RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à l'INSA Toulouse

Planification de mouvement robuste avec prise en compte du contrôle face aux incertitudes paramétriques.

Thèse présentée et soutenue, le 4 mars 2025 par

Simon WASIELA

École doctorale
SYSTEMES

Spécialité
Robotique et Informatique

Unité de recherche
LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes

Thèse dirigée par
Thierry SIMEON et Juan CORTES

Composition du jury
M. Daniel DELAHAYE, Président, ENAC
Mme Marilena VENDITTELLI, Rapporteure, Sapienza Università di Roma
M. Pedro CASTILLO, Rapporteur, CNRS Hauts-de-France
M. Marco TOGNON, Examinateur, INRIA Rennes
M. Thierry SIMEON, Directeur de thèse, CNRS Occitanie Ouest
M. Juan CORTÉS, Co-directeur de thèse, CNRS Occitanie Ouest

Membres invités
M. Marco Cognetti, Co-encadrant, Université de Toulouse

Remerciements

Une thèse est une quête de savoir, un voyage que l'on entreprend par curiosité et passion, sans toujours savoir exactement où il nous mènera. C'est une aventure parsemée d'incertitudes, où il est facile de se perdre sans un entourage fiable sur lequel on peut compter, qui éclaire notre chemin et nous donne la force d'avancer à chaque étape.

C'est pourquoi je tiens avant tout à remercier Nic et Juan, mes directeurs de thèse, pour leur précieux encadrement, leurs conseils avisés et leur soutien tout au long de cette aventure. Merci de m'avoir fait confiance, d'avoir compris mes attentes, de m'avoir consacré votre temps et de tout ce que vous avez pu faire d'autres pour moi. Je suis heureux que nous ayons été sur la même longueur d'onde. Travailler avec vous fut un véritable plaisir et m'a permis de mener ce voyage à bon port avec enthousiasme.

Un merci tout particulier à Marco Cognetti, qui a rejoint ce projet en cours de route. Merci pour ton engagement aussi bien dans le suivi de ma thèse que dans la rédaction. Ton expertise sur le sujet m'a été d'une aide précieuse et, bien que tu n'aies pas été officiellement rattaché à cette thèse, j'espère que ces quelques mots mettront en valeur ta contribution essentielle à sa réussite.

Je tiens ensuite à remercier ma compagne Laure. Au-delà de cette thèse, ma plus belle aventure est sans aucun doute celle que nous vivons tous les deux. Merci pour tes attentions, ta petience, ton soutien, ou encore ton rire. Merci de m'avoir remonté le moral quand il était au plus bas. Merci d'avoir supporté mes interminables discussions sur la génération de mon dataset. Sans toi, cette thèse aurait été bien différente et ma vie, en général, serait beaucoup plus terne.

Merci à ma mère Françoise et mon père Eric. Merci d'avoir soutenu ma décision de me lancer dans cette thèse. Merci d'avoir stressé à ma place tout au long de ces années, surtout toi, papa. Je vous dois ce que je suis aujourd'hui. J'espère vous avoir rendus fiers.

Merci à mes frères Clément et Paulin. Même si la distance ne nous permet plus de nous voir aussi souvent, vous occuperez toujours une place de choix dans mon coeur. Je suis ravi de vous voir vous épanouir ; cela me motive à me surpasser et m'a permis de terminer cette thèse et ce manuscrit sereinement. Bien entendu, j'espère que vous mettrez cedit manuscrit sous vitrine, cela va de soi. Je tiens également à remercier ma belle-soeur Mélanie pour l'intérêt que tu as porté à mes travaux et, au moins toi, de ne pas tricher à Elfenland.

Merci à mes cousins, cousines, oncles, tantes, parrain, marraine et grands-parents pour leur soutien et enthousiasme.

Je tiens ensuite à remercier mes amis: Corentin, Katerina, Erwan, Maelenn, Cyriaque, Seb, Franklin, Jeanne et JC, pour ces bons moments à incarner les meilleurs personnages d'Yskanov, ou encore pour ces échanges frauduleux d'une pierre contre trois argiles. Les moments passés ensemble me sont précieux et m'ont offert des bouffées d'oxygène quand mon esprit ne pensait plus qu'à cette thèse.

Merci également à mes amis de la fameuse chambre 106.5, Bastien et Eugène. Merci de vous être intéressés à mes travaux et d'avoir compris mes absences. J'espère bientôt

pouvoir rattraper le temps perdu et que nous nous remémorerons encore les bêtises de l'internat.

Cette thèse aurait eu une tout autre saveur sans les membres de la "Place des Swifties", avec qui j'ai passé d'excellents moments. Merci à toi, Lou, pour ces franches rigolades, ces moments de malaise, ces concours de lancer de glands, et bien sûr, ce projet de tyrolienne entre Toulouse et les Alpes (en espérant que personne ne te vole l'idée en lisant ceci) ! Merci à toi, Bastien, d'être toujours partant pour engloutir plusieurs cafés avec moi à n'importe quelle heure, pour ces coupes de cheveux rocambolesques, pour ton aide précieuse dans le setup de mes expériences, pour ne pas avoir hésité un seul instant – mais alors pas du tout – à rejoindre ce carré, et bien sûr, pour ton sens de l'humour inégalable. Je suis sûr que cette dernière qualité m'a permis de récupérer pas mal d'heures de vie perdues, consumées par la thèse. Merci à toi, ô terrifiant Guillaume ! Nos joutes verbales (pour rester politiquement correct) me manqueront. Merci pour ton expertise technique en programmation et pour avoir toujours répondu présent pour les impressions 3D. Et bien sûr, merci de m'avoir conseillé d'excellents jeux, tels qu'Andor ! Finalement, merci à toi, Smail, mon acolyte de voyage depuis la première heure de cette thèse. J'ai énormément apprécié les moments passés avec toi, que ce soit au travail, lors des conférences, ou même au McDo à 1h du matin après les deadlines. Avoir travaillé aux côtés de quelqu'un d'aussi compétent que toi m'a poussé à donner le meilleur de moi-même pour rester à la hauteur. Tu seras toujours le bienvenu pour "crash" chez moi, à condition que tu laisses des Chocobons, bien sûr.

Merci à toi, Philippe, pour tous ces bons moments passés ensemble. Merci pour ta gentillesse et ton empathie. Ton expertise photo en détourage de drone m'a sauvé bien des heures de sommeil. Pas sûr de vouloir encore suivre tes "raccourcis" en randonnée, mais c'est avec plaisir que je viendrais prendre une bière au BSC ou regarder un match de rugby.

Merci infiniment Anthony pour toutes ce rigolades. Merci pour ces discussions allant de l'algorithmique au jeu de rôle, en passant par l'élaboration de notre fameux monde gymnastique, où les personnes s'empileraient les unes sur les autres et où différents bâtiments et activités apparaîtraient en fonction des étages. J'espère te recroiser, non pas dans ce monde gymnastique catastrophique, mais bien en enfer, Hellbanger.

Je tiens également à te remercier, William, pour ces discussions, ces après-midis jeux de société, et bien sûr pour ta maintenance sur Blender. Merci à toi Stephy pour ces moments kool passé ensemble et ta gentillesse. Merci, Illinka, pour ta prise de parole au sujet du sapin, pour l'instauration de ce jeudi Ghibli que je n'ai pas su tenir, et d'être toujours partante pour un concours de Glams. Merci à vous Fadma et Maël pour les bons moments passés ensemble.

Merci également aux personnes que j'ai vu partir au court de cette aventure mais qui m'ont permis une intégration plus que chaleureuse, merci Amandine, Antoine, Yannick, Jeremy, Gianluca, Dario. Merci à Arthur, Félix, Anthony, Aurélie, Rachid, Adrien, Roland, Virgile, Jonas, Alessia, Rebecca, Phani, James, et bien d'autres encore, constituant cette équipe RIS. Merci de m'avoir si bien accueilli, et un grand merci à Simon Lacroix d'être à la tête de cette équipe formidable.

Merci aux membres de l'équipe Rainbow, et tout particulièrement à Paolo pour

m'avoir donné cette opportunité et pour ton enthousiasme à propos de mes travaux. Merci également à Pascal et Ali pour leurs échanges enrichissants.

Je souhaite également exprimer ma gratitude envers les membres de mon jury. Un grand merci à Daniel pour avoir présidé ce jury, et à Marilena, Pedro et Marco pour l'accueil réservé à mes travaux.

Merci au Sapin Éternel, ainsi qu'à ceux qui l'entretiennent, égayant le hall de notre bâtiment et y apportant joie et bonne humeur.

Enfin merci au LAAS-CNRS de m'avoir acceuilli et à l'ANR d'avoir financer mes travaux de recherche.

Abstract

This thesis addresses the problem of generating robust and accurate trajectories taking into account uncertainties in the robot dynamic model. Based on the notion of *closed-loop sensitivity*, which quantifies deviations in the closed-loop trajectories of any robot/controller pair against uncertainties in the robot model parameters, so-called "uncertainty tubes" can be derived for bounded parameter variations. Such tubes bound the system evolution both in the state and control input spaces. Based on the "control-aware motion planning" paradigm, this work leverages these "uncertainty tubes" to enforce robust constraints within sampling-based planners.

The first contribution of this thesis focuses on generating globally sensitivity-optimal trajectories while enforcing robust constraints thanks to uncertainty tubes. However, results show that computing these uncertainty tubes at each iteration of a sampling-based planner is a bottleneck for the method. Therefore, a lazy robust feasibility check is proposed to limit the frequency of computing uncertainty tubes, thus improving the computational efficiency of the framework.

Another contribution is to explore deep learning neural network that can be used to speed up closed-loop sensitivity dynamic and subsequent tubes computation. By leveraging the structural similarity between ordinary differential equations and recurrent neural networks, a GRU-based architecture is proposed to directly link a planned trajectory with uncertainty tubes, achieving an order-of-magnitude improvement in computation time.

The thesis also shows how to integrate GRU-based predictions into a sampling-based planner, resulting in a more computationally efficient planning framework. Furthermore, while robust state-of-the-art methods primarily focus on satisfying robust constraints, this work leverages uncertainty tubes to define a cost function that enables the planning of task-specific, accurate trajectories.

Finally, the proposed sensitivity-based planning framework is experimentally validated on a 3D quadrotor in two challenging scenarios: a navigation through a narrow window, and an in-flight "ring catching" task that requires high accuracy.

Keywords: Robotics, Motion planning, Control-aware, Robustness, Uncertainties

Résumé

Cette thèse aborde le problème de la génération de trajectoires robustes et précises en tenant compte des incertitudes dans le modèle dynamique du robot. Basée sur la notion de *sensibilité en boucle fermée*, qui quantifie les déviations des trajectoires en boucle fermée de toute paire robot/contrôleur face aux incertitudes des paramètres du modèle du robot, des "tubes d'incertitude" peuvent être calculés pour des variations bornées de paramètres. Ces tubes bornent l'évolution du système à la fois dans les espaces d'état et d'entrée de commande. En s'appuyant sur le paradigme de la "planification de mouvement sensible au contrôle", ce travail exploite ces "tubes d'incertitude" pour imposer des contraintes robustes au sein des planificateurs basés sur l'échantillonnage.

La première contribution de cette thèse porte sur la génération de trajectoires globalement optimales en sensibilité tout en imposant des contraintes robustes grâce aux tubes d'incertitude. Cependant, les résultats montrent que le calcul de ces tubes d'incertitude à chaque itération d'un planificateur basé sur l'échantillonnage constitue un frein pour la méthode. Ainsi, une vérification de faisabilité robuste paresseuse est proposée afin de limiter la fréquence de calcul des tubes d'incertitude, améliorant ainsi l'efficacité computationnelle des algorithmes.

Une autre contribution consiste à explorer l'utilisation de réseaux de neurones profonds pour accélérer l'intégration de la dynamique de la sensibilité en boucle fermée et le calcul subséquent des tubes. En tirant parti de la similarité structurelle entre les équations différentielles ordinaires et les réseaux neuronaux récurrents, une architecture basée sur les GRU est proposée pour corréler directement une trajectoire planifiée aux tubes d'incertitude, atteignant une amélioration d'un ordre de grandeur en termes de temps de calcul.

La thèse montre également comment intégrer les prédictions basées sur les GRU dans un planificateur basé sur l'échantillonnage, aboutissant à des approches de planification plus efficace en temps de calcul. De plus, alors que les méthodes robustes de l'état de l'art se concentrent principalement sur la satisfaction des contraintes robustes, ce travail exploite les tubes d'incertitude pour définir une fonction de coût permettant la planification de trajectoires précises et spécifiques à la tâche.

Enfin, la méthode de planification basé sur la sensibilité proposé est validé expérimentalement sur un quadrirotor 3D dans deux scénarios exigeants : une navigation à travers une fenêtre étroite, et une tâche de "capture d'anneau" en vol nécessitant une grande précision.

Mots clés: Robotique, Planification de mouvement, Prise en compte du contrôle, Robustesse, Incertitudes

Contents

Acronyms **xi**

1 Introduction **1**
1.1 Context and Objectives . . . 1
1.2 Thesis Outline . . . 2
1.3 Thesis Contributions . . . 3

2 Related work **5**
2.1 Path planning . . . 5
2.2 Motion planning . . . 6
2.3 Motion planning under uncertainty . . . 9
2.4 Control-aware motion planning . . . 10
2.5 Robust control-aware motion planning . . . 11
2.6 Outlook . . . 13

3 Preliminaries **15**
3.1 Closed-loop sensitivity . . . 15
3.1.1 Definition . . . 15
3.1.2 Tube computation . . . 16
3.2 Models considered in this thesis . . . 19
3.2.1 Differential drive robot . . . 19
3.2.2 Quadrotor robot . . . 22

4 Robust Motion Planning with Global Sensitivity Optimization **27**
4.1 Global robust sensitivity-optimal planning . . . 27
4.1.1 Standard asymptotically optimal planners . . . 28
4.1.2 Sensitivity-Aware RRT* (SARRT*) . . . 30
4.1.3 Sensitivity-Aware SST* (SASST*) . . . 32
4.1.4 Cost function . . . 34
4.1.5 Simulation results . . . 34
4.2 Decoupled approach . . . 43
4.2.1 Global robust motion planning with lazy strategy (LazySARRT*) 43
4.2.2 Local robust sensitivity optimization . . . 45
4.2.3 Simulation results . . . 46
4.3 Conclusions . . . 51

5 Learning uncertainty tubes via recurrent neural networks **53**
5.1 Techniques for accelerating ODEs integration . . . 53
5.1.1 ODEs parallelization . . . 54
5.1.2 Learning with embedded dynamics . . . 54
5.1.3 Learning sequences of data . . . 55

5.1.4 Outlook . . . 57
5.2 Proposed RNNs-based approach . . . 58
5.2.1 Problem statement . . . 58
5.2.2 Neural network architecture . . . 58
5.2.3 Dataset . . . 60
5.3 Evaluation . . . 62
5.3.1 Metrics . . . 62
5.3.2 Training . . . 63
5.3.3 Implementation details . . . 63
5.4 Simulation results for a quadrotor application . . . 64
5.4.1 Model comparison . . . 64
5.4.2 Ablation study . . . 66
5.4.3 Qualitative results . . . 67
5.5 Application to the differential drive robot . . . 70
5.5.1 Setup adaptation . . . 70
5.5.2 Simulation results . . . 71
5.6 Conclusion . . . 74

6 Robust motion planning with accuracy optimization via learned sensitivity metrics 75
6.1 Robust and accurate planning framework . . . 76
6.1.1 Robust motion planning via deep learning . . . 77
6.1.2 Robust local accuracy optimization . . . 78
6.2 Simulation results . . . 82
6.2.1 Quadrotor setup . . . 82
6.2.2 DeepSAMP vs SAMP . . . 83
6.2.3 Robust motion planning via deep learning . . . 85
6.2.4 Robust local accuracy optimization . . . 87
6.3 Experimental validation . . . 92
6.3.1 Robust motion planning via deep learning . . . 92
6.3.2 Robust motion planning with accuracy optimization . . . 93
6.4 Conclusion . . . 95

7 Conclusion 97
7.1 Contributions . . . 97
7.2 Future Works . . . 98

A Robust feasibility checking 101

B Learning curves 105

C ExtendedShortcut 107

D Résumé en Français 113

References 117

Acronyms

BVP Boundary Value Problem. 8

CBFs Control Barrier Functions. 9

CHOMP Covariant Hamiltonian Optimization for Motion Planning. 7

COBYLA Constrained Optimization BY Linear Approximations. 82

CoM center of mass. 22

DeepSAMP Deep learning-based Sensitivity-Aware Motion Planner. 77, 98

DeepSARRT Deep learning-based Sensitivity-Aware RRT. 77

DFL dynamic feedback linearization. 20

GRU Gated Recurrent Unit. 56

LazySAMP Lazy Sensitivity-Aware Motion Planner. 43, 51

LazySARRT* Lazy Sensitivity-Aware RRT*. 43

LMT Lozano-Pérez, Mason and Taylor. 9

LPV Linear Parameter Variation. 9

LQR Linear Quadratic Regulator. 11

LSTM Long Short-Term Memory. 56

MPC Model Predictive Control. 2, 10, 114

ODEs ordinary differential equations. 15, 57, 97, 114

OMPL Open Motion Planning Library. 35

PINNs Physics-Informed Neural Networks. 53, 99

POMDP Partially Observable Markov Decision Process. 9

PRM Probabilistic Roadmaps. 5

RNN Recurrent Neural Network. 56

RNNs Recurrent Neural Networks. 53, 57

RRT Rapidly-exploring Random Trees. 6, 28

RRT* Asymptotically Optimal Rapidly-exploring Random Trees. 28

SAExtendedShortcut Sensitivity-Aware ExtendedShortcut. 81

SAMP Sensitivity-Aware Motion Planner. 28, 98

SARRT* Sensitivity-Aware RRT*. 30, 51

SAShortcut Sensitivity-Aware Shortcut. 45

SASST* Sensitivity-Aware SST*. 32, 51

SASTOMP Sensitivity-Aware STOMP. 79

SOS Sums-of-Squares. 11

SST Stable Sparse RRT. 8

SST* Stable Sparse RRT*. 29, 32

STOMP Stochastic trajectory Optimization for Motion Planning. 7, 79

TMPC Tube Model Predictive Control. 11

Chapter 1

Introduction

1.1 Context and Objectives

Robot motion planning is a crucial aspect of ensuring efficient and safe operation in dynamic, real-world environments. Robots are inevitably subjected to various uncertainties, such as external disturbances (e.g., wind), inaccuracies in their models, and errors in state estimation. Effectively managing these uncertainties is essential for robust and reliable robot behavior.

An effective approach to managing the complexity of robots operating in uncertain real-world environments is the decoupled planning and control approach. This process involves two main steps:

1. Planning Phase: A reference trajectory for the robot states and controls is planned based on available information, such as models of the robot and its environment. This step, typically executed offline, incorporates constraints (e.g., collision avoidance, limited actuation) and optimizes metrics like trajectory length or energy efficiency. However, executing this planned trajectory in open-loop often fails in practice due to uncertainties affecting the planned reference.

2. Control Phase: To ensure robust execution, a motion controller is employed to close the loop between the planned and actual motion, compensating for unforeseen effects and uncertainties that were not accounted for during planning.

While this sequential approach, which separates planning and control, has proven useful, it exhibits significant limitations:

- Modern planners are able to generate feasible and globally optimal trajectories for high-dimensional kino-dynamical systems and complex constraints. However, they typically ignore the role of the runtime feedback controller, leading to two key issues. First, the controller must deviate from the planned trajectory to address uncertainties and disturbances, which can quickly compromise feasibility and optimality. Then, the planner does not account for the robustness inherently provided by the controller, missing opportunities to generate more robust motion plans.

- While many adaptive and robust control schemes (e.g., H-infinity, LPV methods) have been designed to handle uncertainties and disturbances effectively, they are local to the vicinity of a reference trajectory. These methods often fall short when it comes to addressing broader challenges such as feasibility under constraints, global optimality, and computational performance, which are better managed by global planning approaches.

To bridge the gap between planning and control, several approaches have been introduced, including less local controllers such as the well-known Model Predictive Control (MPC) [Lim+10]. In addition, the past decade has seen the rise of so-called "control-aware motion planning"[1] [MT17; Sin+21; Tog+18]. However, these methods still face challenges related to generalizability, computational efficiency, and their reliance on potentially inaccurate models of the robot/controller pair due to model parameters uncertainties.

Therefore, the work presented in this thesis aims to go further in the coupling between planning and control. Based on the "control-aware motion planning" paradigm shown in Figure 1.1, it leverages the closed-loop sensitivity concept [BDR21; RDF18] (an extension of the sensitivity notion that incorporates the controller behavior with respect to parametric uncertainties), in order to create robust control-aware motion planners applicable to a broad class of systems and controllers, while addressing uncertainties in their model parameters.

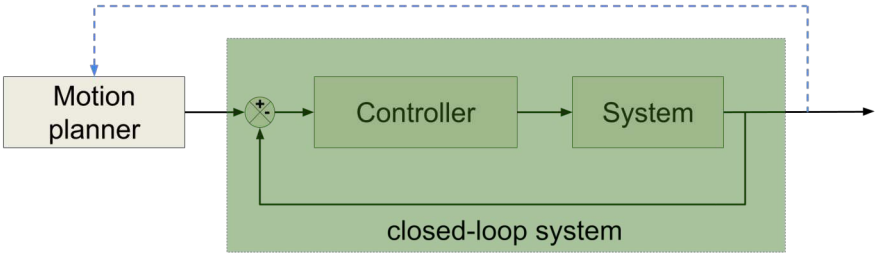


Figure 1.1: Illustration of the control-aware motion planning strategy. Unlike the standard decoupled approach, which first plans a desired motion (gray block) and then executes it in closed-loop with a controller (green block) without incorporating feedback from the closed-loop system at the planning stage (blue dashed line), the control-aware strategy integrates this feedback. It simulates the closed-loop system's response (green) during the offline motion planning phase (gray), as indicated by the blue dashed line.

1.2 Thesis Outline

This manuscript begins with a literature review in Chapter 2, providing an overview of the state of the art. It initially focuses on decoupled approaches to robust motion planning, starting with path finding methods and progressing to kinodynamic planning. The review then explores approaches that tackle the challenge posed by the presence of uncertainties, focusing on solutions at the motion planning level. It further discusses unified methodologies, referred to as control-aware planning techniques, which integrate motion planning with control strategies. The chapter concludes with a focus on unified robust control-aware motion planning approaches, which are the main topic of interest.

Chapter 3 introduces the *closed-loop sensitivity* concept and presents how this concept can be leveraged to derive the so-called uncertainty tubes. These uncertainty tubes are

[1] Also known in the literature as "feedback motion planning".

later employed in this thesis to impose robust constraints on systems while accounting for parametric uncertainties. Then, this chapter presents the quadrotor and differential drive robot models considered in this thesis, along with their respective controllers and local planning techniques used for generating reference trajectories.

The first contribution of this thesis is then introduced in Chapter 4. It presents the general method for integrating sensitivity-based uncertainty tubes into sampling-based planners to create various sensitivity-aware motion planner variants. Furthermore, this chapter emphasizes the generation of sensitivity-optimal trajectories in global context. Initial simulation results, using the models introduced in Chapter 3, highlight that computing uncertainty tubes rapidly become a bottleneck for sampling-based motion planning methods. To address this issue, a more efficient variant is proposed, leveraging a lazy collision-checking approach to reduce the need of uncertainty tube computations.

Chapter 5 introduces the second major contribution of this thesis, which leverages the structural similarities between ordinary differential equations and recurrent neural networks cells to accelerate the computation of sensitivity-based uncertainty tubes. A dataset generation method, based on sampling-based principle, is proposed and applied to the models discussed in Chapter 3. The method achieves an optimal balance between inference time and accuracy, making it highly effective for motion planning and showcasing the efficiency of the approach.

Chapter 6 details the integration of the sensitivity-aware motion planning variants from Chapter 4 with the deep learning approach from Chapter 5, resulting in a more efficient version of the sensitivity-aware motion planner variants. Furthermore, while the previous chapters solely focus on generating robust and sensitivity-optimized trajectories, this chapter emphasizes the generation of task-specific accurate trajectories. The contribution is finally validated experimentally in two challenging scenarios, highlighting the framework's practical applicability and effectiveness in real-world tasks requiring high robustness and accuracy.

Finally, Chapter 7 presents an overall conclusion that summarizes the contributions of this thesis along with their limitations. Additionally, several perspectives are outlined to improve the generalizability of the proposed approach.

1.3 Thesis Contributions

The work conducted throughout this thesis led to the following publications.

[Was+23] Wasiela, S., Giordano, P. R., Cortés, J., and Simeon, T. "A sensitivity-aware motion planner (samp) to generate intrinsically-robust trajectories." In IEEE International Conference on Robotics and Automation (ICRA), 2023.

[Was+24a]: Wasiela, S., Bouhsain, S. A., Cognetti, M., Cortés, J., and Simeon, T. "Learned Uncertainty Tubes via Recurrent Neural Networks for Planning Robust Robot Motions." In 27th European Conference on Artificial Intelligence (ECAI), 2024.

[Was+24b]: Wasiela, S., Cognetti, M., Giordano, P. R., Cortés, J., and Simeon, T. "Robust Motion Planning with Accuracy Optimization based on Learned Sensitivity Metrics." In IEEE Robotics and Automation Letters (RA-L), 2024.

Belvedere, T., Afifi, A., Wasiela, S., and Pupa, A. "Planning under Uncertainties with

Closed-Loop Sensitivity: Recent Results and Perspectives." In European Robotics Forum (ERF), 2025.

Furthermore, all implementations presented and used in this thesis are available at: https://gitlab.laas.fr/CAMP.

Chapter 2

Related work

This chapter provides an overview of the related work on the key concepts that this thesis builds upon and compares with.

2.1 Path planning

This section provides an overview of various path planning approaches. For a more detailed survey, please refer to [LaV06; OCK23; Pad+16]. The path planning problem focuses on finding a collision-free possibly optimal path between a starting configuration and a goal configuration. The work of [LW79] introduced the configuration space concept as a general framework for planning paths of a general class of kinematic systems. This idea provided a solid foundation for the field and, as noted in [Loz90], formally established the path planning problem as the task of finding a feasible path within the configuration space.

However, finding feasible paths in the configuration space poses significant challenges due to the problem computational complexity [Can88]. An alternative local approach is the introduction of artificial potential fields [Kha86], which create a repulsive force field around obstacles and an attractive force field toward the target. However, although this strategy has proven to be efficient, it sacrifices guarantees such as algorithm completeness and global optimality.

Another common approach to the path planning problem involves search-based techniques. These methods operate on a discrete representation of the configuration space, where vertices correspond to a finite set of robot configurations, and edges represent possible transitions between these configurations. The desired path is found by performing a search for a minimum-cost path in such a graph using Dijkstra algorithm [Dij59].

Advances in computational techniques have enabled the development of a new class of algorithms, commonly referred to as sampling-based planners that allow to generate global paths. A key contribution of the domain is the work of [BL91], which improves the aforementioned potential fields approach by integrating a random walk mechanism, thus guaranteeing the probabilistic completeness of the algorithm. Sampling-based planners are divided in two main categories: graph-based planners and tree-based planners.

Graph-based sampling-based planners, such as Probabilistic Roadmaps (PRM) [Kav+96], generate a roadmap by sampling configurations in the free space and connecting them with feasible paths. The technique involves a graph building phase and a query phase. The graph is first built by randomly sampling configurations (denoted nodes) in the robot configuration space. Each sampled configuration is checked for collisions with obstacles. For each collision-free node, the planner attempts to connect it to nearby free configurations by creating edges between them. The edges are valid

if the path connecting the two configurations is collision-free. This procedure typically involves a notion of neighborhood that is space-dependent and generally increases as the dimensionality of the space grows. Once the graph is built, a graph search query is performed to compute the desired path, using graph search techniques such as Dijkstra algorithm [Dij59] or A* [HNR68].

The other family among sampling-based planners are tree-based planners such as the well-known Rapidly-exploring Random Trees (RRT) [LaV98]. This algorithm generates a tree starting from an initial configuration. At each iteration, a new configuration is sampled q^{rand}. Its nearest neighbor q^{near} among the existing tree nodes is found, and then an extension is attempted starting from q^{near} toward q^{rand} leading to a new configuration q^{new}. If the extension is collision free, the new configuration is added to the tree as a new node with an edge connecting the two nodes.

Therefore, sampling-based planners have become the most widely used strategy for generating global paths in the presence of obstacles, due to their practical efficiency in handling complex and high-dimensional configuration spaces, as well as their completeness guarantee. Advances in sampling-based planners have then focused on generating feasible and cost-effective paths through various strategies, such as the transition-based approach [JCS10]. These strategies often achieve asymptotic optimality by employing optimal connection procedures during the tree or graph construction process [DSC15; Jan+15; KF11]. Research on sampling-based planners has expanded into various areas, such as developing suitable sampling strategies [Yer+05], exploring lazy collision checking approaches [BK00], extending to the molecular domain [ASC12], and more.

The aforementioned sampling-based methods are commonly known as global planning methods, as they depend on local planning techniques, referred to as steering methods, to generate the edges connecting sampled configurations. A steering method is a local planner that connects an initial configuration to a goal without accounting for obstacle avoidance. While typically fast, steering methods may produce paths that do not fully adhere to problem constraints, potentially leading to collisions. Traditional steering methods often involve simple path segments, but they frequently need to accommodate additional constraints. For instance, significant work has been done to enforce kinematic constraints on these local paths, ensuring the kinematic feasibility of the resulting global paths. Such work includes, for example, Dubins path generation [Dub57], which enables connecting two configurations under forward motion constraints and curvature constraints for car-like robots. The Reeds-Shepp method [RS90a] extends this local path generation procedure to car-like robots capable of moving both forward and backward.

2.2 Motion planning

While the aforementioned path finding methods focus on generating a route between two robot configurations, they do not address the problem on how the robot should move along this route over time. This problem is known as motion planning and involves considering derivatives of the robot configuration [KW09]. Consequently, motion planning extends beyond searching the robot's configuration space, also relying on a dynamic model of the system.

Moreover, while path planning can handle kinematic constraints based on configuration representation, incorporating the state space allows for the consideration of dynamic constraints, such as maximum velocity and acceleration. The task of generating motion plans that account for both kinematic constraints (e.g., turning radius) and dynamic constraints (e.g., maximum velocity), ensuring that the plans are physically feasible, is known as kinodynamic motion planning.

The distinction between path planning and motion planning naturally emphasizes the importance of local trajectory generation, a key element in refining motion plans. Local trajectory planners play an essential role in motion planning by generating feasible movements between two robot states while respecting various constraints, such as kinematic and dynamic limits, as well as obstacle avoidance. Operating on a localized scale, these planners are key to refining or connecting segments of a global motion plan, ensuring that the overall trajectory remains both feasible and optimized.

Among the widely used local trajectory planners, Bézier curve-based planners [ETH18; JKV09] stand out due to their inherent smoothness and the advantageous convex hull property. The convex hull property ensures that the trajectory remains within the convex hull of its control points, simplifying collision detection. If the convex hull is free of obstacles, the entire trajectory is guaranteed to be obstacle-free. Additionally, adjusting control points affects only a localized portion of the trajectory, allowing precise modifications without impacting the entire path. This approach is particularly useful in environments where efficient collision detection is crucial. Furthermore, its smoothness also naturally accommodates velocity, acceleration, and higher-order dynamic constraints, making it suitable for kinodynamic motion planning in various robotic systems.

Another powerful method is Covariant Hamiltonian Optimization for Motion Planning (CHOMP) [Rat+09], which formulates trajectory generation as an optimization problem starting from an initial trajectory as guess. The cost function in CHOMP typically balances smoothness and obstacle avoidance. Smoothness is ensured by penalizing higher derivatives of the trajectory, while obstacle avoidance is achieved through the use of a signed distance field. CHOMP can incorporate dynamic and kinematic constraints, such as joint limits and velocity bounds, which are important for kinodynamic planning. Using gradient-based optimization, CHOMP iteratively refines the initial trajectory, ensuring that it satisfies all constraints while minimizing the cost function. While effective in high-dimensional or cluttered environments, CHOMP does require a good initial trajectory and can struggle in environments with non-convex obstacles.

While CHOMP can handle differentiable cost function due its gradient-based optimization, the Stochastic trajectory Optimization for Motion Planning (STOMP) algorithm [Kal+11] is designed to optimize non-differentiable cost functions. STOMP iteratively improves trajectories by introducing small perturbations to the initial guess, evaluating the resulting costs, and updating the trajectory based on the best-performing samples. This process also accounts for kinodynamic constraints and obstacle avoidance by adding state cost penalty in the cost function formulation. While STOMP is computationally more expensive than CHOMP as it relies on several sampling-based rollouts, its ability to handle more complex cost functions and find better solutions in challenging environments makes it highly versatile for kinodynamic motion planning.

Nevertheless, the aforementioned methods focus on generating trajectories within a limited, localized scope according to a partial environment knowledge. These methods are typically designed to handle immediate obstacles and environmental changes, allowing for quick responses and adaptability.

In contrast, global planning problems are more effectively addressed by sampling-based global planners that cover the entire environment. Global sampling-based planners usually rely on local planning techniques, also known as steering methods, which do not consider obstacles. To fully address the motion planning problem, these planners can use time-parametrized steering methods. Additionally, to solve the kinodynamic motion planning problem, sampling-based planners can leverage kinodynamic steering methods, which take both kinematic and dynamic constraints into account during trajectory generation.

A common approach to generating kinodynamic steering methods involves solving a two-point Boundary Value Problem (BVP) using polynomial-based methods, such as splines, which are widely employed for trajectory interpolation. For instance, the kinosplines local planner proposed in [BCS19] leverages system differential flatness, ensuring the creation of continuous and differentiable trajectories up to the 4th order by employing a bang-bang snap control strategy. The resulting trajectories respect kinodynamic constraints up to the jerk level, enabling time-optimal connections between two given states. Another example is the minimum snap trajectory generation method [MK11a], which contrasts with the bang-bang snap strategy of kinosplines. Also based on system differential flatness, this approach formulates an optimization problem that minimizes the integral of a weighted combination of squared position snap and squared yaw acceleration. It ensures continuity between trajectory segments, satisfies kinematic conditions at the start and end of the motion, and respects actuation limits.

Global sampling-based motion planners can generate globally kinodynamically feasible trajectories by leveraging the aforementioned kinodynamic steering methods to connect their sampled states. However, such local steering techniques may not be available for every systems, as solving the associated BVP can be particularly challenging. To overcome this, algorithms were developed to bypass this need by performing dynamic system forward propagation instead of solving a complex BVP.

Kinodynamic motion planners, such as Kinodynamic RRT [LK01] and Stable Sparse RRT (SST) [LLB16], are designed to address the challenges of planning for robots with complex dynamics without relying on a steering method. These methods utilize system forward dynamic propagation by sampling control inputs and propagation time. Starting from an initial robot state, the system dynamics are integrated in an open-loop manner to propagate the state, allowing the planner to account for both the robot kinematic limitations and its dynamic behavior. This approach ensures that the generated paths are collision-free and physically executable while eliminating the need for local steering methods.

2.3 Motion planning under uncertainty

While the previous section focused on motion planning algorithms that can deal with kinodynamic constraints of various types of robots, they do not consider the unavoidable presence of uncertainties (e.g. external disturbances, sensor noise, model parameter mismatches, etc.).

A common approach to managing uncertainties is to compensate for them in real-time using robust control strategies, such as H-infinity [Zam81] or Linear Parameter Variation (LPV) [MS12] methods. For instance, the work of [Bre+13] proposes an adaptive law strategy for estimating the parameters of unknown disturbances, integrating them into a closed-loop system. In [CFV22] Control Barrier Functions (CBFs) are leveraged to design closed-form controllers that ensure both the safety and stability of system execution in the presence of obstacles. However, these methods often encounter difficulties in maintaining robustness when applied to non-robust reference trajectories, due to their inherently local nature. Additionally, they struggle to achieve global optimality, which are typically better addressed by global planning approaches. Therefore, research has focused on designing motion planning algorithms that incorporate uncertainty into the trajectory generation process, with the goal of producing robust and feasible plans.

A seminal contribution to the field is the Lozano-Pérez, Mason and Taylor (LMT) framework [LMT84], which addresses the challenge of computing compliant motion strategies in the presence of sensor uncertainties and inaccuracies in robot movement. This approach focuses on developing fine-motion strategies, which involve incrementally generating sequences of robot movements that account for inaccuracies in both the robot state and the sensed environment. The authors consider the robot kinematic constraints and sources of error, such as sensor discrepancies or movement inaccuracies. By ensuring that the robot motions remain collision-free and adaptable to these uncertainties, the framework aims to achieve robustness in real-world tasks. However, a bottleneck of the method is its high computational cost.

By leveraging the probabilistic robotic concept [Thr02], approaches such as Partially Observable Markov Decision Process (POMDP) or, in general, the class of "planning in belief spaces" approaches, are another alternative for providing a principled and general planning framework for dealing with uncertainty. They use stochastic uncertainty propagation functions, to minimize the uncertainty along the trajectory to the desired target [CH10; KS98]. POMDP model sequential decision-making in environments with uncertainty, considering both actions taken by the agent and the uncertain nature of observations and transitions. The proposed frameworks have the advantage of being adaptable to a wide range of non-trivial robot models, with non-Gaussian processes and observation models for trajectory planning with minimum uncertainty. However, these methods still suffer from computational bottleneck when dealing with high dimensional problems.

When probability distributions over uncertain variables are available, chance-constrained motion planning can be used to generate plans that satisfy probabilistic constraints [ACA14; LBM21; LKH10]. These constraints ensure that specific conditions, such as safety or feasibility, are met with a certain probability. Typically, a chance constraint is incorporated into an optimization problem, where the objective (e.g., minimizing

cost or maximizing reward) is optimized while ensuring that the constraint holds with a high probability (e.g. ensuring that a robot avoids obstacles with 95% probability).

Particle-based methods such as Particle RRT [MS07] or RRT-SALM [HG08], use particle filters to model uncertainty. Rather than relying on a single robot state, these algorithms represent the robot state as a set of particles, each corresponding to a possible uncertain state. The particles are propagated through the environment according to the system dynamic, and the tree is expanded by sampling from these particles.

The problem of generating open-loop trajectories with minimum state sensitivity has been recently addressed in [AM13] and further expanded upon in [AM16]. This later work introduces a method for generating robust trajectories using an open-loop optimization routine that accounts for deviations in model parameters. In contrast to the aforementioned probabilistic approaches, this framework operates under the assumption of accurate models and demonstrates that minimizing first-order sensitivities leads to reduced deviations from the nominal state when model parameters are imperfect.

2.4 Control-aware motion planning

The algorithms presented so far have been designed to handle kinodynamic constraints and/or ensure robustness against various uncertainties. However, they do not takes into account the inevitable presence of a feedback controller responsible for executing the generated trajectories. This controller might deviate from the planned trajectory to address uncertainties and disturbances, which can quickly compromise feasibility and optimality of the plan. It is important to note that kinodynamic approaches relying on system dynamics forward propagation, as discussed in Section 2.2, do not incorporate feedback actions. Instead, they directly sample in the control input space, resulting in an open-loop propagation strategy.

The idea of merging planning with control for generating “robust planners” or more “global controllers” for dealing with the robustness problem in a more comprehensive way is however not completely novel in the robotics community. This yields to the “control-aware motion planning” paradigm.

Work has been carried out on the control community side to propose “less local” controllers, resulting in the popular MPC technique [CA13] that iteratively replans an optimal (and feasible) trajectory with a feedback from the current robot state (that may deviate from the desired one because of disturbances/uncertainties). This method enables fast computation of locally optimal trajectories while incorporating the controller feedback action. However, due to its inherently local nature, the MPC approach can become trapped in local minima with respect to cost or obstacles. It often lacks guarantees of completeness and relies heavily on an accurate robot model, without explicitly accounting for uncertainties.

A seminal work in feedback motion planning is the LQR-Trees method [Ted+10]. It addresses the problem of generating global trajectories that are stabilizable. While the decoupled planning and control method is effective in many cases, it may result in trajectories that are either not stabilizable or significantly more costly to stabilize compared to alternative, more optimal trajectories. Therefore, the LQR-Trees method combines

trajectory optimization, feedback control, and Lyapunov verification to construct a tree of locally valid control policies. This structure efficiently covers the state space and facilitates the generation of low-cost stabilization trajectories. The approach operates as follows:

1. Local Control via Linear Quadratic Regulator (LQR): The system starts by generating a trajectory to a goal state using a trajectory optimization method. A local LQR controller is then synthesized to stabilize the system around this trajectory.

2. Sums-of-Squares (SOS) Verification: A polynomial Lyapunov function is used to certify regions of attraction around the trajectory. This involves verifying the system stability and feasibility within the computed region using SOS optimization.

3. Tree Construction: The algorithm iteratively adds new trajectories and their associated controllers to a tree structure. New trajectories are initialized from states that fall outside existing regions of attraction, progressively expanding the tree to cover the state space.

While the LQR-Trees approach provides a scalable and efficient solution for motion planning in nonlinear and high-dimensional spaces, effectively bridging the gap between open-loop planning and control, its applicability is constrained to the LQR control formulation.

Another notable contribution to control-aware motion planning is presented in [Tog+18]. This work focuses on motion planning methods that account for the system dynamic model and control capabilities, particularly in the context of task constraints planning. It introduces an innovative motion planning framework that integrates control limitations and redundancy optimization by conducting system closed-loop simulations during the extension phase of a sampling-based planner, ensuring the feasibility of the generated trajectories for the manipulator control system. However, this method does not explicitly account for uncertainties.

2.5 Robust control-aware motion planning

Although the previously mentioned methods have bridged the gap between motion planning and control, they do not explicitly address uncertainties or how the controller will respond to such uncertainties to mitigate their effects. An effective framework should ensure safe operation under uncertainty by considering the role of a feedback controller. This challenge is known as robust control-aware motion planning, or robust feedback motion planning.

To address this challenge, an extension of the standard MPC, known as Tube Model Predictive Control (TMPC), was introduced in [Lim+10]. TMPC is a robust control strategy designed to handle uncertainties in dynamic systems while maintaining performance and stability. It generates a nominal trajectory using MPC and defines an invariant "tub" around this trajectory, within which the system state must remain. A feedback controller is then applied to ensure that, despite disturbances or model inaccuracies, the system trajectory stays within the tube, thus maintaining robustness and stability. This approach allows TMPC to handle uncertainties effectively while ensuring

the system follows a desired trajectory, making it particularly useful in safety-critical applications. However, TMPC faces similar challenges as its non-robust counterpart, such as the risk of getting trapped in local minima with respect to cost or obstacles. Additionally, TMPC and MPC, being model-based techniques, are highly sensitive to model uncertainties.

The FaSTrack framework [Her+17] is a motion planning approach designed to address the challenges of fast, adaptive, and robust trajectory generation for dynamic systems, particularly in the context of autonomous vehicles or robots. It integrates real-time feedback to rapidly adjust planned trajectories in response to changes in the environment or the system state. The FaSTrack framework relies on a simplified robot dynamic model, and its robustness is ensured by synthesizing a specific control strategy tailored to the system, which helps it adapt to disturbances and uncertainties. Therefore, this technique is limited to the specific synthesized controller, and limited by its simplified representation.

While the LQR-Trees approach previously focused on trajectory stability and relied solely on LQR control, the formulation was later extended in the Funnel Library approach [MT17]. The core idea of the Funnel Library is to generate control policies that ensure the system trajectory remains within a "tube" around a desired reference trajectory, which is a bounded region where the system is guaranteed to stay despite external disturbances. The framework extends its former LQR-Trees approaches to include controllers of polynomial form that can handle non linearities, disturbances, and model uncertainties. It employs an optimization-based method to design an offline library of control policies that are robust to model errors and external uncertainties. A key advantage of the Funnel Library is its ability to perform real-time updates by searching through the library and combining subsequent tubes online. This allows the control policies to adapt dynamically based on feedback, ensuring that the system can respond to environmental or state changes effectively. However, the method is limited to controllers of polynomial form, and its planning capabilities are constrained by the finite set of control laws in the library.

A seminal contribution to the field is the contraction theory approach, which has served as the foundation for several subsequent works [COB21; Sin+17; Sin+21; Zha+22]. Contraction theory is a generalization of Lyapunov theory used to analyze and design robust control systems, particularly in the presence of uncertainties or disturbances. The core idea is to ensure that the system state trajectories converge towards a desired reference trajectory, regardless of initial conditions or external disturbances. This is achieved by requiring that the system dynamics exhibit a "contracting" behavior, meaning that the distance between any two trajectories decreases over time. In contraction theory, the system dynamics are typically modeled using differential equations, and the stability of the system is guaranteed by synthesizing a controller associated with a contraction rate such that the contraction condition holds. Contraction theory has been leveraged to design "tubes" by determining bounds on the controller "contraction rate". These tubes are then employed to develop robust motion planning algorithms. However, this approach is restricted to the specific synthesized controllers, that may not always be easily implementable in real-world scenarios. Additionally, requiring contraction at every point along a trajectory can be too restrictive and challenging.

While the aforementioned methods explicitly focus on generating robust trajectories to external disturbances or unmodeled forces (e.g., wind, friction), none of them considers potential mismatches or fluctuations in the robot model parameters during runtime, such as changes in mass or displacements of the center of mass. Such robot model inaccuracy can strongly impact their efficiency, as they are strong model based approaches. Moreover, many of these methods hold for a specific class of systems or a specific synthesized controller and, thus, lack generality.

To tackle some of these issues, the RandUp-RRT proposed in [Wu+22], builds upon the aforementioned particle-based RRT, by performing a random uncertainty propagation (RandUP) [Lew+22], taking into account a feedback action of a given controller. The idea is to estimate for each node of the tree the set of states that can be reached by the system using "particles", which corresponds to a dynamic propagation of the system in the presence of random parameters uncertainty. Furthermore, these sets can be approximated for any system and controller. However, the guarantee that these reachable sets are conservative relies on additional padding or a large number of particles, and the more particles considered, the longer the computation time will be compared to conventional algorithms. This work is the closest to the one presented in this thesis and will be used as a baseline for comparison.

As mentioned in Section 2.3, *sensitivity*-based planning is a concept that explicitly accounts for parameters model uncertainty. However, in the work of [AM13; AM16], it is considered only in an open loop manner. This concept was recently expanded to include closed-loop sensitivity [BDR21; RDF18][1], which accounts for any controller applying feedback to an uncertain system, thus enabling the consideration of uncertainty for any robot/controller combination. This thesis leverages this *sensitivity* extension to address uncertainties in the parameters of the closed-loop system.

2.6 Outlook

Motion planning under uncertainty is a complex problem that has been approached through various methodologies. Over the past decade, planners have emerged that explicitly consider the feedback controller action in responding to uncertainties. However, the guarantees provided by these planners often depend on having an accurate model of both the robot and the controller. Consequently, uncertainties at this modeling level can compromise the robustness of these approaches. Building on recent advancements in closed-loop sensitivity analysis, this thesis introduces a novel class of robust control-aware motion planners (see Chapters 4 and 6). By leveraging widely used sampling-based motion planning techniques and the concept of closed-loop sensitivity, discussed in detail in Chapter 3, the proposed approach enhances robustness against parametric uncertainties and is applicable to a broad range of robots and controllers.

[1] For clarity, this thesis refers to this new concept as *sensitivity*.

Chapter 3
Preliminaries

This chapter introduces the notion of *closed-loop sensitivity*, which forms a key foundation of this thesis. This concept quantifies how variations of some model parameters (supposed to be uncertain) affect the evolution of the system in closed-loop, i.e., by also taking into account any controller chosen for executing a given task. Then, it is presented how this sensitivity notion can be leveraged to derive *uncertainty tubes* that bounds the system evolution both in the *input* and *state* spaces. These tubes are subsequently used to enforce robust constraints within the motion planning algorithms presented in this thesis. Finally, the differential drive robot and quadrotor models used in this manuscript are introduced.

3.1 Closed-loop sensitivity

3.1.1 Definition

Consider an arbitrary dynamic system with a set of uncertain parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$ (i.e. parameters that are difficult to model). The system dynamics can be described using the following set of ordinary differential equations (ODEs):

$$\dot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \boldsymbol{u}, \boldsymbol{p}), \quad \boldsymbol{q}(t_0) = \boldsymbol{q}_0, \tag{3.1}$$

where $\boldsymbol{q} \in \mathbb{R}^{n_q}$ is the system state vector and $\boldsymbol{u} \in \mathbb{R}^{n_u}$ is the control input vector. Also, assume the presence of a controller $\boldsymbol{\mu}$ of any form whose aim is to track a *desired trajectory* $\boldsymbol{q}_d(t)$ such that:

$$\begin{cases} \dot{\boldsymbol{\xi}} = \boldsymbol{g}(\boldsymbol{\xi}, \boldsymbol{q}, \boldsymbol{q}_d, \boldsymbol{p}_n, \boldsymbol{k}_c, t), \quad \boldsymbol{\xi}(t_0) = \boldsymbol{\xi}_0, \\ \boldsymbol{u} = \boldsymbol{\mu}(\boldsymbol{\xi}, \boldsymbol{q}, \boldsymbol{q}_d, \boldsymbol{p}_n, \boldsymbol{k}_c, t), \end{cases} \tag{3.2}$$

where $\boldsymbol{\xi} \in \mathbb{R}^{n_\xi}$ are the internal states of the controller (e.g., an integral action), $\boldsymbol{k}_c \in \mathbb{R}^{n_k}$ the controller gains, and $\boldsymbol{p}_n \in \mathbb{R}^{n_p}$ is the vector of "nominal" system parameters used in the control loop, i.e. the estimated nominal values of $\boldsymbol{p}$. Note that the controller operates based on the nominal parameters $\boldsymbol{p}_n$, while the actual parameters $\boldsymbol{p}$, which influence the system, remain unknown and cannot be determined. It is also important to note that, in this thesis, the system state $\boldsymbol{q}$ is assumed to be perfectly known, as no sensor models are considered. Consequently, the simulations and subsequent sensitivity computations are performed using the true system state $\boldsymbol{q}$, rather than the usual estimation of this state $\hat{\boldsymbol{q}}$.

In line with the previous definitions, the following sections of this thesis will then differentiate between three types of state vectors of key importance:

1. $\boldsymbol{q}_d$: The *desired* system state vector which refers to the desired values of the controllable system states. This state vector is typically the output of a motion planner.

Note that the dimension of this vector may differ from that of the real system because the vector represents a simplified or abstracted model, which might omit certain physical aspects or constraints that are present in the actual system, particularly in the case of under-actuated systems, where not all degrees of freedom are controlled.

2. $\boldsymbol{q}_n$: The *nominal* system state vector which represents the real state values of the system during the execution under nominal parameters (i.e. it is the behavior of the system obtained by integrating the ODEs with $\boldsymbol{p}$ perfectly matching the parameter values used in the control loop $\boldsymbol{p}_n$). A distinction is made between the nominal states and the desired states, as they may differ, for instance, due to the lack of feedforward terms.

3. $\boldsymbol{q}$: The *uncertain* (real) system state vector which refers to the behavior of the system when the values of real parameters are taken into account.

Note that the same notations apply to the control input vector as well (e.g., $\boldsymbol{u}_n$ represents the "nominal" control input values, i.e., when $\boldsymbol{p} = \boldsymbol{p}_n$).

It is possible to quantify how the presence of uncertain parameters (i.e. when the real system parameters $\boldsymbol{p}$ deviate from the nominal value $\boldsymbol{p}_n$ used in the control loop) affects the evolution of $\boldsymbol{q}(t)$ and $\boldsymbol{u}(t)$ according to the following matrices:

$$\boldsymbol{\Pi}(t) = \left.\frac{\partial \boldsymbol{q}(t)}{\partial \boldsymbol{p}}\right|_{\boldsymbol{p}=\boldsymbol{p}_n} \qquad \boldsymbol{\Theta}(t) = \left.\frac{\partial \boldsymbol{u}(t)}{\partial \boldsymbol{p}}\right|_{\boldsymbol{p}=\boldsymbol{p}_n} \tag{3.3}$$

where $\boldsymbol{\Pi}(t) \in \mathbb{R}^{n_q \times n_p}$ and $\boldsymbol{\Theta}(t) \in \mathbb{R}^{n_u \times n_p}$ are respectively defined in [BDR21; RDF18] as the *state-sensitivity matrix* and the *input-sensitivity matrix*. A closed-form expression for Equation (3.3) is, in general, not available. However, as shown in [BDR21; RDF18], their evolution in time can be computed by differentiating Equation (3.3) according to the following set of ODEs:

$$\begin{cases} \dot{\boldsymbol{\Pi}}(t) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}}\boldsymbol{\Pi} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\boldsymbol{\Theta} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}}, \quad \boldsymbol{\Pi}(t_0) = \boldsymbol{\Pi}_0, \\ \dot{\boldsymbol{\Pi}}_\xi(t) = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{q}}\boldsymbol{\Pi} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{\xi}}\boldsymbol{\Pi}_\xi, \quad \boldsymbol{\Pi}_\xi(t_0) = \boldsymbol{\Pi}_{\xi 0}, \\ \boldsymbol{\Theta}(t) = \frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{q}}\boldsymbol{\Pi} + \frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{\xi}}\boldsymbol{\Pi}_\xi \end{cases} \tag{3.4}$$

where $\boldsymbol{\Pi}_{\boldsymbol{\xi}}(t) \in \mathbb{R}^{n_\xi \times n_p}$ represents the *internal state sensitivity* matrix.

Now that the sensitivity matrices are defined, one can optimize the desired trajectory s.t. a norm of these matrices is minimized (see [BDR21; RDF18]). This optimization produces a trajectory s.t. the closed-loop evolution of $\boldsymbol{q}(t)$ and $\boldsymbol{u}(t)$ closely matches their evolutions in the nominal case $\boldsymbol{q}_n(t)$ and $\boldsymbol{u}_n(t)$.

3.1.2 Tube computation

Another important feature of the sensitivity matrices is that it is possible to derive the so-called *uncertainty tubes*, that bounds the closed-loop system trajectory $\boldsymbol{q}(t)$ and $\boldsymbol{u}(t)$ around their nominal trajectory $\boldsymbol{q}_n(t)$ and $\boldsymbol{u}_n(t)$, as shown in [Afi+24] and illustrated in Figure 3.1. The rest of this subsection shows how to establish such bounds around the

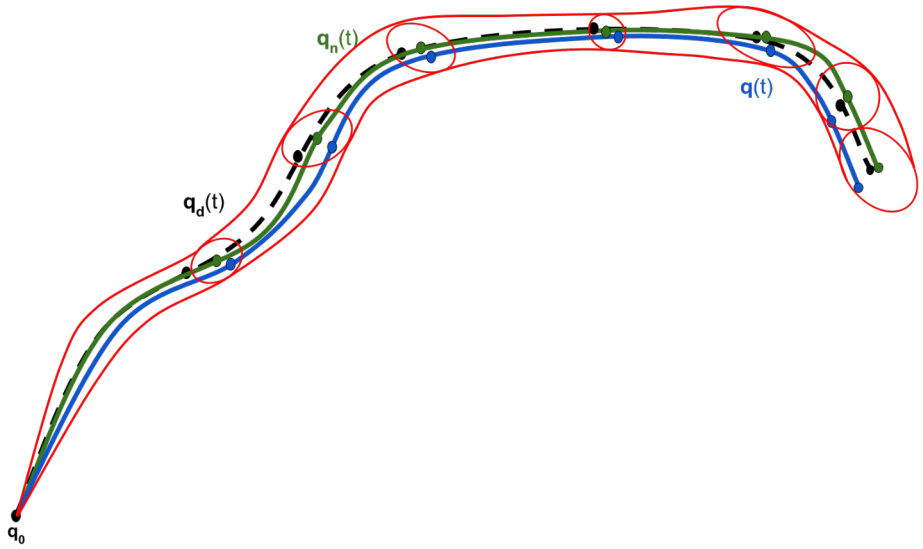


Figure 3.1: Illustration of the uncertainty tube (red) and a perturbed trajectory $\boldsymbol{q}(t)$ (blue), centered around the nominal trajectory $\boldsymbol{q}_n(t)$ (green), which results from following the reference trajectory $\boldsymbol{q}_d(t)$ (dashed black).

nominal state trajectory $\boldsymbol{q}_n(t)$ by focusing solely on the state-sensitivity matrix $\boldsymbol{\Pi}(t)$ for clarity. However, it is important to note that the same procedure can also be applied to compute bounds around the nominal input trajectory $\boldsymbol{u}_n(t)$ by leveraging the input-sensitivity matrix $\boldsymbol{\Theta}(t)$.

Assume that for each uncertain parameter $p_i, i \in [1, n_p]$ in $\boldsymbol{p}$, we have a bounded parameter deviation $\delta p_i \in \mathbb{R}$ s.t.

$$\forall i \in [1, n_p], p_i \in [p_{n_i} - \delta p_i, p_{n_i} + \delta p_i].$$

Such deviations can be mapped into the parameter space by means of the following ellipsoid

$$\Delta \boldsymbol{p}^T \boldsymbol{W}^{-1} \Delta \boldsymbol{p} \leq 1, \tag{3.5}$$

where $\boldsymbol{W}$ is the following diagonal weight matrix

$$\boldsymbol{W} = \begin{bmatrix} \delta p_1^2 & 0 & \cdots & 0 \\ 0 & \delta p_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \delta p_{n_p}^2 \end{bmatrix} \in \mathbb{R}^{n_p \times n_p}.$$

Assuming small parameters variations (i.e. small $\delta \boldsymbol{p}$) it is possible to perform a first-order approximation around the nominal trajectory $\boldsymbol{q}_n(t)$ to obtain

$$\Delta \boldsymbol{q}(t) = \boldsymbol{q}(t) - \boldsymbol{q}_n(t) \approx \boldsymbol{\Pi}(t) \Delta \boldsymbol{p}. \tag{3.6}$$

Without loss of generality, for a well-chosen $\delta \boldsymbol{p}$ s.t. Equation (3.6) holds, [Afi+24] has shown how to map the parameters ellipsoid from Equation (3.5) in the system state

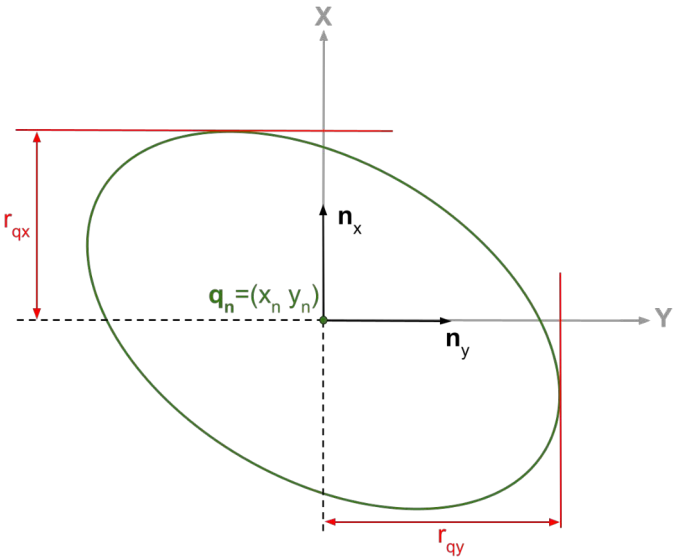


Figure 3.2: 2D representation of an uncertainty ellipse (green) in the x-y state space centered at $\boldsymbol{q}_n$, along with the tube radius (red) that illustrates the worst-case deviations along each state space components.

space in order to obtain the corresponding uncertainty ellipsoid

$$\Delta \boldsymbol{q}(t)^T (\boldsymbol{\Pi}(t) \boldsymbol{W} \boldsymbol{\Pi}(t)^T)^\dagger \Delta \boldsymbol{q}(t) \leq 1. \tag{3.7}$$

However, it is important to note that the ellipsoid axes are in general not aligned with the canonical basis of the state space. This implies that computing the deviation of each state component $q_i(t)$ is not straightforward, as direct use of the ellipsoid semi-axes length is not possible.

Nevertheless, it has been shown in [Afi+24] how to evaluate the radius $r_{q,i}(t) \geq 0$ of the perturbed tubes along the i-th component of the state $q_i(t)$ by mean of the following projection:

$$r_{q,i}(t) = \sqrt{\boldsymbol{n_i}^T \boldsymbol{K}_{\boldsymbol{\Pi}}(t) \boldsymbol{n_i}}, \tag{3.8}$$

where $\boldsymbol{n}_i$ is the unit-norm vector of the i-th state space component, and $\boldsymbol{K}_{\boldsymbol{\Pi}}(t) = \boldsymbol{\Pi}(t) \boldsymbol{W} \boldsymbol{\Pi}(t)^T$ is the kernel matrix of Equation (3.7). It is important to note that the radius along the i-th component represents the maximum possible deviation in that direction and does not directly correspond to the semi-axes of the uncertainty ellipsoid as depicted in Figure 3.2.

The quantities $r_{q,i}(t)$ can then be used to bound components of the states $\boldsymbol{q}(t)$ over time. For example, let $q_{n,i}(t)$ be the behavior of the i-th state component in the nominal case (non-perturbed case), the so-called *uncertainty tube* along this component is defined s.t.:

$$q_{n,i}(t) - r_{q,i}(t) \leq q_i(t) \leq q_{n,i}(t) + r_{q,i}(t), \tag{3.9}$$

for a perturbed state $q_i(t)$, and an analogous derivation is also possible for the control inputs $\boldsymbol{u}(t)$.

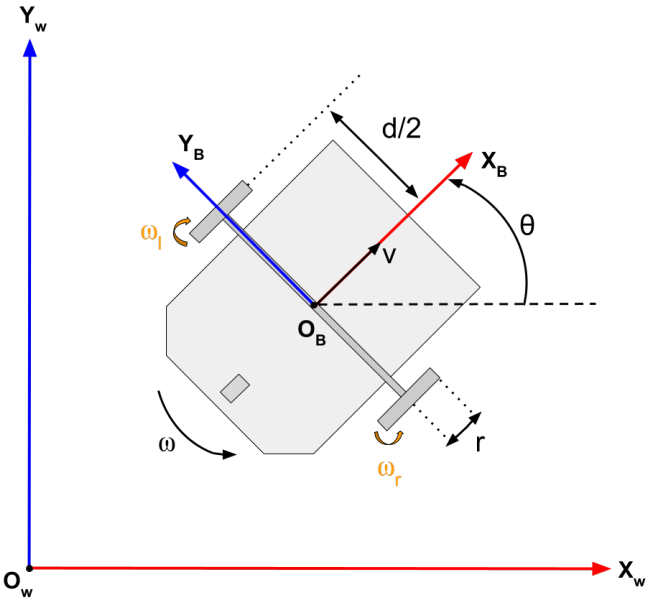


Figure 3.3: Illustration of the main quantities characterizing the differential drive robot.

Finally, it is important to underline that, since the uncertainty tube radii depend on the state sensitivity $\mathbf{\Pi}(t)$, it is necessary to solve the set of ODEs represented by Equation (3.1), Equation (3.2), and Equation (3.4) beforehand. Additionally, it is important to note that the tube radius does not necessarily increase monotonically, as shown in Figure 3.1, possibly due to the influence of feedback actions. Despite the cumulative effect of uncertainties over time, the feedback helps to contain and reduce the tube radius. In contrast, under open-loop propagation, these uncertainty tubes would expand due to the lack of knowledge about the control feedback actions that mitigate their growth.

3.2 Models considered in this thesis

To demonstrate the versatility of the approaches proposed in this thesis across various robots and controllers, a quadrotor and a differential drive robot are considered as benchmarks of the techniques presented in the next chapters, along with two distinct controllers and interpolation methods.

3.2.1 Differential drive robot

3.2.1.1 Dynamic model

Let the world frame be defined as $F_W = \{O_W, X_W, Y_W\}$ and $F_B = \{O_B, X_B, Y_B\}$ be the differential drive robot body frame attached to its geometric center (O_B). The robot state is defined as $\boldsymbol{q} = [\boldsymbol{x}\, \theta] \in \mathbb{R}^3$ where $\boldsymbol{x} = [x,\, y] \in \mathbb{R}^2$ are the linear positions of O_B in F_W and θ is the body heading (see Figure 3.3).

Let define the control input vector as $\boldsymbol{u} = [\omega_r\, \omega_l]^T \in \mathbb{R}^2$, where $(\omega_r,\, \omega_l)$ are respectively the right and left wheel angular velocity. Let also the robot linear and angular velocities

be denoted by v and ω, respectively. The aforementioned linear and angular velocities are related to the control input vector by mean of an allocation matrix $\boldsymbol{T}$ s.t.:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{2d} & \frac{-r}{2d} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} = \boldsymbol{T}\boldsymbol{u}, \tag{3.10}$$

where d is the length between the two wheels, and r is the wheel radius.

The differential drive robot kinematic model can be expressed as:

$$\dot{\boldsymbol{q}} = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{T}\boldsymbol{u}. \tag{3.11}$$

In line with Section 3.1.1, the vector of parameters used in the robot model outlined above that are subject to uncertainty is defined as $\boldsymbol{p} = [r\, d]^T \in \mathbb{R}^2$.

The control task is to let the robot positions $\boldsymbol{x}$ track desired positions $\boldsymbol{x}_d = [x_d,\, y_d] \in \mathbb{R}^2$. The differential drive robot is an under-actuated system, which is why the heading is excluded from the tracking process; it is induced by the robot dynamic instead.

To perform the tracking task, a dynamic feedback linearization (DFL) controller is used (e.g. see [ODV02]). This control strategy considers an extended desired trajectory $\boldsymbol{q}_d(t) = [\boldsymbol{x}_d(t)\, \dot{\boldsymbol{x}}_d(t)\, \ddot{\boldsymbol{x}}_d(t)]^T \in \mathbb{R}^6$, where $\dot{\boldsymbol{x}}_d(t) \in \mathbb{R}^2$ denote the desired linear velocities, and $\ddot{\boldsymbol{x}}_d(t) \in \mathbb{R}^2$ are the desired linear accelerations. This trajectory is tracked by mean of the following controller internal states $\boldsymbol{\xi} = [\xi_v\, \xi_x\, \xi_y]^T \in \mathbb{R}^3$, where ξ_v is the dynamic extension of the differential drive robot linear velocity v (see Figure 3.3), and $(\xi_x,\, \xi_y)$ are integral actions.

By differentiating the robot positions twice, one obtains the following equations for the robot linear accelerations:

$$\ddot{\boldsymbol{x}} = \begin{bmatrix} cos(\theta) & -\xi_v sin(\theta) \\ sin(\theta) & \xi_v cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{v} \\ \omega \end{bmatrix} = \boldsymbol{A} \begin{bmatrix} \dot{v} \\ \omega \end{bmatrix}$$

This differentiation is essential for formulating the dynamic relationships that enable efficient tracking of the desired positions.

Let the following vectors:

$$\begin{cases} \dot{\boldsymbol{x}}_\xi = [cos(\theta)\xi_v\; sin(\theta)\xi_v]^T \\ \boldsymbol{\xi}_{xy} = [\xi_x\, \xi_y]^T \\ \boldsymbol{\varrho} = \ddot{\boldsymbol{x}}_d + k_v(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}}_\xi) + k_p(\boldsymbol{x}_d - \boldsymbol{x}) + k_i\boldsymbol{\xi}_{xy} \end{cases} \tag{3.12}$$

where k_v, k_p and k_i are the controller gains, s.t. in the following chapters of this thesis, the controller gain vector is defined as $\boldsymbol{k}_c = [k_p,\, k_v\, k_i]^T \in \mathbb{R}^3$.

Without loss of generality, the dynamics of the internal control states can be expressed (refer to [ODV02] for further details) as follows:

$$\dot{\boldsymbol{\xi}} = \begin{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{A}^{-1}\boldsymbol{\varrho} \\ \boldsymbol{x}_d - \boldsymbol{x} \end{bmatrix}, \tag{3.13}$$

and the control inputs as:

$$\boldsymbol{u} = \boldsymbol{T}_n^{-1} \begin{bmatrix} \xi_v \\ \begin{bmatrix} 0 & 1 \end{bmatrix} \boldsymbol{A}^{-1}\boldsymbol{\varrho} \end{bmatrix}, \tag{3.14}$$

where $\boldsymbol{T}_n$ represent the same allocation matrix as in Equation 3.11 but evaluated on the nominal parameters $\boldsymbol{p}_n$.

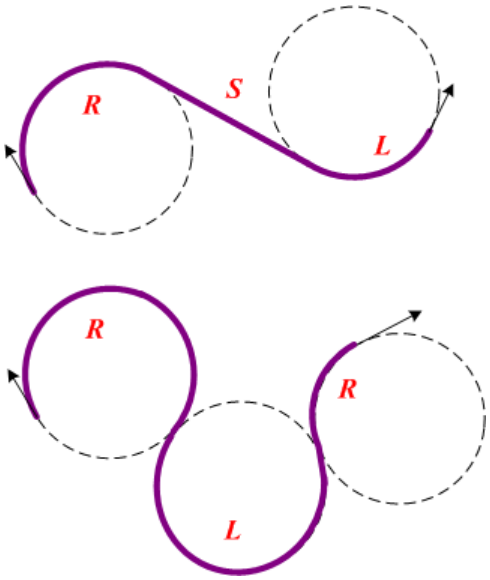


Figure 3.4: Illustration of two possible Dubins paths: (top) a right turn, followed by a straight segment, then a left turn; (bottom) a right turn, followed by a left turn, then another right turn. (extracted from [CZ18])

3.2.1.2 State interpolation

Now that the dynamics of the differential drive robot and its controller are defined, this subsection presents the steering method used to generate the desired trajectory $\boldsymbol{q}_d(t) = [\boldsymbol{x}_d(t)\, \dot{\boldsymbol{x}}_d(t)\, \ddot{\boldsymbol{x}}_d(t)]^T \in \mathbb{R}^6$ that is tracked by the DFL controller. The trajectory generation problem is addressed using Dubins curves [Dub57]. This method is preferred over more aggressive approaches, such as Reeds and Shepp [RS90b] or the shortest path synthesis in [SL96], as it allows only forward motion. This restriction helps prevent the singularity in the DFL controller that occurs when the robot's velocity approaches zero (see [ODV02]), which happens during transitions from forward to backward motion.

The Dubins curves approach ensures smooth and continuous trajectories between two positions (x^0, y^0) and (x^F, y^F) in the x-y plane with constraints on turning radius and forward motion. The method takes advantage of specified initial and final tangents to the curve (e.g. the differential drive robot heading angle θ, in this case) to create smooth curves that respect the turning constraints. A Dubins curve is composed of three possible segment types: left turn, right turn, and straight line as depicted in Figure 3.4. Each segment is designed with an optimal distance, arranged to create the shortest path between the two points (x^0, y^0) and (x^F, y^F), while adhering to the constraints of minimum turning radius and forward motion. This configuration results in one of six possible combinations of segments (e.g., left-straight-right), producing a smooth, continuous path that meets the system turning and heading requirements.

In this thesis, the robot state used to compute such path is defined as $\boldsymbol{\gamma} = [x\, y\, \theta] \in \mathbb{R}^3$. Once the path is computed, the x and y components are discretized using a specified time step and a constant velocity magnitude denoted v_{magn}. The corresponding velocities along the x- and y-axes are then computed using the finite difference method. However, while the resulting velocities are continuous, they are generally not differentiable. Therefore,

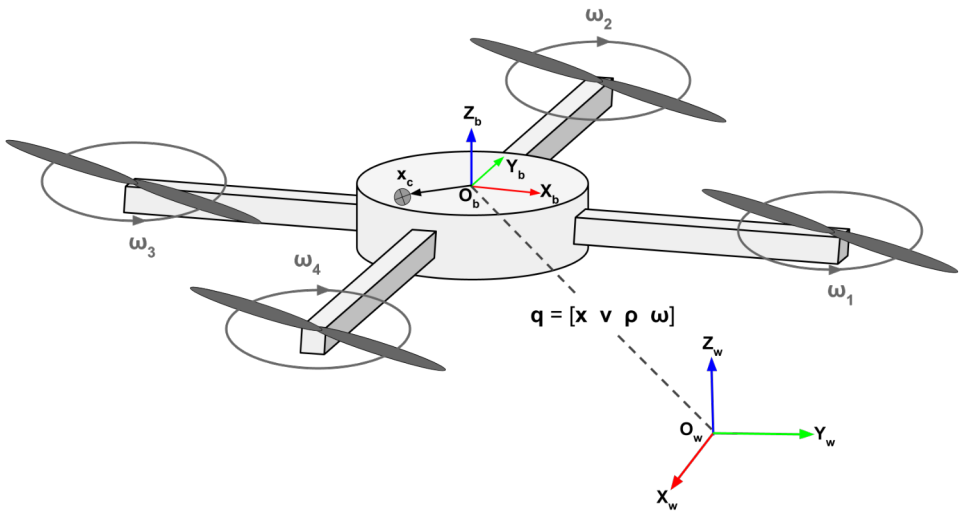


Figure 3.5: Quadrotor robot representation with a shift in the center of mass.

in this thesis, the desired accelerations required by Equation (3.12) are set to be zero. Note that, although the heading is not directly controlled, a desired heading is specified to satisfy the tangent constraints.

3.2.2 Quadrotor robot

3.2.2.1 Dynamic model

Let the ENU (East North Up) world frame be defined as $F_W = \{O_W, X_W, Y_W, Z_W\}$ and $F_B = \{O_B, X_B, Y_B, Z_B\}$ be the quadrotor body frame attached to its geometric center (O_B). The state of the quadrotor is defined as $\boldsymbol{q} = [\boldsymbol{x}\, \boldsymbol{v}\, \boldsymbol{\rho}\, \boldsymbol{\omega}]$ where $\boldsymbol{x} = [x\, y\, z] \in \mathbb{R}^3$ and $\boldsymbol{v} = [v_x\, v_y\, v_z] \in \mathbb{R}^3$ are respectively the position and linear velocity vector of O_B expressed in F_W. The body orientation w.r.t. F_W is represented by the unitary quaternion $\boldsymbol{\rho}$ and its angular velocity as $\boldsymbol{\omega} = [\omega_x\, \omega_y\, \omega_z] \in \mathbb{R}^3$. Finally, let $\boldsymbol{R(\rho)}$ be the rotation matrix associated to $\boldsymbol{\rho}$.

Let the control input vector $\boldsymbol{u} = [\omega_1^2\, \omega_2^2\, \omega_3^2\, \omega_4^2]^T$ represents the squared rotor speeds. The vector $\boldsymbol{u}$ is related to the total propeller thrust f and torques $\boldsymbol{\tau}$ by mean of a standard allocation matrix $\boldsymbol{T}$ s.t.

$$\begin{bmatrix} f \\ \boldsymbol{\tau} \end{bmatrix} = \boldsymbol{T}(l, k_f, k_\tau)\boldsymbol{u}, \tag{3.15}$$

where k_f and k_τ refer to the thrust and drag coefficients of the propellers respectively, and l stands for the quadrotor arm length. Also, consider that the center of mass (CoM) is displaced from the robot geometric center of an offset $\boldsymbol{x_c} = [x_{cx}\, x_{cy}\, x_{cz}]$ expressed in F_B as depicted in Figure 3.5. This displacement can occur due to onboard sensors, the presence of a payload, or other factors. Under this consideration and according to Newton's second law, the total force ($\boldsymbol{f_{tot}}$) and torque ($\boldsymbol{\tau_{tot}}$) acting on the quadrotor can be expressed by taking an additional fictitious force due to the displaced CoM in F_B s.t.

$$\begin{aligned} \boldsymbol{f_{tot}} &= f Z_W - mg\boldsymbol{R(\rho)}^T Z_W - m[\boldsymbol{\omega}]_\times [\boldsymbol{\omega}]_\times \boldsymbol{x_c} \\ \boldsymbol{\tau_{tot}} &= \boldsymbol{\tau} - mg[\boldsymbol{x_c}]_\times \boldsymbol{R(\rho)}^T Z_W - [\boldsymbol{\omega}]_\times \boldsymbol{J\omega} \end{aligned} \tag{3.16}$$

where m is the mass and $\boldsymbol{J}$ is the inertia matrix of the system.

By considering the spatial inertia matrix

$$\boldsymbol{S} = \begin{pmatrix} m\boldsymbol{I_3} & -m[\boldsymbol{x_c}]_\times \\ m[\boldsymbol{x_c}]_\times & \boldsymbol{J} \end{pmatrix}$$

one finally gets the body frame linear acceleration $\boldsymbol{\alpha}$ and angular acceleration $\boldsymbol{\eta}$ as: $\left(\boldsymbol{\alpha}^T\ \boldsymbol{\eta}^T\right)^T = \boldsymbol{S}^{-1}\left(\boldsymbol{f_{tot}}^T\ \boldsymbol{\tau_{tot}}^T\right)^T$. The dynamic model is then defined as follows:

$$\dot{\boldsymbol{q}} = \begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{v} \\ \dot{\boldsymbol{v}} = \boldsymbol{\alpha} \\ \dot{\boldsymbol{\rho}} = \frac{1}{2}\boldsymbol{\rho} \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \\ \dot{\boldsymbol{\omega}} = \boldsymbol{\eta} \end{cases} \tag{3.17}$$

In line with Section 3.1.1, let the vector $\boldsymbol{p} = [m\, x_{cx}\, x_{cy}\, x_{cz}\, J_x\, J_y\, J_z\, k_f\, k_\tau]^T \in \mathbb{R}^9$ with J_x, J_y, and J_z the main inertia coefficients, represent all the parameters used in the robot model outlined above that are subject to uncertainty. This vector can be adjusted based on the various scenarios presented in this thesis.

Finally, as tracking controller, consider the widely used Lee (or geometric) controller [LLM10] to compute the control input vector $\boldsymbol{u}$. This controller takes advantage of the well-known quadrotor flat outputs [MK11b] to track a desired trajectory defined as $\boldsymbol{q}_d(t) = [\boldsymbol{x}_d(t)\, \boldsymbol{v}_d(t)\, \boldsymbol{a}_d(t)\, \psi_d(t)\, \dot{\psi}_d(t)]^T \in \mathbb{R}^{11}$ composed of the desired linear positions, velocities, accelerations, yaw orientation angle, and yaw angular velocity. Note that, since the quadrotor is an under-actuated system, not all states can be controlled. This is why only the desired yaw angle is incorporated into the desired trajectory.

Let the linear position error and linear velocity error be defined as follows:

$$\boldsymbol{e_x}(t) = \boldsymbol{x}(t) - \boldsymbol{x}_d(t) \in \mathbb{R}^3,\ \boldsymbol{e_v}(t) = \boldsymbol{v}(t) - \boldsymbol{v}_d(t) \in \mathbb{R}^3, \tag{3.18}$$

and the controller gains vector $\boldsymbol{k}_c = [\boldsymbol{k}_x^T\, \boldsymbol{k}_v^T\, \boldsymbol{k}_R^T\, \boldsymbol{k}_\omega^T]^T \in \mathbb{R}^{12}$ with their associated diagonal representation $(\boldsymbol{K_x}, \boldsymbol{K_v}, \boldsymbol{K_R}, \boldsymbol{K_\omega})$. This control strategy starts by computing the desired third basis vector of the body frame:

$$\boldsymbol{r}_{3d} = \frac{-\boldsymbol{K_x} \cdot \boldsymbol{e_x} - \boldsymbol{K_v} \cdot \boldsymbol{e_v} + m(gZ_W + \boldsymbol{a}_d)}{\|-\boldsymbol{K_x} \cdot \boldsymbol{e_x} - \boldsymbol{K_v} \cdot \boldsymbol{e_v} + m(gZ_W + \boldsymbol{a}_d)\|} \in \mathbb{R}^3. \tag{3.19}$$

Then, using the desired yaw angle ψ_d, the desired heading vector can be defined as $\boldsymbol{\Omega_{\psi_d}} = [cos(\psi_d)\, sin(\psi_d)\, 0]^T$. This heading vector is then used to compute the desired first and second basis vectors of the body frame:

$$\boldsymbol{r}_{2d} = \frac{[\boldsymbol{r}_{3d}]_\times \cdot \boldsymbol{\Omega}_{\psi_d}}{\left\|[\boldsymbol{r}_{3d}]_\times \cdot \boldsymbol{\Omega}_{\psi_d}\right\|} \in \mathbb{R}^3,\ \boldsymbol{r}_{1d} = [\boldsymbol{r}_{2d}]_\times \cdot \boldsymbol{r}_{3d} \in \mathbb{R}^3, \tag{3.20}$$

where $[\,]_\times$ is the skew-symmetric matrix operator. From the three body frame basis vectors it is then possible to define the desired rotation matrix $\boldsymbol{R}_d(\psi_d, t) = [\boldsymbol{r}_{1d}\, \boldsymbol{r}_{2d}\, \boldsymbol{r}_{3d}]$, and to compute the following attitude error:

$$\boldsymbol{e_R}(t) = \frac{1}{2}[\boldsymbol{R}_d(\psi_d, t)^T \cdot \boldsymbol{R}(\boldsymbol{\rho}, t) - \boldsymbol{R}(\boldsymbol{\rho}, t)^T \cdot \boldsymbol{R}_d(\psi_d, t)^T]^\vee \in \mathbb{R}^3, \tag{3.21}$$

where $[]^\vee$ is the vee matrix operator. Finally, to simplify the overall controller design, the desired yaw rate $\dot{\psi}_d$ is always set to zero in this thesis, allowing the angular velocity error to be expressed as:

$$\boldsymbol{e_\omega}(t) = \omega(t) \in \mathbb{R}^3. \tag{3.22}$$

According to the aforementioned errors, this control strategy computes the desired propeller thrust $f_d(t)$ and desired propeller torques $\boldsymbol{\tau}_d(t)$ that allow the tracking of the desired trajectory $\boldsymbol{q}_d(t)$:

$$\begin{cases} f_d(t) &= (-\boldsymbol{K_x} \cdot \boldsymbol{e_x}(t) - \boldsymbol{K_v} \cdot \boldsymbol{e_v}(t) + m(gZ_W + \boldsymbol{a}_d(t)))^T \cdot (\boldsymbol{R}(\boldsymbol{\rho}, t) \cdot Z_W) \\ \boldsymbol{\tau}_d(t) &= -\boldsymbol{K_R} \cdot \boldsymbol{e_R}(t) - \boldsymbol{K_\omega} \cdot \boldsymbol{e_\omega}(t) \end{cases} \tag{3.23}$$

Note that the choice of setting the desired angular velocities to zero cancels terms in the expression of $\boldsymbol{\tau}_d(t)$ compared to the original expression in [LLM10].

Finally, the control input vector $\boldsymbol{u}$ can be computed using the same standard allocation matrix $\boldsymbol{T}$ from Equation (3.15) using nominal parameters denoted $\boldsymbol{T}_n$ s.t.:

$$\boldsymbol{u} = \boldsymbol{T}_n^{-1} \begin{bmatrix} f_d \\ \boldsymbol{\tau}_d \end{bmatrix}. \tag{3.24}$$

Note that, under the nominal case (i.e. when $\boldsymbol{p} = \boldsymbol{p}_n$), $\boldsymbol{T} = \boldsymbol{T}_n$; thus, the propeller thrust and torques applied to the real system, as for f and $\boldsymbol{\tau}$ in Equation (3.16), perfectly match the desired thrust and torques computed by the control law. Note also that no internal controller states are considered in this controller; therefore, Equation (3.4) simplifies to:

$$\begin{cases} \dot{\boldsymbol{\Pi}}(t) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}} \boldsymbol{\Pi} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} \boldsymbol{\Theta} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}}, \quad \boldsymbol{\Pi}(t_0) = \boldsymbol{\Pi}_0, \\ \boldsymbol{\Theta}(t) = \frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{q}} \boldsymbol{\Pi} \end{cases} \tag{3.25}$$

3.2.2.2 State interpolation

With the quadrotor model and controller now defined, this subsection presents the interpolation method (hereafter referred to as the *steering method*) for computing a desired trajectory that the geometric controller will track.

Let $\boldsymbol{\gamma} = [x_d\, y_d\, z_d\, \Psi_d]^T \in \mathbb{R}^4$ represents the desired position along the x, y, and z axes and the desired yaw angle for the quadrotor. Then, given an initial desired state $\boldsymbol{q}_d^0 = [\boldsymbol{\gamma}^0\, \dot{\boldsymbol{\gamma}}^0\, \ddot{\boldsymbol{\gamma}}^0] \in \mathbb{R}^{12}$ and a goal state $\boldsymbol{q}_d^g = [\boldsymbol{\gamma}^g\, \dot{\boldsymbol{\gamma}}^g\, \ddot{\boldsymbol{\gamma}}^g] \in \mathbb{R}^{12}$ for the quadrotor, the *kinospline* method from [BCS19] is employed to plan a time optimal and continuous desired trajectory $\boldsymbol{q}_d(t)$ connecting $\boldsymbol{q}_d^0$ and $\boldsymbol{q}_d^F$.

The trajectory generation problem is solved independently for each component of $\boldsymbol{\gamma}$. Furthermore, in order to generate smooth trajectories in a global context, the steering method ensures the continuity of their derivatives up to the jerk (acceleration derivative). Note that even if the yaw angular acceleration is not required by the control law of Section 3.2.2, the initial and goal yaw angular accelerations are used to ensure the smoothness of the generated trajectory. As for both initial and goal jerk, they are set to zero to facilitate a smooth transition between trajectories, and are therefore not

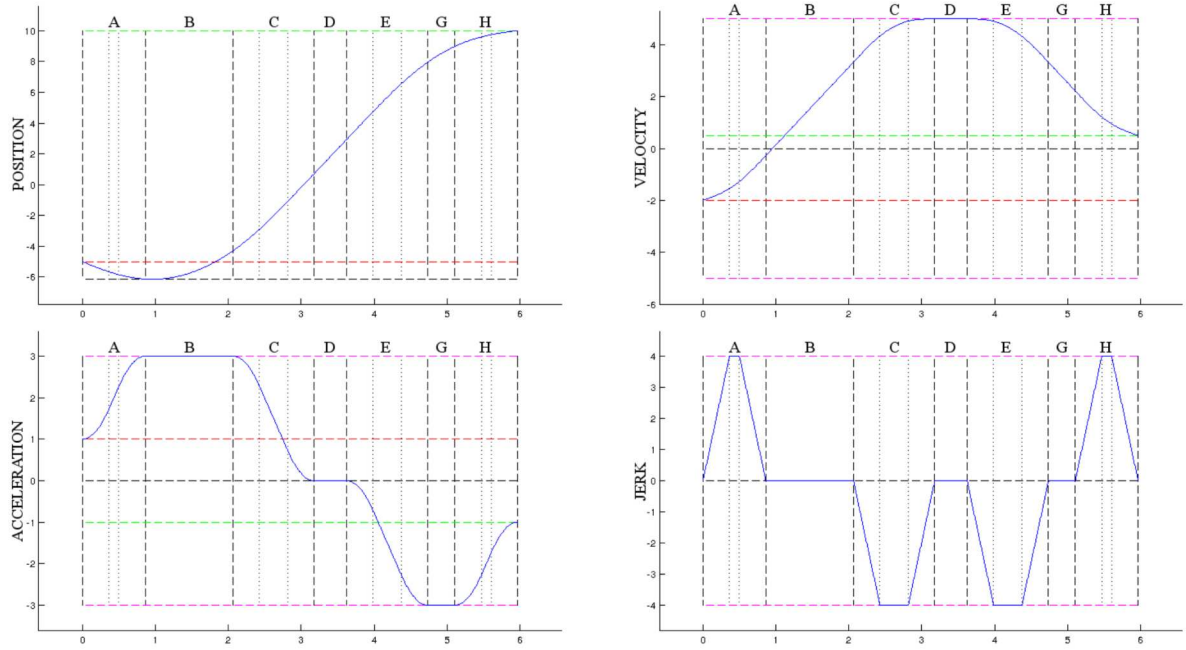


Figure 3.6: Example of a trajectory generated by the steering method for a single coordinate. The pink dashed lines indicate the bounds for each derivative, while the red and green dashed lines represent the initial and final values, respectively. The letters A, B, C, D, E, G, and H represent the different phases that need to be minimized or maximized during the trajectory planning process.(extracted from [BCS19])

considered in $\boldsymbol{\gamma}$. Finally, the kino-spline method also considers bounds on the several derivatives up to the snap (i.e. v_{max}, a_{max}, etc.) and plan the trajectory accordingly.

As previously mentioned, this steering method focuses on generating local trajectories that are time optimal, i.e. that minimize the total time needed to reach $\boldsymbol{q}_d^q$. This is achieved by reaching the full speed as soon as possible and by maintaining it as long as possible for each component of $\boldsymbol{\gamma}$. This also implies that the time spent reaching this velocity must be minimized, which means that the time spent at maximum acceleration during transient phases must be maximized. The same principle applies to jerk and snap; during variations in acceleration, the maximum achievable jerk should be maintained for as long as possible, while the duration of jerk variation phases is minimized. This is achieved through a straightforward bang-bang snap method, which aims to maximize the time spent at maximum snap during the jerk transient phases. By doing so, the duration of transient phases is minimized, and gradually, the duration of the maximum speed phase is maximized. The principle of this trajectory generation is illustrated by Figure 3.6.

Chapter 4

Robust Motion Planning with Global Sensitivity Optimization

This chapter introduces the first major contribution of this thesis, by leveraging the sensitivity concept discussed in Chapter 3. The contribution of this chapter is twofold:

1. Firstly, it addresses the generation of a desired trajectory with minimal sensitivity, ensuring that the closed-loop evolution of $\boldsymbol{q}(t)$ closely matches its nominal evolution, $\boldsymbol{q}_n(t)$. While previous works have explored sensitivity optimization for generating locally sensitivity-optimal trajectories [BDR21; RDF18], this approach has never been applied within a global sampling-based planning framework that accounts for obstacles. Therefore, the first contribution is to propose motion planners for efficiently performing global sensitivity optimization.

2. Secondly, this chapter introduces, for the first time, the use of sensitivity-based uncertainty tubes to enforce constraints in both the state and input spaces. This enables the generation of intrinsically-robust motions (i.e. accounting for the controller behavior with respect to uncertainty) that ensures robustness with respect to collision avoidance with obstacles of the environment, and also with the actuation limits.

This chapter is organized as follows: Section 4.1 presents a unified approach that combines robust trajectory planning with global sensitivity optimization by integrating the sensitivity computation and tube generation from Section 3.1.2 directly into an optimal tree-based planner, such as Asymptotically Optimal Rapidly-exploring Random Trees (RRT*) or Stable Sparse RRT* (SST*). This method enables the simultaneous generation of globally robust and sensitivity-optimal trajectories. The approach is evaluated on the robot models discussed in Chapter 3. Then, Section 4.2 introduces a decoupled approach that sacrifices the global optimality of the unified method in favor of computational efficiency, demonstrating improved scalability for more complex systems while maintaining near sensitivity optimality. Finally, some conclusions are drawn in Section 4.3.

This chapter is related to the ICRA 2023 publication [Was+23][1].

4.1 Global robust sensitivity-optimal planning

This section presents how the sensitivity matrices and resulting uncertainty tube computations, outlined in Section 3.1, can be integrated into an asymptotically-optimal tree

[1] The results presented in this chapter slightly differ from those in the article due to improvements in implementation.

planner to generate robust and sensitivity-optimal trajectories. It introduces a unified approach that allows the planning of globally robust trajectories while simultaneously optimizing sensitivity within a single planner.

The core idea of the method is to compute the state/input sensitivity matrices (and the resulting uncertainty tubes) at each iteration of the sampling-based algorithm. As detailed in Section 3.1, these sensitivity matrices and uncertainty tubes are computed by forward-integrating a system of ODEs (see Equations (3.1), (3.2), and (3.4)). Therefore, in the context of a sampling-based approach, this integration of the ODEs must be performed starting from a set of non-zero initial conditions (e.g., $\mathbf{\Pi}_0$, $\boldsymbol{q}^0$, etc.), denoted S_0, whenever extending a node that is not the root.

The management of these initial conditions is achieved by embedding the set of final conditions (e.g., $\mathbf{\Pi}_F$, $\boldsymbol{q}^F$, etc.), denoted S_F, into the node data after each extension. These final conditions are then reused as initial conditions for future extensions. It is important to note that the initial conditions of a given node are determined by the trajectory from the root node. Consequently, these initial conditions are specific to the parent node, making this approach applicable only in tree-based planners, where each node has exactly one parent. In contrast, graph-based planners, such as PRM [Kav+96], cannot leverage this mechanism due to their higher degree of connectivity. Additionally, note that, since dynamics is taken into account, the algorithms presented in this thesis utilize directed data structures (i.e., structures like directed trees that represent dependencies and directionality in motion).

This mechanism is then applied in Sections 4.1.2 and 4.1.3 to the asymptotically optimal tree-based planners introduced in the following subsection, with the aim of developing robust, sensitivity-aware variants, commonly referred to as Sensitivity-Aware Motion Planner (SAMP). Note that, while the subsequent sections of this chapter focus on asymptotically optimal planner variants for generating globally robust sensitivity-optimal trajectories, these principles can also be easily adapted to non-asymptotically optimal planners (see Chapter 6), such as the standard RRT.

4.1.1 Standard asymptotically optimal planners

This subsection presents the standard asymptotically optimal sampling-based planners, which form the basis for the SAMP variants developed in Sections 4.1.2 and 4.1.3.

Standard RRT* The vanilla Asymptotically Optimal Rapidly-exploring Random Trees (RRT*) [KF11] is a widely used asymptotically optimal sampling-based planner that extends the standard RRT algorithm [LaV98] by incorporating a rewiring step to iteratively improve the quality of the trajectories in the generated tree. It achieves this by first optimally connecting the samples to the tree and then re-evaluating the connections between sampled nodes and their neighbors based on a cost metric. This process ensures that the tree progressively converges toward an optimal solution as the number of samples increases. The RRT* tree extension process and tree rewiring phase are illustrated in Figure 4.1.

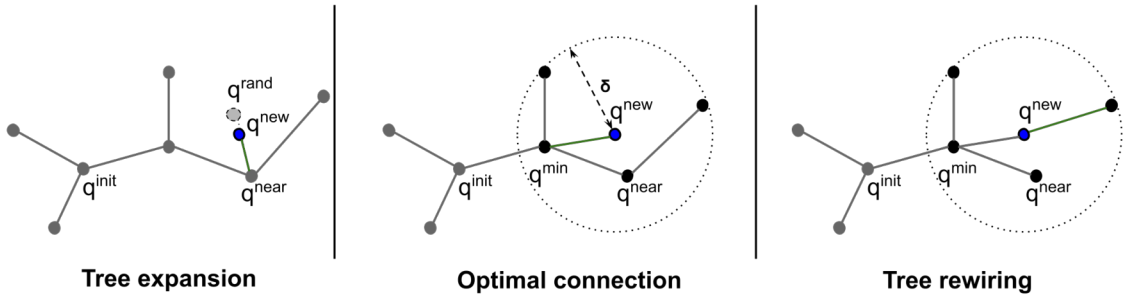


Figure 4.1: Illustration of the Asymptotically Optimal Rapidly-exploring Random Trees (RRT*) tree extension procedure. The first phase (left) consists of extending the nearest node in the tree, q^{near}, towards a randomly sampled state, q^{rand}, in order to add a new collision-free state, q^{new}, to the tree. Then, an optimal connection phase is performed (middle), where the new node is optimally connected to the node with the minimum cost, q^{min}, from the start, within a neighborhood defined by a distance δ. Finally, within this same neighborhood, a rewiring phase (right) is attempted to optimally reconnect the existing nodes if the cost of passing through the new state improves their current cost.

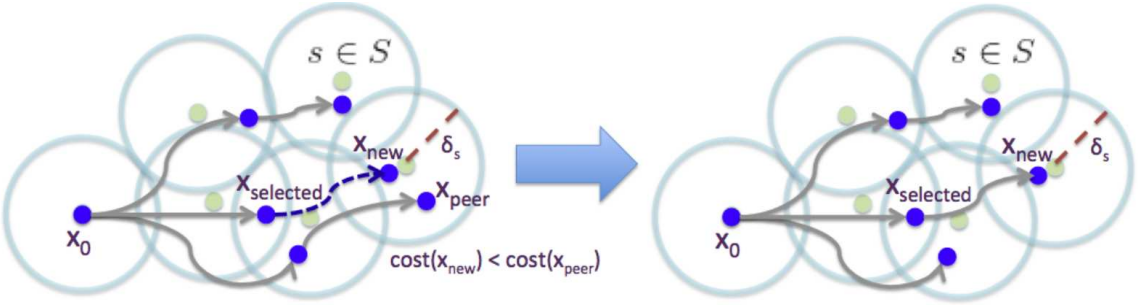


Figure 4.2: Illustration of the SST tree extension procedure where green dots represent witnesses, and blue dots are current witnesses representative (i.e. the node with the best cost in the witness neighborhood) (extracted from [LLB16]). Dynamic propagation from $x_{selected}$ generates a new node x_{new}, which has a better trajectory cost than its current witness representative node x_{peer}. In this case, x_{peer} is pruned, and the newly propagated edge is incorporated into the tree as the new witness representative. If x_{peer} had children with the lowest trajectory cost in their respective neighborhoods, x_{peer} would remain in the tree but would no longer be considered for further propagation. Inversely, if x_{new} has a worse trajectory cost than x_{peer}, the old node x_{peer} remains in the tree, and the propagation from $x_{selected}$ to x_{new} is ignored.

Standard SST* The standard Stable Sparse RRT* (SST*) [LLB16] is another asymptotically optimal sampling-based planner that, unlike RRT*, does not utilize a steering method to extend its tree but instead relies on system dynamics forward propagation. This method grows a tree in the state space by employing Monte Carlo forward dynamic propagation to explore feasible trajectories by sampling control inputs. To improve efficiency, SST* employs a pruning mechanism that preserves only the locally optimal solutions, thanks to a witness and representative system as illustrated in Fig-

ure 4.2. This approach ensures that the planner focuses on promising regions of the state space, reducing redundant computations while maintaining theoretical guarantees of near-optimality [LLB16]. By combining these features, SST* offers a powerful solution for kinodynamic motion planning in complex systems.

4.1.2 Sensitivity-Aware RRT* (SARRT*)

Algorithm 4.1 SARRT*$[q^{init}, q^{goal}]$

1: $T \leftarrow$ InitTree(q^{init}, q^{goal});
2: **while not** StopCondition(T, q^{goal}) **do**
3: $\quad q^{rand} \leftarrow$Sample();
4: $\quad q^{nearest} \leftarrow$ Nearest(T, q^{rand});
5: $\quad \boldsymbol{q}_d \leftarrow$ Steer$(q^{nearest}, q^{rand})$;
6: $\quad S_0 \leftarrow$GetNodeConditions$(q^{nearest})$;
7: $\quad \{\boldsymbol{q}_n, \boldsymbol{u}_n, \boldsymbol{r_q}, \boldsymbol{r_u}, S_F\} \leftarrow$SolveODEs$(\boldsymbol{q}_d, S_0)$;
8: $\quad$ **if** IsRobust$(q_n, \boldsymbol{u}_n, \boldsymbol{r_q}, \boldsymbol{r_u})$ **then**
9: $\quad\quad q^{min} \leftarrow q^{nearest}$;
10: $\quad\quad$ SetNodeConditions(q^{rand}, S_F);
11: $\quad\quad Q^{near} \leftarrow$ NearestNodes(T, q^{rand});
12: $\quad\quad$ RobustOptimalConnection$(T, Q^{near}, q^{rand}, q^{min})$;
13: $\quad\quad$ RobustRewire(T, Q^{near}, q^{min});
14: **return** GetTrajectory(T, q^{init}, q^{goal});

The first implementation presented in Algorithm 4.1[1] is a variant of the widely used RRT* [KF11], denoted as Sensitivity-Aware RRT* (SARRT*), as a particular instance of SAMP. The tree is initialized with the desired initial and goal robot states, (q^{init}, q^{goal}) (line 1), and the algorithm continues until a user-defined stopping condition is met (line 2). Such a criterion can include a maximum number of iterations, a maximum runtime, a cost convergence threshold (i.e., the algorithm will stop optimizing the trajectory when the improvement in cost from one iteration to the next is less than a given threshold), etc.

The first stage of the algorithm follows the same standard procedure as the vanilla RRT*, including sampling a desired robot state q^{rand} and finding its nearest neighbor $q^{nearest}$ in the tree structure (lines 3 and 4, respectively) using an efficient distance metric (further discussed in Section 4.1.5). Then, a steering method (such as the ones presented in Section 3.2.1.2 and 3.2.2.2) is employed to generate a desired local trajectory $\boldsymbol{q}_d$ that extends $q^{nearest}$ toward q^{rand}. Note that if a maximum local distance is imposed, q^{rand} is adjusted accordingly and set to the last state on the trajectory towards q^{rand} that

[1]For clarity in the pseudo-code, the temporal notation is omitted from this subsection. Thus, bold symbols refer to trajectory vectors (e.g., $\boldsymbol{q}_d$ represents a desired trajectory, $\boldsymbol{r_q}$ stands for the uncertainty tube radii along a trajectory, etc.), while non-bold symbols represent values at a single state (e.g., q^{rand} denotes a random state, and S_0 represents the initial conditions of the ODEs at a given trajectory state, etc.).

respects the maximum distance constraint. Next, the set of ODEs is solved (line 7) to compute the nominal trajectory $\boldsymbol{q}_n$, the nominal control inputs $\boldsymbol{u}_n$, and the uncertainty tubes' radii ($\boldsymbol{r_q}, \boldsymbol{r_u}$), utilizing the initial conditions S_0 (line 6).

A robust feasibility check is then performed along the nominal trajectory and control inputs (line 8). This test includes a robust collision check that accounts for the robot shape expansion due to uncertainty tubes along the state trajectory, as well as a robust verification that the control inputs do not saturate (for further details see Appendix A). Remember that this test is carried out along the nominal values, as the uncertainty tubes are valid around them, as detailed in Section 3.1.2. If the extension is feasible, the new node parent is set to $q^{nearest}$, and the final conditions S_F are stored within q^{rand} (line 9-10).

Algorithm 4.2 RobustOptimalConnection$[T, Q^{near}, q^{rand}, q^{min}]$

```
1:  c^min ← Cost(q^rand);
2:  for each q^near ∈ Q^near do
3:      q_d ← Steer(q^near, q^rand);
4:      S_0 ←GetNodeConditions(q^near);
5:      {q_n, u_n, r_q, r_u, S_F} ←SolveODEs(q_d, S_0);
6:      if IsRobust(q_n, u_n, r_q, r_u) then
7:          if Cost(q^near) + J(q_d) < c^min then
8:              q^min ← q^near; c^min ← Cost(q^near) + J(q_d);
9:              SetNodeConditions(q^rand, S_F);
10: AddNewNode(T, q^rand);
11: AddNewEdge(T, q^min, q^rand);
```

Next, the newly added node is initially connected to its nearest neighbor, but this connection does not consider the total trajectory cost from the root node. Therefore, an optimal connection procedure is performed. To determine the optimal parent of q^{rand}, which is temporarily assigned to q^{min}, an additional search is conducted within a specified neighborhood Q^{near}, as further detailed in [KF11](line 11). Such procedure, depicted in Algorithm 4.2, starts by computing the global cost (e.g. length, clearance, etc.) of the unique trajectory from the root node to q^{rand} (line 1). Then, the optimal parent q^{min} is set to the node in Q^{near} that allows a robust extension with the minimum global cost to get to q^{rand} (line 2-9). Finally, the new node is inserted in the tree with an edge connecting it to its optimal parent (line 10-11).

Now that the new state is globally optimally connected to the tree with a feasible extension, SARRT* checks if any optimal rewiring can be performed, as in the standard RRT* implementation, using Algorithm 4.3. This rewiring phase checks, for each neighbor q^{near} within Q^{near}, whether the trajectory from the root to q^{near}, passing through the newly added node q^{rand}, can be improved (lines 1-6). In such case, the parent of q^{near} is updated accordingly, and the final ODEs conditions S_F (e.g. final robot simulated state $\boldsymbol{q}^F$, etc.) are also modified (lines 7-10). Note that such final conditions are reused as initial conditions in subsequent expansions. Finally, in contrast to the standard RRT*, each child of the newly rewired node must be rechecked for robust feasibility, as their

Algorithm 4.3 RobustRewire$[T, Q^{near}, q^{min}]$

1: **for** each $q^{near} \in Q^{near} \backslash \{q^{min}\}$ **do**
2: $\quad \boldsymbol{q}_d \leftarrow \text{Steer}(q^{rand}, q^{near})$;
3: $\quad S_0 \leftarrow$GetNodeConditions(q^{rand});
4: $\quad \{\boldsymbol{q}_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}, S_F\} \leftarrow$SolveODEs$(\boldsymbol{q}_d, S_0)$;
5: $\quad$ **if** IsRobust$(q_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}})$ **then**
6: $\quad\quad$ **if** $\text{Cost}(q^{rand}) + J(\boldsymbol{q}_d) < \text{Cost}(q^{near})$ **then**
7: $\quad\quad\quad q^{parent} \leftarrow \text{GetParent}(q^{near})$;
8: $\quad\quad\quad \text{RemoveEdge}(T, q^{parent}, q^{near})$;
9: $\quad\quad\quad \text{AddNewEdge}(T, q^{rand}, q^{near})$;
10: $\quad\quad\quad \text{SetNodeConditions}(q^{near}, S_F)$;
11: $\quad\quad\quad \text{UpdateChildren}(T, q^{near})$;

uncertainty tubes and control inputs may have changed due to the updated trajectory resulting from the rewiring (line 11). Note that this final operation requires re-solving the ODEs for each child node, as the new trajectory from the root may modify the tubes, potentially resulting in trajectories that are no longer robust.

4.1.3 Sensitivity-Aware SST* (SASST*)

It is important to note that a single iteration of SARRT* presented above requires solving the set of ODEs multiple times which leads to a prohibitive computing time per iteration as it will be shown in Section 4.1.5. To mitigate this, this subsection proposes a robust sensitivity-aware variant of SST* [LLB16], denoted Sensitivity-Aware SST* (SASST*), as a particular instance of SAMP.

The SST* algorithm has two key features: it reduces the number of collision checks to just one per iteration, and it uses Monte Carlo propagation to sample the control input space and perform forward dynamic propagation in an open-loop manner, eliminating the need for a steering method. However, since the system closed-loop dynamics is required for sensitivity computation rather than open-loop Monte Carlo propagation, the latter is replaced by a desired trajectory computed by a steering method and subsequently tracked by the closed-loop system.

The SASST* variant, detailed in Algorithm 4.4, starts with planner parameters and tree initialization (line 1-4) by splitting it into two subsets: V_{active} and $V_{inactive}$. Nodes in V_{active} are those with the optimal trajectory cost from the root within their local neighborhoods. In contrast, nodes in $V_{inactive}$, even if suboptimal in trajectory cost, are preserved in the tree to maintain connectivity, as they have children with better trajectory costs in their respective neighborhoods.

Then, to define local neighbors, the algorithm leverages an auxiliary set of states called "witnesses," denoted as W (line 5). Each witness $w \in W$ is associated with a single representative node in the tree, stored in the $w.rep$ field of the witness (line 6). This representative node corresponds to the trajectory with the lowest cost from the root within a distance δ_s of the witness w. Any node within the δ_s-neighborhood of w that have a higher trajectory cost than $w.rep$ are removed from the active set V_{active}.

Algorithm 4.4 SASST*$[q^{init}, q^{goal}, N_0, \delta_{BN,0}, \delta_{s,0}, \xi]$

```
1: j ← 0; N ← N_0;
2: δ_s ← δ_{s,0}; δ_BN ← δ_{BN,0};
3: V_active ← {q^init, q^goal}, V_inactive ← ∅;
4: T ← InitTree(V_active, V_inactive);
5: w_0 ← q^init, w_G ← q^goal, W ← {w_0, w_G};
6: w_0.rep ← q^init, w_G.rep ← q^goal;
7: while not StopCondition(T, q^goal) do
8:     for N iterations do
9:         {q^selected, q^rand} ←BestFirstSelectionSST(V_active, δ_BN);
10:        q^rand ← Sample(q^selected, d_max);
11:        q_d ← Steer(q^selected, q^rand);
12:        S_0 ←GetNodeConditions(q^selected);
13:        {q_n, u_n, r_q, r_u, S_F} ←SolveODEs(q_d, S_0);
14:        if IsRobust(q_n, u_n, r_q, r_u) then
15:            if IsNodeLocallyTheBestSST(q^rand, W, δ_s) then
16:                V_active ← V_active ∪ {q^rand};
17:                AddNewEdge(T, q^selected, q^rand);
18:                PruneDominatedNodesSST(q^rand, V_active, V_inactive);
19:    δ_s ← ξδ_s; δ_BN ← ξδ_BN;
20:    j ← j + 1;
21:    N ← (1+log j)ξ^{-(d+1)j} N_0;
22: return GetTrajectory(T, q^init, q^goal);
```

Next, until a user-defined condition is met, the algorithm performs for N iterations:

1. A best-first selection that samples a random state from the state space and finds the best node in terms of cost from the root within a distance of δ_{BN} in the active set of the tree. This node, referred as $q^{selected}$, represents the state to be propagated (line 9).

2. Then, instead of the original Monte Carlo propagation, a random state q^{rand} is sampled within a maximum distance of $q^{selected}$ (line 10), analogous to sampling a control input and propagation duration in the vanilla algorithm. The local trajectory connecting the two state is computed (line 11) and, the set of ODEs is solved (line 12-13) before performing a robust feasibility check (line 14) (see Appendix A).

3. If the local trajectory is robustly feasible, the set W is utilized by the IsNodeLocallyTheBestSST procedure (line 15) to determine whether the newly sampled state q^{rand} dominates the δ_s-neighborhood of its nearest witness (i.e., whether q^{rand} has a lower cost than the current witness of the neighborhood it belongs to). The procedure then updates the witness set W accordingly.

4. Then, the sample is added to the tree (line 16-17), and the sets of active and

inactive nodes are updated. Nodes that no longer belong to any of these sets are pruned (line 18).

Finally, after the N iterations, if the stopping condition is not met, the hyperparameters of the algorithm are updated (line 19-21). These parameters are responsible for the algorithm asymptotic optimality and convergence. The algorithm gradually reduces the radii δ_s and δ_{BN}, enabling it to explore new homotopic classes over time.

Note that in this implementation the ODEs need to be solved only once per iteration compared to the SARRT* implementation.

4.1.4 Cost function

While the previously mentioned SAMP variants are sensitivity-aware in the sense that they leverage sensitivity-based uncertainty tubes, they can be applied to optimize various non sensitivity-based cost functions, such as length or obstacle clearance. However, as stated earlier, the goal is to generate robust and globally sensitivity optimized trajectories. To achieve this, the first step is to define an appropriate sensitivity-based cost function. The approaches in [BDR21; RDF18] proposed leveraging the Frobenius norm of the sensitivity matrices (Equation 3.3). However, this formulation does not exploit the available information on the uncertainty ranges, $\delta\boldsymbol{p}$, and the subsequent uncertainty tubes.

Therefore, it is preferable to consider the largest eigenvalue of the kernel matrix $\boldsymbol{\Pi}(t)\boldsymbol{W}\boldsymbol{\Pi}(t)^T$ (see Section 3.1.2) as *sensitivity norm*, since this represents the largest (worst-case) deviation in the state space. In particular, letting $\lambda_i(t)$ be the eigenvalues of $\boldsymbol{\Pi}(t)\boldsymbol{W}\boldsymbol{\Pi}(t)^T$, a smooth approximation of the $\max(\cdot)$ operator is performed with the p-norm:

$$\lambda_{max}(t) \approx \left(\sum \lambda_i(t)^p\right)^{1/p}. \tag{4.1}$$

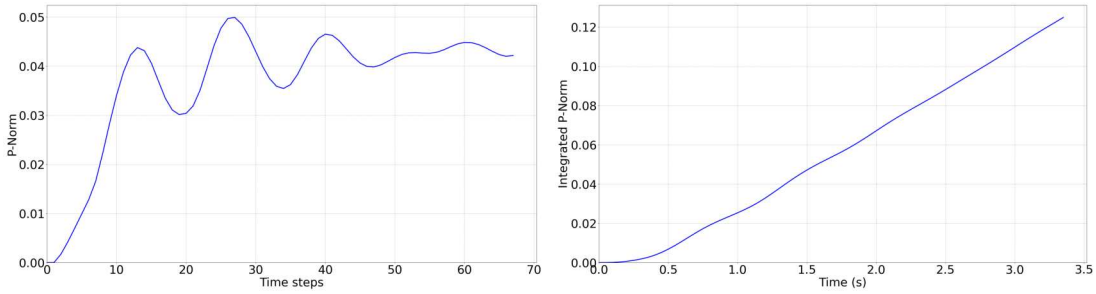
However, such function is neither additive (i.e., considering two desired trajectories $(\boldsymbol{q}_{d,1}(t), \boldsymbol{q}_{d,2}(t))$, the cost of their concatenation $c(\boldsymbol{q}_{d,1}(t)|\boldsymbol{q}_{d,2}(t)) \neq c(\boldsymbol{q}_{d,1}(t)) + c(\boldsymbol{q}_{d,2}(t)))$, nor monotonic as depicted in Figure 4.3. Therefore, it is unsuitable for global optimization using sampling-based motion planners like [KF11; LLB16], since they require additive and monotonic objective functions. The total cost function $c(\boldsymbol{q}_d(t))$ for a desired trajectory $\boldsymbol{q}_d(t)$ is then defined considering its integral along the desired trajectory s.t.:

$$c(\boldsymbol{q}_d(t)) = \int_{t_0}^{t_f} \lambda_{max}(\tau) d\tau. \tag{4.2}$$

Minimization of this cost will smooth deviation of the state tube along the whole motion.

4.1.5 Simulation results

This subsection presents the results of trajectory planning using the previously discussed variants, evaluated based on computing time and robustness for the two systems (quadrotor and differential drive robots) introduced in Chapter 3. To assess the impact of planning robust, globally sensitivity-optimal trajectories, both SARRT* and SASST*



(a) p-norm (b) Integrated p-norm

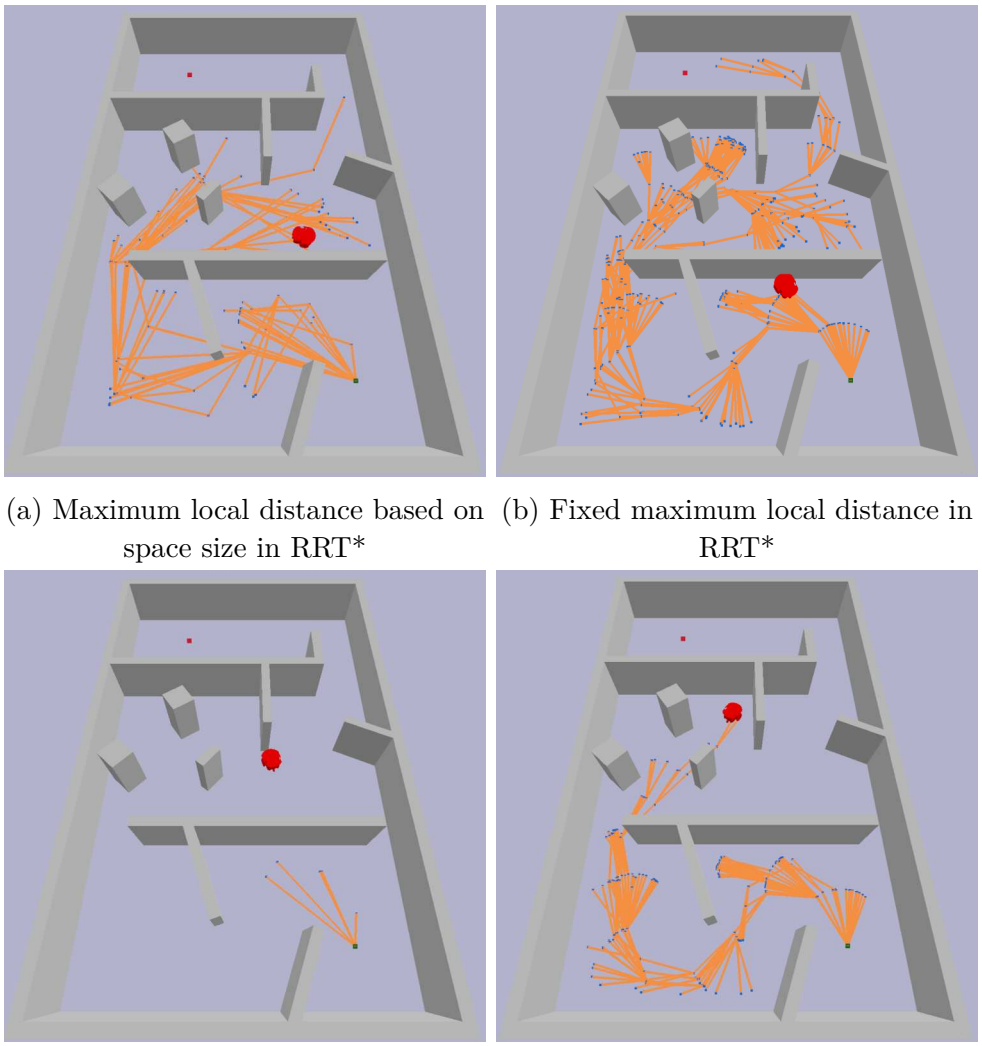
Figure 4.3: Non monotonic p-norm (a) and its integration (b) along a 70-state trajectory for a quadrotor example.

are compared with their respective non-robust vanilla versions, which focus on optimizing trajectory length. The objective is to verify the ability of the proposed variants to generate robust solutions, and to evaluate the computational overhead of solving the huge number[1] of ODEs during the random-trees expansions until an asymptotically optimal solution is found. Results are obtained from the implementations of the planners within the Open Motion Planning Library (OMPL) [ŞMK12], and collision detection is performed using the widely used C implementation of PyBullet [CB23]. The ODEs are solved using the JiTCODE [Ans18] module which converts the equations to be integrated into C-compiled code. The Euler integrator is then used to solve each ODE and take advantage of this compiled function. Although the Euler method is less accurate than a Runge-Kutta 4-based method (e.g., dopri5), it is chosen because the uncertainty tubes are already based on a first-order approximation (see Section 3.1.2), and faster computational efficiency are preferred over higher accuracy in this context.

4.1.5.1 Differential drive robot

Robot setup This subsection presents results for the differential drive robot model depicted in Section 3.2.1, with the following set of uncertain parameters $\boldsymbol{p} = [r,\, d]^T \in \mathbb{R}^2$, where d is the length between the two wheels, and r is the wheel radius. The nominal value vector is set to $\boldsymbol{p}_n = [0.1m,\, 0.4m]^T$, with an associated uncertainty range of $\delta\boldsymbol{p} = [3\%,\, 3\%]^T$, representing the percentage deviation from their corresponding nominal values. Therefore, in this differential drive robot application, $\boldsymbol{\Pi} \in \mathbb{R}^{3\times 2}$, $\boldsymbol{\Pi_\xi} \in \mathbb{R}^{3\times 2}$, and $\boldsymbol{\Theta} \in \mathbb{R}^{2\times 2}$. As a result, the computations necessary to find the uncertainty tubes involve solving 16 ODEs. The uncertainty tubes considered for this application are defined as $\boldsymbol{r_q} = [r_x,\, r_y]^T \in \mathbb{R}^2$, representing the uncertainty tubes along the x,y-axes of the state respectively, and $\boldsymbol{r_u} = [r_{u1},\, r_{u2}]^T \in \mathbb{R}^2$, representing the uncertainty tubes along the two control input space axes (ω_r and ω_l). The controller gains are set to $\boldsymbol{k}_c = [1.5,\, 8.0,\, 0.2]^T$, and the control input limits are set to $\boldsymbol{u}_{max} = [60,\, 60]^T$, corresponding to the maximum speed of the two wheels expressed in rad/s.

[1] From few thousands for non asymptotically optimal planners, up to a hundred of thousands for asymptotically optimal planners.



(a) Maximum local distance based on space size in RRT*

(b) Fixed maximum local distance in RRT*

(c) Maximum local distance based on space size in SARRT*

(d) Fixed maximum local distance in SARRT*

Figure 4.4: Snapshot after 2.000 iterations of the trees generated by: (a) the RRT* with a maximum local trajectory length based on space size, (b) RRT* with fixed maximum local trajectory length, (c) the SARRT* with a maximum local trajectory length based on space size, and (d) the SARRT* with fixed maximum local trajectory length for the differential drive robot (red shape). Note that the local trajectories (in orange) are represented as linear segments only for visualization purposes.

Planning setup Remind that the local planning method used is the Dubins curves (see Section 3.2.1.2) with $v_{magn} = 1.0$ m/s. Using the true cost-to-go from Equation (4.2) as the distance metric is known to be computationally expensive. Since this cost is closely tied to the trajectory length via the integral term, the length of the Dubins curves is employed as the distance metric to define the sample neighborhood. However, within this neighborhood, the cost from Equation (4.2) is used as the true cost-to-go.

The hyperparameters for SASST* are selected based on the recommendations in [LLB16] for a differential drive robot, with the following values: $N_0 = 10000$, $\delta_s = 0.5$,

$\delta_{BN} = 1$, and $\xi = 0.9$. For the maximum local distance, d_{max} is set to 1.0.

The standard OMPL RRT* implementation typically defines a maximum local distance based on the current state space and workspace size. This approach can lead to excessively long trajectories, especially for large spaces when two sampled states are far apart. These trajectories frequently fail the robust feasibility check, leading to significant computational effort spent on computing uncertainty tubes only to detect early infeasibility. This inefficiency, resulting from highly constrained problems, slows down tree growth. To address this, and following the approach used in the original OMPL implementation, a fixed maximum distance is introduced for local trajectories. This adjustment does not compromise the algorithm properties, as longer trajectories can still be formed by chaining several shorter local trajectories. While this approach requires more iterations to ensure adequate space coverage and cost convergence, it results in more efficient tree growth under robustness constraints, as illustrated in Figure 4.4. Note that the choice of this maximum local distance is also required for the applicability of the methods proposed in Chapters 5 and 6. Consequently, throughout this thesis, a maximum local trajectory length of 1.0 (meters or seconds according to the steering method used) is also applied to the proposed SARRT* variant.

Finally, all algorithms use a time step of 0.05s for the solving the ODEs and to perform the robust feasibility checks, and the stopping criterion of the planners is defined by a cost convergence threshold of 5%.

Results Table 4.1 presents comparative results averaged for each planner across 10 trajectories planned in the environment shown in Figure 4.6. Each of these 10 plans was

	RRT*	SARRT*	SST*	SASST*
Success (%)	3.3	**100.0**	17.3	**100.0**
Plan time (s)	952 ± 201	4145 ± 639	819 ± 155	2148 ± 320

Table 4.1: Differential drive robot application: Average planning time and success rate (no crash) of the simulated motions planned by RRT*, SARRT*, SST*, and SASST* over 10 plans and 30 simulations per plan.

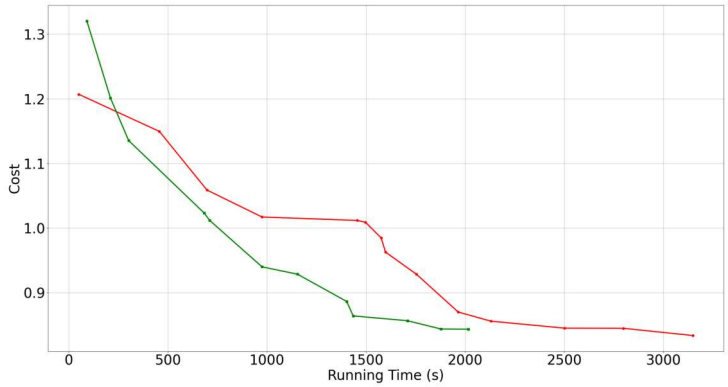


Figure 4.5: Differential drive robot application: Evolution of the sensitivity as a function of the planning time of a SASST* (green), and of SARRT* (red).

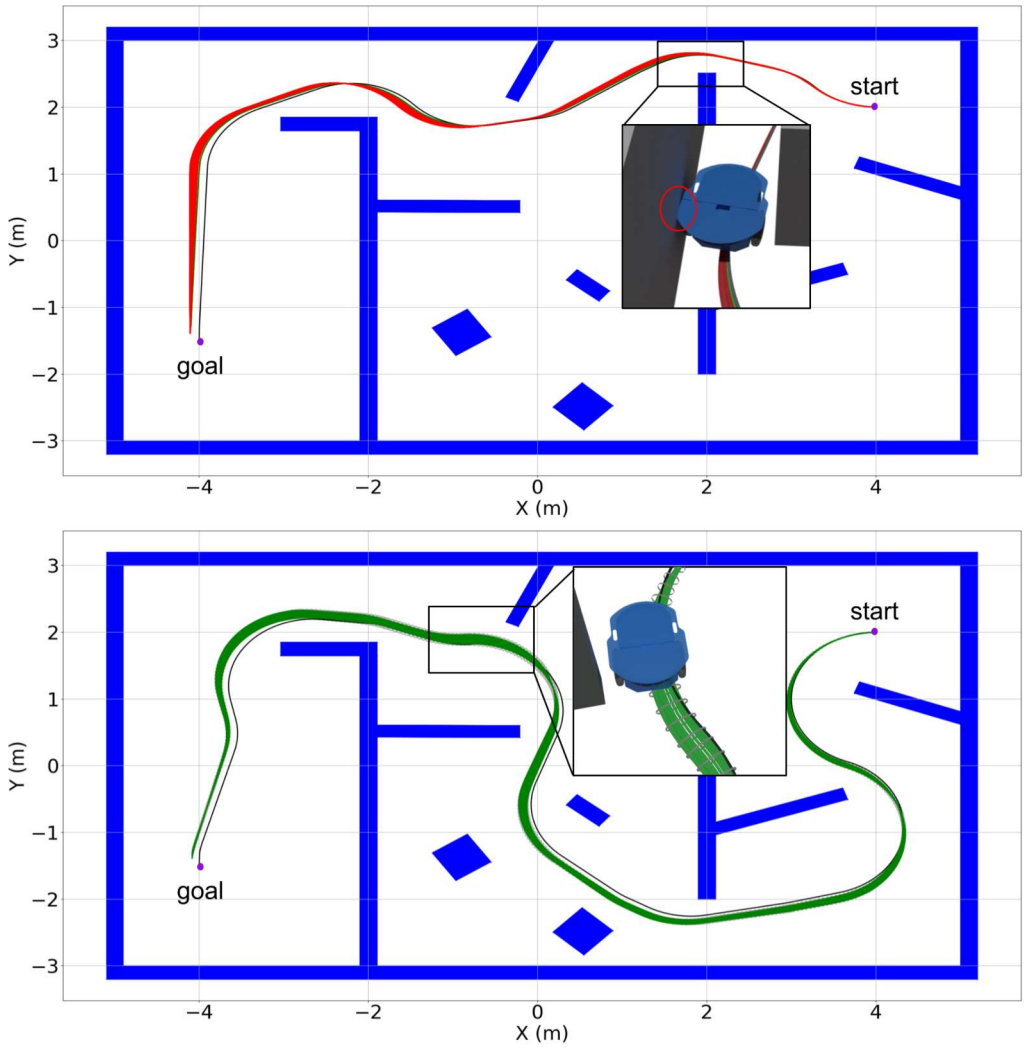


Figure 4.6: Differential drive robot application: Planned trajectory (black) produced by the RRT* (top) and the SARRT* (bottom). Simulated trajectories under uncertainty are displayed in green in the case of success, and in red in the case of a crash.

simulated 30 times with varying values for the uncertain parameters. The perturbations are uniformly sampled within an ellipsoid with semi-axis defined by $\delta \boldsymbol{p}$ as explained in Section 3.1.2. A success is defined as a successful simulation of a plan where no collisions or control input saturations occur, despite the presence of uncertainties.

First, note that SARRT*, and SASST* variants have a robustness of 100% compared to their respective standard, non-robust implementations. This robustness is also illustrated in Figure 4.6, where the vanilla RRT* is able to find the minimum cost path solution among the tree which passes through a narrow corridor, however, this solution is not robust to the presence of uncertain parameters, leading to a poor robustness when executed by the controller. On the other hand, the SARRT*, is able to find the shortest trajectory under robust constraints, leading to a 100% success rate. It is important to note that this success rate is assessed based on both robust collision avoidance and robust control inputs saturation.

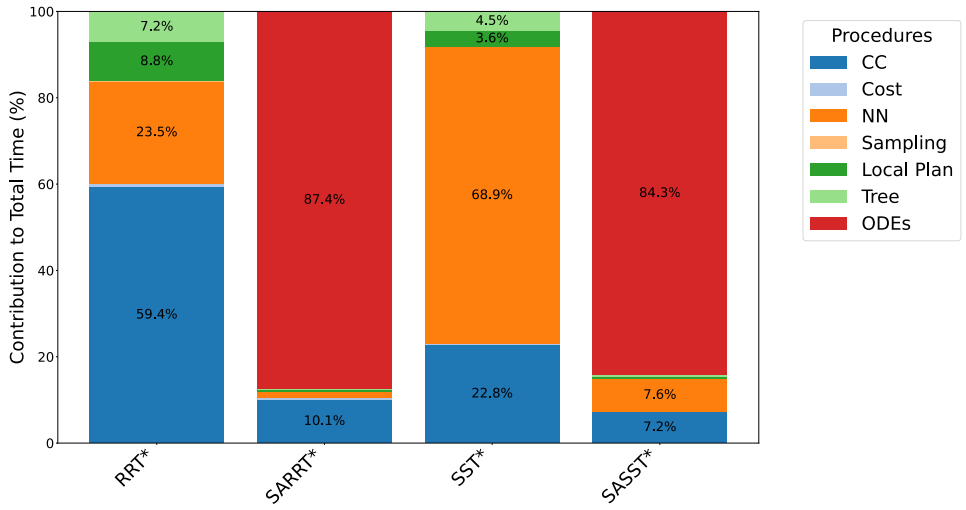


Figure 4.7: Differential drive robot application: Profiling of vanilla planners and robust SAMP variants, illustrating the contributions of various procedures to the total planning time over 1.000 iterations.

However, this robustness comes at the cost of computational efficiency, as highlighted by the significantly higher planning times of the two robust sensitivity-aware variants when compared to their standard counterparts that focus on optimizing trajectory length. The computational overhead is due to the time spent solving the ODEs, as depicted in Figure 4.7, where the *CC* procedure refers to either standard collision checking or the robust feasibility check, *Cost* represents the trajectory cost computation, *NN* stands for the nearest neighbor search, *Sampling* and *Steer* refer to state space sampling and local trajectory planning, respectively, *Tree* handles data structure management (e.g., pruning, updating child costs, etc.), and *ODEs* corresponds to the system dynamics and tubes computations. It is shown that SARRT* and SASST*, which rely on sensitivity computation, dedicate about 80% of their total computing time to solving the ODEs. This result aligns with the findings of [Tog+18], which demonstrated that, in a control-aware context, the majority of a planner computing time is taken by the closed-loop system simulation. However, in this context, the impact is even more significant due to the additional complexity introduced by the tube computation. Nevertheless, it is worth noting that the SASST* is much faster than the SARRT* variant as illustrated in Table 4.1 and in Figure 4.5 that show how the SASST* is able to converge faster than SARRT*, demonstrating that reducing the frequency of ODEs leads to a lower planning time. Furthermore, the results in Figure 4.4 strongly suggest that by incorporating uncertainty tubes, the problem becomes more constrained, requiring more iterations for the planners to converge to lower-cost regions in the state space.

4.1.5.2 Quadrotor robot

Robot setup To further evaluate the effectiveness of the SAMP variants, the quadrotor model described in Section 3.2.2 is used with the following set of uncertain parameters: $\boldsymbol{p} = [k_f, k_\tau, x_{cx}, x_{cy}]$, where k_f and k_τ are coefficients associated with the dynamics of

the propellers, and x_{cx} and x_{cy} are the location of the CoM in the x-axis and y-axis of the quadrotor body frame, which may be uncertain due to the presence of on-board sensors for example. The values of this vector considered in the simulations are either $\delta\boldsymbol{p} = [25\%, 25\%, 10cm, 10cm]$, where the first two components are a percentage of their associated nominal values. Therefore, in this quadrotor application, $\boldsymbol{\Pi} \in \mathbb{R}^{13\times4}$, and $\boldsymbol{\Theta} \in \mathbb{R}^{4\times4}$. As a result, the computations necessary to find the uncertainty tubes involve solving 68 ODEs. The uncertainty tubes considered for this application are defined as $\boldsymbol{r_q} = [r_x, r_y, r_z]^T \in \mathbb{R}^3$, representing the uncertainty tubes along the $\{x, y, z\}$-axes of the state respectively, and $\boldsymbol{r_u} = [r_{u1}, r_{u2}, r_{u3}, r_{u4}]^T \in \mathbb{R}^4$, representing the uncertainty tubes along the four control input space axes. Finally, the controller gains are set to $\boldsymbol{k}_x = [20.0, 20.0, 25.0]^T$, $\boldsymbol{k}_v = [9.0, 9.0, 12.0]^T$, $\boldsymbol{k}_R = [4.6, 4.6, 0.8]^T$, and $\boldsymbol{k}_\omega = [0.5, 0.5, 0.08]^T$, and the control inputs limits are set to $\boldsymbol{u}_{max} = [10.000, 10.000, 10.000, 10.000]^T$, corresponding to the maximum squared rotor speed expressed in $(\text{rad.s}^{-1})^2$.

Planning setup Recall that the local planning method used for the quadrotor is the kinosplines trajectory planner (see Section 3.2.2.2), with the following kinodynamic constraints enforced on the generated splines $[v_{max}, a_{max}, j_{max}, s_{max}] = [5.0\, m.s^{-1}, 1.5\, m.s^{-2}, 15.0\, m.s^{-3}, 30.0\, m.s^{-4}]$. Note that, again, instead of using the cost-to-go as distance metric, an efficient state-space quasi-metric is employed as defined in [BCS19].

The planning environment, referred to as the 2-Ways, is illustrated in Figure 4.9. In this setup, the drone is constrained to operate within a 2D plane by restricting the sampling to that plane.

The hyperparameters for SASST* are selected based on the recommendations in [LLB16] for a quadrotor, with the following values: $N_0 = 10000$, $\delta_s = 3$, $\delta_{BN} = 5$, and $\xi = 0.9$. The maximum trajectory length is set to 1.0s.

Finally, all algorithms use a time step of 0.05s for the solving the ODEs and to perform the robust feasibility checks, and the algorithms stopping criterion is defined by a cost convergence threshold of 5%.

Results Table 4.2 shows comparative results averaged for each planner over 10 plans for the RRT* and SST*, and 3 trajectories only for the SARRT* and SASST* due to very high planning time, and 30 simulations with uncertain parameters. The perturbations are uniformly sampled within an ellipsoid with semi-axis defined by $\delta\boldsymbol{p}$ as explained in Section 3.1.2. First, it is important to note robust efficiency of the proposed SAMP

	RRT*	SARRT*	SST*	SASST*
Success (%)	61.2	**100.0**	58.5	**100.0**
Plan time (s)	2801 ± 427	43451	2513 ± 378	23194

Table 4.2: Quadrotor application: Average planning time and success rate (no crash) of the simulated motions planned over 10 plans for the RRT* and SST*, and 3 trajectories only for the SARRT* and SASST* due to very high planning time. 30 simulations with uncertain parameters where conducted for each plan.

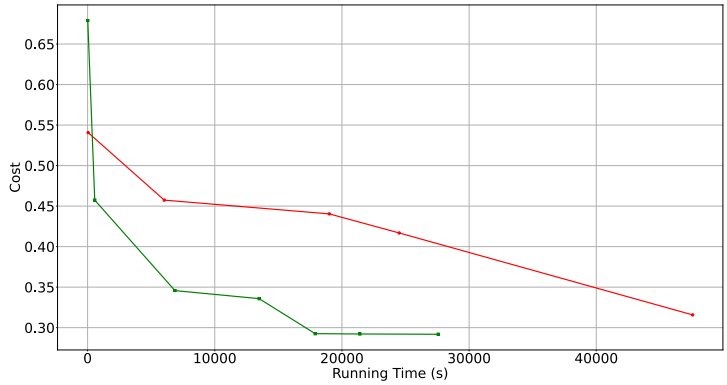


Figure 4.8: Quadrotor application: Evolution of the sensitivity as a function of the planning time of a SASST* (green), and of SARRT* (red).

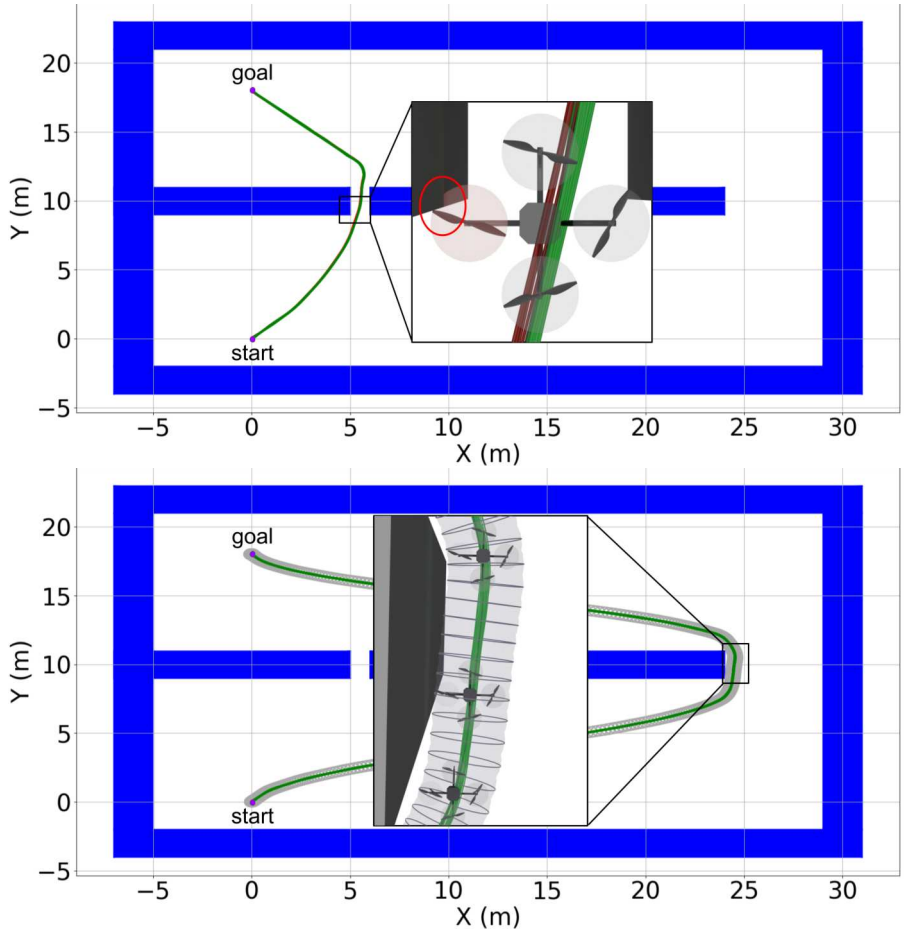


Figure 4.9: Quadrotor application: Planned trajectory (black) produced by the RRT* (top) and the SARRT* (bottom). Simulated trajectories under uncertainty are displayed in green in the case of success, and in red in the case of a crash. Note that explicitly accounting for high uncertainty, which results in large tubes during planning, does not allow the SARRT* to pass through the narrow passage. Additionally, it is important to note that the display tubes are augmented by the drone wingspan.

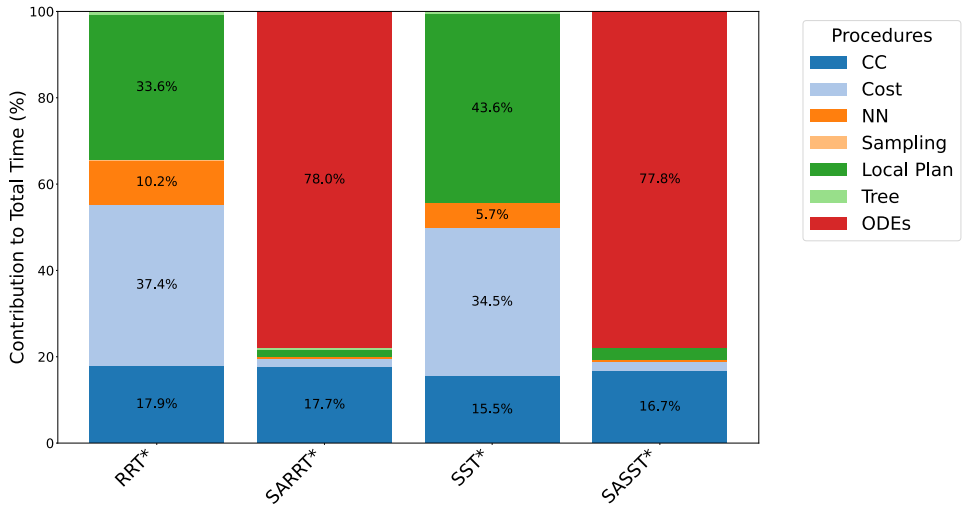


Figure 4.10: Quadrotor application: Profiling of vanilla planners and robust variants, illustrating the contributions of various procedures to the total planning time over 1.000 iterations.

variants which achieve both 100% robustness compared to their non-robust vanilla implementation, as highlighted by Figure 4.9.

However, as explained in Section 4.1, the set of ODEs must be solved at each algorithm iteration, one per iteration in the case of SASST* and at least tens to hundreds of times for the SARRT*. Although this results in high computation times for simple systems like the differential drive robot presented above, the planning time depicted in Table 4.2 and the cost convergence rate show in Figure 4.8 highlight the intractability of such optimal planning techniques for the quadrotor case, which involve solving hundreds of ODEs. Similar to the differential drive robot, the prohibitively long planning time is closely linked to the time spent solving the ODEs, as shown in Figure 4.10, where the *CC* procedure refers to standard collision checking or the robust feasibility check, *Cost* is the trajectory cost computation procedure, *NN* stands for the nearest neighbor search, *Sampling* and *Steer* refer to the state space sampling and local trajectory planning respectively, *Tree* procedure is in charge of the data structure management (e.g. pruning, child cost update, etc.), and finally *ODEs* stands for the system dynamics and tubes computations. Therefore, the frequency of ODEs solving must be further reduced, as even the SASST*, which performs only a single computation per iteration, faces challenges due to the tens of thousands of iterations required to adequately explore the space.

4.1.5.3 Conclusions

The proposed SAMP variants enable the generation of robust trajectories that account for parametric uncertainties for both differential drive and quadrotor robots, outperforming their standard non-robust counterparts in terms of robustness w.r.t. uncertain parameters. However, simulating the systems closed-loop dynamics and computing the tubes at each tree extension is computationally demanding, as highlighted by the very high planning time of the SAMP variants and mentioned in [Tog+18]. Although the SST* re-

quires more iterations to achieve convergence compared to RRT* as shown in [LLB16], its approach of computing uncertainty tubes only once per iteration results in improved planning times for SASST* compared to SARRT*. Therefore, the following section presents a decoupled approach to further reduce the frequency of simulating the closed-loop dynamics and computations of the subsequent uncertainty tubes.

4.2 Decoupled approach

As illustrated in Section 4.1.5, the proposed SAMP variants are efficient for simple systems but, they face challenges with more complex ones, such as the quadrotor. Achieving effective state-space coverage and convergence to an optimal solution w.r.t. sensitivity-based cost requires thousands of iterations, each involving the resolution of many ODEs. This creates a bottleneck for the method, particularly when solving the ODEs requires tens of milliseconds for hundreds of time steps, as common for complex systems. Therefore, this section introduces a decoupled approach that first plans a (near) time-optimal trajectory and optimizes its sensitivity in a second stage. This method minimizes the frequency of solving the ODEs, offering a more efficient algorithm for complex systems, such as quadrotor, to generate robust trajectories with near-optimal sensitivity.

Since the cost function of Equation (4.2) corresponds to the integration of the sensitivity over the trajectory, starting with an initial (near) time-optimal trajectory is a reasonable strategy for generating initial solutions of relatively high quality (i.e. near the optimal solution) w.r.t. the sensitivity cost. The proposed decoupled approach is based on a lazy sensitivity-aware variant, referred to as Lazy Sensitivity-Aware Motion Planner (LazySAMP), that first plans a robust (near) time-optimal trajectory. This trajectory is then locally optimized for minimizing the cost function (4.2) via a robust variant of the "shortcut" technique classically used to smooth solutions of randomized planners [GO07].

The following section introduces a lazy robust checking strategy. Although Section 4.1.5 demonstrates that SST* achieves faster convergence, this lazy approach cannot be applied to SST*. SST* depends on immediate collision verification and aggressive pruning to maintain a sparse and optimal tree structure. It requires prompt evaluation of node quality and trajectory costs to retain only the most promising nodes. Delaying collision checks, as in the lazy approach, would compromise the SST* ability to efficiently prune suboptimal nodes and update its witness set dynamically, potentially resulting in an inefficient and cluttered tree. Consequently, the lazy robust checking strategy outlined in the next subsection is implemented within RRT*.

4.2.1 Global robust motion planning with lazy strategy (LazySARRT*)

This subsection present the Lazy Sensitivity-Aware RRT* (LazySARRT*), as a particular instance of LazySAMP, that plans a robust (near) time-optimal trajectory. The algorithm not only provides a (near) time-optimal trajectory but also ensures that the resulting trajectory is robustly feasible w.r.t. the state uncertainties and also that control inputs remain in their allowed bounds. However, as highlighted in Section 4.1, the number of

ODEs computation within the RRT* must remain as limited as possible to avoid a too long computing time. For this reason, trajectory robustness is checked in a lazy way, only when a better solution is found, and reconnecting the nodes optimally w.r.t. the trajectory time if necessary. The lazy process is inspired from ideas in [BK00; Hau15], with the addition of maintaining a set of non-robust parents ($Q_{collide}$) for each node. Consequently, unlike the SAMP methods described in Section 4.1, the tree built by the lazy variant is not inherently robust; only the final solution is robust.

Algorithm 4.5 LazySARRT*$[q^{init}, q^{goal}]$

$T \leftarrow$ InitTree(q^{init}, q^{goal});
while not StopCondition(T, q^{goal}) **do**
  $q^{rand} \leftarrow$Sample();
  $q^{nearest} \leftarrow$ Nearest(T, q^{rand});
  Extend$(T, q^{rand}, q^{nearest})$;
  $Q_{sol} \leftarrow$ CheckForSolution(T, q^{init}, q^{goal});
  if $Q_{sol} \neq \emptyset$ **then**
    for i $= 0$ and i $<$ size(Q_{sol})-1 **do**
      $S_0 \leftarrow$GetNodeConditions(Q^i_{sol});
      $\boldsymbol{q}_d \leftarrow$Steer$(Q^i_{sol}, Q^{i+1}_{sol})$;
      $\{\boldsymbol{q}_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}, S_F\} \leftarrow$SolveODEs$(\boldsymbol{q}_d, S_0)$;
      if not IsRobust$(q_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}})$ **then**
        $T \leftarrow$RobustReconnect(T, Q^i_{sol});
return GetTrajectory(T, q^{init}, q^{goal});

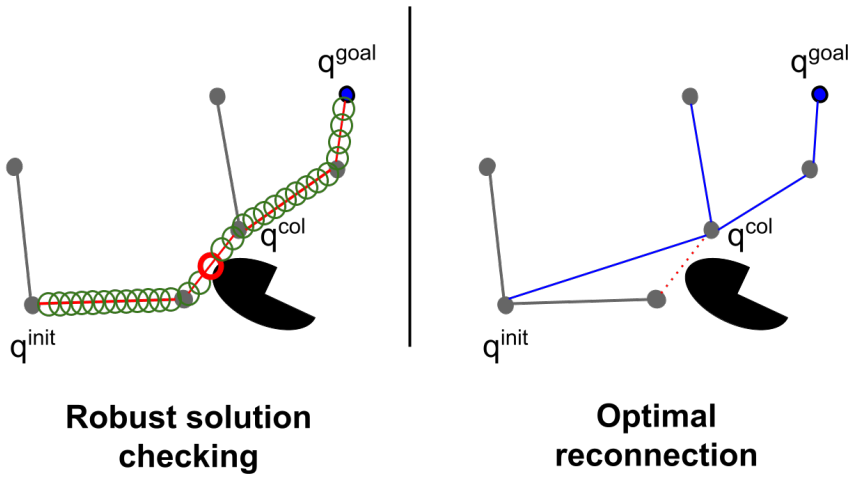


Figure 4.11: Illustration of the LazySARRT* process. When a solution is found (red), the uncertainty tubes (green) are computed and used for robust collision checking. In the case of a collision with the tubes, the tree is disconnected at the first edge found to be in collision and then optimally reconnected (indicated by blue edges) while considering previous non-robust attempts. Specifically, in this case, q^{col} cannot be reconnected to its parent during subsequent tree expansion phases.

Algorithm 4.5 provides the pseudo-code of the LazySARRT* algorithm. The first stage applies the standard RRT* (line 1-5) where the Extend procedure performs the standard collision checking (i.e. without uncertainty tubes), optimal connection, and rewiring phase for the given sample q^{rand} as in [KF11]. The algorithm then verifies at each iteration whether a new, lower-cost solution has been discovered. If a better solution is identified, the set of nodes Q_{sol}, which enables the reconstruction of this solution, is retrieved (line 6).

Next, for each local trajectories between two consecutive nodes in this set, the ODEs are solved, and the uncertainty tubes computed (line 9-11). The algorithm then verifies the robust feasibility of the local trajectory using a robustness test (see Appendix A) w.r.t. obstacles but also control inputs saturation. If a local trajectory is found to be non-robustly feasible, all nodes in the tree connected to Q^i_{sol} are disconnected and reconnected using the RobustReconnect procedure explain below (line 13).

The RobustReconnect procedure may differ depending on the specific LazySAMP variant used (see Chapter 6). In the case of LazySARRT*, where maintaining an optimal tree is required, it is illustrated by Figure 4.11 and operates as follows:

- For all tree nodes, a set of non-robust parents $Q_{collide}$ is maintained. The RobustReconnect procedure starts by adding Q^i_{sol} in the set of non-robust parents of Q^{i+1}_{sol} (i.e., Q^i_{sol} will no longer be considered as a potential parent for the node Q^{i+1}_{sol}).
- Then, all the nodes of the tree from Q^i_{sol} are disconnected and, for each of them, the vanilla Extend procedure is executed considering their associated $Q_{collide}$ set. As in the standard RRT*, an optimal re-connection phase is first attempted. All nodes must be evaluated for optimal reconnection, as the non-robust invalid edge may be reconnected, but its children nodes could potentially find improved connections.
- If no re-connection is found for a node, then the node is deleted from the tree. Otherwise, the rewiring phase is carried out considering the $Q_{collide}$ set.

The output of LazySARRT* is a (near) time-optimal trajectory that is feasible in terms of both collision avoidance and actuator saturation, accounting for uncertainties. It is however not optimized in terms of the sensitivity. Therefore, its robustness can be further improved.

4.2.2 Local robust sensitivity optimization

In order to improve the sensitivity-based cost function, a local optimization is performed using a simple robust sensitivity-aware variant of the "shortcut" smoothing algorithm [GO07], called Sensitivity-Aware Shortcut (SAShortcut) and described in Algorithm 4.6.

The algorithm is first initialized with the robust trajectory resulting from the LazySARRT* presented above (line 1). It is important to note that the robustness of the initial trajectory is a crucial assumption, as the SAShortcut algorithm does only guarantee the generation of a robust trajectory if the initial trajectory is robust. Nevertheless, the algorithm preserves this robustness throughout its local optimization process.

Algorithm 4.6 SAShortcut [$\boldsymbol{q}_{d,SARRT^*}$]

1: $\{\boldsymbol{q}_{d,best}, cost_{best}\} \leftarrow \{\boldsymbol{q}_{d,SARRT^*}, \text{Cost}(\boldsymbol{q}_{d,SARRT^*})\}$;
2: $q_d^{init} \leftarrow \boldsymbol{q}_{d,best}^0$; $q_d^{goal} \leftarrow \boldsymbol{q}_{d,best}^F$;
3: **while not** StopCondition() **do**
4: $\quad \{q_d^1, q_d^2\} \leftarrow \text{SampleOnTraj}(\boldsymbol{q}_{d,best})$;
5: $\quad \boldsymbol{q}_{d,shct} \leftarrow \text{Steer}(q_d^1, q_d^2)$;
6: $\quad$ **if** CollisionFree($\boldsymbol{q}_{d,shct}$) **then**
7: $\quad\quad \boldsymbol{q}_{d,start} \leftarrow \text{Steer}(q_d^{init}, q_d^1)$;
8: $\quad\quad \boldsymbol{q}_{d,end} \leftarrow \text{Steer}(q_d^2, q_d^{goal})$;
9: $\quad\quad \boldsymbol{q}_{d,new} \leftarrow \boldsymbol{q}_{d,start} + \boldsymbol{q}_{d,shct} + \boldsymbol{q}_{d,end}$;
10: $\quad\quad S_0 \leftarrow \text{GetNodeConditions}(q_d^{init})$;
11: $\quad\quad \{\boldsymbol{q}_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}, S_F\} \leftarrow \text{SolveODEs}(\boldsymbol{q}_{d,new}, S_0)$;
12: $\quad\quad cost_{new} \leftarrow \text{Cost}(\boldsymbol{q}_{d,new})$;
13: $\quad\quad$ **if** CostBetterThan($cost_{new}, cost_{best}$) **then**:
14: $\quad\quad\quad$ **if** IsRobust($q_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}$) **then**
15: $\quad\quad\quad\quad \{\boldsymbol{q}_{d,best}, cost_{best}\} \leftarrow \{\boldsymbol{q}_{d,new}, cost_{new}\}$;
16: **return** $\boldsymbol{q}_{d,best}$

At each iteration, the algorithm randomly samples two states $\{q_d^1, q_d^2\}$, along the current best trajectory (line 4). Next, it computes the local trajectory between the two samples $\boldsymbol{q}_{d,shct}$ (line 5). To minimize the frequency of solving the ODEs, a lazy approach is employed. First, the algorithm ensures that the proposed shortcut is collision-free (line 6). Unlike the classical shortcut method, which compares only the costs of the original and proposed trajectory segments, the entire trajectory is re-evaluated to incrementally compute the sensitivity cost (lines 7–12). It is important to note that both the cost and uncertainty tubes are derived from the ODEs. Thus, to avoid redundant computations, the ODEs are solved before performing the cost comparison. Finally, if the new solution has a lower sensitivity cost (line 13), then a robust feasibility check is performed before updating the trajectory portion in case of success (line 14-15).

4.2.3 Simulation results

4.2.3.1 Differential drive robot

Robot setup The robot setup is the same as the one described in Section 4.1.5.1.

Planning setup The environment is the same as presented in Figure 4.6. The LazySARRT* is performed with a cost convergence of 5% and the SAShortcut with a cost convergence of 1%. The decoupled approach is compared to the SASST* as it is the fastest compared to SARRT*. Finally, all algorithms use a time step of 0.05s for the solving the ODEs and to perform the robust feasibility checks.

Results The decoupled approach is tested for the same differential drive robot settings as in Section 4.1.5 where the LazySARRT* produces robust (near) length-optimal tra-

	SASST*	LazySARRT*	SAShortcut	Decoupled approach
Planning Time (s)	2148 ± 320	1233 ± 286	291 ± 106	1513 ± 253

Table 4.3: Differential drive robot application: Comparison of the average planning times between the proposed decoupled approach and the SASST*.

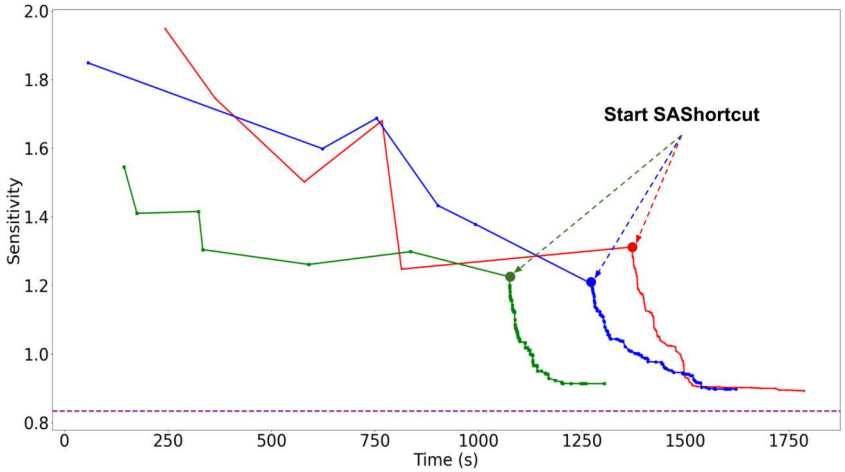


Figure 4.12: Differential drive robot application: Trajectory cost resulting of the decoupled approach over 3 runs (red, green, blue), the dashed purple cost correspond to the sensitivity-optimal reference trajectory found by the SASST*.

jectories. Table 4.3 presents the average computation times over 10 runs for SASST* and the two core procedures of the proposed decoupled approach (LazySARRT*, SAShortcut), demonstrating a time savings of only 29.56%. The average cost found by the decoupled approach is of 0.925 ± 0.032 against 0.894 ± 0.046 for the SASST*, denoting a sub-optimallity of 3.47%.

The process is illustrated by the Figure 4.12 where it is worth noting that during the LazySARRT* planning process, trajectory length optimization leads to a reduction in sensitivity cost, even though this was not the primary objective. This supports the choice of a (near) length-optimal trajectory as the initial guess for SAShortcut and validates the use of a length-based metric as the state-space metric when optimizing sensitivity within the SARRT* and SASST* methods discussed in Section 4.1.5.

4.2.3.2 Quadrotor robot

Setup The quadrotor setup is the same as described in Section 4.1.5.2. Another set of uncertain parameters is introduced, such that in the 2-Ways environment, one set allows the robot to pass through the narrow passage, while the other does not. The values of these vectors considered in the simulations are either $\delta \boldsymbol{p}_{low} = [10\%, 10\%, 5cm, 5cm]$ or $\delta \boldsymbol{p}_{high} = [25\%, 25\%, 10cm, 10cm]$, where the first two components are a percentage of their associated nominal values.

Planning The performances of the decoupled approach are evaluated in three different environments, also considering different parametric uncertainties. In the 2D environments

	U-shape	2-Way$_{low}$	2-Way$_{high}$	3D
Time SASST* (s)	7221	16615	23194	26875
Time LazySARRT* (s)	**673** ± 247	**2506** ± 305	**3271** ± 440	**2388** ± 342
Time SAShortcut (s)	**453** ± 191	**374** ± 173	**753** ± 104	**577** ± 229
Decoupled approach time gain (%)	84.4	82.7	82.6	89.0
Cost SASST*	0.317	0.292	5.127	0.510
Cost decoupled approach	**0.324** ± 0.003	**0.298** ± 0.002	**5.274** ± 0.041	**0.523** ± 0.071
Sub-optimality decoupled approach (%)	2.21	2.01	2.87	2.55

Table 4.4: Quadrotor application: Average values of costs and computing times for the different methods in several environments. Standard deviations are provided for the SAMP results only as the SST* results do not contain enough runs.

(U-shape and 2-Way), the quadrotor is forced to evolve at fix altitude by generating kino-splines only between samples within a plane and by not allowing desired displacements along the z-axis. The LazySARRT* is performed with a cost convergence of 5% (i.e., the algorithm will stop optimizing the trajectory when the improvement in cost from one iteration to the next is less than 5%) and the SAShortcut with a cost convergence of 1%. The decoupled approach is compared to the SASST* as it is the fastest compared to SARRT*. Finally, all algorithms use a time step of 0.05s for the solving the ODEs and to perform the robust feasibility checks.

Results Table 4.4 gathers the average computing times of SASST* and of the two core procedures of the proposed decoupled approach (LazySARRT*, SAShortcut), as well as the average final cost found by SASST* and SAShortcut. The mean values associated with SASST* were obtained over 3 runs (due to the very high computational time) while those associated with the proposed decoupled approach are averaged over 10 runs.

2D U-Shape Environment Figure 4.13 shows how the decoupled approach is able to produce solutions that mimic the optimal one found by the SASST* in the so called U-shape environment considering the $\delta \boldsymbol{p}_{low}$ parametric uncertainty vector. Furthermore, according to the results shown in Table 4.4 and in Figure 4.13, SAShortcut produces solutions with sensitivity costs close to the optimal reference found by SASST*. Also note the huge time saving (6.4 time faster) of the proposed method compared to the SASST*. Additionally, it is again worth to note that during the LazySARRT* planning, trajectory

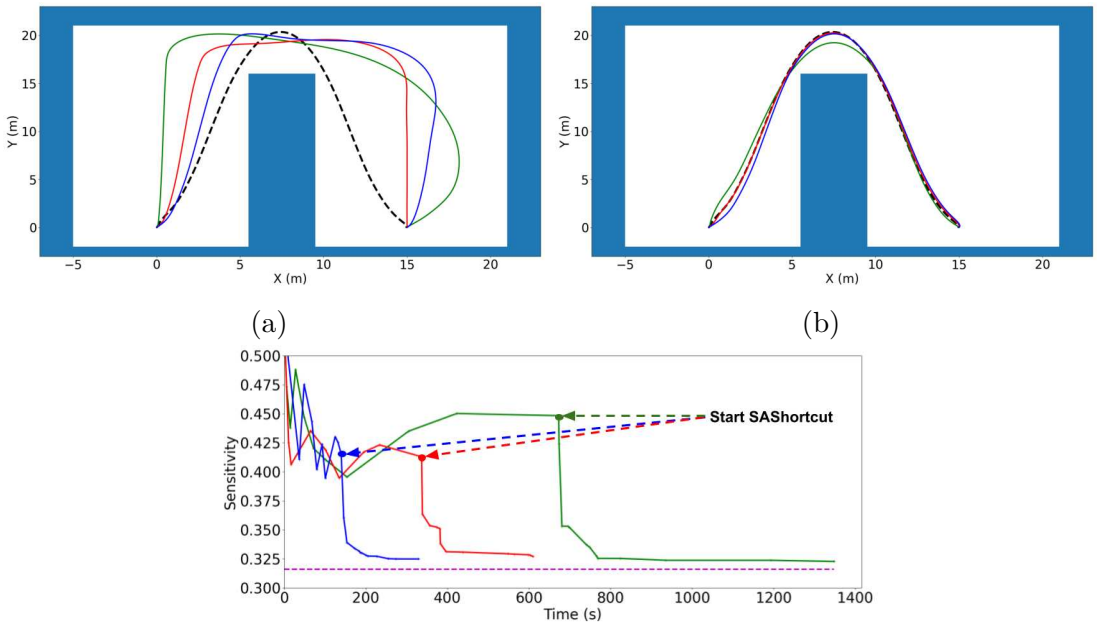


Figure 4.13: Quadrotor application: Three trajectories (red, green, blue) produced by LazySARRT* (a), and the three trajectories resulting from SAShortcut (b), with the evolution of their respective sensitivity (bottom). The dashed black trajectory (a, b) and the dashed purple cost (bottom) correspond to the SASST* sensitivity-optimal trajectory.

length optimization results in a reduction of the sensitivity cost, further corroborating the observations made in the differential drive robot application.

Finally, note that the average computing time of LazySARRT* in this case is low compared to the other environments. This is mainly due to the fact that in this case the time-optimal trajectory is far from the obstacles and therefore few re-connections via the LazySARRT* RobustReconnect procedure are performed when extending the tree.

2D 2-way Environment This environment referred to Figure 4.14 where the two sets of uncertainties $\delta \boldsymbol{p}_{low}$ and $\delta \boldsymbol{p}_{high}$ are considered.

As shown in Figure 4.14, in the presence of small uncertainties the decoupled approach produces a trajectory allowing to use the narrow passage. However, while considering large uncertainties, the non-robust time optimal trajectory, which goes through the narrow passage, cannot be taken as a collision is found by considering the uncertainty tube as shown in red in the Figure 4.14. Nevertheless, the approach is able to find the fastest trajectory apart from those passing through this passage.

In both cases the average cost of the solutions produced by the decoupled approach is close to that found by SASST* as shown in Table 4.4, where 2-Way$_{low}$ refers to the presence of small uncertainties and 2-Way$_{high}$ to the presence of large uncertainties. The average computing time of the SASST* is equivalent in both cases as it is performed on exactly the same environment. However, note that the average computing time of the LazySARRT* is longer than in the U-shape case because the time-optimal trajectories must pass through a location where many collisions may occur. Moreover, in the 2-Way$_{high}$ case the LazySARRT* computing time is longer than in the 2-Way$_{low}$ as even

more non-robust trajectories are found. This is because more iterations are needed before considering a neighborhood without nodes passing through the narrow corridor. Finally, one can note that the SAShortcut computing time remains of the same order of magnitude for both cases as that of the U-shape, and again the decoupled approach produces a solution much faster (5.7 time faster) than SASST*.

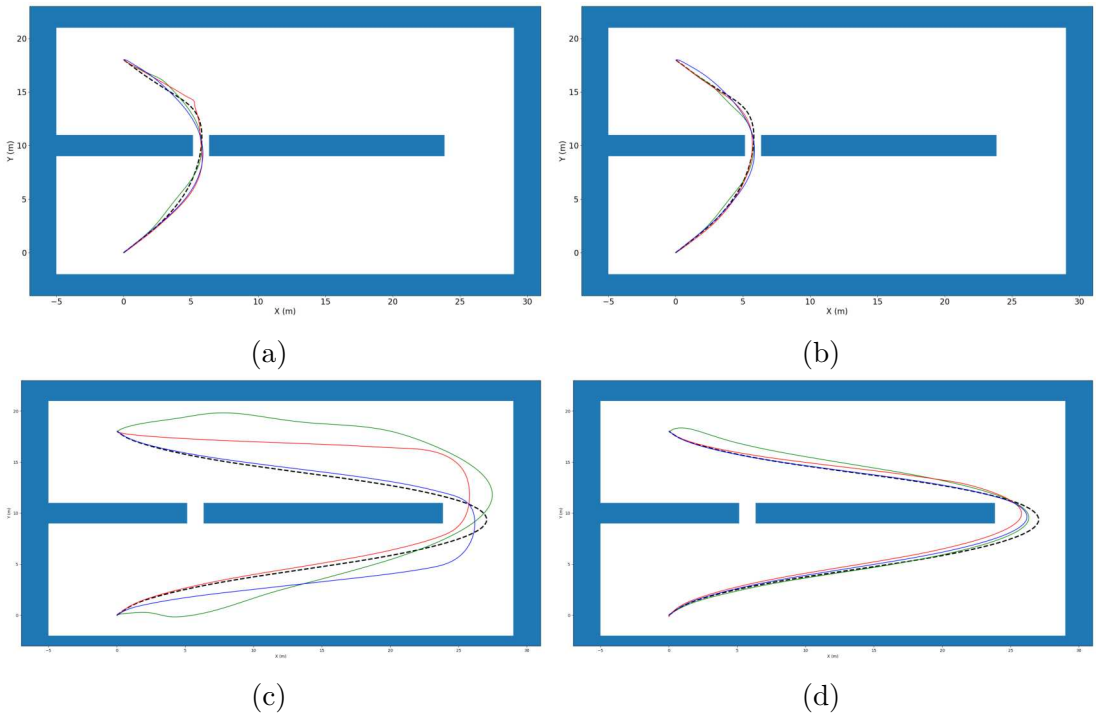


Figure 4.14: Quadrotor application: Trajectories planned by: (a) the LazySARRT* with low parameter uncertainties, (b) after the SAShortcut with low parameter uncertainties, (c) the LazySARRT* with high parameter uncertainties, and (d) after the SAShortcut with high parameter uncertainties. Note that one (near) sensitivity-optimal trajectory planned by the SASST* is displayed in dashed black.

3D Environment The uncertainty set considered in this full 3D environment is $\delta \boldsymbol{p}_{low}$. As shown in Figure 4.15, once again the decoupled approach produces trajectories that mimic the optimal obtained by the SASST*. It is interesting to note the impact of the SAShortcut procedure on the z component in addition to the impact on the $\{x, y\}$ components already seen in the previous 2D environments. The average computing time of the shortcut is similar to other environments according to Table 4.4. Again, the proposed method produces a near-optimal trajectory much faster (9 time faster) than conventional optimal planners.

Finally, it is worth noting that in all cases, the significantly higher planning time of the SASST* compared to the lazy approach indicates that robust planning in a constrained environment requires a greater number of iterations to achieve adequate state-space coverage and converge effectively. Furthermore, the significantly greater time gains observed in the quadrotor application compared to the differential drive robot application indicate that minimizing the number of times the ODEs need to be solved by the planner

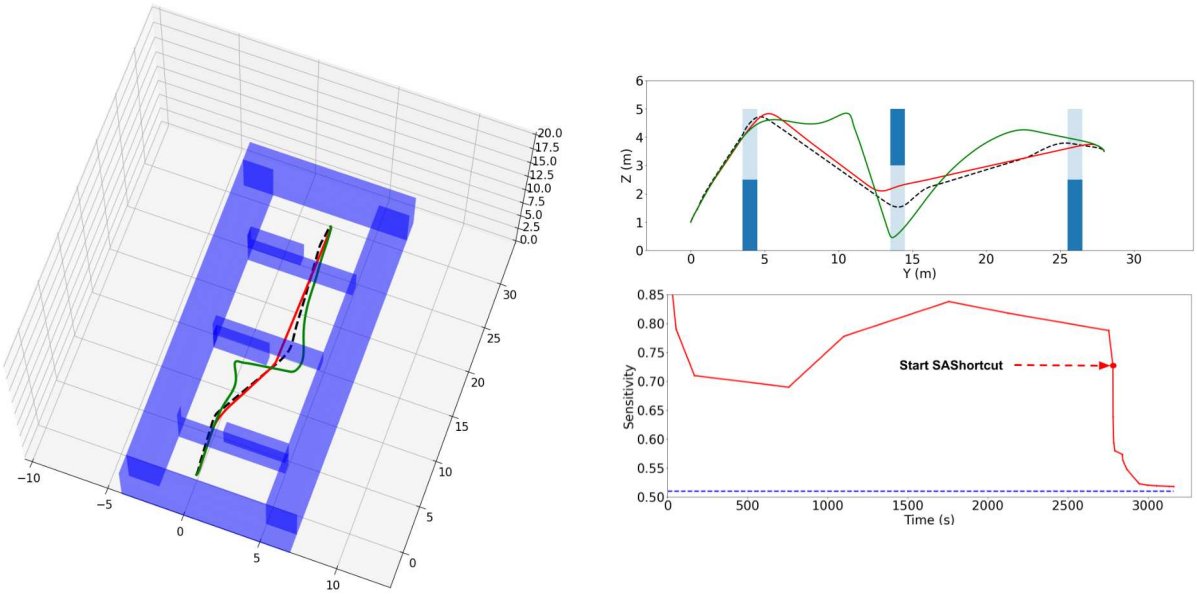


Figure 4.15: Quadrotor application: Trajectories produced by LazySARRT* (green), SAShortcut (red), and SASST* (dashed black) in a 3D environment (top). A profile view along the YZ plane is given (middle) as well as the evolution of the sensitivity (red) and the optimal one found by the SASST* (dashed blue) (bottom).

for more complex systems is crucial for keeping computation times manageable when planning with sensitivity.

4.3 Conclusions

This chapter presented the general methodology for generating robust, sensitivity-aware variants, referred to as SAMP. To address the challenge of robust sensitivity-optimal trajectory generation, two variants—SARRT* and SASST*—were introduced. While these methods were demonstrated to be effective for simpler systems, such as differential drive robots, they result in high computation times. For more complex systems, such as quadrotors, their direct application becomes computationally unmanageable. This limitation is primarily due to the significant time required to solve the ODEs.

To overcome this bottleneck, a decoupled framework was specifically designed to manage global planning using sensitivity-based metrics more efficiently than traditional asymptotically optimal sampling-based tree planners. This chapter introduced the LazySAMP variants, which aim to reduce the frequency of solving the ODEs. Simulations demonstrated that while the decoupled approach yields near sensitivity-optimal trajectories with moderate time savings for simpler systems like differential drive robots, it achieves substantial time savings as system complexity increases.

However, computing time remain affected by the need to solve the ODEs, as the associated computational cost scales approximately linearly with their number. Additionally, the decoupled approach generates trees that are not inherently robust; only the final trajectory satisfies robustness requirements, limiting the reusability of the generated plans

for tasks such as re-planning.

To address these challenges, the next chapters of this thesis will explore learning techniques to accelerate ODEs computation and propose motion planning algorithms that leverage these learned models.

Chapter 5

Learning uncertainty tubes via recurrent neural networks

This chapter introduces the second major contribution of this thesis: a deep learning approach for rapidly and accurately estimating sensitivity-based uncertainty tubes. As highlighted in Chapter 4, the computation of such tubes using ODEs integration considerably reduces the efficiency of sampling-based methods, particularly for complex systems. To overcome this limitation, this chapter introduces a method based on Recurrent Neural Networks (RNNs) to eliminate the dependency on solving ODEs. By leveraging structural similarities between ODEs and RNNs, the proposed neural network architecture achieves significantly faster predictions of uncertainty tubes compared to traditional solvers, thereby improving computational efficiency while maintaining accuracy.

This chapter is organized as follows: Section 5.1 begins with an overview of parallelization techniques and learning methods for solving ODEs. Section 5.2 details our proposed method, including the neural network architecture and dataset generation process applied to the quadrotor robot due to its greater complexity compared to the differential drive robot. Evaluation metrics, training procedures, and implementation details are discussed in Section 5.3, and Section 5.4 presents the results for the quadrotor. Finally, the method's application to the differential drive robot is also presented in Section 5.5 before drawing some conclusions in Section 5.6.

For clarity, in the following sections of this chapter, methods are referred to in non-bold, and models are referred to in bold (e.g., **GRU** refers to the proposed GRU-based architecture, and GRU refers to the Gated Recurrent Unit method).

This chapter is related to the ECAI 2024 publication [Was+24a][1].

5.1 Techniques for accelerating ODEs integration

This section begins with an overview of parallelization techniques for ODEs, such as the Parareal algorithm [Lio01], which aims to accelerate computations by distributing the workload across multiple processors. It then explores learning approaches related to ODEs, including Physics-Informed Neural Networks (PINNs) and NeuralODEs, which incorporate the structure and dynamics of differential equations into the learning process. Finally, the section introduces RNNs, emphasizing their structural similarities to ODEs and their potential for efficiently modeling temporal and sequential data in the context of dynamical systems.

[1] The results presented in this chapter slightly differ from those in the article due to improvements in implementation.

The set of ODEs that the integration has to be computationally speed-up has the following intrinsic structure within a single time step: Equation (3.2) computes the control inputs, Equation (3.1) updates the robot state, and Equation (3.4) computes the sensitivity matrices. These matrices are then leverage to derive the uncertainty tubes, as described in Equation (3.8).

5.1.1 ODEs parallelization

This subsection discusses various techniques used to parallelize the computation of ODEs, such as the Parareal algorithm [Lio01], which divides the solution process into multiple steps to be processed in parallel, improving efficiency.

- The first and most straightforward approach to parallelizing the ODEs is to compute each equation in parallel across different threads, exploiting the decoupling between the equations. However, in the current set, each equation depends on the outputs of the preceding one within a single time step, making efficient parallelization difficult since each equation must wait for the results of the previous one. Additionally, certain robot dynamics exhibit inherent coupling (e.g., for a quadrotor). These dependencies make parallelization inefficient in this context, as the computation cannot be fully decoupled across threads.

- Another well-known method for parallelizing the computation of ODEs is the Parareal algorithm [Lio01]. The approach involves dividing the time domain into smaller intervals, which can then be computed in parallel using different processors for each interval. The Parareal algorithm works by initially approximating the solution of each sub-interval using a coarse solver, followed by a more accurate fine solver that is applied in parallel. The iterative process allows the fine solver to correct discrepancies between the coarse solutions, ultimately improving the overall accuracy of the solution. However, its effectiveness can be limited by the coupling between different parts of the ODEs, as seen in the current case. This coupling limits the potential for Parareal method. Furthermore, since the current case involves control application, the first step of the algorithm, which consist of a coarse solver, might struggle to converge to a sufficiently accurate solution, making it difficult to refine it further in subsequent iterations.

While ODEs parallelization methods can be useful for speeding up large-scale simulations or off-line computations, they are not well-suited for the current set of ODEs considered in this thesis. Their application is limited because of the intrinsic coupling between different parts of the ODEs, and numerical accuracy. Furthermore, although some parts of the set of ODEs can be parallelized, the techniques will still be constrained by the limited number of CPU threads available, given the large number of equations that need to be processed in parallel.

5.1.2 Learning with embedded dynamics

This subsection explores learning techniques that directly incorporate the ODEs structure into the learning process, facilitating accurate predictions of system dynamics.

- Neural ODEs: Neural ODEs [Che+18a] offer a powerful approach to learning dynamic systems by treating ODE solvers as neural networks. This method is particularly valuable in situations where the system dynamics are not explicitly known and a flexible, data-driven approach is required to model the system behavior. In robotics, Neural ODEs have shown potential for tasks such as system identification, control, and trajectory prediction. For instance, they have been used to model and predict dynamic in cases where the exact physical models are hard to derive due to complex non-linear behavior [Che+18a]. Additionally, they have been applied to predict robust motions of robotic systems (e.g. under disturbances that are difficult to model [Naw+24]). However, in the current case, since the system dynamics is already known, the goal is to bypass the need to model the dynamic entirely. Therefore, the proposed approach in this chapter contrasts with Neural ODEs, which aims to learn the system dynamics when ODEs are not explicitly defined.

- Physics-Informed Neural Networks (PINNs): PINNs [RPK19] are a class of neural networks that integrate physical laws directly into the learning process, enabling the solution of differential equations with fewer data requirements. Unlike Neural ODEs, which learn the set of ODEs that govern the system dynamics from data, PINNs enforce the physical constraints within the network training process itself. The use of PINNs in robotics has gained significant interest, with applications ranging from developing PINN-based controllers [LBD24; Sun+22], improving robot parameter estimation [Yan+23], to solving trajectory optimization problems [DFM24]. These approaches enable more accurate and efficient modeling of robotic systems under complex constraints, facilitating better control and motion planning in real-time scenarios. However, PINNs are more of a learning process than a specific neural network architecture. They require knowledge of the system governing equations (or their differential form), which may not always be available or easily defined. Although PINNs are designed for high prediction accuracy, in the current context, where uncertainty tubes are approximations, computational resources spent on their training process may be unnecessary. However, it should be noted that while PINNs are not considered in the remainder of this chapter, they can still be leveraged to achieve better prediction accuracy if necessary.

- Some methods aim to learn the equation dynamics without relying on the previously mentioned approaches, such as the work in [FAT20], which focuses on learning the uncertainty tube dynamics. However, [FAT20] differs from the approach proposed in this chapter, as it learns the tube dynamic equations rather than directly predicting the tube values. Additionally, the referenced work operates at the time step and control level, while the proposed method works across an entire desired trajectory.

5.1.3 Learning sequences of data

This subsection discusses learning methods suited for modeling sequences of data, such as time series or trajectories, which are common while dealing with dynamic systems.

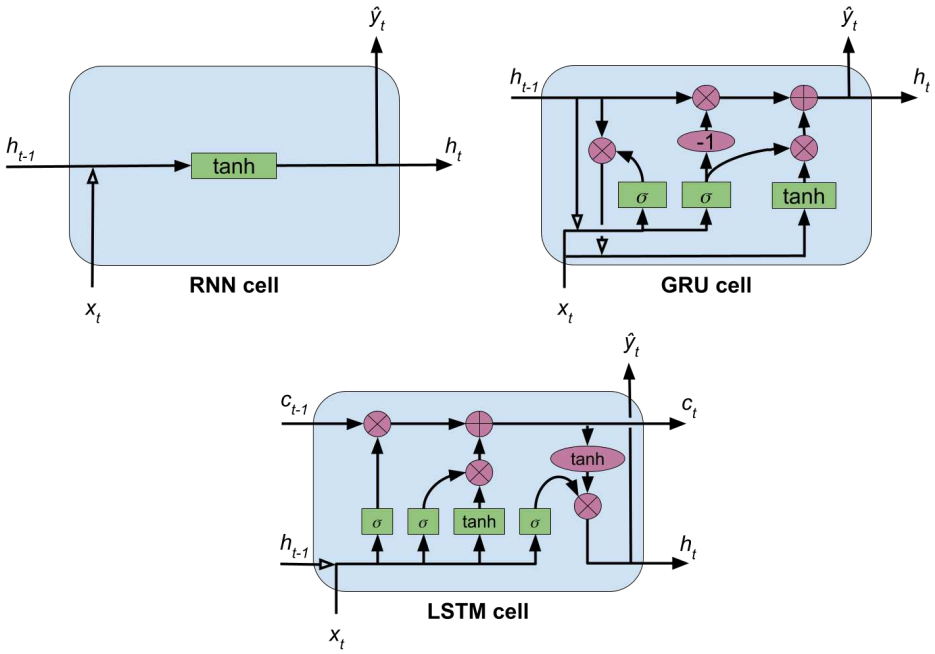


Figure 5.1: Representation of a Recurrent Neural Network (RNN) cell (top left), a Gated Recurrent Unit (GRU) cell (top right), and a Long Short-Term Memory (LSTM) cell (bottom). The three models (RNN, LSTM, and GRU) feature different update mechanisms (blue box). They each take as input a hidden state h (with LSTM also taking a cell state c) and a current state from the sequence x, and output the updated hidden state as well as the predicted value $\hat{y}$.

- Recurrent Neural Networks (RNNs): RNNs are well known for their excellent application to temporal data sequences, and more specifically in this case, to trajectories. Their application in motion planning frameworks has rapidly grown in recent years, e.g., to predict trajectories in sampling-based algorithms [Lia+21; NS20]. Furthermore, because of their time-series nature and their ability to handle dependencies between time steps, they can take advantage of the ODEs structure [Che+18b] or approximate their solutions directly [Gaj+23]. Among the most effective are Recurrent Neural Network (RNN) [RHW86], Long Short-Term Memory (LSTM) [HS97] and Gated Recurrent Unit (GRU) [Cho+14] with their respective cell depicted in Figure 5.1. The three methods differ in their gating mechanisms for updating and retaining information, but they all depend on the same *hidden state* h to propagate information across iterations. This structure is notably similar to ODEs, where the update mechanism corresponds the ODE function, and the hidden state is analogous to initial conditions, establishing a parallel between iterative updates in neural networks and dynamic systems modeling.

- Transformers: Transformers [Vas+17] have recently emerged as powerful models for sequential data processing and have shown potential in robotics. One of its current main application is its use for interpreting and explaining robot states through sequences of tokenized representations via Generative Pre-Trained Trans-

formers (GPTs)(e.g. [Vem+24]). However, unlike RNNs, Transformers do not inherently model temporal dependencies directly through recurrent connections. Instead, temporal information is incorporated using positional encodings, which may be less effective for certain tasks where explicit temporal progression is crucial. Furthermore, using Transformers starting from non-zero initial conditions is not at straightforward as in the RNNs, where this information is naturally encoded within the hidden state. Finally, Transformers poorly generalize to longer sequences than those in the training set because the self-attention mechanism processes fixed-sized segments of data and positional encodings may not adequately capture relationships in much longer contexts.

5.1.4 Outlook

This section has presented various techniques for accelerating the integration of ODEs, focusing on parallelization, learning approaches, and sequence modeling methods. Parallelization strategies, such as the straightforward approach of computing each ODE in parallel and the Parareal algorithm, were discussed. However, due to the inherent coupling between different parts of the ODEs and the constraints of control applications, these methods are not ideal for the current problem.

The section also explored learning techniques that embed ODEs dynamics directly into the model, such as Neural ODEs and Physics-Informed Neural Networks (PINNs), which are particularly useful when system dynamics are not explicitly known. While Neural ODEs offer flexibility, they are conceptually opposed to the current application, where the system dynamics are known and must be bypassed. On the other hand, Physics-Informed Neural Networks (PINNs) represent more of a learning process than a distinct architecture. They rely on accessing the gradient between the desired predicted outputs and the neural network inputs, a process that can be challenging and computationally intensive. This gradient computation often requires additional training effort, as it involves optimizing the network to capture these relationships accurately.

Additionally, learning methods like RNNs and Transformers, which are well-suited for modeling temporal data, were discussed for their potential in trajectory prediction and dynamic system modeling. However, due to the need for explicit temporal dependencies in Transformers, and because RNNs inherently encode temporal correlations and initial conditions within their hidden state, RNNs are more suitable for the current application.

Therefore, in this chapter, a RNN-based architecture will be explored to directly correlate a desired trajectory with the corresponding uncertainty tubes, eliminating the need to integrate a set of ODEs. A deeper comparison of different RNN cell types will be conducted to choose the most appropriate one for this task.

The subsequent sections introduce the proposed RNNs-based approach, initially applied to the quadrotor robot due to its greater complexity compared to the differential drive robot. However, the method's application to the differential drive robot is also discussed in the last section of this chapter.

5.2 Proposed RNNs-based approach

This section presents a multi-task learning neural network based on a GRU architecture, which takes as input the desired robot states and outputs the nominal control inputs $\boldsymbol{u}_n$, as well as the uncertainty tubes around the states $\boldsymbol{r_q}$ and the control inputs $\boldsymbol{r_u}$.

The resulting predictor, denoted as $\boldsymbol{g}$, aims at bypassing the intrinsic structure of the original set of ODEs considered in this thesis. These dependencies arise from the structure of the equations of Chapter 3: Equation (3.2) computes the control inputs, Equation (3.1) updates the robot state, and Equation (3.4) computes the sensitivity matrices before projecting them to derive the uncertainty tubes (see Equation (3.8)). Note that each system of equations depends on the outputs of the preceding one within a single time step (i.e., to solve Equation (3.4), one must first solve Equation (3.1)).

5.2.1 Problem statement

The goal is to train a neural network to approximate sensitivity-based uncertainty tubes, hence avoiding the computational cost of solving many ODEs. Moreover, the model aims at predicting the control inputs that the system will exert to successfully track a desired trajectory, in addition to the uncertainty tubes on these control inputs. Given a sequence of desired robot state vectors $\mathbf{M} = \{\boldsymbol{q}_d^0, \boldsymbol{q}_d^1, ..., \boldsymbol{q}_d^n\}$ representing the desired robot motion (i.e. a desired trajectory to follow), the task at hand is to learn a function $\boldsymbol{g}$ that estimates the radii of uncertainty tubes for each state $\boldsymbol{q}_d^k$ as well as the robot control inputs at this state such that:

$$\{\boldsymbol{r_q}^k, \boldsymbol{r_u}^k, \boldsymbol{u}^k\} = \boldsymbol{g}(\boldsymbol{q}_d^0, \boldsymbol{q}_d^1, ...\boldsymbol{q}_d^{k-1}) \tag{5.1}$$

Since evaluating the function $\boldsymbol{g}$ on $\boldsymbol{q}_d^k$ depends on all the previous states of the robot in $\mathbf{M}$, recurrent neural network architectures are a good fit given their ability to encode and accumulate temporal information while keeping inference time low.

5.2.2 Neural network architecture

A representation of the neural network architecture is presented in Figure 5.2. Blue blocks refer to the inputs of the model which are composed of an initial hidden state $\boldsymbol{h}_0$ and of a sequence of desired robot states evaluated along a desired trajectory denoted $\boldsymbol{q_{in}^k}$, where k refers to the k-th state of the desired trajectory. As the proposed model aims to bypass solving the ODEs, which starts by computing the appropriate control inputs, a reasonable initial guess is to start the prediction using the same inputs as those used by the controller. In the quadrotor case, the controller inputs consist of the desired robot position, linear velocities, accelerations, yaw angle orientation, and yaw angular velocities, as shown in Equation 3.23. However, in order to make the learned model independent of workspace boundaries used during planning (i.e. robot position) and initial robot orientations, only the desired linear velocities, accelerations, and yaw angular velocities ($\dot{\Psi}_d$), are kept as the network input components. Hence, the input to the neural network is a vector $\boldsymbol{q_{in}^k} = [\boldsymbol{v_d}, \boldsymbol{a_d}, \dot{\Psi}_d]^T \in \mathbb{R}^7$, where k refers to the k-th state of the desired trajectory.

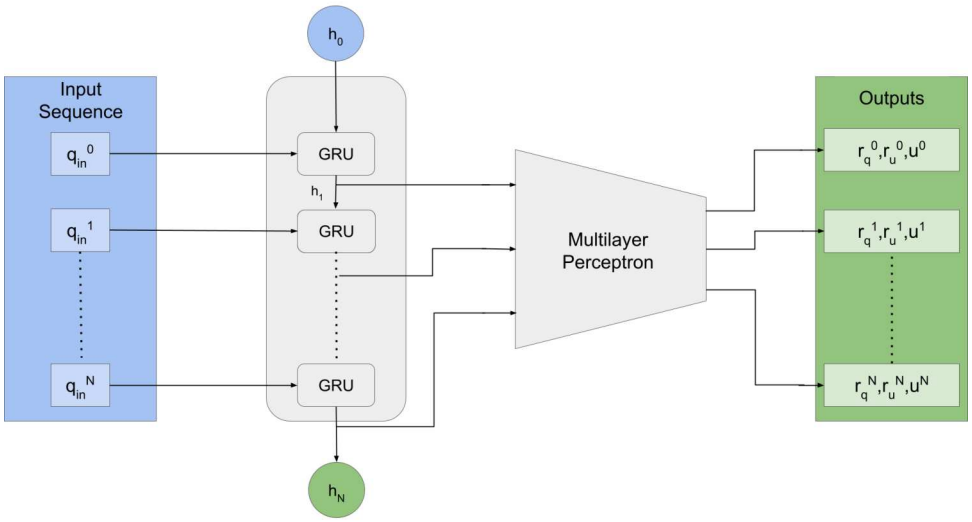


Figure 5.2: Representation of the proposed neural network architecture.

The outputs of the neural network correspond to the green blocks. In the case of a quadrotor, it is composed of the predicted uncertainty tubes radii along the $\{x, y, z\}$-axis of the state $\boldsymbol{r_q} = [r_x, r_y, r_z]^T \in \mathbb{R}^3$, and the uncertainty tubes radii associated with the control inputs of the system $\boldsymbol{r_u} = [r_{u1}, r_{u2}, r_{u3}, r_{u4}]^T \in \mathbb{R}^4$. Moreover, the control input values $\boldsymbol{u} = [u_1, u_2, u_3, u_4]^T \in \mathbb{R}^4$,which correspond to the squared propeller speeds, are required for the motion planning algorithm to perform robust feasibility checks, and their computation depends on ODEs forward integration (as shown in Section 3.1.2). Since the proposed approach aims at eliminating the need for solving ODEs, the neural network is also trained to predict the control inputs. Finally, $\boldsymbol{h}_N$ corresponds to the hidden state at the last point of our sequence (i.e., the last desired trajectory state).

At $k = 0$, the first state in the input sequence $\boldsymbol{q_{in}^0}$ and the initial hidden state $\boldsymbol{h}_0$ are given to a single-layer GRU block, which outputs an updated hidden state $\boldsymbol{h}_1$. The latter is then fed to a 3-layer multi-layer perceptron (MLP), each layer followed by a ReLU activation function except the final one, to obtain the predicted control inputs $\boldsymbol{u}^0$, the state uncertainty tubes $\boldsymbol{r_q}^0$ as well as the control uncertainty tubes $\boldsymbol{r_u}^0$. The updated hidden state $\boldsymbol{h}_1$ is then given back to the GRU block along with the second element of the input sequence. This process is repeated for each element $\boldsymbol{q_{in}^k}$ until all predictions are obtained. Details of the architecture's implementation are further presented in Section 5.3.2.

The network is intended to be used in a sampling-based tree planner (see Chapter 6), where local trajectories are concatenated to form a global one. Thus, since the hidden state encodes and accumulates temporal information about the input sequence, the final hidden state $\boldsymbol{h}_N$ of a local trajectory obtained after an iteration of a sampling-based tree planner can be saved and then given back as the initial hidden state $\boldsymbol{h}_0$ for future local trajectories considered during the next iterations. Therefore, in practice, this initial hidden state is generally not null.

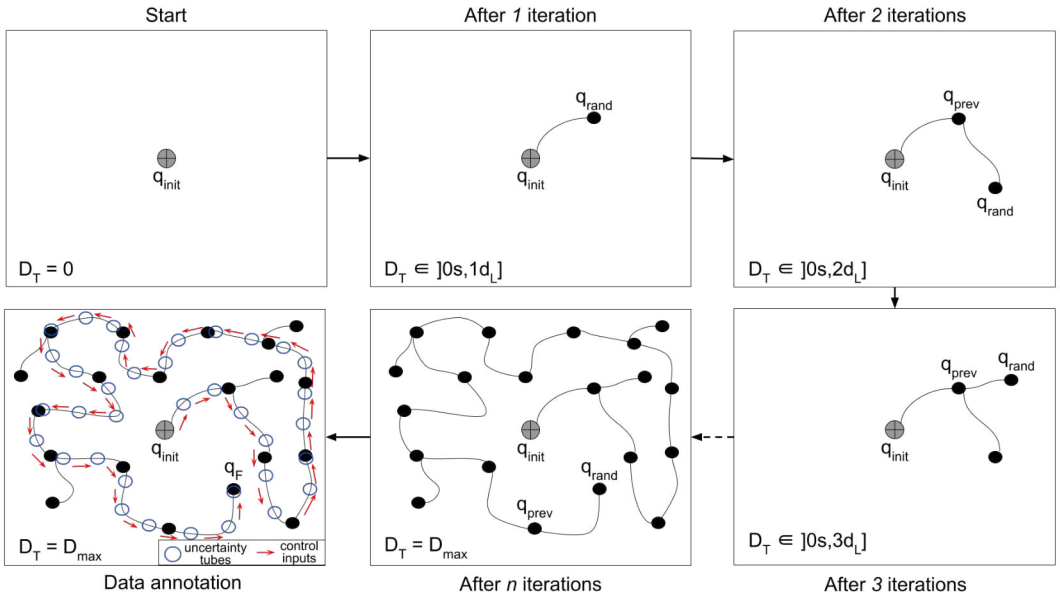


Figure 5.3: Dataset generation process. Starting from a stationary state $\boldsymbol{q}_{init}$, at each iteration a new state $\boldsymbol{q_{rand}}$ is randomly sampled and connected to the nearest one among the previous states $\boldsymbol{q_{prev}}$ until a total distance (D_T) equal to D_{max} is reached. Data annotation is performed by simulating the tracking of the generated trajectory under nominal parameters.

5.2.3 Dataset

In order to train the proposed neural network, a dataset of trajectories computed in an obstacle-free environment is generated, as depicted in Figure 5.3. This ensures that the resulting learned model is totally independent of the environment and only depends on the system. As a result, it is highly efficient when used in a sampling-based motion planning context, where the same robot model can be applied across different environments.

Each global trajectory starts from the same initial hovering state ($\boldsymbol{q}_{init}$) initialized at zero velocities and accelerations, except along the z-axis to compensate for gravity. Note that this initialization does not hurt the generalizability of the model to different initial conditions as every dynamic state can be connected to the hovering state by means of a steering method. Based on the same principle as a sampling-based planner, the global trajectory is made up of local sub-trajectories until a total distance D_{max} (according to the state space metric) is reached. Each local sub-trajectory is generated by uniformly sampling an arrival state ($\boldsymbol{q}_{rand}$) and connecting it to the nearest one among the previous sampled states ($\boldsymbol{q}_{prev}$) using an appropriate steering method, based on the robot being considered. If the local trajectory (between $\boldsymbol{q_{rand}}$ and $\boldsymbol{q_{prev}}$) distance (according to the state space metric) is greater than a maximum local distance d_l, it is truncated to d_l. This maximum local distance refers to the planner maximum local distance (see Section 4.1.5). Similarly, when the total distance exceeds D_{max} after adding a new state, it is truncate at D_{max}.

To generate the outputs, i.e. to annotate the data with uncertainty tubes and control inputs, the closed-loop dynamic and the sensitivity matrices are computed by simulating

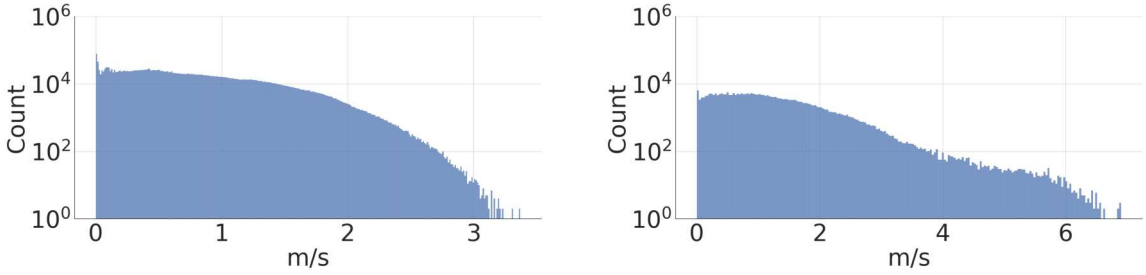


Figure 5.4: Velocity norm distribution in the training (left) and test sets (right).

Output	Validation set	Test set
$\boldsymbol{r_q}$	$1.0e^{-1} \pm 2.0e^{-2}$	$1.1e^{-1} \pm 2.1e^{-2}$
$\boldsymbol{u}$	12469.3 ± 861.6	12476.8 ± 1016.5
$\boldsymbol{r_u}$	7782.6 ± 3172.3	7828.6 ± 2845.7

Table 5.1: Quadrotor application: Mean and standard deviation of the output vector components norm after data annotation for the validation and test sets. $\boldsymbol{r_q}$ is expressed in m, and $(\boldsymbol{u}, \boldsymbol{r_u})$, are squared propeller angular velocities $[(\text{rad/s})^2]$.

the tracking of the global trajectories using an integration time step ΔT which corresponds to the controller time step. The data annotation was performed by computing and integrating the set of ODEs composed of Equation (3.1), Equation (3.2), and Equation (3.4) by mean of the Euler ODEs solver along a desired trajectory. Once $\boldsymbol{\Pi}$ and $\boldsymbol{\Theta}$ had been computed, a simple projection was performed to recover the tubes thanks to Equation (3.8). Note that the control inputs $\boldsymbol{u}$ are computed during the ODEs resolution.

Using this mechanism, a training set composed of 8.000 trajectories and a validation set composed of 2.000 trajectories were generated, making sure that every trajectory in the dataset is different. The trajectories of the dataset were generated considering a total distance D_{max} of 15s, a maximum local distance of $d_l = 1s$ and an integration time step $\Delta T = 0.05s$.

The steering method used to connect the samples during the datasets' generation process is the kinosplines method described in Section 3.2.2.2, that enforced the following kinodynamic constraints on the generated splines $[v_{max}, a_{max}, j_{max}, s_{max}] = [5.0\, m.s^{-1}, 1.5\, m.s^{-2}, 15.0\, m.s^{-3}, 30.0\, m.s^{-4}]$. The robot model used for data annotation is the one described in Section 3.2.2, with the following set of uncertain parameters $\boldsymbol{p} = [m,\ x_{cx},\ x_{cy},\ J_x,\ J_y,\ J_z]^T \in \mathbb{R}^6$, with m the quadrotor mass, $x_{cx,y}$ a shift of the system center of mass along the {x,y}-axis, and $J_{x,y,z}$ the main inertia coefficients. The nominal values of the uncertain parameters used for the robot model are $\boldsymbol{p}_n = [1.113, 0.0, 0.0, 0.015, 0.015, 0.007]^T$ and their associated uncertainty range used for the tubes computation are $\delta\boldsymbol{p} = [7\%, 3cm, 3cm, 10\%, 10\%, 10\%]^T$, which represents the variation of the parameters w.r.t. their associated nominal value. This choice of uncertain parameters is discussed in relation to the experimental validation presented in Chapter 6. The controller gains are $\boldsymbol{k}_x = [20.0, 20.0, 25.0]^T$, $\boldsymbol{k}_v = [9.0, 9.0, 12.0]^T$, $\boldsymbol{k}_R = [4.6, 4.6, 0.8]^T$, and $\boldsymbol{k}_\omega = [0.5, 0.5, 0.08]^T$.

In order to show the reliability and generalizability of the learned model, a test set composed of 1.000 trajectories was generated in the same way, but considering a maximum local duration $d_l = 2s$. As a result, trajectories with higher velocities are encountered in the test set compared to the validation set, as depicted in Figure 5.4 where velocity norms can reach up to 7 m.s^{-1} in the test set, compared with only 3 m.s^{-1} in the validation set. Note that the velocity norms exceed the velocity limit v_{max}, this is expected since this limit applies to the components of the velocity vector rather than the norm.

The mean and standard deviation values of the various components of the output vector for the validation and test sets generated by this setup are provided in Table 5.1.

5.3 Evaluation

In order to demonstrate the necessity of using a recurrent neural network due to the temporal dependencies of the predictions, a simple **MLP** baseline is implemented by substituting the recurrent layer with a single-layer linear encoder which takes as input a single element in the sequence and outputs its corresponding predictions. Also, in order to justify the choice of a **GRU** architecture, the proposed model is compared to two other versions which replace the GRU block by a basic RNN [RHW86] and an LSTM [HS97] block respectively.

Finally, in order to measure the computational cost gain achieved by the proposed method and compare the inference time of the neural network to traditional methods, two known ODEs integrators are implemented for computing uncertainty tubes. The first is **dopri5**, which leverages the Runge Kutta-4 algorithm to numerically approximate the solutions to ODEs. The second is the **Euler** method, which is known to be faster but yields less accurate results.

5.3.1 Metrics

The performance of the model is evaluated using the Mean Absolute Error (MAE) over the different outputs. Rather than comparing the results on each sub-component of the prediction (e.g $\{x, y, z\}$ for $\boldsymbol{r_q}$ in the quadrotor case), they are combined into a single metric in order to obtain simpler and more general comparisons, as follows:

- $\mathbf{MAE}_{\boldsymbol{r_q}}$: represents the mean absolute error on the norm of $\boldsymbol{r_q}$. For a given datapoint, given the ground truth and predicted state uncertainty tubes $\boldsymbol{r_q}$ and $\widehat{\mathbf{r}}_\mathbf{q}$, their respective norms $\|\boldsymbol{r_q}\|$ and $\|\widehat{\mathbf{r}}_\mathbf{q}\|$, are computed. The mean absolute error, $\mathbf{MAE}_{\boldsymbol{r_q}}$, is then defined as:

$$\mathbf{MAE}_{\boldsymbol{r_q}} = MAE(\|\boldsymbol{r_q}\|, \|\widehat{\mathbf{r}}_\mathbf{q}\|) \tag{5.2}$$

  This metric is expressed in meters (m).

- $\mathbf{MAE_u}$: represents the mean absolute error on the norm of $\boldsymbol{u}$. The norms of the ground truth and predicted control inputs, $|\boldsymbol{u}|$ and $|\hat{\boldsymbol{u}}|$, are computed. The mean absolute error, $\mathbf{MAE_u}$, is then calculated as the mean absolute error between these

two norms. It is expressed in [(rad/s)^2] for the quadrotor case, and in [(rad/s)] for the differential drive robot one.

- $\mathbf{MAE}_{\boldsymbol{r_u}}$: represents the mean absolute error on the norm of $\boldsymbol{r_u}$. As for the two previous metrics, $\mathbf{MAE}_{\boldsymbol{r_u}}$ is defined as the mean absolute error between the norms $\|\boldsymbol{r_u}\|$ and $\|\widehat{\mathbf{r}}_\mathbf{u}\|$. This metric is expressed in [(rad/s)^2] for the quadrotor case, and in [(rad/s)] for the differential drive robot one.

These metrics are averaged over all elements of a sequence, then averaged over all sequences in the validation and test sets. In addition, the inference time is also used as metric in order to evaluate the computational cost of the method compared to the defined baselines, as well as traditional uncertainty tubes computation methods involving an ODE solver. Time is denoted $T_{solver/NN}$ according to the model or solver used.

5.3.2 Training

Before training the neural network, the input features of the dataset (train, val, and test) were min-max scaled according to the maximum velocity and acceleration values used to generate the kinosplines (see Section 5.2.3) as it is guaranteed that the velocities and accelerations generated cannot exceed these values. This ensures that all velocity values are in the interval $[-v_{max}, v_{max}]$, while all the acceleration values are in $[-a_{max}, a_{max}]$. The annotations were also normalized using a standard scaling based on the mean and standard deviation values of the training set annotations.

Finally, after data normalization, the neural networks were trained for 200 epochs using the Adam optimizer [KB14] with an appropriate learning rate mentioned below and a batch size of 128. The **RNN**,**LSTM** and **GRU** models were trained with a hidden state size of 512, a **MLP** composed of 3 linear layers, and a learning rate of $1e^{-4}$ for the **RNN** case against $1e^{-3}$ for both the **LSTM** and **GRU** models. Note that an LSTM layer is composed of a hidden state and a cell state (contrarily to GRUs and RNNs which have a hidden state only), so the real LSTM latent state size is 2×512. All used hyper-parameters were validated using a grid search. They were trained using the MSE (Mean Squared Error) loss function and evaluated continuously on the validation set. The model weights achieving the best performance on the validation set were saved and used during the experiments. The resulting learning curves can be found in appendix B.

5.3.3 Implementation details

The ODEs were implemented using the JiTCODE [Ans18] module which converts the equations to be integrated into C-compiled code. The **Euler** method or the **dopri5** integrator were then used to solve each ODE and take advantage of this compiled function. Note that the **Euler** method is the one used to solve the ODEs during planning (see Chapter 4 and Chapter 6). Additionally, the **dopri5** integrator is also considered as a baseline for comparison, as it is a widely used implementation of the Runge-Kutta 4 method.

All the neural networks were implemented using the PyTorch library [Pas+19] and the lightning module. All following results were obtained on an Intel i9 CPU@2.6GHz processor with one RTX A3000 GPU.

5.4 Simulation results for a quadrotor application

5.4.1 Model comparison

This subsection focuses on identifying the most appropriate RNNs cell for the proposed architecture, selecting from RNN, GRU, and LSTM. A comparison of the models is provided for this purpose.

Table 5.2 reports the MAE for the different output vector components on the validation and test sets. First of all, note that the **MLP** demonstrates a poor performance, confirming the need for a recurrent neural network. A deviation of up to 30% from the average expected values (cf. Table 5.1) is observed in the validation and test sets.

Among recurrent models, **RNN** offers the least accurate predictions, with up to 7% error on the $\boldsymbol{r_u}$ components of the test set. On the other hand, **GRU** provides the best accuracy on all the components on both sets except for $\boldsymbol{r_u}$ on the validation set, but for which predictions remain close to expected values with less than 1% deviation from expected mean values. **GRU** shows the best reliability and generalizability of the trained model to unseen samples from a different distribution. **GRU** performance on the test set shows that the predictions along $\boldsymbol{r_q}$ remain highly accurate (less than 1 millimeter average error) and that the highest errors are obtained on $\boldsymbol{r_u}$ but do not exceed 4% of the expected average value. The **LSTM** provides the best predictions on the $\boldsymbol{r_u}$ component of the validation set. However, the results show a slightly lower accuracy than **GRU** on the other components with an average error of 4.5% on test set $\boldsymbol{r_u}$ predictions. Nevertheless, **LSTM** is more accurate on all components than **RNN**. Overall, **RNN** performs the

Method	**Validation set**		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
MLP	$8.9e^{-3}$	559.6	2079.0
RNN	$4.5e^{-4}$	35.8	166.4
LSTM	3.1^{-4}	17.7	**58.7**
GRU	$\mathbf{2.6e^{-4}}$	**17.3**	64.5
	Test set		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
MLP	$9.5e^{-3}$	608.5	2050.9
RNN	$1.4e^{-3}$	103.3	544.8
LSTM	$1.4e^{-3}$	79.7	308.1
GRU	$\mathbf{1.1e^{-3}}$	**71.0**	**279.8**

Table 5.2: Quadrotor application: MAE on the $\boldsymbol{r_q}$, $\boldsymbol{u}$ and $\boldsymbol{r_u}$ components of the output vector computed on the validation and test sets for a trained RNN, GRU, and LSTM model.

Time (ms)	100 states	200 states	300 states
T_{euler}	26.2 ± 1.2	39.1 ± 3.6	58.7 ± 4.9
T_{dopri5}	123.7 ± 17.3	255.2 ± 17.8	419.6 ± 24.5
T_{RNN}	$\mathbf{0.9} \pm 0.3$	$\mathbf{1.8} \pm 0.3$	$\mathbf{2.2} \pm 0.5$
T_{LSTM}	2.5 ± 0.3	4.8 ± 0.3	7.7 ± 0.8
T_{GRU}	2.1 ± 0.2	4.3 ± 0.3	5.8 ± 0.4

Table 5.3: Quadrotor application: Average prediction time (ms) over 100 predictions on trajectories composed of 100 states, 200 states and 300 states, for an **RNN**, **GRU**, **LSTM**, the **Euler** integrator, and the **dopri5** integrator.

worst while **GRU** and **LSTM** provide similar results, with an overall better accuracy for **GRU**. Moreover, the latter shows a better generalization to unseen trajectories that are slightly different from the training set.

In order to choose the most advantageous model or method for computing uncertainty tubes in terms of prediction time, the methods/models are applied to trajectories of different lengths (i.e. made up of a certain number of desired states). Table 5.3 shows the results obtained on trajectories of several hundred states. Note that with the current system, the number of ordinary equations solved for each element in the sequence (i.e. trajectory state) is equal to 91. The results indicate that the average prediction time for neural networks is an order of magnitude faster compared to the **Euler** method and two orders of magnitude than the **dopri5** integrator. Additionally, the computing times for all methods exhibit an approximately linear relationship with the number of states in the trajectory. Also note that among these models, **RNN** is the fastest to perform the predictions, which is explained by the fact that the network size is smaller than **LSTM**, and the **RNN** cell performs fewer internal operations than the **GRU** cell. In the context of robust sampling-based motion planning, where the neural network needs to be queried tens of thousands of times for multi-state trajectories, the observed gains in computation time can become much more significant as the number of trajectories to be evaluated increases.

Finally, even if **RNN** excels in inference time when making predictions; its accuracy is noticeably lower when contrasted with that of **GRU**. As for the **LSTM**, it provides the slowest inference time and less accurate predictions than the **GRU**, except for the $\boldsymbol{r_u}$ component of the validation set. Additionally, its hidden state implementation results in twice as many parameters having to be saved per sampling-based motion planner iteration compared to **GRU**, which translates by a higher memory cost when growing trees or graphs with thousands of nodes. Therefore, based on the model implementation details and results presented above, the use of **GRU** is recommended in the context of a sampling-based motion planner for the quadrotor application. Indeed, the latter is much faster than methods based on ODEs solvers while offering the most accurate predictions, thus offering the best trade-off between inference time and accuracy.

Method	Validation set		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
$\mathbf{GRU}_{\boldsymbol{V}}$	$\mathbf{2.5e^{-4}}$	25.2	74.6
$\mathbf{GRU}_{\boldsymbol{A}}$	$2.6e^{-4}$	17.9	77.6
$\mathbf{GRU}_{\boldsymbol{VA}}$	$2.6e^{-4}$	**17.3**	**64.5**
	Test set		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
$\mathbf{GRU}_{\boldsymbol{V}}$	$\mathbf{1.0e^{-3}}$	92.8	365.6
$\mathbf{GRU}_{\boldsymbol{A}}$	$1.3e^{-3}$	82.7	418.5
$\mathbf{GRU}_{\boldsymbol{VA}}$	$1.1e^{-3}$	**71.0**	**279.8**

Table 5.4: Quadrotor application: Results of the ablation study measured in term of MAE on the $\boldsymbol{r_q}$, $\boldsymbol{u}$ and $\boldsymbol{r_u}$ components of the output vector. V, A refer to the different inputs considered (linear and angular velocities, accelerations respectively). The presence of one as a subscript to the **GRU** model means that it is used as an input (e.g. $\boldsymbol{GRU_A}$ takes as input the acceleration only).

5.4.2 Ablation study

In theory, all the inputs to the proposed neural network are needed by the controller and the uncertainty tubes computation methods to obtain the desired outputs. In order to confirm this for the neural network as well, an ablation study is conducted on the **GRU** model to show that the choice of input vector components described in Section 5.2.2 is the most relevant for the quadrotor case.

The components considered for the study are the desired linear and angular velocities, and the desired accelerations, denoted V and A respectively. The proposed model takes both inputs and is denoted $\mathbf{GRU}_{\boldsymbol{VA}}$. Two ablations are defined, one for each input, resulting in two models: $\mathbf{GRU}_{\boldsymbol{V}}$ which takes as input the linear and angular velocities only, and $\mathbf{GRU}_{\boldsymbol{A}}$ which only takes the accelerations as input. Each model is trained in the same way detailed in Section 5.3.2 and is evaluated according to the MAE metrics described in Section 5.3.1.

Note that since the network is intended to be used in a sampling-based approach, no ablation study is performed on the output vector since our objective is to use a single model within the planner, keeping the memory and computational costs low.

Table 5.4 provides the results of the ablation study. Results show that each input component specializes in predicting one of the desired outputs. Specifically, the model $\mathbf{GRU}_{\boldsymbol{V}}$ using the velocity component only yields accurate results for $\boldsymbol{r_q}$ and $\boldsymbol{r_u}$ on both the validation and test sets, while the one solely using the accelerations leads to accurate results on $\boldsymbol{u}$ on both sets.

These outcomes are expected since $\boldsymbol{r_q}$ depends indirectly on the state of the robot $\boldsymbol{q}$ only, which can be inferred from the velocities easily. On the other hand, the inner workings of the chosen controller give more importance to the accelerations of the robot, while the velocities are used for computing internal errors. This explains the fact that $\mathbf{GRU}_{\boldsymbol{A}}$ yields good results for the $\boldsymbol{u}$ predictions.

On the other hand, Table 5.4 shows an improvement on both the control inputs $\boldsymbol{u}$

and the control uncertainty tubes $\boldsymbol{r_u}$ using both the velocities and the accelerations as inputs to the neural network. Indeed, even though a slight degradation can be noted in the performance of $\boldsymbol{r_q}$ predictions, $\mathbf{GRU}_{\boldsymbol{VA}}$ achieves the best overall results on both the validation and test sets, thus justifying the implementation of Section 5.2.2. This can be explained by the fact that the output values are not related to the input components independently, they also depend on the interactions between these variables. Also, remind that the chosen controller and uncertainty tubes computation methods use both the velocities and the accelerations of the robot to obtain the desired outputs.

Note that the results of this ablation study are specific to the chosen case of the quadrotor/controller. Another dynamical system might need a different set of inputs to obtain accurate predictions.

5.4.3 Qualitative results

The robust generalization of the **GRU** model to the test set is highlighted in Figure 5.6, depicting predictions for all the components of the output vector for a trajectory composed of 300 states (i.e. $T_F = 15s$). One can observe larger prediction errors during transient phases involving the higher velocities present in the test set than in the training one. Note that the neural network predictions in these phases have a moving average "behavior". However, an overall good fitting of the predictions can be highlighted.

In addition to the results discussed in Section 5.4.1, Figure 5.7 shows an example of **MLP**'s inability to provide accurate predictions on a 300-state test set trajectory, with a tendency to simply average expected values over the latter. In comparison, the **GRU**

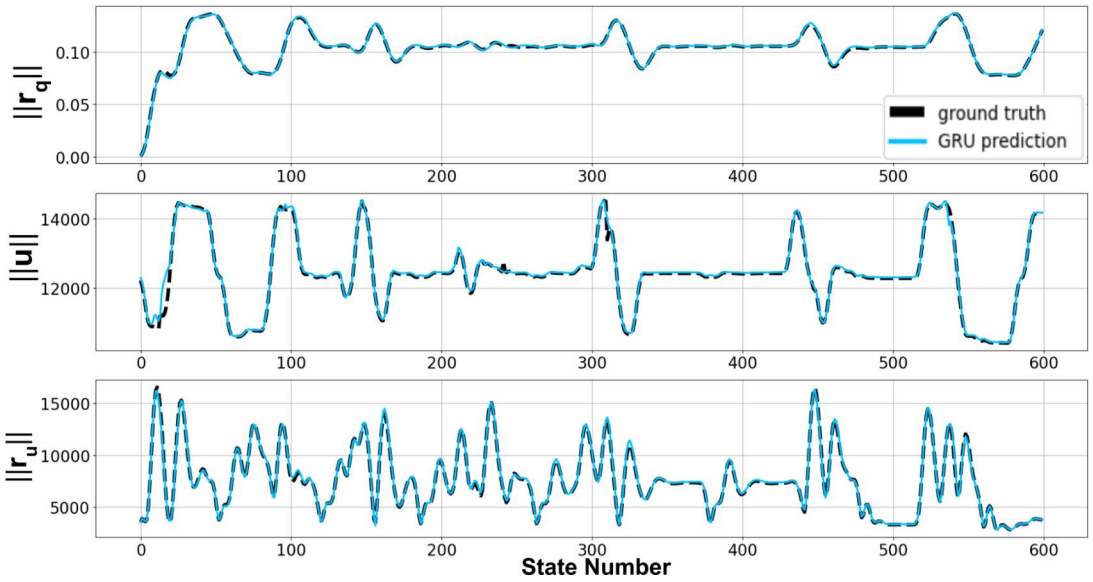


Figure 5.5: Quadrotor application: Example of **GRU** predictions on a 600-state trajectory generated in the same way as the test set. $||\boldsymbol{r_q}||, ||\boldsymbol{u}||$ and $||\boldsymbol{r_u}||$ refer to the norm of their respective vector. Predicted outputs are displayed in blue against true values in black. $||\boldsymbol{r_q}||$ is expressed in m, and control input associated values ($||\boldsymbol{u}||, ||\boldsymbol{r_u}||$) are squared propeller angular velocities [(rad/s)^2].

model is able to accurately predict the control inputs and the uncertainty tube across the whole trajectory. Also note a close performance between **GRU** and **LSTM** on all tasks, while **RNN** lags behind and tends to underestimate large variations of control uncertainty tubes $\boldsymbol{r_u}$.

In order to show the stability of the proposed method, Figure 5.5 depicts the norm of predicted vectors for a 600-state trajectory generated in the same way as the test set but considering a total length of 30s (i.e. $T_F = 30s$). Note that this trajectory is twice as long as those used in training, validation and test sets. The results show the same behavior in transient phases where higher velocities are encountered. However, one can observe that the model is consistent throughout longer sequences even with a different distribution from the one it was trained one.

Finally, Figure 5.8 highlights the ability of the proposed model to provide predictions with good accuracy when starting from non-zero hidden state. This is convenient in a sampling-based motion planning context, where successive local trajectories must be evaluated. Note that, in the dataset all trajectories start from a hovering state (i.e., null velocities and accelerations). Hence, in order to generalize to trajectories where initial velocities and accelerations are not null one can simply append a planned sub-trajectory arriving at this initial state and starting from the hovering state.

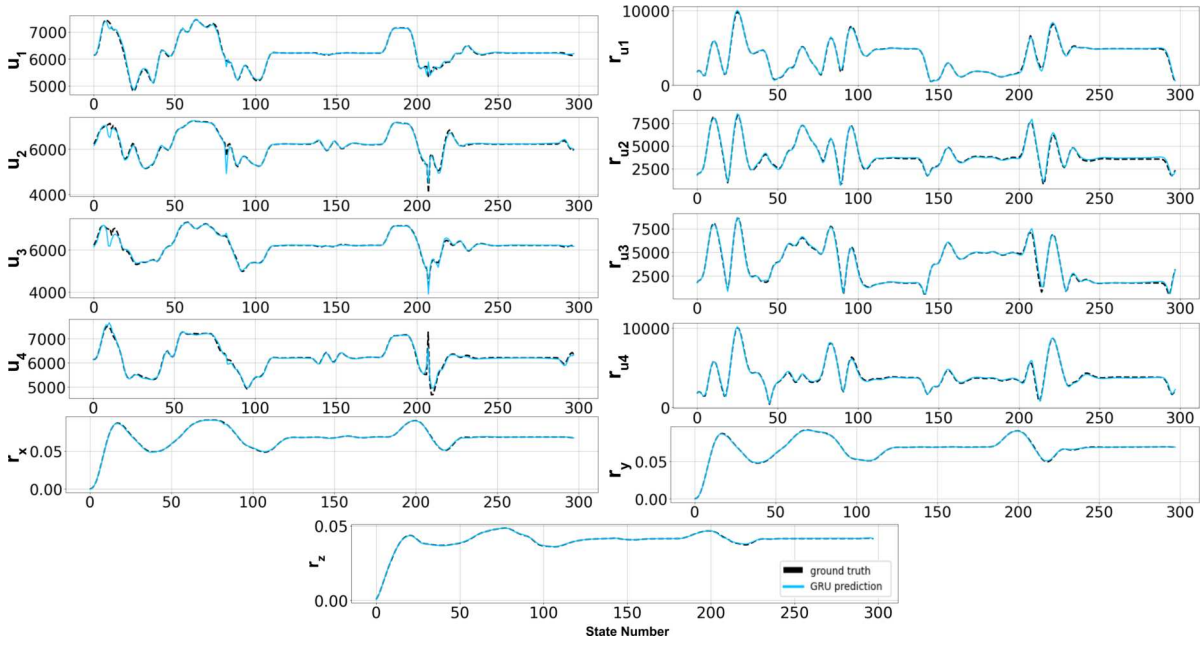


Figure 5.6: Quadrotor application: Example of **GRU** predictions on a 300-state trajectory of the test set. Predicted outputs are displayed in blue against true values in black. r_x, r_y, r_z are expressed in m, and control input associated values (u_i, r_{ui}) are squared propeller angular velocities $[(\text{rad/s})^2]$.

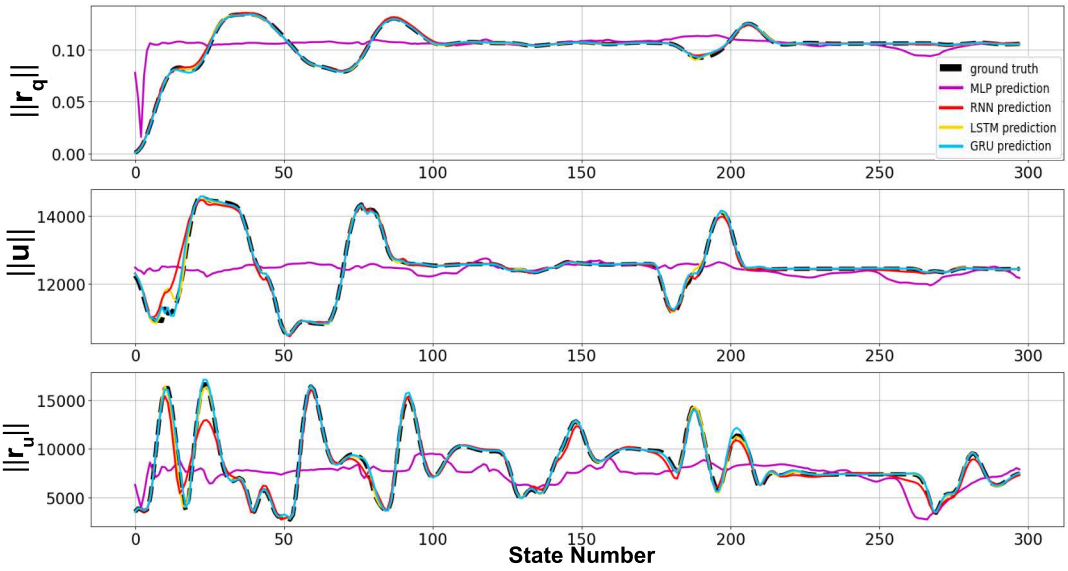


Figure 5.7: Quadrotor application: Comparison of predictions obtained using the different recurrent neural network architectures as well as the **MLP** on a 300-state trajectory of the test set. $||\boldsymbol{r_q}||$, $||\boldsymbol{u}||$ and $||\boldsymbol{r_u}||$ refer to the norm of their respective vector. True values are displayed in black, **GRU** predictions in blue, **MLP** predictions in purple, **LSTM** outputs in yellow, and basic **RNN** predictions are in red.

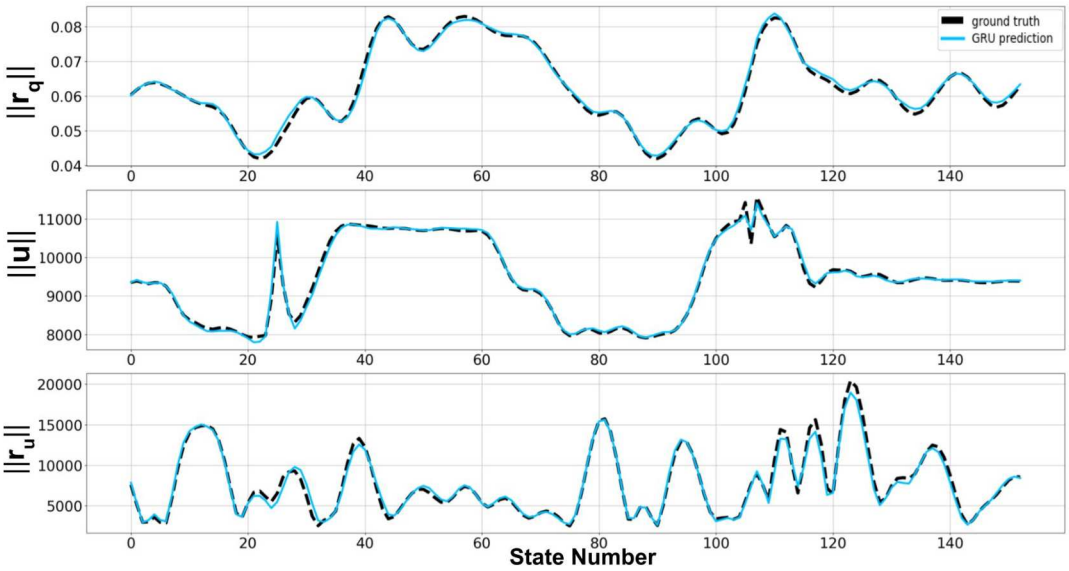


Figure 5.8: Quadrotor application: Example of **GRU** predictions on a 150-state trajectory of the test set starting from a non-zero hidden state. $||\boldsymbol{r_q}||$, $||\boldsymbol{u}||$ and $||\boldsymbol{r_u}||$ refer to the norm of their respective vector. True values are displayed in black and **GRU** predictions in blue.

5.5 Application to the differential drive robot

This section illustrates the versatility of the proposed method by extending its application to other systems, specifically to the differential drive robot discussed in Section 3.2.1.

5.5.1 Setup adaptation

First, this subsection explains how the proposed architecture, the dataset generation, and training process are adapted to the differential drive robot.

Architecture As mentioned in Section 5.2.2, a reasonable initial guess is to start the prediction using the same inputs as those used by the controller. The DFL controller inputs consist of the desired robot position, linear velocities, and accelerations, as shown in Equation 3.12. However, to ensure that the learned model is independent of the workspace boundaries used during planning (i.e., the robot's position limits) and since accelerations are not considered for this robot in this thesis, only the desired linear velocities are retained as input components for the network. Therefore, the input to the neural network for the differential drive robot case is a vector $\boldsymbol{q}_{\boldsymbol{in}}^{\boldsymbol{k}} = [\boldsymbol{v_d}]^T \in \mathbb{R}^2$, where k refers to the k-th state along the discretized desired trajectory.

The predicted outputs are composed of the predicted uncertainty tubes radii along the $\{x, y\}$-axis of the state $\boldsymbol{r_q} = [r_x,\, r_y]^T \in \mathbb{R}^2$, and the uncertainty tubes radii associated with the control inputs of the system $\boldsymbol{r_u} = [r_{u1},\, r_{u2}]^T \in \mathbb{R}^2$. The control input values predicted by the neural network are $\boldsymbol{u} = [u_1,\, u_2]^T \in \mathbb{R}^2$, which correspond to the right and left wheel speeds.

Dataset The dataset is generated using the same method described in Section 5.2.3, using the Dubins steering method (see Section 3.2.1.2) to connect the sampled states together. The trajectories in the dataset were generated with a maximum total distance D_{max} of 15m, a maximum local distance of $d_l = 1m$, a velocity magnitude of $v_{magn} = 1m/s$, and an integration time step $\Delta T = 0.05s$.

The robot model used for data annotation is the one described in Section 3.2.1, with the following set of uncertain parameters $\boldsymbol{p} = [r,\, d]^T \in \mathbb{R}^2$. The nominal values of the uncertain parameters used for the robot model are $\boldsymbol{p}_n = [0.1m,\, 0.4m]^T$, with an associated uncertainty range of $\delta\boldsymbol{p} = [3\%,\, 3\%]^T$, representing the percentage deviation from their corresponding nominal values. The controller gains are $\boldsymbol{k}_c = [1.5,\, 8.0,\, 0.2]^T$.

To demonstrate the reliability and generalizability of the learned model, a test set of 1.000 trajectories was also generated, but with a velocity magnitude of $v_{magn} = 1.1m/s$ for the Dubins curve parametrization. Consequently, the test set includes trajectories with higher velocities than those in the validation set.

Table 5.5 presents the mean and standard deviation values of the various components of the output vector for both the validation and test sets generated by this setup. These values indicate a slight increase in the norms of all components due to the higher velocities.

Training Before training the neural network, the input features of the dataset (train, val, and test) were min-max scaled according to the velocity magnitude value used when

Output	Validation set	Test set
$\boldsymbol{r_q}$	$5.8e^{-2} \pm 2.2e^{-2}$	$6.1e^{-2} \pm 2.3e^{-2}$
$\boldsymbol{u}$	15.1 ± 1.3	15.7 ± 1.4
$\boldsymbol{r_u}$	$2.0e^{-1} \pm 7.8e^{-2}$	$2.0e^{-1} \pm 8.1e^{-2}$

Table 5.5: Differential drive robot application: Mean and standard deviation of the output vector components norm after data annotation for the validation and test sets. $\boldsymbol{r_q}$ is expressed in m, and $(\boldsymbol{u}, \boldsymbol{r_u})$, are wheel angular velocities [(rad/s)].

generating the dataset (see Section 5.2.3) as it is guaranteed that the velocities components generated cannot exceed this value, ensuring that all velocity values are in the interval $[-v_{magn}, v_{magn}]$. The annotations were also normalized using a standard scaling based on the mean and standard deviation values of the training set annotations.

Following data normalization, the neural networks were trained for 100 epochs using the Adam optimizer [KB14]. The **RNN**, **LSTM**, and **GRU** models were configured with a hidden state size of 128 and 3 linear layers. All hyperparameters were validated through a grid search.

The models were trained using the Mean Squared Error (MSE) loss function and continuously evaluated on the validation set. The model weights that achieved the best performance on the validation set were saved and used for the experiments. The resulting learning curves are provided in Appendix B.

5.5.2 Simulation results

5.5.2.1 Model comparison

Table 5.6 reports the MAE for the different output vector components on the validation and test sets. Firstly, one can note how the simple **MLP** model provides poor prediction

Method	Validation set		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
MLP	$1.8e^{-2}$	$8.1e^{-1}$	$6.2e^{-2}$
RNN	$3.8e^{-3}$	$1.1e^{-1}$	$1.3e^{-2}$
LSTM	$9.5e^{-4}$	$7.2e^{-2}$	$4.2e^{-3}$
GRU	$\mathbf{6.9e^{-4}}$	$\mathbf{4.4e^{-2}}$	$\mathbf{3.3e^{-3}}$
	Test set		
	$\mathbf{MAE}_{\boldsymbol{r_q}}$ (m)	$\mathbf{MAE}_{\boldsymbol{u}}$ $[(rad/s)^2]$	$\mathbf{MAE}_{\boldsymbol{r_u}}$ $[(rad/s)^2]$
MLP	$1.9e^{-2}$	1.4	$6.5e^{-2}$
RNN	$5.9e^{-3}$	$6.1e^{-1}$	$2.3e^{-2}$
LSTM	$4.3e^{-3}$	$5.6e^{-1}$	$\mathbf{9.4e^{-3}}$
GRU	$\mathbf{3.3e^{-3}}$	$\mathbf{5.0e^{-1}}$	$1.4e^{-2}$

Table 5.6: Differential drive robot application: MAE on the $\boldsymbol{r_q}$, $\boldsymbol{u}$ and $\boldsymbol{r_u}$ components of the output vector computed on the validation and test sets for a trained RNN, GRU, and LSTM model.

Time (ms)	100 states	200 states	300 states
T_{euler}	6.1 ± 0.1	12.0 ± 0.1	19.3 ± 0.5
T_{dopri5}	8.5 ± 0.4	16.8 ± 0.6	23.9 ± 0.6
T_{RNN}	$\mathbf{3.2e^{-1}} \pm 4.9e^{-1}$	$\mathbf{3.7e^{-1}} \pm 4.7e^{-1}$	$\mathbf{4.7e^{-1}} \pm 5.2e^{-1}$
T_{LSTM}	$4.1e^{-1} \pm 4.6e^{-1}$	$6.9e^{-1} \pm 5.1e^{-1}$	$9.2e^{-1} \pm 4.9e^{-1}$
T_{GRU}	$3.9e^{-1} \pm 6.4e^{-1}$	$4.4e^{-1} \pm 5.5e^{-1}$	$5.6e^{-1} \pm 5.9e^{-1}$

Table 5.7: Differential drive robot application: Average prediction time (ms) over 100 predictions on trajectories composed of 100 states, 200 states and 300 states, for an **RNN**, **GRU**, **LSTM**, the **Euler** integrator, and the **dopri5** integrator.

accuracy across all components in both the validation and test sets. This suggests that the **MLP** struggles to capture the strong temporal dependencies between inputs and outputs. Consequently, the **MLP** shows the highest deviation from the ground truth values, as reported in Table 5.1, with deviations reaching up to 30% for the $\boldsymbol{r_q}$ component in the validation set, and 32% for the $\boldsymbol{r_u}$ component in the test set.

As for recurrent networks, the **RNN** demonstrates the poorest performance, with errors reaching up to 7% on the validation set and 11% on the $\boldsymbol{r_u}$ components of the test set. The **GRU** and **LSTM** models exhibit comparable accuracy overall. However, the **GRU** slightly outperforms on the validation set, achieving only a 1% deviation on both the $\boldsymbol{r_q}$ and $\boldsymbol{r_u}$ components, compared to the **LSTM**, which shows a 2% deviation on $\boldsymbol{r_u}$. While the **LSTM** generalizes better on $\boldsymbol{r_u}$ according to the results over the test set, the **GRU** achieves superior generalization on $\boldsymbol{r_q}$ and $\boldsymbol{u}$, making it the most robust model overall. It is worth noting that the deviations observed on the test set are a bit high, reaching up to 6% for the **GRU**. This indicates a slight overfitting of the model to the validation set.

Nevertheless, in a sampling-based context, the model accuracy alone is not sufficient as a performance metric. As discussed in Section 5.3.1, it is also essential to compare the inference time of the various models with the time required to integrate the ODEs using standard methods. Table 5.7 shows the results obtained on trajectories of several hundred states. Note that with the current system, the number of ordinary equations solved for each element in the sequence (i.e. trajectory state) is equal to 16. The results indicate that the average prediction gain time for neural networks is approximately of an order of magnitude compared to the **Euler** and **dopri5** methods. Additionally, note that even if the inference time appears to scale linearly with the number of time steps to be integrated, the time savings become increasingly significant with the number of states. Specifically, the **GRU** achieves a speedup of 15 times compared to **Euler** for 100 states, and this advantage grows to 35 times for 300 states. This result suggests that the intrinsic structure of the original ODEs, as discussed in Section 5.2, has been reformulated into a simplified representation.

In conclusion, the **GRU** once again proves to be the most suitable model for sampling-based applications, as demonstrated in the quadrotor case, offering an optimal balance between accuracy and inference time. Although the **RNN** is the fastest, it has the lowest accuracy among the three recurrent architectures. Meanwhile, the **LSTM** achieves

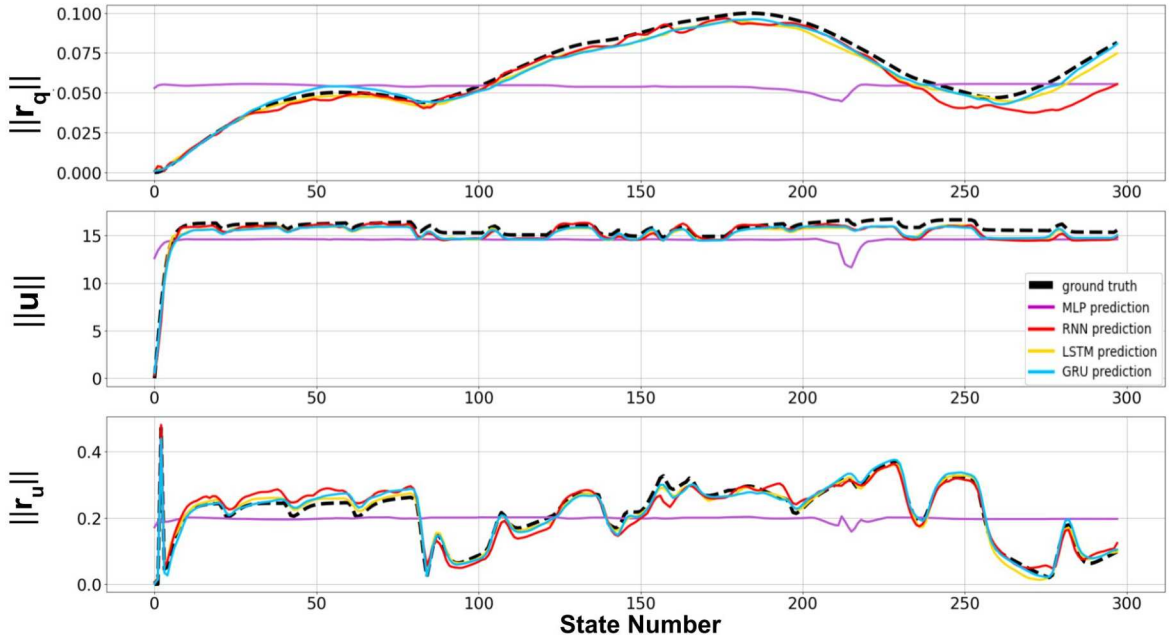


Figure 5.9: Differential drive robot application: Comparison of predictions obtained using the different recurrent neural network architectures as well as the **MLP** on a 300-state trajectory of the test set. $||\boldsymbol{r_q}||$, $||\boldsymbol{u}||$ and $||\boldsymbol{r_u}||$ refer to the norm of their respective vector. True values are displayed in black, **GRU** predictions in blue, **MLP** predictions in purple, **LSTM** outputs in yellow, and basic **RNN** predictions are in red.

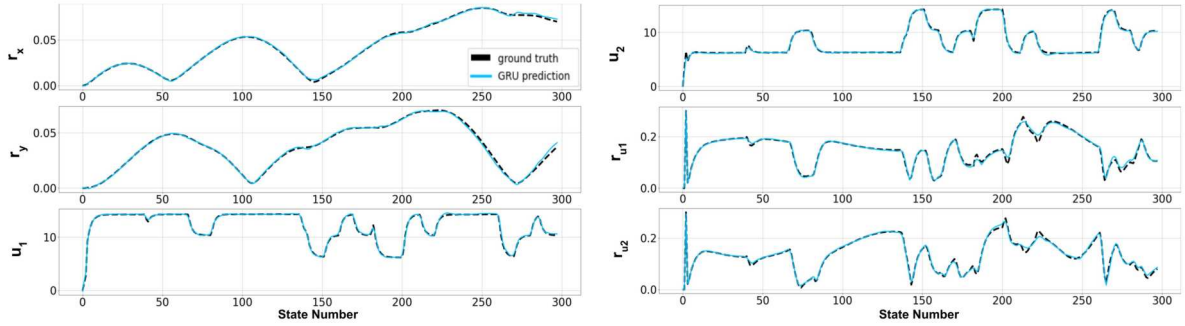


Figure 5.10: Differential drive robot application: Example of **GRU** predictions on a 300-state trajectory of the validation set. Predicted outputs are displayed in blue against true values in black. r_x, r_y are expressed in m, and control input associated values (u_i, r_{ui}) are wheel angular velocities [(rad/s)].

accuracy comparable to the **GRU** but lags behind in inference speed.

5.5.2.2 Qualitative results

This subsection presents qualitative results for differential drive robot. Firstly, Figure 5.9 illustrates the predictions of the **RNN**, **GRU**, **LSTM**, and **MLP** models on a 300-state trajectory from the test set. It is noticeable that the **MLP** model provides poor accuracy, as discussed in Section 5.5.2.1. However, one can note its averaging behavior.

Additionally, the **GRU** and **LSTM** models perform similarly across all tasks, while the **RNN** shows lower performance and tends to underestimate the variations. However, a slight resulting offset can be observed on the **GRU** and **LSTM** predictions, denoting of the slight overfitting as mentioned in Section 5.5.2.1 due to higher speed encountered in the test set.

However, this slight overfitting is not an issue for the proposed model application, since in the sampling-based context the trajectories encountered are more likely to match those of the validation set rather than those of the test set. Figure 5.10 shows the predictions for all output components independently, for a 300-state trajectory from the validation set.

This section has demonstrated that the proposed RNNs-based method and architecture presented in Section 5.2 can be easily extended to the differential drive robot, showing a significant improvement in tube computation inference time. Furthermore, the **GRU** once again emerged as the most suitable model for sampling-based applications.

5.6 Conclusion

This chapter has presented a **GRU**-based neural network architecture to predict uncertainty tubes and control inputs along sequences of desired states for any dynamical system. Results on a quadrotor and differential drive robot use cases show that leveraging recurrent neural network architectures is of key importance due to the temporal dependency of the predictions. Furthermore, it has been shown that a **GRU** is more appropriate in a sampling-based tree planner context than **RNN** or **LSTM** as it proposes the best compromise between prediction accuracy, generalizability, inference time and memory cost. By directly correlating the inputs and outputs of the neural network, the proposed predictor $\boldsymbol{g}$ eliminates the need for sequential computation inherent in the ODEs addressed in this thesis. This design simplifies the process and reduces the computational burden, making it more efficient than the traditional ODEs framework while maintaining a good accuracy.

Additionally, great care was taken to develop a learning process that ensures the learned models are independent of the environment, enabling their application to motion planning problems. However, although the proposed **GRU** is independent of the robot's environment, it is still dependent on the system's nominal parameters and controller gains.

The remainder of this thesis will focus on incorporating the GRU-based architecture into sampling-based motion planners to plan robust robot motions against parametric uncertainties in a more efficient way than what was proposed in Chapter 4.

Chapter 6

Robust motion planning with accuracy optimization via learned sensitivity metrics

This contribution presents an enhanced version of the sensitivity-aware variants from Chapter 4, integrating the GRU-based architecture from Chapter 5 to reduce the computational cost of solving thousands of ODEs in sampling-based contexts. Moreover, methods from Chapter 4 and the ones in the literature focus on computing robust trajectories but do not consider the problem of *also* minimizing uncertainty tubes at a desired location for increasing the accuracy of specific tasks (e.g., insertion). Therefore, in this chapter, a novel sensitivity-based cost function is introduced, differing from the one in Chapter 4. Instead of minimizing sensitivity along the entire trajectory, it focuses on minimizing the radius of uncertainty tubes at desired states for the task at hand. Additionally, unlike Chapter 4, where trajectory states were the sole optimization variables, this chapter also considers controller gains as optimization variables during the planning process. With respect to these considerations, the contribution of this chapter is twofold:

1. A computationally efficient version of the SAMP variants (see Chapter 4) is proposed, relying on a Gated Recurrent Unit neural network (GRU)[Cho+14] (see Chapter 5), which quickly and accurately estimates time-varying uncertainty tubes and control input profiles along trajectories. A general method for incorporating this network into sampling-based tree planners is presented to predict uncertainty tubes and control inputs along trajectories.

2. Based on this new planner variants, a new framework is proposed that plans robust trajectories and locally optimizes them, along with the controller gains, to maximize accuracy at desired states of the planned trajectory for any system/controller.

This chapter is structured as follows: Section 6.1 discusses the integration of the GRU-based architecture into sampling-based planners and outlines the accuracy optimization process. Then, Section 6.2 demonstrates the approach efficiency through simulation results. Finally, Section 6.3 validates these contributions experimentally through two challenging scenarios (Figure 6.9) involving a quadrotor model with uncertain parameters: (i) robust navigation through a narrow window, and (ii) an in-flight ring-catching task. These real world applications showcase the framework's robustness and accuracy under parameter uncertainties.

This chapter is related to the RA-L 2024 publication [Was+24b][1].

[1]The results presented in this chapter slightly differ from those in the article due to improvements in implementation.

6.1 Robust and accurate planning framework

This section presents the main algorithm used to plan robust trajectories, which are then locally optimized for accuracy at specified desired states. The method consists of two stages:

1. First, it generates a robust trajectory based on a Deep learning-based Sensitivity-Aware Motion Planner (DeepSAMP) – explained in Section 6.1.1 – that utilizes the GRU-based computation of the uncertainty tubes.

2. Second, it optimizes the accuracy at some given states along this trajectory by minimizing the size of the uncertainty tubes at these locations.

The motivation behind the second stage is to improve the accuracy of the planned robust trajectory for tasks – e.g., pick-and-place or insertion tasks – where minimizing the deviation from the nominal trajectory is important only at specific designed locations, as for picking the ring in Figure 6.9.

The workflow of the method is presented in Algorithm 6.1[1]. It takes as input (line 1) a list of desired states $list_d = (q_d^0, \dots, q_d^n)$ for which the accuracy should be optimized, and the initial controller gains vector k_c^{init} considered constant all along the trajectory. The role of the controller gains are further discussed in Section 6.2.4.

Algorithm 6.1 Robust and Accuracy Optimization $[list_d, k_c^{init}]$

1: $\boldsymbol{q}_d^{tot} \leftarrow \emptyset$;
2: **for** $(i = 1;\ i < len(list_d);\ i = i + 1)$ **do**
3: $\quad \boldsymbol{q}_d^{tot} \leftarrow \boldsymbol{q}_d^{tot}$+DeepSAMP$(list_d(i-1), list_d(i))$;
4: $\{\boldsymbol{q}_d^{tot}, k_c^{opt}\} \leftarrow$A-Optim$(list_d, \boldsymbol{q}_d^{tot}, k_c^{init})$;
5: **return** $\{\boldsymbol{q}_d^{tot}, k_c^{opt}\}$;

The first step of the algorithm consists in generating robust trajectories between successive desired states in the list (line 3 of Algorithm 6.1) by means of a deep learning-based sensitivity-aware motion planner variant called DeepSAMP (explained in Section 6.1.1) that uses the learning approach presented in Chapter 5. These trajectories are concatenated into a global one $\boldsymbol{q}_d^{tot}$, connecting all the desired states of $list_d$ (lines 2-3 in Algorithm 6.1). It is important to note that, unlike the variants discussed in Chapter 4, this step may not incorporate any aspect of optimality, as sensitivity is primarily leveraged to enforce robust constraints.

The trajectory from DeepSAMP is then locally modified by A-Optim (line 4 in Algorithm 6.1), aiming at optimizing the accuracy at specific desired states along the trajectory. This algorithm iteratively samples both the trajectory from DeepSAMP and the controller gains, adjusting the former to minimize uncertainty at these states. Indeed, as demonstrated in [SFR23], optimizing both factors concurrently results in minimizing the

[1]For clarity in the pseudo-code, the temporal notation is omitted in this chapter. Thus, bold symbols refer to trajectory vectors (e.g., $\boldsymbol{q}_d$ represents a desired trajectory, $\boldsymbol{r_q}$ stands for the uncertainty tube radii along a trajectory, etc.), while non-bold symbols represent values at a single state (e.g., q^{rand} denotes a random state, and S_0 represents the initial conditions of the ODEs at a given trajectory state, etc.).

uncertainty. The algorithm produces two offline outputs: (i) a robust desired trajectory $\boldsymbol{q}_d^{tot}$ optimized for accuracy at the desired states, and (ii) the optimized controller gains vector k_c^{opt}, considered constant throughout the trajectory.

6.1.1 Robust motion planning via deep learning

6.1.1.1 Sampling-based motion planning with RNNs

This paragraph explains how the deep learning method presented in Chapter 5 can be incorporated into any sampling-based tree planner in order to create Deep learning-based Sensitivity-Aware Motion Planner (DeepSAMP) variants, which use the learned model to predict uncertainty tubes. As highlighted before, a key challenge in computing such tubes for a given trajectory lies in the high computational cost of numerically integrating the dynamics of $\boldsymbol{\Pi}(t)$ and $\boldsymbol{\Theta}(t)$. Additionally, when extending the tree and computing these sensitivity matrices, various initial conditions (such as the initial robot state, $\boldsymbol{\Pi}_0$, etc.) need to be incorporated into the tree nodes (as outlined in Chapter 4).

This problem is addressed thanks to the GRU network, which naturally encodes this information in its "memory" terms, i.e., the so-called hidden state (see [Cho+14]). An interesting feature of the DeepSAMP variants is to leverage this latent state to embed the initial conditions into each node analogous to the ODEs initial conditions mentioned in Chapter 4. This enables the reuse of the updated initial conditions for predictions in future extensions. Note that a hidden state h is unique according to its parent, analogous to the initial conditions of ODEs discussed in Chapter 4. Therefore, its use is only applicable to tree-based planners, where each node has a single parent. In the remainder of this chapter, it is demonstrated how to integrate this mechanism into the RRT algorithm and its optimal variant, RRT*.

6.1.1.2 Deep Sensitivity-Aware RRT (DeepSARRT)

This subsection proposes a deep learning-based sensitivity-aware variant of the RRT [LaV98], denoted Deep learning-based Sensitivity-Aware RRT (DeepSARRT) as a particular instance of DeepSAMP, and presented in Algorithm 6.2. It demonstrates how to integrate the GRU hidden state and tube predictions within a standard RRT planner [LaV98]. It is important to note that this approach, using GRU hidden state and tube predictions, can be similarly applied to other tree-based planners. For instance, in the results presented in Section 6.2.3, a deep learning-based sensitivity-aware variant of the RRT* [KF11] implementation, denoted Deep learning-based Sensitivity-Aware RRT* (DeepSARRT), leverage the same principle presented here. Additionally, note that the standard non-optimal RRT is used here, as sensitivity is only employed to enforce robust constraints rather than to find a sensitivity-optimal trajectory, in contrast to the objective of Chapter 4.

First, DeepSARRT performs the standard RRT procedure (lines 1-5) that produces a local desired trajectory $\boldsymbol{q}_d$ between a sampled state (q^{rand}) and its nearest state (q^{near}) in the tree. Then, the starting hidden state (as for h_0 of the GRU) is recovered from the tree node (line 6). Such initial condition is used together with this desired trajectory by the

GRU that returns all the radii and the control inputs profiles ($\boldsymbol{r_q}, \boldsymbol{r_u}, \boldsymbol{u}$), together with the final hidden state h_F to be reused as initial condition in subsequent iterations (line 7). Next, an initial filter is applied using the predicted control inputs and uncertainty tubes, with a robust control input check for saturation (line 8). The nominal trajectory is then computed (line 9). At this stage, the control inputs are recomputed in a closed-loop manner w.r.t. the desired trajectory $\boldsymbol{q}_d$, even though these control inputs were initially predicted by the GRU. Indeed, directly using the predicted control inputs in an open-loop manner could lead to unsuitable nominal trajectories due to the accumulation of prediction errors over time in the simulation process. Then, for each state of the nominal trajectory $\boldsymbol{q}_n$, the function *IsStateRobust* (line 10) performs a robust collision checking by using the uncertainty radii (for further details see Appendix A). If the extension is valid, the final state of the desired trajectory is inserted in the tree as a new node, embedding at the same time the final hidden state h_F to be reused as initial condition in next iterations (line 10-12). Finally, the algorithm returns a global trajectory connecting q^{init} and q^{goal} if one exists in the tree (lines 15).

Note that, unlike the SAMP variants discussed in Chapter 4, which perform the robust feasibility check in a single step, the operation is decoupled here. This approach first leverages the predicted control inputs and associated tubes to apply an initial filtering stage, as this step is less computationally expensive than the subsequent nominal trajectory simulation and robust collision checking. This decoupling is made possible by the GRU, whereas in SAMP variants, both the control inputs and associated tubes are derived from the full resolution of the ODEs.

Algorithm 6.2 DeepSARRT $[q^{init}, q^{goal}]$

```
1: T ← InitTree(q^init, q^goal);
2: while not StopCondition(T, q^goal) do
3:     q^rand ←Sample();
4:     q^near ← Nearest(T, q^rand);
5:     q_d ← Steer(q^near, q^rand);
6:     h_0 ←GetNodeConditions(q^near);
7:     {r_q, r_u, u, h_F} ←GRU(q_d, h_0);
8:     if IsControlRobust(r_u, u) then
9:         q_n ←SimulateExecution(q_d);
10:        if IsStateRobust(r_q, q_n) then
11:            SetNodeConditions(q^rand, h_F);
12:            AddNewNode(T, q^rand);
13:            AddNewEdge(T, q^near, q^rand);
14: return GetTrajectory(T, q^init, q^goal);
```

6.1.2 Robust local accuracy optimization

This subsection introduces robust local optimizer variants, highlighting their capability to address accuracy-related costs effectively.

6.1.2.1 Cost function

The application of a local optimization method at this level is justified by the cost function considered in order to optimize the accuracy at desired states. Indeed, the cost of a desired trajectory $\boldsymbol{q}_d$ is defined as:

$$c(\boldsymbol{q}_d) = w_1\mathbb{E}[L] + w_2\mathbb{V}[L], \quad L = [\lambda_0 ... \lambda_n] \tag{6.1}$$

with $\mathbb{E}[L]$ and $\mathbb{V}[L]$ the mean and the variance of L, where λ_k is the p-norm of the radii of interest in the k-th state in the $list_d$ of Section 6.1, and w_1, w_2 are user-defined weights. The variance is considered in this cost function so that the minimization of a radius at a given point does not lead to the growth of another radius at another waypoint. As the cost function of Chapter 4, this function is neither additive (i.e., considering two trajectories $(\boldsymbol{q}_{d,1}(t), \boldsymbol{q}_{d,2}(t))$, the cost of their concatenation $c(\boldsymbol{q}_{d,1}(t)|\boldsymbol{q}_{d,2}(t)) \neq c(\boldsymbol{q}_{d,1}(t)) + c(\boldsymbol{q}_{d,2}(t))$), nor monotonic. Therefore, it is unsuitable for global optimization using sampling-based motion planners like [KF11; LaV98], since they require additive and monotonic objective functions. Given that the analytical expression of the cost function derivatives is unavailable, the accuracy optimization in A-Optim must be conducted using a derivative-free method to optimize the cost function in Equation (6.1). In the following section several robust variant derivative-free algorithms are proposed to maintain the robustness of the initial trajectory computed by the DeepSAMP variant.

6.1.2.2 Algorithms

This thesis considers four robust sensitivity-aware local variants for optimizing the aforementioned cost function, aiming to identify the most suitable approach for optimizing the accuracy cost function while appropriately scaling its complexity to the robust feasibility check described in Chapter 4. It is important to note that none of the proposed robust variants utilize the deep learning method introduced in Chapter 5, as the trajectories encountered during this local optimization process differ significantly from those in the training set. Also note that in this subsection, only the trajectory states are optimization variables. The addition of controller gains in the optimization variables, as mentioned in the general framework (see Section 6.1), is detailed in Section 6.2.4.

Shortcut The first variant is the robust sensitivity-aware variant of the shortcut algorithm [GO07] presented in Chapter 4. This variant offers the advantage of a straightforward implementation, as it simply samples two states from the current best trajectory, attempts a new connection between them, and conducts a single robust feasibility test per algorithm iteration. While this algorithm is computationally efficient, it is constrained by sampling on the current trajectory, limiting exploration w.r.t. the initial guess.

STOMP A robust sensitivity-aware variant of the STOMP algorithm [Kal+11], denoted Sensitivity-Aware STOMP (SASTOMP), is also studied given the widespread use of STOMP for non-differentiable or non-smooth cost functions. The pseudo code is presented in Algorithm 6.3 and operates as follows:

Algorithm 6.3 Sensitivity-Aware STOMP (SASTOMP) $[\boldsymbol{q}^i_{d,init}, N_{roll}]$

1: $\boldsymbol{q}^i_{d,best} \leftarrow \boldsymbol{q}^i_{d,init}, K \leftarrow N_{roll}$;
2: $A \leftarrow$ FiniteDifferenceMatrix();
3: $R^{-1} = (A^T A)^{-1}$;
4: $M \leftarrow$ Scale(R^{-1});
5: **while not** StopCondition() **do**
6: $\quad Q^{noisy} \leftarrow$CreateNoisyTraj$(\boldsymbol{q}^i_{d,best}, K)$;
7: $\quad$ **for** $k = 1 \dots K$ **do do**
8: $\quad\quad c^{noisy}_k \leftarrow$ ComputeSACost(Q^{noisy}_k);
9: $\quad\quad P^{noisy} \leftarrow$ ComputeProbabilities(c^{noisy}_k);
10: $\quad \delta\boldsymbol{q}^{noisy} \leftarrow$ ComputeUpdateVector(P^{noisy});
11: $\quad \delta\boldsymbol{q} = M\delta\boldsymbol{q}^{noisy}$;
12: $\quad \boldsymbol{q}^i_{d,best} \leftarrow \boldsymbol{q}^i_{d,best} + \delta\boldsymbol{q}$;
13: **return** $\boldsymbol{q}^i_{d,best}$;

1. It takes as input an initial discretized 1-D guess $\boldsymbol{q}^i_{d,init}$ and the number of rollouts to perform K (line 1) (i.e. the discretized trajectory along the i-th position axis only). Therefore, the algorithm must be performed independently for each dimension.

2. Then, the algorithm precomputes some smoothing matrices (R and M) based on the sum-of-squared acceleration along the initial trajectory (line 2-4).

3. Next, until a stopping criterion is met (i.e. cost convergence, planning time, etc.), the algorithm generates K noisy trajectories around the current best trajectory using the CreateNoisyTraj procedure (line 6). The noise for each time step is drawn from a normal distribution, with the smoothing matrices used as the covariance to ensure smooth, noisy trajectories.

4. For each of the K noisy trajectories, the ComputeSACost procedure computes the trajectory cost at each time step (line 8), along with the probabilities for each time step that the current noisy parameters contribute to an improved cost (ComputeProbabilities) (line 9). Note that this cost function is also responsible for penalizing the collisions. In the current sensitivity-aware variant, the cost at each time step is therefore a sum of the following:

   - If the time step is the last, the cost described in Section 6.1.2.1 is added; otherwise, a value of 0.0 is added for the other states.
   - Infeasible control inputs and collisions are penalized by adding a value of 1.0.

5. Such probabilities are then used to create the update vector $\delta\boldsymbol{q}^{noisy}$ (line 10), by selecting, at each time step, the noisy update that has the best contribution for improving the cost function. The update vector is then smoothed (line 11), and finally, the current best trajectory is updated (line 12).

While this algorithm allows for broader exploration relative to the initial guess, it suffers from the need to perform multiple rollouts at each iteration.

ExtendedShortcut This thesis introduces the Sensitivity-Aware ExtendedShortcut (SAExtendedShortcut) algorithm (Algorithm 6.4), designed to merge the benefits of the aforementioned algorithms while minimizing their drawbacks.

The proposed algorithm follows the same workflow as the SAShortcut proposed in Chapter 4. It is initialized with an initial trajectory q_d^{init} and its associated cost (line 1-2). Note that the algorithm, unlike to the SAShortcut, takes two additional arguments: K which refers to a number of samples, and δ, that refers to a ball radius. Then, until a stopping criterion is met, the algorithm samples K states within a ball of radius δ centered at a first random state of the current best trajectory, and K others around a second random state of the trajectory, thus generating two sets of samples Q_d^1 and Q_d^2 (line 4). The choice of the sampling strategy (e.g. gaussian sampling) is further discussed in Section 6.2.4. Then, all the possible shortcuts between the two sets are computed and tested for collisions without considering robust constraints (line 5-8). If a current shortcut is collision free, the full trajectory from the initial state to the goal state using this shortcut is recomputed and stored in a set denoted $Q_{d,new}$ (line 9-11). Then, for each trajectory in $Q_{d,new}$, the ODEs are solved before performing the cost comparison (line 13-15). Finally, if the new solution has a lower sensitivity cost (line 16), then a robust feasibility check is performed before updating the trajectory portion in case of success (line 17-18).

Note that the idea behind the proposed algorithm is to explore the vicinity of the initial trajectory while keeping the computational cost low, based on well-chosen hyper-parameters K and δ.

Algorithm 6.4 SAExtendedShortcut $[\boldsymbol{q}_{d,init}, K, \delta]$

1: $\{\boldsymbol{q}_{d,best}, cost_{best}\} \leftarrow \{\boldsymbol{q}_{d,init}, \text{Cost}(\boldsymbol{q}_{d,init})\}$;
2: $q_d^{init} \leftarrow \boldsymbol{q}_{d,best}^0$; $q_d^{goal} \leftarrow \boldsymbol{q}_{d,best}^F$;
3: **while not** StopCondition() **do**
4: $\quad \{Q_d^1, Q_d^2\} \leftarrow \text{SampleOnTraj}(\boldsymbol{q}_{d,best}, K, \delta)$;
5: $\quad$ **for** each $q_d^1 \in Q_d^1$ **do**
6: $\quad\quad$ **for** each $q_d^2 \in Q_d^2$ **do**
7: $\quad\quad\quad \boldsymbol{q}_{d,shct} \leftarrow \text{Steer}(q_d^1, q_d^2)$;
8: $\quad\quad\quad$ **if** CollisionFree($\boldsymbol{q}_{d,shct}$) **then**
9: $\quad\quad\quad\quad \boldsymbol{q}_{d,start} \leftarrow \text{Steer}(q_d^{init}, q_d^1)$;
10: $\quad\quad\quad\quad \boldsymbol{q}_{d,end} \leftarrow \text{Steer}(q_d^2, q_d^{goal})$;
11: $\quad\quad\quad\quad Q_{d,new} \leftarrow \boldsymbol{q}_{d,start} + \boldsymbol{q}_{d,shct} + \boldsymbol{q}_{d,end}$;
12: $\quad$ **for** each $\boldsymbol{q}_{d,new} \in Q_{d,new}$ **do**
13: $\quad\quad S_0 \leftarrow \text{GetNodeConditions}(q_d^{init})$;
14: $\quad\quad \{\boldsymbol{q}_n, \boldsymbol{u}_n, \boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}, S_F\} \leftarrow \text{SolveODEs}(\boldsymbol{q}_{d,new}, S_0)$;
15: $\quad\quad cost_{new} \leftarrow \text{Cost}(\boldsymbol{q}_{d,new})$;
16: $\quad\quad$ **if** $(cost_{new} < cost_{best})$ **then**:
17: $\quad\quad\quad$ **if** IsRobust($\boldsymbol{r}_{\boldsymbol{q}}, \boldsymbol{r}_{\boldsymbol{u}}, \boldsymbol{q}_n, \boldsymbol{u}_n$) **then**
18: $\quad\quad\quad\quad \{\boldsymbol{q}_{d,best}, cost_{best}\} \leftarrow \{\boldsymbol{q}_{d,new}, cost_{new}\}$;
19: **return** $\boldsymbol{q}_{d,best}$

COBYLA The last algorithm considered for optimizing the accuracy cost function of Section 6.1.2.1 under robust constraints is the widely used Constrained Optimization BY Linear Approximations (COBYLA) [Pow94]. The algorithm does not require the computation of gradients, instead it linearly approximates the cost function through each iteration of the optimization process. Furthermore, it handles both equality and inequality constraints such as control inputs saturation and collision checking.

Note that, unlike the aforementioned methods, COBYLA works deterministically by iteratively improving a set of candidate solutions using linear approximations of the cost function. At each iteration, the algorithm replaces the worst candidate with a new one, which is chosen through a reflection process that consist of reflecting the worst candidate across the mean of the remaining candidates. Then, starting from the same initial guess, the linear approximation and the subsequent rejected and added candidates will always be identical.

6.2 Simulation results

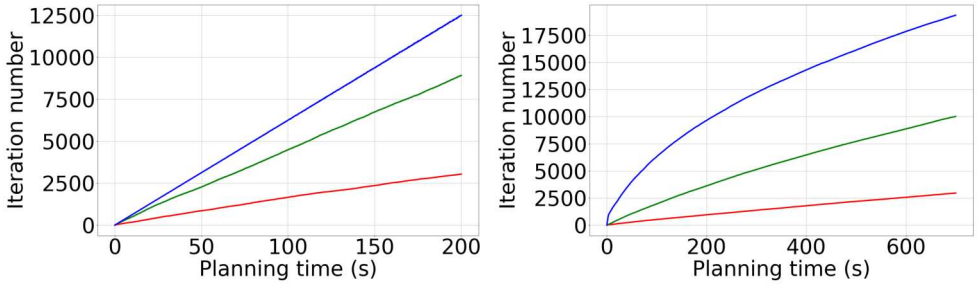
This section presents simulation results for the quadrotor robot. It begins with an overview of the quadrotor setup in Section 6.2.1. Preliminary results on the integration of GRU-based neural networks within DeepSAMP variants are then discussed. Following this, the robustness of motion plans leveraging this deep learning technique is evaluated. Finally, a comparison of robust local accuracy optimization methods is presented.

6.2.1 Quadrotor setup

To further evaluate the effectiveness of the decoupled approach, the quadrotor model described in Section 3.2.2 is used with the following set of uncertain parameters: $\boldsymbol{p} = [m,\, x_{cx},\, x_{cy},\, J_x,\, J_y,\, J_z]^T \in \mathbb{R}^6$. These parameters are chosen according to the experimental scenarios described in Section 6.3. In these scenarios, the quadrotor either carries or catches objects with unknown mass, introducing uncertainty in the total system mass as well as its principal moment of inertia (J_x, J_y, J_z). Transporting or catching this unknown mass also displaced the CoM of the robot.

The values of the nominal parameters are $\boldsymbol{p}_c = [1.113, 0.0, 0.0, 0.015, 0.015, 0.007]^T$ and their associated uncertainty range $\delta\boldsymbol{p} = [7\%, 3cm, 3cm, 10\%, 10\%, 10\%]^T$, which represents the percentage variation of the parameters w.r.t. their associated nominal value, except for x_{cx} and x_{cy} whose nominal values are zeros. Therefore, in this quadrotor application, $\boldsymbol{\Pi} \in \mathbb{R}^{13\times 6}$, and $\boldsymbol{\Theta} \in \mathbb{R}^{4\times 6}$ resulting in solving 115 ODEs to compute the uncertainty tubes. The uncertainty tubes considered for this application are defined as $\boldsymbol{r_q} = [r_x,\, r_y,\, r_z]^T \in \mathbb{R}^3$, representing the uncertainty tubes along the {x,y,z}-axes of the state respectively, and $\boldsymbol{r_u} = [r_{u1},\, r_{u2},\, r_{u3},\, r_{u4}]^T \in \mathbb{R}^4$, representing the uncertainty tubes along the four control input space axes. The controller gains are set to $\boldsymbol{k}_x = [20.0,\, 20.0,\, 25.0]^T$, $\boldsymbol{k}_v = [9.0,\, 9.0,\, 12.0]^T$, $\boldsymbol{k}_R = [4.6,\, 4.6,\, 0.8]^T$, and $\boldsymbol{k}_\omega = [0.5,\, 0.5,\, 0.08]^T$. Finally, considering that the robot hovering state is achieved with propeller speed around 70 $rad.s^{-1}$ (i.e. 5.000 $(rad.s^{-1})^2$), the input limits are set to propeller speed of 100 $rad.s^{-1}$, which results in $\boldsymbol{u}_{max} = [10.000,\, 10.000,\, 10.000,\, 10.000]^T$.

6.2.2 DeepSAMP vs SAMP



(a) RRT (b) RRT*

Figure 6.1: Number of (a) RRT / (b) RRT* iterations as a function of planning time in an obstacle-free environment using the standard (non-robust) RRT/RRT* implementation (blue), compared to DeepSAMP variants that leverage the GRU-based tube prediction (green) or the SAMP variants (see Chapter 4) (red), that solve the entire set of ODEs.

This section presents comparison between the DeepSAMP variants (DeepSARRT and DeepSARRT*), that leverage the GRU-based neural network, and the SAMP variants that solve the set of ODEs.

Figure 6.1 shows the significant performance improvement of using this learning-based prediction within a sampling-based tree planner for checking the robustness of the local tree expansions (see Algorithm 6.2), against the SAMP variants that solve the entire set of ODEs (see Chapter 4). Results provided for RRT and its RRT* near time-optimal variant compare the number of iterations of the main loop of the algorithm as a function of computing time in an obstacle-free environment, showing in both cases a significant time gain thanks to the proposed learning method compared with integrating the entire set of ODEs. Note that in the case of RRT, this time gain is constant (3 times faster) because the expansion benefits from the neural network only once per iteration. In the case of RRT*, the denser the tree, the more robust collision tests are required for the rewiring connection phase. Therefore, much more time is saved when using the learning method. The gain on the planning time can reach more than one order of magnitude for problems requiring a significant amount of iterations.

This time saving is also illustrated by Figure 6.2, that shows for both asymptotically optimal and non-optimal sensitivity-aware variants, the contribution of solving the ODEs or performing GRU predictions to the total planning time after 1.000 algorithms iterations. Note that the RRT* and its sensitivity-aware variant optimize the trajectory time. The sensitivity is then only used to enforce robust constraints. The results show that the combination of solving solely the system dynamic and performing GRU predictions is less costly than solving the entire ODEs set. Furthermore, one can note that in all sensitivity-aware variants, the contribution of the collision checking is overall higher than in their standard counterparts.

However, note that the observed time gains differ from those reported in Chapter 5 due to certain limitations:

- The current implementation employs the Libtorch C library, which supports only

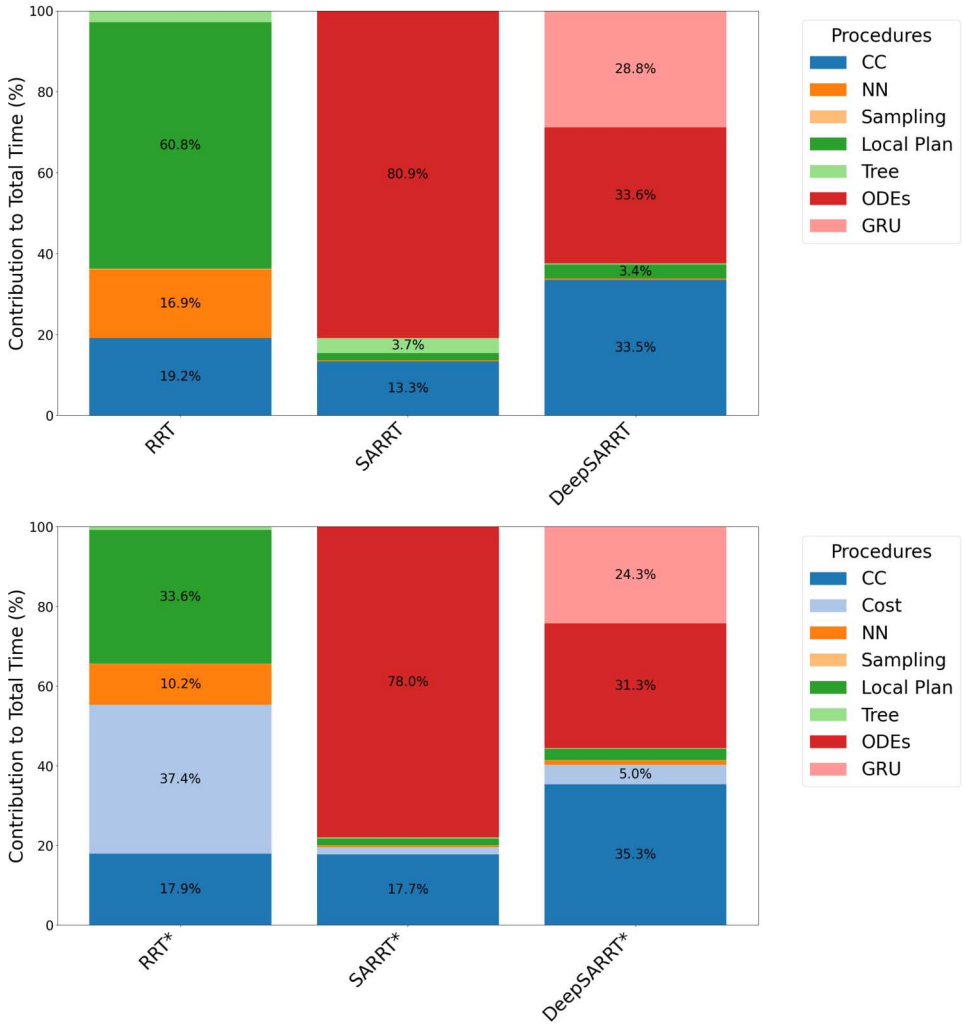


Figure 6.2: Profiling comparison between vanilla planners, DeepSAMP variants and SAMP variants, illustrating the contributions of various procedures to the total planning time over 1.000 iterations in a free space environment.

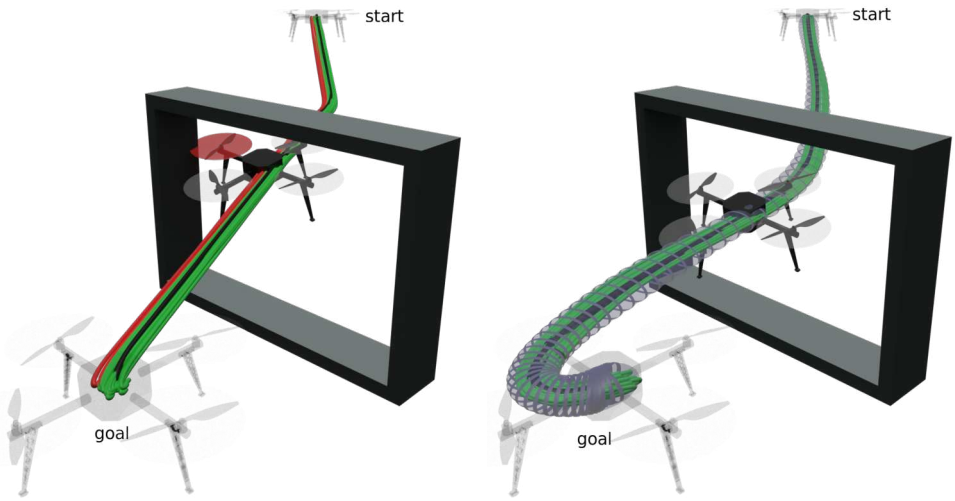
CPU computation and not GPU acceleration. Consequently, the GRU inference time is a bit slower, showing an average time of 2.8 ms on CPU against 2.1 ms on GPU for 1.000 predictions. However, this does not hurt the overall gain of the GRU-based neural networks.

- The GRU-based approach allows to directly correlate the desired trajectory to the sensitivity-based uncertainty tubes. However, the proposed DeepSAMP algorithms still require solving the nominal system dynamic through the ODEs of Equation (3.1). Nevertheless, the use of the GRU enables decoupling the robust feasibility test, unlike the SAMP method (see Chapter 4), where control inputs depend on the current robot state, and therefore must be recovered by solving the ODEs. This decoupling allows to perform a first robust control inputs checking that allow to filter 18% of trajectories during a 200-second planning in a free space environment. This process eliminates the need to solve the system dynamic ODEs

(Equation (3.1)), and perform robust collision checks for these trajectories.

Finally, note that the GRU predictions are only valid for the parameter maximum range $\delta \boldsymbol{p}$ chosen during the generation of the training set, and that the model is trained for given values of the controller gains.

6.2.3 Robust motion planning via deep learning



(a) RRT* (b) R-SARRT*

Figure 6.3: Planned trajectory (black) produced by a (a) RRT* and (b) DeepSARRT*. Simulated trajectories under uncertainty are displayed in green in the case of success, and in red in the case of a crash.

The efficiency and robustness of the DeepSARRT planner (see Section 6.1.1) are also demonstrated through comparative simulation results against a standard (non-robust) RRT, RandUp-RRT [Wu+22], the SARRT variant solving the full set of ODEs as detailed in Chapter 4, and LazySARRT, an RRT implementation of LazySAMP (see Chapter 4). The RandUp-RRT has been implemented with 20 "RandUP particles" to approximate the reachable set, and no ε-padding is used. The padding value is a user parameter that is difficult to tune. Choosing a bad padding value can result in set estimations that are too conservative. Therefore, it is set to zero in the following experiments as in experiments of [Wu+22]. The LazySARRT performs robust feasibility tests only when a solution is found, disconnecting and reconnecting the tree if necessary via a RobustReconnect procedure, as shown in Algorithm 4.5. This procedure maintains and updates a set of non-robust connections for each node in the tree. In the case of LazySARRT*, the RobustReconnect procedure also ensures tree optimality. Consequently, all nodes, starting from the first non-robust node, are disconnected and reconnected optimally. In contrast, the LazySARRT does not consider optimality. Therefore, children nodes are disconnected from their parent and reconnected to the tree only if their parent is deleted. This strategy significantly reduces the number of reconnection attempts.

	Basic Planners				
	RRT	RandUp-RRT	SARRT	LazySARRT	DeepSARRT
Success (%)	61.8	99.2	**100.0**	**100.0**	**100.0**
Plan time (s)	**7.1** ± 7.6	57.8 ± 49.1	78.5 ± 55.1	45.6 ± 33.4	22.3 ± 15.8

Table 6.1: Quadrotor application: Average planning time and robust feasibility success rates of the simulated motions planned using standard non-robust RRT, RandUP-RRT, the SARRT variant (see Chapter 4), LazySARRT (see Chapter 4), and DeepSARRT. The results are averaged over 20 plans and 30 simulations per plan.

	Optimal Planners			
	RRT*	SARRT*	LazySARRT*	DeepSARRT*
Success (%)	56.5	**100.0**	**100.0**	**100.0**
Plan time (s)	**308.7** ± 235.7	5459.5 ± 817.5	834.3 ± 380.0	584.3 ± 394.7

Table 6.2: Quadrotor application: Average planning time and robust feasibility success rates of the simulated motions planned using standard non-robust RRT*, SARRT* (see Chapter 4), LazySARRT* (see Chapter 4) and the DeepSARRT* variant, all of them optimizing trajectory length. The results are averaged over 20 plans and 30 simulations per plan, except for the SARRT*, which values are averaged over 3 runs and 30 rollouts per plan due to long planning time.

A DeepSAMP variant of the RRT*, denoted DeepSARRT*, is compared to the classic non-robust RRT* and the two variants proposed in Chapter 4, SARRT* and LazySARRT*. These optimal planners compute near time-optimal trajectories and run until the solution cost converged below a threshold of 15 seconds. The RandUp-RRT was excluded from this comparison, as no optimal version of the algorithm exists.

The comparison is based on the several planning times and success rate of the planners for the scenario depicted in Figure 6.9a, using an Intel i9 CPU@2.6GHz. Table 6.1 and Table 6.2 show comparative results for optimal and non-optimal planners, averaged for each planner over 20 trajectories and 30 simulations with uncertain parameters uniformly sampled in the range $\delta \boldsymbol{p}$ (see Section 6.2.1).

First note that RandUp-RRT, SARRT, LazySARRT and DeepSARRT have a much stronger robustness than standard non-robust RRT. All SARRT, LazySARRT and DeepSARRT have a success rate of 100% compared to 99.2% for RandUp-RRT. Indeed, as mentioned in [Lew+22; Wu+22], the computation of a conservative reachable set requires some additional padding step, which is here set to zero as mentioned earlier.

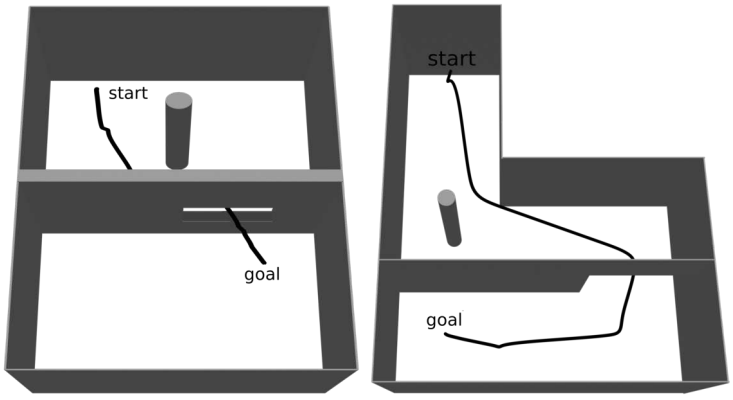
Also note the higher efficiency of DeepSARRT which only uses one prediction per iteration whereas RandUp-RRT needs to perform a propagation per particle, yielding to a longer planning time. Additionally, DeepSARRT demonstrates greater efficiency compared to its SARRT counterparts, achieving a threefold reduction in planning time. As for LazySARRT, it does not build a robust tree like DeepSARRT. Instead, it only robustly checks the final solution, causing frequent disconnections and re-connections of non-robust nodes, which results in a higher planning time. RRT, which does not account for robustness, remains faster but with a significantly lower success rate. Similar results

are observed on the optimal versions. One can note the efficiency of the DeepSARRT* that reach approximately one order of magnitude time gain compared to its SARRT* counterparts. An example of trajectories planned by RRT* and a robust version DeepSARRT* optimizing the trajectory duration, and their simulations of their execution by the controller under different parameters values within range $\delta \boldsymbol{p}$, is illustrated in Figure 6.3. It shows the effective robustness of the proposed algorithm as illustrated by the higher success rate indicated in Table 6.1 and Table 6.2.

6.2.4 Robust local accuracy optimization

This section provides a comparison of the robust sensitivity-aware local optimizers introduced in Section 6.1.2. Since none of the robust local optimizers guarantee convergence to a robust solution, but do guarantee the preservation of initial robustness, the initial trajectories are planned using the DeepSARRT variant. The evaluation focuses on their computation time and resulting costs, with the trajectory generated by the DeepSARRT variant serving as the optimization variable. The objective is to identify the most suitable optimizer for the accuracy optimization stage required for the experimental scenario outlined in Section 6.3.2.

All algorithms are tested on three different environments and stop when a cost convergence at 1% is reached (i.e., the algorithm will stop optimizing the trajectory when the improvement in cost between consecutive best solutions is less than 1%). The cost function used to compute trajectory cost is the one of Equation (6.1), where the radii of interest are the ones along the $\boldsymbol{x}$ (positions), $\boldsymbol{\rho}$ (orientations) and $\boldsymbol{\omega}$ (angular velocities) components of $\boldsymbol{q}$.



(a) Window environment (b) Corridor environment

Figure 6.4: The two environments considered for the comparison of robust local optimizers, displayed with an initial trajectory guess for the optimization planned by DeepSARRT (black): (a) the window environment with an additional obstacle, and (b) a narrow corridor environment.

Free space environment First, the algorithms are tested in an environment without obstacles to evaluate their ability to escape the local minimum defined by the initial tra-

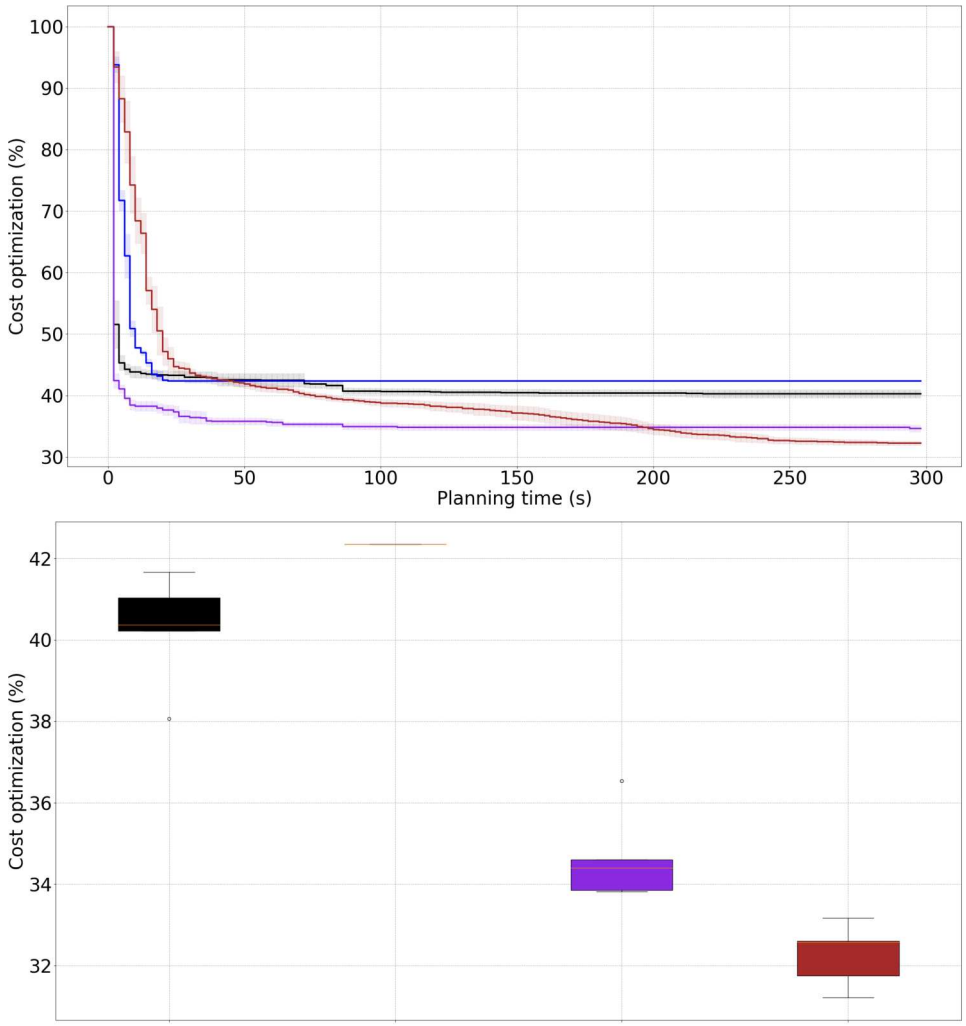


Figure 6.5: Free-space environment: cost optimization results for the four methods outlined in Section 6.1.2, averaged over 10 plans in a free space environment. The standard deviation is represented as an envelope around the mean curves. The SASTOMP is shown in red, SAShortcut in black, SAExtendedShortcut in purple, and COBYLA in blue.

jectory. SASTOMP performs 10 rollouts and uses a standard deviation of 0.2 m to generate noisy trajectories. SAExtendedShortcut is evaluated with different hyperparameters chosen as follows: $\delta \in \{0.1, 0.01\}$ and $K \in \{1, 2, 3, 5, 10\}$. Two different sampling strategies are also tested: a uniform sampling, where the same radius δ is used to sample state values within a ball of radius δ around the current trajectory state, and a Gaussian sampling, where δ is used as the standard deviation for the sampling distribution.

Figure 6.5 provides the optimization results of the four local optimizers in a free space environment. The SAExtendedShortcut algorithm is executed using a Gaussian sampling, with $\delta = 0.1$ and $K = 1$, as they are the best hyperparameters find for this scenario according to a grid search that can be found in Appendix C. First, note how the COBYLA algorithm remains local in the vicinity of the initial guess. This behavior suggests the inherent difficulty in linearly approximating the cost function at hand. Furthermore, note that no standard deviation is observed for COBYLA denoting

the deterministic nature of the algorithm.

Then, one can note that the SASTOMP achieves the best optimization, but with significantly slower convergence compared to both SAShortcut and SAExtendedShortcut. Finally, the latter two methods achieve convergence within the same planning time, with a better optimization achieved by SAExtendedShortcut due to its better ability to explore the surrounding of the initial guess.

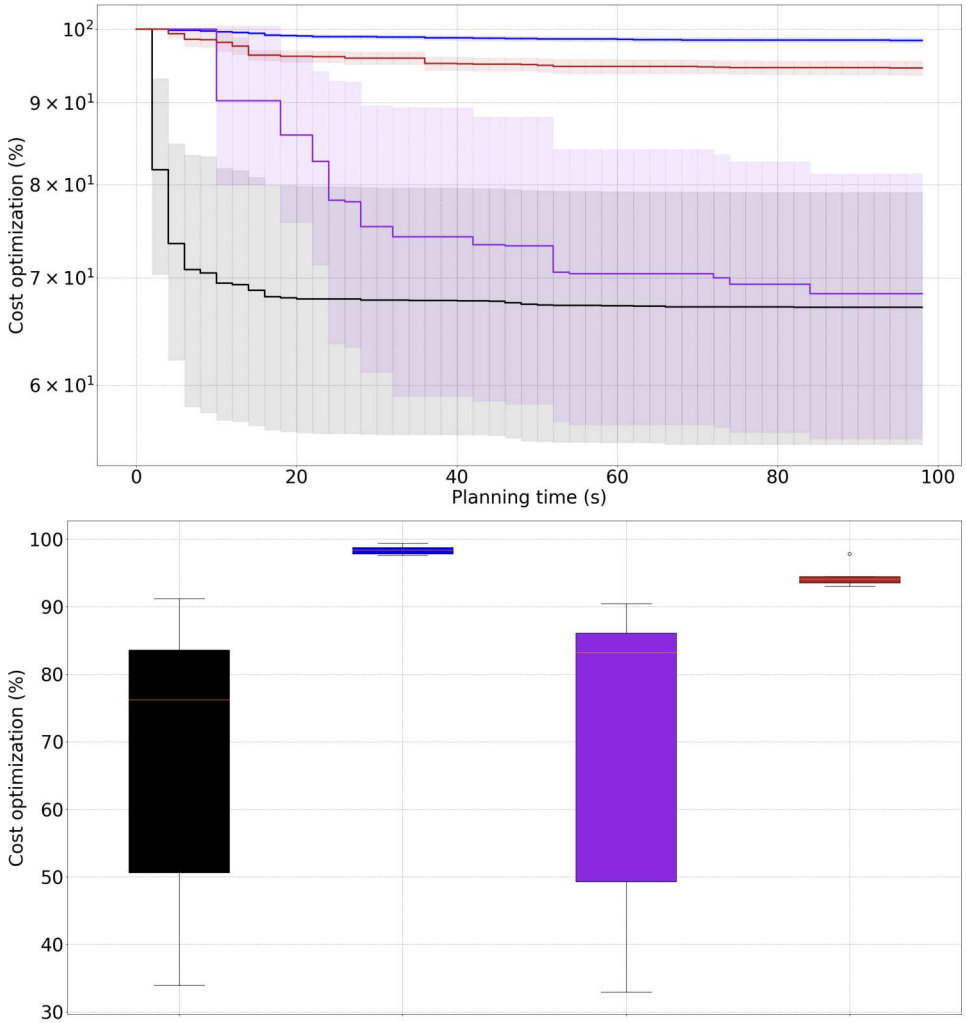


Figure 6.6: Window environment: cost optimization results for the four methods outlined in Section 6.1.2, averaged over 10 plans in the window environment. The standard deviation is represented as an envelope around the mean curves. The SASTOMP is shown in red, SAShortcut in black, SAExtendedShortcut in purple, and COBYLA in blue.

Window environment In order to assess the scalability of the proposed robust optimizer when handling the robust feasibility check, the methods are compared in the environment depicted in Figure 6.4a. The optimization results are shown in Figure 6.6. First, note that the COBYLA algorithm still struggles to explore larger cost areas beyond the vicinity of the initial solution. Furthermore, the SASTOMP also faces difficulty to

efficiently explore the surrounding of the initial trajectory, as denoted by its very slow convergence. This indicates an inefficient scalability of the algorithm to robust constrained problems.

Both the SAShortcut and SAExtendedShortcut achieve fast convergence with efficient optimization. However, while SAExtendedShortcut outperforms SAShortcut in the obstacle-free environment presented above, this is not the case in the presence of obstacles. This suggests that SAExtendedShortcut is sensitive to the choice of appropriate hyperparameters when robust collision checks are involved.

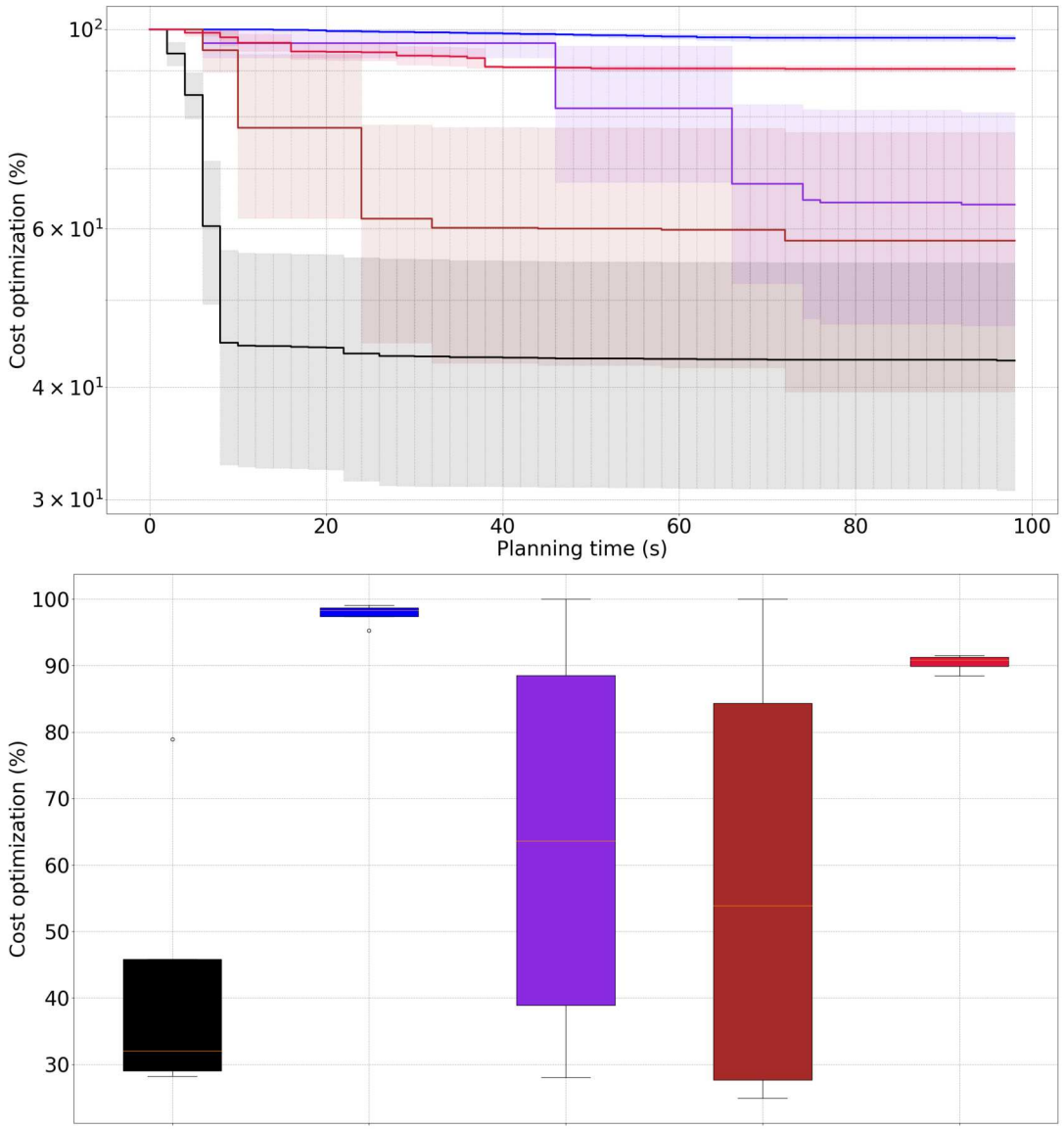


Figure 6.7: Corridor environment: cost optimization results for the four methods outlined in Section 6.1.2, averaged over 10 plans in the corridor environment. The standard deviation is represented as an envelope around the mean curves. The SASTOMP is shown in red, SAShortcut in black, and COBYLA in blue. Two SAExtendedShortcut setup are considered: both use a Gaussian sampling with $K = 1$, but the purple curve uses a standard deviation of $\delta = 0.1$, while the brown one uses $\delta = 0.01$

Corridor environment A second environment is also considered, as depicted in Figure 6.4b, to further analysed the SAExtendedShortcut sensitivity to its hyperparameters. The results are illustrated by Figure 6.7, where COBYLA and SASTOMP present the same behavior as previously observed.

The SAShortcut achieves the best optimization together with the best convergence rate. The SAExtendedShortcut is tested with a Gaussian sampling using two different standard deviation $\delta \in \{0.1, 0.01\}$ and $K = 1$. The results show that although $\delta = 0.1$ is the best setup to achieve convergence in the obstacle-free environment, it is not the best choice in this case. This confirms the sensitivity of the algorithm to its hyperparameters.

Controller gains optimization While the robust local optimizers compared above consider only the trajectory states as optimization variables, the work in [SFR23] demonstrates that optimizing both the states and controller gains achieves the best accuracy performance. Therefore, SAShortcut has been implemented such that, at each iteration, a shortcut is attempted between two states of the input trajectory, randomly sampled along with controller gain values, which are chosen within 50% to 150% of their nominal values (see Section 6.2.1). Figure 6.8 illustrates how the uncertainty radius can be reduced by locally optimizing either the trajectory alone (yellow), the controller gains alone (blue), or both simultaneously (green), compared to the robust initial trajectory (red). This motivates the interest of optimizing both the trajectory and the controller gains at the same time in the A-Optim function in order to minimize uncertainty for a given point. In fact, these results corroborate the findings of [SFR23], but this time by employing a sampling-based motion planner that considers obstacles in the environment.

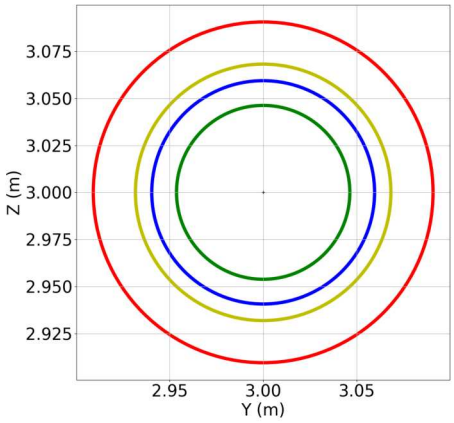


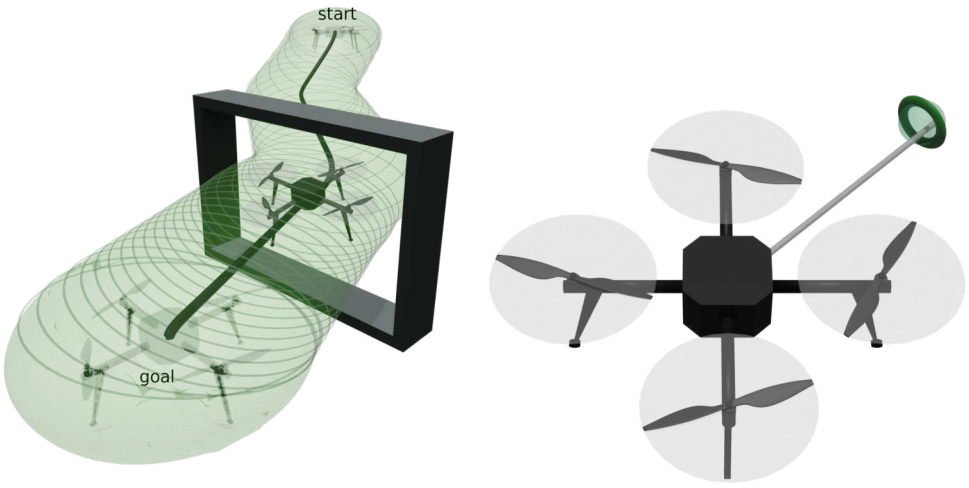
Figure 6.8: Example of uncertainty ellipsoid without optimization (red), with local trajectory optimization (yellow), with gains optimization (blue), and with local trajectory and gains optimization at the same time (green).

Conclusion The COBYLA algorithm remains highly localized to the vicinity of the reference trajectory, while SASTOMP exhibits a very slow convergence rate. Although SAExtendedShortcut was designed to enable better exploration of the cost space compared to SAShortcut, while maintaining an efficient convergence rate, it proves challenging to tune effectively. Therefore, in the following experimental section, the robust local

accuracy optimization is performed using SAShortcut considering both the trajectory states and controller gains as optimization variables.

Recall that, because the GRU is trained for specific controller gains, and the trajectories encountered during the SAShortcut process deviate significantly from those used during training (due to the violation of the maximum local distance during the shortcut process), SAShortcut does not benefit from the neural network.

6.3 Experimental validation



(a) Robust motion planning (b) Accuracy optimization

Figure 6.9: Two scenarios considered for the experimental validation of the proposed method: (a) Robust navigation of a drone through a narrow window. (b) Precision in-flight "ring catching" task, where the uncertainty on the position of the perch end-effector is minimized to successfully accomplish the task. A video of the experiments is available at: https://laas.hal.science/hal-04642257

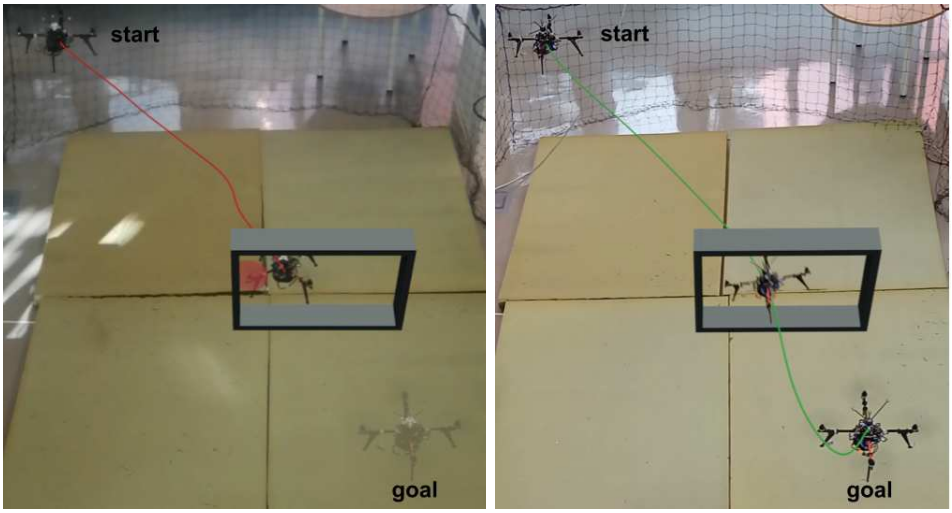
This section provides an experimental validation of both the deep learning-based robust motion planning variants, and the accuracy optimization stage in two challenging scenarios depicted in Figure 6.9.

6.3.1 Robust motion planning via deep learning

The window scenario, depicted in Figure 6.9a and evaluated in simulation in Section 6.2.3, is experimentally validated on a real quadrotor. In this scenario, uncertainties are added to the system by randomly attaching a mass of up to 80g (not known by the controller) to the drone as depicted in Figure 6.11a.

In this experiment, a non-robust trajectory planned by RRT* and a robust one planned by DeepSARRT* were executed ten times, using the same masses and attachment points between the two algorithms. All trajectories were planned offline on a remote computer. To make the robot execute them, the geometric controller [LLM10] ran online

on the quadrotor onboard computer, tracking the trajectories provided as input thanks to the Genom3 software [Mal+10]. The robot state was measured using a motion capture system with millimeter accuracy, ensuring that the only source of uncertainty was the attached unknown mass. Figure 6.10 illustrates the experimental execution of a non-robust RRT* trajectory and a robust DeepSARRT* trajectory. The figure shows the recorded executions within a virtual environment to detect virtual collisions, thus mitigating the risk of real crashes and damages to the robot. The experimental results confirm the simulation observations, providing an overall success rate of 100% in the case of the robust trajectory computed with DeepSARRT*, against 40% for the classic RRT*.



(a) RRT* (b) DeepSARRT*

Figure 6.10: Experimental execution by a quadrotor with uncertainty of trajectories planned by RRT* (a) and DeepSARRT* (b). Both trajectories are executed with the same uncertainty and a virtual collision is found in the RRT* case while the DeepSARRT* execution is robust.

6.3.2 Robust motion planning with accuracy optimization

A second experimental scenario is presented in order to evaluate the complete framework presented in Section 6.1. This scenario involves a challenging in-flight retrieval of two 2 cm radius rings in a cluttered environment using a drone equipped with a perch (see Figure 6.11b) in a robust (near) time optimal way. Note that, unlike the first experimental scenario described earlier, the uncertainties in the system parameters in this case arise from the task itself, specifically during the recovery of the rings.

The experimental setup is shown in Figure 6.12, where the quadrotor is equipped with a perch of 50 cm length, aligned at a 45° angle relative to the body frame x and y axes. The perch is made of carbon fiber and 3D-printed parts, with a total mass of 20g. As a result, its addition to the drone is considered negligible in terms of inertia uncertainty.

The positions and orientations of the rings are recovered before the planning process using a motion capture system, as depicted in Figure 6.11c. Based on the positions and

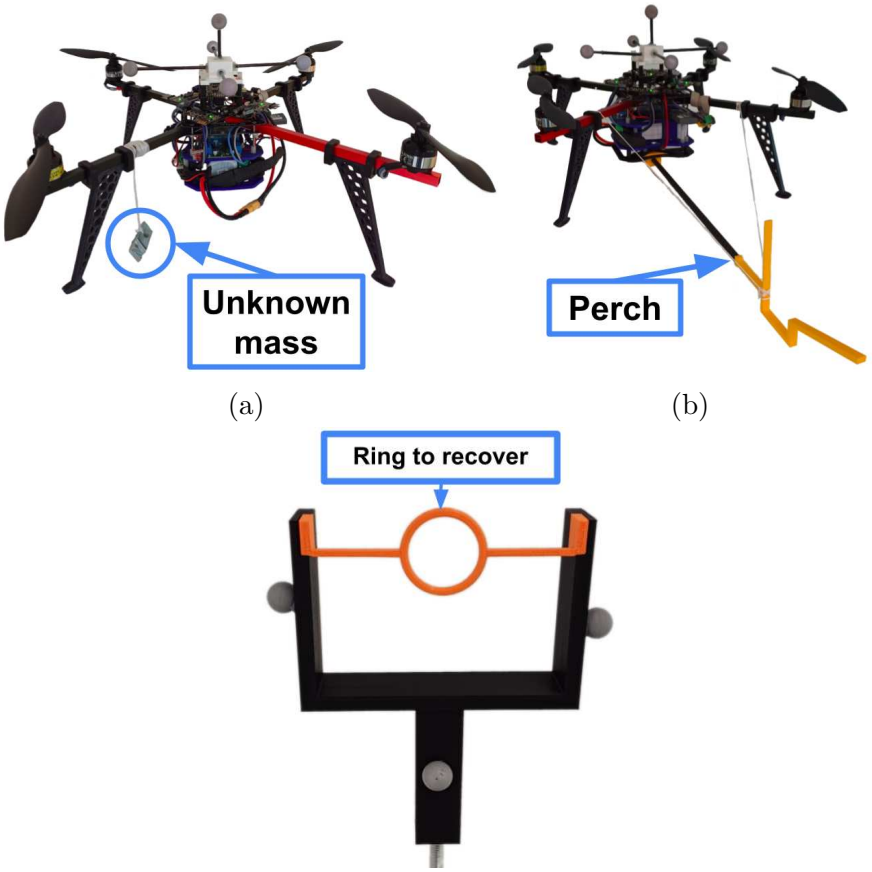


Figure 6.11: Quadrotor setups for the two scenarios considered for the experimental validation. (a) a drone equipped with a random mass to perform a robust navigation through a window (b) a drone equipped with a perch to catch (c) the rings.

orientations of the rings, desired quadrotor states (as for the states in $list_d$ in Section 6.1) is computed to ensure that the incidence angle of the perch is perpendicular to the rings during capture, with zero acceleration. Based on these informations, 10 trajectories were planned using a vanilla (non-robust) RRT* planner and the DeepSARRT* algorithm, both designed to optimize trajectory time, and then executed by the real quadrotor. The RRT* does not use the A-Optim method to optimize accuracy while the DeepSARRT* does, in addition to guaranteeing the robustness. The offline optimization in A-Optim aimed at minimizing the uncertainty at the location of the two rings.

The first ring is weighted between 10 g and 50 g, so when it is caught, it becomes part of the drone and modifies the overall mass/inertia and center of mass of the system in an unmodeled way. Note that the rings are recovered on the perch, which is located at the origin of the z-axis in the body frame. Consequently, it is assumed that catching the rings does not introduce any uncertainties in the center of mass along the z-axis. Therefore, this is why only shifts along the x and y axes are taken into account in the setup (see Section 6.2.1). Note that, to minimize the impact of catching the rings as a source of perturbation, the ring supports are oiled to reduce the friction when picking the rings. A success is characterized by the recovery of both rings, otherwise the execution

is considered as a failure.

Figure 6.12 shows the perch end-effector position at the second ring location in the non-optimized case and in the optimized one. In the latter case, the perch tip is closer to the reference point in the middle of the ring than in the former case. This translates into a higher success rate of nine out of ten attempts to catch the ring with the optimized approach, against only three times out of ten for the non-optimized case. However, given the chosen system and controller parameters, there is no guarantee that the computed tube will be enclosed in within the ring. This explains the occurrence of one failure in the optimized case. Overall, the experimental results show a success rate of 90% for DeepSARRT* against only 30% for RRT*.

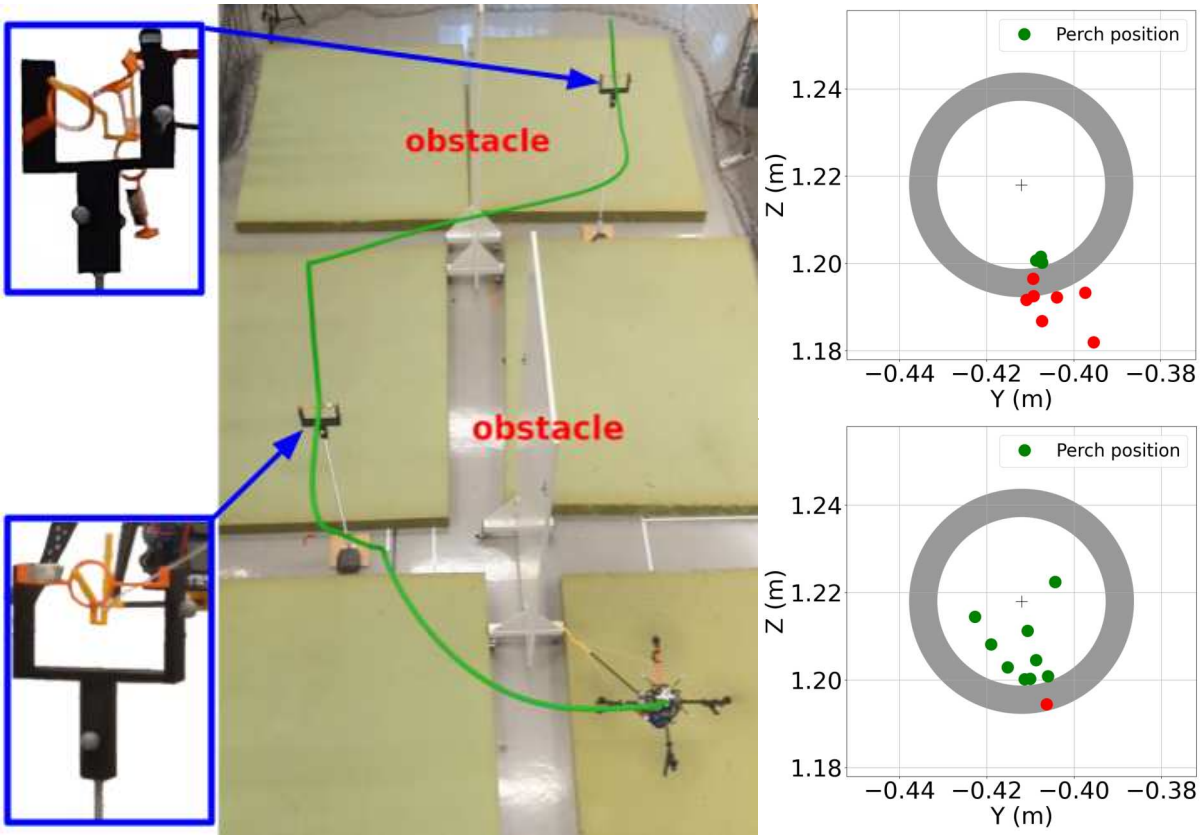


Figure 6.12: Experimental validation of the "ring catching" scenario with a perch-equipped drone (left) with the position of the perch end-effector at the second ring location over 10 trajectories non accuracy optimized (top right) and accuracy optimized (bottom right).

6.4 Conclusion

This chapter has presented a framework able to generate trajectories that are both robust and accurate in the presence of model uncertainties. By leveraging the GRU-based architecture introduced in Chapter 5, more efficient robust sensitivity-aware variants were developed compared to those discussed in Chapter 4. The results on a quadrotor robot

confirm the efficiency of the proposed learning method and highlight the benefit of its integration within a motion planner, resulting in a significant reduction of the planning times.

Moreover, several robust sensitivity-aware variants were proposed for local optimizers in order to minimize the size of the uncertainty tubes of the state at some desired locations, allowing the system to accurately perform a precision task. Extensive simulation results demonstrated that the SAShortcut method is the most suitable for this task in the quadrotor case. Additionally, the optimization variables were extended to include the controller gains, as simultaneously optimizing both the trajectory and the gains achieves the most effective minimization of the cost function. However, it is important to note that while the proposed deep learning-based sensitivity-aware variants leverage the neural network architecture described in Chapter 5, the local optimizers cannot take advantage of this architecture, as the model is specific to the controller gains used during training.

Two experimental demonstrations involving a quadrotor navigating through a narrow window, and a ring-catching task allowed to validate the approach in real conditions. It is worth noting that the experimental validation was made possible by the use of a motion capture system, ensuring that the assumption of a perfectly known system state (see Section 3.1.1) hold. Additionally, the experiments were conducted indoors, eliminating external disturbances and leaving only model uncertainties to affect the system. Therefore, it is crucial that future work consider uncertainties not only in the dynamic model, by extending the computation of the tubes for state estimation uncertainties and external disturbances.

Finally, a new local optimizer called SAExtendedShortcut has been proposed. It leverages the STOMP concept by generating noisy states around the initial trajectory, and combines it with the shortcut method to explore the vicinity of the initial trajectory while maintaining a low computational cost. Preliminary results have shown that the algorithm is sensitive to the sampling radius used to generate noisy states around the initial trajectory. However, adaptive noise generation (i.e. using broader noise when the cost is trapped in local minima and narrower noise when the cost needs refinement) should be further explored as a potentially more efficient technique.

Chapter 7
Conclusion

This thesis proposed motion planners for producing collision-free motions that are robust against parametric uncertainties for a large class of complex dynamical systems. The work proposed in this thesis is hinged on the concept of closed-loop sensitivity, a quantity that relates parameter variations to deviations of the closed-loop trajectory of any given system/controller pair. This metric is employed in sampling-based tree planners to generate trajectories that are inherently robust to parameter uncertainties in various ways, as outlined by the following contributions.

7.1 Contributions

Chapter 4 presented how to incorporate the closed-loop sensitivity concept and the resulting uncertainty tubes within a global sampling-based framework, whereas previous works primarily focused on local trajectory generation. This approach enabled the generation of globally sensitivity-optimal trajectories. Furthermore, for the first time, sensitivity-based uncertainty tubes were used as robust constraints during the planning process, both on the state and control input spaces. However, while this approach results in the generation of robust trajectories to parametric uncertainties, the overall cost remains too high when considering the thousands of computations required by a global sampling-based planner. Thus, a decoupled approach has also been studied, based on a robust lazy strategy, which offers significant computational gains, especially as the system complexity increases.

Whereas Chapter 4 leverages a lazy robust collision checking approach to address the computational challenges posed by high cost of sensitivity-based uncertainty tube computations required to ensure robustness in standard tree-based applications, Chapter 5 exploits the structural similarity between the set of ordinary differential equations (ODEs) required for tube computation and Recurrent Neural Networks (RNNs), directly correlating the desired trajectory to the uncertainty tubes and thereby eliminating the need to solve them. According to the task at hand, the approach proposed a method for generating the dataset using a sampling-based approach. Relying on a simple Gated Recurrent Unit (GRU) cell, the proposed architecture demonstrates an effective balance between prediction accuracy and inference time, achieving a time reduction of one order of magnitude compared to solving the set of ODEs.

This GRU-based architecture has been used in Chapter 6 to propose an efficient alternative to the lazy approach presented in Chapter 4. Deep learning-based sensitivity-aware variants were proposed, demonstrating their efficiency in generating robust motions within a manageable offline planning time. Furthermore, while in Chapter 4 the sensitivity was used to generate globally sensitivity-minimal trajectories throughout the entire

motion, it is used in the proposed Deep-learning based variants to achieve accuracy optimization, considering the uncertainty tubes only at specific locations along the motion, based on the task at hand. An extensive simulation campaign was conducted to identify the most effective local optimizer for optimizing such task-based cost functions, by considering both the trajectory and the controller gains as optimization variables. Finally, both the robust planners and the accuracy optimization stage were evaluated in two challenging scenarios: a quadrotor navigating through a narrow window and an in-flight ring-catching task requiring high precision. These tests validated the efficiency of the proposed methods in an indoor environment.

7.2 Future Works

Although this work represents a step toward an efficient globally robust and general control-aware motion planner framework, it could be further improved both from a theoretical perspective and an algorithmic point of view.

Theoretical First, as mentioned in Chapter 4, none of the proposed Sensitivity-Aware Motion Planner (SAMP) variants are proven complete. While the Deep learning-based Sensitivity-Aware Motion Planner (DeepSAMP) variants offer more practical implementations, they lack formal guarantees due to the unpredictable nature of learning models. However, future work could focus on bounding the first-order sensitivity-based tube approximations to establish the robust feasibility of the SAMP solutions and, consequently, their completeness.

As discussed in Section 3.1.1, the computation of closed-loop sensitivity and the subsequent uncertainty tubes rely on the assumption of perfect knowledge of the robot state. Therefore, the current system does not account for uncertainty in state measurements, which is unavoidable due to sensor bias and noise. Consequently, one could consider computing uncertainty tubes by also accounting for a priori known measurable sensor biases and noise, using the Gramian [Sal+19], which is a metric that quantifies how the robot state can be inferred from its outputs.

While this thesis focuses primarily on parametric uncertainties and presents indoor experiments, a step toward a more general framework applicable outdoors would involve accounting for external system disturbances, such as wind. Since the closed-loop sensitivity of any function of the state can be computed, external disturbances might be incorporated by evaluating the closed-loop sensitivity of barrier functions, for instance.

Algorithmic From an algorithmic perspective, a new local optimizer called ExtendedShortcut was proposed in Chapter 6. This algorithm aims to combine a broader exploration of the initial trajectory surroundings by using noisy state sampling within a shortcut process. Simulation results have shown promising outcomes. However, further investigation into a more effective noisy state generation process, based on adaptive sampling, could help reduce the algorithm sensitivity to its hyperparameters.

The Lazy Sensitivity-Aware Motion Planner (LazySAMP) variants can be further improved by developing a more sophisticated and robust reconnection procedure that not

only maintains a set of non-robust parents but also leverages the GRU-based architecture to ensure robust node reconnections. The Lazy Sensitivity-Aware Motion Planner (LazySAMP) variants encounter difficulties in efficiently robustly reconnecting the tree when facing numerous obstacles. Therefore, lazy robust feasibility checks may be employed to minimize tube computation frequency, and a deep learning-based reconnection strategy may be used to decrease the number of reconnection attempts per node, resulting in a faster convergence toward a robust solution.

Addressing additional constraints, as mentioned earlier, in the presence of external disturbances and state estimation uncertainties can be challenging to learn using the proposed learning architecture, as these factors can lead to more complex closed-loop sensitivity computations. Therefore, the learning process can be improved to handle more complex dynamics, whether from the robot or the sensitivity formulation, by leveraging a Physics-Informed Neural Networks (PINNs) learning procedure that integrates ODEs dynamics within the loss function.

Furthermore, the current learning method is limited to the model parameters and controller gains used for the dataset generation. Therefore, to further address these dependencies, future investigations could explore strategies such as:

- GPU acceleration leverages the massive parallelization capabilities of modern GPUs to enhance computational performance. Utilizing GPU acceleration techniques can enable rapid tube computation while overcoming limitations of the learned model, such as dependency on specific parameters.

- Transfer learning is a method that enables a model to apply knowledge gained from one task or dataset to a new, related task, thereby reducing the need for extensive retraining. This approach can be leveraged when model parameters or controller gains change, allowing for the rapid retraining of learned models.

- Ensemble learning is a technique that involves combining multiple models to improve prediction accuracy and generalized performance. It can be employed to enhance the stability of predictions when dealing with changes in model parameters or controller gains.

Finally, the proposed techniques can also be extended to more complex systems like aerial manipulators, multi-robots systems, or human-robot (physical) interactions, where ensuring the accuracy and robustness of robot motion is critical for guaranteeing user safety.

Appendix A

Robust feasibility checking

Sampling-based algorithms generate global trajectories by combining multiple continuous local trajectories. During the process, each of these local trajectories are subject to collision checks to determine their feasibility. However, in this thesis, the collision check is extended to include a more general feasibility test, which also accounts for the control input space. This extension ensures that the generated trajectories are not only free from obstacles but also prevent control inputs from reaching saturation. Additionally, it accounts for uncertainty in both the state and control input spaces, further enhancing the robustness of the trajectories. The following appendix describes how these extended feasibility check is performed to ensure the robustness of each local trajectory in this context. It is important to note that these test is not continuous; rather, it is performed along a discrete representation of the local trajectories, sampled at the controller's operating frequency.

Robust collision checking In this thesis, collision detection is performed using the widely used C implementation of PyBullet [CB23], which operates as follows:

1. It starts with a broad-phase collision detection using Axis-Aligned Bounding Boxes (AABBs) to quickly eliminate pairs of objects that are too far apart to collide, allowing the more computationally intensive collision checks to focus only on pairs that are potentially close to each other.

2. Then, it performs a narrow-phase collision detection that, after potential collision pairs are identified, checks for each pair. For each identified potential collision pair, this phase uses a more precise robot representation (as defined by the user) and specialized collision algorithms to detect actual intersections and determine contact points, normals, and penetration depths.

Extending this procedure to account for robot state uncertainty aims to verify that the resulting extended bounding volume, which the robot may occupy due to the uncertainty, is clear of obstacles. In this work, such bounding volume is computed by considering only the uncertainty in the position subspace for simplicity (i.e., the $\{x, y, z\}$-subspace for the quadrotor or the $\{x, y\}$-subspace for the differential drive robot).

This work employ a robust collision check generic to all free flying robots that operates as follows:

1. As mentioned above, the first phase performs a broad collision detection considering the current robot AABB. Therefore, in the robust collision detection of this work, this phase involves creating an extended AABB by scaling the current robot AABB in all directions according to their respective uncertainty radii as shown in Figure A.1.

2. The narrow phase approach leverage a fine robot representation. However, generating the accurate extended collision mesh that incorporates uncertainty, required for the second phase, is challenging, as it involves deforming all the vertices of the robot mesh. To address this, this work approximate the extended collision mesh by sampling on the surface of the uncertainty ellipsoid bounding box defined by the uncertainty tube radii (see Figure A.1). While this method accurately approximates the true extended collision shape, it requires multiple calls to the collision-checking function for each robot state tested.

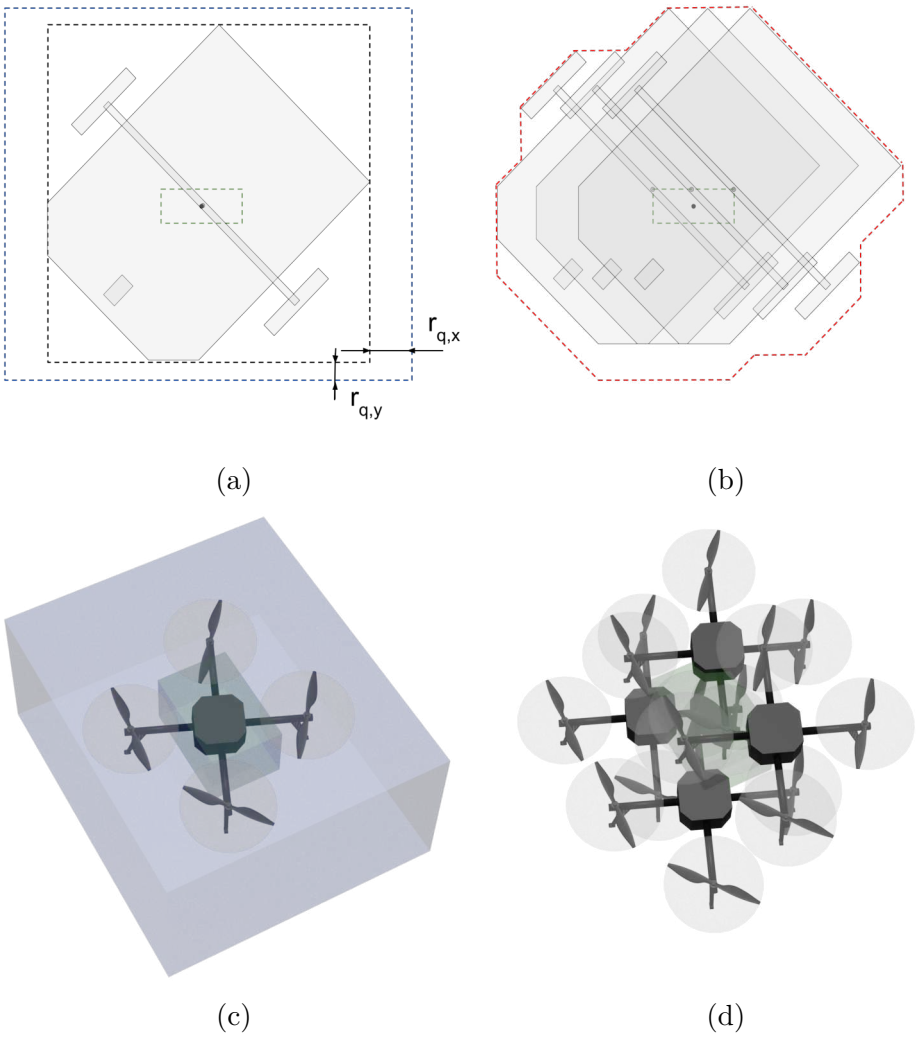


Figure A.1: Figure showing the resulting shapes tested for collisions: (a) differential drive robot extended AABB, (b) sampling-based approximated uncertain mesh, for a differential drive robot with its associated uncertainty ellipsoid bounding box (green), (c) quadrotor extended AABB, and (d) sampling-based approximated uncertain mesh, for a quadrotor with its associated uncertainty ellipsoid bounding box (green). Note that not all the sampled configurations are displayed for clarity.

Although the methods employed in this thesis for robust collision checking rely on the uncertainty ellipsoid bounding box computed using Equation 3.8, the tubes are repres-

ented by ellipsoids in the various figures of this manuscript for smoother visualization.

Robust saturation checking Then, in this manuscript, the feasibility check is not restricted to the aforementioned collision test, but it also verifies that the robot control inputs does not saturate. This test is performed by checking that the tube associated with each control input remains in its feasibility domain. An example of infeasible input for the quadrotor case is presented in Figure A.2, where the tube (green) around the nominal control input of the first actuator (blue) exceeds the maximum allowed input (red). Note that this simple test is less costly than the robust collision checking one, it is therefore performed first by mean of computational efficiency.

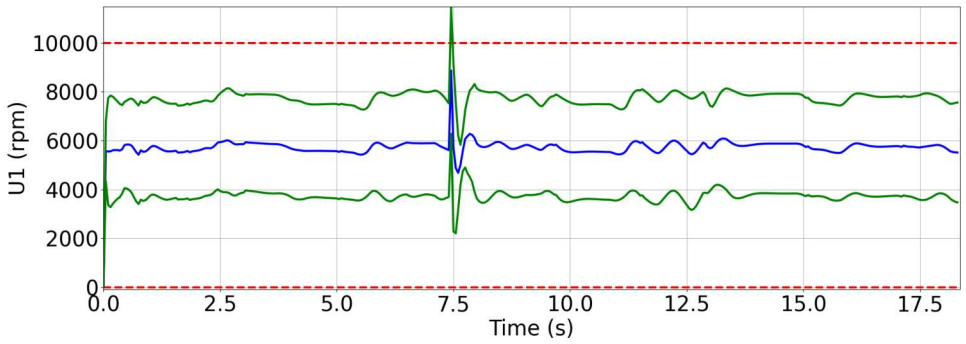


Figure A.2: Non-robust nominal control input profile for the first rotor of a quadrotor (blue) along a specified trajectory, depicted with its uncertainty tube (green) and the control input limits (red).

Appendix B
Learning curves

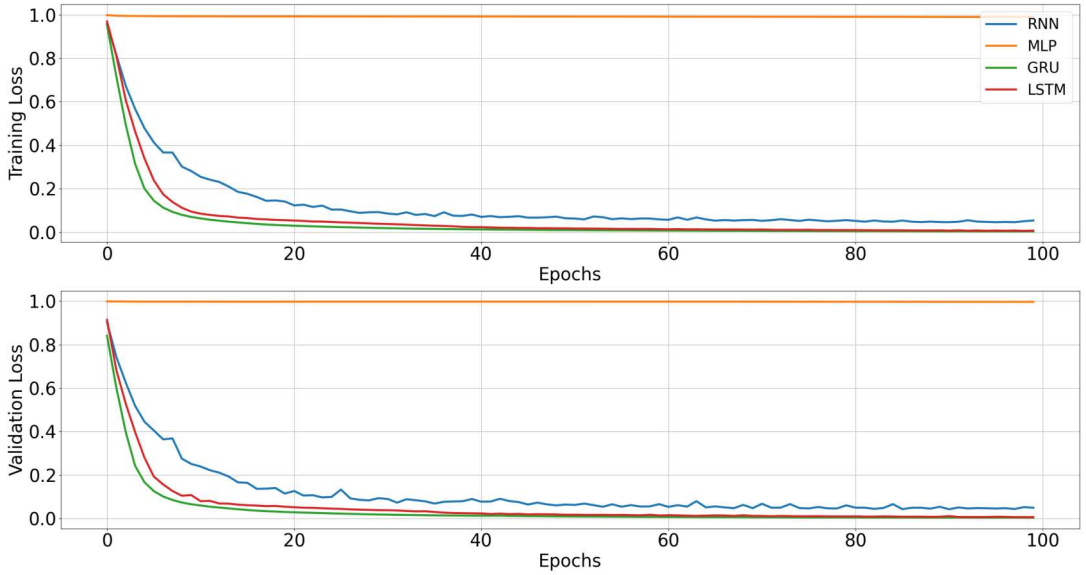


Figure B.1: Learning curves showing the evolution of training loss (top) and corresponding validation loss (bottom) for the differential drive robot application.

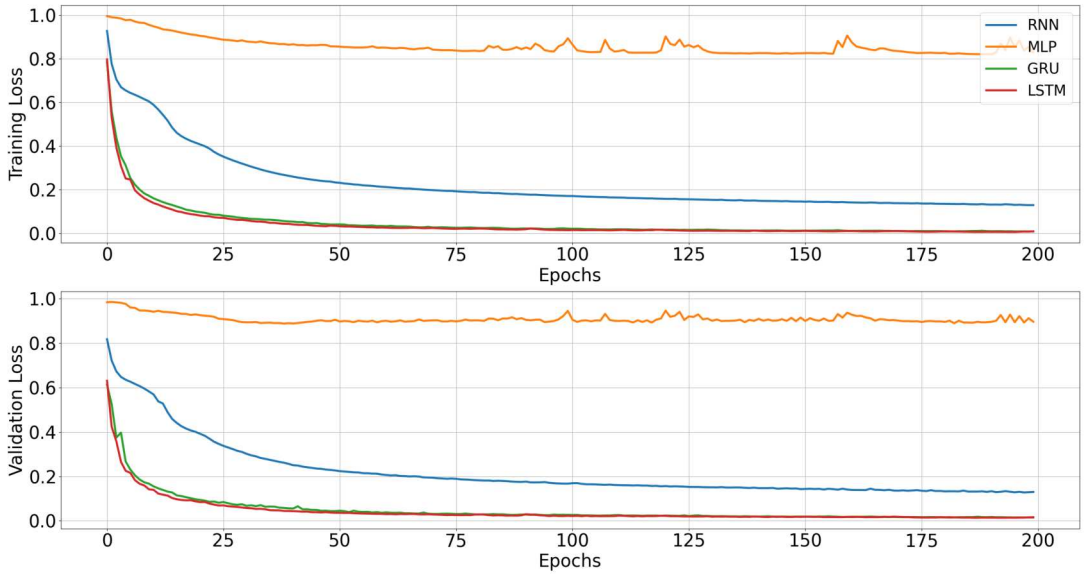


Figure B.2: Learning curves showing the evolution of training loss (top) and corresponding validation loss (bottom) for the quadrotor application.

Appendix C

ExtendedShortcut

This appendix provides grid search results to determine the optimal hyperparameters for the Sensitivity-Aware ExtendedShortcut (SAExtendedShortcut) algorithm.

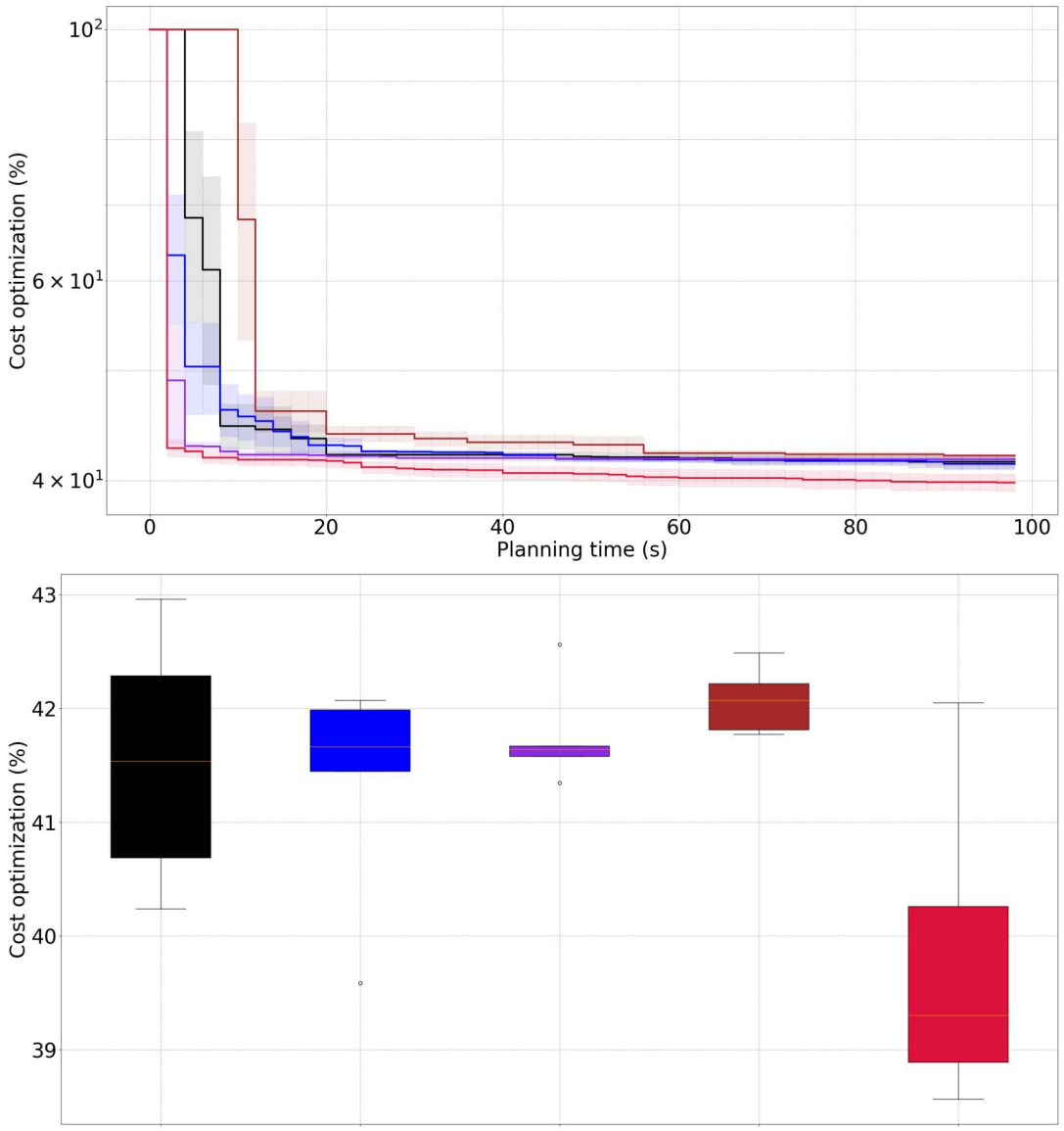


Figure C.1: Cost optimization results using a uniform sampling and a radius $\delta = 0.01$, averaged over 10 plans in a free space environment. The standard deviation is represented as an envelope around the mean curves. The red curve corresponds to $K = 1$, the purple curve to $K = 2$, the blue one to $K = 3$, the black to $K = 5$, and the brown curve to $K = 10$.

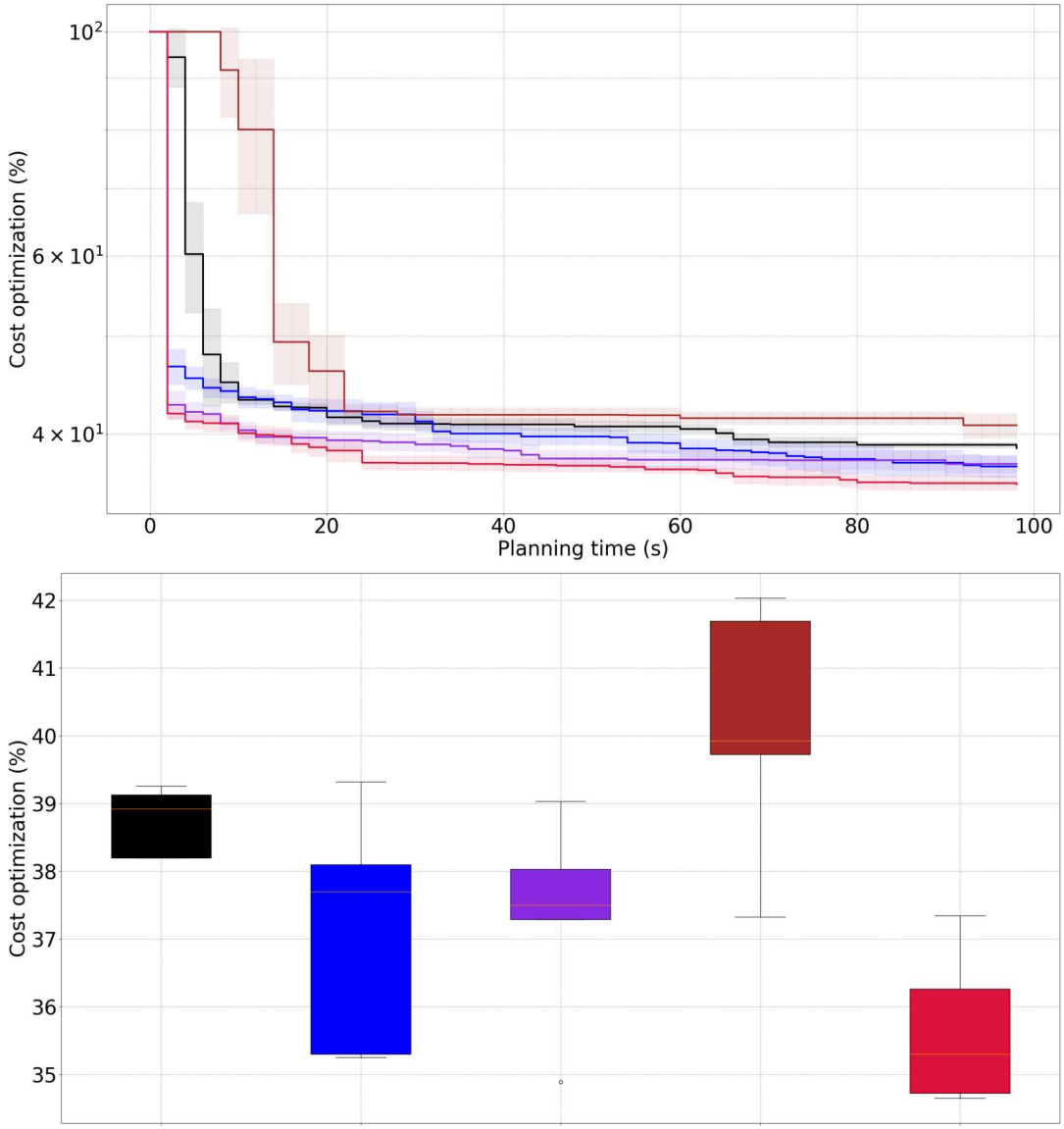


Figure C.2: Cost optimization results using a uniform sampling and a radius $\delta = 0.1$, averaged over 10 plans in a free space environment. The standard deviation is represented as an envelope around the mean curves. The red curve corresponds to $K = 1$, the purple curve to $K = 2$, the blue one to $K = 3$, the black to $K = 5$, and the brown curve to $K = 10$.

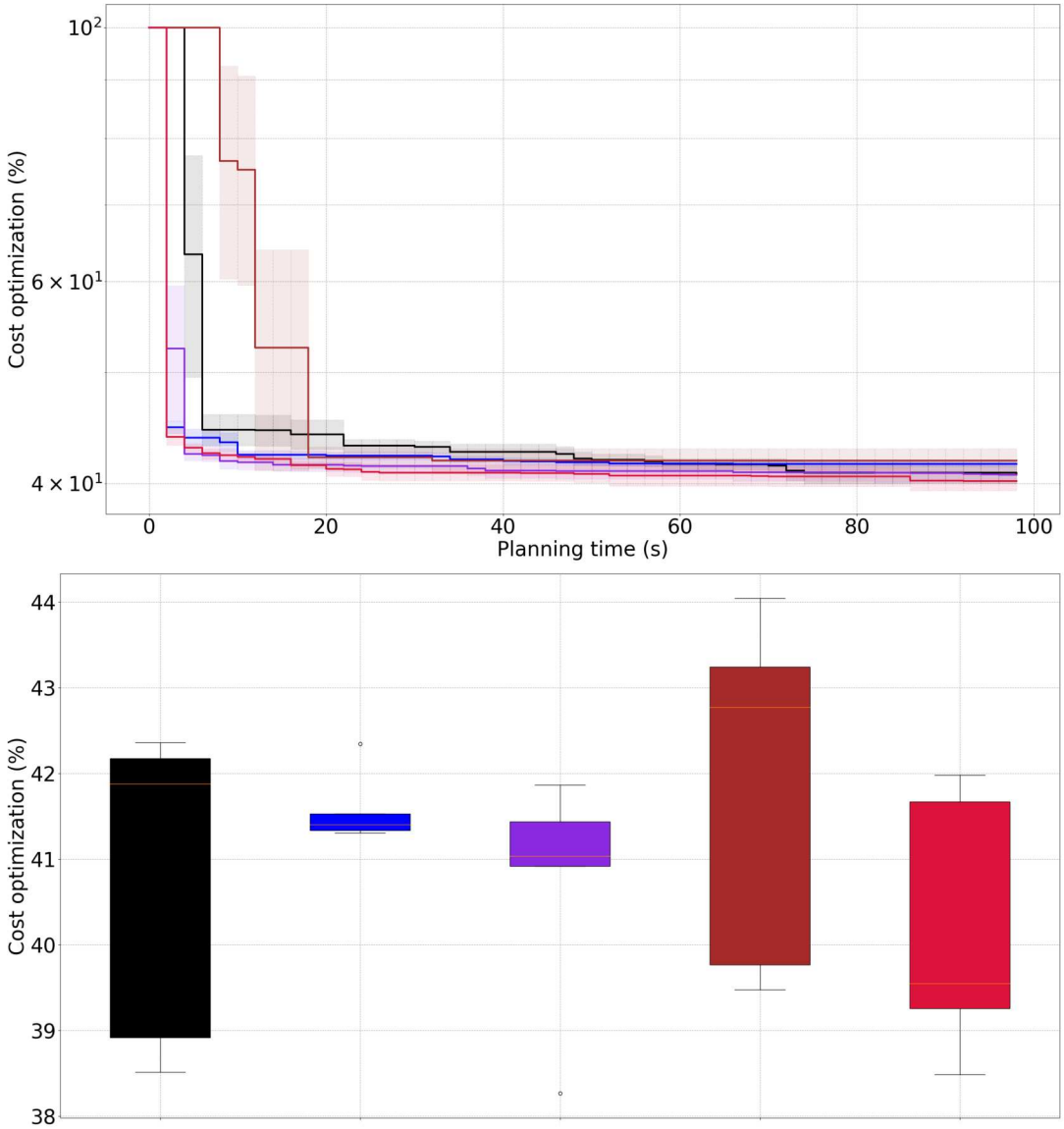


Figure C.3: Cost optimization results using a Gaussian sampling with a standard deviation $\delta = 0.01$, averaged over 10 plans in a free space environment. The standard deviation is represented as an envelope around the mean curves. The red curve corresponds to $K = 1$, the purple curve to $K = 2$, the blue one to $K = 3$, the black to $K = 5$, and the brown curve to $K = 10$.

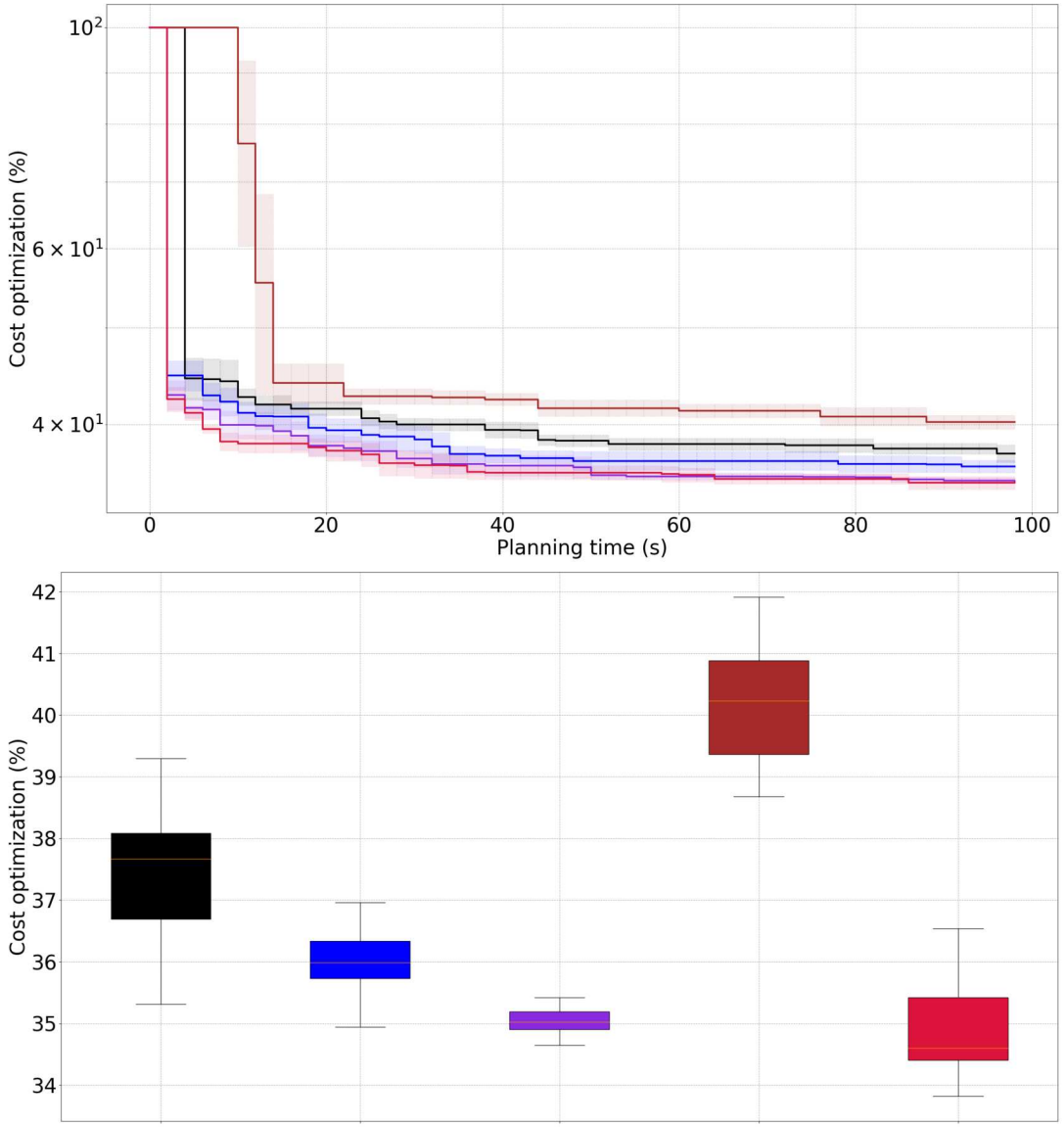


Figure C.4: Cost optimization results using a Gaussian sampling with a standard deviation $\delta = 0.1$, averaged over 10 plans in a free space environment. The standard deviation is represented as an envelope around the mean curves. The red curve corresponds to $K = 1$, the purple curve to $K = 2$, the blue one to $K = 3$, the black to $K = 5$, and the brown curve to $K = 10$.

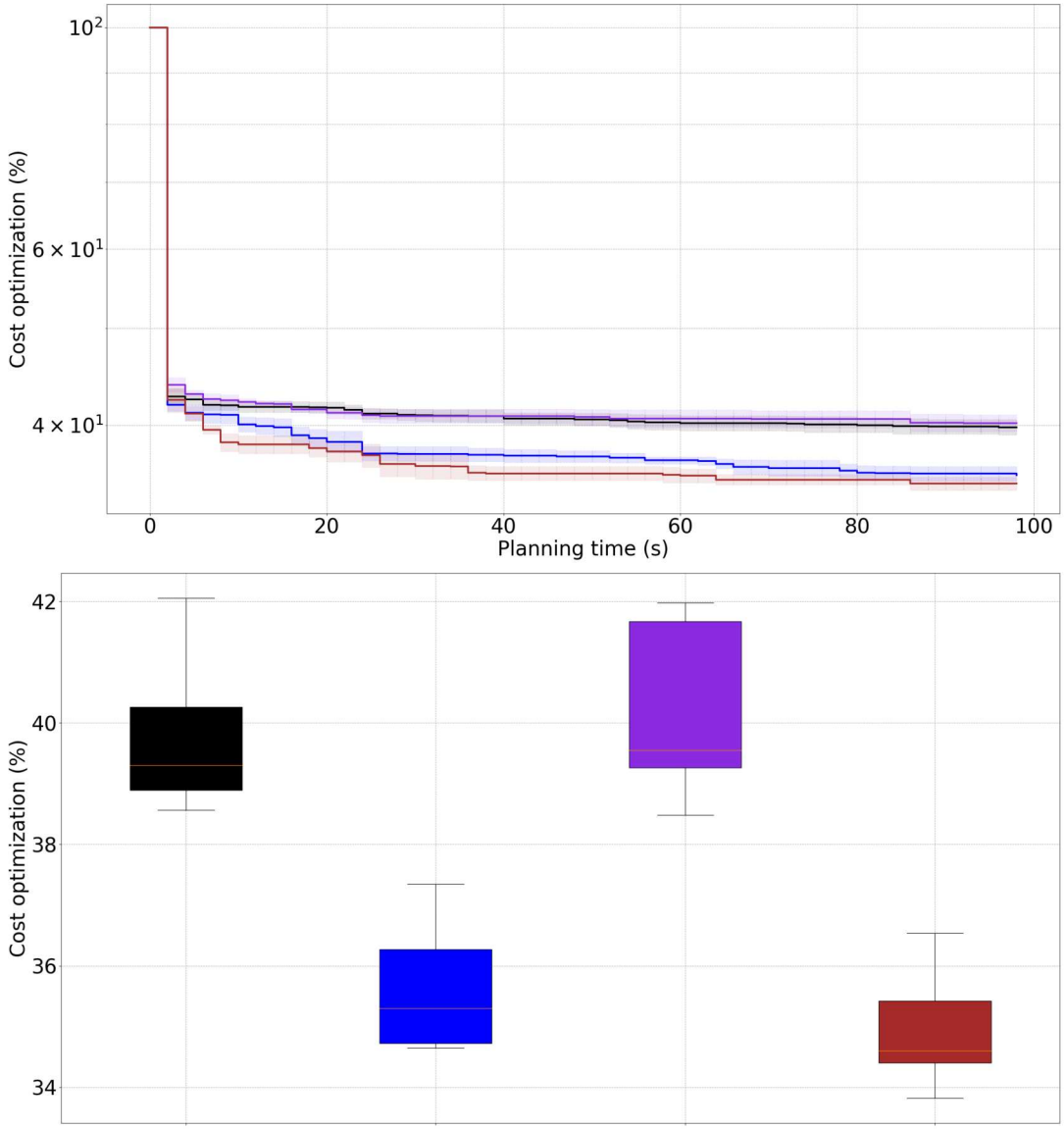


Figure C.5: Comparison of the best cases found in their respective classes: The red curve corresponds to a Gaussian sampling with $K = 1$ and $\delta = 0.1$, the blue curve is also associated to a Gaussian sampling with $K = 1$ and $\delta = 0.01$. The cases using uniform sampling are represented by the black and purple curves, both achieved with $K = 1$, and $\delta = 0.01$ and $\delta = 0.1$ respectively.

Appendix D

Résumé en Français

La planification du mouvement des robots est essentielle pour garantir un comportement robotique à la fois efficace et sûr. Cependant, les robots opérant dans des environnements réels font inévitablement face à des incertitudes, qu'il s'agisse de perturbations externes (e.g. le vent), d'inexactitudes dans les modèles, ou d'erreurs d'estimation d'état. Une approche efficace pour gérer la complexité d'évoluer and présences de ces incertitudes repose sur le paradigme de la "prévision/rétroaction" ou "planification/contrôle". Ce processus se déroule en deux étapes principales :

1. Phase de Planification (Prévision) : Une trajectoire de référence pour les états et commandes du robot est planifiée en se basant sur les informations disponibles, telles que les modèles du robot et de son environnement. Cette étape, généralement réalisée hors ligne, intègre des contraintes (e.g. évitement de collisions, limitations des actionneurs) et optimise des métriques comme la longueur de la trajectoire ou l'efficacité énergétique. Cependant, une exécution en boucle ouverte de cette trajectoire planifiée échoue souvent en pratique en raison des incertitudes qui affectent les références prévues.

2. Phase de Contrôle (Rétroaction) : Pour garantir une exécution robuste, un contrôleur de mouvement est employé pour fermer la boucle entre le mouvement planifié et le mouvement réel. Ce contrôleur compense les effets imprévus et les incertitudes qui n'ont pas été prises en compte lors de la planification.

Bien que cette approche séquentielle séparant planification et contrôle soit efficace, elle présente des limitations significatives :

- Les planificateurs modernes excellent à générer des trajectoires réalisables et globalement optimales pour des systèmes à haute dimension et des contraintes complexes. Cependant, ils ignorent généralement le rôle du contrôleur en temps réel, ce qui entraîne deux problèmes majeurs : (1) le contrôleur doit s'écarter de la trajectoire planifiée pour gérer les incertitudes et perturbations, compromettant ainsi rapidement la faisabilité et l'optimalité ; (2) le planificateur ne tient pas compte de la robustesse intrinsèque offerte par le contrôleur, manquant ainsi des opportunités pour produire des plans de mouvement plus robustes.

- De nombreuses méthodes de contrôle adaptatif ou robuste (par exemple, H-infini, méthodes LPV) ont été conçues pour gérer efficacement les incertitudes et perturbations, mais elles sont souvent locales vis à vis de la trajectoire de référence. Ces méthodes peinent à relever des défis plus larges comme la faisabilité sous contraintes, l'optimalité globale et les performances, qui sont mieux gérés par des approches de planification globale.

Pour combler l'écart entre ces deux communautés, plusieurs approches ont été introduites, notamment des contrôleurs plus globaux tels que le Model Predictive Control (MPC) [Lim+10]. De plus, la dernière décennie a vu émerger le concept de "planification de mouvement avec rétroaction" (feedback motion planning) [Her+17; Lew+22; MT17; Sin+17; Sin+21; Tog+18; Wu+22]. Cependant, ces méthodes continuent de rencontrer des défis en termes de généralisabilité, d'efficacité en temps de calcul et de dépendance à des modèles potentiellement inexacts de la pair robot/contrôleur en raison des incertitudes sur les paramètres du modèle.

Ainsi, ce travail, basé sur le paradigme de "planification de mouvement avec rétroaction" (ou "planification de mouvement consciente du contrôle"), exploite le concept de sensibilité en boucle fermée [BDR21; RDF18] (une extension de la notion de sensibilité qui incorpore le comportement du contrôleur vis-à-vis des incertitudes paramétriques) afin de créer des planificateurs robustes et conscients du contrôle pour une large classe de systèmes et de contrôleurs, tout en abordant les incertitudes dans leurs représentations modélisées.

Cette thèse débute par une revue de la littérature au Chapitre 2, qui fournit un aperçu de l'état de l'art. Elle se concentre d'abord sur les approches découplées pour la planification robuste de mouvement, en commençant par les méthodes de recherche de chemin et en progressant vers la planification kinodynamique de trajectoire associée à des stratégies de contrôle robuste. La revue se poursuit avec des approches unifiées, présentant des méthodes de planification de mouvement robustes prenant en compte les incertitudes, ainsi que des techniques de planification consciente du contrôle. Le chapitre se termine par une mise en avant des approches unifiées et robustes conscientes du contrôle, qui constituent le sujet principal de cette thèse.

Les chapitres suivants explorent en détail les contributions principales de cette thèse, depuis l'introduction des concepts de sensibilité en boucle fermée jusqu'à leur application pratique via des planificateurs conscients du contrôle optimisés par apprentissage profond, démontrant leur efficacité dans des scénarios exigeant robustesse et précision élevées.

Le chapitre 4 a présenté l'intégration du concept de sensibilité en boucle fermée et des tubes d'incertitude correspondants dans un cadre de planification global basé sur l'échantillonnage, répondant ainsi aux limitations des travaux antérieurs qui se concentraient principalement sur la génération locale de trajectoires. Cette méthodologie a permis de générer des trajectoires globalement optimales en termes de sensibilité. De plus, pour la première fois, les tubes d'incertitude basés sur la sensibilité ont été utilisés comme contraintes robustes dans le processus de planification, à la fois dans les espaces d'état et d'entrée de contrôle.

Bien que le calcul de ces tubes d'incertitude soit peu coûteux, la charge de calcul globale reste élevée en raison du grand nombre de calculs nécessaires dans les applications basées sur l'échantillonnage. Pour remédier à cela, une approche découplée s'appuyant sur une stratégie robuste et paresseuse a été explorée, réduisant considérablement les besoins en calcul, en particulier pour des systèmes de complexité croissante.

Dans cette continuité, le chapitre 5 a abordé davantage les défis computationnels en exploitant la similarité structurelle entre le système d'ODEs nécessaire au calcul des tubes et les Recurrent Neural Networks (RNNs). Le réseau de neuronne proposé a permis

d'établire une corrélation directe entre la trajectoire planifiée et les tubes d'incertitude, éliminant ainsi le besoin de résoudre les ODEs. La méthode a impliqué la génération d'un jeu de données basé sur une approche par échantillonnage adaptée à la tâche. En utilisant une cellule simple de type Gated Recurrent Unit (GRU), l'architecture proposée a atteint un compromis efficace entre la précision des prédictions et la vitesse d'inférence, réduisant le temps de calcul d'un ordre de grandeur par rapport aux solveurs d'ODEs traditionnels.

Dans le chapitre 6, cette architecture basée sur les GRU a été appliquée pour développer une alternative efficace à la stratégie ponctuelle de vérification robuste des collisions présentée dans le chapitre 4. Des variantes de planification conscient de la sensibilité basées sur l'apprentissage profond ont été proposées, démontrant leur efficacité pour générer des trajectoires robustes dans des temps de planification acceptables. De plus, alors que le chapitre 4 se concentrait principalement sur la minimisation globale de la sensibilité tout au long du mouvement, les méthodes de ce chapitre se sont orientées vers l'optimisation de la précision. Les tubes d'incertitude ont été appliqués de manière sélective à des segments spécifiques du mouvement en fonction des exigences de la tâche.

Une campagne de simulation approfondie a permis d'identifier le meilleur optimiseur local pour cette fonction de coût spécifique, en optimisant à la fois les trajectoires et les gains du contrôleur. Les méthodes proposées ont été validées expérimentalement dans deux scénarios exigeants: la navigation d'un quadrirotor à travers une fenêtre étroite et la capture en vol d'anneaux nécessitant une grande précision. Ces tests ont mis en évidence l'efficacité des méthodes proposés dans un environnement intérieur. Cependant, une validation supplémentaire est nécessaire pour évaluer son applicabilité à des systèmes plus complexes, tels que les manipulateurs aériens.

References

[ACA14] Ali-Akbar Agha-Mohammadi, Suman Chakravorty and Nancy M Amato. 'FIRM: Sampling-based feedback motion-planning under motion uncertainty and imperfect measurements'. In: *The International Journal of Robotics Research* 33.2 (2014), pp. 268–304 (cit. on p. 9).

[Afi+24] Amr Afifi, Tommaso Belvedere, Andrea Pupa, Paolo Robuffo Giordano and Antonio Franchi. 'Safe and Robust Planning for Uncertain Robots: A Closed-Loop State Sensitivity Approach'. In: *IEEE Robotics and automation letters* (2024) (cit. on pp. 16–18).

[AM13] Alex Ansari and Todd Murphey. 'Minimal parametric sensitivity trajectories for nonlinear systems'. In: *2013 American Control Conference*. IEEE. 2013, pp. 5011–5016 (cit. on pp. 10, 13).

[AM16] Alex Ansari and Todd Murphey. 'Minimum sensitivity control for planning with parametric and hybrid uncertainty'. In: *The International Journal of Robotics Research* 35.7 (2016), pp. 823–839 (cit. on pp. 10, 13).

[Ans18] Gerrit Ansmann. 'Efficiently and easily integrating differential equations with JiTCODE, JiTCDDE, and JiTCSDE'. In: vol. 28. 4. 2018 (cit. on pp. 35, 63).

[ASC12] Ibrahim Al-Bluwi, Thierry Siméon and Juan Cortés. 'Motion planning algorithms for molecular simulations: A survey'. In: *Computer Science Review* 6.4 (2012), pp. 125–143 (cit. on p. 6).

[BCS19] Alexandre Boeuf, Juan Cortés and Thierry Siméon. 'Motion planning'. In: Springer, 2019 (cit. on pp. 8, 24, 25, 40).

[BDR21] Pascal Brault, Quentin Delamare and Paolo Robuffo Giordano. 'Robust Trajectory Planning with Parametric Uncertainties'. In: *IEEE ICRA*. 2021 (cit. on pp. 2, 13, 16, 27, 34, 114).

[BK00] Robert Bohlin and Lydia E Kavraki. 'Path planning using lazy PRM'. In: *Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065)*. Vol. 1. IEEE. 2000, pp. 521–528 (cit. on pp. 6, 44).

[BL91] Jerome Barraquand and Jean-Claude Latombe. 'Robot motion planning: A distributed representation approach'. In: *The International Journal of Robotics Research* 10.6 (1991), pp. 628–649 (cit. on p. 5).

[Bre+13] Alexandru Brezoescu, Tadeo Espinoza, Pedro Castillo and Rogelio Lozano. 'Adaptive trajectory following for a fixed-wing UAV in presence of crosswind'. In: *Journal of Intelligent & Robotic Systems* 69 (2013), pp. 257–271 (cit. on p. 9).

[CA13] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control*. Springer science & business media, 2013 (cit. on p. 10).

[Can88] John Canny. *The complexity of robot motion planning*. MIT press, 1988 (cit. on p. 5).

[CB23] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. `http://pybullet.org`. 2016–2023 (cit. on pp. 35, 101).

[CFV22] Andrea Cristofaro, Marco Ferro and Marilena Vendittelli. 'Safe trajectory tracking using closed-form controllers based on control barrier functions'. In: *2022 IEEE 61st Conference on Decision and Control (CDC)*. 2022, pp. 3329–3334 (cit. on p. 9).

[CH10] S. Candido and S. Hutchinson. 'Minimum uncertainty robot path planning using a POMDP approach'. In: *IEEE IROS*. 2010 (cit. on p. 9).

[Che+18a] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt and David K Duvenaud. 'Neural ordinary differential equations'. In: *Advances in neural information processing systems* 31 (2018) (cit. on p. 55).

[Che+18b] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt and David K Duvenaud. 'Neural ordinary differential equations'. In: vol. 31. 2018 (cit. on p. 56).

[Cho+14] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk and Yoshua Bengio. 'Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014 (cit. on pp. 56, 75, 77).

[COB21] Glen Chou, Necmiye Ozay and Dmitry Berenson. 'Model error propagation via learned contraction metrics for safe feedback motion planning of unknown systems'. In: *IEEE CDC*. 2021 (cit. on p. 12).

[CZ18] Wenyu Cai and Meiyan Zhang. 'Smooth 3D dubins curves based mobile data gathering in sparse underwater sensor networks'. In: *Sensors* 18.7 (2018), p. 2105 (cit. on p. 21).

[DFM24] Kristofer Drozd, Roberto Furfaro and Daniele Mortari. 'Rapidly Exploring Random Trees with Physics-Informed Neural Networks for Constrained Energy-Optimal Rendezvous Problems'. In: *The Journal of the Astronautical Sciences* 71.1 (2024), p. 9 (cit. on p. 55).

[Dij59] Edsger. W. Dijkstra. 'A note on two problems in connexion with graphs.' In: *Numerische Mathematik* 1 (1959), pp. 269–271 (cit. on pp. 5, 6).

[DSC15] Didier Devaurs, Thierry Siméon and Juan Cortés. 'Optimal path planning in complex cost spaces with sampling-based algorithms'. In: *IEEE Transactions on Automation Science and Engineering* 13.2 (2015) (cit. on p. 6).

[Dub57] Lester E Dubins. 'On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents'. In: *American Journal of mathematics* 79.3 (1957), pp. 497–516 (cit. on pp. 6, 21).

[ETH18] Mohamed Elhoseny, Alaa Tharwat and Aboul Ella Hassanien. 'Bezier curve based path planning in a dynamic field using modified genetic algorithm'. In: *Journal of Computational Science* 25 (2018), pp. 339–350 (cit. on p. 7).

[FAT20] David D Fan, Ali-akbar Agha-mohammadi and Evangelos A Theodorou. 'Deep learning tubes for tube MPC'. In: 2020 (cit. on p. 55).

[Gaj+23] K Gajamannage, DI Jayathilake, Y Park and EM Bollt. 'Recurrent neural networks for dynamical systems: Applications to ordinary differential equations, collective motion, and hydrological modeling'. In: vol. 33. 1. AIP Publishing, 2023 (cit. on p. 56).

[GO07] Roland Geraerts and Mark H Overmars. 'Creating high-quality paths for motion planning'. In: *The international journal of robotics research* 26.8 (2007) (cit. on pp. 43, 45, 79).

[Hau15] Kris Hauser. 'Lazy collision checking in asymptotically-optimal motion planning'. In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 2951–2957 (cit. on p. 44).

[Her+17] Sylvia L Herbert, Mo Chen, SooJean Han, Somil Bansal, Jaime F Fisac and Claire J Tomlin. 'FaSTrack: A modular framework for fast and guaranteed safe motion planning'. In: *IEEE CDC*. 2017 (cit. on pp. 12, 114).

[HG08] Yifeng Huang and Kamal Gupta. 'RRT-SLAM for motion planning with motion and map uncertainty for robot exploration'. In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2008, pp. 1077–1082 (cit. on p. 10).

[HNR68] Peter E Hart, Nils J Nilsson and Bertram Raphael. 'A formal basis for the heuristic determination of minimum cost paths'. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107 (cit. on p. 6).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. 'Long short-term memory'. In: vol. 9. 8. MIT press, 1997 (cit. on pp. 56, 62).

[Jan+15] Lucas Janson, Edward Schmerling, Ashley Clark and Marco Pavone. 'Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions'. In: *The International journal of robotics research* 34.7 (2015) (cit. on p. 6).

[JCS10] Léonard Jaillet, Juan Cortés and Thierry Siméon. 'Sampling-based path planning on configuration-space costmaps'. In: *IEEE Transactions on Robotics* 26.4 (2010) (cit. on p. 6).

[JKV09] KG Jolly, R Sreerama Kumar and R Vijayakumar. 'A Bezier curve based path planning in a multi-agent robot soccer system without violating the acceleration limits'. In: *Robotics and Autonomous Systems* 57.1 (2009), pp. 23–33 (cit. on p. 7).

[Kal+11] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor and Stefan Schaal. 'STOMP: Stochastic trajectory optimization for motion planning'. In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 4569–4574 (cit. on pp. 7, 79).

[Kav+96] Lydia E Kavraki, Petr Svestka, J-C Latombe and Mark H Overmars. 'Probabilistic roadmaps for path planning in high-dimensional configuration spaces'. In: *IEEE Transactions on Robotics and Automation* 12.4 (1996) (cit. on pp. 5, 28).

[KB14] Diederik P Kingma and Jimmy Ba. 'Adam: A method for stochastic optimization'. In: 2014 (cit. on pp. 63, 71).

[KF11] Sertac Karaman and Emilio Frazzoli. 'Sampling-based algorithms for optimal motion planning'. In: vol. 30. 7. Sage Publications Sage UK: London, England, 2011 (cit. on pp. 6, 28, 30, 31, 34, 45, 77, 79).

[Kha86] Oussama Khatib. 'Real-time obstacle avoidance for manipulators and mobile robots'. In: *The international journal of robotics research* 5.1 (1986), pp. 90–98 (cit. on p. 5).

[KS98] S. Koenig and R. Simmons. 'A Robot Navigation Architecture Based on Partially Observable Markov Decision Process Models'. In: *Kortenkamp AI-ROBOTS* (1998) (cit. on p. 9).

[KW09] Torsten Kröger and Friedrich M Wahl. 'Online trajectory generation: Basic concepts for instantaneous reactions to unforeseen events'. In: *IEEE Transactions on Robotics* 26.1 (2009), pp. 94–111 (cit. on p. 6).

[LaV06] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006 (cit. on p. 5).

[LaV98] Steven LaValle. 'Rapidly-exploring random trees: A new tool for path planning'. In: Department of Computer Science, Iowa State University, 1998 (cit. on pp. 6, 28, 77, 79).

[LBD24] Jingyue Liu, Pablo Borja and Cosimo Della Santina. 'Physics-Informed Neural Networks to Model and Control Robots: A Theoretical and Experimental Investigation'. In: *Advanced Intelligent Systems* 6.5 (2024), p. 2300385 (cit. on p. 55).

[LBM21] Paul Lathrop, Beth Boardman and Sonia Martínez. 'Distributionally safe path planning: wasserstein safe RRT'. In: *IEEE Robotics and Automation Letters* 7.1 (2021), pp. 430–437 (cit. on p. 9).

[Lew+22] Thomas Lew, Lucas Janson, Riccardo Bonalli and Marco Pavone. 'A simple and efficient sampling-based algorithm for general reachability analysis'. In: *Learning for Dynamics and Control Conference*. PMLR. 2022 (cit. on pp. 13, 86, 114).

[Lia+21] Yuxuan Liang, Kun Ouyang, Hanshu Yan, Yiwei Wang, Zekun Tong and Roger Zimmermann. 'Modeling Trajectories with Neural Ordinary Differential Equations.' In: *IJCAI*. 2021 (cit. on p. 56).

[Lim+10] Daniel Limón, Ignacio Alvarado, TEFC Alamo and Eduardo F Camacho. 'Robust tube-based MPC for tracking of constrained linear systems with additive disturbances'. In: *Journal of Process Control* 20.3 (2010) (cit. on pp. 2, 11, 114).

[Lio01] J-L Lions. 'Résolution d'EDP par un schéma en temps «pararéel» A"parareal"in time discretization of PDE's'. In: *Academie des Sciences Paris Comptes Rendus Serie Sciences Mathematiques* 332.7 (2001), pp. 661–668 (cit. on pp. 53, 54).

[LK01] Steven M LaValle and James J Kuffner Jr. 'Randomized kinodynamic planning'. In: *The international journal of robotics research* 20.5 (2001), pp. 378–400 (cit. on p. 8).

[LKH10] Brandon Luders, Mangal Kothari and Jonathan How. 'Chance constrained RRT for probabilistic robustness to environmental uncertainty'. In: *AIAA guidance, navigation, and control conference*. 2010, p. 8160 (cit. on p. 9).

[LLB16] Yanbo Li, Zakary Littlefield and Kostas E Bekris. 'Asymptotically optimal sampling-based kinodynamic planning'. In: *The International Journal of Robotics Research* 35.5 (2016) (cit. on pp. 8, 29, 30, 32, 34, 36, 40, 43).

[LLM10] T. Lee, M. Leok and N. H. McClamroch. 'Geometric Tracking Control of a Quadrotor UAV on SE(3)'. In: *IEEE CDC*. 2010 (cit. on pp. 23, 24, 92).

[LMT84] Tomas Lozano-Perez, Matthew T Mason and Russell H Taylor. 'Automatic synthesis of fine-motion strategies for robots'. In: *The International Journal of Robotics Research* 3.1 (1984), pp. 3–24 (cit. on p. 9).

[Loz90] Tomas Lozano-Perez. *Spatial planning: A configuration space approach*. Springer, 1990 (cit. on p. 5).

[LW79] Tomás Lozano-Pérez and Michael A Wesley. 'An algorithm for planning collision-free paths among polyhedral obstacles'. In: *Communications of the ACM* 22.10 (1979), pp. 560–570 (cit. on p. 5).

[Mal+10] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan and Félix Ingrand. 'GenoM3: Building middleware-independent robotic components'. In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 4627–4632 (cit. on p. 93).

[MK11a] Daniel Mellinger and Vijay Kumar. 'Minimum snap trajectory generation and control for quadrotors'. In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 2520–2525 (cit. on p. 8).

[MK11b] Daniel Mellinger and Vijay Kumar. 'Minimum snap trajectory generation and control for quadrotors'. In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 2520–2525 (cit. on p. 23).

[MS07] Nik A Melchior and Reid Simmons. 'Particle RRT for path planning with uncertainty'. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE. 2007, pp. 1617–1624 (cit. on p. 10).

[MS12] Javad Mohammadpour and Carsten W Scherer. *Control of linear parameter varying systems with applications*. Springer Science & Business Media, 2012 (cit. on p. 9).

[MT17] Anirudha Majumdar and Russ Tedrake. 'Funnel libraries for real-time robust feedback motion planning'. In: *The International Journal of Robotics Research* 36.8 (2017) (cit. on pp. 2, 12, 114).

[Naw+24] Farhad Nawaz, Tianyu Li, Nikolai Matni and Nadia Figueroa. 'Learning Complex Motion Plans using Neural ODEs with Safety and Stability Guarantees'. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024, pp. 17216–17222 (cit. on p. 55).

[NS20] Ramya S Nair and P Supriya. 'Robotic path planning using recurrent neural networks'. In: *IEEE ICCCNT*. 2020 (cit. on p. 56).

[OCK23] Andreas Orthey, Constantinos Chamzas and Lydia E Kavraki. 'Sampling-based motion planning: A comparative review'. In: *Annual Review of Control, Robotics, and Autonomous Systems* 7 (2023) (cit. on p. 5).

[ODV02] Giuseppe Oriolo, Alessandro De Luca and Marilena Vendittelli. 'WMR control via dynamic feedback linearization: design, implementation, and experimental validation'. In: *IEEE Transactions on control systems technology* 10.6 (2002), pp. 835–852 (cit. on pp. 20, 21).

[Pad+16] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov and Emilio Frazzoli. 'A survey of motion planning and control techniques for self-driving urban vehicles'. In: *IEEE Transactions on intelligent vehicles* 1.1 (2016), pp. 33–55 (cit. on p. 5).

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai and Soumith Chintala. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on p. 64).

[Pow94] Michael JD Powell. *A direct search optimization method that models the objective and constraint functions by linear interpolation*. Springer, 1994 (cit. on p. 82).

[Rat+09] Nathan Ratliff, Matt Zucker, J Andrew Bagnell and Siddhartha Srinivasa. 'CHOMP: Gradient optimization techniques for efficient motion planning'. In: *2009 IEEE international conference on robotics and automation*. IEEE. 2009, pp. 489–494 (cit. on p. 7).

[RDF18] Paolo Robuffo Giordano, Quentin Delamare and Antonio Franchi. 'Trajectory generation for minimum closed-loop state sensitivity'. In: *IEEE ICRA*. 2018 (cit. on pp. 2, 13, 16, 27, 34, 114).

[RHW86] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. 'Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, ed. DE Rumelhart and J. McClelland. Vol. 1. 1986'. In: vol. 71. 1986 (cit. on pp. 56, 62).

[RPK19] Maziar Raissi, Paris Perdikaris and George E Karniadakis. 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations'. In: *Journal of Computational physics* 378 (2019), pp. 686–707 (cit. on p. 55).

[RS90a] James Reeds and Lawrence Shepp. 'Optimal paths for a car that goes both forwards and backwards'. In: *Pacific journal of mathematics* 145.2 (1990), pp. 367–393 (cit. on p. 6).

[RS90b] James Reeds and Lawrence Shepp. 'Optimal paths for a car that goes both forwards and backwards'. In: *Pacific journal of mathematics* 145.2 (1990), pp. 367–393 (cit. on p. 21).

[Sal+19] Paolo Salaris, Marco Cognetti, Riccardo Spica and Paolo Robuffo Giordano. 'Online optimal perception-aware trajectory generation'. In: *IEEE Transactions on Robotics* 35.6 (2019), pp. 1307–1322 (cit. on p. 98).

[SFR23] Ali Srour, Antonio Franchi and Paolo Robuffo Giordano. 'Controller and Trajectory Optimization for a Quadrotor UAV with Parametric Uncertainty'. In: *IEEE IROS*. 2023 (cit. on pp. 76, 91).

[Sin+17] Sumeet Singh, Anirudha Majumdar, Jean-Jacques Slotine and Marco Pavone. 'Robust online motion planning via contraction theory and convex optimization'. In: *IEEE ICRA*. 2017 (cit. on pp. 12, 114).

[Sin+21] Sumeet Singh, Hiroyasu Tsukamoto, Brett T Lopez, Soon-Jo Chung and Jean-Jacques Slotine. 'Safe motion planning with tubes and contraction metrics'. In: *IEEE CDC*. 2021 (cit. on pp. 2, 12, 114).

[SL96] P. Soueres and J.-P. Laumond. 'Shortest paths synthesis for a car-like robot'. In: *IEEE Transactions on Automatic Control* 41.5 (1996), pp. 672–688 (cit. on p. 21).

[ŞMK12] Ioan A. Şucan, Mark Moll and Lydia E. Kavraki. 'The Open Motion Planning Library'. In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012). `https://ompl.kavrakilab.org`, pp. 72–82 (cit. on p. 35).

[Sun+22] Wentao Sun, Nozomi Akashi, Yasuo Kuniyoshi and Kohei Nakajima. 'Physics-informed recurrent neural networks for soft pneumatic actuators'. In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 6862–6869 (cit. on p. 55).

[Ted+10] Russ Tedrake, Ian R Manchester, Mark Tobenkin and John W Roberts. 'LQR-trees: Feedback motion planning via sums-of-squares verification'. In: *The International Journal of Robotics Research* 29.8 (2010) (cit. on p. 10).

[Thr02] Sebastian Thrun. 'Probabilistic robotics'. In: *Communications of the ACM* 45.3 (2002), pp. 52–57 (cit. on p. 9).

[Tog+18] Marco Tognon, Elisabetta Cataldi, Hermes Amadeus Tello Chavez, Gianluca Antonelli, Juan Cortés and Antonio Franchi. 'Control-aware motion planning for task-constrained aerial manipulation'. In: *IEEE Robotics and Automation Letters* 3.3 (2018) (cit. on pp. 2, 11, 39, 42, 114).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez et al. 'Attention is all you need'. In: *Advances in neural information processing systems* 30.1 (2017) (cit. on p. 56).

[Vem+24] Sai H Vemprala, Rogerio Bonatti, Arthur Bucker and Ashish Kapoor. 'Chatgpt for robotics: Design principles and model abilities'. In: *IEEE Access* (2024) (cit. on p. 57).

[Was+23] Simon Wasiela, Paolo Robuffo Giordano, Juan Cortés and Thierry Simeon. 'A Sensitivity-Aware Motion Planner (SAMP) to Generate Intrinsically-Robust Trajectories'. In: *IEEE ICRA*. 2023 (cit. on pp. 3, 27).

[Was+24a] Simon Wasiela, Smail Ait Bouhsain, Marco Cognetti, Juan Cortés and Thierry Simeon. 'Learned Uncertainty Tubes via Recurrent Neural Networks for Planning Robust Robot Motions'. In: *27th European Conference on Artificial Intelligence (ECAI)*. 2024 (cit. on pp. 3, 53).

[Was+24b] Simon Wasiela, Marco Cognetti, Paolo Robuffo Giordano, Juan Cortés and Thierry Simeon. 'Robust Motion Planning with Accuracy Optimization based on Learned Sensitivity Metrics'. In: *IEEE Robotics and Automation Letters* (2024) (cit. on pp. 3, 75).

[Wu+22] Albert Wu, Thomas Lew, Kiril Solovey, Edward Schmerling and Marco Pavone. 'Robust-RRT: Probabilistically-Complete Motion Planning for Uncertain Nonlinear Systems'. In: *The International Symposium of Robotics Research*. Springer. 2022 (cit. on pp. 13, 85, 86, 114).

[Yan+23] Xingyu Yang, Yixiong Du, Leihui Li, Zhengxue Zhou and Xuping Zhang. 'Physics-Informed Neural Network for Model Prediction and Dynamics Parameter Identification of Collaborative Robot Joints'. In: *IEEE Robotics and Automation Letters* (2023) (cit. on p. 55).

[Yer+05] Anna Yershova, Léonard Jaillet, Thierry Siméon and Steven M LaValle. 'Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain'. In: *Proceedings of the 2005 IEEE international conference on robotics and automation*. IEEE. 2005, pp. 3856–3861 (cit. on p. 6).

[Zam81] G. Zames. 'Feedback and optimal sensitivity: Model reference transformations, multiplicative seminorms, and approximate inverses'. In: *IEEE Transactions on Automatic Control* 26.2 (1981), pp. 301–320 (cit. on p. 9).

[Zha+22] Pan Zhao, Arun Lakshmanan, Kasey Ackerman, Aditya Gahlawat, Marco Pavone and Naira Hovakimyan. 'Tube-certified trajectory tracking for nonlinear systems with robust control contraction metrics'. In: *IEEE Robotics and Automation Letters* 7.2 (2022) (cit. on p. 12).