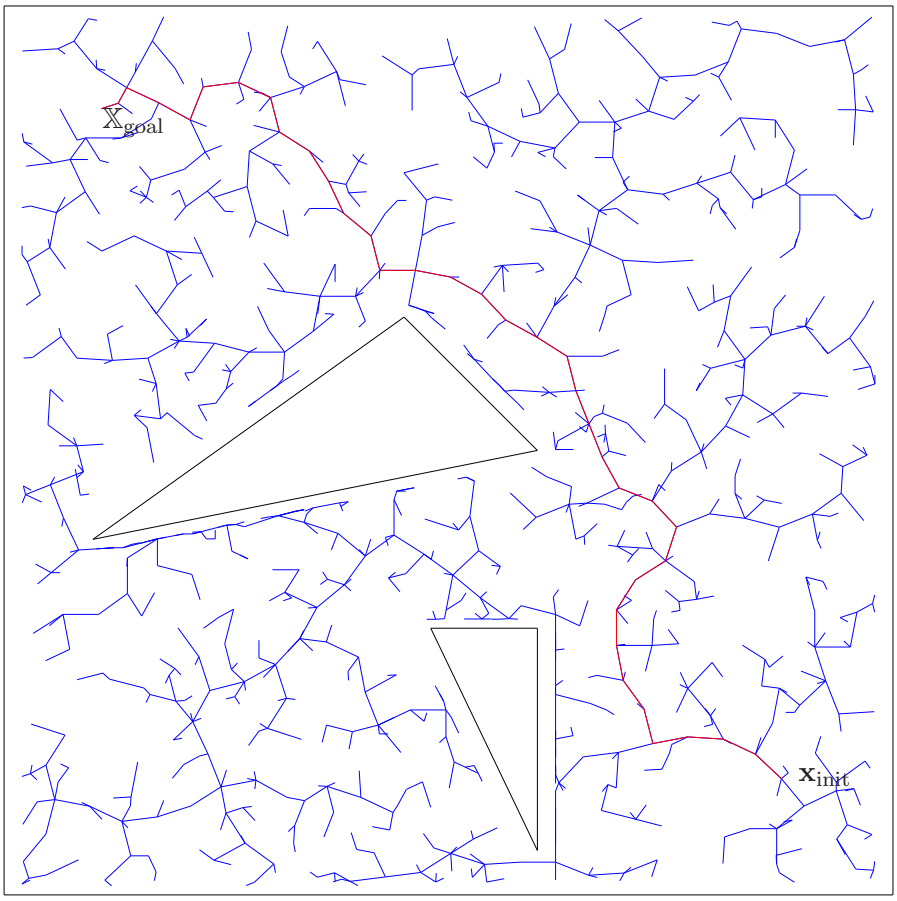


FIG. 1.14 – Trajectoire planifiée par un RRT utilisant la primitive Connect

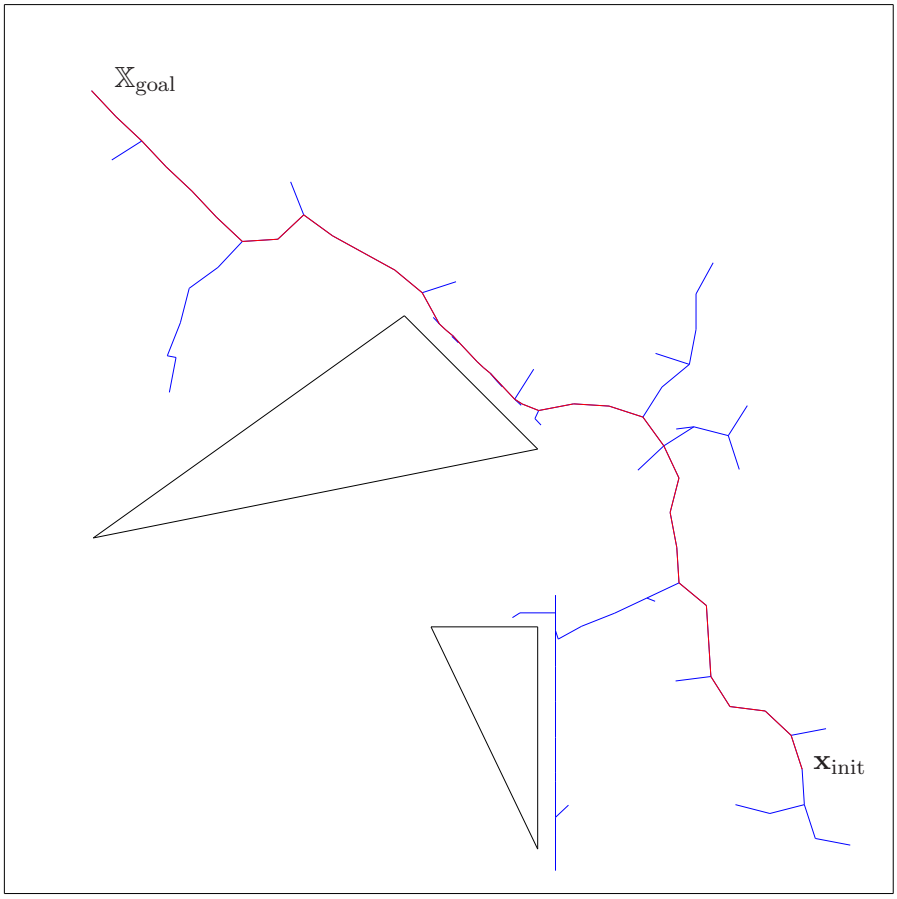


FIG. 1.15 – Trajectoire planifiée par un RRT-*Goalbias* de probabilité $p = 0.1$ utilisant la primitive Connect

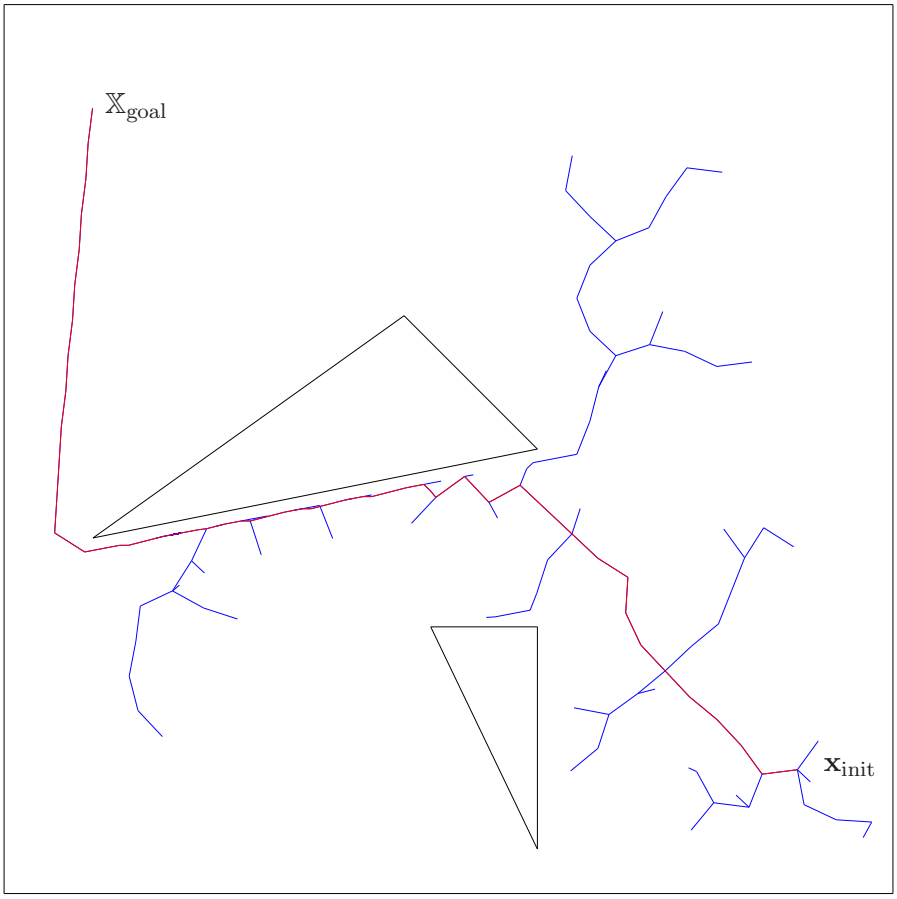


FIG. 1.16 – Trajectoire planifiée par un RRT-*Goalbias* de probabilité $p = 0.5$ utilisant la primitive Connect

Chapitre 2

Planification sous incertitudes

Sommaire

Le chapitre précédent a présenté le problème de planification de trajectoires dans un contexte où les incertitudes n'avaient pas leurs places. Les problèmes présentés et les méthodes de résolution proposées considéraient en effet un ensemble de modèles déterministes, que ce soit pour le système lui-même, pour la façon de le contrôler ou bien pour l'environnement dans lequel il évolue.

Cette approche permet de résoudre bon nombre de problèmes mais ne garantit pas que la solution retournée soit adaptée au système à déplacer. Ce système a en effet toujours des défauts intrinsèques, par exemple du jeu dans des engrenages ou des glissements au niveau d'une courroie. Le système n'est d'ailleurs pas seul en cause : un écart de température, une légère brise ou un plan légèrement incliné peut modifier les valeurs retournées par les capteurs du système. La description de l'environnement est nécessairement approximative.

Les commandes retournées par le planificateur peuvent, une fois appliquées au système, ne pas résoudre du tout le problème à cause de ces imperfections. Les conséquences peuvent être diverses ; le système peut sortir de $\mathbb{X}_{\text{free}}$ et se détériorer suite au heurt d'un obstacle. Il se peut également que dans le cadre d'un système totalement autonome, par exemple un drone d'exploration spatial, le système se perde et devienne ainsi inutilisable.

Pour pallier ce problème, il faut donc estimer l'influence de ces imperfections sur la réaction du système et prendre en compte ces imperfections durant la phase de planification. Le but est de s'assurer que, malgré ces erreurs, la solution retournée est fiable et permet des changements d'états sûrs.

Nous présentons dans ce chapitre plusieurs méthodes permettant la planification de trajectoires en tenant compte des incertitudes. Une première méthode basique est présentée dans le paragraphe 2.1. Dans le paragraphe 2.2, une méthode de planification en robotique et utilisant les capteurs du robot est présentée. Dans le paragraphe 2.3, certains états sont privilégiés, ceux situés dans des zones permettant une relocalisation. Finalement, le paragraphe 2.4 présente des techniques où les méthodes de planification traditionnelles sont utilisées dans un espace d'état étendu.

2.1 Chaînage inverse

Une première méthode très rustre pour tenir compte des incertitudes procède par chaînage inverse. Le principe est de partir de l'état d'arrivée et de déterminer quels sont les états à partir desquels on peut rejoindre cet état en propageant l'évolution de l'incertitude à l'envers. On doit donc trouver une suite d'actions en boucle ouverte $f_1, f_2, \ldots, f_n$ qui permet de rejoindre l'état d'arrivée désiré et qui vérifie

$$f_n\left(\ldots f_2\left(f_1\left(\mathbf{x}_{\text{init}}\right)\right)\ldots\right) \in \mathbb{X}_{\text{goal}}. \tag{2.1}$$

Cette méthode a été utilisée en robotique dans [26] et [27]. On considère que la zone d'arrivée est sûre et circulaire. Une zone sûre est une zone dans laquelle l'incertitude est considérée par la méthode comme annulée.

La méthode fait l'hypothèse que l'incertitude peut être caractérisée par des disques dont le rayon croit linéairement avec la distance au point d'arrivée. Comme les zones sûres sont des disques, ceci se traduit par des cônes (figure 2.1). On considère que n'importe quel point du cône peut rejoindre la zone sûre qui lui a donné naissance.

Le principe est donc de partir de la zone d'arrivée qui est sûre et de passer de zone sûre en zone sûre jusqu'à rejoindre le point de départ. Ainsi, le passage d'une zone sûre A à une autre zone sûre B est possible si le cône généré à partir de A a un point d'intersection avec B. Le cône sert donc uniquement à estimer la distance que peut faire le robot entre deux zones sûres. Le processus est itéré jusqu'à ce que l'on puisse rejoindre la zone de départ. Un exemple de trajectoire planifiée est visible sur la figure 2.2. Sur cette figure, le point de départ du système est I. L'arrivée est notée g_o,

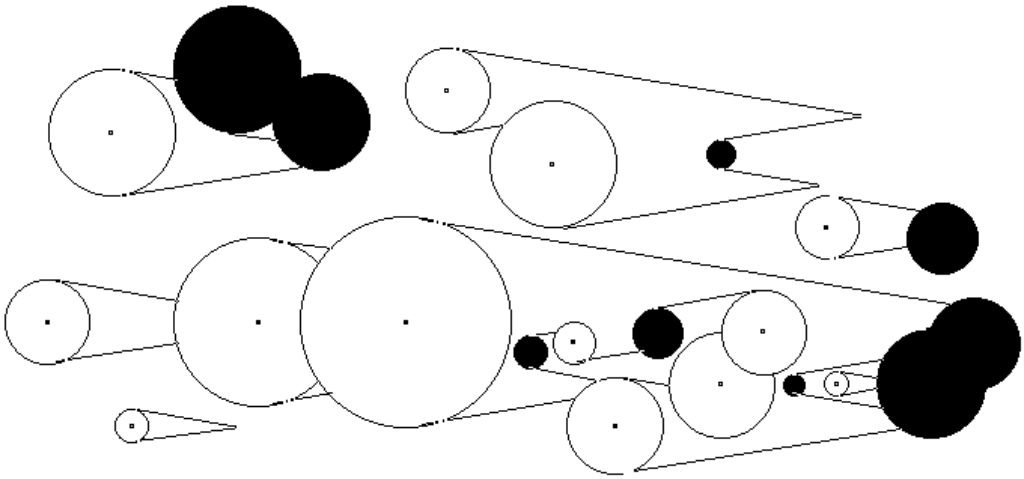


FIG. 2.1 – Exemple issu de [26] illustrant la projection inverse en présence d'obstacles (en noir)

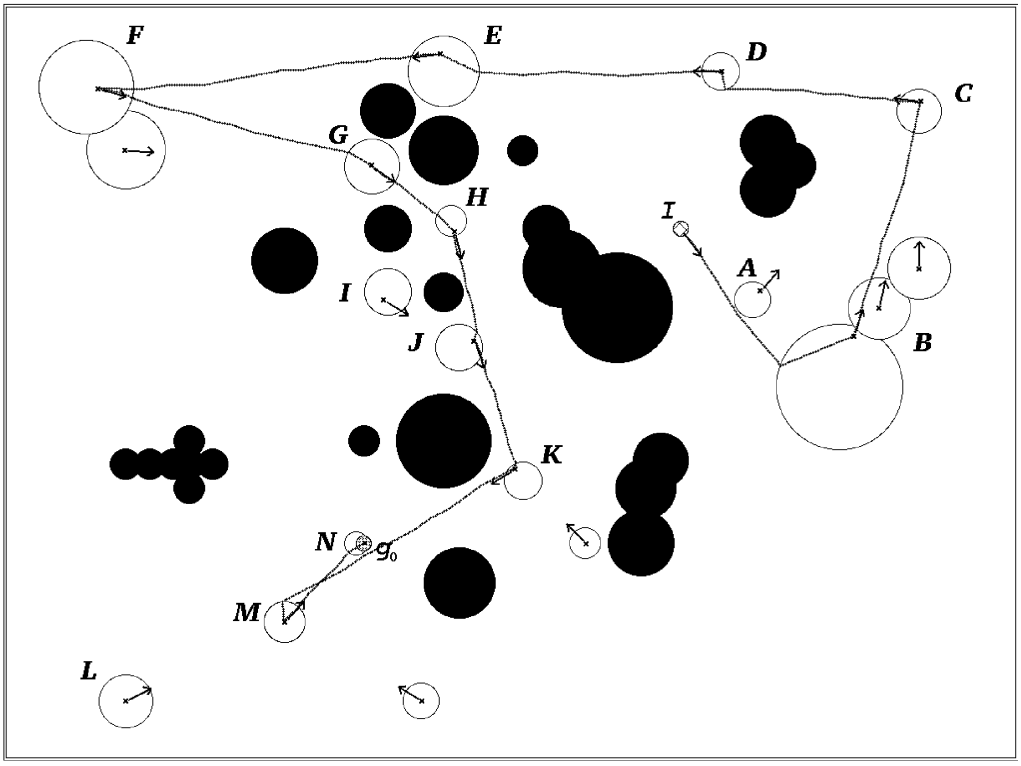


FIG. 2.2 – Figure issue de [26] illustrant les trajectoires générées en utilisant le chaînage inverse en présence d'obstacles (en noir)

il se trouve dans la zone N. Les zones sûres sont illustrées par des cercles blancs, les obstacles par des noirs.

Un problème majeure de cet algorithme réside bien évidemment dans la définition des zones de sûreté. Il est en effet impossible de concevoir dans le monde réel des zones dans lesquelles la localisation serait parfaite. De plus, ces amers doivent être placés avant le déplacement du système, ce qui implique qu'il n'est pas autonome. Enfin l'hypothèse sur l'évolution linéaire des incertitudes est difficile à justifier et suppose des réglages de paramètres délicats.

2.2 Planification avec capteurs de contact

La planification avec capteurs de contact [28,29] est également une technique utilisée en robotique. Elle comporte deux phases. Dans la première, une trajectoire est cherchée dans un espace discrétisé en prenant en compte les incertitudes. Le planificateur parcourt l'ensemble des nœuds de la grille en utilisant un front d'onde et attribue une incertitude pour chacun de ces nœuds.

L'incertitude est représentée par le rayon du cercle (plus généralement de la boule) situé autour d'un état. Ce rayon est supposé augmenter linéairement lorsque le système se déplace dans un espace libre (loin d'un objet grâce auquel il pourrait se relocaliser) et s'annule lorsque le système passe dans une zone de relocalisation (figure 2.3). Le rayon représentant la quantité d'incertitude ne diminue pas mais la boule se déforme au contact des obstacles (figure 2.4).

Afin de trouver une trajectoire, l'espace est discrétisé et une fonction de potentiel y est propagée. Une fois la trajectoire trouvée, on associe à chacun de ses nœuds une étiquette qui sera utilisée pour générer le plan en terme de mouvements élémentaires utilisant les capteurs (figure 2.5). Par exemple, si la trajectoire passe par un état proche d'un mur, le mouvement élémentaire sera « continuer tout droit jusqu'à arriver à un obstacle ». Si ensuite cette trajectoire continue le long du mur, le mouvement sera « longer le mur ». Les mouvements élémentaires présentés sur la figure 2.5 sont « suivre un mur », « suivre un mur jusqu'à atteindre un coin » et « passer de l'autre côté du mur ».

Un exemple de trajectoire complète est disponible sur la figure 2.6. Les cercles représentent l'incertitude qui grossie de façon linéaire dans les zones libres. Au contraire, dès que le système atteint un mur ou un objet permettant d'utiliser ses capteurs de contact, l'incertitude sur sa position s'annule jusqu'à ce qu'il ne soit plus en contact avec cet objet. Sur la figure 2.7 n'apparaît que la trajectoire issue de cette planification.

Tout comme pour le chaînage inverse, l'utilisation de zones de relocalisation rend cette méthode difficilement utilisable dans un environnement réel et l'hypothèse sur la propagation des incertitudes reste difficile à justifier.

2.3 Sensory uncertainty fields

Cette méthode, développée elle aussi dans le contexte de la robotique dans [30–32], ne considère pas la propagation de l'erreur le long de la trajectoire. Le principe est, une fois l'espace discrétisé par une grille standard, de calculer une *localisabilité* en chacun des points de cette grille.

La *localisabilité* est calculée en amont de la phase de planification. Ce processus hors-ligne permet donc de définir si le système, situé en un état donné, arrivera à s'y localiser aisément ou pas. La carte qui permet de connaître la *localisabilité* du système est appelée *Sensory Uncertainty Field* (SUF).

La construction de cette carte est complexe : elle s'appuie sur des mesures capteurs simulées ou bien sur des mesures réelles faites en toutes configurations du système. Des détails de construction de SUF, et notamment lors de la simulation de capteurs à ultrasons sont donnés dans [31,33]. D'autres détails de construction lors de l'utilisation de caméras sont donnés dans [34].

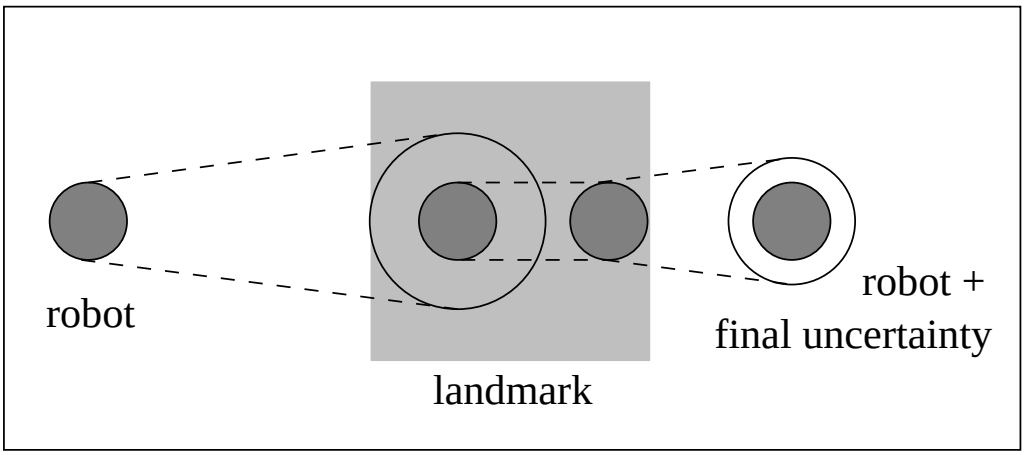


FIG. 2.3 – Annulation de l'incertitude lors de l'entrée dans une zone de relocalisation [29]

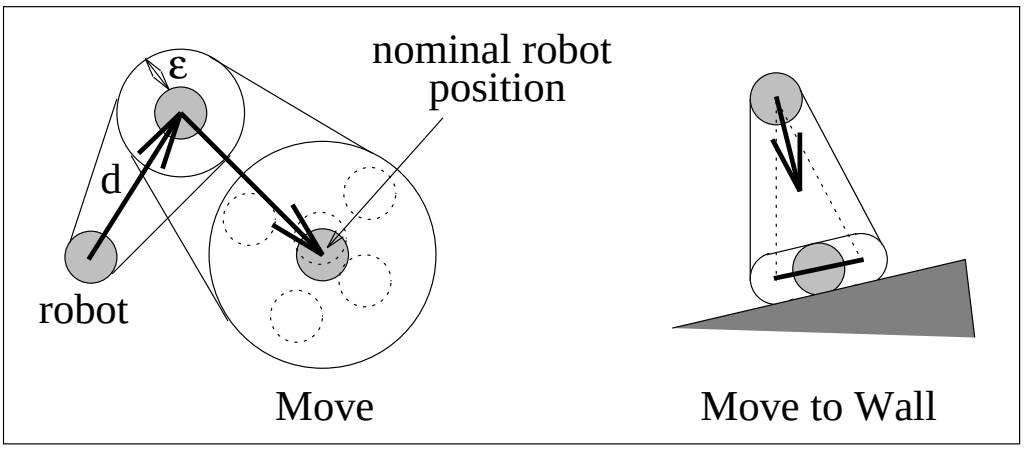


FIG. 2.4 – Déformation de la boule d'incertitude au contact d'un obstacle [29]

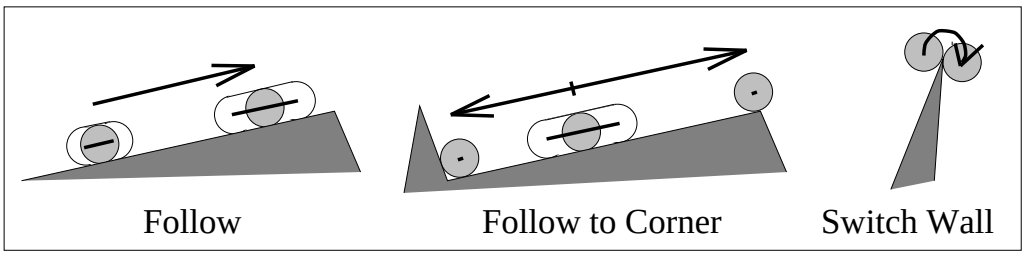


FIG. 2.5 – Mouvements élémentaires possibles [29]

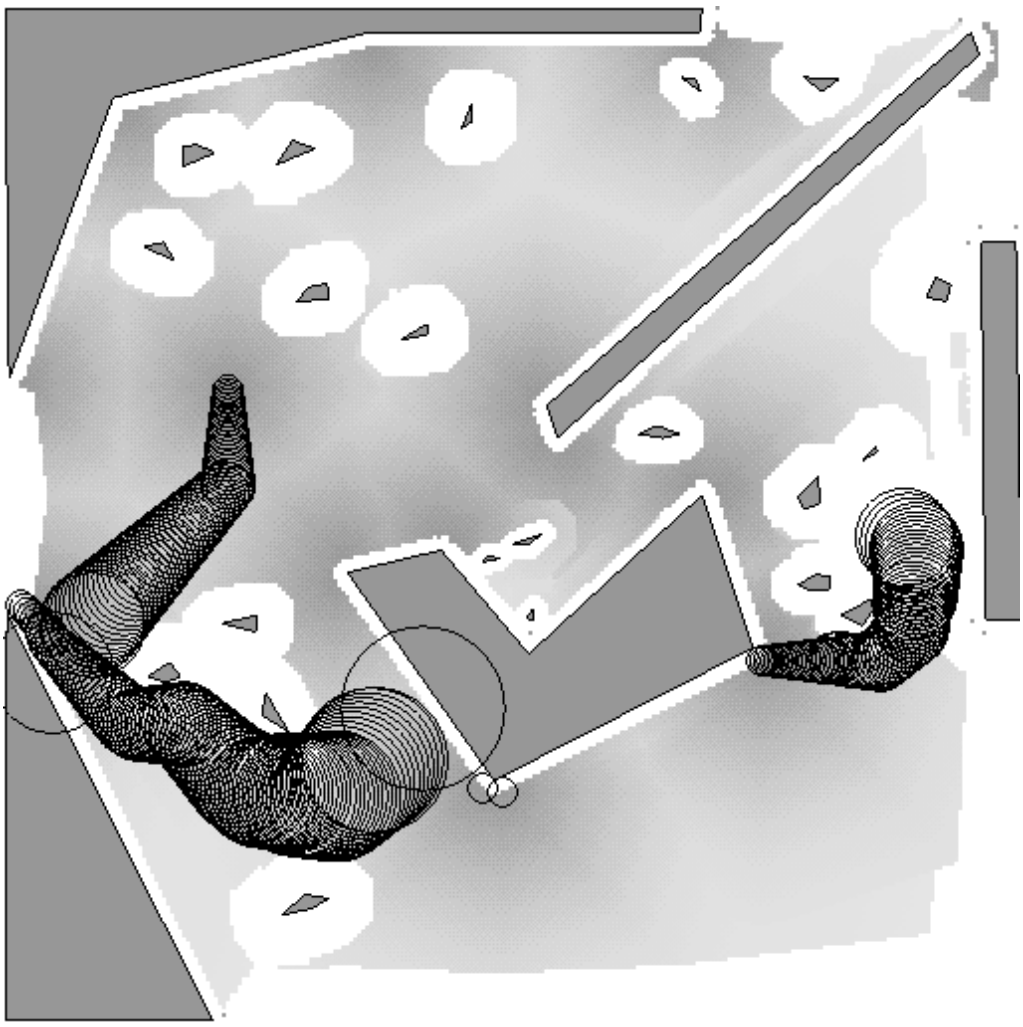


FIG. 2.6 – Trajectoire générée à l'aide du planificateur avec capteurs de contact, l'incertitude est représentée le long de cette trajectoire [29]

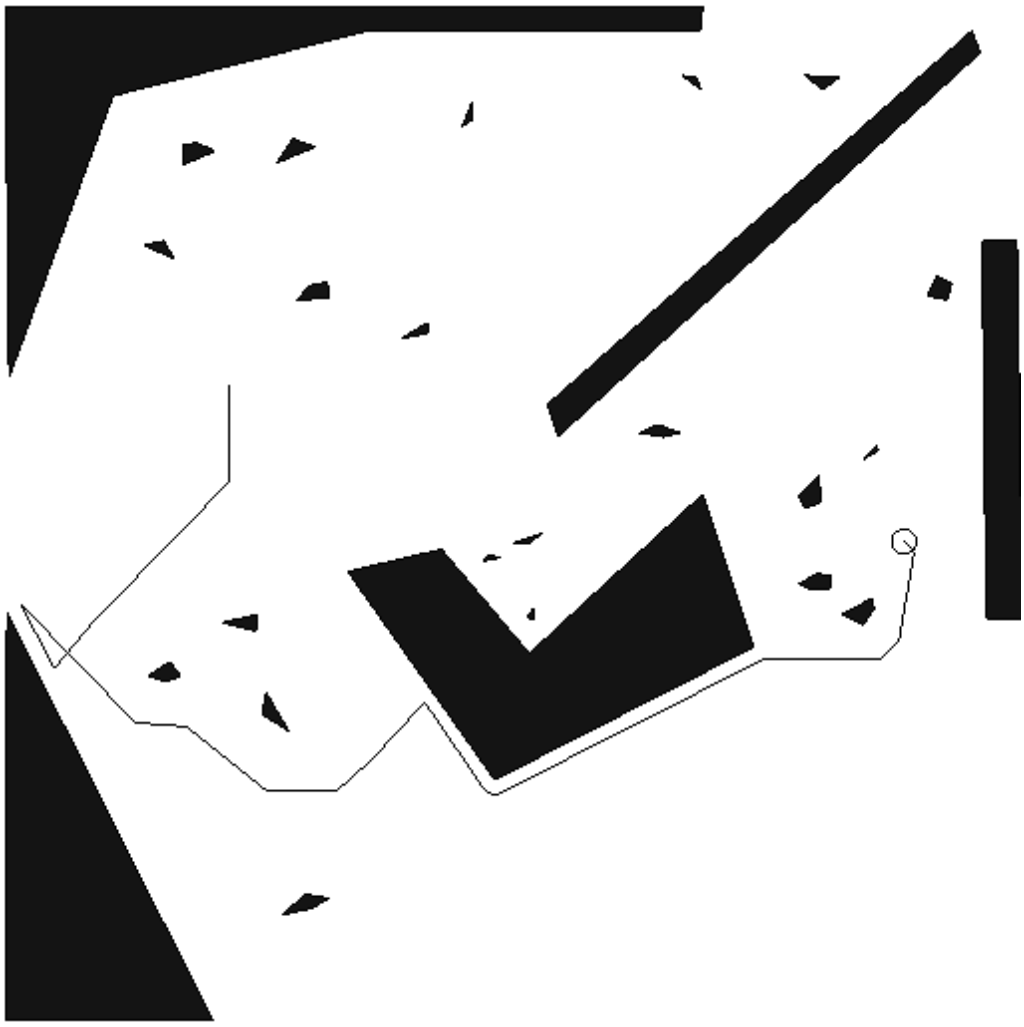
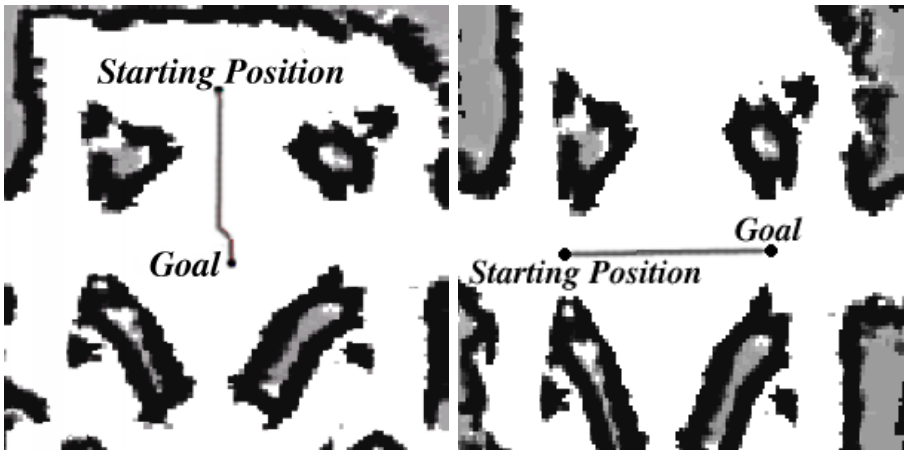
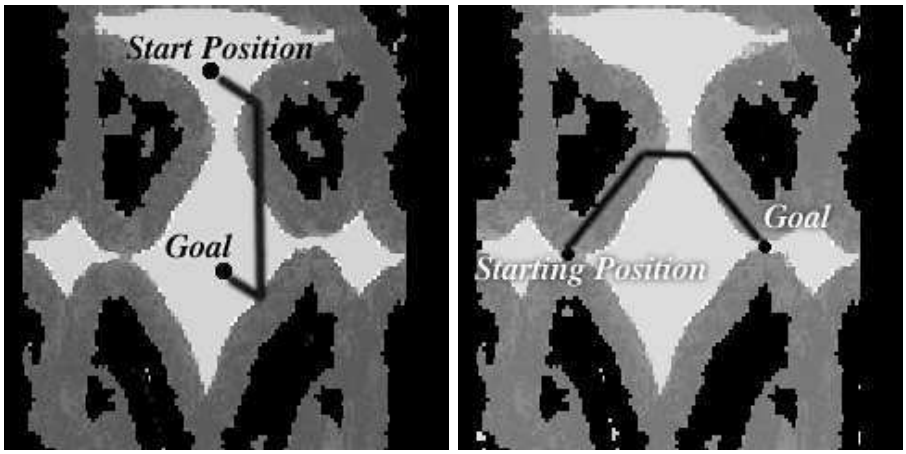


FIG. 2.7 – Trajectoire générée à l'aide du planificateur avec capteurs de contact [29]



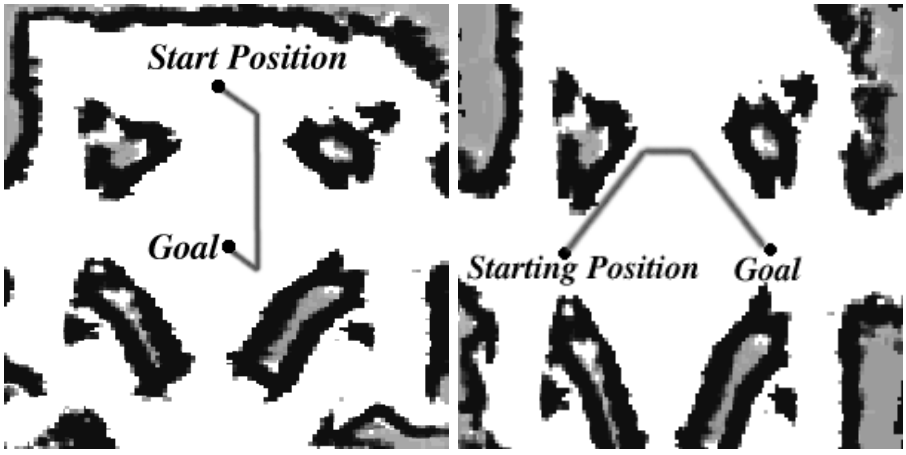
(a) (b)

FIG. 2.8 – Trajectoires générées à l'aide d'un planificateur classique cherchant la trajectoire la plus courte [32]



(a) (b)

FIG. 2.9 – Zones en gris dans lesquelles une relocalisation est possible [32]



(a) (b)

FIG. 2.10 – Trajectoires générées à l'aide d'un planificateur cherchant à maximiser la localisabilité tout en gardant une trajectoire courte [32]

La *localisabilité* n'est pas forcement caractérisée par une valeur numérique. Elle peut aussi être un booléen indiquant si une relocalisation est possible pour un état donné.

Le but de l'algorithme est de minimiser la longueur de la trajectoire tout en passant par les états permettant la meilleure localisation. Il n'y a donc ici aucune propagation de l'erreur. La trajectoire générée est la meilleure en terme de *localisabilité* mais ne garantie pas que le système reste dans $\mathbb{X}_{\text{free}}$. La trajectoire générée est donc celle permettant uniquement d'utiliser au mieux les capteurs extéroceptifs.

Sur la figure 2.8, les trajectoires sont planifiées afin de minimiser leurs longueurs et sans se soucier d'une quelconque capacité à se localiser. La figure 2.9 représente la carte de l'environnement où la relocalisation est possible. Elle a été construite expérimentalement à l'aide d'un robot parcourant l'environnement. Les zones où la relocalisation est possible sont indiquées en gris foncé. Finalement, la figure 2.10 prend en compte ces zones et renvoie une trajectoire qui tente de passer par ces zones tout en minimisant sa longueur totale.

2.4 Recherche dans un espace d'état étendu

Ces méthodes utilisent des algorithmes de planification traditionnels tels que A* après discrétisation de l'espace d'état ou bien des algorithmes à échantillonnage comme le RRT.

Le but est de faire fonctionner ces algorithmes dans un espace d'état étendu. Cet espace d'état est souvent composé de l'état et de la matrice de covariance correspondante qui caractérise l'incertitude sur cet état.

Dans [35,36], un filtre de Kalman est utilisé afin d'estimer l'état du système. Les deux étapes de prédiction et d'estimation utilisent un simulateur de capteur. La planification est effectuée en utilisant un algorithme de type A* sur un graphe issu de la discrétisation de l'espace d'état. La figure 2.11 montre le type de trajectoire planifiée avec cette méthode. En chaque point de discrétisation de l'environnement par lequel passe la trajectoire, une ellipse (plus généralement un ellipsoïde) déduite de la matrice de covariance calculée par le filtre de Kalman représente l'incertitude sur la position. En chacun de ces points, un test de collision est utilisé afin de vérifier que la position est sûre.

L'idée d'associer à chaque état une mesure de son incertitude est reprise dans [37] où un filtre particulaire est utilisé en complément d'un planificateur de type RRT dénommé Particle RRT. L'état du système est ainsi représenté par un ensemble discret d'états. La précision de l'estimation est fonction du nombre de particules utilisées dans le filtre. La figure 2.12 illustre une trajectoire renvoyée par le planificateur. On voit autour de chaque nœud de l'arbre un nuage de point représentant les différents états que peut prendre le système lorsqu'il est situé en ce nœud. Le nœud est sur cette figure positionné à la moyenne des états composant le nuage associé.

Dans [38,39], le processus de planification est mis en œuvre sur un robot réel. En considérant que l'erreur sur l'orientation est bornée, les auteurs montrent par la pratique que l'utilisation d'un cercle d'incertitude autour du véhicule qui grossit de façon linéaire avec la distance parcourue permet au véhicule de se déplacer de façon sûre. Ils utilisent un algorithme A* afin de planifier la trajectoire et font le choix d'utiliser des zones de relocalisation dans leur environnement. L'exemple de la figure 2.13 montre une trajectoire générée en utilisant cette méthode. Les différents obstacles sont représentés par des lignes de niveaux. Aucune zone de relocalisation n'est utilisée sur cette figure.

2.5 Conclusion

Nous avons dans ce chapitre présenté divers planificateurs permettant de planifier des trajectoires en considérant les incertitudes liées au déplacement du système.

Les deux nouveaux planificateurs que nous présentons dans les chapitres suivants appartiennent à cette catégorie de planificateurs sous incertitude et plus particulièrement à la catégorie des planificateurs recherchant une trajectoire dans un espace d'état étendu à cause de l'incertitude. Le premier, basé sur l'algorithme RRT, recherche des trajectoires dans l'espace d'état étendu (état, matrice de covariance de cet état). La représentation de l'incertitude utilisée par le second n'est plus probabiliste, l'état étendu étant représenté plus généralement par un ensemble comprenant tous les états possibles du système.

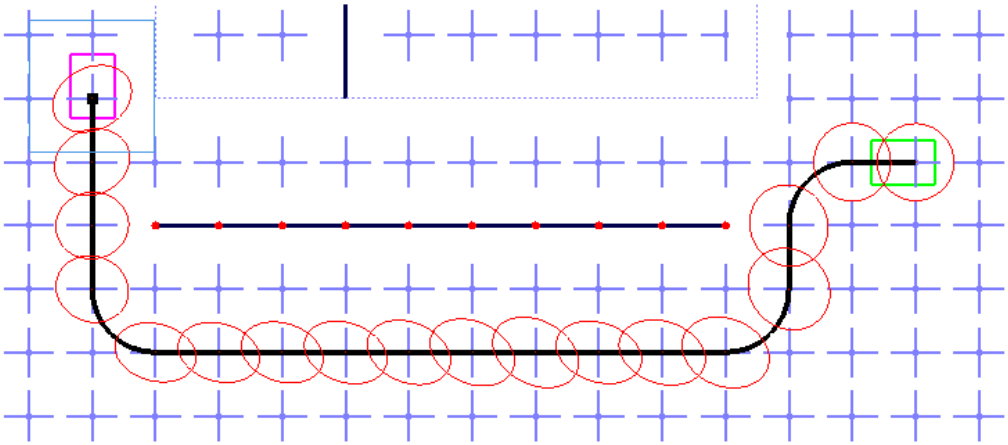


FIG. 2.11 – Exemple de trajectoire planifiée en utilisant les méthodes de [35]

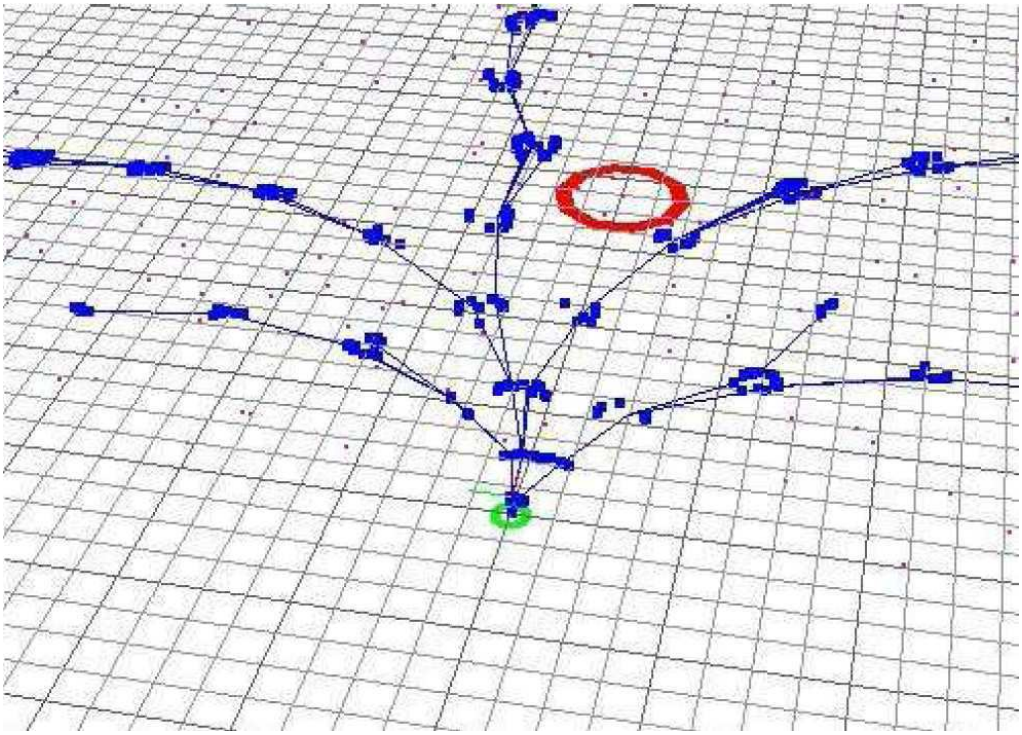


FIG. 2.12 – Exemple issu de [37] illustrant la trajectoire générée par le Particle RRT

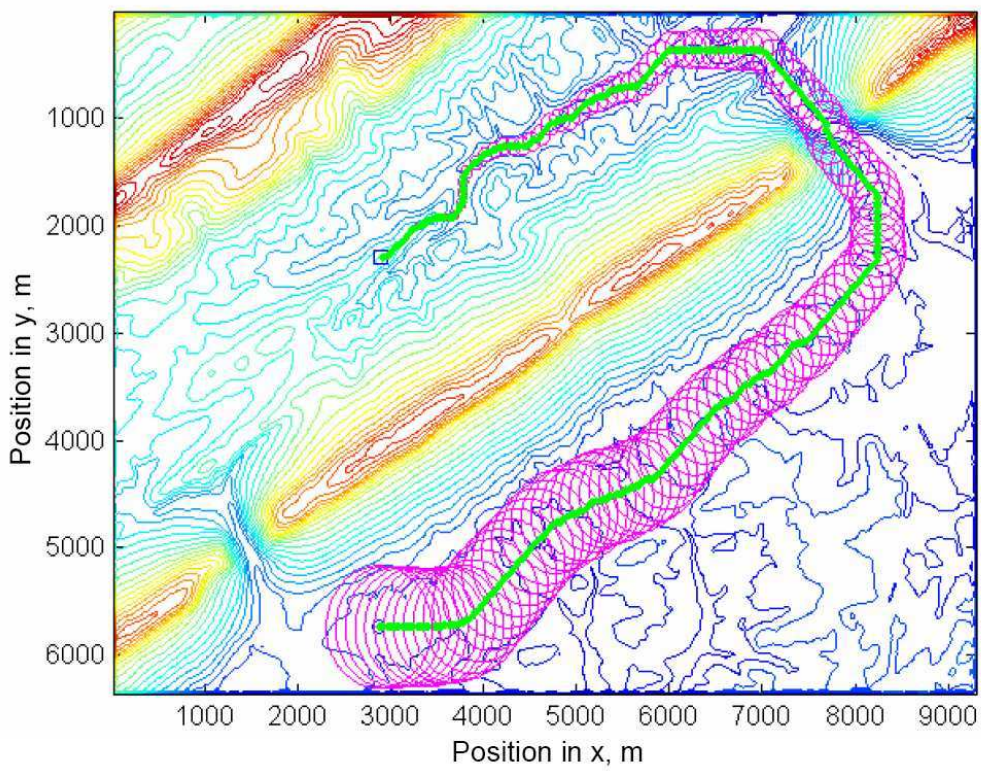


FIG. 2.13 – Exemple issu de [38] illustrant la trajectoire générée par le planificateur

Chapitre 3

Planificateur avec estimation par filtrage de Kalman

Sommaire

Nous proposons dans ce chapitre une nouvelle méthode de planification dans la famille des planificateurs travaillant dans un espace d'état étendu (paragraphe 2.4).

Nous utilisons ici un modèle probabiliste pour représenter les différents états possibles du système. L'état du système est représenté par un vecteur aléatoire gaussien, dont la moyenne et la matrice de covariance sont évaluées au cours du temps et précisent la probabilité que le système soit dans cet état.

Nous ferons l'hypothèse que les changements d'état du système sont décrits par la dynamique linéaire

$$\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{v}_k. \tag{3.1}$$

Par ailleurs, des mesures bruitées de l'état sont supposées disponibles

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{w}_k. \tag{3.2}$$

On note

- $\mathbf{x}_k$ le vecteur d'état du système à l'instant k, de dimension n,
- $\mathbf{A}_k$ la matrice d'évolution du système, de dimension $n \times n$,
- $\mathbf{u}_k$ le vecteur commande, de dimension o,
- $\mathbf{B}_k$ la matrice de commande, permettant le passage de l'espace des commandes à l'espace d'état, de dimension $n \times o$,
- $\mathbf{v}_k$ le vecteur de bruit d'état de dimension n, gaussien, indépendant de $\mathbf{x}_k$, de moyenne nulle et de matrice de covariance connue notée $\mathbf{V}_k$,
- $\mathbf{z}_k$ le vecteur d'observation à l'instant k, de dimension m,
- $\mathbf{H}_k$ la matrice d'observation, permettant le passage de l'espace d'état vers l'espace des observations, de dimension $m \times n$,
- $\mathbf{w}_k$ le vecteur de bruit de mesure de dimension m supposé gaussien, indépendant des vecteurs d'observation passés $\mathbf{z}_{0:k}$, de moyenne nulle et de matrice de covariance connue notée $\mathbf{W}_k$.

Les bruits d'état $\mathbf{v}_k$ et de mesure $\mathbf{w}_k$ sont supposés indépendants de sorte que

$$E(\mathbf{v}_k {\mathbf{w}_k}^{\mathrm{T}}) = \mathbf{0}. \tag{3.3}$$

Le vecteur aléatoire gaussien représentant l'état à l'instant initial est centré en $\hat{\mathbf{x}}_{\text{init}}$ et a pour matrice de covariance $\mathbf{P}_{\text{init}}$. On cherche à atteindre un ensemble $\mathbb{X}_{\text{goal}}$. On considérera que le problème est résolu lorsque l'état final appartient à $\mathbb{X}_{\text{goal}}$ avec une certaine probabilité, cette probabilité étant à définir en fonction du problème à résoudre.

Pour résoudre ce problème, nous présenterons au paragraphe 3.4 un planificateur fondé sur l'algorithme RRT qui permettra d'explorer l'espace d'état en prenant en compte l'incertitude du système. Il utilisera pour cela un filtre de Kalman que nous commencerons donc par présenter.

3.1 Rappels sur le filtre de Kalman

Le filtre de Kalman [40] est utilisé dans de nombreux domaines tels que les télécommunications, la détection radar ou la vision. Nous nous bornerons à présenter ici sa version à temps discret.

Le but du filtre est d'estimer au mieux l'état $\mathbf{x}$ d'un système à un instant $k+1$ donné à partir de la connaissance de l'équation d'état et de l'ensemble des entrées et sorties observées $\mathbf{z}_{0:k+1}$ jusqu'à l'instant $k+1$ tels qu'ils sont définis par (3.1) et (3.2).

Le filtre calcule la loi conditionnelle du vecteur $\mathbf{x}_{k+1}$ sachant $\mathbf{z}_{0:k+1}$. Puisque, dans le cas d'une équation d'état linéaire et d'une distribution initiale gaussienne, cette loi est une gaussienne, il suffit

de trouver la moyenne et la matrice de covariance de cette loi

$$\hat{\mathbf{x}}_{k+1|k+1} = E\left(\mathbf{x}_{k+1}|\mathbf{z}_{0:k+1}\right), \tag{3.4}$$
$$\mathbf{P}_{k+1|k+1} = E\left((\mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1|k+1})(\mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1|k+1})^{\mathrm{T}}|\mathbf{z}_{0:k+1}\right). \tag{3.5}$$

En utilisant les même notations, on a

$$\hat{\mathbf{x}}_{k+1|k} = E\left(\mathbf{x}_{k+1}|\mathbf{z}_{0:k}\right), \tag{3.6}$$
$$\mathbf{P}_{k+1|k} = E\left((\mathbf{x}_{k} - \hat{\mathbf{x}}_{k+1|k})(\mathbf{x}_{k} - \hat{\mathbf{x}}_{k+1|k})^{\mathrm{T}}|\mathbf{z}_{0:k}\right), \tag{3.7}$$

qui sont les estimées de la moyenne de la variance de $\mathbf{x}$ à l'instant $k+1$ lorsque seules les observations jusqu'à l'instant k sont disponibles.

Le filtre de Kalman fonctionne de façon récursive alternant prédiction et correction. En supposant connus $\hat{\mathbf{x}}_{k|k}$ et $\mathbf{P}_{k|k}$, il effectue d'abord une étape de prédiction afin de déterminer $\hat{\mathbf{x}}_{k+1|k}$ et $\mathbf{P}_{k+1|k}$. Il effectue ensuite une étape de correction en tenant compte de la nouvelle mesure $\mathbf{z}_{k+1}$ qui permet de calculer $\hat{\mathbf{x}}_{k+1|k+1}$ et $\mathbf{P}_{k+1|k+1}$.

3.1.1 Prédiction

L'étape de prédiction permet donc de calculer la loi de $\mathbf{x}_{k+1}$ sachant $\mathbf{z}_{0:k}$. Les valeurs de $\hat{\mathbf{x}}_{k+1|k}$ et $\mathbf{P}_{k+1|k}$ sont appelées valeurs *a priori*. Il vient

$$\hat{\mathbf{x}}_{k+1|k} = E\left(\mathbf{x}_{k+1|k}|\mathbf{z}_{0:k}\right), \tag{3.8}$$
$$= E\left(\mathbf{A}_k\mathbf{x}_k + \mathbf{B}_k\mathbf{u}_k + \mathbf{v}_k|\mathbf{z}_{0:k}\right), \tag{3.9}$$
$$= \mathbf{A}_k\hat{\mathbf{x}}_{k|k} + \mathbf{B}_k\mathbf{u}_k, \tag{3.10}$$

puisque la moyenne de $\mathbf{v}_t$ est nulle.

Ainsi, la prédiction correspond à l'évolution naturelle du système en l'absence de bruit. L'erreur sur cette prédiction est

$$\tilde{\mathbf{x}}_{k+1|k} = \mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1|k} = \mathbf{A}_k\left(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k}\right) + \mathbf{v}_k = \mathbf{A}_k\tilde{\mathbf{x}}_{k|k} + \mathbf{v}_k. \tag{3.11}$$

La matrice de covariance de $\mathbf{x}_{k+1|k}$ qui correspond donc à la covariance de l'erreur sur la prédiction $\tilde{\mathbf{x}}_{k+1|k}$ est donnée par

$$\mathbf{P}_{k+1|k} = E\left(\tilde{\mathbf{x}}_{k+1|k}\tilde{\mathbf{x}}_{k+1|k}^{\mathrm{T}}\right), \tag{3.12}$$
$$= E\left(\left(\mathbf{A}_k\tilde{\mathbf{x}}_{k|k} + \mathbf{v}_k\right)\left(\tilde{\mathbf{x}}_{k|k}^{\mathrm{T}}\mathbf{A}_k^{\mathrm{T}} + \mathbf{v}_k^{\mathrm{T}}\right)\right), \tag{3.13}$$
$$= \mathbf{A}_k E\left(\tilde{\mathbf{x}}_{k|k}\tilde{\mathbf{x}}_{k|k}^{\mathrm{T}}\right)\mathbf{A}_k^{\mathrm{T}} + E(\mathbf{v}_k\mathbf{v}_k^{\mathrm{T}}), \tag{3.14}$$
$$= \mathbf{A}_k\mathbf{P}_{k|k}\mathbf{A}_k^{\mathrm{T}} + \mathbf{V}_k, \tag{3.15}$$

le passage de (3.13) à (3.14) est possible puisque $\mathbf{x}_k$ et $\mathbf{v}_k$ sont indépendants.

3.1.2 Correction

L'étape de correction utilise l'observation $\mathbf{z}_{k+1}$ afin de corriger l'estimée précédente. On passe ainsi de $\hat{\mathbf{x}}_{k+1|k}$ et $\mathbf{P}_{k+1|k}$ à $\hat{\mathbf{x}}_{k+1|k+1}$ et $\mathbf{P}_{k+1|k+1}$. Ce sont les valeurs dites *a posteriori*.

La prédiction du vecteur des sorties à l'instant $k+1$ à partir des informations disponibles jusqu'à

l'instant k est donnée par

$$\hat{\mathbf{z}}_{k+1} = E\left(\mathbf{z}_{k+1}|\mathbf{z}_{0:k}\right), \tag{3.16}$$
$$= E\left(\mathbf{H}_{k+1}\mathbf{x}_{k+1} + \mathbf{w}_{k+1}|\mathbf{z}_{0:k}\right), \tag{3.17}$$
$$= \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1|k}. \tag{3.18}$$

En effet $E\left(\mathbf{w}_{k+1}|\mathbf{z}_{0:k}\right) = \mathbf{0}$ puisque le bruit présent est indépendant des sorties passées.

L'écart entre la vraie valeur du vecteur des sorties à l'instant $k+1$ et sa prédiction

$$\tilde{\mathbf{z}}_{k+1} = \mathbf{z}_{k+1} - \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1|k} \tag{3.19}$$

est appelé innovation. Son espérance est nulle et sa matrice de covariance est

$$\mathbf{S}_{k+1} = \mathbf{W}_{t+1} + \mathbf{H}_{t+1}\mathbf{P}_{k+1|k}\mathbf{H}_{t+1}^{\mathrm{T}}. \tag{3.20}$$

Cette matrice de covariance intervient dans le calcul du gain du filtre $\mathbf{K}_{k+1}$, appelé gain de Kalman

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^{\mathrm{T}}\mathbf{S}_{k+1}^{-1} \tag{3.21}$$

et utilisé pour calculer la nouvelle estimée $\hat{\mathbf{x}}_{k+1|k+1}$:

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1}\tilde{\mathbf{z}}_{k+1}. \tag{3.22}$$

La matrice de covariance modifiée par la nouvelle information devient

$$\mathbf{P}_{k+1|k+1} = \left(\mathbf{I} - \mathbf{K}_{k+1}\mathbf{H}_{k+1}\right)\mathbf{P}_{k+1|k}. \tag{3.23}$$

On peut montrer que [41]

$$\mathbf{P}_{k+1|k+1} \leqslant \mathbf{P}_{k+1|k}, \tag{3.24}$$

ce qui implique que le processus de correction ne peut que diminuer l'incertitude sur l'état.

3.1.3 Résumé

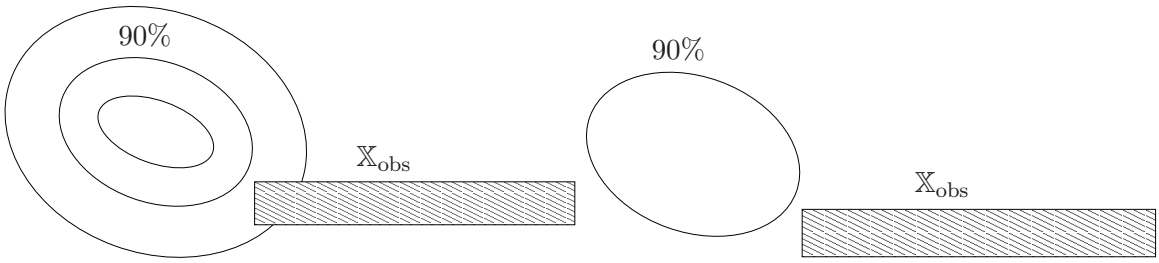
Les deux étapes de prédiction et de correction peuvent être résumées par les équations suivantes :

Prédiction : (3.25)
$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{A}_k\mathbf{x}_k + \mathbf{B}_k\mathbf{u}_k, \tag{3.26}$$
$$\mathbf{P}_{k+1|k} = \mathbf{A}_k\mathbf{P}_{k|k}\mathbf{A}_k^{\mathrm{T}} + \mathbf{V}_k, \tag{3.27}$$
Correction : (3.28)
$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1}\tilde{\mathbf{z}}_{k+1}, \tag{3.29}$$
$$\mathbf{P}_{k+1|k+1} = \left(\mathbf{I} - \mathbf{K}_{k+1}\mathbf{H}_{k+1}\right)\mathbf{P}_{k+1|k}, \tag{3.30}$$

avec

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^{\mathrm{T}}\mathbf{S}_{k+1}^{-1}, \tag{3.31}$$
$$\mathbf{S}_{k+1} = \mathbf{W}_{t+1} + \mathbf{H}_{t+1}\mathbf{P}_{k+1|k}\mathbf{H}_{t+1}^{\mathrm{T}}, \tag{3.32}$$
$$\tilde{\mathbf{z}}_{k+1} = \mathbf{z}_{k+1} - \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1|k}, \tag{3.33}$$

qui définissent le filtre de Kalman.



(a) La probabilité que l'état appartienne à $\mathbb{X}_{\text{obs}}$ est très délicate à évaluer

(b) $\mathbf{x} \notin \mathbb{X}_{\text{obs}}$ avec un niveau de confiance d'au moins 90%

FIG. 3.1 – Représentation de l'incertitude sur l'état de dimension 2 du système

3.1.4 Filtre de Kalman étendu

Le filtre de Kalman étendu (EKF) permet d'utiliser le filtre de Kalman dans le cas d'une dynamique et d'une équation d'observation non-linéaires.

Considérons le système non-linéaire à temps discret suivant :

$$\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{v}_k, \tag{3.34}$$

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k) + \mathbf{w}_k. \tag{3.35}$$

Le filtre de Kalman étendu linéarise les fonctions $\mathbf{f}$ et $\mathbf{h}$ en utilisant leurs jacobiennes évaluées avec les états estimés courants. Ainsi, $\mathbf{f}_k$ est linéarisé autour de $\hat{\mathbf{x}}_{k|k}$ et $\mathbf{h}_{k+1}$ autour de $\hat{\mathbf{x}}_{k+1|k}$.

Le filtre de Kalman étendu est défini par les équations suivantes :

$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{f}_k(\hat{\mathbf{x}}_{k|k}, \mathbf{u}_k), \tag{3.36}$$

$$\mathbf{P}_{k+1|k} = \mathbf{A}_k \mathbf{P}_{k|k} \mathbf{A}_k^{\mathrm{T}} + \mathbf{V}_k, \tag{3.37}$$

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1} \tilde{\mathbf{z}}_{k+1}, \tag{3.38}$$

$$\mathbf{P}_{k+1|k+1} = (\mathbf{I} - \mathbf{K}_{k+1} \mathbf{H}_{k+1})\, \mathbf{P}_{k+1|k}, \tag{3.39}$$

avec

$$\mathbf{A}_k = \left.\frac{\partial \mathbf{f}_k}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k|k}, \mathbf{u}_k}, \tag{3.40}$$

$$\mathbf{H}_{k+1} = \left.\frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k+1|k}}, \tag{3.41}$$

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1|k} \mathbf{H}_{k+1}^{\mathrm{T}} \mathbf{S}_{k+1}^{-1}, \tag{3.42}$$

$$\mathbf{S}_{k+1} = \mathbf{W}_{t+1} + \mathbf{H}_{t+1} \mathbf{P}_{k+1|k} \mathbf{H}_{t+1}^{\mathrm{T}}, \tag{3.43}$$

$$\tilde{\mathbf{z}}_{k+1} = \mathbf{z}_{k+1} - \mathbf{h}_{k+1}(\hat{\mathbf{x}}_{k+1|k}). \tag{3.44}$$

3.2 Représentation de l'incertitude

L'incertitude sur l'état est caractérisée dans le filtre de Kalman par la matrice de covariance $\mathbf{P}$. La densité de probabilité d'un vecteur gaussien $\mathbf{x}$ de dimension n, de moyenne $\hat{\mathbf{x}}$ et de matrice de covariance $\mathbf{P}$ est

$$p(\mathbf{x}) = \frac{1}{(\sqrt{2\pi})^n \sqrt{\det \mathbf{P}}} e^{-\frac{1}{2}\left((\mathbf{x}-\hat{\mathbf{x}})^{\mathrm{T}} \mathbf{P}^{-1} (\mathbf{x}-\hat{\mathbf{x}})\right)}. \tag{3.45}$$

La gaussienne est ainsi centrée en l'estimée $\hat{\mathbf{x}}$.

Cette représentation probabiliste n'est pas toujours pratique. Elle permet de calculer la probabilité

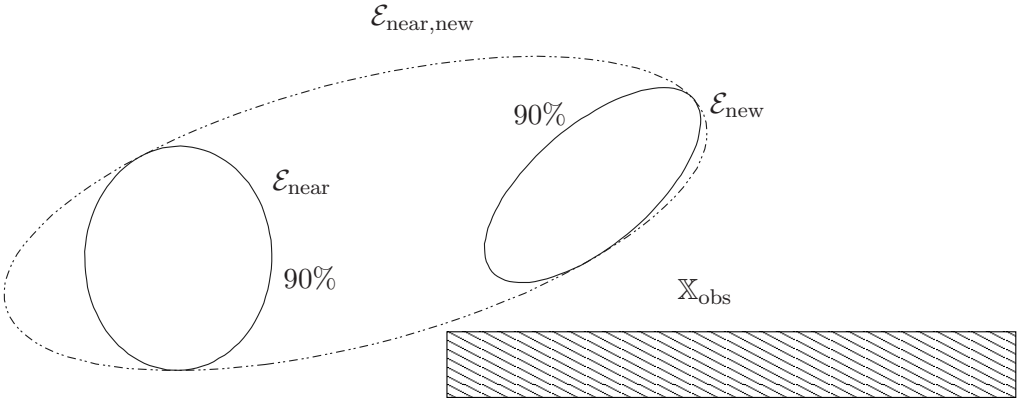


FIG. 3.2 – Trajectoire sûre pour un niveau de confiance donné

que l'état appartienne à $\mathbb{X}_{\text{obs}}$ qui s'avère très délicate à évaluer (figure 3.1(a)). C'est pourquoi nous lui préférons l'utilisation de l'ellipsoïde contenant le système pour un niveau de confiance donné [42] (figure 3.1(b)). Voir l'annexe A pour le détail des calculs. On peut parler alors d'ellipsoïde d'état puisque cet ellipsoïde contient, avec une probabilité donnée, l'ensemble des états du système à un instant donné. L'ensemble des approximations ayant conduit au calcul de $\hat{\mathbf{x}}$ et $\mathbf{P}$ oblige bien sûr à prendre cette affirmation avec une certaine méfiance.

3.3 Test de collision

Le test de collision doit s'assurer de deux choses. Tout d'abord, avant d'exécuter l'algorithme, il faut s'assurer que l'état initial $(\hat{\mathbf{x}}, \mathbf{P})_{\text{init}}$ du véhicule appartient à $\mathbb{X}_{\text{free}}$ pour un niveau de confiance donné. Il faut donc vérifier que l'ellipsoïde d'état obtenu appartient entièrement à $\mathbb{X}_{\text{free}}$.

Dans une seconde étape, il faut vérifier que la trajectoire entre les deux état incertains successifs $(\hat{\mathbf{x}}, \mathbf{P})_{\text{near}}$ à l'instant t et $(\hat{\mathbf{x}}, \mathbf{P})_{\text{new}}$ à l'instant $t + \Delta t$ est garantie pour le même niveau de confiance a chaque insertion d'un nouveau nœud dans l'arbre. Pour faire cela, nous allons utiliser un ellipsoïde, notée $\mathcal{E}_{\text{near,new}}$, qui englobe les deux ellipsoïdes d'état $\mathcal{E}_{\text{near}}$ et $\mathcal{E}_{\text{new}}$. Nous prenons comme hypothèse que tous les ellipsoïdes obtenues à l'instant $\tau, (t \leqslant \tau \leqslant t + \Delta t)$ sont contenus à l'intérieur de cet ellipsoïde englobant.

Afin de ne pas ajouter trop de pessimisme, nous cherchons le plus petite ellipsoïde au sens du volume qui contient $\mathcal{E}_{\text{near}}$ et $\mathcal{E}_{\text{new}}$. C'est un problème de maximisation d'un déterminant sous contraintes d'inégalités de matrices linéaires (LMI). L'ellipsoïde résultant est appelée ellipsoïde de *Lowner-John* [43–45]. Pour résoudre un tel problème, on utilise la solution proposée par [46] et plus précisément le logiciel Maxdet[1] qui permet de maximiser le déterminant d'une matrice sujette à des contraintes LMI. En utilisant Maxdet, on détermine l'ellipsoïde englobant. Sur la figure 3.2, on peut voir un exemple en dimension deux représentant la plus petite ellipse contenant deux autres ellipses.

Une fois $\mathcal{E}_{\text{near,new}}$ obtenu, il faut vérifier que cet ellipsoïde englobant est contenu dans $\mathbb{X}_{\text{free}}$.

3.4 Présentation du planificateur

Nous présentons dans cette section le planificateur Kalman-RRT [47] et notamment l'exploitation qu'il fait des informations fournies par les capteurs.

[1] `http://www.stanford.edu/~boyd/old_software/MAXDET.html`

3.4.1 Intégration des mesures capteurs et de leurs erreurs

Le filtre de Kalman, éventuellement étendu, a besoin d'informations retournées par des capteurs pour en déduire l'ellipsoïde contenant l'état avec une probabilité donnée. Ces capteurs sont de deux types :

- les capteurs proprioceptifs qui prennent des informations sur le système lui-même et permettent une localisation relative à l'estimée de l'état précédente,
- Les capteurs extéroceptifs qui prennent les informations sur l'environnement entourant le système et permettent une localisation absolue.

Les capteurs proprioceptifs sont utilisés pendant l'étape de prédiction de l'algorithme de localisation. Ainsi, grâce à l'état estimé à l'instant k et à l'équation de changement d'état, on va pouvoir estimer l'état à l'instant $k+1$. Les erreurs liées au système, à son contrôle, aux glissements, aux jeux, sont définis dans le vecteur de bruit $\mathbf{v}_k$. Les caractéristiques statistiques de ce bruit doivent être obtenues expérimentalement.

Les capteurs extéroceptifs sont quant à eux utilisés pendant l'étape de correction du filtre. Ils apportent une information sur l'état du système par rapport à son environnement. L'incertitude liée à l'environnement doit donc être spécifiée dans le vecteur de bruit de mesure $\mathbf{w}_k$, dont les caractéristiques statistiques doivent également être obtenues expérimentalement.

La simulation des capteurs extéroceptifs peut poser problème : afin de pouvoir simuler les mesures capteurs, il faut savoir où se situe le système. Deux solutions sont alors possibles : il est possible de simuler les mesures capteurs en considérant que le système se trouve dans l'état qui possède la plus forte densité de probabilité, c'est-à-dire la moyenne du vecteur Gaussien. Cette solution simplifie le problème d'estimation mais ajoute de l'optimisme. Si l'on devait simuler les mesures capteurs pour tous les états probables du système, le processus d'estimation ne permettrait pas de réduire l'incertitude. L'autre possibilité est donc de n'utiliser qu'une phase de prédiction dans le processus de planification.

Le planificateur doit également tenir compte de la fréquence d'échantillonnage de ces capteurs. Dans le contexte de la robotique, pour des capteurs proprioceptifs, les données sont généralement disponibles à tout instant. Chaque fois que l'on a besoin d'une mesure, il suffit de lire la valeur des différents compteurs afin de l'obtenir. Au contraire, les capteurs extéroceptifs peuvent avoir une fréquence d'échantillonnage assez faible. Si l'on prend l'exemple d'un GPS, il n'est pas rare que les données soient disponibles avec une période de deux à trois secondes, ce qui est souvent bien supérieur à l'intervalle de temps Δt nécessaire pour parcourir la trajectoire reliant deux nœuds de l'arbre.

Ainsi, au contraire de l'étape de prédiction qui est réalisée à chaque génération de nœud dans l'arbre, l'étape de correction ne peut être utilisée que de temps en temps. Le planificateur doit donc prendre en compte la fréquence de rafraîchissement des données provenant des capteurs extéroceptifs et ne lancer la phase de correction que lorsque de nouvelles données sont disponibles.

Pour simuler ceci, il faut connaître la période de rafraîchissement T_c des capteurs extéroceptifs ainsi que le temps que va mettre le mobile à parcourir la trajectoire générée pour arriver au nœud courant. Cette valeur peut être obtenue si chaque nœud connaît sa profondeur p au sein de l'arbre. Le temps mis par le véhicule pour parcourir l'arbre depuis $\mathbf{x}_{\text{init}}$ jusqu'à un nœud à la profondeur p est alors

$$t_p = p \cdot \Delta t. \tag{3.46}$$

En utilisant le même principe, il est également possible d'obtenir la longueur des trajectoires générées. En notant la vitesse sur le i-ème segment de la trajectoire v_i, on obtient comme distance entre le

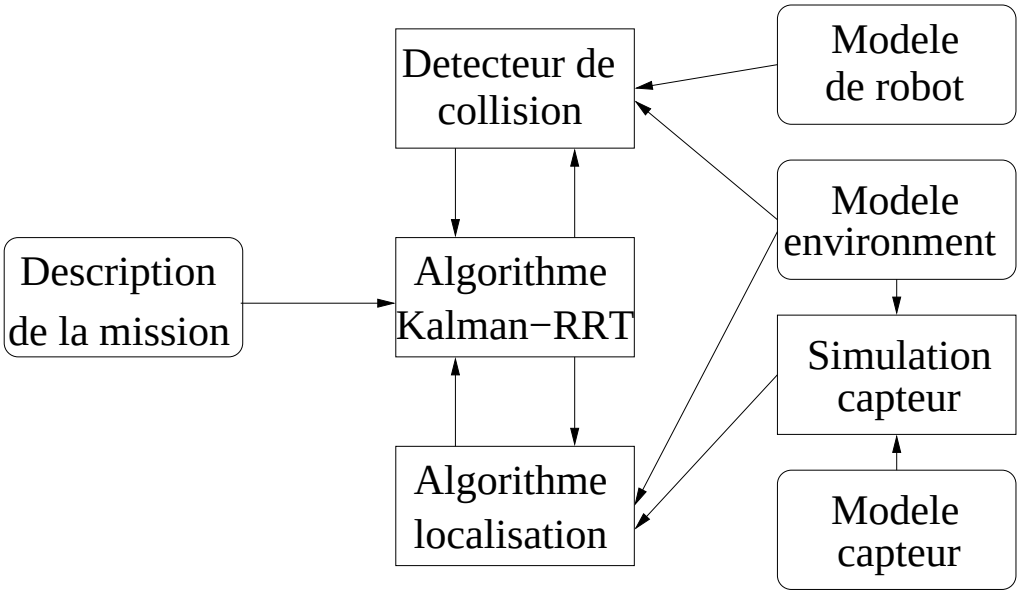


FIG. 3.3 – Kalman-RRT

nœud de départ et un nœud à la profondeur p

$$\sum_{i=0}^{p} v_i \cdot \Delta t. \tag{3.47}$$

Ainsi, lorsque t_p atteint une valeur multiple de T_c, on procède à une étape de correction du filtre de Kalman. Si cette valeur est atteinte au cours d'un intervalle de temps Δt, cet intervalle sera découpé en trois parties :

- une prédiction est utilisée sur l'intervalle de temps avant que t_p ne soit un multiple de T_c,
- la correction est exécutée lorsque les mesures seront disponibles,
- une prédiction est de nouveau utilisée pour finir l'intervalle de temps Δt.

3.4.2 Schéma fonctionnel

Notre planificateur permettant la recherche de solutions dans un espace d'état incertain associe trois composantes :

- le planificateur à échantillonnage RRT,
- un algorithme de localisation (le filtre de Kalman étendu),
- un test de collision qui fait le lien entre l'algorithme de localisation et le planificateur.

Le test de collision dépend fortement du problème à résoudre mais est indispensable afin de déterminer si un état incertain appartient à $\mathbb{X}_{\text{free}}$. C'est ce test qui décide si malgré l'incertitude, cet état peut être considéré comme admissible.

Nous montrons un exemple de test de collision dans la partie plus particulièrement consacrée à la robotique de ce mémoire (paragraphe 6.3 page 123).

Le schéma fonctionnel du planificateur que nous avons dénommé Kalman-RRT [47] est présenté à la figure 3.3.

Ce schéma montre bien l'importance de chacun des composants précédents. Le composant central dans un planificateur à échantillonnage n'est plus l'algorithme de planification mais le test de collision nécessaire à chacune des tentatives d'ajout de nœud dans l'arbre.

Trois composants interdépendants peuvent donc être considérés :

Algorithme 9 KalmanRRT(**in** : $K \in \mathbb{N}$, $\mathbf{x}_{\text{init}} \in \mathbb{X}_{\text{free}}$, $\Delta t \in \mathbb{R}$, **out** : G)

1: G.Init$(\hat{\mathbf{x}}, \mathbf{P})_{\text{init}})$
2: **Pour** $i = 0$ **à** K **faire**
3: $\quad \mathbf{x}_{\text{rand}} \leftarrow$ RandomConfig$((\mathbb{X}_{\text{free}})$
4: $\quad$ KalmanRRTExtend$(G, \mathbf{x}_{\text{rand}})$
5: **FinPour**
6: **Retourner** G

Algorithme 10 KalmanRRTExtend(**in** : $\mathbf{x}_{\text{rand}}$, **inout** : G)

1: $(\hat{\mathbf{x}}, \mathbf{P})_{\text{near}} \leftarrow$ NearestNeighbor$(G, \mathbf{x}_{\text{rand}})$
2: $\mathbf{u} \leftarrow$ SelectInput$(\mathbf{x}_{\text{rand}}, \hat{\mathbf{x}}_{\text{near}})$
3: $(\hat{\mathbf{x}}, \mathbf{P})_{\text{new}} \leftarrow$ Prediction$((\hat{\mathbf{x}}, \mathbf{P})_{\text{near}}, \mathbf{u}, \Delta t)$
4: **Si** CollisionFreePath$((\hat{\mathbf{x}}, \mathbf{P})_{\text{near}}, (\hat{\mathbf{x}}, \mathbf{P})_{\text{new}})$ **alors**
5: $\quad$ **Si** mesures sont disponibles **alors**
6: $\quad\quad (\hat{\mathbf{x}}, \mathbf{P})_{\text{new}} \leftarrow$ Estimation$((\hat{\mathbf{x}}, \mathbf{P})_{\text{new}})$
7: $\quad$ **FinSi**
8: $\quad G$.AddNode$((\hat{\mathbf{x}}, \mathbf{P})_{\text{new}})$
9: $\quad G$.AddEdge$(\hat{\mathbf{x}}_{\mathbf{new}}, \hat{\mathbf{x}}_{\text{new}}, \mathbf{u})$
10: **FinSi**

- L'algorithme de localisation utilise le planificateur puisqu'il lui faut l'ancienne incertitude ainsi que la commande à appliquer. En retour, il renvoie une incertitude qui sera analysée par le test de collision.
- Le planificateur utilise le résultat du test de collision ainsi que la valeur de l'incertitude pour décider s'il convient d'ajouter un état incertain dans l'arbre.
- Le test de collision utilise l'incertitude retournée par l'algorithme de localisation et, en fonction du type d'algorithme utilisé, renvoie une valeur indiquant qu'il y a ou non collision avec une probabilité donnée.

3.4.3 Algorithme

L'algorithme Kalman-RRT (algorithme 9) est très proche de celui du RRT traditionnel. La différence principale réside dans l'utilisation permanente du couple composé de l'état estimé $\hat{\mathbf{x}}$ et de la matrice de covariance associée $\mathbf{P}$ afin de définir les nœuds de l'arbre.

Insistons sur le fait que $\mathbf{x}_{\text{rand}}$ n'est pas incertain. Le but de notre planificateur est en effet de se rendre en un état donné malgré les incertitudes, il n'est nullement question de se rendre en un état incertain donné. On ne souhaite pas que l'incertitude sur l'état du système ait une valeur particulière, nous voulons seulement que cette incertitude soit inférieure à une valeur qui pourrait entraîner une collision.

La sous-fonction `KalmanRRTExtend` (algorithme 10) diffère de la fonction `RRTExtend` (algorithme 8) puisqu'un algorithme de localisation est utilisé à la place de la fonction `NewState`. L'algorithme choisi tout d'abord (algorithme 10 ligne 1) le nœud de l'arbre (nœud qui contient l'état étendu $(\hat{\mathbf{x}}, \mathbf{P})_{\text{near}}$) le plus proche de celui du nœud choisi aléatoirement $(\hat{\mathbf{x}}, \mathbf{P})_{\text{rand}}$.

Comme pour le RRT classique, une commande est alors choisie (ligne 2). Cet état étendu ainsi que la commande sont alors envoyées à la fonction `Prediction` qui va retourner le nouvel état étendu $(\hat{\mathbf{x}}, \mathbf{P})_{\text{new}}$ (ligne 3). Cette fonction est responsable de la partie prédiction de l'algorithme de localisation.

La ligne suivante utilise un test de collision (fonction `CollisionFreePath` à la ligne 4) afin de tester si la trajectoire élémentaire entre les deux états étendus successifs $(\hat{\mathbf{x}}, \mathbf{P})_{\text{near}}$ et $(\hat{\mathbf{x}}, \mathbf{P})_{\text{new}}$

appartient à $\mathbb{X}_{\text{free}}$ pour un niveau de confiance donné (paragraphe 3.3).

A la ligne 5, si des données issues du simulateur de capteurs extéroceptifs sont disponibles, alors l'algorithme exécute la phase de correction du filtre de Kalman (fonction `Estimation` à la ligne 6). Tout comme la fonction `Prediction`, la fonction `Estimation` renvoie la valeur mise à jour de $(\hat{\mathbf{x}}, \mathbf{P})_{\text{new}}$.

Une fois ces différentes phases du filtre réalisées, l'arbre est modifié (lignes 8 et 9) afin de faire apparaître le nouveau nœud relié à l'arbre par la nouvelle branche. L'exécution de la fonction `KalmanRRTExtend` est itérée autant de fois que spécifié par le paramètre K de la fonction `KalmanRRT`.

Des exemples de trajectoires planifiées en utilisant Kalman-RRT dans le contexte de la robotique sont disponibles au chapitre 6.

3.4.4 Propriétés

Comme nous venons de le voir, les algorithmes du RRT et Kalman-RRT, sa version avec incertitudes sont très proches.

La complexité de l'algorithme suite à l'ajout des incertitudes ne change pas par rapport au RRT traditionnel. La complexité maximale correspond toujours à la recherche du plus proche voisin suite au tirage d'une configuration aléatoire. L'utilisation d'un algorithme de localisation modifie ainsi la constante multiplicative mais pas l'ordre de la complexité globale de l'algorithme. Les performances en terme de temps de calcul s'avèrent donc très proches de celles du RRT.

3.5 Conclusion

L'algorithme que nous venons de présenter permet la planification de trajectoires en considérant les incertitudes. Cet algorithme repose sur le RRT pour la planification et sur le filtre de Kalman pour l'estimation de l'état dans lequel se trouve le système.

Cet algorithme vise à résoudre des problèmes de planification en réduisant les risques de collision dus à l'incertitude sur l'état du système. La sûreté de la trajectoire n'est cependant pas *garantie*. En effet, outre le fait que les calculs effectués sont approximatifs et reposent sur une linéarisation de l'équation d'état, l'utilisation d'une représentation probabiliste de l'état incertain ne permet pas de garantir à 100% que l'état du système se trouve dans une zone donnée. Il est nécessaire de considérer un certain niveau de confiance, notamment dans le test de collision. De plus comme nous l'expliquions dans le paragraphe 3.4, l'utilisation de la phase de correction, et notamment la simulation des capteurs extéroceptifs, induit de nouvelles imprécisions. Nous verrons au chapitre suivant une approche qui ne présente pas ces inconvénients.

Chapitre 4

Planificateur ensembliste

Sommaire

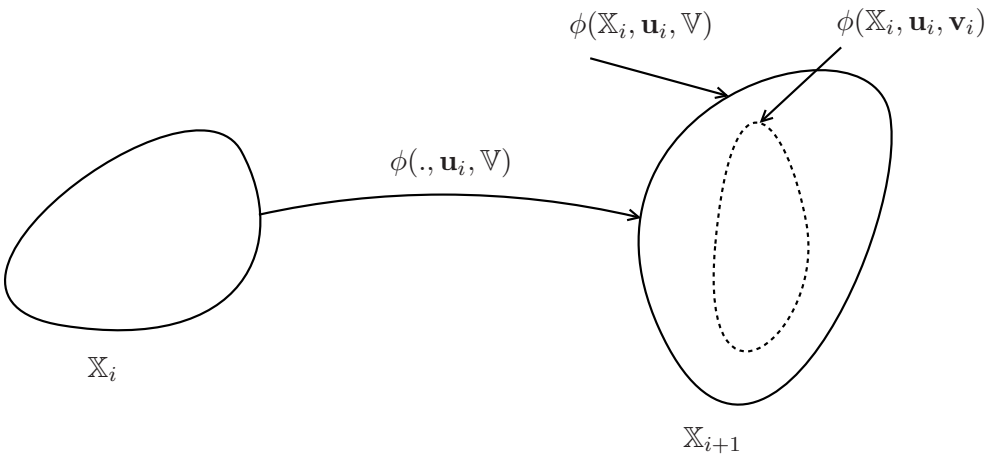


FIG. 4.1 – Prédiction ensembliste

Dans le chapitre précédent, nous avons présenté un planificateur intégrant une estimation par filtrage de Kalman. Comme nous l'avons vu en conclusion, l'utilisation de cette méthode, et notamment sa phase de correction induit un nombre important d'approximations, dues à la simulation des capteurs extéroceptifs, qui ne garantissent pas que les trajectoires générées n'intersectent pas $\mathbb{X}_{\text{obs}}$.

Nous présentons ici une modification du RRT afin de travailler avec des états incertains, les incertitudes étant représentées par des ensembles. Ainsi, un ensemble $\mathbb{X}_i$ contient toutes les valeurs possibles d'un état à un instant i donné, compte tenu de toutes les incertitudes du modèle.

Travailler et surtout représenter des ensembles de formes quelconques s'avère très délicat voire impossible. Nous les enveloppons dans des ensembles aux formes plus simples (par exemple des ellipsoïdes [48], des zonotopes [49], des vecteurs d'intervalles [50] ou encore des unions de vecteurs d'intervalles [51, 52]). L'utilisation de tels ensembles plus simples à manipuler que des ensembles quelconques engendrera un pessimisme qui ne fera pas perdre à nos résultats leur caractère garanti mais qui pourra rendre la recherche de plans sûrs plus délicate.

Nous considérons que les équations régissant l'évolution du système sont incertaines. L'équation d'état non-linéaire s'écrit

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \mathbf{v}(t)) \tag{4.1}$$

où $\mathbf{x}(t) \subset \mathbb{X}$, $\mathbf{u}(t) \in \mathbb{U}$ est la commande appliquée au système et où $\mathbf{v}(t)$ représente les perturbations agissant sur le système à l'instant t. Nous considérons que ces perturbations sont bornées, $\mathbf{v}(t) \in \mathbb{V}$ pour tout t.

Nous supposerons que le système (4.1) admet une solution partant de $\mathbf{x}_i \in \mathbb{X}_i$. On connaît $\mathbb{X}_i$ à l'instant t, on suppose que sur $[t, t + \Delta t]$ $\mathbf{u}_i$ est constant et pour tout $v : [t, t + \Delta t] \to \mathbb{V}$. La solution de (4.1) définit alors la trajectoire sur l'intervalle de temps $[t, t + \Delta t]$ entre $\mathbf{x}_i$ et $\mathbf{x}_{i+1}$

$$\mathbf{x}_{i+1} = \phi(\mathbf{x}_i, \mathbf{u}_i, \mathbf{v}_i). \tag{4.2}$$

Ainsi, le nouvel ensemble $\mathbb{X}_{i+1}$ obtenu à partir de $\mathbb{X}_i$ en utilisant une commande $\mathbf{u}_i$ donnée s'écrit

$$\mathbb{X}_{i+1} = \phi(\mathbb{X}_i, \mathbf{u}_i, \mathbb{V}). \tag{4.3}$$

Cet ensemble contient bien sûr l'ensemble $\phi(\mathbb{X}_i, \mathbf{u}_i, \mathbf{v}_i)$ obtenu lorsque la réalisation du vecteur de perturbations $\mathbf{v}$ prend une valeur particulière (figure 4.1).

Le problème que l'on cherche dans un premier temps à résoudre est un problème de planification classique. L'état initial du système est incertain et l'incertitude sur cet état est bornée. On note $\mathbf{x}_{\text{init}} \in \mathbb{X}_{\text{init}}$ l'état initial du système. L'état du système $\mathbf{x}_{\text{goal}}$ que l'on cherche à atteindre appartient lui-même

Algorithme 11 SetRRT(**in** : $K \in \mathbb{N}$, $\mathbb{X}_{\text{init}} \subset \mathbb{X}_{\text{free}}$, $\mathbb{X}_{\text{goal}} \subset \mathbb{X}_{\text{free}}$, $\Delta t \in \mathbb{R}^+$, **out** : G)

1: G.Init($\mathbb{X}_{\text{init}}$)
2: $i \leftarrow 0$
3: **Répéter**
4: $\quad \mathbb{X}_{\text{rand}} \leftarrow$ RandomSet($\mathbb{X}_{\text{free}}$)
5: $\quad \mathbb{X}_{\text{new}} \leftarrow$ SetRRTExtend(G, $\mathbb{X}_{\text{rand}}$)
6: **Jusqu'à** i++> K **or** ($\mathbb{X}_{\text{new}} \neq \emptyset$ **et** $\mathbb{X}_{\text{new}} \subseteq \mathbb{X}_{\text{goal}}$)
7: **Retourner** G

Algorithme 12 SrtRRTExtend(**in** : G, $\mathbb{X}_{\text{rand}}$, **out** : $\mathbb{X}_{\text{new}}$)

1: $\mathbb{X}_{\text{near}} \leftarrow$ NearestNeighbor(G, $\mathbb{X}_{\text{rand}}$)
2: $\mathbf{u} \leftarrow$ SelectInput($\mathbb{X}_{\text{rand}}$, $\mathbb{X}_{\text{near}}$)
3: $\mathbb{X}_{\text{new}} \leftarrow$ Prediction($\mathbb{X}_{\text{near}}$, $\mathbf{u}$, Δt)
4: **Si** CollisionFreePath($\mathbb{X}_{\text{near}}$,$\mathbb{X}_{\text{new}}$,$\mathbf{u}$,Δt) **alors**
5: $\quad G$.AddGuaranteedNode($\mathbb{X}_{\text{new}}$)
6: $\quad G$.AddGuaranteedEdge($\mathbb{X}_{\text{near}}$, $\mathbb{X}_{\text{new}}$, $\mathbf{u}$)
7: $\quad$ **Retourner** $\mathbb{X}_{\text{new}}$
8: **FinSi**
9: **Retourner** $\emptyset$

à un ensemble $\mathbb{X}_{\text{goal}}$. On ne cherchera pas à rejoindre un état précis $\mathbf{x}_{\text{goal}}$ mais plus modestement un ensemble d'états $\mathbb{X}_{\text{goal}}$. La condition d'arrivée est donc vérifiée lorsqu'il existe un ensemble d'états $\mathbb{X}_i$ entièrement inclus dans $\mathbb{X}_{\text{goal}}$.

Partant de n'importe quel état $\mathbf{x}_{\text{init}}$ appartenant à l'ensemble initial $\mathbb{X}_{\text{init}}$, on recherche une suite de commandes unique $\mathbf{u}_i, i = 1 \dots m$ telle que l'on rejoigne l'arrivée quelle que soit la réalisation du bruit $\mathbf{v} \in \mathbb{V}$:

$$\phi(\phi(\dots \phi(\mathbb{X}_{\text{init}}, \mathbf{u}_m, \mathbb{V}) \dots), \mathbf{u}_2, \mathbb{V}), \mathbf{u}_1, \mathbb{V}) \subset \mathbb{X}_{\text{goal}}. \tag{4.4}$$

4.1 Set-RRT

Afin de résoudre ce problème, nous présentons le planificateur Set-RRT [53] inspiré du RRT et qui permet de travailler avec des ensembles contenant les différents états possibles du système en présence d'incertitude.

L'algorithme Set-RRT (algorithme 11) décrit le fonctionnement de ce planificateur. Les arguments d'entrée en sont :

- l'ensemble initial $\mathbb{X}_{\text{init}}$ qui doit être inclus dans $\mathbb{X}_{\text{free}}$,
- l'ensemble d'arrivée $\mathbb{X}_{\text{goal}}$ qui doit être inclus dans $\mathbb{X}_{\text{free}}$,
- le nombre de nouveaux nœuds à générer dans l'arbre K,
- l'intervalle de temps Δt sur lequel le système va être intégré à chaque étape.

L'arbre G est créé à la ligne 1 avec comme racine l'ensemble initial $\mathbb{X}_{\text{init}}$. On rentre ensuite dans la boucle de création des nœuds, qui sont maintenant associés à des ensembles et non plus à des vecteurs ponctuels (ligne 3).

La création d'un nœud se fait en deux étapes. On choisit tout d'abord un ensemble aléatoire. Il faut noter que cet ensemble peut parfaitement être ponctuel. Cependant, utiliser un ensemble non ponctuel permet de remplacer $\mathbb{X}_{\text{rand}}$ par $\mathbb{X}_{\text{goal}}$ ce qui peut être intéressant pour les variantes ensembllistes de RRT-*Goalbias* et RRT-*Goalzoom*. Une fois cet ensemble créé, la fonction `SetRRTExtend` (algorithme 12) est appelée. Son fonctionnement est identique à celui de la fonction `RRTExtend` du RRT mais dans une version ensembliste. A la ligne 1, la fonction `NearestNeighbor` renvoie $\mathbb{X}_{\text{near}}$,

le nœud de G le plus proche de $\mathbb{X}_{\text{rand}}$. Il faut donc définir une métrique permettant de mesurer la distance entre deux ensembles. Cette métrique dépend bien sûr du type d'ensemble utilisé. Une commande est ensuite choisie à la ligne 2. A la ligne 3, la fonction `prediction` est utilisée. Cette fonction, tout comme la fonction `NewState` du RRT calcule le nouvel état ensembliste $\mathbb{X}_{\text{new}}$ en fonction de l'état actuel et de la commande. Finalement, un test de collision est effectué, qui doit garantir qu'aucun état ensembliste entre $\mathbb{X}_{\text{near}}$ et $\mathbb{X}_{\text{new}}$ ne soit situé dans $\mathbb{X}_{\text{obs}}$. Si aucune collision n'est détectée, l'état ensembliste est ajouté à l'arbre.

Cette fonction est itérée jusqu'à ce que K nœuds soient générés ou bien qu'un des nouveaux états $\mathbb{X}_{\text{new}}$ soit un sous-ensemble de $\mathbb{X}_{\text{goal}}$

Cet algorithme ressemble donc au RRT traditionnel, cependant plusieurs parties diffèrent et doivent être définies en fonction du type d'ensemble utilisé : la (pseudo-)métrique utilisée dans les fonctions `NearestNeighbor` et `SelectInput`, la fonction de prédiction ainsi que le test de collision. Nous spécialisons maintenant ces considérations au cas où $\mathbb{X}_i$ est enveloppé dans un vecteur d'intervalles ou pavé qui sont plus simples à manipuler que des ensembles quelconques.

4.2 Box-RRT

Afin d'illustrer l'algorithme Set-RRT présenté précédemment, nous prenons comme exemple d'ensemble les vecteurs d'intervalles. Nous avons appelé Box-RRT [53] cette mise en œuvre du Set-RRT avec des vecteurs d'intervalles. L'état du système est donc représenté par un vecteur d'intervalles aussi appelé pavé $[\mathbf{x}] \in \mathbb{IR}^n$, où n est la dimension de l'état du système.

Comme nous l'avons vu dans l'algorithme Set-RRT, les trois parties à définir sont la métrique, l'étape de prédiction ainsi que le test de collision.

4.2.1 Métrique

De nombreuses (pseudo-)métriques peuvent être utilisées afin de calculer la distance entre deux pavés. Par exemple, il est possible d'utiliser la borne supérieure de la distance euclidienne entre des éléments de deux pavés :

$$d([\mathbf{x}], [\mathbf{y}]) = \text{ub}\left(\sum_{i=1}^{n} ([x_i] - [y_i])^2\right). \tag{4.5}$$

La figure 4.2 illustre cette pseudo distance entre deux boites de dimensions 2. Il s'agit bien d'une pseudo distance puisque la fonction ne renvoie pas une valeur nulle lorsque les deux boites sont identiques.

Il est également possible d'utiliser la distance de Hausdorff [54] qui quantifie la dilatation qu'il faut faire subir à un pavé pour qu'il contienne l'autre.

La distance de Haussdorf entre deux vecteurs d'intervalles $[\mathbf{a}]$ et $[\mathbf{b}]$ est donnée par

$$h_\infty([\mathbf{a}], [\mathbf{b}]) = \max\{h^0_\infty([\mathbf{a}], [\mathbf{b}]), h^0_\infty([\mathbf{b}], [\mathbf{a}])\}, \tag{4.6}$$

avec

$$h^0_\infty([\mathbf{a}], [\mathbf{b}]) = \inf\{r \in \mathbb{R}^+ | \, [\mathbf{A}] \subset [\mathbf{B}] + r\,[\mathbf{U}]\} \tag{4.7}$$

et

$$[\mathbf{U}] = [-1, 1]^{\times n}. \tag{4.8}$$

A la différence de la pseudo-métrique précédente, la distance d'Hausdorff est une véritable mé-

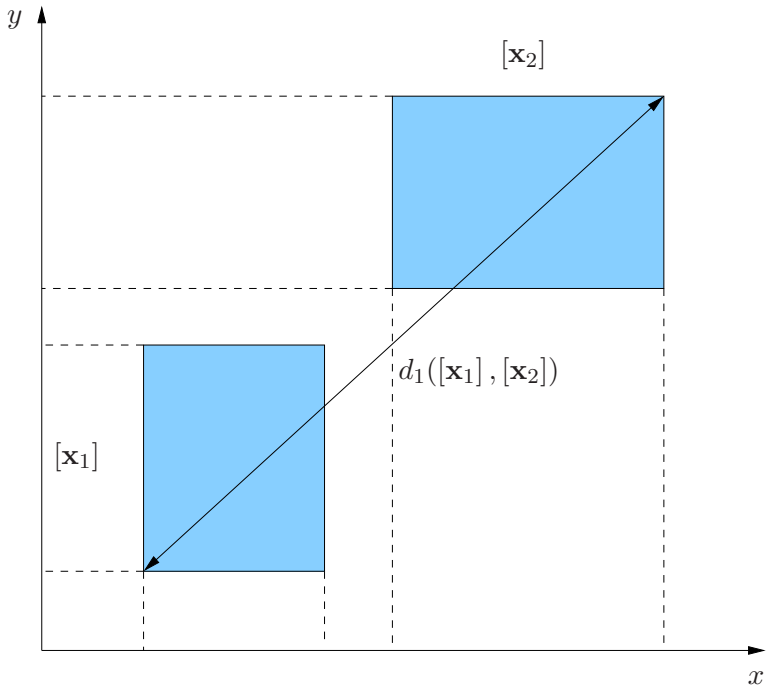


FIG. 4.2 – Borne supérieure de la distance euclidienne entre des éléments de deux pavés de dimension 2

trique puisque les trois propriétés suivants sont satisfaites :

$$h_\infty([\mathbf{a}]\,,[\mathbf{b}]) = 0 \rightarrow [\mathbf{a}] = [\mathbf{b}] \quad \text{(séparation)} \tag{4.9}$$

$$h_\infty([\mathbf{a}]\,,[\mathbf{b}]) = h_\infty([\mathbf{b}]\,,[\mathbf{a}]) \quad \text{(symétrie)} \tag{4.10}$$

$$h_\infty([\mathbf{a}]\,,[\mathbf{c}]) \leqslant h_\infty([\mathbf{a}]\,,[\mathbf{b}]) + h_\infty([\mathbf{b}]\,,[\mathbf{c}]) \quad \text{(inégalité triangulaire)} \tag{4.11}$$

Un exemple pour des vecteurs d'intervalles de dimension 2 est donné sur la figure 4.3.

4.2.2 Étape de prédiction

L'étape de prédiction doit être réalisée en utilisant une méthode d'intégration numérique garantie [55, 56] à l'instant $t + \Delta t$ de (4.1) pour calculer $[\mathbf{x}_{\text{new}}]$. Ainsi, partant de $[\mathbf{x}_{\text{near}}]$ à l'instant t et appliquant la commande $\mathbf{u} \in \mathbb{U}$ (considérée comme constante sur l'intervalle de temps Δt), on obtient un pavé $[\mathbf{x}_{\text{new}}]$ contenant le nouvel état de façon garantie à nos hypothèses de modélisation des incertitudes sont satisfaites.

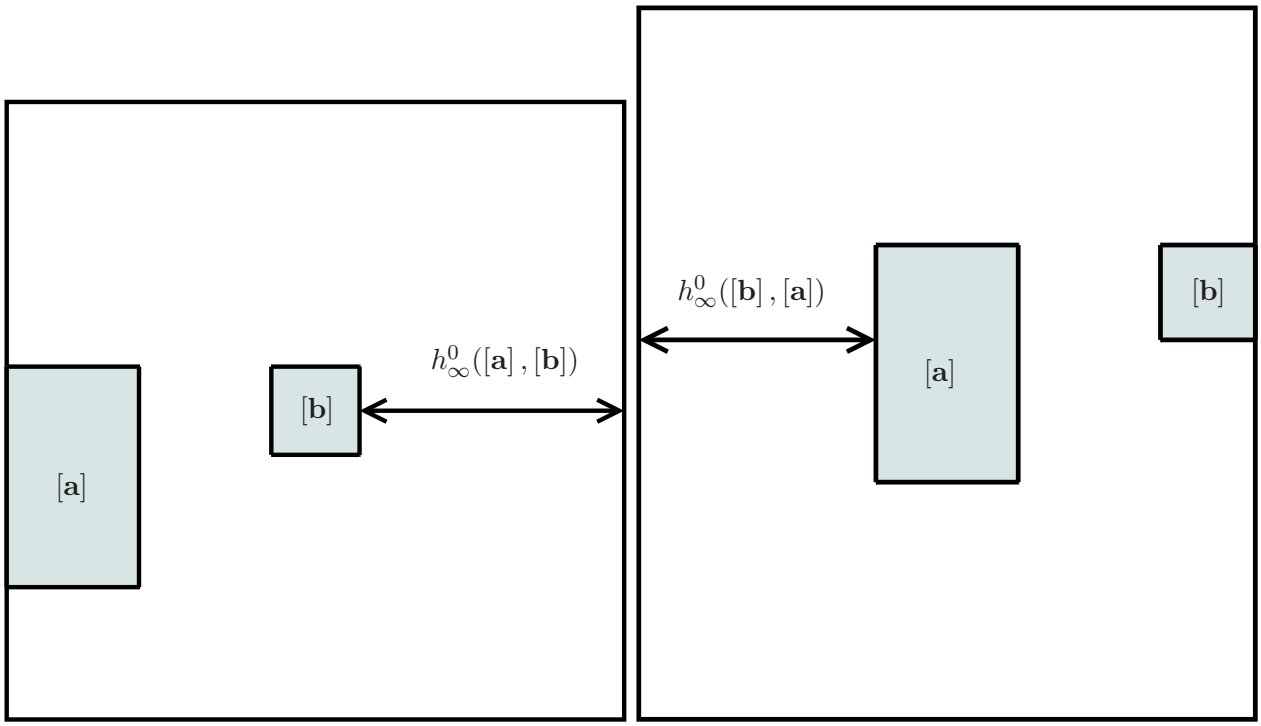
L'intégration numérique garantie [55,56] permet également de vérifier qu'aucune collision n'intervient durant l'intervalle de temps $[t, t + \Delta t]$. Pour cela, nous essayons de trouver une approximation extérieure $[\tilde{\mathbf{x}}_1] \subset \mathbb{IR}^n$ telle que

$$[\mathbf{x}_0] + [0, \Delta t]\mathbf{f}([\tilde{\mathbf{x}}_1], \mathbf{u}, \mathbb{V}) \subset [\tilde{\mathbf{x}}_1]. \tag{4.12}$$

Ceci permet de garantir [55] que :

$$\forall \mathbf{x} \in [\mathbf{x_0}]\,,\ \forall \mathbf{v} \in \mathbb{V},\ \forall t \in [t_0, t_0 + \Delta t]\ \mathbf{x}(t) \in [\tilde{\mathbf{x}}_1]. \tag{4.13}$$

Pour trouver $[\tilde{\mathbf{x}}_1]$, nous commençons par $[\tilde{\mathbf{x}}_1] = [\mathbf{x}_{\text{near}}] \sqcup [\mathbf{x}_{\text{new}}]$ le pavé qui enveloppe $[\mathbf{x}_{\text{near}}]$ et $[\mathbf{x}_{\text{new}}]$. Nous augmentons ensuite sa taille si nécessaire jusqu'à ce que (4.12) soit vérifiée. L'algo-



(a) $h^0_\infty([\mathbf{a}],[\mathbf{b}])$ (b) $h^0_\infty([\mathbf{b}],[\mathbf{a}])$

FIG. 4.3 – Distance de Hausdorff entre deux pavés de dimension 2

Algorithme 13 ApproximationExterne(**in** : $[\mathbf{x}_{\text{near}}]$, $[\mathbf{x}_{\text{new}}]$, $\mathbf{u}$, Δt, **out** : $[\tilde{\mathbf{x}}_1]]$)

1: $[\tilde{\mathbf{x}}_1] = [\mathbf{x}_{\text{near}}] \sqcup [\mathbf{x}_{\text{new}}]$
2: **TantQue** $[\mathbf{x}_{\text{near}}] + [0, \Delta t]\mathbf{f}([\tilde{\mathbf{x}}_1], \mathbf{u}, \mathbb{V}) \not\subset [\tilde{\mathbf{x}}_1]$ **faire**
3: $\quad [\tilde{\mathbf{x}}_1] \leftarrow [\tilde{\mathbf{x}}_1] + \epsilon\,[-1,1]^{\times n}$
4: **FinTantQue**

rithme 13 décrit l'obtention de cette approximation externe $[\tilde{\mathbf{x}}_1]$.

Il serait possible une fois cette approximation trouvée de l'affiner pour diminuer le pessimisme, notamment lorsque cette forme est utilisée dans le test de collision.

Une fois que l'on a une approximation externe de l'ensemble des états contenant la trajectoire du système, il reste à vérifier que cette approximation appartient à $\mathbb{X}_{\text{free}}$.

4.2.3 Test de collision

Avant d'ajouter un nœud au graphe G, nous devons déterminer si ce nœud est sûr, c'est-à-dire s'assurer que les états contenus dans un pavé d'états sont tous contenus dans $\mathbb{X}_{\text{free}}$. C'est le rôle du test de collision. Il faut dans un premier temps garantir que les pavés d'états de départ $[\mathbf{x}_{\text{init}}]$ et d'arrivée $[\mathbf{x}_{\text{goal}}]$ sont sûrs.

Il faut aussi vérifier que la trajectoire entre deux pavés consécutifs est sûre. Si c'est le cas, le nœud peut être ajouté. Ce test est utilisé dans la fonction `CollisionFreePath`. Si nous utilisons une simple enveloppe convexe dans ce cas, il est tout à fait possible que l'un des états du système appartienne à $\mathbb{X}_{\text{obs}}$ sans qu'on puisse le détecter. On voit ainsi sur la figure 4.4 que l'enveloppe convexe de $[\mathbf{x}]_i$ et $[\mathbf{x}]_{i+1}$ est entièrement contenue dans $\mathbb{X}_{\text{free}}$ alors que si l'on regarde la trajectoire réelle du système, certains états appartiennent à $\mathbb{X}_{\text{obs}}$. Cette trajectoire ne peut donc être ajoutée à notre arbre. Pour pouvoir ajouter cette trajectoire, une fois l'approximation extérieure $[\tilde{\mathbf{x}}_1]$ trouvée, il faut vérifier qu'aucun des états de $[\tilde{\mathbf{x}}_1]$ n'appartienne à $\mathbb{X}_{\text{obs}}$. L'utilisation d'une méthode d'intégration garantie

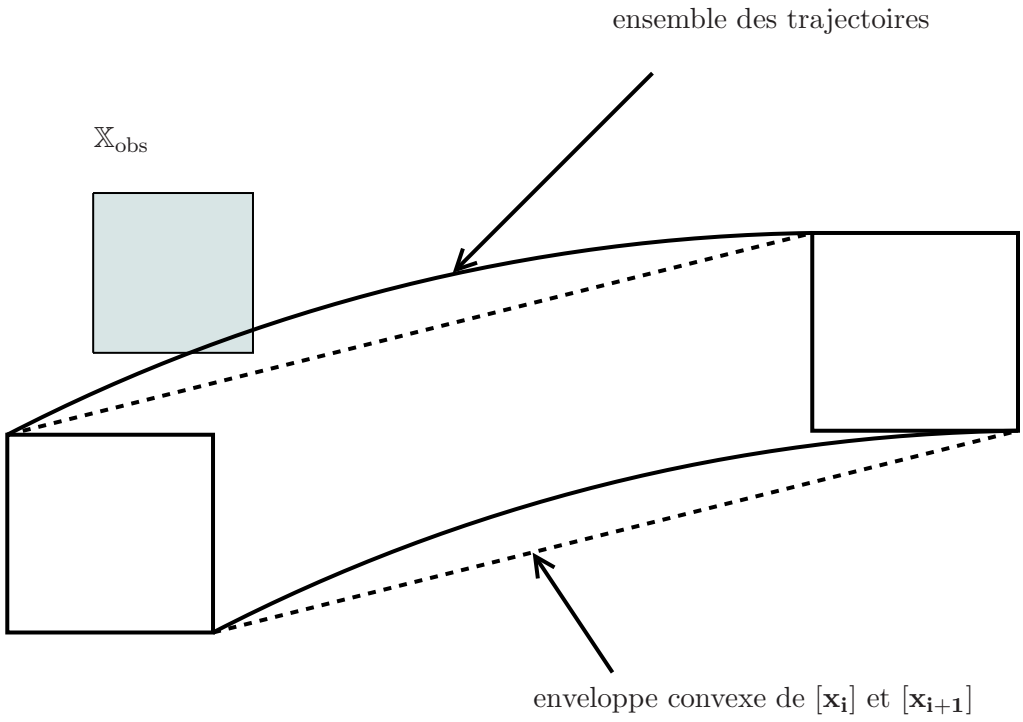


FIG. 4.4 – Différence entre l'ensemble des trajectoires entre $[\mathbf{x_i}]$ et $[\mathbf{x_{i+1}}]$ et l'enveloppe convexe de $[\mathbf{x_i}]$ et $[\mathbf{x_{i+1}}]$

Algorithme 14 BoxRRT(**in** : $K \in \mathbb{N}$, $[\mathbf{x}_{\text{init}}] \subset \mathbb{X}_{\text{free}}$, $[\mathbf{x}_{\text{goal}}] \subset \mathbb{X}_{\text{free}}$, $\Delta t \in \mathbb{R}^+$, **out** : G)

1: G.Init($[\mathbf{x}_{\text{init}}]$)
2: $i \leftarrow 0$
3: **Répéter**
4: $\quad [\mathbf{x}_{\text{rand}}] \leftarrow \text{RandomBox}(\mathbb{X}_{\text{free}})$
5: $\quad [\mathbf{x}_{\text{new}}] \leftarrow \text{BoxRRTExtend}(G, [\mathbf{x}_{\text{rand}}])$
6: **Jusqu'à** $i{+}{+} > K$ **ou** ($[\mathbf{x}_{\text{new}}] \neq \emptyset$ **et** $[\mathbf{x}_{\text{new}}] \subseteq [\mathbf{x}_{\text{goal}}]$)
7: **Retourner** G

ajoutera du pessimisme mais permettra de garantir qu'aucun état n'appartienne à $\mathbb{X}_{\text{obs}}$.

4.2.4 Algorithme

L'algorithme Box-RRT (algorithme 14) ainsi que la fonction `BoxRRTExtend` utilisée (algorithme 15) sont directement adaptés de la version ensembliste Set-RRT.

4.2.5 Exemples de trajectoires planifiées

Nous présentons dans cette section des exemples de trajectoires obtenues pour un objet ponctuel à deux degrés de liberté et ayant un vecteur commande de dimension 2. La figure 4.5 représente une trajectoire planifiée pour ce système dans un environnement comportant peu d'obstacles. L'erreur maximale sur le déplacement est de 2% sur x et y et de 1% sur θ. Le système régissant les mouvements de ce mobile est le suivant :

$$\dot{\mathbf{x}} = \frac{1}{1-\mathbf{v}}\mathbf{u} \tag{4.14}$$

avec $\mathbf{v} \in [-v_{\text{err}}, v_{\text{err}}]$ où v_{err} représente l'erreur maximale sur le déplacement, ici $v_{\text{err}} = 0.02$

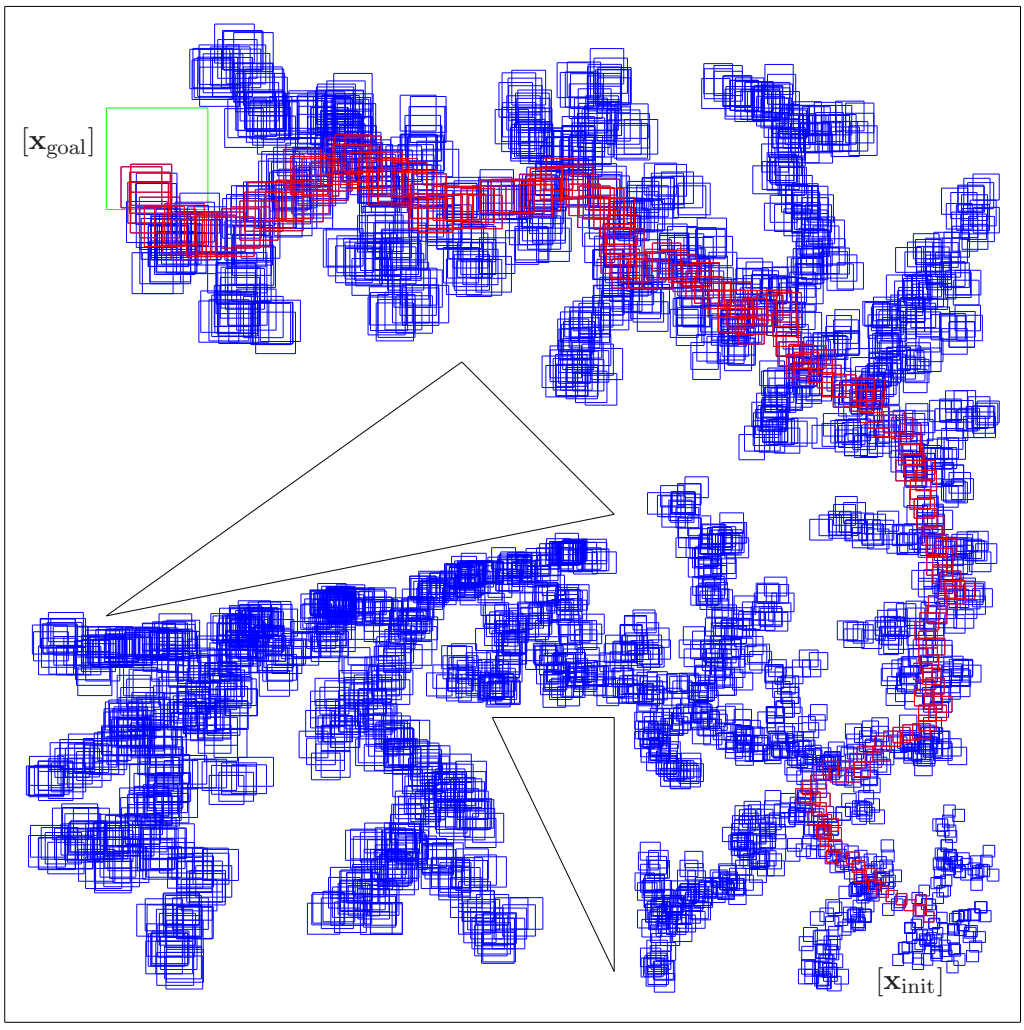


FIG. 4.5 – Exemple de trajectoire planifiée avec Box-RRT

Algorithme 15 BoxRRTExtend(**in** : G, $\mathbb{X}_{\text{rand}}$, **out** : $[\mathbf{x}_{\text{new}}]$)

```
[x_near] ← NearestNeighbor(G, [x_rand])
u ← SelectInput([x_rand], [x_near])
[x_new] ← Prediction([x_near], u, Δt)
Si CollisionFreePath([x_near],[x_new], u, Δt) alors
   G.AddGuaranteedNode([x_new])
   G.AddGuaranteedEdge([x_near], [x_new], u)
   Retourner [x_new]
FinSi
Retourner ∅
```

Algorithme 16 RéductionBoite(**in** : $[\mathbf{x}_{\text{near}}]$, $[\mathbf{x}_{\text{new}}]$)

```
[x_reduit] ← [x_new]
[x_near]_j ← Découpage([x_near], j)
Répéter
   PourTout [x_near]_j ∈ [x_near] faire
      estreduit ← RechercheCommande([x_near]_j, [x_reduit])
      Si (estreduit == FAUX) alors
         Retourner [x_reduit]
      FinSi
   FinPour
   [x_reduit] ← Réduire([x_reduit])
Jusqu'à VRAI
```

La figure 4.6 illustre le fait que l'extension *Goalbias* du RRT peut s'appliquer aisément au Set-RRT. La probabilité utilisée pour le tirage des points de *Goalbias* est ici $p = 0.1$. Nous utilisons ici un $[\mathbf{x}_{\text{rand}}]$ ponctuel.

Le planificateur que nous venons de présenter permet de générer des trajectoires garantissant qu'à aucun moment l'état ne rentre dans $\mathbb{X}_{\text{obs}}$ à condition que les hypothèses sur le bruit d'état soient satisfaites. Ainsi, contrairement à la représentation probabiliste utilisée dans le planificateur Kalman-RRT, l'incertitude détermine ici l'ensemble des états que peut prendre le système. Tous les états possibles du système sont connus et pris en compte à chaque instant de la planification. De plus, seule une phase de prédiction est ici utilisée. Il n'existe donc aucune approximation due à la simulation des mesures capteurs lors de la planification.

L'algorithme de planification proposé permet donc de garantir la sûreté de la trajectoire proposée. Cependant, la seule utilisation d'une phase de prédiction ne permet pas de résoudre certains problèmes puisque l'incertitude augmente à cause des perturbations d'état à chaque pas d'intégration du système.

4.3 Planification à commandes différenciées

L'algorithme recherche une suite de commandes *unique* qui permet au système partant d'un état quelconque contenu dans $[\mathbf{x}_{\text{init}}]$ de rejoindre $[\mathbf{x}_{\text{goal}}]$ et cela quelque soit les réalisations des perturbations d'état. Nous allons relaxer cette exigence et essayer maintenant non plus de trouver une commande unique quel que soit l'état initial incertain mais de trouver pour chaque état initial, une suite de commandes permettant d'atteindre $\mathbb{X}_{\text{goal}}$ quelles que soient les réalisations des perturbations d'état.

Précédemment, nous utilisions une commande unique notée $\mathbf{u}$ afin de faire passer du pavé $[\mathbf{x}_{\text{near}}]$

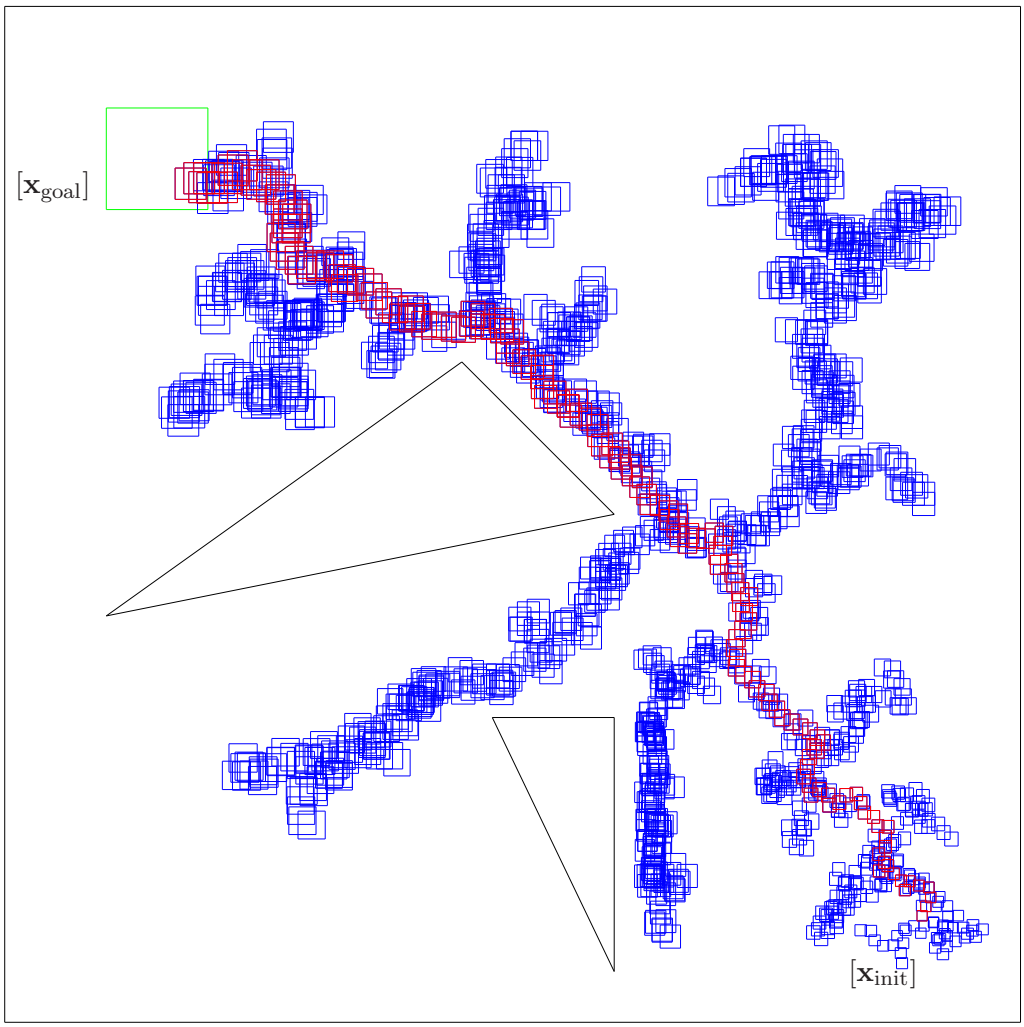


FIG. 4.6 – Exemple de trajectoire planifiée avec la modification *Goalbias* de Box-RRT

Algorithme 17 RechercheCommande(**in** : $[\mathbf{x}_{\text{near}}]$, $[\mathbf{x}_{\text{reduit}}]$)

1: $\mathbb{A} \leftarrow [\mathbf{u}]$
2: **TantQue** $\mathbb{A} \neq \emptyset$ **faire**
3: $[\mathbf{c}] \leftarrow \text{Pop}(\mathbb{A})$
4: **Si** $\phi\left([\mathbf{x}_{\text{near}}]_j, \text{mid}([\mathbf{c}]), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}]$ **alors**
5: **Retourner** SUCCES
6: **Sinon**
7: **Si** $w([\mathbf{c}]) < \epsilon$ **alors**
8: **Retourner** ECHEC
9: **Sinon**
10: $\{[\mathbf{c}_{\text{left}}], [\mathbf{c}_{\text{right}}]\} \leftarrow \text{Bissection}([\mathbf{c}])$
11: $\mathbb{A} += [\mathbf{c}_{\text{left}}]$
12: $\mathbb{A} += [\mathbf{c}_{\text{right}}]$
13: **FinSi**
14: **FinSi**
15: **FinTantQue**

au pavé $[\mathbf{x}_{\text{new}}]$ (figure 4.7) en utilisant (4.1).

Nous allons maintenant, pour un état incertain contenu dans un pavé $[\mathbf{x}_{\text{near}}]$, chercher un ensemble de commandes γ_j permettant à chacun des états contenu dans $[\mathbf{x}_{\text{near}}]$ de rejoindre un pavé entièrement contenu $[\mathbf{x}_{\text{new}}]$.

Utiliser une commande différenciée permettra ainsi d'obtenir des vecteurs d'intervalles plus petits et ainsi de trouver des trajectoires qui n'appartenaient pas à $\mathbb{X}_{\text{free}}$ précédemment.

Nous allons procéder en deux étapes. Tout d'abord, nous étendrons l'arbre comme nous le faisions avec une commande unique $\mathbf{u}$. Après avoir obtenu l'état $[\mathbf{x}_{\text{new}}]$, nous tenterons de le réduire dans une seconde phase.

Pour cela, nous choisissons tout d'abord un pavé $[\mathbf{x}_{\text{reduit}}] \subset [\mathbf{x}_{\text{new}}]$, une réduction de $[\mathbf{x}_{\text{new}}]$ qui caractérisera l'ensemble des états à tenter de rejoindre (figure 4.8). Nous partitionnerons ensuite le pavé de départ $[\mathbf{x}_{\text{near}}]$ en sous-pavés, que nous appellerons $[\mathbf{x}_{\text{near}}]_j$ où j représente le j-ème sous-pavé ainsi produit. Pour chacun de ces sous-pavés $[\mathbf{x}_{\text{near}}]_j$, nous cherchons une commande γ_i qui permette de rejoindre $[\mathbf{x}_{\text{reduit}}]$ (figure 4.9).

Pour cela, nous prenons l'intervalle $[\mathbf{u}]$ comme point de départ de notre recherche de la commande $[\mathbf{c}]_j$. Nous allons tester si la commande située au milieu de cet intervalle $[\mathbf{c}]_j$ que nous notons $\text{mid}([\mathbf{c}]_j)$ permet, une fois appliquée à $[\mathbf{x}_{\text{near}}]_j$, de rejoindre $[\mathbf{x}_{\text{reduit}}]$ malgré la perturbation $\mathbf{v}_j \in [\mathbf{v}]$. Ainsi, nous voulons tester si

$$\phi\left([\mathbf{x}_{\text{near}}]_j, \text{mid}([\mathbf{c}]_j), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}]\,. \tag{4.15}$$

Si (4.15) est vérifiée, nous avons prouvé que pour chaque état contenu dans $[\mathbf{x}_{\text{near}}]_j$, la commande $\gamma_j = \text{mid}([\mathbf{c}]_j)$ permet de rejoindre un pavé entièrement contenu dans $[\mathbf{x}_{\text{reduit}}]$, c'est-à-dire que

$$\forall \mathbf{x} \in [\mathbf{x}_{\text{near}}]_j, \phi\left(\mathbf{x}, \text{mid}([\mathbf{c}]_j), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}]\,. \tag{4.16}$$

Si ce n'est pas le cas, nous bissectons $[\mathbf{c}]_j$ en sous-pavés afin de tester si d'autres commandes le permettent (figure 4.10). La bissection s'arrêtera lorsqu'une commande est trouvée, c'est à dire que

$$\forall \mathbf{x} \in [\mathbf{x}_{\text{near}}]_j, \exists [\mathbf{c}]_j \subset [\mathbf{u}]\,, \phi\left(\mathbf{x}, \text{mid}([\mathbf{c}]_j), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}] \tag{4.17}$$

ou bien lorsque la taille des pavés issus de la bissection devient trop petite.

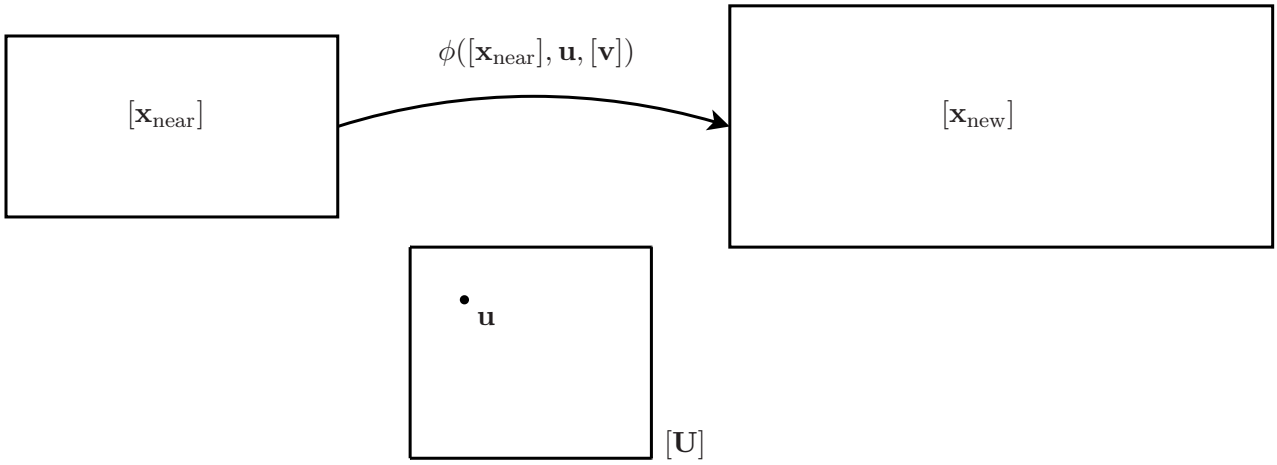


FIG. 4.7 – Le RRT applique la commande unique $\mathbf{u}$ à $[\mathbf{x}_{\text{near}}]$ afin d'aller dans $[\mathbf{x}_{\text{new}}]$

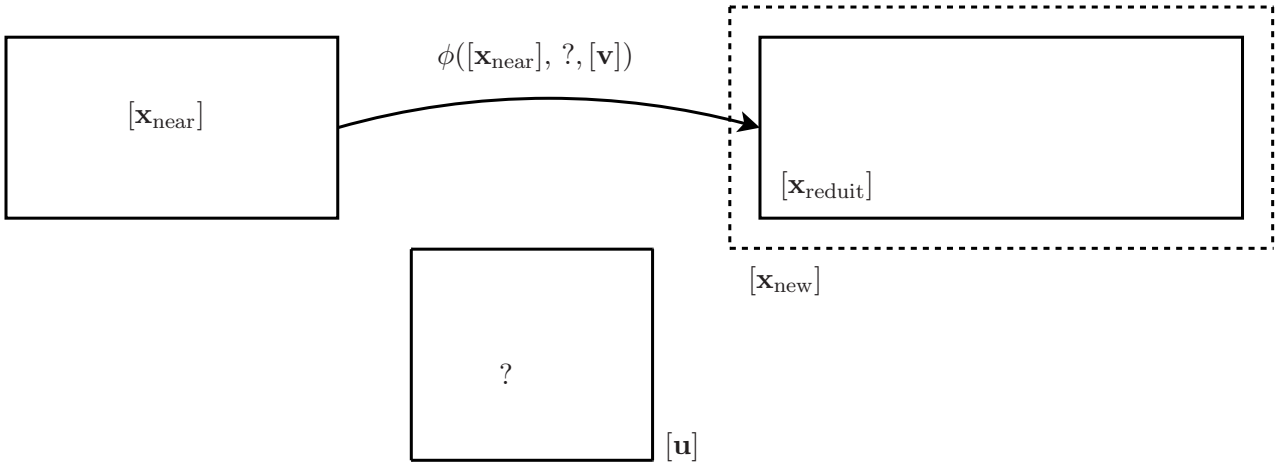


FIG. 4.8 – Le but est de trouver une commande qui permette à $[\mathbf{x}_{\text{near}}]$ de rejoindre $[\mathbf{x}_{\text{reduit}}] \subset [\mathbf{x}_{\text{new}}]$

Nous devons maintenant trouver une telle commande pour tous les sous-pavés $[\mathbf{x}_{\text{near}}]_j$ contenus dans $[\mathbf{x}_{\text{near}}]$. Nous prouverons ainsi qu'il existe une commande pour chacun de ses sous-pavés qui permet de rejoindre des vecteurs d'intervalles entièrement contenus dans $[\mathbf{x}_{\text{reduit}}]$:

$$\forall [\mathbf{x}_{\text{near}}]_j \subset [\mathbf{x}_{\text{near}}], \forall \mathbf{x} \in [\mathbf{x}_{\text{near}}]_j, \exists [\mathbf{c}]_j \subset [\mathbf{u}], \phi\left(\mathbf{x}, \text{mid}([\mathbf{c}]_j), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}]. \tag{4.18}$$

Lorsqu'une commande est trouvée pour chaque $[\mathbf{x}_{\text{near}}]_j$, nous pouvons réduire $[\mathbf{x}_{\text{reduit}}]$ plus significativement et réitérer le processus afin de voir si l'on peut toujours rejoindre $[\mathbf{x}_{\text{reduit}}]$. Cette réduction de $[\mathbf{x}_{\text{reduit}}]$ continuera tant qu'une commande existera pour chaque $[\mathbf{x}_{\text{near}}]_j$.

Dans le plan retourné par le planificateur, il se sera plus question d'une suite de commandes à appliquer afin de passer de l'état $[\mathbf{x}_{\text{init}}]$ à l'état $[\mathbf{x}_{\text{goal}}]$. En effet, le planificateur retourne ici un ensemble de commandes, chacune permettant de faire passer le système d'un état appartenant à $[\mathbf{x}_{\text{near}}]_i$ à un état appartenant à $[\mathbf{x}_{\text{reduit}}]$. Dans la pratique, il sera possible de déduire dans quelle partie de $[\mathbf{x}_{\text{near}}]$ se situe le système en utilisant par exemple des amers ou des zones de relocalisation. On pourra ainsi décider de la commande à utiliser.

Cette réduction de la taille de $[\mathbf{x}_{\text{new}}]$ permet de trouver un nombre plus important de trajectoires appartenant à $\mathbb{X}_{\text{free}}$ et ainsi de trouver des trajectoires pour des problèmes qui étaient insolubles précédemment.

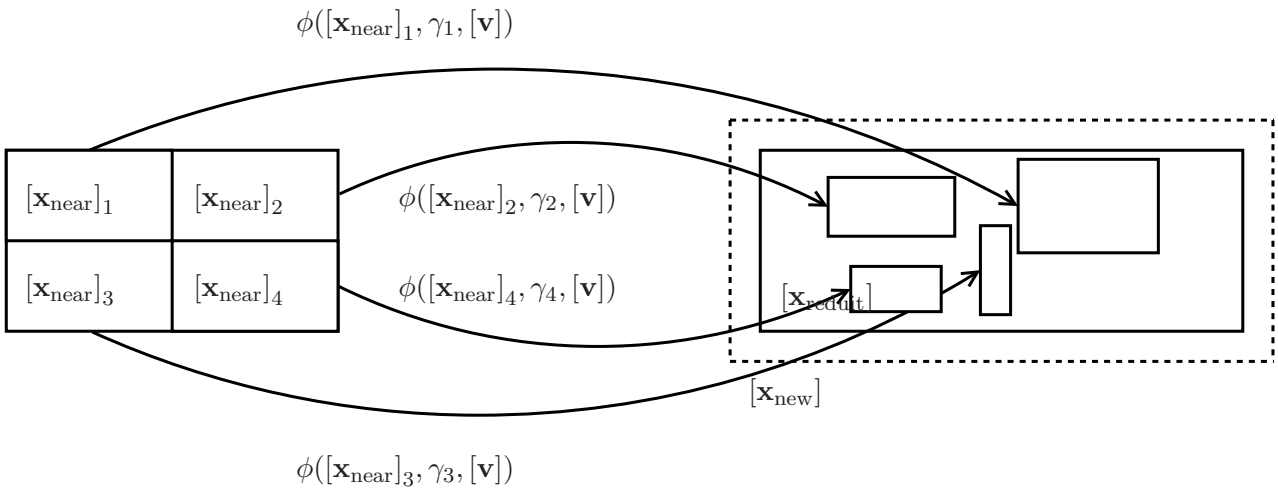


FIG. 4.9 – On bissecte $[\mathbf{x}_{\text{near}}]$ et pour chaque sous-intervalle $[\mathbf{x}_{\text{near}}]_j$ on cherche une commande γ_j telle que $\phi([\mathbf{x}_{\text{near}}]_j, \gamma_j, \mathbf{v}_j) \subset [\mathbf{x}_{\text{reduit}}]$

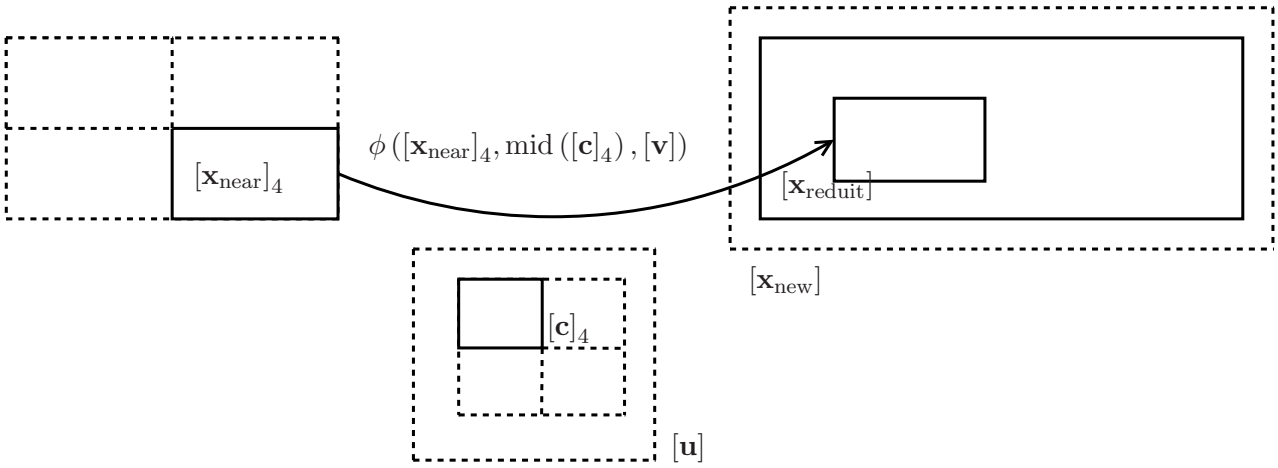


FIG. 4.10 – On bissecte le vecteur d'intervalle $[\mathbf{c}]_4$ jusqu'à trouver $[\mathbf{c}]_4$ tel que $\phi\left([\mathbf{x}_{\text{near}}]_4, \text{mid}\left([\mathbf{c}]_4\right), [\mathbf{v}]\right) \subset [\mathbf{x}_{\text{reduit}}]$

4.4 Conclusion

L'algorithme présenté permet de planifier des trajectoires qui garantissent qu'il n'y aura pas collision durant la phase de navigation. Cette garantie est due à l'utilisation d'ensembles pour représenter tous les états que peut prendre le système à un instant donné et à l'utilisation d'outils d'intégration numérique garantie qui permettent d'enfermer toutes les solutions d'un système d'équations différentielles incertaines. De plus, afin d'éviter les problèmes dûs à la simulation des capteurs extéroceptifs, seule une étape de prédiction est utilisée.

Dans un deuxième temps, nous avons relaxé la contrainte sur la commande afin de trouver plusieurs commandes pour un même ensemble d'états. L'utilisation d'une telle méthode permet de réduire la taille de l'incertitude pour un état donné. Il en suit que de nombreuses trajectoires qu'on ne peut détecter lorsque l'on cherche une commande unique pour tous les cas peuvent être utilisées maintenant.

Deuxième partie

Planification sous contraintes différentielles

Chapitre 5

Planification sous contraintes en robotique

Sommaire

Ce chapitre présente ce que sont les contraintes différentielles et comment les intégrer à un planificateur de trajectoires. Nous nous focaliserons sur les problème de planification en robotique, et plus particulièrement sur la planification pour un véhicule de type voiture.

Les modèles différentiels s'expriment sous la forme d'un système différentiel d'état

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}). \tag{5.1}$$

Cette équation différentielle, déjà utilisée notamment dans la présentation du planificateur RRT, décrit comment l'état va évoluer en fonction de sa valeur actuelle et des commandes à appliquer.

Dans le paragraphe 5.1, nous présenterons les différentes contraintes rencontrées en robotique et un modèle simple de véhicule. Puis, nous intégrerons ce modèle dans un planificateur à échantillonnage de type RRT dans le paragraphe 5.3. Nous ne détaillerons pas l'intégration de modèles dans d'autres types de planificateurs. Dans le paragraphe 5.4, nous présenterons des modèles de véhicules plus évolués et nous résoudrons des problèmes de planification les intégrant.

5.1 Contraintes en robotique

Dans la première partie, nous avons considéré des problèmes de planification génériques. Dans cette seconde partie, le domaine d'application est la robotique et les techniques présentées précédemment doivent être mises en œuvre pour un véhicule ce qui impose la prise en compte de contraintes spécifiques.

5.1.1 Non-holonomie

Les système robotiques (ou robots) doivent en général faire face à une contrainte forte appelée *non-holonomie*. Cette contrainte s'explique par un nombre de commandes inférieur au nombre de degrés de liberté du système et implique que certaines directions ne sont pas autorisées pour les changements d'état.

Prenons l'exemple d'une voiture. Nous savons bien qu'il est impossible de réaliser un mouvement de translation latérale et que cela nécessite un créneau. Ceci est dû aux roues arrières qui devraient glisser au lieu de rouler pour réaliser un tel mouvement. Si les roues arrières étaient elles aussi orientables, la manœuvre pour garer une voiture serait plus simple. On dit que la voiture est un système non-holonome. Le cas d'une voiture est assez explicite, le nombre de variables de configuration étant de trois (les coordonnées du point de référence de la voiture (x, y) et son orientation θ) alors que le conducteur ne peut activer que deux commandes : les pédales d'accélérateur/frein et le volant.

5.1.2 Robot et voiture

En environnement intérieur, les robots ne se déplacent en général pas comme des voitures. le type de robot utilisé est souvent un modèle à deux roues équipé d'un différentiel permettant à chacune des roues d'évoluer indépendamment de l'autre (figure 5.1). Le vecteur de commande $\mathbf{u}$ est alors composé des vitesses angulaires de chacune des roues : u_{r} pour la roue droite, u_{l} pour la gauche. Si les deux vitesses angulaires sont égales et non nulles, alors le véhicule se déplace en ligne droite. De même, si $u_{\mathrm{l}} = -u_{\mathrm{r}}$ alors le robot tournera sur lui-même.

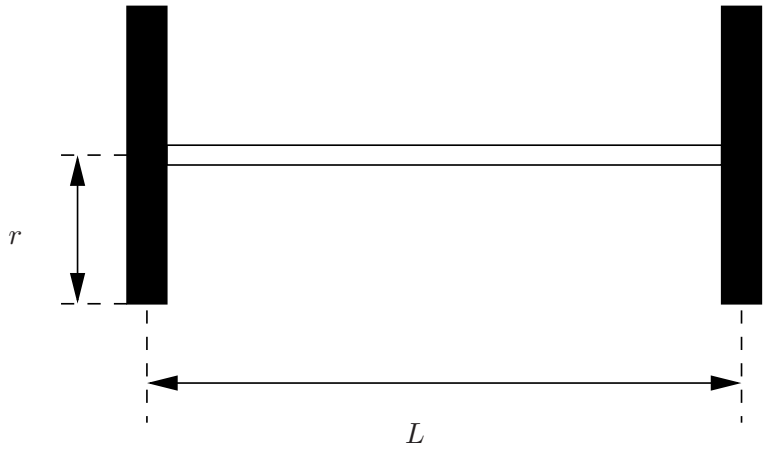


FIG. 5.1 – Modèle de robot à un essieu

Le système différentiel régissant les mouvements de ce robot est le suivant :

$$\dot{x} = \frac{r(u_r + u_l)}{2} \cos\theta, \tag{5.2}$$

$$\dot{y} = \frac{r(u_r + u_l)}{2} \sin\theta, \tag{5.3}$$

$$\dot{\theta} = \frac{r(u_r - u_l)}{L}, \tag{5.4}$$

avec r le rayon des roues et L la distance entre les deux roues.

Ce modèle rentre dans la catégorie des système non-holonomes puisque le nombre de commandes, ici 2, est inférieur à la dimension du vecteur d'état, 3. Cependant, en utilisant un tel modèle, il est possible de planifier comme nous le faisions dans les chapitres précédents en utilisant une suite de segments de droite. Bien qu'un robot de ce type ne soit pas capable d'effectuer un mouvement de translation latéral, il peut très bien compenser ceci grâce à sa capacité à tourner sur lui-même. En effet, le passage entre deux états quelconque $\mathbf{x}_1 = (x_1, y_1, \theta_1)$ et $\mathbf{x}_2 = (x_2, y_2, \theta_2)$, en imaginant que $\mathbb{X}_{\text{obs}} = \emptyset$, peut se faire en trois étapes :

– Le robot, dont l'état est $\mathbf{x}_1$ va tourner sur lui même afin de pourvoir se diriger vers (x_2, y_2). Son état passe ainsi de $\mathbf{x}_1$ à
$$\left(x_1, y_1, \tan\left(\frac{y_2 - y_1}{x_2 - x_1}\right)\right). \tag{5.5}$$

– Le robot se déplace alors en ligne droite afin de rejoindre l'état
$$\left(x_2, y_2, \tan\left(\frac{y_2 - y_1}{x_2 - x_1}\right)\right). \tag{5.6}$$

– Il tourne finalement sur lui-même pour atteindre $\mathbf{x}_2$.

Ainsi, dans le cadre d'un problème de planification, la contrainte de non-holonomie s'appliquant au système pour ce modèle précis de robot ne pose pas de difficulté. Il suffit de rechercher une suite de segments de droites permettant d'atteindre l'état désiré.

Pour une voiture, cette solution n'existe pas puisqu'elle ne peut tourner sur elle-même.

Dans la suite de ce manuscrit, nous ne nous intéresserons qu'au cas des véhicules de type voiture, et le terme de véhicule ne sera d'ailleurs utilisé que pour désigner les systèmes non-holonomes de type voiture.

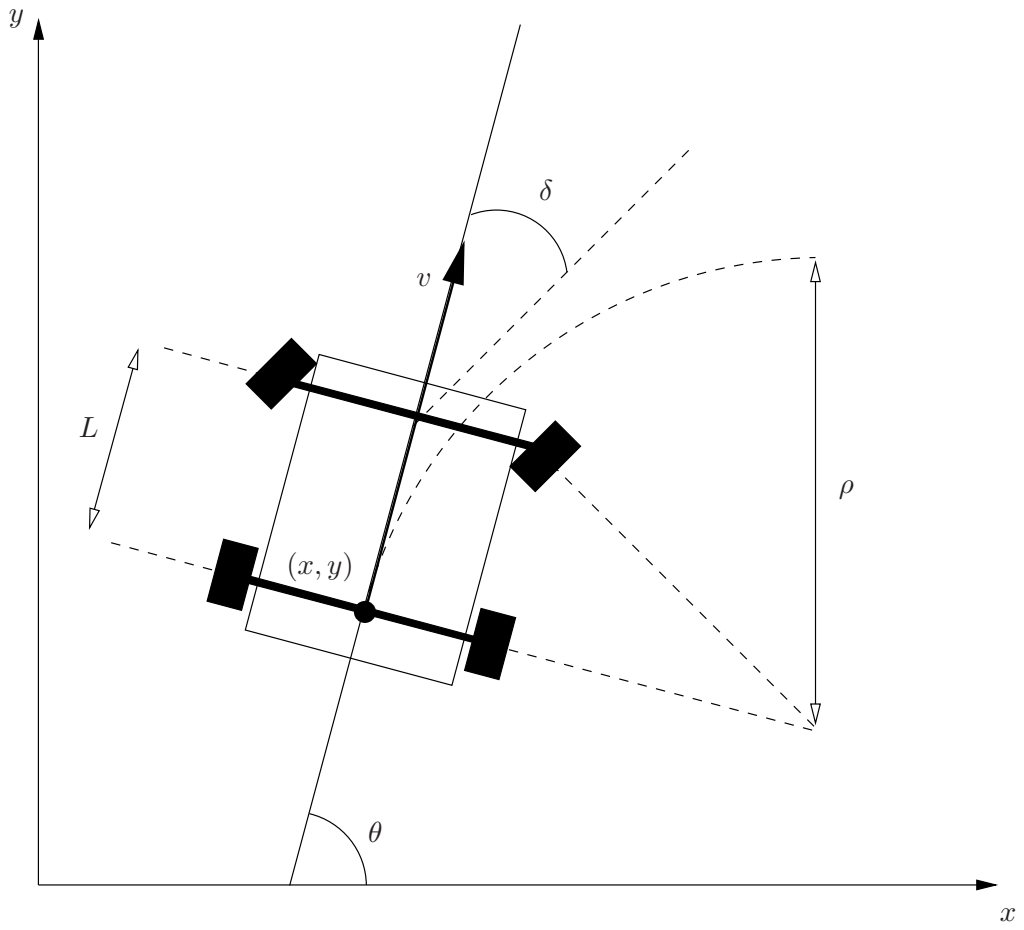


FIG. 5.2 – *Voiture simple*

5.1.3 *Voiture simple*

Le modèle de type voiture présenté sur la figure 5.2 appelé *voiture simple* [57] est un modèle cinématique de véhicule intégrant la contrainte de non-holonomie.

A la différence du robot présenté dans le paragraphe 5.1.2, la *voiture simple* ne peut pas tourner sur elle-même et enchaîner des segments de droites en guise de trajectoires.

Sur la figure 5.2, v représente la vitesse du véhicule, θ son orientation par rapport au repère de référence, (x, y) les coordonnées de son point de référence dans le repère de référence, δ est l'angle de braquage, c'est à dire l'angle entre l'orientation des roues et l'orientation du véhicule. ρ est le rayon de braquage.

Dans un petit intervalle de temps Δt, le mouvement de la voiture peut être approximé dans la direction des roues avant. Si l'intervalle de temps Δt tend vers 0, cela implique que [25]

$$\frac{dy}{dx} = \tan\theta. \tag{5.7}$$

L'équation (5.7) peut être récrite comme

$$-\dot{x}\sin\theta + \dot{y}\cos\theta = 0. \tag{5.8}$$

Cette contrainte est vérifiée si $\dot{x} = \cos\theta$ et $\dot{y} = \sin\theta$. Ces valeurs peuvent être multipliées par n'importe quel scalaire et toujours vérifier la contrainte. Ce scalaire correspond directement à la

vitesse du véhicule v. Ainsi, les deux premières composantes du système différentiel sont

$$\dot{x} = v\cos\theta, \tag{5.9}$$
$$\dot{y} = v\sin\theta. \tag{5.10}$$

Afin de trouver la dernière composante qui spécifie $\dot{\theta}$, nous devons considérer la distance parcourue par le véhicule que l'on note w. Ainsi, $dw = \rho d\theta$. De plus par trigonométrie on a $\rho = \frac{L}{\tan\delta}$. On obtient

$$d\theta = \frac{\tan\delta}{L} dw. \tag{5.11}$$

En divisant les deux cotés de l'équation par dt et en utilisant le fait que $\dot{w} = v$, on obtient la troisième composante de l'équation d'état

$$\dot{\theta} = \frac{v}{L}\tan\delta. \tag{5.12}$$

La *voiture simple* est dont régie par l'équation d'état suivante :

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \frac{v}{L}\tan\delta. \end{aligned} \tag{5.13}$$

Ce modèle n'est bien sûr pas complet si l'on ne spécifie pas l'ensemble des commandes $\mathbf{u} = (v, \theta)$ utilisables. On part du principe que la *voiture simple* se déplace lentement afin de pouvoir négliger sa dynamique. Il faut noter que si l'on prend une vitesse positive ou nulle dans ce type de modèle, le véhicule ne pourra se déplacer qu'en marche avant. Ainsi, une configuration dans laquelle un véhicule fait face au mur ne pourra plus être utilisée si la marche arrière est impossible. La commande au niveau de l'angle de braquage doit également être bornée afin de refléter la situation d'une voiture. En plus d'être problématique pour l'intégration de notre système différentiel, un angle de braquage de $\pi/2$ ou de $-\pi/2$ n'est pas admissible par une voiture. Ainsi, l'angle de braquage maximal δ_{max} devra respecter la contrainte $|\delta_{max}| < \frac{\pi}{2}$.

Il existe plusieurs versions de ce modèle [25] en fonction des contraintes appliquées sur les commandes :

- la *voiture simple* que nous venons de présenter lorsque $v \in [-1; 1]$ et $\theta \in [-\delta_{max}; \delta_{max}]$ avec $|\delta_{max}| < \frac{\pi}{2}$,
- la voiture de Reeds-Sheep [58] limite la vitesse à $v = \{-1, 0, 1\}$,
- la voiture de Dubins [59] est identique à la voiture de Reeds-Sheep mais n'autorise pas la marche arrière. C'est ce modèle qui sera utilisé dans la suite du manuscrit.

Géométriquement, les trajectoires générées sont des clothoïdes, c'est à dire qu'elles sont composées de segments de droites et d'arcs de cercles donc le rayon minimal dépend de l'angle de braquage maximum δ_{max}.

5.2 Problème de planification pour véhicule

Le problème de planification à résoudre consiste toujours en la recherche d'une trajectoire entre l'état initial $\mathbf{x}_{\text{init}}$ et un sous ensemble $\mathbb{X}_{\text{goal}}$ de l'espace d'état $\mathbb{X}$.

La principale différence entre les problèmes génériques de planification, que nous présentions dans la première partie, et les problèmes de planification en robotique se situe au niveau du système à déplacer. Comme nous l'avons vu pour le modèle de la *voiture simple*, ce système intègre la non-holonomie dans le système différentiel régissant son changement d'état. Cet état est au minimum de

dimension 3 pour un véhicule non-holonome puisqu'en plus de la position du robot sur le plan $\{x, y\}$, il faut considérer son orientation par rapport à ce plan. Ainsi, si l'on note $\mathcal{V}$ le repère du véhicule et $\mathcal{W}$ le repère lié à l'environnement, les trois composantes obligatoires de l'espace d'état seront la position de $\mathcal{V}$ dans $\mathcal{W}$ et l'orientation de $\mathcal{V}$ par rapport à $\mathcal{W}$.

Le sous-ensemble $\mathbb{X}_{\text{obs}} \subset \mathbb{X}$ est, dans le cadre de la robotique, directement lié aux obstacles que contient l'environnement. Ce sous-ensemble est donc composé de l'ensemble des états pour lesquels le véhicule rentre en collision avec un obstacle.

Un obstacle peut être défini comme une partie du monde occupée en permanence. Ce peut être, par exemple, une maison ou un mur sur les cartes de l'environnement empêchant le passage du véhicule. Les obstacles peuvent être représentés de nombreuses façons. La manière la plus commune est l'utilisation de polygones qui peuvent être convexes ou non. Un polygone convexe est défini par l'intersection de demi-plans. La représentation informatique d'une telle forme géométrique peut se faire par la définition des coordonnées de ses sommets ou bien par les arêtes entre deux sommets consécutifs. Un polygone non convexe est en général défini par la combinaison de plusieurs polygones convexes. Ainsi, un polygone non convexe $\mathcal{P}$ peut être défini par

$$\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \dots \tag{5.14}$$

où les $\mathcal{P}_i$ sont des polygones convexes.

D'autres représentations sont possibles telles que les polygônes à trou, les pyramides à base triangulaire qui sont fréquemment utilisées en informatique ou bien les bitmaps. Dans le cas d'un bitmap, chaque pixel correspond à un carré de coté donné. Dans le cas d'une image en niveau de gris, l'intensité du pixel correspond à une probabilité que l'obstacle soit situé à l'emplacement indiqué par le pixel. Ce qui revient à parler de difficulté à traverser certaines zones de l'environnement.

Définir l'ensemble $\mathbb{X}_{\text{obs}}$ pour un véhicule et un environnement donné est difficile. Dans le cas d'un système ponctuel, l'ensemble contenant les obstacles est confondu avec l'ensemble $\mathbb{X}_{\text{obs}}$. Ainsi, si la configuration du système ponctuel est sur un obstacle, alors le système est en collision avec l'obstacle et son état appartient donc à $\mathbb{X}_{\text{obs}}$.

En robotique, un véhicule n'est pas un système ponctuel. La solution la plus simple réside donc souvent dans l'utilisation d'un test de collision qui indique si un véhicule, situé en une position donnée, rentre en collision avec un obstacle.

5.3 Intégration au planificateur de type RRT

Dans cette partie, nous allons intégrer le modèle de *voiture simple* au planificateur de type RRT en ne considérant pas, pour le moment, les incertitudes liées au système.

L'intégration d'un modèle quel qu'il soit dans le RRT ne pose pas de problème puisque c'est la fonction `NewState` qui se charge de calculer le nouvel état du véhicule en fonction de son état actuel et de la commande à lui appliquer. Ainsi, tout système régit par une équation de la forme

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{5.15}$$

peut être pris en charge par le planificateur.

L'intégration de (5.15) peut parfois se faire analytiquement, par exemple lorsque la commande est considérée comme constante pendant l'intervalle de temps sur lequel on intègre le système. Elle peut également se faire par des méthodes numériques telles que la méthode d'Euler ou la méthode de Runge Kutta à l'ordre 4, par exemple.

L'ensemble des améliorations (RRT-*Goalbias*, RRT-*Goalzoom*, RRT-*Connect*) présentées dans le premier chapitre restent ici d'actualité. Il faut cependant noter que le Dual-RRT (paragraphe 1.3.7.1

page 41) n'est ici plus possible. Il est en effet très peu probable, pour ne pas dire impossible, que les deux arbres se rejoignent et que le point de jonction respecte les contraintes différentielles. Il faudrait en effet que les deux arbres aient en commun un nœud. La possibilité que cela arrive est d'autant plus faible que la dimension de l'espace d'état est grand. On utilisera donc dans la suite un seul arbre afin de planifier la trajectoire.

5.3.1 Mise en œuvre et optimisation du RRT

La mise en œuvre du RRT respecte strictement l'algorithme présenté dans le chapitre 1. Cependant, afin d'optimiser la vitesse de planification et de profiter pleinement des capacités des processeurs modernes, le RRT a été parallélisé.

La façon la plus simple de procéder à cette parallélisation est d'utiliser plusieurs arbres comme expliqué précédemment avec les Dual-RRT. Les processus légers ont ainsi chacun leur propre arbre dans lequel ils ajoutent leurs nœuds, il n'y a donc pas, a priori, besoin de partager une zone mémoire. Cependant, si l'on veut trouver une trajectoire qui utilise plusieurs de ces arbres, il est nécessaire d'avoir au moins un processus contenant l'ensemble de ces arbres afin de voir si le nœud ajouté est très proche voire confondu avec un nœud d'un des autres arbres. Si ce nœud existe, on a trouvé la trajectoire entre les racines des deux arbres.

Cette solution est loin d'être idéale. Le processus qui scrute les différents arbres a une utilisation importante du processeur puisque les calculs qu'il doit réaliser sont lourds. De plus, comme nous venons de le voir, le Dual-RRT s'applique très mal aux systèmes dont le nombre de degrés de liberté est important.

L'idéal est donc de paralléliser le RRT simple. On garde ainsi une des propriété principale du RRT, qui faisait défaut avec les deux arbres : l'arbre est toujours connecté. La mise en place d'un parallélisme sur le RRT reste relativement simple. Chaque processus va travailler sur le même arbre. Pendant la lecture de l'arbre afin de trouver le plus proche voisin, tous les processus vont lire le même espace mémoire. Lorsqu'un d'entre eux a fini et veut ajouter un nouveau nœud à l'arbre, il prend le sémaphore, ajoute son nœud et le rend aussitôt. La procédure d'ajout d'un nœud dans l'arbre se résumant à la création d'un objet par son constructeur, le temps pendant lequel les autres processus sont en attente est faible.

Il se peut cependant qu'un des processus soit en train de lire l'arbre pendant qu'un autre ajoute un nœud. Dans ce cas, suivant la structure de données utilisée et le sens de parcours de cette structure (par exemple, si l'on prend une structure de type arbre, elle peut être parcourue en largeur, en profondeur ou bien encore de façon chaotique), il se peut que le nouveau nœud ne soit pas pris en compte dans la recherche du plus proche voisin. Ce problème n'en est pas vraiment un puisque l'arbre s'étend tout de même. Cependant, si l'on veut avoir strictement le même fonctionnement que pour un RRT classique, il suffit d'utiliser une structure de données particulière. En effet, si les nœuds de l'arbre sont stockés dans une file (FIFO), le processus examinant les nœuds devra passer par le dernier nœud, celui qui vient d'être ajouté. Cette structure permet d'augmenter les chances que les nœuds nouvellement créés soient pris en compte mais ne le garantit tout de même pas (puisque lorsque le constructeur est exécuté, le processus continu de parcourir l'arbre et peut très bien finir ce parcours avant la fin du constructeur), surtout lorsque l'arbre ne contient que peu de nœuds.

Les exclusions mutuelles doivent être placées juste avant l'accès en écriture dans la structure arbre. Autrement dit, lorsqu'un nœud est ajouté dans l'arbre par un processus, il faut qu'aucun autre processus ne soit déjà en train d'écrire dedans. Il en résulterait qu'un des nœuds ne serait finalement pas dans l'arbre à la fin. Les mutex sont placés juste avant l'ajout d'un nouveau nœud comme indiqué sur l'algorithme 18. Les primitives de prise (P pour prendre) et de libération (V pour vendre) de la variable d'exclusion mutuelle sont fidèles à celles définies par Dijkstra.

Les tests portent sur la génération de 40 000 nœuds dans un environnement clos contenant deux

Algorithme 18 Parallelisation du Rrt

```
1: Mutex ← 0
2: ...
3: Pour i = 0 à K faire
4:    x_rand ← RandomState() ;
5:    x_near ← NearestNeighbor(G, x_rand )
6:    u ← SelectInput(x_rand, x_near )
7:    x_new ← NewState(x_near, u, Δt )
8:    P(Mutex)
9:    G.AddNode(x_new)
10:   G.AddEdge(x_near, x_new, u)
11:   V(Mutex)
12: FinPour
13: ...
```

temps de calcul	nombre de processus	temps de calcul/nombre de processus	rendement
207.4	1	207	
211.5	2	105.8	0.98
217.9	4	54.5	0.95

TAB. 5.1 – Temps de calcul de 40 000 nœuds en fonction du nombre de processeurs

obstacles placés en T au milieu de la carte. Ces résultats sont des moyennes sur 100 calculs et ont été obtenus sur une lame JS21 comportant quatre processeurs Power PC. Les résultats obtenus dans la table 5.1 sont très bons puisqu'ils montrent que le temps perdu à cause des exclusions mutuelles est très faible, le rendement étant très proche de l'unité.

5.3.2 Test de collision

Lors de la présentation du RRT pour un système quelconque, il a été dit qu'un état était ajouté à l'arbre G s'il appartenait à l'ensemble $\mathbb{X}_{\text{free}}$. Là où ce test était évident pour un système ponctuel puisque, dans ce cas, l'ensemble des obstacles de l'environnement est confondu avec l'ensemble $\mathbb{X}_{\text{obs}}$, il doit être redéfini dans le cas de notre véhicule.

Comme celui-ci n'est pas ponctuel, le fait de tester si son état actuel $\mathbf{x}$ ne prend pas des valeurs qui le placent sur un obstacle ne suffit pas à garantir qu'il n'y ait pas de collision entre le véhicule et un mur de l'environnement.

La composante en x et en y du vecteur d'état $\mathbf{x}$ représente en effet la position géométrique du point de référence du véhicule dans le repère de référence $\mathcal{W}$ de l'environnement.

Si le véhicule est représenté par un polygone P_v, il faut, après avoir calculé les coordonnées de ce polygone dans $\mathcal{W}$ tester qu'aucun des segments de ce polygone n'ait de point d'intersection avec un des segments des polygones représentant les obstacles de l'environnement. Il faut également tester qu'aucun des segments de l'environnement ne soit entièrement contenu dans P_v. En effet, le saut dû à l'intégration du système différentiel sur un intervalle de temps Δt peut faire qu'un obstacle soit entièrement contenu dans le polygone représentant le véhicule sans qu'il n'ait de point d'intersection.

5.3.3 Pseudo-métrique

Une pseudo-métrique est nécessaire afin calculer quel nœud est associé à un état qui est le plus proche de $\mathbf{x}_{\text{rand}}$. De nombreuses options sont possibles.

Si l'on utilise une distance euclidienne dans $\mathbb{R}^3$, le principal problème vient du fait que l'angle est alors considéré au même titre que la position sur le plan $\{x, y\}$.

Nous préférerons retenir la pseudo-métrique formée par la somme de la distance euclidienne entre les composantes x et y des vecteurs d'état afin de représenter la distance entre les deux points de référence des deux véhicules et de la longueur de l'arc de cercle permettant de passer d'une orientation à l'autre qui n'est autre que la longueur L du véhicule multipliée par la valeur absolue de la différence entre les orientations. Ainsi, l'une des pseudo-métriques utilisables pour notre modèle de *Voiture simple* afin de calculer la distance entre $\mathbf{x}_1 = (x_1, y_1, \theta_1)^{\mathrm{T}}$ et $\mathbf{x}_2 = (x_2, y_2, \theta_2)^{\mathrm{T}}$ est

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + L|\theta_1 - \theta_2| \tag{5.16}$$

5.3.4 Choix de la commande

Comme pour tout système à déplacer, la commande dans la fonction `SelectInput` de l'algorithme 8 page 38 doit être définie afin que les branches de l'arbre s'étendent correctement et rapidement au sein de l'environnement. Comme pour le cas holonome, il est tout à fait possible d'utiliser une des commandes disponibles au hasard (figure 5.3). Dans ce cas, les branches de l'arbre s'étendent rapidement dans des directions mais ne sont pas très étoffées.

Dans la suite de ce manuscrit, sauf indication contraire, nous préférerons ici minimiser la distance (5.16) entre $\mathbf{x}_{\mathrm{near}}$ et $\mathbf{x}_{\mathrm{rand}}$ afin de nous rapprocher le plus possible de l'état aléatoire choisi. Comme nous pouvons le voir sur la figure 5.4, l'arbre s'étoffe rapidement ce qui permet de couvrir plus uniformément l'environnement, $\mathbb{X}_{\mathrm{goal}}$ tendra donc à être atteint plus rapidement.

5.3.5 Exemple de trajectoires planifiées

Nous présentons dans cette section des résultats issus de la planification de trajectoire en utilisant le RRT et le modèle de *voiture simple*.

5.3.5.1 Environnement avec peu d'obstacles

Regardons tout d'abord le cas d'un véhicule dans un environnement avec peu d'obstacles afin de bien voir la progression des différentes branches de l'arbre.

Les figures 5.5 à 5.8 montrent la progression d'un arbre partant du point indiqué par le véhicule et s'étendant sur tout le plan. La figure 5.5 représente l'arbre après l'ajout de 150 nœuds, la figure 5.6 après l'ajout de 1000 nœuds, la figure 5.7 après l'ajout de 2000 nœuds et finalement une fois que 4000 nœuds ont été ajoutés, on obtient la figure 5.8.

Comme le RRT mis en œuvre est ici la version de base, aucune aide n'est donnée au graphe afin qu'il rejoigne l'arrivée. L'algorithme finit lorsque le hasard fait qu'un des nœuds est créé dans $\mathbb{X}_{\mathrm{goal}}$. Il faut aussi noter que l'entrée $\mathbf{u}$ de ce mobile l'empêche d'aller en marche arrière et que son rayon de braquage est relativement faible par rapport à sa taille. On peut d'ailleurs voir à plusieurs endroits du graphe apparaître le rayon de courbure maximal (celui correspondant au δ_{max} défini par $\mathbb{U}$).

Lors des tests réalisés en utilisant cette méthode, on remarque que la vitesse à laquelle le graphe se rapproche de $\mathbb{X}_{\mathrm{goal}}$ dépend très fortement de l'environnement, des contraintes que l'on donne au mobile au niveau de $\mathbb{U}$ ainsi que de la position de $\mathbf{x}_{\mathrm{init}}$ et $\mathbb{X}_{\mathrm{goal}}$. Le temps pour rejoindre l'arrivée (exprimé en nombre de nœuds du graphe) n'est pas du tout constant, même pour deux situations identiques, du fait de l'influence du germe initial de la fonction de génération de nombres aléatoires.

Toutes les branches de l'arbre, même celles qui ne sont pas utilisées dans la trajectoire finale, ont été laissées pour permettre une meilleur compréhension du fonctionnement de l'algorithme.

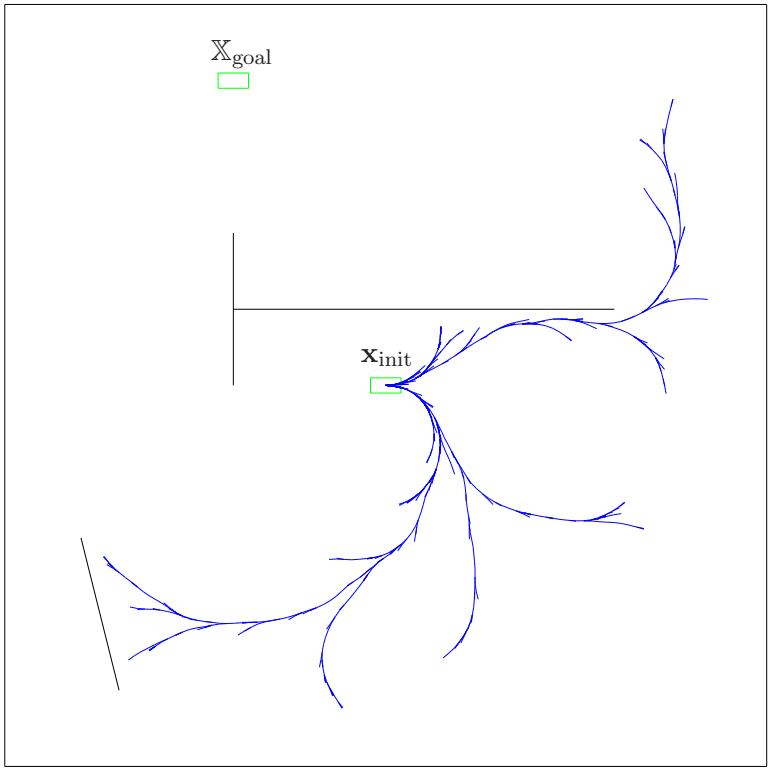


FIG. 5.3 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 1000 nœuds et choix de la commande aléatoire

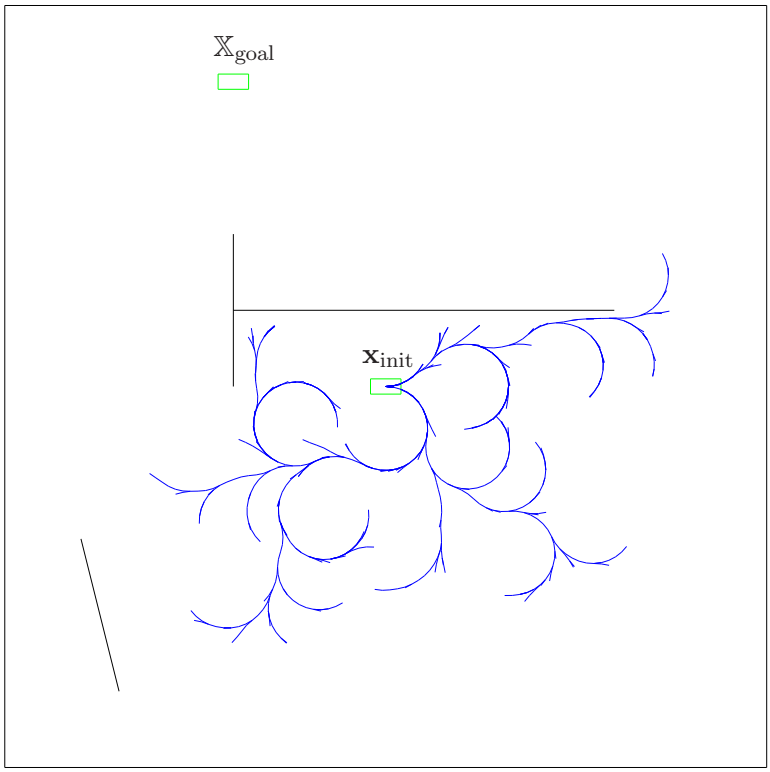


FIG. 5.4 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 1000 nœuds et choix de la commande minimisant la distance entre $\mathbf{x}_{\text{near}}$ et $\mathbf{x}_{\text{rand}}$

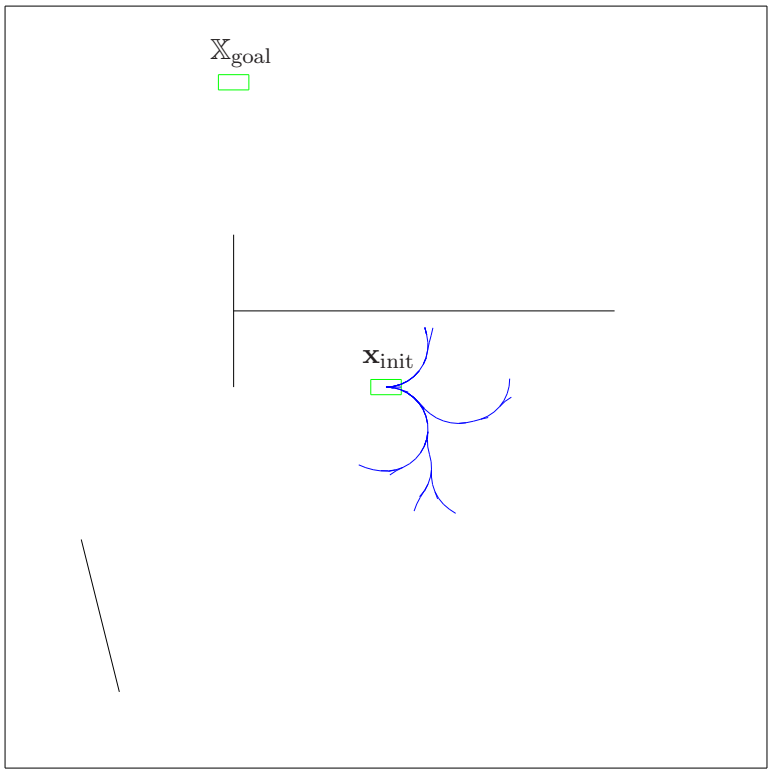


FIG. 5.5 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 150 nœuds

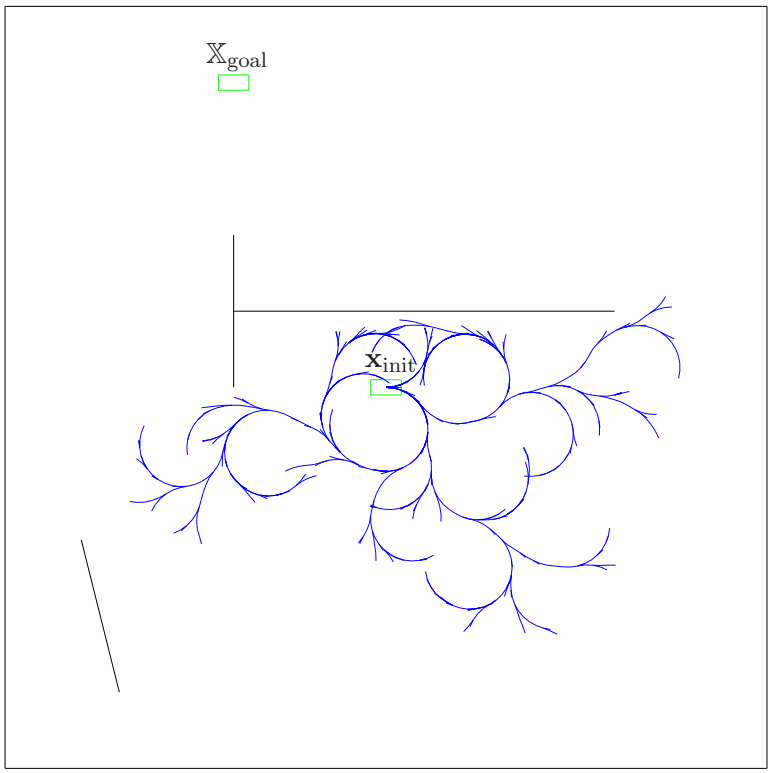


FIG. 5.6 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 1000 nœuds

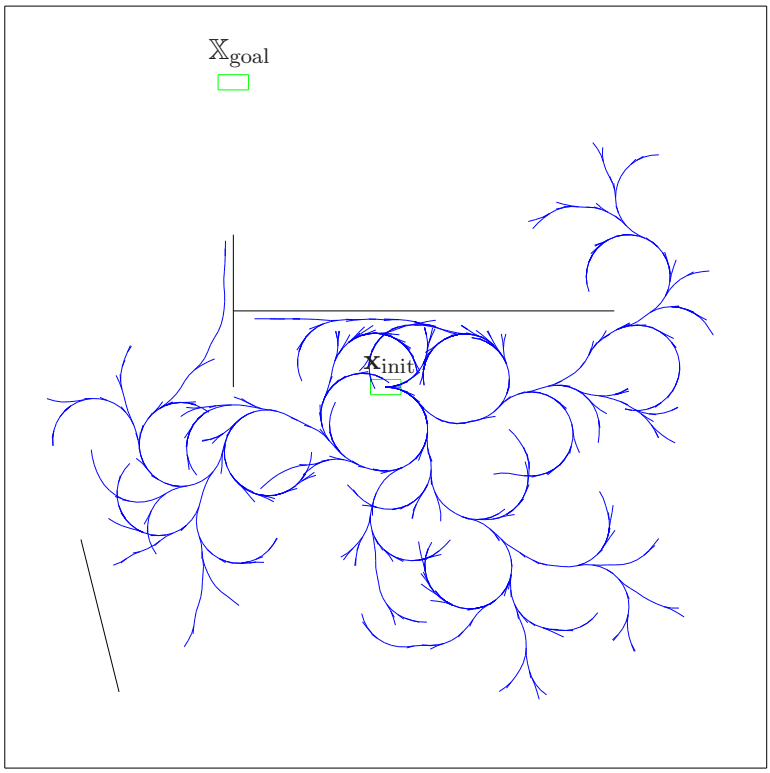


FIG. 5.7 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 2000 nœuds

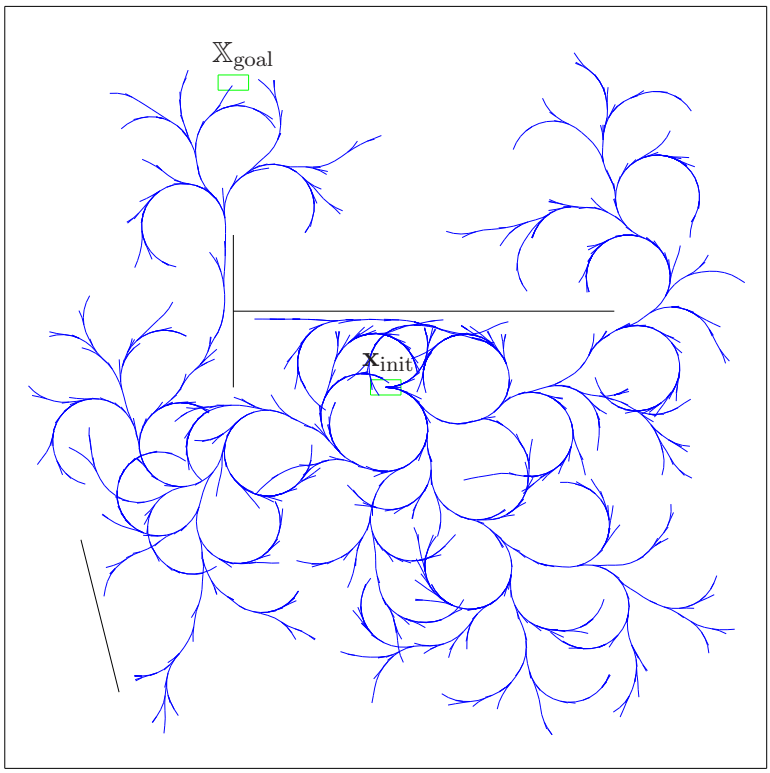


FIG. 5.8 – Génération de l'arbre pour une *voiture simple* non ponctuelle après ajout de 4000 nœuds

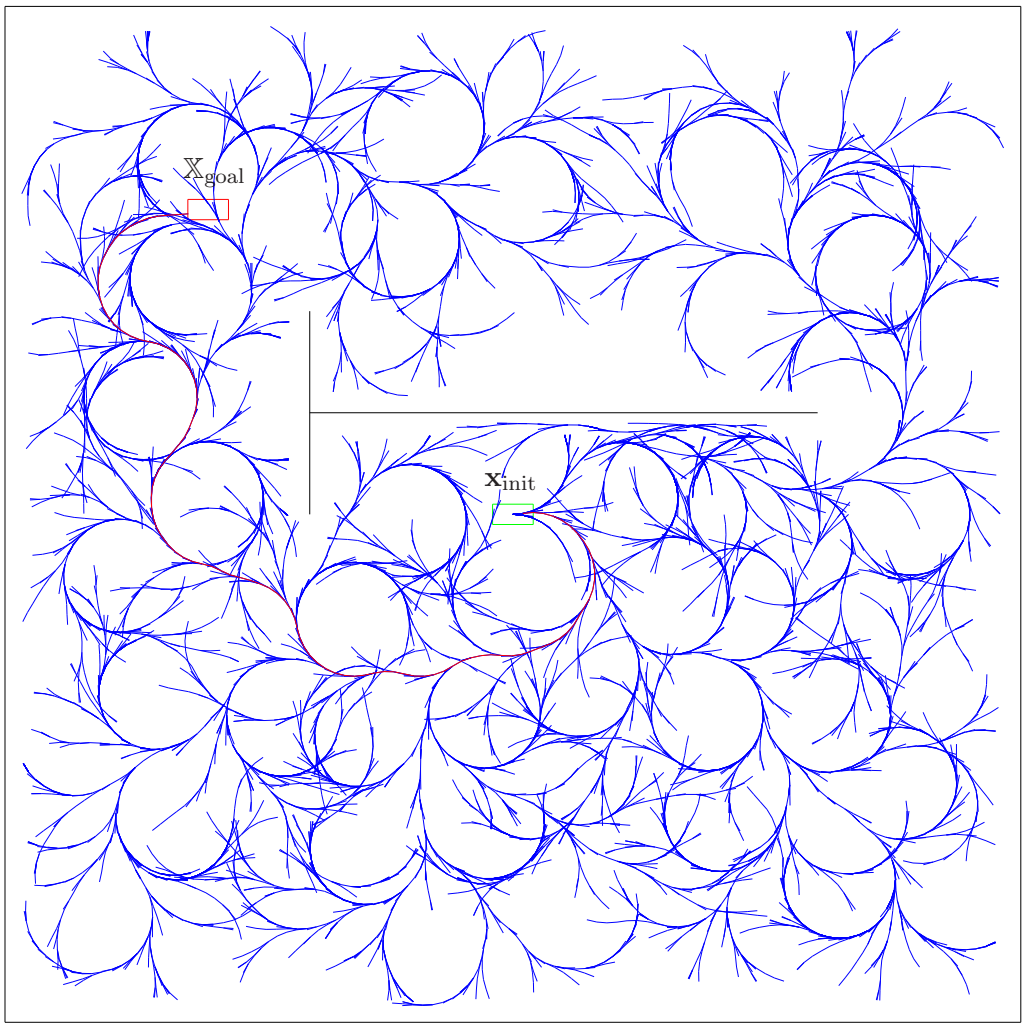


FIG. 5.9 – RRT de base connecté pour une *voiture simple* non ponctuelle

Sur la figure 5.9, la construction de cette trajectoire a nécessité la création de plus de 3000 nœuds. Le temps de calcul est de l'ordre de la seconde. Bien que le temps ne soit pas la priorité principale lors d'une étape de planification (la phase de planification précédant le démarrage du véhicule), il arrive que le temps nécessaire à la recherche d'une trajectoire pour un problème donné varie d'un facteur trois entre deux exécutions du planificateur. L'utilisation des primitives *Goalbias* et *Goalzoom* peut donc s'avérer utile pour diminuer le nombre de nœuds nécessaires et donc le temps de calcul (table 5.2). La figure 5.10 montre le type de résultat obtenu lors de l'utilisation du *Goalbias* avec une probabilité de 0.1 de choisir $\mathbf{x}_{\text{goal}}$ comme $\mathbf{x}_{\text{rand}}$.

5.3.5.2 Environnement avec de nombreux obstacles

Afin d'illustrer le cas où l'environnement comporte de nombreux obstacles, prenons une carte ressemblant à un labyrinthe (figure 5.11).

La pertinence des résultats et surtout leur vitesse de génération ne penche plus dans le sens du RRT-*Goalbias* dans ce cas. L'explication vient du fait que lorsque $\mathbf{x}_{\text{init}}$ et $\mathbb{X}_{\text{goal}}$ sont séparés par un mur, l'algorithme RRT-*Goalbias* a tendance à trouver la distance minimale en un point qui se trouve de l'autre coté du mur. La création du point est alors annulée puisque celui-ci ne respecte pas les contraintes de l'environnement : le véhicule heurterait le mur. On voit alors un amas de points sur chacun des murs, du coté opposé à celui où se trouve $\mathbb{X}_{\text{goal}}$. Plus la probabilité p sera élevée et plus le graphe aura tendance à stagner derrière le mur. Même avec $p = 0.1$, soit une valeur relativement faible, les résultats ne sont pas probant (figure 5.12), le graphe ayant énormément de mal à rejoindre sa destination. Le nombre de nœuds qu'il a fallu générer pour atteindre $\mathbb{X}_{\text{goal}}$ est en moyenne d'environ 10 610.

Au contraire, dans ce type d'environnement, le caractère totalement aléatoire du RRT fait que la façon qu'il a de se déployer sans prendre en compte l'arrivée le rend insensible au fait qu'un mur sépare deux états. La trajectoire souhaitée est alors plus rapide à obtenir qu'avec *Goalbias*. On peut ainsi voir sur la figure 5.13 le résultat obtenu pour le même environnement que précédemment. En comparant les figures 5.12 et 5.13, on remarque que le deuxième schéma est nettement moins chargé en points que le premier, surtout au niveau des murs séparant le départ de l'arrivée. Il ne faut en effet en moyenne qu'environ 6720 nœuds pour générer une trajectoire avec l'algorithme RRT dans un tel environnement.

La mise en œuvre du RRT-*Goalzoom* a nécessité plusieurs modifications afin de donner les résultats escomptés. En effet, la méthode décrite par J. Kuffner ne permettait pas, telle quelle, d'améliorer de façon significative les performances du RRT. On se rend vite compte que, même en dirigeant les branches de l'arbre, la durée du calcul peut être très longue à cause de l'orientation du mobile. Le graphe se rend en effet rapidement sur le lieu d'arrivée mais pas forcement avec l'orientation désirée. Le temps gagné sur la vitesse de déplacement vers $\mathbb{X}_{\text{goal}}$ pourrait être perdu si le mobile n'a pas un θ qui appartient à $\mathbb{X}_{\text{goal}}$.

Pour réduire l'impact de ce phénomène, le point tiré aléatoirement dans le cercle de centre $\mathbf{x}_{\text{goal}}$ et de rayon $\min_i d(\mathbf{x}_i, \mathbf{x}_{\text{goal}})$ prendra pour θ une valeur proche de l'orientation de $\mathbf{x}_{\text{goal}}$. On peut ainsi tirer le θ aléatoirement dans l'intervalle $[\theta - \epsilon, \theta + \epsilon]$, où ϵ peut être choisi en fonction de la largeur de l'intervalle $[\theta]$ de $\mathbb{X}_{\text{goal}}$ par exemple.

Une autre solution aurait été de tirer les points aléatoirement dans un cône de rayon $\min_i d(\mathbf{x}_i, \mathbf{x}_{\text{goal}})$ et d'ouverture ϵ, en orientant le cône avec le θ de $\mathbf{x}_{goal}$. Cette solution aurait permis de diriger correctement les branches de l'arbre, mais les performances auraient été similaires à celles obtenues avec le RRT-*Goalbias*, ce qui n'est nullement notre but.

Le RRT-*Goalzoom* nous permet d'obtenir des résultats qui sont proches de ceux du RRT-*Goalbias* avec une certaine souplesse qui manque au RRT-*Goalbias* lorsque l'environnement contient de nom-

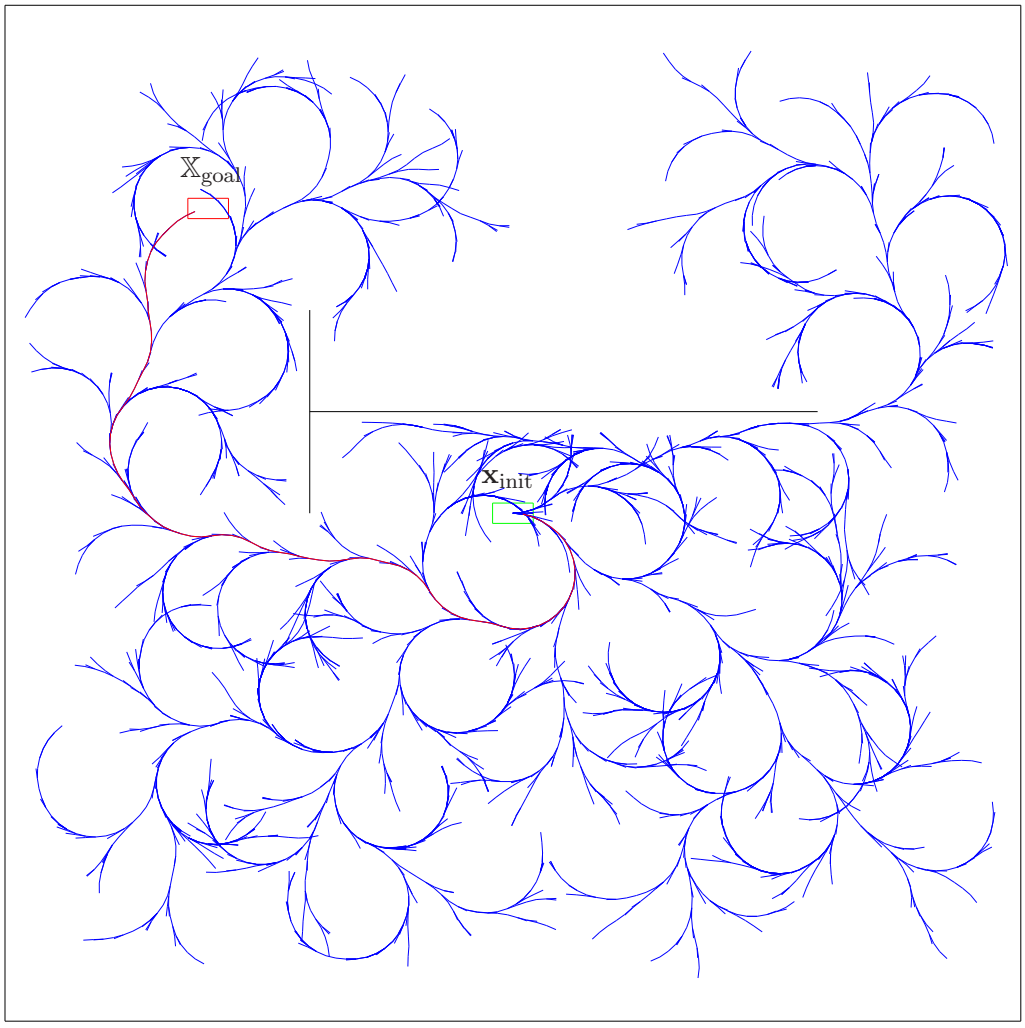


FIG. 5.10 – RRT-*Goalbias*

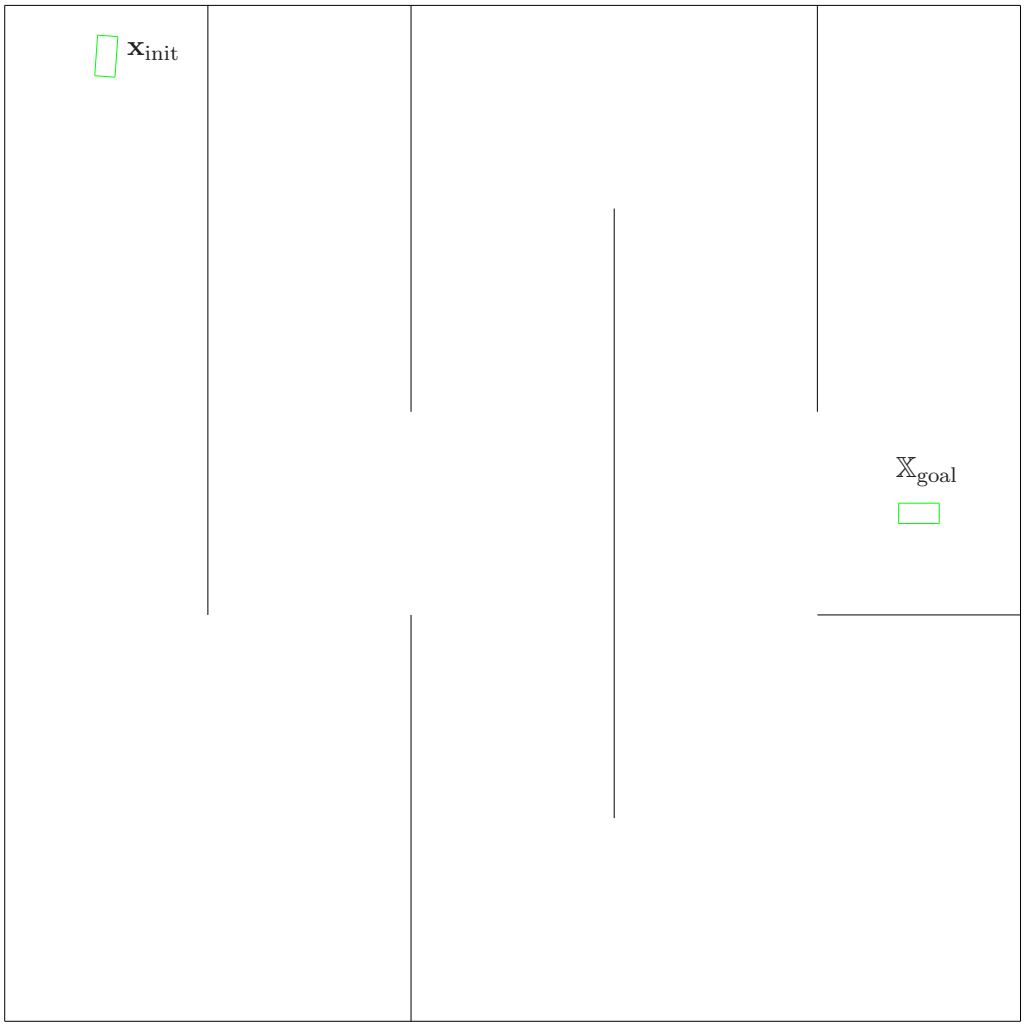


FIG. 5.11 – Environnement en forme de labyrinthe pour une *voiture simple* non ponctuelle

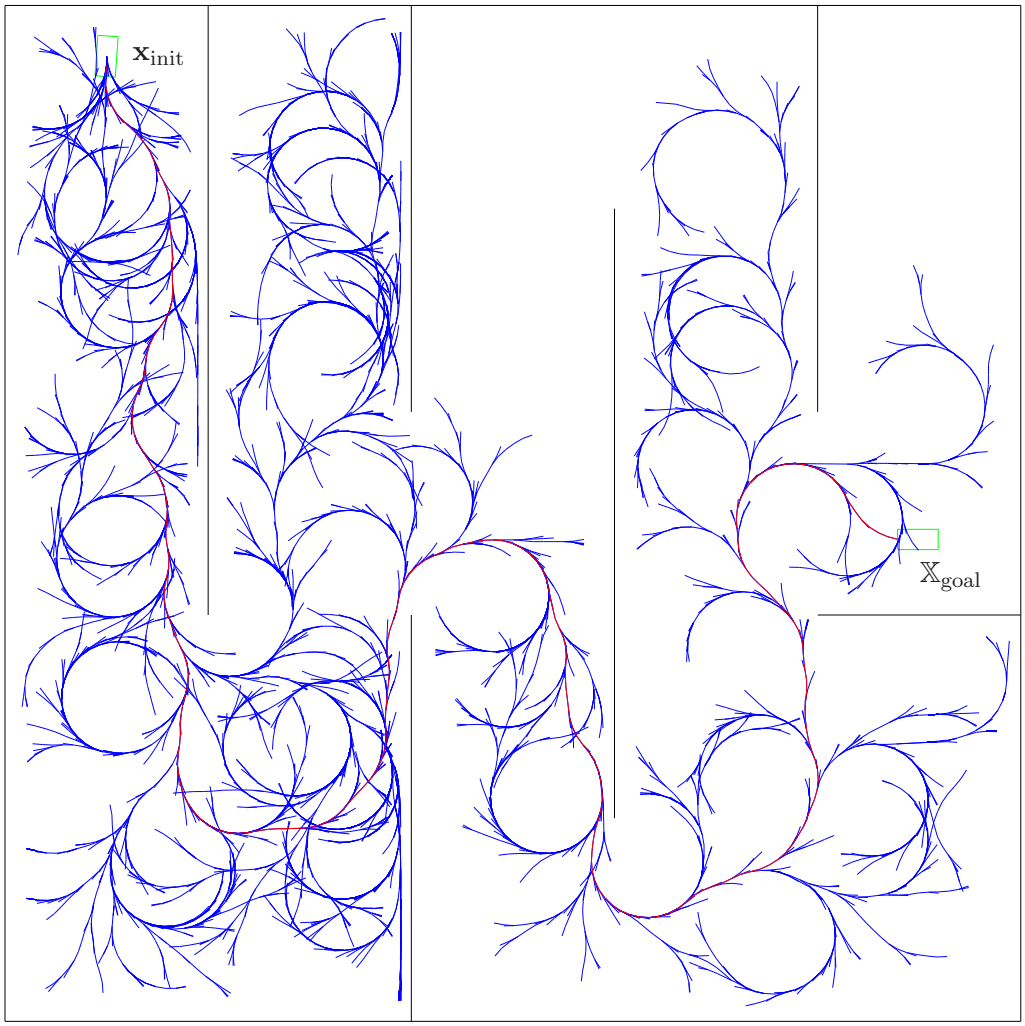


FIG. 5.12 – RRT-*Goalbias* en environnement contenant de nombreux obstacles pour une *voiture simple* non ponctuelle

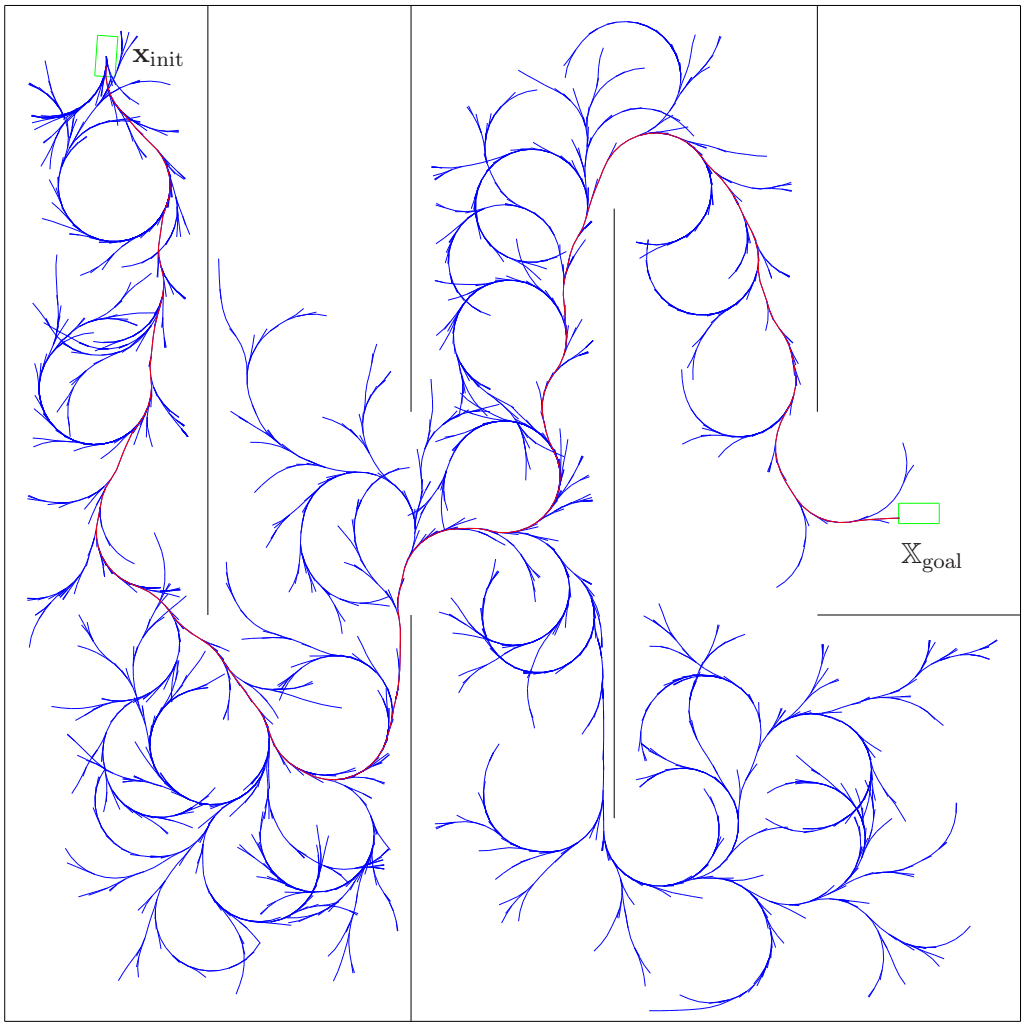


FIG. 5.13 – RRT en environnement contenant de nombreux obstacles pour une *voiture simple* non ponctuelle

breux obstacles (voir la figure 5.14). La génération de trajectoires dans ce type d'environnement prend un peu moins de temps qu'avec *Goalbias* puisqu'il ne faut en moyenne plus que 6150 nœuds pour trouver une trajectoire.

Nous pouvons ainsi récapituler le nombre de nœuds nécessaires à la construction d'une trajectoire dans divers environnements (tableau 5.2). Les tests ont été effectués sur un ensemble de 100 tentatives. Deux types d'environnement ont été considérés, un environnement vide et l'environnement en forme de labyrinthe.

	Environnement	
	Vide	Labyrinthe
RRT	3480 (687)	6720 (1067)
RRT-*Goalbias* ($p = 0.1$)	1170 (198)	10610 (1817)
RRT-*Goalzoom* ($p = 0.1$)	1910 (207)	6150 (612)

TAB. 5.2 – Nombres moyen (entre parenthèse l'écart type) de nœuds à générer lors de l'utilisation des différents RRT dans deux environnements différents

5.4 Modèles plus complexes

Nous allons maintenant présenter ici des modèles cinématiques un peu plus complexes que *voiture simple* et qui respectent mieux les contraintes sur le déplacement d'un véhicule engendrées par sa mécanique. Nous présenterons également un modèle dynamique que nous avons intégré au planificateur.

5.4.1 Modèle à angle de braquage continu

Dans *voiture simple*, l'orientation du véhicule θ varie de manière continue mais ce n'est nullement le cas de l'angle de braquage. Ainsi, dans ce genre de modèle, on considère que l'angle de braquage peut varier instantanément, ce qui n'est pas le cas dans la réalité.

Voiture simple peut être facilement amélioré en faisant varier non plus l'angle de braquage θ mais la dérivée de l'angle de braquage $\dot{\theta}$, c'est-à-dire la vitesse de braquage. On obtient alors un modèle [60] dans un espace d'état à quatre dimensions

$$\dot{x} = \cos\theta \tag{5.17}$$

$$\dot{y} = \sin\theta \tag{5.18}$$

$$\dot{\theta} = \frac{v}{l}\tan\delta \tag{5.19}$$

$$\dot{\delta} = \omega \tag{5.20}$$

La commande n'est dès lors plus l'angle de braquage θ mais la vitesse angulaire de braquage ω. Un ensemble de trajectoires générées en utilisant ce modèle est représenté sur la figure 5.15.

5.4.2 Modèle à vitesse de braquage continue

Un modèle encore plus réaliste consiste à considérer une fonction décrivant l'évolution de l'angle de braquage de classe C^1 en fonction du temps. La commande ne sera donc plus la vitesse angulaire

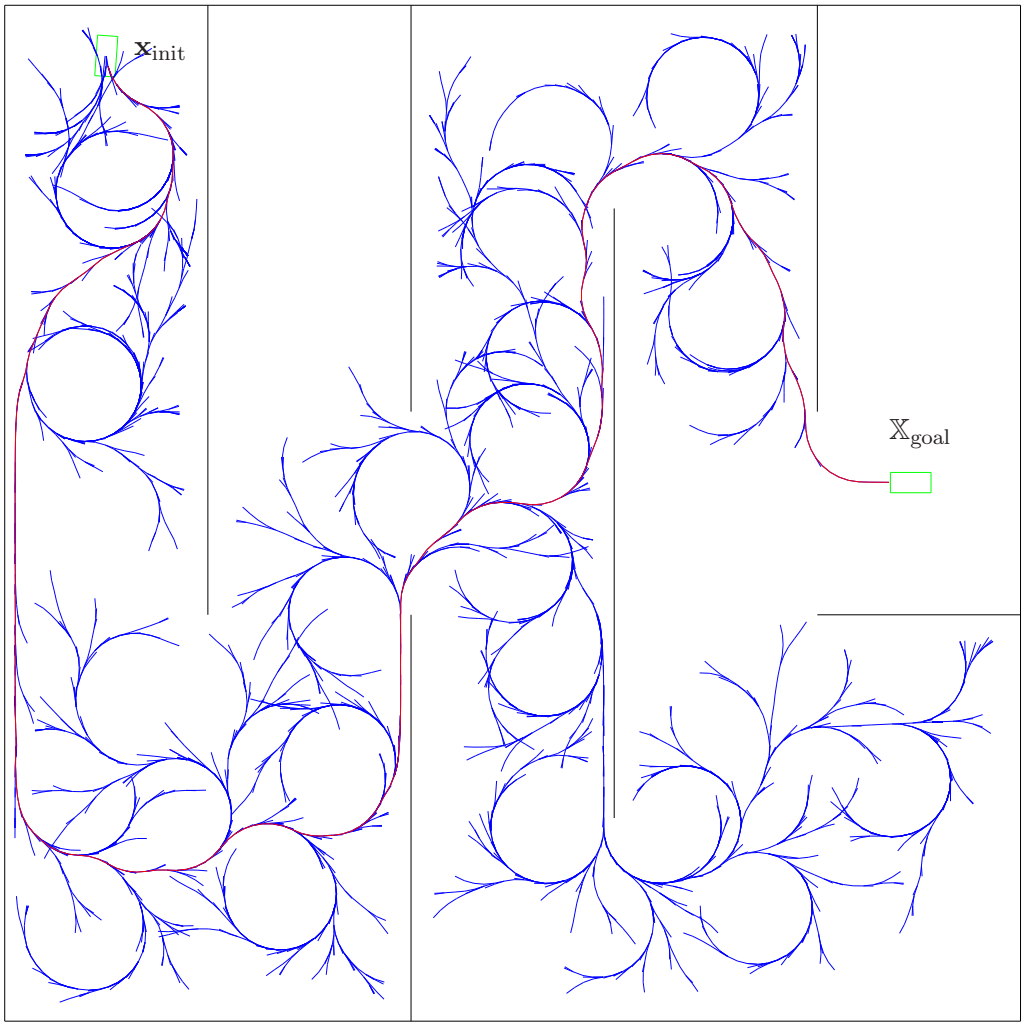


FIG. 5.14 – RRT-*Goalzoom* en environnement contenant de nombreux obstacles

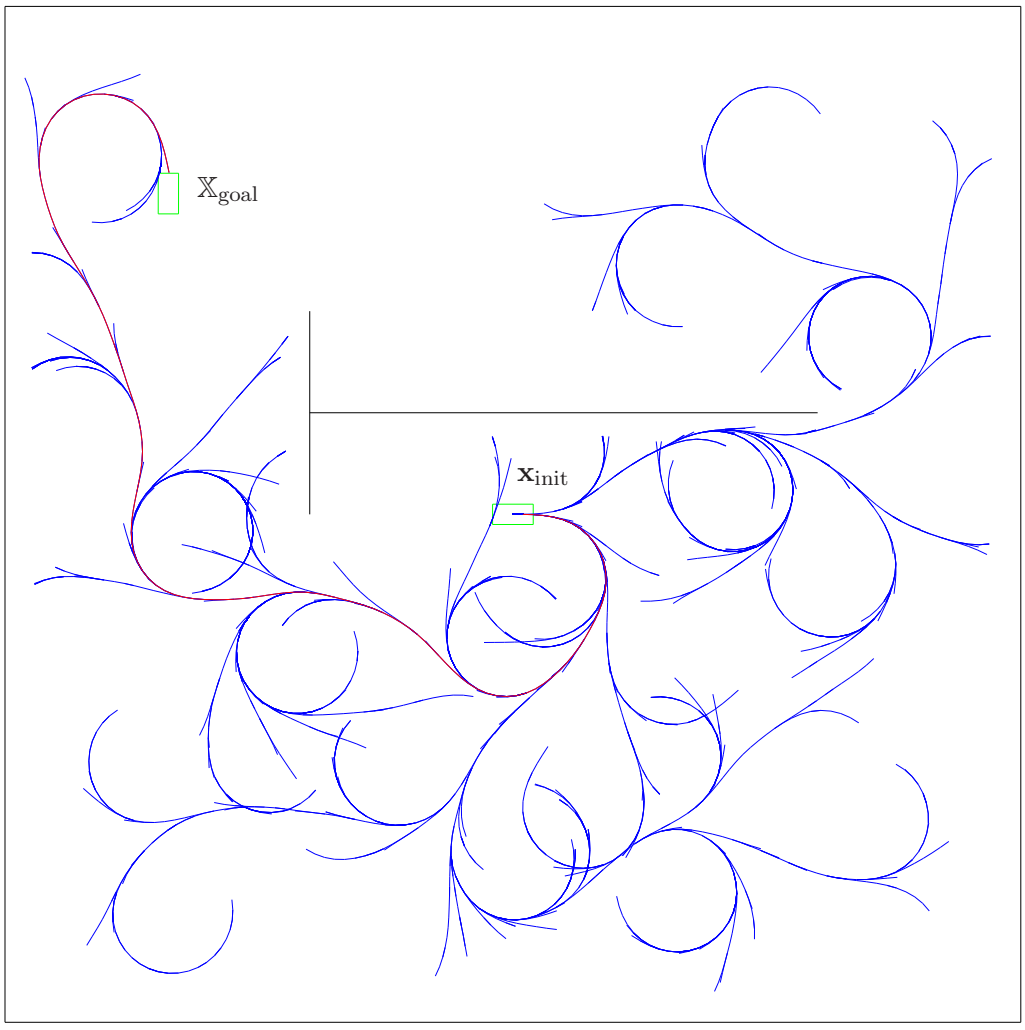


FIG. 5.15 – Modèle à angle de braquage continu

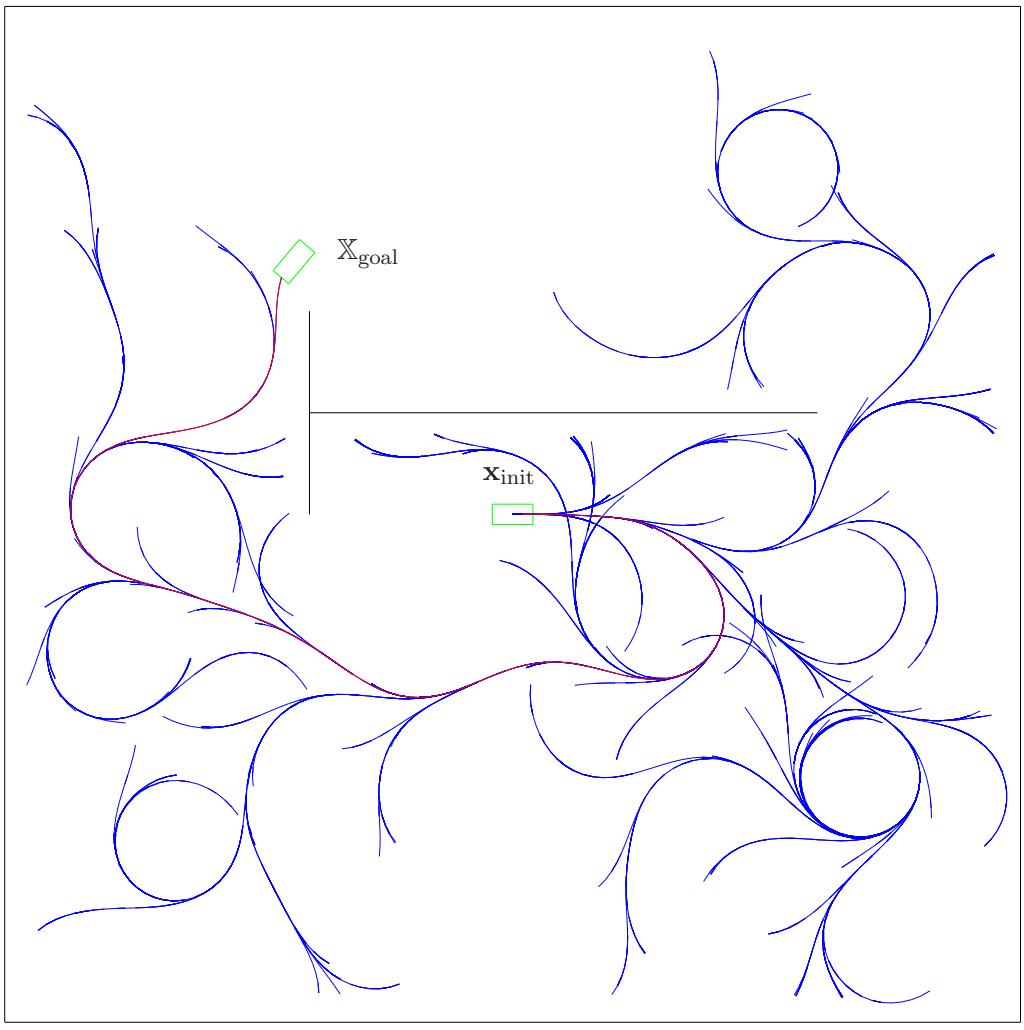


FIG. 5.16 – Modèle à vitesse de braquage continue

FIG. 5.17 – Picar

de braquage ω mais l'accélération angulaire de braquage α. Le modèle est dès lors le suivant :

$$\begin{aligned} \dot{x} &= \cos\theta \\ \dot{y} &= \sin\theta \\ \dot{\theta} &= \frac{v}{l}\tan\delta \\ \dot{\delta} &= \omega \\ \dot{\omega} &= \alpha \end{aligned} \tag{5.21}$$

l'espace d'état étant maintenant de dimension 5. On obtient un ensemble de trajectoires représentées sur la figure 5.16 encore plus lisses que celles de la figure 5.15.

5.4.3 Modèle avec deux trains orientables

Pour générer des trajectoires un peu plus précises qui pourraient être utilisées sur la plate-forme expérimentale de l'Institut d'Électronique Fondamentale, nous allons présenter le modèle cinématique de la plate-forme Picar. Le modèle que nous utilisons est celui décrit dans le rapport technique sur la plate-forme expérimentale Cycab de L'INRIA [61]. La différence majeure entre ce type de modèle et le modèle de type voiture est la présence de deux paires de roues directrices.

Picar [62] est une plate-forme expérimentale pour les applications routières. Ce véhicule est équipé de capteurs proprioceptifs et extéroceptifs. Ses roues arrières sont, tout comme ses roues avants, directrices. L'angle de braquage arrière est lié à l'angle de braquage avant par l'intermédiaire d'un coefficient k. Ainsi, si l'angle de braquage des roues avant est δ_{av}, celui des roues arrières sera $\delta_{\text{ar}} = k\delta_{\text{av}}$.

La valeur de k, qui peut être considérée comme constante, est déterminée expérimentalement.

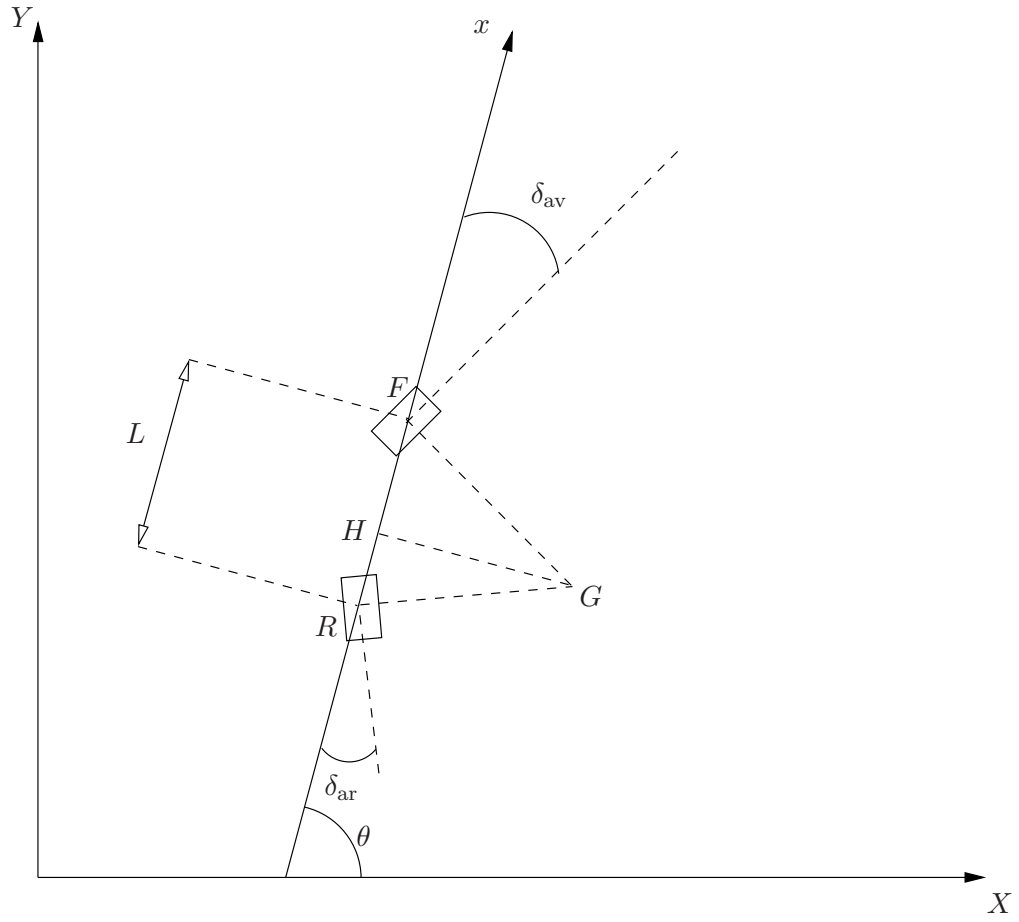


FIG. 5.18 – Picar kinematic model

Nous reprenons ici la même valeur que dans [61]

$$k = 0.69 \tag{5.22}$$

lorsque l'on considère que δ_{av} appartient à l'intervalle $[-0.4; 0.4]$ radian.

Le modèle *voiture simple* peut être adapté au cas où les deux trains de roues sont directeurs (figure 5.18). Pour le simplifier, nous allons projeter les roues avant et arrière sur une roue virtuelles située au milieu de chacun des essieux. On note G l'intersection des deux droites perpendiculaires à la direction des roues et passant par leurs centres. On note H la projection orthogonale de G sur l'axe longitudinal de la voiture.

La longueur GH est définie par

$$GH = \frac{RH}{|\tan(k\delta_{av})|} = \frac{HF}{|\tan(\delta_{av})|} \tag{5.23}$$

Ainsi, on obtient

$$RH = HF * \frac{\tan(k\delta_{av})}{\tan(\delta_{av})}. \tag{5.24}$$

Comme

$$L = RH + HF, \tag{5.25}$$

on a

$$\begin{cases} RH &= \frac{L}{1+\frac{\tan(\delta_{\text{av}})}{\tan(k\delta_{\text{av}})}} = L\frac{\cos(\delta_{\text{av}})\sin(k\delta_{\text{av}})}{\sin(\delta_{\text{av}}+k\delta_{\text{av}})} \\ HF &= \frac{L}{1+\frac{\tan(k\delta_{\text{av}})}{\tan(\delta_{\text{av}})}} = L\frac{\cos(k\delta_{\text{av}})\sin(\delta_{\text{av}})}{\sin(\delta_{\text{av}}+k\delta_{\text{av}})}. \end{cases} \tag{5.26}$$

Les rayons de braquage avant ρ_{av} et ρ_{ar} sont

$$\begin{cases} \rho_r &= \frac{RH}{|\sin(k\delta_{\text{av}})|} = L\frac{\cos(\delta_{\text{av}})}{|\sin(\delta_{\text{av}}+k\delta_{\text{av}})|} \\ \rho_f &= \frac{HF}{|\sin(\delta_{\text{av}})|} = L\frac{\cos(k\delta_{\text{av}})}{|\sin(\delta_{\text{av}}+k\delta_{\text{av}})|}. \end{cases} \tag{5.27}$$

La vitesse de rotation $\dot{\theta}$ est

$$\dot{\theta} = \frac{v_{\text{ar}}}{\rho_{\text{ar}}} = \frac{v_{\text{av}}}{\rho_{\text{av}}} \tag{5.28}$$

En utilisant (5.27) et (5.28), on obtient

$$v_{\text{ar}} = v_{\text{av}}\frac{\rho_{\text{ar}}}{\rho_{\text{av}}} = v_{\text{av}}\frac{\cos\delta_{\text{av}}}{\cos k\delta_{\text{av}}}. \tag{5.29}$$

Ainsi, les équations du mouvement de R, obtenues géométriquement, sont

$$\begin{cases} \dot{x}_{\text{ar}} &= v_{\text{ar}}.\cos(\theta + k\delta_{\text{av}}) \\ \dot{y}_{\text{ar}} &= v_{\text{ar}}.\sin(\theta + k\delta_{\text{av}}) \\ \dot{\theta}_{\text{ar}} &= v_{\text{ar}}\frac{\sin(\delta_{\text{av}}+k\delta_{\text{av}})}{L\cos\delta_{\text{av}}} \end{cases} \tag{5.30}$$

et celles du mouvement de F sont

$$\begin{cases} \dot{x}_{\text{av}} &= v_{\text{av}}.\cos(\theta + \delta_{\text{av}}) \\ \dot{y}_{\text{av}} &= v_{\text{av}}.\sin(\theta + \delta_{\text{av}}) \\ \dot{\theta}_{\text{av}} &= v_{\text{av}}\frac{\sin(\delta_{\text{av}}+k\delta_{\text{av}})}{L\cos\delta_{\text{av}}} \end{cases} \tag{5.31}$$

Ce modèle décrit le mouvement du véhicule lorsque nous considérons uniquement un aspect géométrique. Bien qu'il soit plus adapté à Picar que le précédent, il n'est pas encore vraiment satisfaisant puisqu'il prend comme principe le mouvement sans glissement du véhicule. Ainsi, ce type de modèle ne peut être utilisé que lorsque l'on peut s'assurer que le véhicule se déplace à une vitesse faible. En effet, lorsque les vitesses deviennent plus élevées, il faut prendre en compte la dynamique du véhicule.

Ainsi, utiliser ce type de modèle au sein du planificateur va générer un nombre important d'approximations. Le véhicule ne pourra pas suivre la trajectoire planifiée et même s'il y arrivait par l'utilisation d'une loi de commande adaptée, il n'est pas garanti que les zones qu'il visite afin de rejoindre la trajectoire planifiée soient sans obstacle. L'utilisation d'un tel modèle de véhicule peut donc générer un nombre important de collisions durant la navigation.

Afin de réduire le risque de collision et également de diminuer la correction nécessaire durant la navigation, nous devons utiliser un modèle qui tient compte de la dynamique du véhicule.

5.4.4 Modèle dynamique

Nous présentons maintenant un modèle dynamique de Picar à cinq degrés de liberté. Comme pour le modèle cinématique présenté auparavant, nous utilisons un modèle de type bicyclette (figure 5.19). Ce modèle à cinq degrés de liberté est inspiré de [63]. L'état du véhicule est défini par le vecteur $\mathbf{x} = (x_g, y_g, \theta, v_y, r)^T$; x_g et y_g représentent les coordonnées du centre de gravité du robot, θ est toujours l'orientation du véhicule, v_y est sa vitesse latérale et r est le taux de lacet.

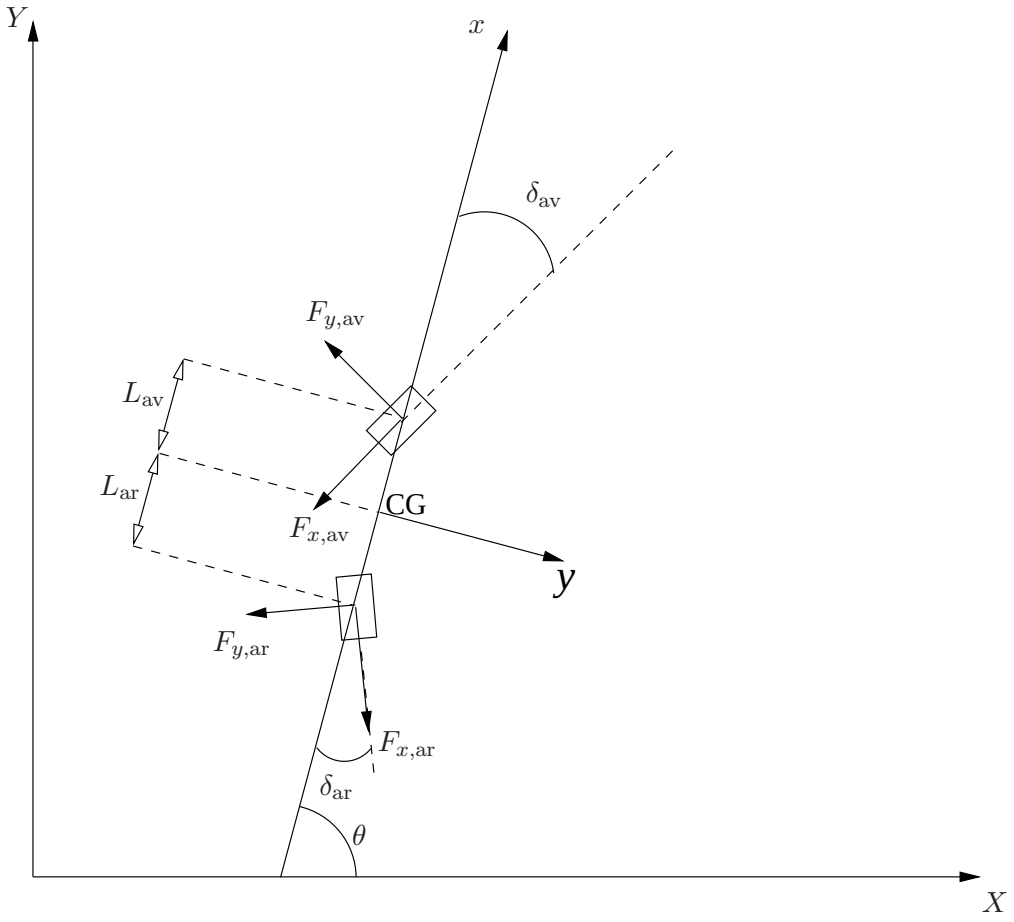


FIG. 5.19 – Simplified dynamic model

En utilisant la seconde loi de Newton et le principe fondamental de la dynamique, nous obtenons :

$$\begin{aligned} m(\dot{v}_x - v_y r) &= -F_{x,\mathrm{av}} \cos\delta_{\mathrm{av}} + F_{y,\mathrm{av}} \sin\delta_{\mathrm{av}} - F_{x,\mathrm{ar}} \cos\delta_{\mathrm{ar}} - F_{y,\mathrm{ar}} \sin\delta_{\mathrm{ar}} \\ m(\dot{v}_y + v_x r) &= F_{y,\mathrm{av}} \cos\delta_{\mathrm{av}} + F_{x,\mathrm{av}} \sin\delta_{\mathrm{av}} + F_{y,\mathrm{ar}} \cos\delta_{\mathrm{ar}} + F_{x,\mathrm{ar}} \sin\delta_{\mathrm{ar}} \\ I_z \dot{r} &= L_{\mathrm{av}} \left(F_{y,\mathrm{av}} \cos\delta_{\mathrm{av}} - F_{x,\mathrm{av}} \sin\delta_{\mathrm{av}}\right) - L_{\mathrm{ar}} \left(F_{y,\mathrm{ar}} \cos\delta_{\mathrm{ar}} - F_{x,\mathrm{ar}} \sin\delta_{\mathrm{ar}}\right) \end{aligned} \tag{5.32}$$

où v_x et v_y sont respectivement les vitesses longitudinales et latérales. m est la masse du véhicule, r est la vitesse de lacet, I_z le moment d'inertie, $F_{x,\mathrm{av}}$ et $F_{x,\mathrm{ar}}$ sont les forces latérales et $F_{y,\mathrm{av}}$ et $F_{y,\mathrm{ar}}$ sont les forces longitudinales qui s'appliquent sur les roues avant et arrière. δ_{ar} et δ_{av} sont les deux angles de braquage.

Nous considérons une vitesse v_x longitudinale constante. De plus, comme la résistance aérodynamique est négligée, les forces $F_{x,\mathrm{av}}$ et $F_{x,\mathrm{ar}}$ s'annulent. On obtient ainsi

$$\dot{v}_y = \frac{F_{y,\mathrm{av}}}{m} \cos\delta_{\mathrm{av}} + \frac{F_{y,\mathrm{ar}}}{m} \sin\delta_{\mathrm{ar}} - v_x r, \tag{5.33}$$

$$\dot{r} = \frac{L_{\mathrm{av}}}{I_z} F_{y,\mathrm{av}} \cos\delta_{\mathrm{av}} - \frac{L_{\mathrm{ar}}}{I_z} F_{y,\mathrm{ar}} \cos\delta_{\mathrm{ar}}. \tag{5.34}$$

En considérant que que l'on se trouve dans la partie linéaire de la courbe donnant la force latérale en fonction de l'angle de dérive [64], on peut écrire

$$F_{y,\mathrm{av}} = C_{\mathrm{av}} \alpha_{\mathrm{av}}, \tag{5.35}$$

$$F_{y,\mathrm{ar}} = C_{\mathrm{ar}} \alpha_{\mathrm{ar}}, \tag{5.36}$$

où C_{av} et C_{ar} sont les raideurs d'envirage des pneus avant et arrière. Les variables α_{av} et α_{ar} sont respectivement les angles de dérive (*slip angle* en anglais) des roues avant et arrière. Les valeurs de ces angles sont [65]

$$\alpha_{\mathrm{av}} = \delta_{\mathrm{av}} - \arctan\left(\frac{v_y + L_{\mathrm{av}} r}{v_x}\right), \tag{5.37}$$

$$\alpha_{\mathrm{ar}} = \delta_{\mathrm{ar}} - \arctan\left(\frac{v_y - L_{\mathrm{ar}} r}{v_x}\right), \tag{5.38}$$

où L_{av} et L_{ar} sont la distance entre le centre de gravité du véhicule et les roues avant et arrière.

Les coordonnées du centre de gravité ainsi que l'orientation du véhicule sont

$$\begin{aligned} \dot{x}_g &= v_x \cos(\theta) - v_y \sin(\theta), \\ \dot{y}_g &= v_x \sin(\theta) + v_y \cos(\theta), \\ \dot{\theta} &= r. \end{aligned} \tag{5.39}$$

Nous avons maintenant un modèle non-linéaire à cinq degrés de liberté régi par les équations (5.33), (5.34) et (5.39). Cette équation de changement d'état peut être utilisée dans notre RRT afin de calculer $\mathbf{x}(t + \Delta t)$ en connaissant $\mathbf{x}(t)$ et la commande $\mathbf{u}$.

Nous avons utilisé ce modèle dans [66, 67] et des exemples de résultats obtenus avec ce type de modèle sont présentés sur les figures 5.20 et 5.21. Sur ces figures, le véhicule non ponctuel se déplace à une vitesse constante de $80\mathrm{km.h}^{-1}$.

Sur la figure 5.20, dans un environnement contenant peu d'obstacles, une moyenne de 12 000 nœuds est nécessaire pour construire la trajectoire. Sur la figure 5.21, dans un environnement contenant plus d'obstacles, le nombre moyen de nœuds nécessaires pour trouver une trajectoire est d'environ 15 000.

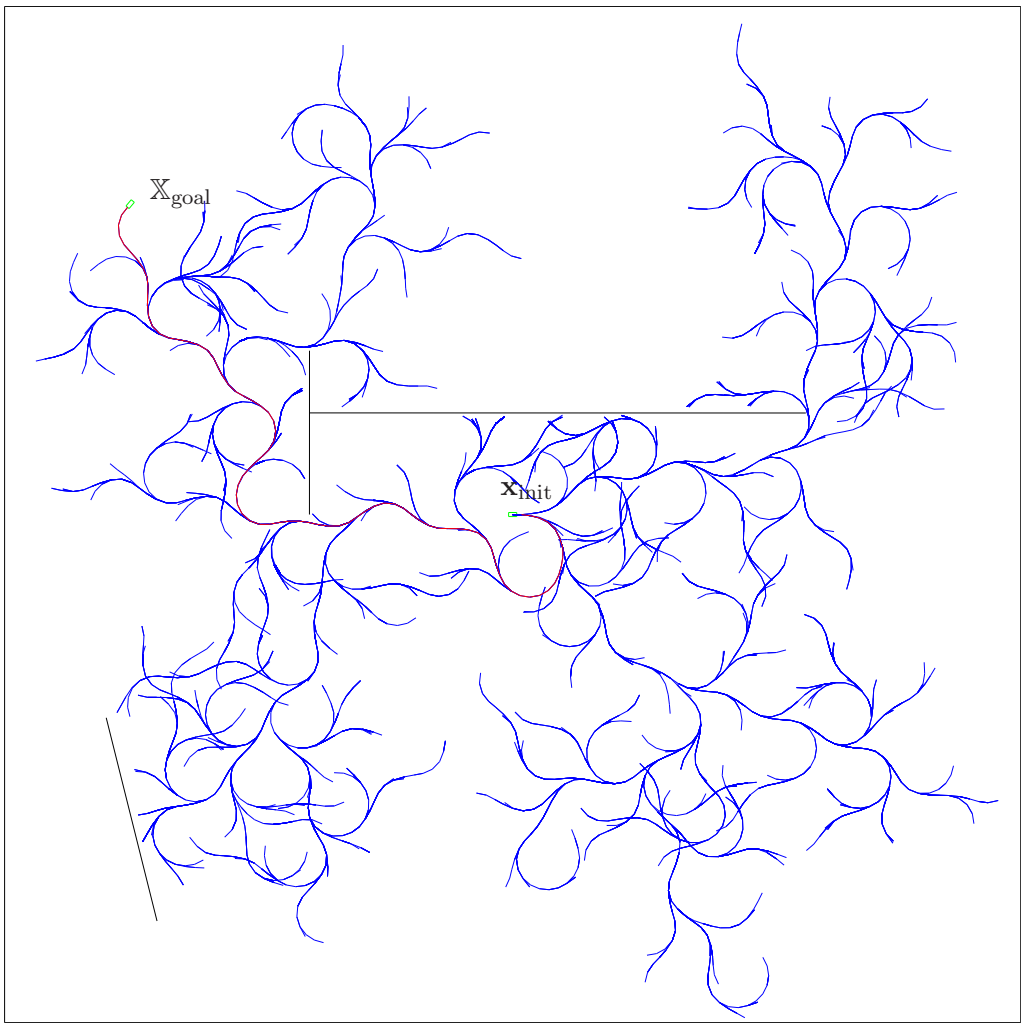


FIG. 5.20 – Trajectoire planifiée par RRT avec le modèle dynamique présenté dans le paragraphe 5.4.4 pour un véhicule non ponctuel dans un environnement contenant peu d'obstacles

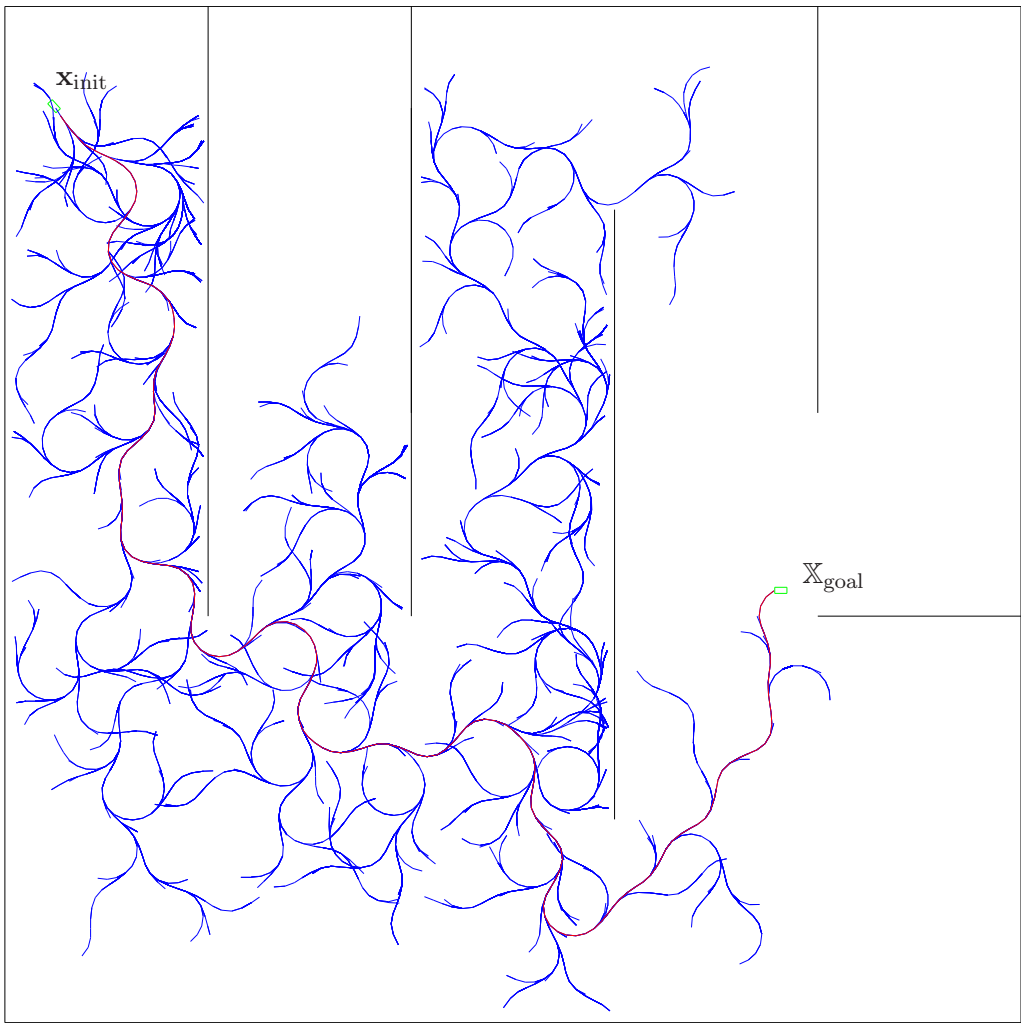


FIG. 5.21 – Trajectoire planifiée avec le modèle dynamique présenté dans le paragraphe 5.4.4 pour un véhicule non ponctuel dans un environnement en forme de labyrinthe

5.5 Conclusion

Nous avons présenté dans ce chapitre les contraintes liées à la planification de trajectoires dans le contexte de la robotique, différents modèles de véhicules cinématiques ainsi qu'un modèle simplifié tenant compte de la dynamique.

L'utilisation d'un modèle dynamique permet une meilleur modélisation des mouvements du véhicule et réduit ainsi la correction nécessaire lors de la phase de navigation. Il ne va pas pour autant assurer une sûreté quelconque puisque ce modèle dynamique ne tient pas compte des incertitudes de certains paramètres. Nous avons ici présenté un modèle dynamique simple de type bicyclette avec cinq degrés de liberté. Ce modèle simplifié n'est bien évidemment pas assez complet pour pouvoir modéliser l'ensemble du véhicule. L'état du véhicule devrait en effet être représenté par un plus grand nombre de degrés de liberté pour que les trajectoires simulées se rapprochent mieux des trajectoires réelles du véhicule.

Une approche par utilisation de modèles dynamiques excessivement réalistes n'est pourtant pas une solution pour s'assurer que les trajectoires générées par le planificateur soient sûres. Il semble irréaliste, par exemple, de modéliser le jeu dans les commandes dû à l'usure naturelle des pièces ou l'influence du revêtement du sol...

Que le modèle soit cinématique ou dynamique, excepté pour des manœuvres de courtes durées avec des conditions optimales, les trajectoires générées par un planificateur déterministe ne peuvent être utilisées pour espérer naviguer en toute sécurité.

Afin de pouvoir planifier des trajectoires sûres, c'est-à-dire des trajectoires qui pourront être suivies avec un très faible risque de collision, il est nécessaire de prendre en compte les incertitudes liées au modèle dès la phase de planification. Les algorithmes présentés aux chapitres 3 et 4 montrent que cela est possible. Ces algorithmes sont détaillés dans le cadre d'une utilisation en robotique dans les deux chapitres qui suivent.

Chapitre 6

Planification avec estimation par filtrage de Kalman en robotique

Sommaire

Nous présentons dans ce chapitre un cas concret de mise en œuvre du planificateur Kalman-RRT décrit dans le chapitre 3. Le problème à résoudre est le même que dans l'introduction du chapitre 3. Le système à déplacer est un véhicule dont le modèle est la *voiture simple* présentée au paragraphe 5.1.3 page 92.

Ce véhicule est supposé équipé d'odomètres placés sur ses roues arrières et de capteurs à ultrasons. L'odomètres est un capteur proprioceptif qui permet de connaître la distance parcourue par la roue sur laquelle il est placé entre deux instants. Le fait d'avoir disposé un odomètre sur chacune des roues arrières permet ainsi d'estimer le mouvement longitudinal ainsi que le mouvement de rotation du véhicule.

Afin de pouvoir se localiser plus précisément, et notamment de réduire les erreurs qui peuvent s'être accumulées par l'utilisation exclusive des odomètres, nous avons décidé d'utiliser aussi des capteurs télémétriques à ultrasons (capteurs extéroceptifs). Les modèles de fonctionnement des odomètres et d'obtention de mesures télémétriques adaptées à notre problème seront détaillés au paragraphe 6.1.1. Dans le chapitre 3, nous discutions de la difficulté d'intégrer les mesures de capteurs extéroceptifs à cause de l'optimisme que cela peut engendrer dans la recherche de trajectoires. Nous considérons ici que le véhicule se trouve sur la position la plus probable, la moyenne du vecteur gaussien, lors de la simulation des capteurs extéroceptifs. Nous utiliserons dans ce chapitre une ceinture de capteurs à ultrasons disposés autour du véhicule. Cette partie sera détaillée au paragraphe 6.1.2.

Après cette présentation des capteurs et de leurs interactions avec notre algorithme de planification Kalman-RRT, nous présenterons un test de collision simplifié (paragraphe 6.3) qui nous permettra de résoudre des problèmes de planification. Les trajectoires ainsi planifiées seront présentées et discutées au paragraphe 6.4.

6.1 Intégrations des capteurs

Dans cette partie, nous présentons les capteurs utilisés et une approche premettant d'intégrer les mesures qu'ils fournissent à notre planificateur.

6.1.1 Capteurs proprioceptifs et prédiction

Nous utilisons ici comme capteurs proprioceptifs des codeurs incrémentaux sur les roues. Ces capteurs, appelés odomètres, retournent le nombre de quantièmes de tours fait par la roue à laquelle ils sont liés. Ces données seront utilisées dans la phase de prédiction du filtre de Kalman (paragraphe 3.1.1 page 65). Installés sur chacune des roues motrices, ils permettent de savoir quels sont les déplacements effectués par le véhicule.

Ainsi, à un instant k, nous sommes en mesure de calculer la distance parcourue pendant l'intervalle de temps $[0;k]$, d'où se déduit simplement la distance parcourue entre les instants k et $k+1$. Nous noterons par la suite $\Delta\text{odo}_\text{g}$ et $\Delta\text{odo}_\text{d}$ les mesures de distances renvoyées par les odomètres pour les roues gauche et droite.

Connaissant ces deux distances ainsi que la distance entre les deux roues, notée e, nous pouvons en déduire les déplacements longitudinaux ainsi que la rotation entre deux positions du véhicule :

$$\Delta s = \frac{\Delta\text{odo}_\text{g} + \Delta\text{odo}_\text{d}}{2} \tag{6.1}$$

$$\Delta\theta = \frac{\Delta\text{odo}_\text{g} - \Delta\text{odo}_\text{d}}{e} \tag{6.2}$$

Ces deux mesures composent ce que nous considèrerons comme le vecteur de commande appliqué au

véhicule entre les instants k et $k+1$

$$\mathbf{u}_k = \begin{bmatrix} \Delta s \\ \Delta\theta \end{bmatrix}. \tag{6.3}$$

Soit le modèle d'état

$$\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{v}_k, \tag{6.4}$$

où $\mathbf{x} = [x\ y\ \theta]^{\mathrm{T}}$ est l'état du système, $\mathbf{u}_k$ est sa commande et $\mathbf{v}_k$ rassemble les effets de l'ensemble des sources d'erreur sur la mesure du déplacement.

Comme le vecteur de commande du modèle de véhicule pris par le planificateur (5.13) diffère de celui utilisé ici, il nous faut transformer la commande composée de la vitesse et de l'angle de braquage en un vecteur exprimant un déplacement longitudinal Δs et une rotation $\Delta\theta$. Pour le modèle de type *voiture simple*, on pourra utiliser

$$\Delta s = v \cdot \Delta t \tag{6.5}$$

$$\Delta\theta = \frac{v}{L} \tan\delta \cdot \Delta t. \tag{6.6}$$

Pour calculer la prédiction $\mathbf{x}_{k+1|k}$ de l'état à un pas, on utilise la partie déterministe du modèle (6.4)

$$\mathbf{x}_{k+1|k} = \mathbf{f}_k(\mathbf{x}_{k|k}, \mathbf{u}_k). \tag{6.7}$$

La matrice de covariance de l'erreur de prédiction est approximée par

$$\mathbf{P}_{k+1|k} = \mathbf{A}_k \mathbf{P}_{k|k} \mathbf{A}_k^{\mathrm{T}} + \mathbf{V}_k, \tag{6.8}$$

où $\mathbf{A}_k$ s'obtient en linéarisant l'équation de changement d'état autour de $\hat{\mathbf{x}}_{k|k}$

$$\mathbf{A}_k = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}_k}\right)_{\mathbf{x}_k = \hat{\mathbf{x}}_{k|k}} = \begin{bmatrix} 1 & 0 & -\Delta s \cdot \sin\left(\hat{\theta}_{k|k} + \frac{\Delta\theta}{2}\right) \\ 0 & 1 & \Delta s \cdot \cos\left(\hat{\theta}_{k|k} + \frac{\Delta\theta}{2}\right) \\ 0 & 0 & 1 \end{bmatrix}, \tag{6.9}$$

et où $\mathbf{V}_k$, la matrice de covariance de $\mathbf{v}_k$ est obtenue en utilisant le même modèle que dans [68].

Selon le modèle expérimentale utilisé dans [68], les incertitudes engendrées par $\mathbf{v}_k$ sont proportionnelles au déplacement. Les variances sur x, y et θ sont ainsi majorées proportionnellement au déplacement du système en introduisant deux coefficients : k_{ss}, facteur de dérive en translation, lors d'un déplacement en translation (en %) et $k_{\theta\theta}$, facteur de dérive en rotation, lors d'un déplacement en rotation (en %). L'expression utilisée dans [68] est

$$\begin{aligned} \sigma_x^2(\text{majoré}) &= \left(\sigma_x + k_{ss}|\Delta s \cos\theta_k|\right)^2, \\ \sigma_y^2(\text{majoré}) &= \left(\sigma_y + k_{ss}|\Delta s \sin\theta_k|\right)^2, \\ \sigma_\theta^2(\text{majoré}) &= \left(\sigma_\theta + k_{\theta\theta}|\Delta\theta|\right)^2. \end{aligned} \tag{6.10}$$

La matrice de covariance $\mathbf{V}_k$ de $\mathbf{v}_k$ s'écrit donc

$$\mathbf{V}_k = \begin{bmatrix} 2\sigma_x k_{ss}|\Delta s \cos\theta_k| + k_{ss}^2 \Delta s^2 \cos^2\theta_k & 0 & 0 \\ 0 & 2\sigma_y k_{ss}|\Delta s \sin\theta_k| + k_{ss}^2 \Delta s^2 \sin^2\theta_k & 0 \\ 0 & 0 & 2\sigma_\theta k_{\theta\theta}|\Delta\theta| + k_{ss}^2 \Delta\theta_k{}^2 \end{bmatrix}, \tag{6.11}$$

et dans la suite, nous prendrons comme [68] $k_{ss} = 2\%$ et $k_{\theta\theta} = 3\%$.

Nous sommes ainsi en mesure d'utiliser les odomètres situés sur les roues du véhicule pour prédire

son état. L'utilisation de cette seule étape de prédiction augmenterait sans cesse l'incertitude sur l'état du système. Il est donc nécessaire de lui adjoindre une étape de correction pour tenter de la réduire.

6.1.2 Capteurs extéroceptifs et correction

La phase de correction du filtre de Kalman (paragraphe 3.1.2 page 65) exploite des observations $\mathbf{z}_{k+1}$ pour affiner le positionnement du robot. Ces observations sont obtenues à partir de données issues de capteurs exteroceptifs. Deux types de capteurs peuvent être utilisés : ceux fournissant une mesure de positionnement globale comme le GPS ou Galileo, qui peuvent être utilisés à l'extérieur, et ceux fournissant une mesure locale, c'est-à-dire faisant référence aux obstacles de l'environnement. C'est le cas d'une ceinture de capteurs à ultrasons ou d'un télémètre laser pour les robots d'intérieur ou d'extérieur.

Lors de l'utilisation d'un système de positionnement global, la simulation est aisée. Il n'y a aucun mécanisme de mise en relation de données capteurs avec les données sur l'environnement à simuler. Une des difficultés liées à l'utilisation d'un tel capteur est l'obtention de l'orientation. Elle peut être déduite de la trajectoire ou en couplant ce système de positionnement à une boussole. La mise en œuvre d'un tel couple de capteurs avec un filtre de Kalman est aisée puisque les erreurs sur les capteurs portent directement sur la position et l'orientation du robot. L'information nouvelle $\hat{\mathbf{z}}_{k+1}$ issue des capteurs est donc la différence entre les coordonnées renvoyées par le couple GPS/boussole et les coordonnées calculées par la prédiction. Le bruit de mesure $\mathbf{w}_{k+1}$ est alors directement lié à la précision des capteurs en terme de distance et d'orientation.

L'utilisation de capteurs à ultrasons s'applique particulièrement à la robotique mobile en environnement d'intérieur. Le fonctionnement du simulateur utilisé est décrit dans [69]. Chacun des capteurs va renvoyer une mesure de distance, par rapport à l'obstacle le plus proche. Cette mesure sera utilisée dans un premier temps dans un processus de mise en correspondance afin de trouver quel obstacle de l'environnement a renvoyé l'onde émise par le capteur, et plus précisément les coordonnées du point de l'obstacle ayant renvoyé l'onde. Ces coordonnées sont alors utilisées dans l'étape de correction du filtre afin de corriger l'incertitude sur l'état du véhicule.

Lors de la simulation de ces capteurs, il faut également prendre en compte le temps nécessaire à l'obtention d'une nouvelle mesure. Ainsi, la phase de relocalisation ne pourra être utilisée que de temps en temps afin d'affiner la position.

Nous avons décidé dans la suite de simuler les mesures issues d'une ceinture de capteurs à ultrasons fournissant des informations toutes les secondes. La simulation est faite en considérant que le véhicule se trouve dans l'état qui possède la plus forte densité de probabilité (paragraphe 3.4.1). Le modèle des capteurs ainsi que le code utilisé pour les simuler sont issus de [69].

6.2 Représentation de l'incertitude

Nous avons vu précédemment dans la section 3.2 page 67 que l'incertitude sur l'état du système est représentée par une densité de probabilité gaussienne multidimensionnelle. Dans le cas du modèle de *voiture simple*, la gaussienne a trois dimensions qui correspondent à chacune des composantes de l'état, x, y et θ.

En utilisant les calculs fournis dans l'annexe A, nous sommes capable d'afficher l'incertitude sous la forme d'une ellipse $\mathcal{E}_{x,y}$ qui représente l'incertitude sur la position du point de référence du véhicule pour un niveau de confiance donné.

Cette ellipse $\mathcal{E}_{x,y}$ est utilisée dans le test de collision.

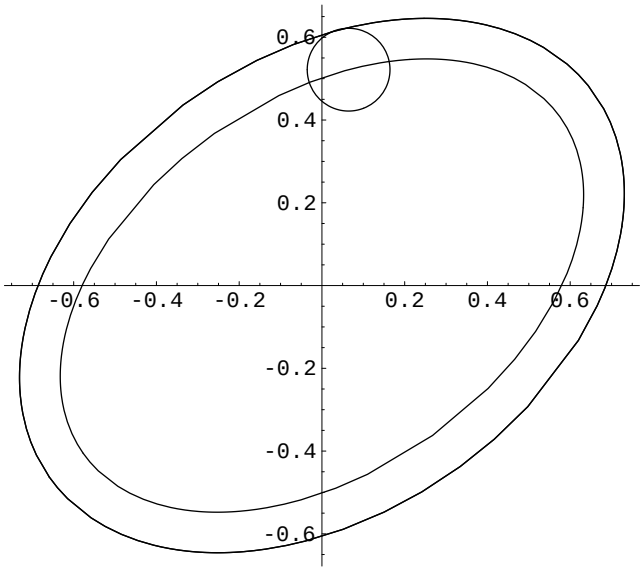


FIG. 6.1 – Surface balayée par l'approximation circulaire du robot lorsqu'il se situe dans une ellipse d'état obtenue pour un niveau de confiance donné

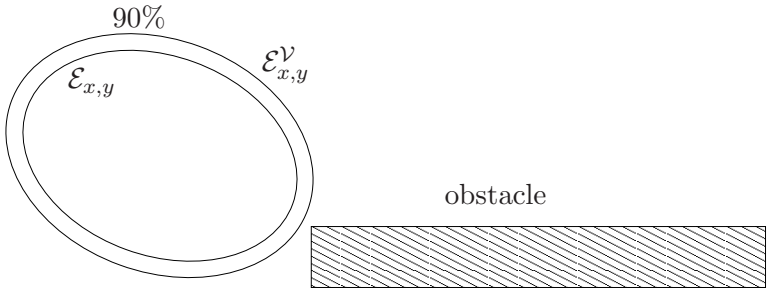


FIG. 6.2 – Vérification de la sûreté de l'état du véhicule pour un niveau de confiance donné

6.3 Test de collision

Le test de collision est identique à celui présenté dans le paragraphe 3.3 page 68. Les ellipses utilisées sont celles obtenues par la suppression de la composante en θ de l'ellipsoïde d'incertitude sur l'état (Annexe A) et qui sont notées $\mathcal{E}_{x,y}$.

Il n'est pas suffisant de tester si l'ellipse $\mathcal{E}_{x,y}$ rentre en collision avec l'environnement puisqu'il faut également tenir compte du caractère non ponctuel du véhicule.

Pour simplifier ce problème, nous englobons le véhicule dans un cercle de rayon r_v. La forme obtenue lorsque le cercle englobant le véhicule parcourt l'ellipse (figure 6.1) est la somme de Minkowski de l'ellipse et du cercle. Nous l'approximons par une ellipse $\mathcal{E}^{\mathcal{V}}_{x,y}$ dont les valeurs des demis axes sont égales à celles de $\mathcal{E}_{x,y}$ auxquelles ont été ajoutées r_v. L'orientation et le centre de $\mathcal{E}^{\mathcal{V}}_{x,y}$ restent les mêmes que ceux de $\mathcal{E}_{x,y}$.

Pour vérifier que le robot ne rentre pas en collision pour un niveau de confiance donné, il faut vérifier que l'ellipse $\mathcal{E}^{\mathrm{v}}_{x,y}$, qui représente la surface balayée par le robot malgré l'incertitude sur son état pour un niveau de confiance donné, ne rentre pas en collision avec un des obstacles de l'environnement (figure 6.2). Nous utilisons pour cela l'algorithme présenté dans l'annexe B.

De la même façon, une fois calculée l'ellipse $\mathcal{E}_{\mathrm{near,new}_{x,y}}$ englobant $\mathcal{E}_{\mathrm{near}_{x,y}}$ et $\mathcal{E}_{\mathrm{new}_{x,y}}$ en utilisant Maxdet (paragraphe 3.3 page 68), nous augmentons les demi-axes de $\mathcal{E}_{\mathrm{near,new}_{x,y}}$ avec le rayon de l'approximation circulaire du robot r_v afin d'obtenir la surface balayée par le robot entre les états

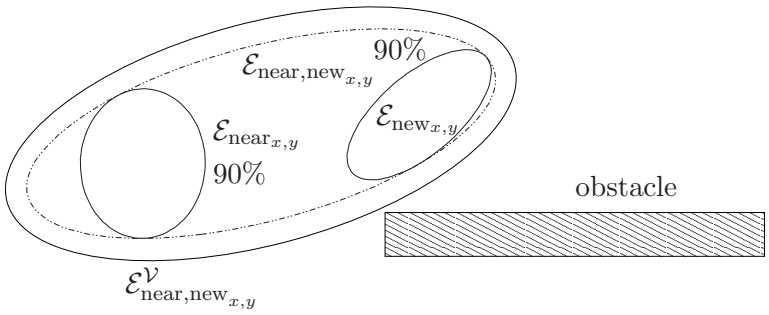


FIG. 6.3 – Vérification de la sûreté de la trajectoire du véhicule pour un niveau de confiance donné

$(\mathbf{x}, \mathbf{P})_{\text{near}}$ et $(\mathbf{x}, \mathbf{P})_{\text{new}}$ (figure 6.3). Pour cette nouvelle ellipse, notée $\mathcal{E}^{\mathcal{V}}_{\text{near,new}_{x,y}}$, nous utilisons également l'algorithme de l'annexe B pour vérifier qu'elle ne rentre en collision avec aucun des segments qui composent l'environnement.

6.4 Résultats obtenus

Nous présentons ici les résultats de planification obtenus avec l'algorithme Kalman-RRT en utilisant le modèle *voiture simple*. Ces résultats sont issus de [47]. Sur l'ensemble des figures de cette section, nous ne dessinons que les ellipses qui font suite à une étape de correction (nous dessinons donc une ellipse toutes les secondes compte tenu de la fréquence de rafraîchissement des mesures issues des capteurs que nous considérons) pour un niveau de confiance donné.

L'incertitude initiale est d'un mètre pour x et y et de 3° pour θ. Quand aucune valeur n'est spécifiée, les coefficients k_{ss} et $k_{\theta\theta}$ ont respectivement pour valeurs 2% et 3%. Les obstacles dessinés avec une ligne pleine sont détectables par les capteurs à ultrasons. Ceux dessinés avec une ligne en pointillés ne le sont pas.

Il est intéressant tout d'abord de reprendre les exemples d'environnements utilisés par LaValle et d'ajouter les incertitudes au mouvement du véhicule. La figure 6.4 représente une trajectoire générée dans cet environnement pour un véhicule de type *voiture simple*. Sur la figure 6.5, le véhicule ne peut tourner que dans un sens.

Sur les figures 6.6 et 6.7, nous montrons la nécessité d'utiliser Kalman-RRT plutôt que le RRT traditionnel afin de pouvoir rentrer dans une place de parking sans risquer d'heurter un obstacle.

Sur la figure 6.6, nous avons dessiné la trajectoire trouvée par un planificateur RRT traditionnel. Nous avons ensuite ajouté les incertitudes sur chacun des états composant la trajectoire trouvée. Le RRT trouve une trajectoire qui permet de rentrer dans la place. Cependant, lorsque nous regardons la taille de l'ellipse à l'arrivée près de la place de parking, nous voyons qu'il est impossible que le véhicule rentre dans cette place sans risquer une collision.

Sur la figure 6.7, nous utilisons le planificateur Kalman-RRT pour résoudre le même problème. Afin de pouvoir rejoindre la place de parking sans risque de collision à son arrivée, le véhicule doit longer le mur situé à sa droite. Il pourra ainsi se localiser et donc réduire l'incertitude sur son positionnement. A son arrivée à proximité de la place de parking, l'incertitude sur sa position est faible, il peut donc rejoindre cette place sans risque.

La figure 6.8 illustre une trajectoire générée dans un environnement en forme de labyrinthe. Sur cette figure, les coefficients k_{ss} et $k_{\theta\theta}$ ont pour valeur respective 3% et 5%. Avec de telles valeurs, le véhicule doit passer au milieu des couloirs afin de pouvoir rejoindre l'arrivée sans risque de collision.

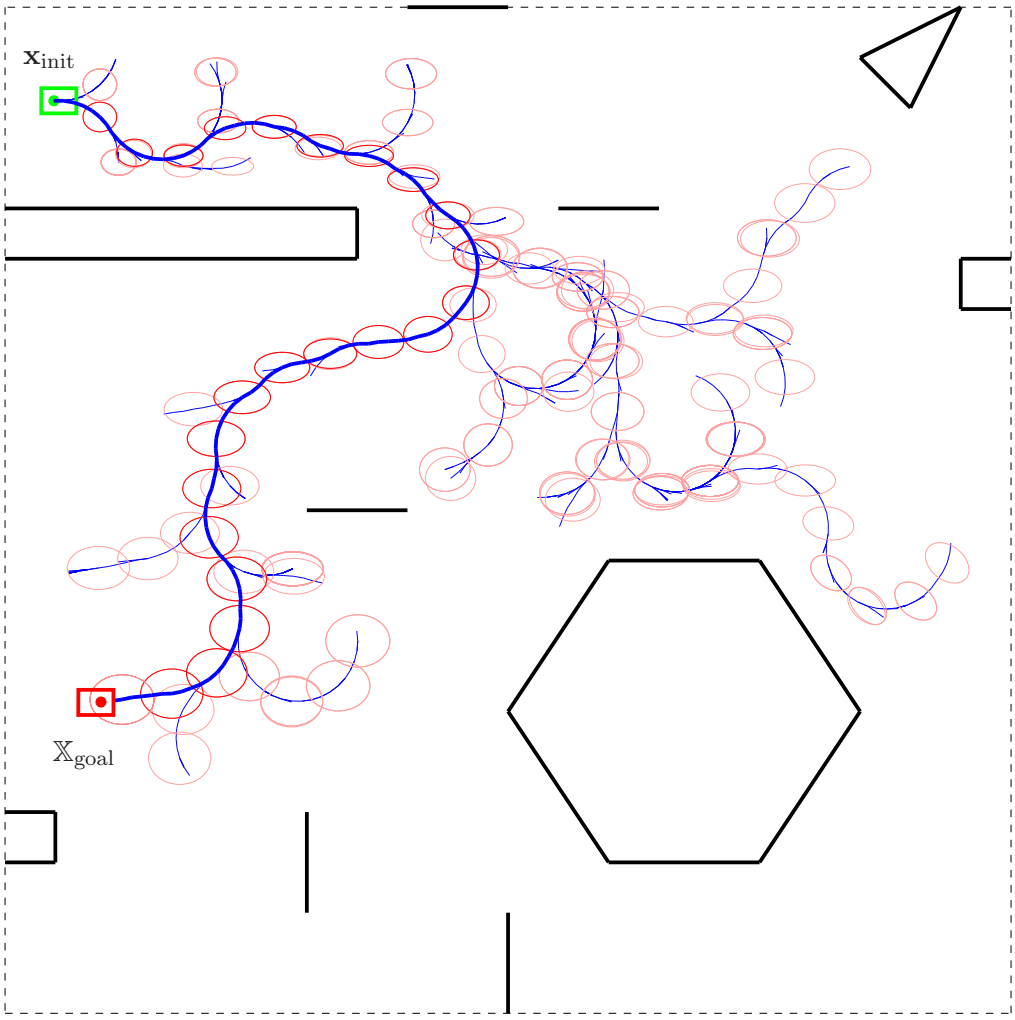


FIG. 6.4 – Trajectoire planifiée pour un véhicule de type *voiture simple*

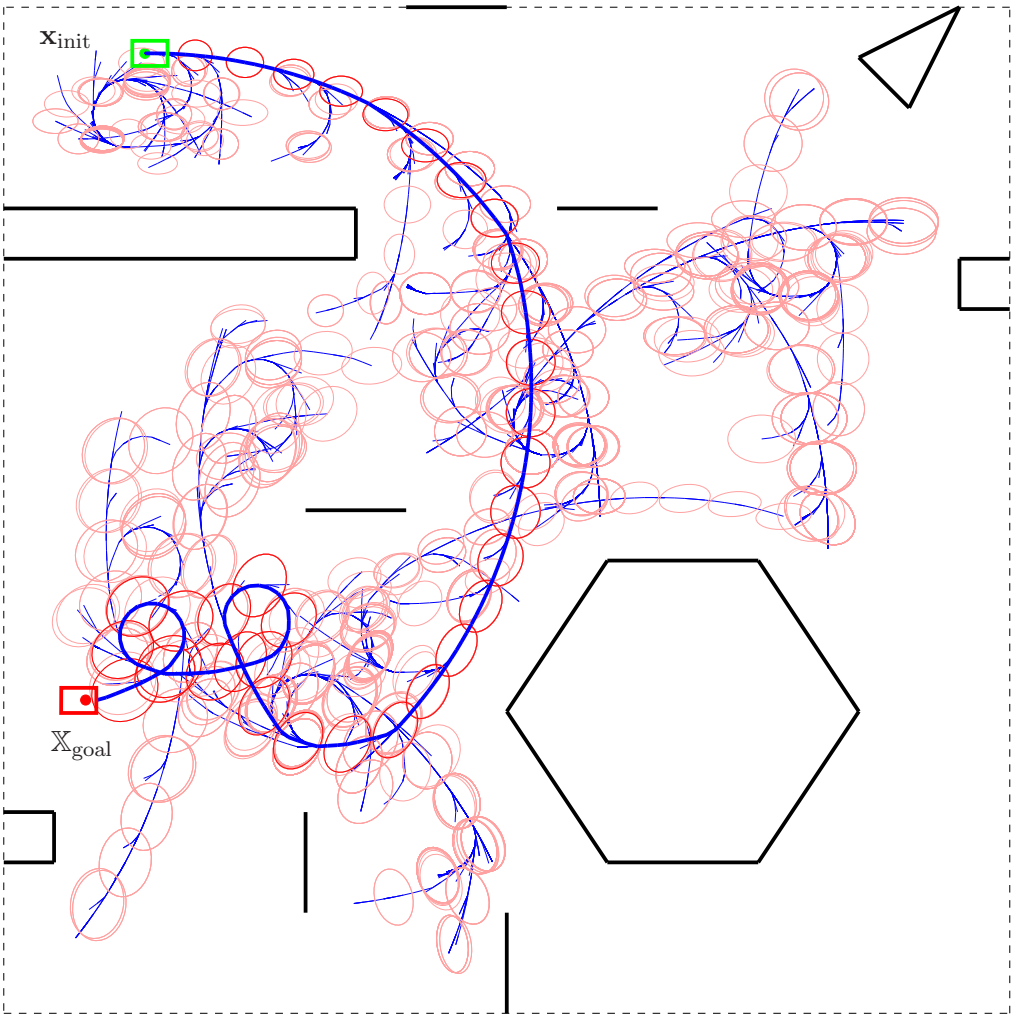


FIG. 6.5 – Trajectoire planifiée pour un véhicule ne pouvant tourner que dans un sens

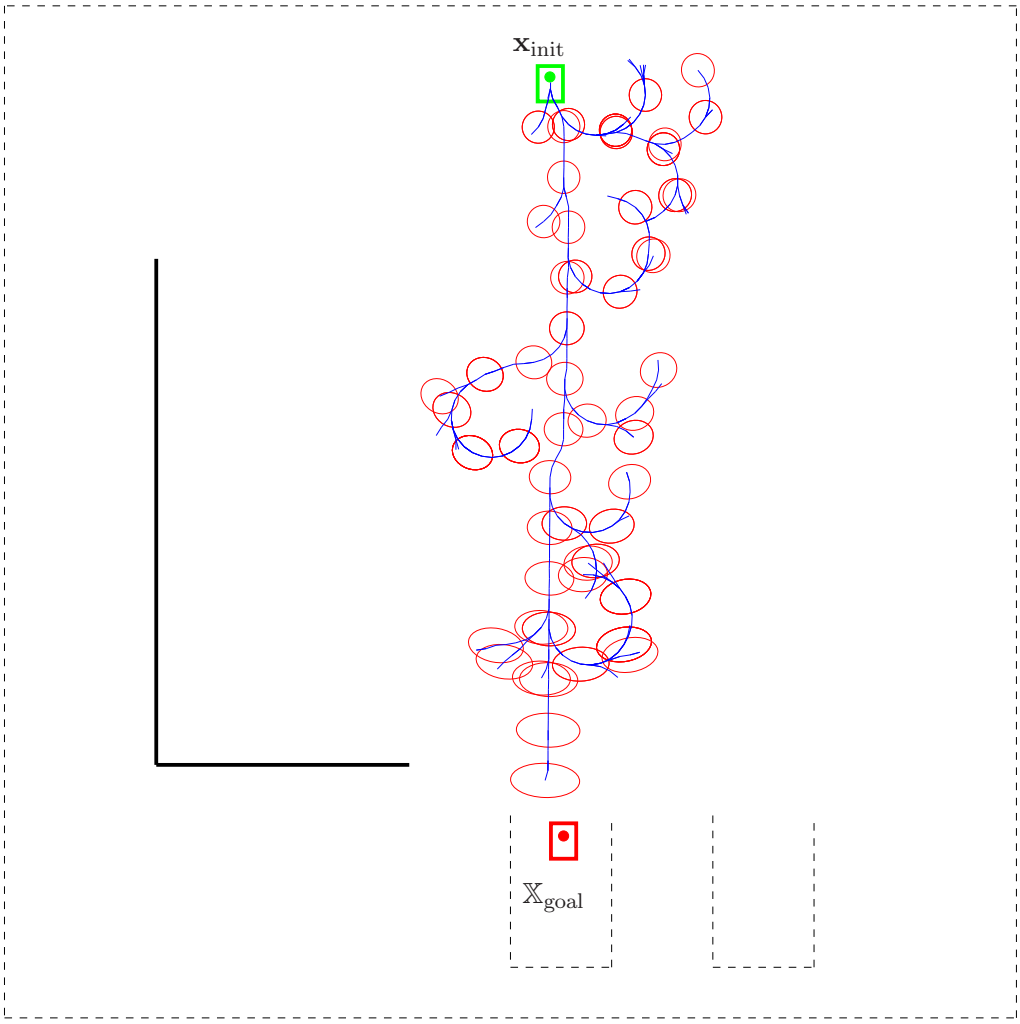


FIG. 6.6 – Le véhicule ne peut rejoindre la place de parking

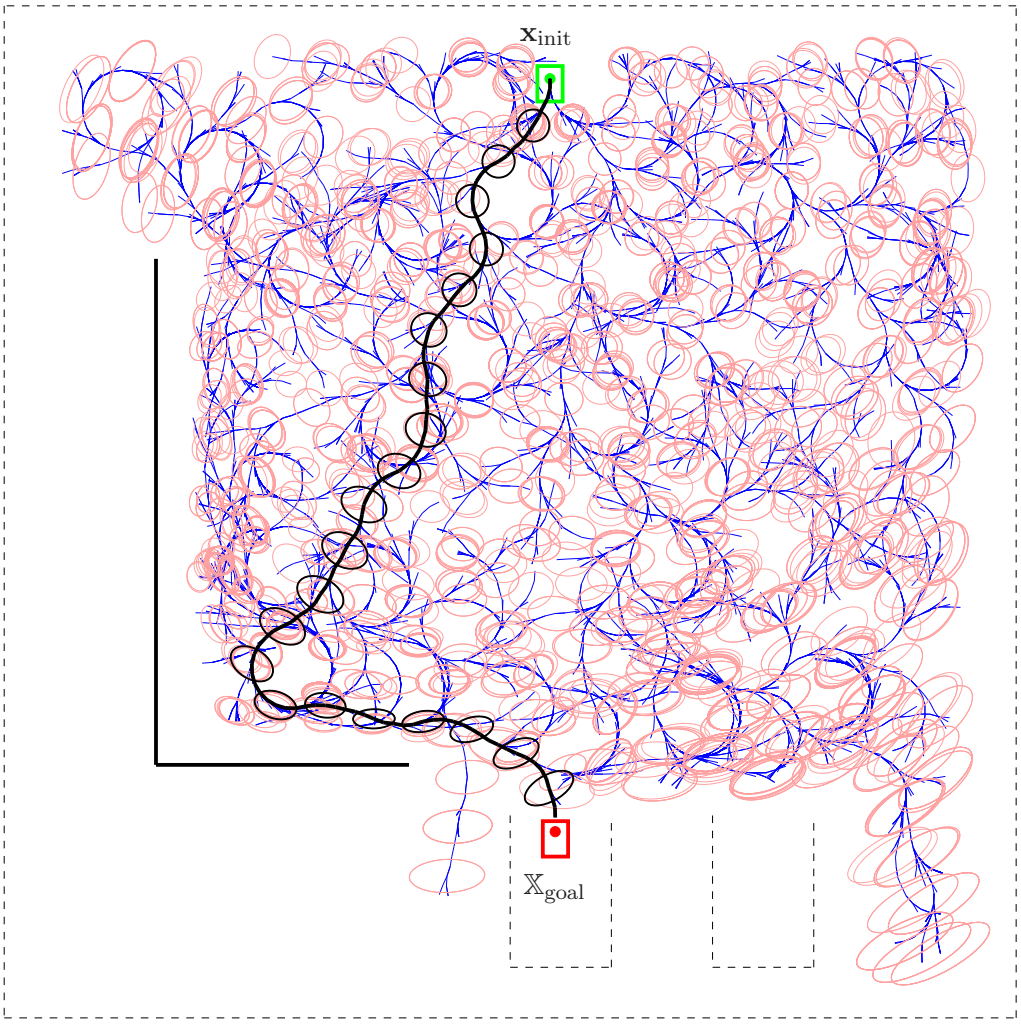


FIG. 6.7 – Le véhicule peut rejoindre la place de parking s'il longe le mur

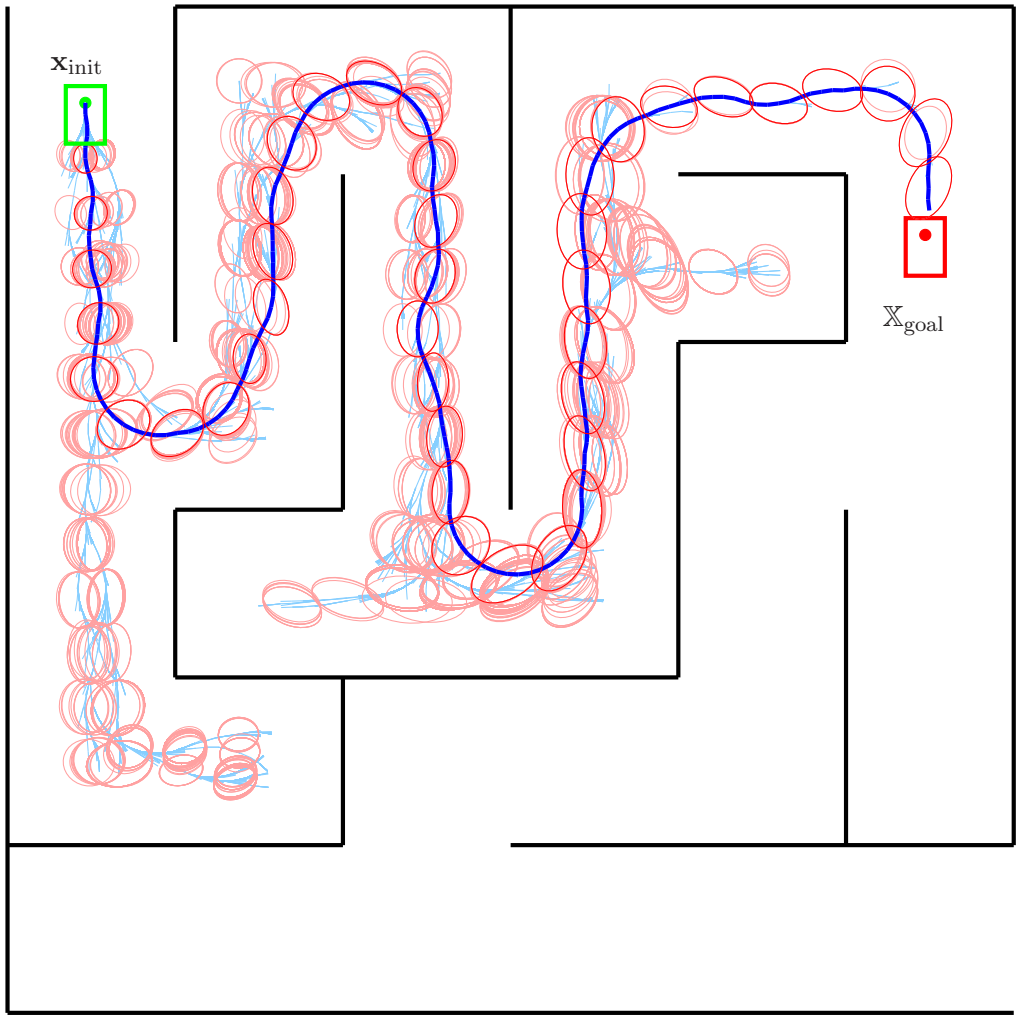


FIG. 6.8 – Planification avec le Kalman-RRT dans un environnement en forme de labyrinthe

6.5 Conclusion

Dans ce chapitre, nous avons mis en œuvre le planificateur Kalman-RRT avec un modèle de véhicule de type *voiture simple*. L'algorithme prend en compte les incertitudes sur le positionnement du véhicule et cela permet de planifier des trajectoires en diminuant les risques de collision lors de son suivi pendant la phase de navigation. Cependant, l'utilisation d'une représentation probabiliste de l'incertitude, le fait de simuler les mesures capteurs en considérant que le véhicule se trouve dans l'état qui possède la plus forte densité de probabilité pendant les phases de correction du filtre de Kalman et l'approximation faite durant le test de collision ne peuvent garantir la sûreté de la trajectoire. Cet algorithme est donc une première étape dans la prise en compte des incertitudes. Il permet de minimiser les risques mais ne garantit pas que le véhicule ne rentrera pas en collision avec un des obstacles de l'environnement.

Malgré ces imperfections, cet algorithme permet de générer des trajectoires qui donnent une idée pertinente des zones de l'espace à explorer afin de profiter de mesures qui favoriseront la relocalisation lors de la phase de navigation. La figure 6.7 illustre parfaitement cette capacité puisque le planificateur préconise de faire passer le véhicule près du mur afin de s'en servir comme point de repère pour sa localisation.

Dans le chapitre suivant, nous utiliserons le Set-RRT que nous avons présenté au chapitre 4 afin de dépasser ces problèmes et de générer des trajectoires où le risque de collision est nul, à condition que des hypothèses explicites de bornes sur les quantités incertaines soient vérifiées.

Chapitre 7

Planificateur ensembliste en robotique

Sommaire

Dans ce chapitre, nous présentons une mise en œuvre de l'algorithme Set-RRT présenté au chapitre 4 et plus particulièrement de sa spécialisation aux vecteurs d'intervalles, le Box-RRT dans le cadre d'une planification de trajectoires en robotique. Nous utiliserons le modèle *voiture simple* présenté au paragraphe 5.1.3 page 92 et définirons dans ce cas un planificateur permettant une planification sûre dans un environnement à deux dimensions.

Le véhicule dont nous planifions la trajectoire est équipé de deux odomètres qui le renseignent sur le déplacement des roues arrières. Contrairement au chapitre 6, où nous planifiions des trajectoires avec l'algorithme Kalman-RRT, aucun capteur extéroceptif ne sera considéré afin d'éviter le problème lié à la simulation de ces capteurs (paragraphe 3.4 page 68).

Comme dans le chapitre précédent, il nous faut définir la métrique ainsi que le test de collision utilisé. L'étape de prédiction est, quant à elle, identique à celle présentée dans au paragraphe 4.2.2 page 77.

7.1 Metrique

Nous utilisons la (pseudo) métrique présentée dans le paragraphe 5.3.3 page 96 pour calculer la distance entre deux vecteurs d'intervalles. Elle est définie ici comme suit :

$$d([\mathbf{x}_1], [\mathbf{x}_2]) = d_1([\mathbf{x}_1], [\mathbf{x}_2]) + d_2([\mathbf{x}_1], [\mathbf{x}_2]), \tag{7.1}$$

où les fonctions d_1 et d_2 sont définies par :

$$\begin{aligned} d_1([\mathbf{x}_1], [\mathbf{x}_2]) &= \mathrm{ub}\left(\sqrt{([x_1] - [x_2])^2 + ([y_1] - [y_2])^2}\right), \\ d_2([\mathbf{x}_1], [\mathbf{x}_2]) &= L \cdot \mathrm{ub}\left(|[\theta_1] - [\theta_2]|\right). \end{aligned}$$

Géométriquement, $d_1([\mathbf{x}_1], [\mathbf{x}_2])$ peut être interprété comme la distance maximale entre les sommets des pavés associés aux deux première composantes des vecteurs d'intervalles représentant l'état $[\mathbf{x}_1]$ et $[\mathbf{x}_2]$ (figure 7.1(a)). Quant à $d_2([\mathbf{x}_1], [\mathbf{x}_2])$, il correspond à la longueur maximale de l'arc de cercle que suivrait le véhicule s'il pouvait effectuer une rotation de $[\theta_1]$ à $[\theta_2]$ en tournant sur lui-même autour de l'origine de son repère $\mathbf{V}$ (Figure 7.1(b)).

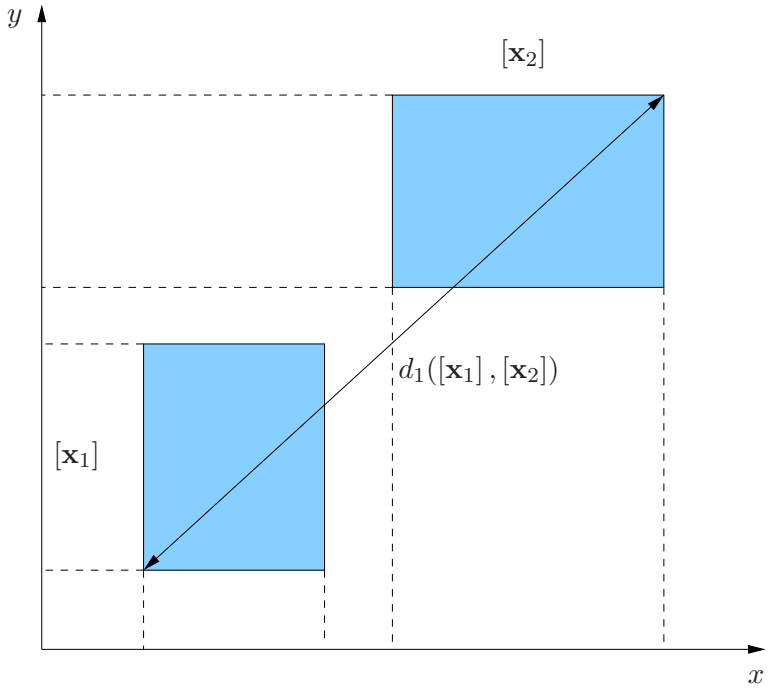
Comme pour le RRT, le choix de cette pseudo métrique n'est pas primordial. Elle n'est là que pour donner une idée de la distance entre deux états. Il n'est ainsi absolument pas nécessaire d'utiliser une véritable distance pour voir l'arbre s'étendre rapidement dans les parties inexplorées du plan. C'est pourquoi nous n'utilisons donc pas une distance d'Haussdorf (section 4.2.1 page 76), car celle-ci est plus difficile à évaluer.

7.2 Prise en compte des mesures issues des capteurs

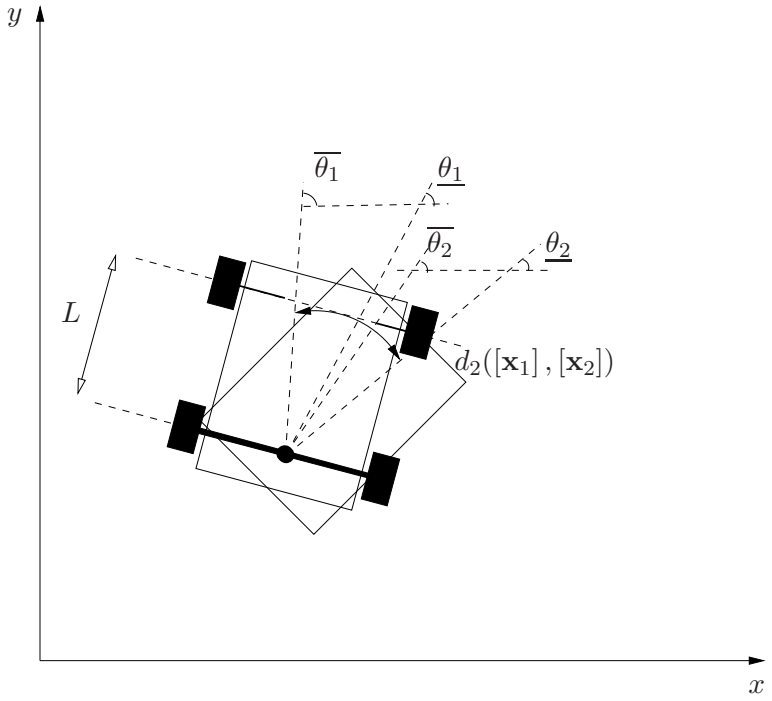
Les odomètres renvoient la distance parcourue par chacune des roues. Ces mesures sont utilisées pour calculer le mouvement longitudinal Δs et le mouvement de rotation $\Delta\theta$.

Les erreurs de mesures résultent en partie de l'erreur de quantification due à la résolution r des odomètres. Ainsi, quand un odomètre renvoie une valeur entière p, nous devons considérer le résultat sous forme d'intervalle $[p] = [p, p + 1]$.

Les mouvements longitudinaux et de rotation doivent donc appartenir respectivement aux inter-



(a) Distance maximale entre les sommets de deux pavées 2D



(b) Longueur de l'arc de cercle

FIG. 7.1 – Les deux composants de la pseudo métrique

valles :

$$[\Delta s] = [R].\frac{[\pi]\,([p_l] + [p_r])}{r}, \tag{7.2}$$

$$[\Delta\theta] = [R].\frac{[\pi]\,([p_l] - [p_r])}{[e].r}, \tag{7.3}$$

ou $[R]$ est l'intervalle contenant le rayon de la roue, r est la résolution de l'odomètre, $[\pi]$ est un intervalle dont les bornes sont des nombres représentables en machine et qui contient π, $[e]$ est un intervalle contenant la distance entre les roues avants et arrières et $[p_\mathrm{r}]$ et $[p_\mathrm{l}]$ sont les intervalles associés aux positions retournées respectivement par les odomètres droit et gauche.

Les erreurs dues aux glissements doivent également être prises en compte et ajoutées à $[\Delta s]$ et $[\Delta\theta]$. Ces erreurs doivent être ajoutées aux valeurs renvoyées par le simulateur de capteurs en fonction des conditions dans lesquelles évolue le véhicule. Elles correspondent au bruit $\mathbf{v}$ défini lors de la présentation de l'algorithme de résolution Box-RRT. L'intervalle $[\mathbf{v}]$ auquel appartient la mesure $\mathbf{v}$ du bruit doit donc être ajouté aux valeurs de $[\Delta s]$ et $[\Delta\theta]$.

7.3 Test de collision

Avant d'exécuter l'algorithme Box-RRT, nous devons vérifier que les états contenus dans les intervalles de départ $[\mathbf{x}_\mathrm{init}]$ et d'arrivée $[\mathbf{x}_\mathrm{goal}]$ appartiennent à $\mathbb{X}_\mathrm{free}$. De plus, avant d'ajouter un nœud au graphe G, nous devons déterminer si ce nœud est sûr et si la trajectoire entre deux boites consécutives l'est également. Ce n'est que dans ce cas que ce nœud peut être ajouté. Ce test est contenu dans la fonction `CollisionFreePath` décrite dans ce qui suit.

7.3.1 État sûr

La forme du véhicule est approximée dans ce test de collision par un polygone convexe. On considère deux référentiels, le référentiel de l'environnement $\mathcal{W}$ et celui lié au système $\mathcal{V}$. Chaque sommet $\mathbf{c}_i, i = 1 \ldots n_\mathrm{v}$, du polytope a des coordonnées connues $(x_i^\mathcal{V}, y_i^\mathcal{V})$ dans le référentiel $\mathcal{V}$. Nous devons calculer ces coordonnées dans le référentiel de l'environnement en considérant que les coordonnées du système dans $\mathcal{W}$ sont imprécises et spécifiées par le vecteur d'intervalles $[\mathbf{x}] = ([x], [y], [\theta])^\mathrm{T}$. Les coordonnées du sommet $([x_i^\mathcal{W}], [y_i^\mathcal{W}])$ dans le référentiel de l'environnement sont calculées grâce à

$$\begin{pmatrix}[x_i^\mathcal{W}]\\ [y_i^\mathcal{W}]\end{pmatrix} = \begin{pmatrix}[x]\\ [y]\end{pmatrix} + \begin{pmatrix}\cos[\theta] & -\sin[\theta]\\ \sin[\theta] & \cos[\theta]\end{pmatrix}\begin{pmatrix}x_i^\mathcal{V}\\ y_i^\mathcal{V}\end{pmatrix}. \tag{7.4}$$

On obtient ainsi n_v pavés $(\left[x_i^\mathcal{W}\right], \left[y_i^\mathcal{W}\right]), i = 1 \ldots n_\mathrm{v}$. Il ne suffit pas de tester si ces pavés entrent en collision avec l'environnement. Nous devons les réunir comme ils le sont dans $\mathcal{V}$ pour représenter l'approximation extérieure polygonale du véhicule. Ainsi, nous calculons l'enveloppe convexe de ces n_v pavés en utilisant la méthode *Graham scan* [70]. Cet algorithme (algorithme 19) trouve l'enveloppe convexe d'un ensemble fini de points avec une complexité en temps de $\mathrm{O}(n \log n)$.

Parmi les quatre sommets de chaque pavé $(\left[x_i^\mathcal{W}\right], \left[y_i^\mathcal{W}\right]), i = 1 \ldots n_\mathrm{v}$, l'algorithme *Graham scan* va d'abord chercher (ligne 1) le sommet appelé pivot Q_1, qui est le sommet avec la plus petite coordonnée y (si deux sommets ont la même coordonnée en y, il prendra celui avec la plus petite coordonnée en x). À la ligne 2, les sommets sont triés en fonction de l'angle que la droite passant par eux et Q_1 fait avec l'axe des x. A la ligne 6, l'algorithme teste si un sommet Q_i fait partie de l'enveloppe en calculant le produit vectoriel des deux vecteurs (Pile.second, Pile.top) et (Pile.top, Q_i). Finalement, l'algorithme retourne la pile qui contient les sommets de l'enveloppe convexe.

Cette enveloppe convexe est une approximation extérieure de la surface balayée par le véhicule

Algorithme 19 CalculeEnveloppeConvexe(**in** : Points $P_1 \dots P_N$,**out** : Pile)

```
1: Trouver le pivot Q_1 in {P_1 ... P_N} ;
2: Trier les points dans {P_1 ... P_N} \ {Q_1} par angle croissant pour avoir {Q_2 ... Q_N}
3: Pile.push(Q_1) ;
4: Pile.push(Q_2) ;
5: Pour i = 3 à N faire
6:    TantQue ProduitVecoriel(Pile.second, Pile.top, Q_i) < 0 et Pile.size() ≥ 2 faire
7:       Pile.pop() ;
8:    FinTantQue
9:    Pile.push(Q_i) ;
10: FinPour
11: Retourner Pile
```

dont la configuration appartient à un vecteur d'intervalles donné. Nous pouvons donc tester si le véhicule est situé en un état sûr. Il y a en effet collision entre le véhicule et l'environnement s'il existe un segment de la frontière de l'enveloppe convexe qui intersecte un segment de l'environnement ou bien si l'un des segments de l'environnement en contenu dans cette enveloppe.

Il est important de vérifier, avant d'exécuter l'algorithme de planification, que $[\mathbf{x}_{\text{init}}]$ et $[\mathbf{x}_{\text{goal}}]$ sont sûrs. Ce test n'est par contre pas suffisant pour ajouter un nouveau nœud dans l'arbre. En effet, lorsque l'on calcule $[\mathbf{x}_{\text{new}}]$, l'équation d'état doit être intégrée sur un intervalle de temps Δt. Ainsi, il est tout a fait possible qu'une collision intervienne durant cet intervalle de temps bien que $[\mathbf{x}_{\text{near}}]$ et $[\mathbf{x}_{\text{new}}]$ soient sûrs.

7.3.2 Test de collision pour une trajectoire entre deux nœuds consécutifs

Après avoir calculé $[\mathbf{x}_{\text{new}}]$, nous devons vérifier qu'aucune collision n'intervient sur la trajectoire entre $[\mathbf{x}_{\text{near}}]$ et $[\mathbf{x}_{\text{new}}]$ afin d'assurer qu'il peut être ajouté au graphe. C'est le but de la fonction `CollisionFreePath`.

Au paragraphe 4.2.2 page 77, nous présentions une étape de prédiction qui garantissait que tous les états qui composent la trajectoire étaient contenus dans un ensemble $[\tilde{\mathbf{x}}_1]$. Nous allons utiliser ici la même technique d'intégration garantie afin d'obtenir cette approximation externe $[\tilde{\mathbf{x}}_1]$.

Pour trouver $[\tilde{\mathbf{x}}_1]$, nous utilisons la même méthode que dans l'algorithme 13 page 78. Une fois $[\tilde{\mathbf{x}}_1]$ calculé, nous testons s'il est sûr avec la même méthode que pour $[\mathbf{x}_{\text{init}}]$ et $[\mathbf{x}_{\text{goal}}]$ (décrite dans la section 7.3.1) : il faut prendre en compte la forme du véhicule et effectuer les changements de repère nécessaires afin de s'assurer que le véhicule n'entre pas en collision avec l'environnement quel que soit son état dans la trajectoire entre $[\mathbf{x}_{\text{near}}]$ et $[\mathbf{x}_{\text{new}}]$. Si ce test s'avère concluant, nous pourrons garantir qu'aucune collision n'intervient durant le déplacement entre les pavés d'états $[\mathbf{x}_{\text{near}}]$ et $[\mathbf{x}_{\text{new}}]$.

L'algorithme 20 décrit le fonctionnement complet de ce test de collision. La fonction `CollisionFreeState` correspond à la méthode décrite dans le paragraphe 7.3.1.

Maintenant que nous pouvons prouver qu'aucune collision n'intervient entre deux nœuds consécutifs du graphe, nous pouvons en déduire par récurrence que la trajectoire entre $[\mathbf{x}_{\text{init}}]$ et $[\mathbf{x}_{\text{goal}}]$ est, si elle existe, sûre.

7.4 Résultats obtenus avec Box-RRT en robotique

Ce paragraphe présente des trajectoires planifiées en utilisant l'algorithme Box-RRT couplé au modèle de *voiture simple*. Ces résultats ont été obtenus pour $\Delta t = 100$ ms et une résolution des odomètres $r = 1024$. Sur toutes ces figures, bien que l'espace d'état soit de dimension trois, nous ne dessinons que les pavés $[x] \times [y]$ afin de faciliter la visualisation des résultats.

Algorithme 20 CollisionFreePath(**in** : $[\mathbf{x}_{\text{near}}]$, $[\mathbf{x}_{\text{new}}]$, $\mathbf{u}$, Δt)

1: $[\tilde{\mathbf{x}}_1] = [\mathbf{x}_{\text{near}}] \sqcup [\mathbf{x}_{\text{new}}]$
2: **TantQue** $[\mathbf{x}_{\text{near}}] + [0, \Delta t]\mathbf{f}([\tilde{\mathbf{x}}_1], \mathbf{u}) \not\subset [\tilde{\mathbf{x}}_1]$ **faire**
3: $\quad [\tilde{\mathbf{x}}_1] \leftarrow [\tilde{\mathbf{x}}_1] + \epsilon\,[-1, 1]^{\times 3}$
4: **FinTantQue**
5: **Si** CollisionFreeState($[\tilde{\mathbf{x}}_1]$, $\mathbf{u}$, Δt) **alors**
6: $\quad$ **Retourner** VRAI
7: **Sinon**
8: $\quad$ **Retourner** FAUX
9: **FinSi**

7.4.1 Succès

Dans cette section, nous montrons tout d'abord des résultats obtenus à une faible vitesse ($1\ \text{m.s}^{-1}$), nous considérons les glissements comme nuls.

La figure 7.2 illustre la résolution d'un problème de planification simple. L'erreur initiale sur la localisation est ± 10 cm sur x et y. Elle est de ± 3 degrés sur l'orientation θ de la voiture. La zone d'arrivée $[\mathbf{x}_{\text{goal}}]$ est un carré de 10 m$\times$10 m. L'orientation d'arrivée n'a pas d'importance. La distance euclidienne entre $[\mathbf{x}_{\text{init}}]$ and $[\mathbf{x}_{\text{goal}}]$ est d'environ 100 mètres. Ce problème a été résolu après génération en moyenne de 30 000 nœuds.

Sur la figure 7.3, nous considérons le même problème mais avec une zone d'arrivée $[\mathbf{x}_{\text{goal}}]$ réduite à un carré de 5 m$\times$5 m, l'orientation n'ayant toujours pas d'importance. Le problème s'avère plus difficile à résoudre pour le planificateur du fait de la réduction de la zone d'arrivée. Afin de trouver une trajectoire permettant de rejoindre $[\mathbf{x}_{\text{goal}}]$, le planificateur doit générer en moyenne 100 000 nœuds (figure 7.2).

Ces problèmes utilisent le modèle de *voiture simple*, d'autres types de modèles auraient pu être utilisés. Dans la figure 7.4, la trajectoire a été planifiée en utilisant un modèle de voiture ne pouvant tourner que du coté droit comme cela avait été fait avec le Kalman-RRT [47].On pourrait envisager d'utiliser d'autres modèles cinématiques, des modèles dynamiques [67] ou même des chaînes cinématiques [71] comme un bras articulé ou un camion avec plusieurs remorques.

Le planificateur Box-RRT peut résoudre des problèmes plus difficiles, dans des environnements contenant plus d'obstacles que ceux des figures 7.2 à 7.4. Dans la figure 7.5, nous utilisons un environnement en forme de labyrinthe. Dans cet exemple, le Box-RRT trouve une trajectoire sûre malgré un nombre important d'obstacles.

7.4.2 Difficultés rencontrées

Au paragraphe précédent, et notamment lors du passage de la figure 7.2 à la figure 7.3, la taille de $[\mathbf{x}_{\text{goal}}]$ a été réduite. Le problème de planification est devenu plus difficile à résoudre comme l'illustre le nombre de nœuds nécessaires qui passe de 30 000 à 100 000. Si on continue à réduire cette zone d'arrivée, aucune trajectoire ne pourra être trouvée. Ainsi, sur la figure 7.6, le planificateur ne peut trouver de trajectoire bien qu'il en existe peut-être toujours une. Comme nous n'utilisons qu'une phase de prédiction, la taille du pavé contenant les états possibles du système ne cesse de grandir avec le temps.

Le même problème apparaît lorsque la distance entre les pavés d'états de départ $[\mathbf{x}_{\text{init}}]$ et d'arrivée $[\mathbf{x}_{\text{goal}}]$ devient trop importante ou lorsque la borne sur l'erreur due aux glissements devient trop grande. La figure 7.7 illustre ce problème. Sur cette figure, l'erreur due aux glissements est au maximum de 5% sur p_{l} et p_{r}. L'incertitude devient dans ce cas trop importante et nous ne pouvons plus garantir que le véhicule passe dans les couloirs du labyrinthe. Ce problème ne peut donc être

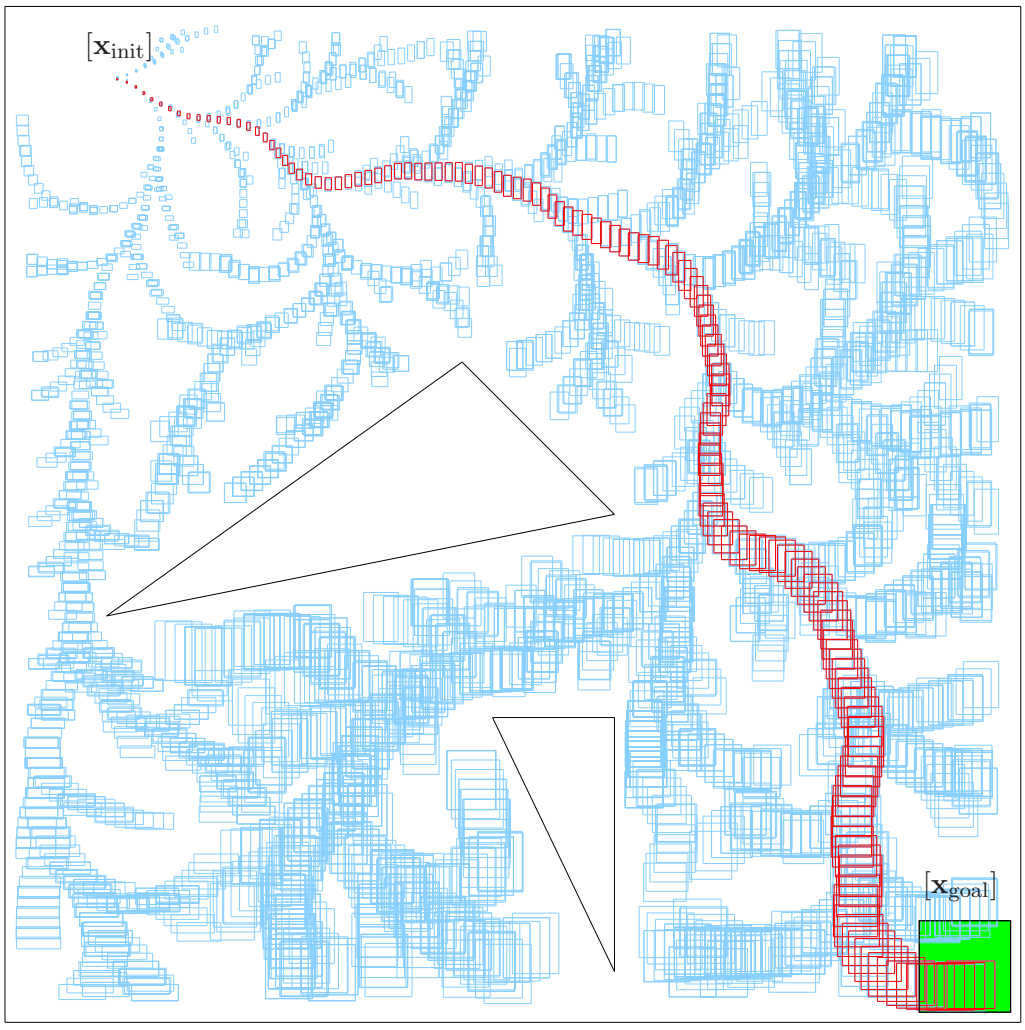


FIG. 7.2 – Trajectoire sûre dans un environnement avec peu d'obstacles et une grande zone d'arrivée

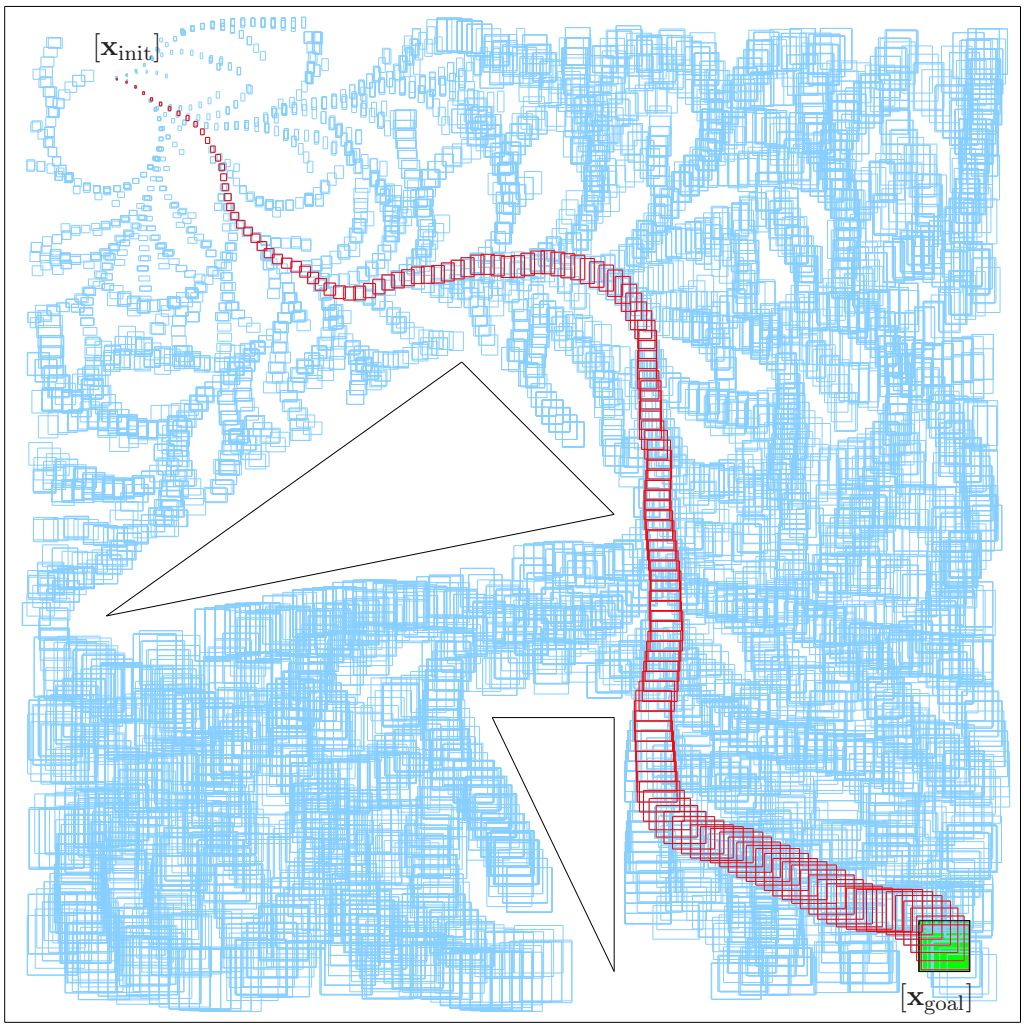


FIG. 7.3 – Trajectoire sûre dans un environnement avec peu d'obstacles et une petite zone d'arrivée

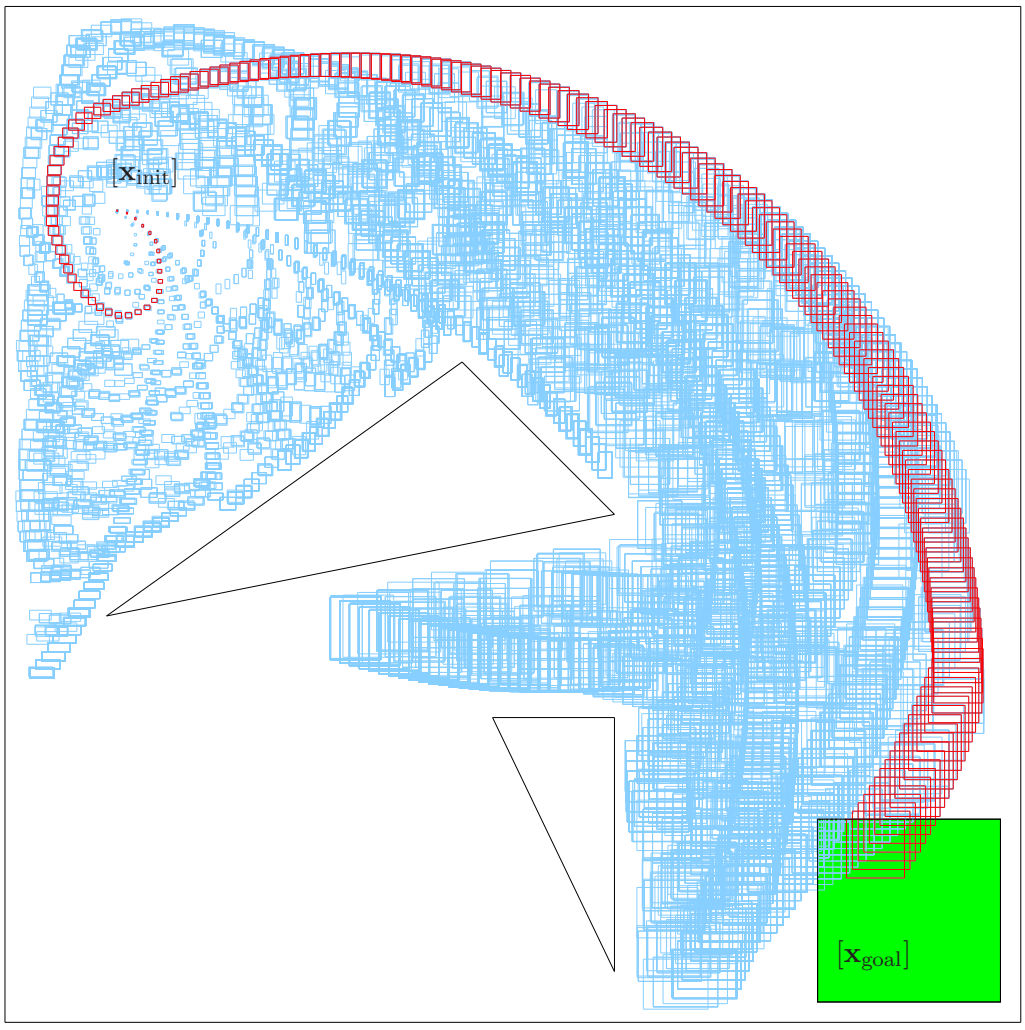


FIG. 7.4 – Trajectoire sûre pour une voiture qui ne peut tourner qu'à droite

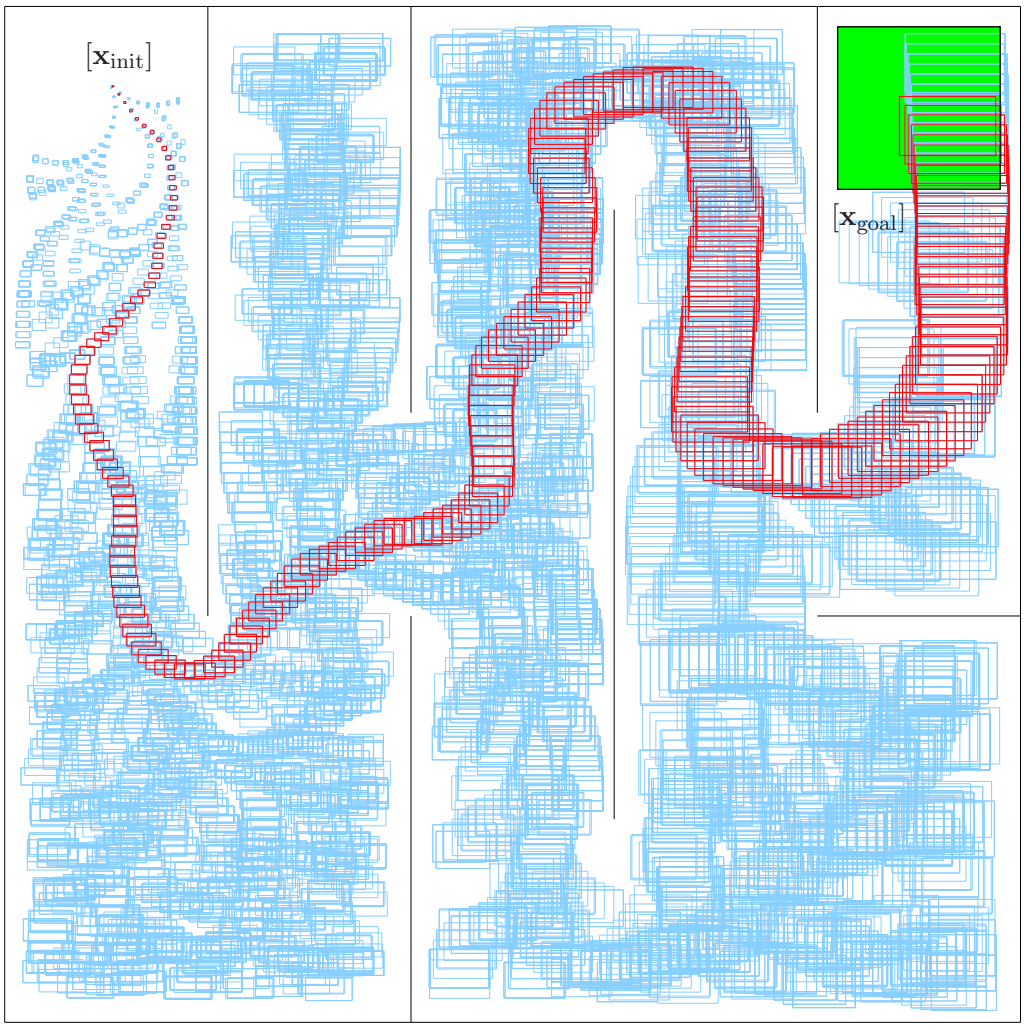


FIG. 7.5 – Trajectoire sûre un environnement en forme de labyrinthe

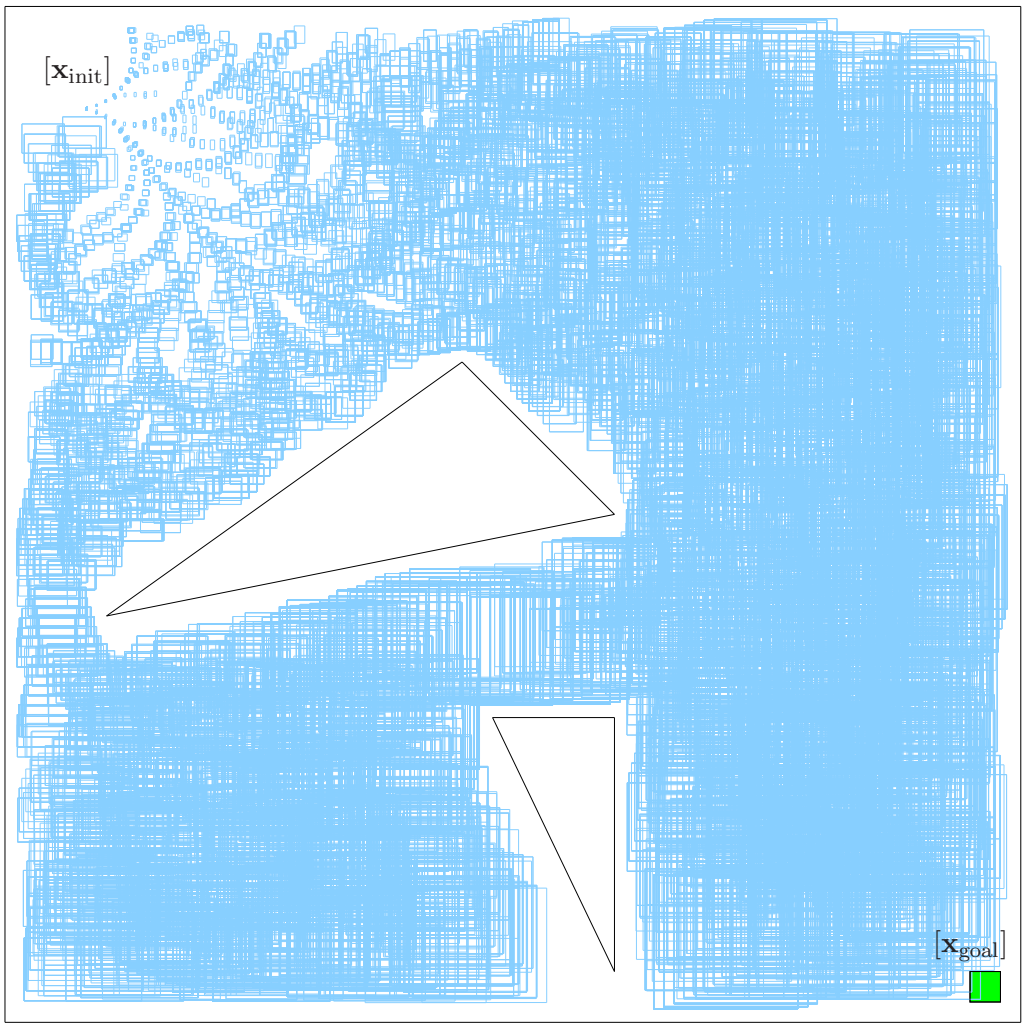


FIG. 7.6 – Aucune trajectoire sûre ne peut être trouvée car la zone d'arrivée est trop petite

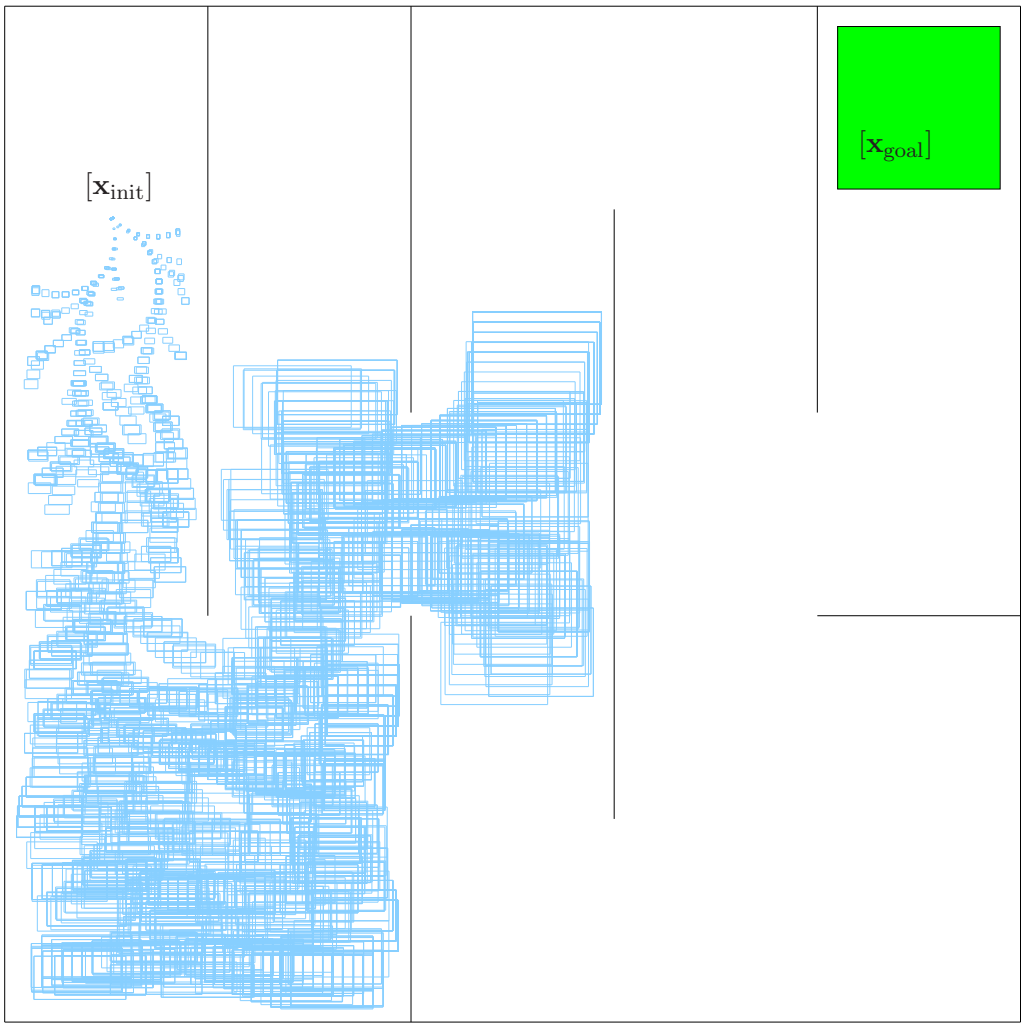


FIG. 7.7 – Aucune trajectoire sûre ne peut être trouvée avec une commande indifférenciée car l'erreur de glissement sur les odomètres est trop importante

résolu en utilisant la version de notre algorithme que nous venons de décrire.

Afin de résoudre ce type de problème, nous allons utiliser la méthode de planification a commandes différenciées présentée dans le paragraphe 4.3 page 81 et ainsi relaxer la contrainte nous imposant une commande unique pour tous les états $\mathbf{x} \in [\mathbf{x}]$ entre t et $t + \Delta t$.

7.5 Planification à commandes différenciées

Le but est de réduire le pavé d'arrivée $[\mathbf{x}_{\text{new}}]$ et de découvrir des trajectoires inaccessibles avec une commande indifférenciée. On découpe pour cela le pavé d'état $[\mathbf{x}_{\text{near}}]$, afin de pouvoir attribuer une commande différente pour chacun de ces sous-pavés.

7.5.1 Mise en œuvre

La mise en œuvre de la planification à commandes différenciées, pour un problème de robotique utilisant un modèle de véhicule, peut se faire en utilisant directement les algorithmes 16 et 17. Cependant, il est possible d'améliorer le procédé de réduction du pavé (algorithme 16 page 81) et notamment de l'accélérer. Pour un modèle de *voiture simple*, avant d'exécuter la fonction de recherche de commandes (algorithme 17 page 83) qui découpe l'ensemble des commandes admissibles, il est possible de supprimer certaines commandes sans intérêt, comme nous allons le voir.

Une fois intégré sur un intervalle de temps Δt, la composante θ du modèle *voiture simple* décrit dans la section 5.1.3 page 92 peut s'écrire :

$$[\theta_{k+1}] = [\theta_k] + \left(\frac{[v_k]}{L} \tan [\delta_k] \right) \Delta t \tag{7.5}$$

avec $[\theta_{k+1}]$ l'intervalle contenant l'orientation du véhicule une fois le système intégré. On peut donc en déduire les deux équations suivantes :

$$[v_k] = \frac{([\theta_{k+1}] - [\theta_k])L}{\tan [\delta_k] \cdot \Delta t}, \tag{7.6}$$

$$[\delta_k] = \arctan \frac{([\theta_{k+1}] - [\theta_k])L}{[v_k] \cdot \Delta t}. \tag{7.7}$$

De même, les composantes x et y du modèle *voiture simple* permettent de déduire

$$[v_k] = \sqrt{([x_{k+1}] - [x_k])^2 + ([y_{k+1}] - [y_k])^2}. \tag{7.8}$$

Soit $[v]$ et $[\delta]$ les intervalles contenant les ensembles des vitesses et angles de braquage admissibles pour le véhicule. L'algorithme 21 permet de réduire ces intervalles et donc de débuter le découpage de l'algorithme 17 avec des intervalles plus petits.

Les deux premières lignes de l'algorithme 21 servent à initialiser les intervalles $[v_{\text{réduit}}]$ et $[\delta_{\text{réduit}}]$ qui prendront comme valeurs respectivement $[v]$ et $[\delta]$. Une fois cette initialisation effectuée, nous utiliserons (7.6), (7.7) et (7.8) successivement afin de réduire $[v_{\text{réduit}}]$ et $[\delta_{\text{réduit}}]$. Cette phase est itérée jusqu'à ce qu'aucun des deux intervalles ne puisse plus être réduit. Les deux intervalles $[v_{\text{réduit}}]$ et $[\delta_{\text{réduit}}]$ sont alors retournés. Le nouveau pavé $[\mathbf{u}_{\text{réduit}}] = [v_{\text{réduit}}] \times [\delta_{\text{réduit}}]$ est utilisé comme point de départ de l'algorithme de recherche de commandes (algorithme 17).

7.5.2 Résultats

L'utilisation de commandes différenciées dans le cas de la robotique et d'un modèle de *voiture simple* permet de réduire significativement la taille du pavé $[\mathbf{x}_{\text{new}}]$ à chaque exécution. La réduction

Algorithme 21 RéductionCommande(**in** : $[\mathbf{x}_{\text{near}}]$, $[\mathbf{x}_{\text{new}}]$, $[v]$, $[\delta]$, **out** : $[v_{\text{réduit}}]$, $[\delta_{\text{réduit}}]$)

1: $[v_{\text{réduit}}] \leftarrow [v]$
2: $[\delta_{\text{réduit}}] \leftarrow [\delta]$
3: **Répéter**
4: estRéduit $\leftarrow$ FAUX
5: $[v_{\text{temp}}] \leftarrow \frac{([\theta_{k+1}]-[\theta_k])L}{\tan[\delta_{\text{réduit}}].\Delta t}$
6: **Si** $[v_{\text{réduit}}] \cap [v_{\text{temp}}] \neq \emptyset$ **alors**
7: **Si** $[v_{\text{réduit}}] \neq [v_{\text{réduit}}] \cap [v_{\text{temp}}]$ **alors**
8: $[v_{\text{réduit}}] \leftarrow [v_{\text{réduit}}] \cap [v_{\text{temp}}]$
9: estRéduit $\leftarrow$ VRAI
10: **FinSi**
11: **FinSi**
12: $[v_{\text{temp}}] \leftarrow \sqrt{([x_{k+1}]-[x_k])^2 + ([y_{k+1}]-[y_k])^2}$
13: **Si** $[v_{\text{réduit}}] \cap [v_{\text{temp}}] \neq \emptyset$ **alors**
14: **Si** $[v_{\text{réduit}}] \neq [v_{\text{réduit}}] \cap [v_{\text{temp}}]$ **alors**
15: $[v_{\text{réduit}}] \leftarrow [v_{\text{réduit}}] \cap [v_{\text{temp}}]$
16: estRéduit $\leftarrow$ VRAI
17: **FinSi**
18: **FinSi**
19: $[\delta_{\text{temp}}] \leftarrow \arctan \frac{([\theta_{k+1}]-[\theta_k])L}{[v_{\text{réduit}}].\Delta t}$
20: **Si** $[\delta_{\text{réduit}}] \cap [\delta_{\text{temp}}] \neq \emptyset$ **alors**
21: **Si** $[\delta_{\text{réduit}}] \neq [\delta_{\text{réduit}}] \cap [\delta_{\text{temp}}]$ **alors**
22: $[\delta_{\text{réduit}}] \leftarrow [\delta_{\text{réduit}}] \cap [\delta_{\text{temp}}]$
23: estRéduit $\leftarrow$ VRAI
24: **FinSi**
25: **FinSi**
26: **Jusqu'à** estRéduit = FAUX
27: **Retourner** $[v_{\text{réduit}}]$, $[\delta_{\text{réduit}}]$

du pavé est en moyenne de 17% sur chacun des intervalles qui le compose.

Pour arriver à de tels taux de réduction, il faut découper $[\mathbf{x}_{\text{near}}]$ en au moins 64 sous-pavés. Pour chaque nouveau test de réduction du pavé $[\mathbf{x}_{\text{new}}]$, il faut en moyenne 20 secondes afin de trouver un ensemble de commandes permettant de le rejoindre. Ce test ralentit donc le processus de planification et ne peut être utilisé pour chaque ajout de nœuds dans l'arbre. On peut par exemple imaginer réduire la taille de l'intervalle toutes les secondes sur une trajectoire.

Les résultats illustrés sur les figures 7.8 et 7.9 montrent comment l'utilisation des commandes différenciées permet de réduire les pavés et ainsi prouver que des solutions sûres existent pour le problème illustré sur la figure 7.7. Sur les deux figures 7.8 et 7.9, le tirage aléatoire est celui du RRT *Goalbias* avec $p = 0.1$. Sur la figure 7.8, des commandes différenciées sont utilisées toutes les secondes. Le planificateur arrive à trouver une trajectoire en générant environ 10 000 nœuds. De la même façon, une trajectoire est trouvée sur la figure 7.9 avec l'utilisation de commandes différenciées toutes les 2 secondes. Le nombre de nœuds générés est alors plus important (18 000).

Bien que cette méthode permette de prouver l'existence d'une trajectoire sûre, les trajectoires générées ne peuvent être utilisées durant la navigation du véhicule. En effet, afin de pouvoir choisir la commande réelle à appliquer au véhicule, il faut connaître précisément son état ou au moins dans quel sous-ensemble de $[\mathbf{x}_{\text{near}}]$ se situe cet état.

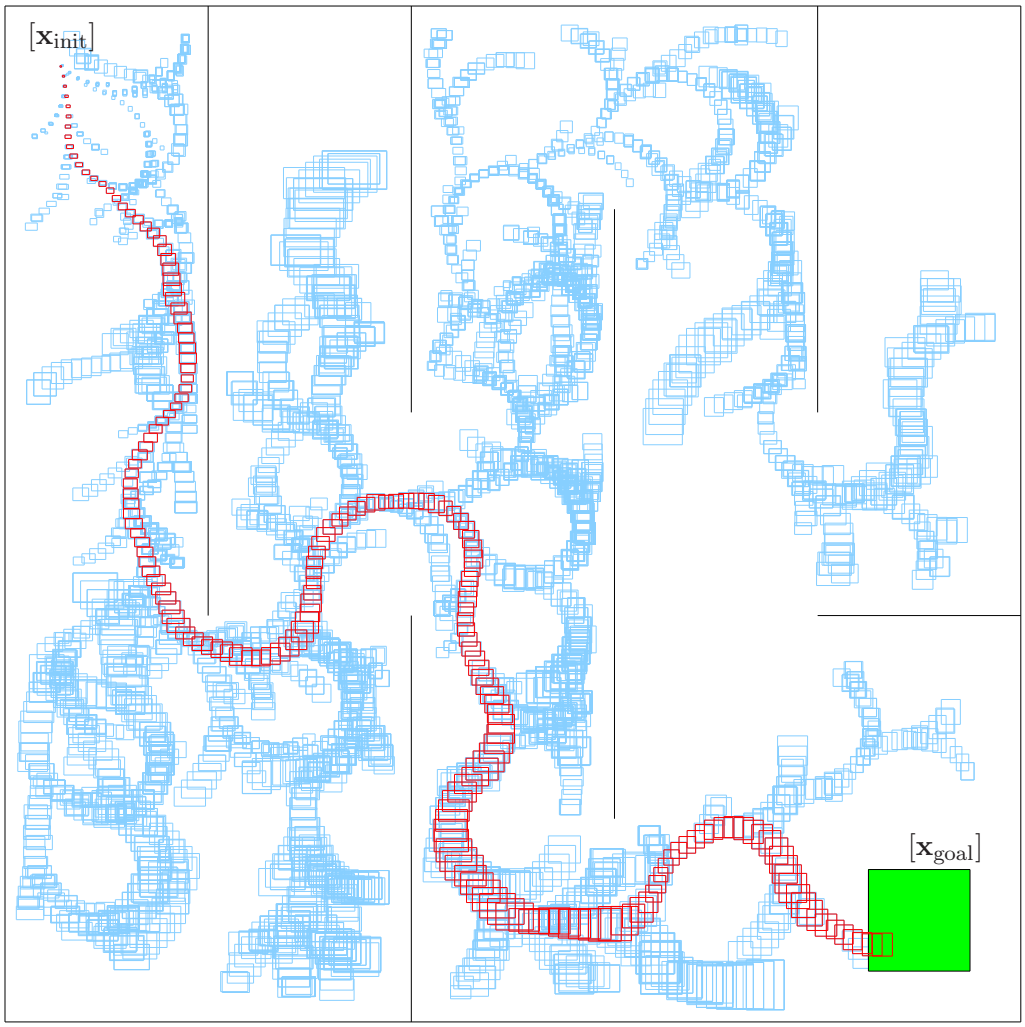


FIG. 7.8 – Trajectoire sûre trouvée pour le problème de la figure 7.7 en utilisant des commandes différenciées toutes les secondes

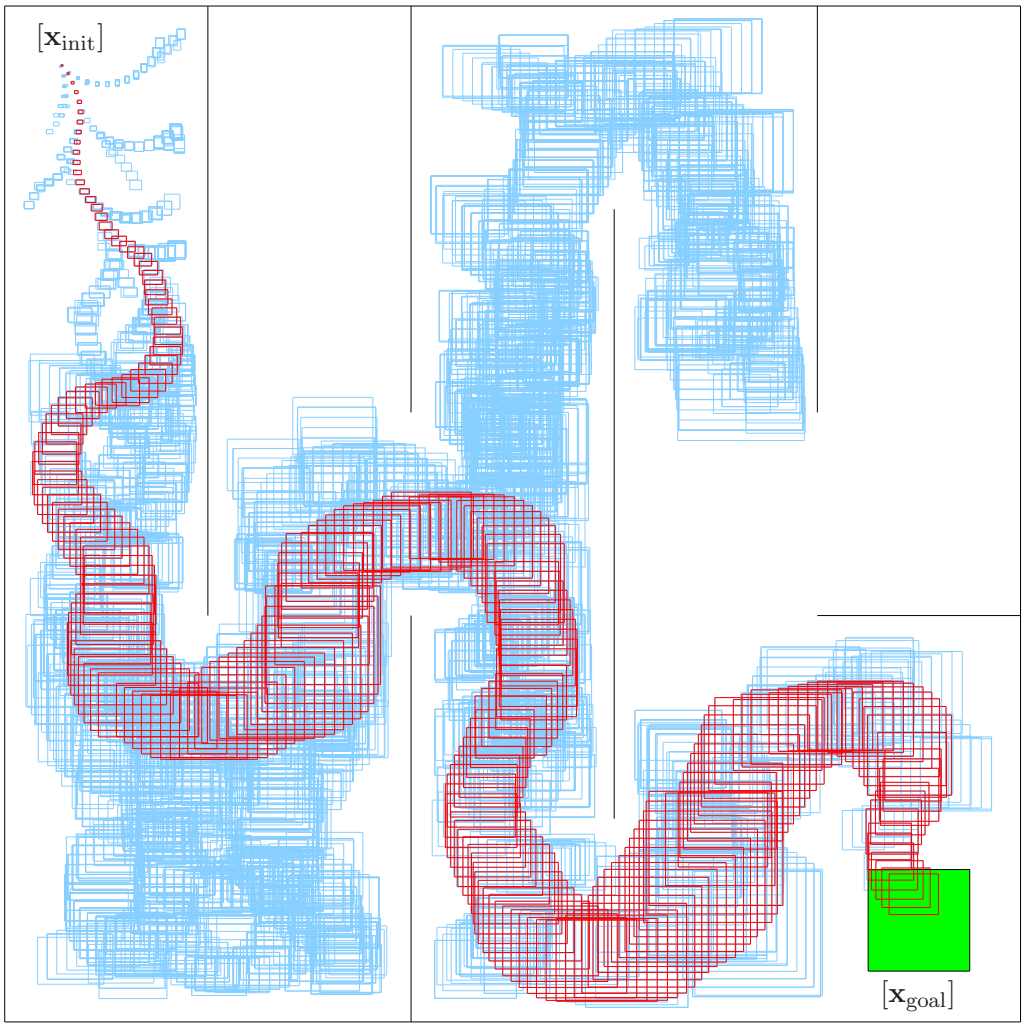


FIG. 7.9 – Trajectoire sûre trouvée pour le problème de la figure 7.7 en utilisant des commandes différenciées toutes les 2 secondes

7.6 Conclusion

Nous avons présenté dans ce chapitre une application à la robotique de l'algorithme de planification ensembliste Box-RRT introduit au chapitre 4. Cet algorithme permet dans ce contexte de trouver des trajectoires qui garantissent la sûreté du véhicule. Ceci est possible notamment grâce à l'utilisation d'une phase de prédiction (sans phase de correction) et d'une méthode d'intégration garantie.

Le fait d'utiliser uniquement une phase de prédiction induit une constante augmentation de l'incertitude. Certains problèmes ne peuvent donc pas être résolus : par exemple ceux où la distance entre $[\mathbf{x}_{\text{init}}]$ et $[\mathbf{x}_{\text{goal}}]$ est importante, ceux où la zone d'arrivée est de taille trop faible ou ceux pour lesquels les erreurs induites par les capteurs proprioceptifs sont trop importantes.

Une solution à ce problème a été trouvée en relaxant l'exigence d'une suite de commandes identique pour chacun des états contenus dans le pavé d'états initial. L'utilisation de plusieurs commandes pour des valeurs de l'état appartenant à un même pavé permet de réduire considérablement l'incertitude sur la phase de prédiction. Ceci rend soluble de nombreux problèmes qui ne l'étaient pas.

Conclusions et perspectives

Nous avons traité dans ce mémoire du problème de planification de trajectoires, en concentrant notre attention sur le cas où l'état du système à déplacer n'est connu que de façon imprécise. La gestion de cette imprécision durant la phase de planification permet ainsi d'anticiper les problèmes éventuels qui pourraient survenir lorsque le système suit le plan.

Dans les deux nouveaux planificateurs que nous proposons, nous avons fait le choix d'utiliser un algorithme de planification moderne, le RRT, qui permet une exploration rapide et efficace de l'espace d'état sans recourir à une discrétisation préalable.

Deux approches ont été suivies. Dans la première, la planificateur utilise une représentation probabiliste de l'état du système à un instant donné, par une densité de probabilité gaussienne. La propagation des erreurs est effectuée en utilisant un filtre de Kalman étendu. Le planificateur résultant, que nous avons appelé Kalman-RRT, utilise également une phase de correction afin de diminuer l'incertitude sur l'état du système en utilisant d'éventuelles mesures issues de capteurs extéroceptifs. Nous avons décidé de simuler les capteurs depuis l'espérance de l'état pour nous permettre de réduire les incertitudes lors de cette étape de correction. La diminution des incertitudes grâce à l'utilisation de mesures permet de rejoindre des états qui n'étaient pas atteignables en enchaînant uniquement des étapes de prédiction.

L'utilisation d'un modèle probabiliste gaussien afin de représenter l'incertitude ne permet pas de garantir la sûreté du système le long de la trajectoire puiqu'aucune position n'est impossible. Le test de collision utilise en effet un niveau de confiance que l'on doit choisir au préalable et qui est calculé à l'aide d'approximations discutables.

Les trajectoires générées en utilisant Kalman-RRT ne peuvent donc pas être considérées comme totalement sûres mais elles suggèrent cependant des zones de l'environnement à explorer et d'autres à éviter.

La deuxième approche que nous privilégions avec les algorithmes Set-RRT et Box-RRT est très différente. Dans le cas du Set-RRT, nous englobons les états que peut prendre le système à un instant donné compte tenu de bornes sur les erreurs commises dans un ensemble calculable. Contrairement à l'approche probabiliste précédente, cette approche permet de fournir une garantie sur la sûreté du système à condition bien sûr que les hypothèses sur les bruits d'états qui la fondent soient satisfaites. L'algorithme Box-RRT est une mise en œuvre de cet algorithme utilisant l'analyse par intervalles et approximant les ensembles d'états par des pavés (ou vecteurs d'intervalles).

La sûreté de la trajectoire est assurée par l'utilisation d'une méthode garantie pour intégrer l'équation d'état incertaine et prédire ainsi les états futurs. Conjointement à cette intégration garantie, un pavé englobant est utilisé pour assurer que la trajectoire entre deux états consécutifs ne comporte

aucun risque.

Pour ne pas commettre d'approximations lors de la simulation des capteurs extéroceptifs, seules une phase de prédiction est utilisée par le planificateur. Il en résulte une augmentation continuelle de la taille des pavés contenant l'état du système, à cause des accumulations d'erreurs successives. Ainsi, certains problèmes ne peuvent pas être résolus, notamment lorsque l'erreur sur les capteurs proprioceptifs est importante où lorsque le système doit effectuer un grand nombre de changements d'état.

Afin de dépasser partiellement cette limitation, une des contraintes du problème a été relaxée. Au lieu de chercher une suite unique de commandes permettant à tous les états contenus dans l'ensemble de départ de rejoindre l'arrivée, nous avons décidé d'autoriser des suites de commandes différentes pour chacun de ces états. Cela a permis de réduire la taille des ensembles contenant les états et ainsi de trouver des solutions admissibles à certains problèmes de planification autrement insolubles.

Ces deux techniques de planification ont été mises en œuvres dans le contexte de la robotique mobile et plus précisément pour résoudre des problèmes de planification de trajectoire pour un véhicule de type voiture circulant dans un environnement contenant des obstacles à éviter.

Lors de leur utilisation en robotique, ces algorithmes doivent être adaptés afin de pouvoir gérer un système non ponctuel. Ainsi, la zone de l'espace où l'état ne peut être situé est définie en fonction du positionnement des obstacles qui composent l'environnement mais également en fonction de la forme du véhicule. La définition d'un tel ensemble s'avère déjà ardue lorsque les incertitudes ne sont pas prises en compte. L'ajout des incertitudes rend cette phase extrêmement délicate. Pour résoudre ce problème, nous avons défini un test de collision pour le véhicule, que ce soit lors de l'utilisation du Kalman-RRT ou du Box-RRT.

Les résultats obtenus illustrent la capacité des deux algorithmes à trouver des trajectoires dans des environnements variés malgré la contrainte de non-holonomie et l'incertitude sur la position du véhicule.

Perspectives

Plusieurs extensions des planificateurs présentés sont envisageables à court ou moyen termes. Le principal problème se situe dans le découplage actuel entre les techniques de planification et de localisation, notamment dans le choix de la commande à utiliser. Que ce soit dans Kalman-RRT ou dans Box-RRT, les informations sur l'état actuel du système ne sont pas utilisées dans le choix de la commande. Ces informations pourraient cependant s'avérer utiles afin de privilégier certaines trajectoires par rapport à d'autres en fonction de la valeur de l'incertitude.

Dans l'algorithme Box-RRT, lorsque nous résolvons le problème de planification classique, l'utilisation d'une phase de prédiction seule implique que l'ensemble contenant les états du système ne cesse de grandir. Il serait donc possible de comparer la taille de cet ensemble à la taille de l'ensemble d'arrivée afin d'élaguer les branches de l'arbre du RRT. Ainsi, si un ensemble d'états est de taille supérieure à la taille de l'ensemble d'arrivée, le nœud ne sera d'aucune utilité pour résoudre le problème. Il serait alors possible de supprimer la branche de l'arbre contenant ce nœud pour qu'il ne soit plus pris en compte dans la recherche du plus proche voisin. Cela aurait pour conséquence de réduire le temps de calcul nécessaire à la résolution du problème ainsi que l'empreinte mémoire du

planificateur.

Dans la modification du problème de planification que nous proposons afin de réduire l'incertitude avec Box-RRT les fonctions de sélection de la commande et de réduction des ensembles sont complètement indépendantes. Une troisième perspective serait de les utiliser conjointement afin de choisir une commande qui permettrait d'augmenter le facteur de réduction.

A plus long terme, il serait intéressant de coupler ces méthodes à des techniques de planification dans des environnements dynamiques. Les travaux actuels simulent des déplacements dans des environnements où certaines parties sont en mouvements sans prendre en compte les incertitudes à la fois sur le véhicule à déplacer ni sur les trajectoires des obstacles mouvants.

Remarquons enfin que les nombreuses implémentations d'algorithmes de planification au sein de véhicules réels (par exemple ceux utilisés lors de l'*Urban Challenge* organisé par le DARPA) utilisent très souvent les valeurs retournées par les capteurs sans considérer le bruit sur ces mesures. Les algorithmes utilisés étant très souvent à base de RRT, il serait possible d'y adjoindre notre gestion de l'incertitude afin de générer des trajectoires et ainsi vérifier expérimentalement la sûreté des trajectoires retournées par nos planificateurs.

Références bibliographiques

[1] T. Lengauer et M. Rarey. Computational methods for biomolecular docking. *Current Opinion in Structural Biology*, 6(3) :402–406, 1996.

[2] Jorgensen W. L. Rusting of the lock and key model for protein-ligand binding. *Science*, 254(5034) :954–955, 1991.

[3] Morris G. M., Goodsell D. S., Halliday R. S., Huey R., Hart W. E., Belew R. K., et Olson A. J. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, 19(14) :1639–1662, 1998.

[4] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[5] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9) :569, 1965.

[6] P. Svestka et M. H. Overmars. Probabilistic Path Planning. Rapport technique UU-CS-1995-22, Institute of Information and Computing Sciences, Utrecht University, 1995.

[7] P. Svestka et M.H. Overmars. Motion planning for car-like robots, a probabilistic learning approach. 6 :119–143, 1997.

[8] M. H. Overmars et P. Svestka. A probabilistic learning approach to motion planning. In *WAFR : Proc. Workshop on Algorithmic foundations of robotics*, pages 19–37, Natick, MA, USA, 1995. A. K. Peters, Ltd.

[9] L. E. Kavraki, P. Svestka, J. C. Latombe, et M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *International Trans. Robotics and Automation*, 12(4) :566–580, June 1996.

[10] L. E. Kavraki et J. C. Latombe. Probabilistic roadmaps for robot path planning. In K. Gupta et A. del Pobil, editors, *Practical Motion Planning in Robotics : Current Approaches and Future Challenges, John Wiley, West Sussex, UK*, pages 33–53, 1998.

[11] J. Barraquand et J.-C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1712–1717, 1990.

[12] J. Barraquand et J.-C. Latombe. Robot motion planning : A distributed representation approach. *International Journal of Robotics Research*, 10(6) :628–649, 1991.

[13] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, USA, 1991.

[14] E. Mazer, G. Talbi, J. M. Ahuactzin, et P. Bessiere. The Ariadne's clew algorithm. In *Proc.of the International Conference on Simulation of Adaptive Behavior SAB92 : From animals to animats*, pages 7–11, Honolulu, Hawaii, 1992.

[15] E. Mazer, J. M. Ahuactzin, et P. Bessiere. The Ariadne's clew algorithm. *Journal of Artificial Intelligence Research*, 9 :295–316, November 1998.

[16] D. Hsu, J.-C. Latombe, et R. Motwani. Path planning in expansive configuration spaces. *International Journal of Computationnal Geometry & Applications*, 4 :495–512, 1999.

[17] G. Sanchez et J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proc. International Symposium on Robotics Research*, pages 403–417, Lorne, Victoria, Australia, 2001.

[18] S. Carprin et G. Pillonetto. Robot motion planning using adaptive random walks. *IEEE Transactions on Robotics and Automation*, 21(1) :129–136, 2005.

[19] S. Carprin et G. Pillonetto. Merging the adaptive random walks planner with the randomized potentiel field planner. In *Proc. IEEE International Workshop on Robot Motion and Control*, pages 151–156, 2005.

[20] S. M. LaValle. Rapidly-exploring Random Trees : A new tool for path planning. Rapport technique, Computer Science Dept., Iowa State University, November 1998.

[21] S. M. LaValle et J. J. Kuffner. Rapidly-exploring Random Trees : progress and prospects. In B. R. Donald, K. M. Lynch, et D. Rus, editors, *Algorithmic and Computational Robotics : New Directions*, pages 293–308. A. K. Peters, Wellesley, MA, 2001.

[22] S. M. LaValle et J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE International Conference on Robotics and Automation*, Detroit, USA, 1999.

[23] J. J. Kuffner et S. M. LaValle. RRT-connect : An efficient approach to single-query path planning. In *Proc. IEEE International Conference on Robotics and Automation*, pages 995–1001, San Francisco, USA, 2000.

[24] P. Cheng, Z. Shen, et S. M. LaValle. RRT-based trajectory design for autonomous automobiles and spacecraft. *Archives of Control Sciences*, 11(3-4) :167–194, 2001.

[25] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K.Cambridge, U.K., 2006. Available at http ://planning.cs.uiuc.edu/.

[26] A. Lazanas et J. C. Latombe. Motion planning with uncertainty : A landmark approach. *Artificial Intelligence*, 76(1-2) :287–317, 1995.

[27] Th. Fraichard et R. Mermond. Path planning with uncertainty for car-like robots. In *Proc. IEEE International Conference on Robotic and Automation*, pages 27–32, Leuven, Belgium, 1998.

[28] R. Alami et T. Simeon. Planning robust motion strategies for a mobile robot. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1312–1318, San Diego, CA, USA, 1994.

[29] B. Bouilly, T. Simeon, et R. Alami. A numerical technique for planning motion strategies of a mobile robot in presence of uncertainty. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1327–1332, Nagoya, Japan, 1995.

[30] H. Takeda et J. C. Latombe. Sensory uncertainty field for mobile robot navigation. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2465–2472, 1992.

[31] H. Takeda, C. Facchinetti, et J. C. Latombe. Planning the motions of a mobile robot in a sensory uncertainty field. In *Proc. IEEE transactions on Pattern Analysis and Machine Intelligence*, pages 1002–1017, October 1994.

[32] N. Roy et S. Thrun. Coastal navigation with a mobile robot. In *Proc. of Conference on Neural Information Processing Systems*, pages 1043–1049, 1999.

[33] N. Roy, W. Bugard, D. Fox, et S. Thrun. Coastal navigation - mobile robot navigation with uncertainty in dynamic environments. In *Proc. IEEE Conference on Robotics and Automation*, pages 35–40, 1999.

[34] A. Adam, E. Rivlin, et I. Shimshoni. Computing the sensory uncertainty field of a vision-basedlocalization sensor. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2993–2999, 2000.

[35] A. Lambert et D. Gruyer. Safe path planning in an uncertain-configuration space. In *Proc. IEEE International Conference on Robotics and Automation*, pages 4185–4190, Taipei, Taiwan, 2003.

[36] A. Lambert et Th. Fraichard. Landmark-based safe path planning for car-like robots. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2046–2051, 2000.

[37] N. Melchior et R. Simmons. Particle RRT for path planning with uncertainty. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1617–1624, 2007.

[38] J. P. Gonzalez et A. Stentz. Planning with uncertainty in position : An optimal and efficient planner. In *Proc. IEEE International Conference on Intelligent Robots and Systems*, pages 2435–2442, Edmonton, Canada, August 2005.

[39] J. P. Gonzalez et A. Stentz. Planning with uncertainty in position using high-resolution maps. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1015–1022, Roma, Italia, April 2007.

[40] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME, Journal of Basic Engineering*, 82 :35–45, Mars 1960.

[41] E. Walter et L. Pronzato. *Identification of Parametric Models from Experimental Data*. Springer-Verlag, London, 1997.

[42] R. C. Smith et P. Cheeseman. On the representation and estimation of space uncertainty. *International Journal of Robotics Research*, 5(4) :56–68, 1986.

[43] L. Danzer, B. Grunbaum, et V. Klee. Helly's theorem and its relatives. In *Proc. of Symposia in Pure Mathematics*, pages 102–180. American Mathematical Society, 1963.

[44] M. Grotschel, L. Lovasz, et A. Schrijver. Geometric algorithms and combinatorial optimization. In *Algorithms and Combinatorics*, volume 2. Springer-Verlag, 1988.

[45] F. John. Extremum problems with inequalities as subsidiary conditions. In J. Moser, editor, *Fritz John, Collected Papers*, pages 543–560. Birkhauser, Boston, Massachussets, USA, 1985.

[46] L. Vanderberghe, S. Boyd, et S. Wu. Determinant maximization with linear matrix inequality constraints. *Siam Journal on Matrix Analysis and Applications*, 19(2) :499–533, 1998.

[47] R. Pepy et A. Lambert. Safe path planning in an uncertain-configuration space using RRT. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5376–5381, Beijing, China, 2006.

[48] F. C. Schweppe. *Uncertain Dynamic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[49] T. Alamo, J.M. Bravo, E.F. Camacho, et U. de Sevilla. Guaranteed state estimation by zonotopes. In *Proc. 42nd Conference on Decision and Control*, pages 1035–1043, Hawaii, USA, 2003.

[50] R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, PA, 1979.

[51] M. Kieffer, L. Jaulin, et E. Walter. Guaranteed recursive nonlinear state bounding using interval analysis. *International Journal of Adaptative Control and Signal Processing*, 6(3) :193–218, 2002.

[52] M. Kieffer, L. Jaulin, I. Braems, et E. Walter. Guaranteed set computation with subpavings. In W. Kraemer et J. W. von Gudenberg, editors, *Scientific Computing, Validated Numerics, Interval Methods*, pages 167–178, Boston, 2001.

[53] R. Pepy, M. Kieffer, et E. Walter. Reliable robust path planner. In *Proc. IEEE/RSJ International Conference on Robots and Systems*, pages 1655–1660, Nice, France, 2008.

[54] M. Berger. *Espaces euclidiens, triangles, cercles et sphères*, volume 2 of *Géométrie*. Cedic/Fernand Nathan, Paris, France, 1979.

[55] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, 1966.

[56] R. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E. Kaucher, U. Kulisch, et Ch. Ullrich, editors, *Computer Arithmetic : Scientific Computation and Programming Languages*, pages 255–286. BG Teubner, Stuttgart, 1987.

[57] J. P. Laumond. *Robot Motion Planning and Control*. Springer, Berlin, 1998.

[58] J. A. Reeds et L.A. Shepp. Optimal path for a car that goes both forwards and backwards. *Pacific J. Math.*, 145(2) :367–393, 1990.

[59] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed and terminal position tangents. *American Journal of Mathematics*, (79) :497–516, 1957.

[60] A. Scheuer et C. Laugier. Planning sub-optimal and continuous-curvature paths for car-like robots. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 25–31, Grenoble, France, 1997.

[61] G. Baille, P. Garnier, H. Mathieu, et R. Pissard-Gibollet. Le cycab de l'inria rhône-alpes. Rapport technique, Institut National de Recherche en Informatique et Automatique, 1999.

[62] S. Bouaziz, M. Fan, A. Lambert, T. Maurin, et R. Reynaud. Picar : experimental platform for road tracking applications. In *Proc. IEEE Intelligent Vehicle Symposium*, pages 495–499, Colombus, OH, USA, June 2003.

[63] S. Taheri. *An Investigation and Design of Slip Control Braking Systems Integrated with Four Wheel Steering*. Thèse de doctorat, Clemson University, 1990.

[64] T. Gillespie. *Fundamentals of Vehicle Dynamics*. SAE, 1992.

[65] J.Y. Wong. *Theory of Ground Vehicles*. John Wiley and Sons, New York, 1993.

[66] R. Pepy, A. Lambert, et H. Mounier. Path planning using dynamic vehicle model. In *Proc. IEEE International Conference on Information and Communication Technologies : from Theory to Applications*, pages 781–786, Damascus, Syria, 2006.

[67] R. Pepy, A. Lambert, et H. Mounier. Reducing navigation errors by planning with realistic vehicle model. In *Proc. IEEE Intelligent Vehicle Symposium*, pages 300–307, Tokyo, Japan, 2006.

[68] F. Chenavier et J. L. Crowley. Position estimation for a mobile using vision and odometry. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2588–2593, Nice, France, 1992.

[69] A. Lambert. *Planification de tâches sûres pour robot mobile par prise en compte des incertitudes et utilisation de cartes locales*. Thèse de doctorat, Université de Technologie de Compiègne (UTC), 1998.

[70] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4) :132–133, 1972.

[71] J. Yakey, S. M. LaValle, et L. E. Kavraki. Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6) :951–958, 2001.

Troisième partie

Annexes

Annexe A

Obtention de l'ellipsoïde représentant l'incertitude sur l'état du système

La forme générale d'une densité de probabilité gaussienne en dimension n est la suivante :

$$P(\mathbf{x}) = \frac{1}{(\sqrt{2\pi})^n \sqrt{\det \mathbf{P}}} e^{-\frac{1}{2}(\mathbf{x}-\hat{\mathbf{x}})^{\mathrm{T}} \mathbf{P}^{-1} (\mathbf{x}-\hat{\mathbf{x}})}, \tag{A.1}$$

où $\hat{\mathbf{x}}$ est la moyenne et $\mathbf{P}$ la matrice de covariance.

Dans certains cas, notamment pour pouvoir prendre des décisions sur la sûreté de l'état, il est nécessaire d'utiliser des lignes de niveau de cette densité (qui sont des ellipsoïdes). À chaque ellipsoïde est associé une probabilité que $\mathbf{x}$ soit contenu dedans. Chaque ellipsoïde a pour équation

$$(\mathbf{x} - \hat{\mathbf{x}})^{\mathrm{T}} P^{-1} (\mathbf{x} - \hat{\mathbf{x}}) = k^2, \tag{A.2}$$

où k est une constante définie par le niveau de confiance choisi. La relation entre k et la probabilité p que l'état soit contenu dans l'ellipsoïde est [42]

$$n = 1; \quad p = -\frac{1}{\sqrt{2\pi}} + 2\mathrm{erf}(k) \tag{A.3}$$

$$n = 2; \quad p = 1 - e^{-\frac{k^2}{2}} \tag{A.4}$$

$$n = 3; \quad p = -\frac{1}{\sqrt{2\pi}} + 2\mathrm{erf}(k) - \sqrt{\frac{2}{k}} k e^{-\frac{k^2}{2}} \tag{A.5}$$

où n est la dimension du vecteur d'état et erf est la fonction d'erreur de Gauss.

Cas particulier du modèle de *voiture simple*

Afin d'évaluer l'incertitude sur les coordonnées du point de référence du véhicule dans le plan $\{x, y\}$, on utilise la procédure décrite dans [42]. On extrait de la matrice de covariance en trois dimensions la sous-matrice 2×2 associée à x et y. On obtient ainsi la matrice d'incertitude sur la position

$$\mathbf{P}_{2\times 2} = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}, \tag{A.6}$$

où ρ est le coefficient de corrélation de x et y.

Cela correspond a une ellipse dont l'équation, donnée par (A.2), peut encore s'écrire

$$Ax^2 + Bxy + Cy^2 = k^2 \tag{A.7}$$

avec

$$A = \frac{1}{(1-\rho^2)\sigma_x^2} \tag{A.8}$$

$$B = \frac{1}{(1-\rho^2)\sigma_x\sigma_y} \tag{A.9}$$

$$C = \frac{1}{(1-\rho^2)\sigma_y^2}. \tag{A.10}$$

Le coefficient k est obtenu en utilisant (A.4) en fonction de la probabilité p choisie comme niveau de confiance

$$k = \sqrt{-2\ln(1-p)}. \tag{A.11}$$

Par exemple, pour un niveau de confiance de 90%, on prend $k = 2.146$.

Afin de pouvoir afficher et utiliser cette ellipse, on utilise ses paramètres de demi grand axe a, de demi petit axe b, ainsi que son orientation φ

$$a = \sqrt{\frac{2k^2}{A + C - \sqrt{A^2 + C^2 - 2AC + B^2}}}, \tag{A.12}$$

$$b = \sqrt{\frac{2k^2}{A + C + \sqrt{A^2 + C^2 - 2AC + B^2}}}, \tag{A.13}$$

$$\varphi = \frac{1}{2}\arctan\left(\frac{B}{A-C}\right). \tag{A.14}$$

Annexe B

Test de collision entre une ellipse et l'environnement

Une collision entre un segment

$$\mathbf{p} = \mathbf{p}_0 + t\mathbf{v},\ 0 \leqslant t \leqslant 1$$

et une ellipse

$$(\mathbf{x} - \hat{\mathbf{x}})^{\mathrm{T}}\mathbf{M}(\mathbf{x} - \hat{\mathbf{x}}) = 1,$$

existe lorsqu'il y a un ou plusieurs points d'intersection entre eux ou bien lorsque le segment est inclus dans l'ellipse.

Pour déterminer s'il y a des points d'intersection, il suffit de résoudre en t

$$((\mathbf{p}_0 + t\mathbf{v}) - \hat{\mathbf{x}})^{\mathrm{T}}\,\mathbf{M}\,((\mathbf{p}_0 + t\mathbf{v}) - \hat{\mathbf{x}}) = 1, \tag{B.1}$$

qu'on peut écrire sous la forme

$$at^2 + ct + c = 1 \tag{B.2}$$

Il y a un point d'intersection entre l'ellipse et l'environnement si et seulement si il existe une solution réelle de (B.1) telle que $t \in \mathbb{R}$ et $0 \leqslant t \leqslant 1$.

Nous devons également regarder si les deux extrémités du segment sont dans l'ellipse, c'est à dire

$$(\mathbf{p}_0 - \hat{\mathbf{x}})^{\mathrm{T}}\mathbf{M}(\mathbf{p}_0 - \hat{\mathbf{x}}) \leqslant 1 \tag{B.3}$$

et

$$((\mathbf{p}_0 + \mathbf{v}) - \hat{\mathbf{x}})^{\mathrm{T}}\,\mathbf{M}\,((\mathbf{p}_0 + \mathbf{v}) - \hat{\mathbf{x}}) \leqslant 1 \tag{B.4}$$

Ce test de collision peut être résumé par les algorithmes 22 et 23. L'algorithme 22 a pour but de tester s'il y a collision entre une ellipse et un segment. Il utilise (B.1) pour tester si le segment et l'ellipse ont un point d'intersection, et fait référence à (B.3) et (B.4) pour tester si un segment est contenu dans l'ellipse. L'algorithme 23 utilise une boucle afin de tester si une ellipse ne rentre en collision avec aucun des segments qui composent l'environnement.

Algorithme 22 CollisionEllipseSegment(**in** : Ellipse $\mathcal{E}_v = \{\mathbf{x} : (\mathbf{x}-\hat{\mathbf{x}})^{\mathrm{T}}\mathbf{M}(\mathbf{x}-\hat{\mathbf{x}}) = k_v^2$, Segment $S)$

Résoudre$\Big(((\mathbf{p}_0 + t\mathbf{v}) - \hat{\mathbf{x}})^{\mathrm{T}}\,\mathbf{M}((\mathbf{p}_0 + t\mathbf{v}) - \hat{\mathbf{x}}) = k_v^2, t\Big)$
Si $t \in [0,1]$ **alors**
  Retourner VRAI {Il y a un point d'intersection}
FinSi
Si $(\mathbf{p}_0 - \hat{\mathbf{x}})^{\mathrm{T}}\mathbf{P}_v(\mathbf{p}_0 - \hat{\mathbf{x}}) \leqslant k_v^2$ **and** $((\mathbf{p}_0 + \mathbf{v}) - \hat{\mathbf{x}})^{\mathrm{T}}\,\mathbf{P}_v\,((\mathbf{p}_0 + \mathbf{v}) - \hat{\mathbf{x}}) \leqslant k_v^2$ **alors**
  Retourner VRAI {Le segment est dans l'ellipse}
FinSi
Retourner FAUX {Il n'y a pas de collision}

Algorithme 23 CollisionEllipseEnvironnement(**in** : Ellipse $\mathcal{E}$, Environment E)

PourTout Segment $S_i \in E$ **faire**
  Si CollisionEllipseSegment($\mathcal{E}$, S_i) = VRAI **alors**
    Retourner VRAI
  FinSi
FinPour
Retourner FAUX