Journal of Artificial Intelligence Research 75 (2022) 1-81 Submitted 09/2021; published 09/2022

# Motion Planning Under Uncertainty with Complex Agents and Environments via Hybrid Search

**Daniel Strawser** DSTRAWSE@MIT.EDU
*Massachusetts Institute of Technology*
*77 Massachusetts Avenue*
*Cambridge, MA 02139*

**Brian Williams** WILLIAMS@MIT.EDU
*Massachusetts Institute of Technology*
*77 Massachusetts Avenue*
*Cambridge, MA 02139*

## Abstract

As autonomous systems and robots are applied to more real world situations, they must reason about uncertainty when planning actions. Mission success oftentimes cannot be guaranteed and the planner must reason about the probability of failure. Unfortunately, computing a trajectory that satisfies mission goals while constraining the probability of failure is difficult because of the need to reason about complex, multidimensional probability distributions. Recent methods have seen success using chance-constrained, model-based planning. However, the majority of these methods can only handle simple environment and agent models. We argue that there are two main drawbacks of current approaches to goal-directed motion planning under uncertainty. First, current methods suffer from an inability to deal with expressive environment models such as 3D non-convex obstacles. Second, most planners rely on considerable simplifications when computing trajectory risk including approximating the agent's dynamics, geometry, and uncertainty. In this article, we apply hybrid search to the risk-bound, goal-directed planning problem. The hybrid search consists of a region planner and a trajectory planner. The region planner makes discrete choices by reasoning about geometric regions that the autonomous agent should visit in order to accomplish its mission. In formulating the region planner, we propose landmark regions that help produce obstacle-free paths. The region planner passes paths through the environment to a trajectory planner; the task of the trajectory planner is to optimize trajectories that respect the agent's dynamics and the user's desired risk of mission failure. We discuss three approaches to modeling trajectory risk: a CDF-based approach, a sampling-based collocation method, and an algorithm named Shooting Method Monte Carlo. These models allow computation of trajectory risk with more complex environments, agent dynamics, geometries, and models of uncertainty than past approaches. A variety of 2D and 3D test cases are presented including a linear case, a Dubins car model, and an underwater autonomous vehicle. The method is shown to outperform other methods in terms of speed and utility of the solution. Additionally, the models of trajectory risk are shown to better approximate risk in simulation.

## 1. Introduction

Motion planning under uncertainty is a challenging problem that is applicable to many intelligent systems. With the application of autonomous systems to more real-world environments, algorithms must deal with increasingly uncertain surroundings. Autonomous

cars must quickly make decisions while navigating urban environments. Autonomous underwater vehicles must accomplish a range of scientific and exploratory tasks while dealing with uncertain ocean currents.

Motion planning for general dynamical systems is a hard problem. Adding uncertainty makes the problem much more computationally-intensive. This is because the agent must reason about a large number of scenarios, which represent what *could* happen over the course of a mission. Mathematically, mission uncertainty is often represented as a probability distribution describing the agent's state or environment. In the context of motion planning, reasoning about uncertainty involves integrating the probability distribution to compute quantities such as the expected trajectory cost and the probability of collision. Because the integrand and the limits of integration are often complex, it is unlikely that there is an analytic solution to these integrals. Compounding the problem is the fact that the agent must not only reason about uncertainty, but must often do so quickly.

Consider the example of an autonomous underwater glider carrying out a science mission around Kolumbo, an undersea volcano off the coast of Greece. The mission calls for the glider to explore features of the seafloor such as underwater hydrothermal vents. While the seafloor geometry is known fairly well, there are uncertain ocean currents in the volcano's caldera, which can sweep it off course. Because recharging batteries is a time-consuming and labor-intensive operation, the glider must try to optimize its energy usage in order to remain operational as long as possible. The glider works in tandem with a team of scientists. The scientists dictate a set of regions they would like to explore and the glider plans its trajectory. The science team knows the mission is risky for the glider; they accept there is a likelihood they will lose some of their gliders to the scorching hydrothermal vents. However, they would like to keep their failures to less than 5% of the missions attempted.

This setting shares many characteristics of a risk-bound motion planning problem and poses a number of challenges to the motion planner. First, the glider's motion plan must be safe. While the science team has accepted that the mission is risky, they want to limit this risk to some predetermined mission failure rate. This bound on failure probability or risk, termed a probabilistic or chance constraint, gives the risk-bound motion planning problem its name. Second, the glider must accomplish a set of goals. For example, the glider might be tasked with exploring a region of the ocean where it is required to remain no more than a certain distance from the seafloor for observations. Third, in a typical use case, a team of scientists uses the motion planner as a tool to evaluate different motion plans. Therefore the motion planner should produce solutions quickly. Fourth, the glider should minimize energy usage to maximize uptime; therefore, optimizing a utility function is also important.

While this example is used to motivate the requirements for our method, the problem is very general and encountered in many areas of autonomous systems. The planner's requirements are summarized in the following four points:

- **Integrating risk-bound motion planning with complex environments**: Real world agents must interact with a large variety of obstacles. They must be able to quickly reason about complex, three-dimensional, non-convex obstacles and produce trajectories that respect risk-bounds with respect to obstacle collision.

- **Reasoning about complex environment-constrained goals**: Agents are often required to accomplish complicated goals that constrain their trajectories in relation to their environment.
- **Meet performance metrics in terms of computation time and trajectory utility**: The planner must not only model stochasticity but oftentimes produce solutions very quickly such as for real time systems. Additionally, many systems have cost metrics that should be minimized such as energy usage.
- **Modeling risk for non-Gaussian stochasticity & non-trivial agent geometry**: A common approach to risk-bound motion planning is to simplify the problem by assuming the agent is a point robot and that all uncertainty is Gaussian. These assumptions often do not hold in real world applications. Particularly in the case of nonlinear dynamical systems, state distributions are non-Gaussian and the agent's geometry must be considered when computing trajectory risk.

### 1.1 Overview of the Approach

Our approach is to solve the risk-bound motion planning problem using hybrid search. The search consists of a region planner and a trajectory planner. These components are introduced in the following subsections; a formal problem statement is given in Section 3.

#### 1.1.1 The region planner

The purpose of the region planner is to determine an appropriate set of active geometric constraints that allow the agent to accomplish its mission. To do this, the region planner reasons about geometric regions of the agent workspace, termed *regions* throughout this article. The regions are used to model obstacle-avoidance constraints, goal constraints, and other geometric constraints on the agent's trajectory. The region planner connects sequences of regions into paths. Each path can be compiled into a set of constraints on the agent's continuous state trajectory.

The region planner receives as inputs an agent and a sequence of goals that the agent must accomplish. The agent is described by dynamical equations of motion and its geometry. A set of obstacles is also input, which the agent must avoid with some probability. The output of the region planner is an ordered set of regions, a *path*, that represents a sequence of geometric constraints on the agent's position. This path is input to the trajectory planner.

#### 1.1.2 The trajectory planner

The task of the trajectory planner is to optimize a trajectory that respects the path's implied geometric constraints as well as other constraints such as the agent's dynamics. Importantly, the trajectory planner models trajectory risk. In this article, we describe three models of risk. The trajectory planner uses techniques from trajectory optimization and off-the-shelf optimization libraries.

The trajectory planner receives as inputs a sequence of region constraints from the region planner. It requires a model of the agent's dynamics and a predetermined bound on mission failure, i.e. the chance constraint. The output of the trajectory planner is a trajectory that satisfies the geometric path constraints, dynamics constraints, and chance constraint.

## 2. Related Work

In spite of the complexity of motion planning under uncertainty, its real world applicability has driven many advancements in recent years. The relevant approaches are summarized into three subsections: motion planning approaches relying on hybrid search, approaches used to model the chance constraint, and methods for workspace decomposition.

### 2.1 Hybrid Search for Motion Planning

The core of our approach to solving the risk-bound motion planning problem is a hybrid search. Hybrid search is a technique that has seen interest in recent years to solve problems with discrete and continuous decision variables, such as combined task and motion planning problems.

An early example of a hybrid planner is Kongming, which searches over discrete and continuous decision variables to solve the goal-directed planning problem (Li, 2010). Kongming introduces a Hybrid Flow Graph, which is similar to our Region Graph; the Hybrid Flow Graph uses flow tubes to describe continuous actions. Kongming shows that hybrid search can be effective in planning for real world scenarios including an underwater vehicle. Unlike the present work, Kongming does not reason about risk and the focus is on planning for simple linear systems in 2D environments. Innovative work presents ScottyActivity, an approach that combines activity and trajectory planning over long horizons (Fernandez-Gonzalez, Williams, & Karpas, 2018). The Scotty planner decomposes the workspace into convex subregions and performs graph search over the subregions. The insight underlying the Scotty trajectory planner is that, assuming linear equations of motion, fast convex optimization routines can be used by the trajectory planner and trajectories over long time horizons generated. The Scotty planner shows the promise of hybrid search; however, this article distinguishes itself in a number of ways. First, Scotty focuses on deterministic problems rather than those with uncertainty. Secondly, while Scotty relies on agents with linearized dynamics and environments with 2D convex obstacles, our work relaxes these assumptions to allow for a wider variety of agents and environments.

Another example of hybrid search is the pSulu planner (Ono, Williams, & Blackmore, 2013). The pSulu planner solves a Chance-Constrained Qualitative State Plan (CCQSP) using a form of hybrid search. Unlike Scotty, pSulu is focused on solving risk-bound path planning problems. Rather than decompose the workspace via convex subregions, pSulu models the motion planning problem with obstacles as a disjunctive linear program. Determining how an agent avoids an obstacle reduces to making assignments to a disjunctive clause. The resulting hybrid search is solved using a branch and bound scheme and convex optimization. While our work solves a similar problem as pSulu, it expands on pSulu's capabilities in various ways. First, as the authors note, pSulu's disjunctive program scales poorly to cases with complex environments. Secondly, pSulu's computation of trajectory risk is limited to cases for which there is an easily-evaluated cumulative distribution function, such as the Gaussian case.

Other approaches have proposed hybrid search to solve motion planning problems with complex goals expressible in Linear Temporal Logic (LTL) (Bhatia, Kavraki, & Vardi, 2010), (Fainekos, Kress-Gazit, & Pappas, 2005), (Vasile & Belta, 2014). These are similar to our present method by relying on a high level planner that reasons about discrete choices

in combination with a lower level planner that considers models of the agent's dynamics. Work in (Bhatia, Maly, Kavraki, & Vardi, 2011) is representative of these approaches and contains themes similar to our work. The authors propose a geometry-based abstraction of the robot workspace and show that it outperforms a geometry-ignoring abstraction. The authors also discuss a lazy high level search that allows for considerable computational speed-up. Unlike our proposed method, these methods are not focused on planning under uncertainty. Furthermore, the geometric discretizations that allow the discrete planner to reason about continuous workspaces tend to be applicable to simple 2D environments, e.g. triangulations of polygonal workspaces.

A large number of other approaches to the motion planning problem combine aspects of hybrid search. Work in (Plaku, Kavraki, & Vardi, 2009) uses hybrid search with nonlinear systems to quickly determine unsafe states. In this work, hybrid search is necessitated by the system's dynamics; systems are considered which are described by continuous and discrete dynamics. Learning has seen success in a hybrid approach to motion planning. In (Faust, Ramirez, Fiser, Oslund, Francis, Davidson, & Tapia, 2018), a probabilistic roadmap (PRM) planner is combined with reinforcement learning to generate long trajectories for nonlinear systems.

### 2.2 Models of the Chance Constraint

Many techniques have been developed in literature to simplify the problem of motion planning under uncertainty. A good overview of various techniques for planning under uncertainty is described in (Prékopa, 1995).

A common method of approximating trajectory risk is to assume a state distribution for which there is a quick-to-evaluate cumulative distribution function (CDF), such as the Gaussian distribution. This is the approach taken in (Blackmore, Ono, & Williams, 2011) where the authors combine a disjunctive program with a CDF-based evaluation of the chance constraint. Another approach does not directly assume Gaussian uncertainty but models the state distribution of an agent that collides with an obstacle as a truncated Gaussian (Patil, van den Berg, & Alterovitz, 2012). This can still be considered a CDF-based method as computing the cumulative distribution of the truncated Gaussian requires the ability to compute the Gaussian CDF. Finally, an extension of the popular rapidly-exploring random tree (RRT) algorithm is proposed in (Luders, Kothari, & How, 2010). Chance constraints are added to the basic RRT formulation; it is not only necessary that a path avoids collisions but also that the path is feasible with respect to the chance constraint. To evaluate the risk, the method deploys an approximation similar to (Blackmore et al., 2011). Another sampling-based motion planner is presented in (van den Berg, Abbeel, & Goldberg, 2011) where the focus is on linear systems with partial observability. Similar to other approaches presented in this paragraph, system noise is assumed Gaussian.

A limitation of the methods that rely on cumulative distribution functions is the availability of easy-to-evaluate CDFs. Because of this, sampling-based Monte Carlo methods have also seen wide interest. An early work in sampling-based trajectory risk approximation is provided in (Blackmore, Ono, Bektassov, & Williams, 2010). This work samples from a distribution that represents the agent's state uncertainty and uses an indicator variable to signal whether a sample is in collision or not. Because of its use of indicator decision

variables, this work requires a mixed integer linear program (MILP) to compute a trajectory. Furthermore, because there is a one-to-one correspondence between indicator variables and samples, accurately modeling the collision risk can generate a very large optimization problem. One advantage of sampling-based techniques is that they can readily take advantage of trends in hardware development, specifically using the parallelization provided by graphics processing units (GPUs). This idea is explored in (Ichter, Schmerling, Aghamohammadi, & Pavone, 2017) where a probabilistic roadmap (PRM) is built and the risk is evaluated by collision checking trajectory scenarios. While sampling-based methods can approximate complex stochastic processes, a downside is their convergence in approximating the underlying uncertainty. There are a number of variance-reduction methods, one of which is discussed in (Janson, Schmerling, & Pavone, 2018). In this work, the method of control variates and importance sampling is used to reduce the variance in Monte Carlo computations of the collision probability.

In spite of the prevalence of CDF and sampling-based methods, other innovative approaches to the risk-bound motion planning problem have also been discussed in literature. A method is discussed in (Jasour, Hofmann, & Williams, 2018) where the authors model complex distributions via a sequence of moments. While the method can model complex probability distributions of the obstacles' surface, it is unclear how the method scales to large 3D environments.

### 2.3 Workspace Decompositions

The central idea behind this work's region planner is to decompose the agent workspace into regions and then use graph search to determine good paths or sequences of regions. This section reviews select strategies in literature to generate partitions of the workspace.

One of the most popular approaches is to discretize the workspace into a grid, as shown in Fig. 1, Pane A (LaValle, 2006). There are many variations of this idea and techniques to generate the discretization. A common method is to use the discretization to produce an occupancy map where each cell relays information about its contents, i.e. whether or not it is obstacle-free (Elfes, 1989). Additionally, many discretization schemes are possible with uniform being the simplest to generate and others attempting to discretize the robot configuration space (Ziegler & Stiller, 2009). The authors in (Hrabar, 2008) show it is possible to extend the method to 3D workspaces and present the innovative idea of using different map resolutions for modeling the environment and for planning. These methods can be readily applied to kinodynamic motion planning with complex dynamics as shown in (Sucan, Ioan A. and Kavraki, Lydia E., 2010). In this work, the focus is on efficiently generating cells to ensure the agent state space is explored and that cells are generated on an as-needed basis. Because the partitioning of the workspace is typically complete, searches that utilize this strategy can often produce globally optimal solutions (LaValle, 2006). In general, grid-based approaches have the downside of choosing an appropriate scale for the discretization. It may be very difficult to determine a priori how many discretizations are required for a given problem.

Another set of techniques are similar to grid-base occupancy maps but decompose the workspace using geometric features such as obstacle vertices. For example, the Polyanya algorithm performs any-angle path planning on a 2D map that relies on a partitioning

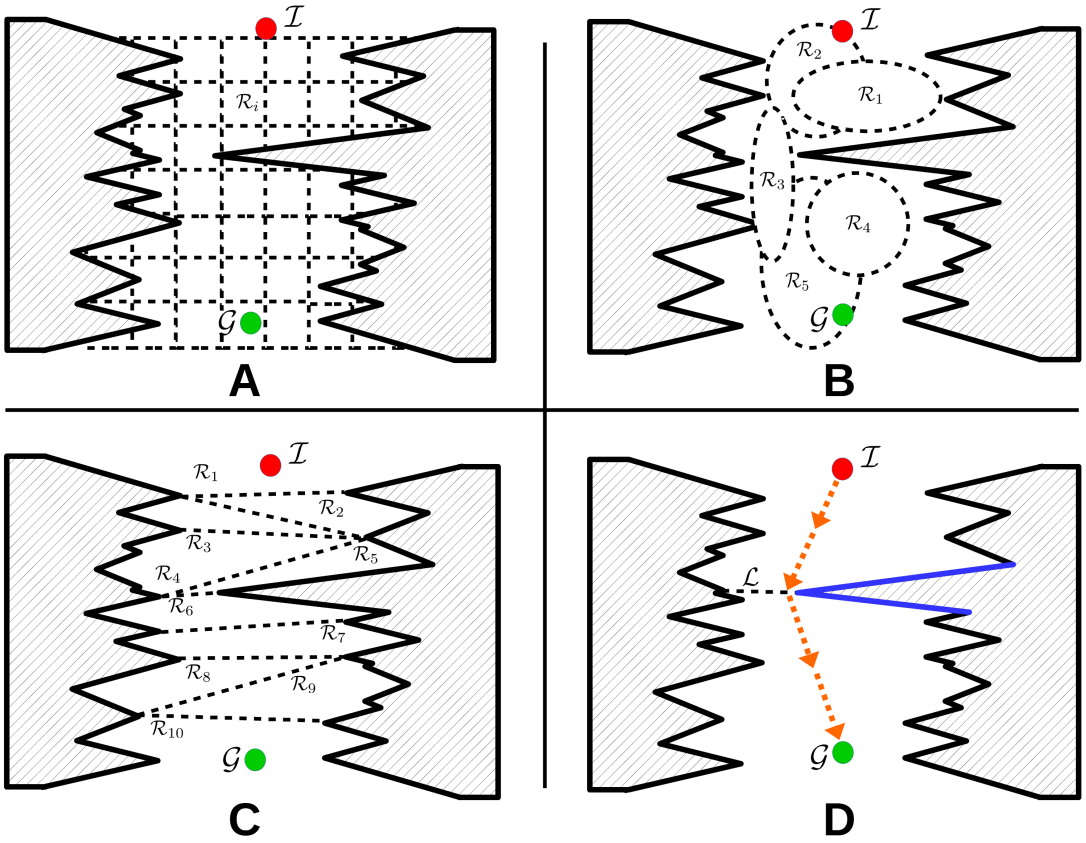



Figure 1: An illustration of various methods to partition the workspace. Assume an agent must travel from $\mathcal{I}$ to $\mathcal{G}$ and avoid the shaded obstacles on the left and right. Pane A illustrates a uniform grid-based discretization, with an individual cell labeled $\mathcal{R}_i$. A challenge of this approach is that the cells may be a poor approximation of the obstacle geometry. Therefore, some approximation scheme is typically necessary where information about the cell, such as whether it contains an obstacle, must be averaged over the entire cell. Pane B illustrates methods that attempt to generate convex regions from the free region of the workspace, $\mathcal{W}_{free}$. These approaches have the benefit that paths traveling within a convex region are guaranteed to be collision free (Gonzalez, 2018; Deits & Tedrake, 2015). For example, an agent whose trajectory lies in convex regions $\mathcal{R}_1$ - $\mathcal{R}_5$ in Pane B will avoid obstacles. This has the drawback that a separate method is required to connect the regions and it can be difficult to determine if enough regions have been generated to connect $\mathcal{I}$ and $\mathcal{G}$. Pane C illustrates methods that rely on geometric algorithms such as triangulations or trapezoidal decompositions. Unlike the grid-based methods, these approaches rely on the obstacles' geometry to generate regions. However, they are typically uninformed with respect to the underlying motion planning problem and can generate unnecessary regions. Pane D illustrates the strategy taken in this work. Our approach attempts to only reason about obstacle geometry relevant to the task of navigating from $\mathcal{I}$ to $\mathcal{G}$, such as the blue obstacle facets in Pane D. Using these facets, we form geometric constraints that the trajectory must satisfy, depicted here as the ray $\mathcal{L}$ emanating from the vertex connecting the two blue facets incident to $\mathcal{L}$. The resulting trajectory is depicted in orange.

of the workspace via a Constrained Delaunay Triangulation. The algorithm is shown to deliver optimal solutions quickly (Cui, Harabor, & Grastien, 2017). The trapezoidal map algorithm decomposes 2D workspaces into sets of trapezoids, which can then be used to solve the path planning problem (Berg, Cheong, Kreveld, & Overmars, 2008). The algorithm creates the map by drawing vertical lines from every obstacle vertex. An advantage of these methods is that the algorithms to generate the decompositions are typically efficient. The trapezoidal decomposition and Delaunay triangulation can be computed in $\mathcal{O}(n \log n)$ time (Berg et al., 2008). One downside of these strategies is that, similar to grid partitionings, the partitioning is typically done agnostic of the search problem. This means that they can generate unnecessarily complex partitionings. This is an undesirable property for hybrid search because the complexity is exponential in the depth of the search tree. The problem of unnecessarily-generated regions is illustrated in Fig. 1, Pane C.

An innovative scheme that relies on semi-definite programming (SDP) to create convex regions is proposed in (Deits & Tedrake, 2015); this algorithm is named Iterative Regional Inflation by Semidefinite Programming, IRIS. IRIS is integrated into a hybrid search for multi-agent path planning in (Gonzalez, 2018). ScottyPath is proposed, which combines a region planner that selects sequences of convex regions with a trajectory planner that optimizes a trajectory constrained to lie inside those regions. To generate convex regions from the obstacle free region of the workspace, ScottyPath relies on IRIS. In the case of convex obstacle fields, IRIS is fast because it can take advantage of efficient SDP solvers. ScottyPath then connects the regions together to form a graph. There are two main drawbacks to the ScottyPath / IRIS approach. First, it requires computation of the connectivity graph a priori. Second, IRIS generates convex regions by sampling from the free region of the workspace; it is difficult to determine a priori whether the number of samples is sufficient to generate a connectivity graph that allows for a path from the initial state to the goal state. Rather than requiring them to be explicitly computed, the regions presented in our present work are implicitly defined given an obstacle surface.

### 2.4 Summary of Contributions

To the best of our knowledge, our proposed method is the first risk-bound, goal-directed motion planning algorithm that enables hybrid search for agents with non-trivial geometry, nonlinear dynamics, and non-Gaussian stochasticity.

The region planner's contributions focus on enabling fast hybrid search in complex environments. We propose:

1. A type of obstacle-avoidance constraint, termed *landmark region*, that is fast to generate, allows missions with complex polytope obstacle geometries, and provides a geometric simplification for modeling risk (Section 4.1)

2. A hybrid search strategy First Feasible Hybrid Search that generates feasible solutions quickly for complex agents and environments by attempting to reduce expensive trajectory optimization calls (Section 5)

The trajectory planner's contributions focus on three chance constraint models of increasing complexity:

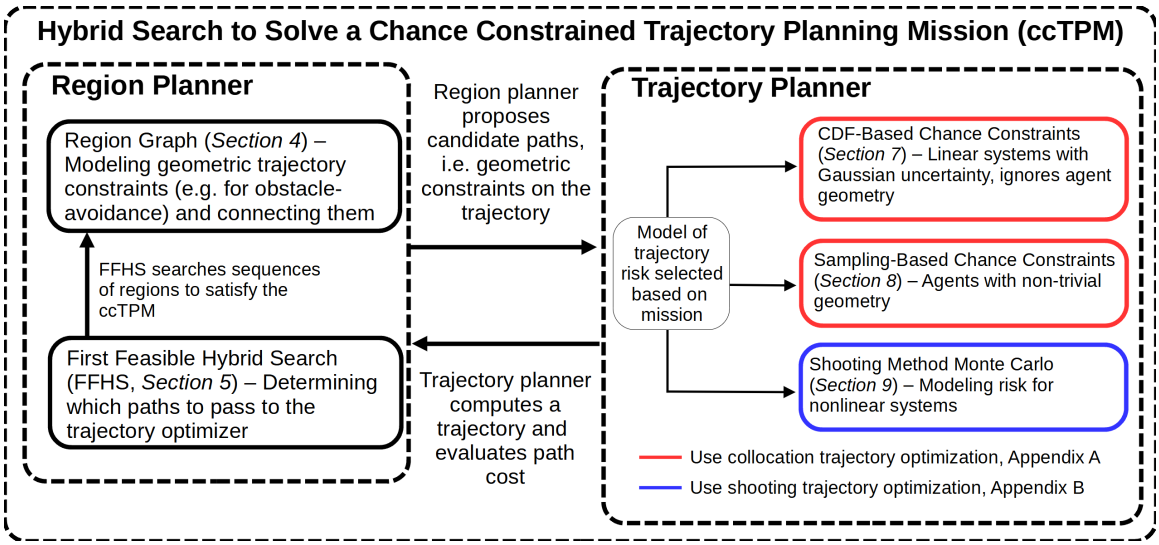



Figure 2: Overview of the hybrid search components and their corresponding sections

1. A model that combines CDF-based chance constraints and landmark regions to provide a fast, coarse risk approximation (Section 7)

2. A model of risk for agents whose geometry is important when calculating the chance constraint; our insight is to combine sampling-based collision checking with trajectory optimization (Section 8)

3. Shooting Method Monte Carlo whose insight is to combine a Monte Carlo method with the shooting method of trajectory optimization to model non-Gaussian stochasticity in nonlinear dynamical systems (Section 9)

### 2.5 Organization

This article is organized around the components of the hybrid search, first explaining the region planner and then the trajectory planner. Section 3 defines the problem statement and provides an overview of the notation associated with the hybrid search. A formalism is introduced: the chance-constrained trajectory planning mission. Section 4 describes the region planner in detail. A key idea in this section is our proposed *landmark region*, a type of obstacle-avoidance constraint. Section 5 introduces First Feasible Hybrid Search, which is a state space search used by the region planner for finding good paths through the workspace. The second half of the paper focuses on the trajectory planner and three models of risk. Section 6 introduces the trajectory planner. Section 7 describes how a CDF-based model may be integrated with the region planner's landmark regions as a coarse model of risk. Section 8 introduces a sampling-based model of the chance constraint that is appropriate for agents whose geometry must be considered when computing trajectory risk. Third, Section 9 discusses Shooting Method Monte Carlo, a risk model appropriate for nonlinear, non-Gaussian systems. Finally, the paper concludes with a number of test cases and benchmarks in Sections 10.2 through 10.5. While the trajectory planner's three models of risk are the

focus of the article, Appendices A and B provide details of the underlying collocation and shooting trajectory optimizations used by the trajectory planner. An overview of the architecture and how components interact is shown in Fig. 2.

## 3. Problem Statement

We consider the problem of an agent navigating from an initial state $\mathcal{I}$ to a sequence of goal states $\boldsymbol{\mathcal{G}}$ while avoiding obstacles with probability $1-\epsilon$. The purpose of the algorithm is to output an open loop control trajectory $\mathbf{u}$. The control outputs $\mathbf{u}$ represent inputs to the agent's actuators. Throughout this work we use the term *agent* to denote a general autonomous system.

The navigation problem stated above can be framed as the following optimal control problem:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{u},\mathbf{h}}{\text{minimize}} \quad & \mathbb{E}\left[J\left(\mathbf{x}_{0:K}, \mathbf{u}_{0:K-1}, \mathbf{h}_{0:K-1}\right)\right] && (1)\\
\text{subject to} \quad & \dot{\mathbf{x}} = \mathbf{f}\left(\mathbf{x}, \mathbf{u}\right) + \mathbf{w} && (2)\\
& P\left(\bigwedge_{k \in K} \mathbf{x}_k \in \mathcal{W}_{free}\right) \geq 1-\epsilon && (3)\\
& \mathbf{x}_0 \in \mathcal{I} && (4)\\
& \mathbf{x}_{k_g} \in \mathcal{G}_g \;\; \forall \;\; 0 \leq g \leq G && (5)
\end{aligned}$$

Term (1) is the objective function; it requires minimization of the expected trajectory cost. There are a wide number of forms the objective can take from minimizing mission time to minimizing fuel consumption or path distance. Constraint (2) represents the agent's dynamics. It is written generally because there are many possible numerical integration / discretization schemes when performing a trajectory optimization (e.g. Euler, trapezoidal). The left-hand side is the time derivative of the agent's state $\mathbf{x}$. The right-hand side consists of two terms. The term $\mathbf{f}\left(\mathbf{x}, \mathbf{u}\right)$ represents the agent's deterministic equations of motion. The second term, $\mathbf{w}$, represents the problem's source of uncertainty; $\mathbf{w}$ is a vector of continuous random variables. A simple case is zero mean additive Gaussian noise where $\mathbf{w} \sim \mathcal{N}\left(0, \Sigma\right)$. While it is possible to consider other sources of uncertainty, e.g. environmental uncertainty due to incomplete mapping data, this article focuses on uncertainty in the agent's state generated by various forms of $\mathbf{w}$ in Eq. (2).

Constraint (3) is the probabilistic or chance constraint which ensures that the probability of obstacle collision is at most $\epsilon$. The constraint is challenging because of the need to integrate the agent's state probability distribution over $\mathcal{W}_{free}$, the obstacle free region of the workspace. This region is often non-convex and the integral intractable. Term (4) is a constraint on the agent's initial state. Term (5) constrains the agent's trajectory to visit various goal regions at specific time steps, indexed by $k$, i.e. $\mathbf{x}_{k_g} \in \mathcal{G}_g$. The total number of goals is $G$. $\mathcal{G}$ represents a geometric region of the workspace that constrains the agent in some way. Boldfaced $\boldsymbol{\mathcal{G}}$ is a sequence of goals indexed by g, i.e. $\boldsymbol{\mathcal{G}} = [\mathcal{G}_0, \ldots, \mathcal{G}_g, \ldots, \mathcal{G}_G]$. To fully specify the problem, we introduce a formalism, the chance-constrained trajectory planning mission, ccTPM. The ccTPM is a tuple $< A, E, \boldsymbol{\mathcal{G}}, \mathcal{I}, J, \epsilon >$ where:

- $A =< \mathbf{f}, \mathcal{B} >$ is the agent consisting of a set of dynamical equations of motion $\mathbf{f}$ and a geometric shape $\mathcal{B}$
- $E$ represents the environment, composed of polytope obstacles
- $\boldsymbol{\mathcal{G}}$ is a sequence of goal regions that constrain the agent in some way along its trajectory
- $\mathcal{I}$ is an initial state
- $J$ is an objective function to be minimized
- $\epsilon$ is the value of the chance constraint, i.e. a bound on the probability of trajectory failure

The formalism is inspired by the qualitative state plan (QSP) in (Léauté, 2005) and the chance-constrained qualitative state plan (ccQSP) from (Ono et al., 2013). The goal region $\mathcal{G}$ is similar to the ccQSP *episode* in that episodes impose constraints on the evolution of the agent's state. However, rather than focus explicitly on temporal constraints as episodes do, the region construct imposes geometric constraints on the agent's trajectory over a sequence of time steps. Additionally, we assume that the input sequence $\boldsymbol{\mathcal{G}}$ is ordered. In many cases, given an unordered set of goals and generating an ordering that is both feasible and minimizes a cost function is a non-trivial task. However, assuming $\boldsymbol{\mathcal{G}}$ is ordered allows the present article to focus on planning with respect to more expressive environments than those considered by the ccQSP in (Ono et al., 2013). Orderings may be determined by using methods discussed in (Ono et al., 2013) or (Fernandez-Gonzalez et al., 2018).

### 3.1 Hybrid Search

The hybrid search consists of two levels: an upper level region planner and a lower level trajectory planner. The region planner determines which regions of the workspace should be explored as potential candidates for trajectories that travel from the initial state to a goal or between goal states. Given an ordered set of regions, the lower level trajectory planner optimizes a trajectory that is constrained to travel through these regions. An overview of the hybrid search is depicted in Fig. 3.

The region planner's regions are denoted with $\mathcal{R}$. In general, $\mathcal{R}$ (and the term *region*) refers to some Euclidean space in the agent's workspace, $\mathcal{W}$. In this article, $\mathcal{R}$ has dimensionality less than or equal to $d$ where $d$ is the dimensionality of the workspace. For example, in a 2D environment, $\mathcal{R}$ can refer to a ray (1D) or a rectangle (2D) or a variety of other 2D shapes. Another assumption in this work is that $\mathcal{R}$ are convex. This allows them to be readily translated into trajectory constraints input to the trajectory planner. One of the advantages of defining $\mathcal{R}$ to be general is to allow them to model a wide range of desired agent behaviors in the workspace. Both $\mathcal{I}$ and $\mathcal{G}$ discussed in the previous section are types of regions, i.e. $\mathcal{I} \in \mathcal{R}$ and $\mathcal{G} \in \mathcal{R}$. For example, a rectangle, line, or point could all represent goal regions. In Section 4.1, we introduce landmark regions, denoted by $\mathcal{L}$, which guide the agent around obstacles. Regions can also be combined to form motion primitives. Ordered sets of regions form paths, i.e. $\mathcal{P} = [\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_n, \ldots, \mathcal{R}_N]$. Regions along paths are indexed by $n$ with the total number of regions being $N$. The region planner

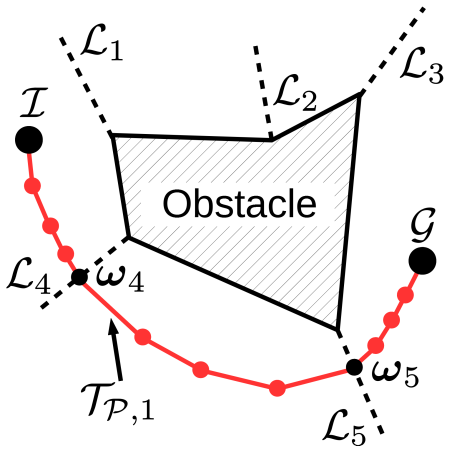



Figure 3: Example of the region planner and trajectory planner. The region planner produces a candidate sequence of regions and the trajectory planner optimizes a trajectory constrained to these regions. Dashed segments represent landmark regions, denoted with $\mathcal{L}$; these are a type of obstacle-avoidance region introduced in Section 4.1. Variables $\boldsymbol{\omega}_4$ and $\boldsymbol{\omega}_5$ are waypoint states which must lie on landmark regions $\mathcal{L}_4$ and $\mathcal{L}_5$. Red dots represent knot points of the trajectory optimization where the trajectory is indicated as $\mathcal{T}_{\mathcal{P},1}$.

passes paths to the trajectory planner as candidate sequences of geometric constraints that guide the agent from some initial state to some goal state or sequence of goal states.

The trajectory planner then optimizes a trajectory $\mathcal{T}$ constrained to lie on or in $\mathcal{P}$. A trajectory constrained in this way is denoted $\mathcal{T}_{\mathcal{P}}$. Trajectories $\mathcal{T}$ are sequences of state and control variables that describe the agent's motion. The form of $\mathcal{T}$ depends on the type of trajectory optimization. This article uses collocation trajectory optimization, where trajectories are sequences of tuples $(\mathbf{x}_k, \mathbf{u}_k, h_k)$ and the shooting method, which uses sequences of $(\mathbf{u}_k, h_k)$. Variable $\mathbf{x}_k$ represents the robot's state at knot index $k$, i.e. $\mathbf{x}_k \in \mathbb{R}^m$ where $m$ is the dimension of the robot's state. Variable $\mathbf{u}_k$ represents the control input at knot index $k$, i.e. $\mathbf{u}_k \in \mathbb{R}^p$ where $p$ is the number of control inputs. Variable $h_k \in \mathbb{R}$ represents the time step, i.e. the amount of time between two knot points. It may be held constant (a discrete time formulation) but is often included as a decision variable output by the trajectory optimizer. An important point is that trajectory knots are indexed using $k$ and regions are indexed using $n$. This is because the number of knots on a trajectory is typically much greater than the number of regions on a path. The number of knots in a trajectory optimization is determined by how accurate the interpolated trajectory must be with respect to the true trajectory; the number of regions on a path is dictated by obstacle geometry and goal states.

In order to constrain the trajectory to lie in or on $\mathcal{P}$, we denote certain states *waypoint* states, $\boldsymbol{\omega}$. These represent the agent's position in the workspace. For 2D workspaces, $\boldsymbol{\omega}_n = (x, y)$ and for 3D workspaces $\boldsymbol{\omega}_n = (x, y, z)$. Waypoints are indexed by $n$, which means only select trajectory knot points are constrained by a region. For example, if there are $K_n$ trajectory knots per waypoint, then the $3K_n^{th}$ knot is constrained by $\mathcal{R}_3$. This construction is illustrated for a simple toy example in Fig. 3. A number of methods are

possible to determine the number of knot points between regions, $K_n$. The simplest is to assume $K_n$ is constant; another method is to use the distance between regions to determine $K_n$. In this work, we assume the agent has a unique set of waypoint states, i.e. we assume the agent is a point or rigid body and do not consider multi-link robots.

A second set of agent states are termed *configuration* states, in reference to the configuration space used in robot motion planning (LaValle, 2006). Configuration states define the agent's configuration; if the agent consists of a rigid body $\mathcal{B}$ then the configuration states specify the position of all points in $\mathcal{B}$ in the workspace (Choset, 2010). For example, if the agent consists of a polyhedron in 3D, one representation of the configuration states is $(x, y, z, \psi, \theta, \phi)$ where $\psi$, $\theta$, and $\phi$ are Euler angles as described in (Triantafyllou, 2004). Configuration states are important for computing the collision risk in agents with non-trivial geometry (presented in Section 8 of this paper).

### 3.2 Assumptions Modeling Agent & Environment Geometry

We assume the environment $E$ consists of a set of closed, continuous obstacle polytopes, $\boldsymbol{\mathcal{E}}$, i.e. polygons in 2D workspaces and polyhedra in 3D. Each polytope is composed of facets, $\mathcal{F}$, which are polygons in 3D and edges in 2D. Each facet is bordered by a set of sub-facets, $\boldsymbol{\mathcal{S}}$. Geometrically, the sub-facets are line segments in 3D workspaces and points in 2D workspaces. For a triangulated polyhedron, each facet has three sub-facets, i.e. $|\boldsymbol{\mathcal{S}}| = 3$. In 2D, each edge facet has two points, i.e. $|\boldsymbol{\mathcal{S}}| = 2$. For each facet, an outward pointing normal vector $\hat{n}$ is assumed known. Each facet can access its neighboring facets; in 2D each vertex contains a pointer to the neighboring edges; in 3D the halfedge data structure as described in (Kettner, 2018) is used to link neighboring facets.

Polytopes are used for three reasons. First, polytopes are a linear approximation of a surface and can efficiently represent many real-world objects and surfaces (Berg et al., 2008). Second, polytopes enable generation of landmark constraints that allow for computation of the collision probability, as described in Section 7. Thirdly, polytopes greatly simplify the collision-checking process because many efficient algorithms exist to check collisions between polytope facets (Ericson, 2005). Because of this, many approaches in literature also implicitly require polytope-based obstacles (Jimenez, Thomas, & Torras, 1998).

## 4. The Region Planner

The purpose of the region planner is to determine a path $\mathcal{P}$, which the trajectory planner can compile into constraints and use to compute a feasible risk-bound trajectory. The path must contain the agent's initial state and all goals necessary to solve the ccTPM. The path should be constructed with regard to the ccTPM's objective, i.e. minimizing the time or distance traveled, and avoid generating unnecessary constraints for the trajectory planner. Furthermore, the path should help the trajectory planner produce feasible paths with respect to the obstacle-avoidance chance constraint. The region planner is characterized by discrete choices, i.e. whether to include a region on a path. It has three chief components: a type of obstacle-avoidance region (termed *landmark region*) covered in Section 4.1; a set of methods to efficiently connect regions to form paths, covered in Section 4.2; and a search strategy termed First Feasible Hybrid Search to determine which paths are passed to the trajectory planner, Section 5. The region planner can be viewed as reasoning about and

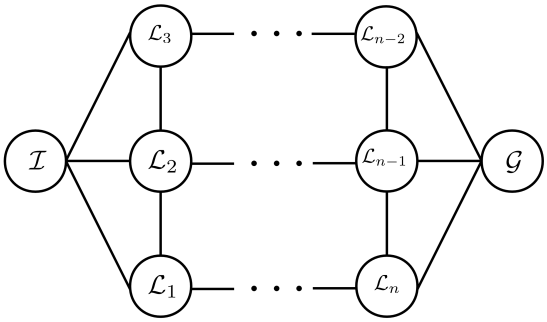



Figure 4: The region planner's task is to find paths through the region graph and pass them to the trajectory planner.

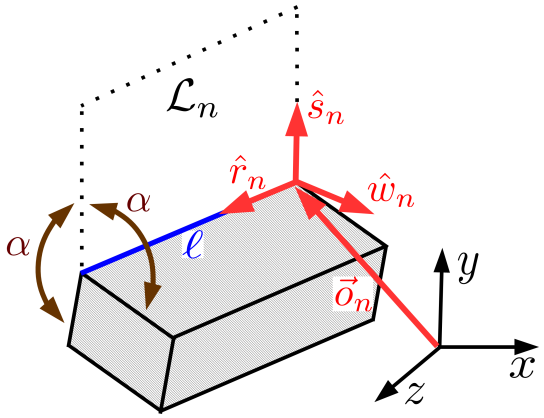



Figure 5: The plane and coordinate system that represent a landmark region $\mathcal{L}_n$ for a 3D workspace. The dotted lines do not represent a constraint; they are shown to illustrate the plane.

searching a region graph. In this undirected graph, regions are vertices and connections between regions are edges. Directed edges in the region graph represent orderings of regions, i.e. paths. This construction is depicted in Fig. 4.

### 4.1 Landmark Regions

Landmark regions are types of regions designed to model obstacle-avoidance constraints. Geometrically, landmark regions are bounded hyperplanes. In three dimensions, this hyperplane is a bounded plane and in two dimensions, the constraint is a ray. The plane is bounded because it is bordered by a line $\ell_n$ incident to an obstacle sub-facet. For each sub-facet $\mathcal{S}$ in an environment, there is a corresponding implied landmark constraint $\mathcal{L}$. The geometry is illustrated in Figs. 5 and 6. The bounded hyperplane extends upwards and in all directions from the blue line in Fig. 5. The hyperplane's orientation divides the angle between the two neighboring obstacle facets. If the angle between the facets is $2\alpha$

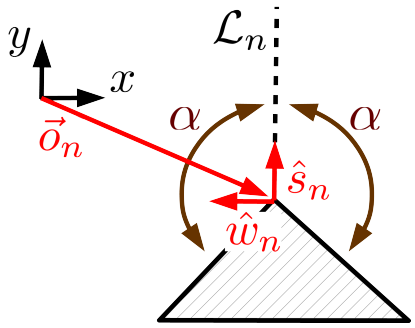



Figure 6: The ray that represents a landmark region $\mathcal{L}_n$ for a 2D workspace.

(measured on the side of the obstacle surface with outward pointing normal vectors), then the hyperplane is positioned $\alpha$ from each neighboring facet.

An advantage of using a hyperplane is that the transformation from the global waypoint states $\boldsymbol{\omega}_n$ to a coordinate system embedded in the plane is linear. In three dimensions this transformation is:

$$s_n = (\boldsymbol{\omega}_n - \vec{o}_n) \cdot \hat{s}_n \tag{6}$$
$$r_n = (\boldsymbol{\omega}_n - \vec{o}_n) \cdot \hat{r}_n \tag{7}$$
$$w_n = (\boldsymbol{\omega}_n - \vec{o}_n) \cdot \hat{w}_n \tag{8}$$

where $(\hat{s}_n, \hat{r}_n, \hat{w}_n)$ form an orthogonal basis of the embedded coordinate system. Direction $\hat{r}_n$ runs along the line $\ell_n$ (i.e. it runs along the obstacle's surface), direction $\hat{s}_n$ measures the distance from the obstacle's surface to the waypoint and will be used in Section 7 to approximate the collision risk. Direction $\hat{w}_n$ is normal to the plane. Vector $\vec{o}_n$ is the vector from the origin of the global coordinate system to the origin of the embedded coordinate system. Variables $(s_n, r_n, w_n)$ are the agent's waypoint state transformed into the embedded coordinate system. In order for waypoint states to lie on the corresponding landmark regions, the constraints are necessary:

$$w_n = 0 \ \ \forall \ \ n \in N \tag{9}$$

In two dimensions, the formulation is analogous. Instead of lying on a plane, the waypoint is constrained to lie on a ray that emanates from a vertex. The 2D embedded basis $(\hat{s}_n, \hat{w}_n)$ is illustrated in Fig. 6.

There are a number of insights that motivate the landmark region construction. First, they are completely defined given a polytope surface. This means that they do not require optimization to generate as required by convex decomposition methods such as described in (Deits & Tedrake, 2015). Secondly, they do not place restricting conditions on the obstacle shape or dimensions such as requiring obstacles to be 2D or convex. Third, the number of landmark regions is dictated by obstacle complexity; more complex obstacles will have larger numbers of landmarks and simpler environments will have fewer. This contrasts with uniform workspace discretizations typical of occupancy maps where the discretization scale is difficult to select a priori. Finally, the landmark region compiles into simple linear

constraints on the agent trajectory. The challenge is not constructing the landmark regions, but connecting them into paths. This is covered in the next section.

### 4.2 Enumerating Region Graph Edges

Given the landmark regions described in the preceding section, it is necessary to connect them to form paths through the region graph. This section describes two methods for region graph edge enumeration. First, in Section 4.2.1 a heuristic-based "Polytope to the Goal" technique is developed that helps determine which regions may be active on the shortest path to the goal. This method uses the insight that environments are likely very complex and only a small number of surfaces will serve as active constraints for motion planning. It attempts to bypass irrelevant regions by shooting a polytope directly to the next goal region. The method is useful for problems whose objective seeks to minimize the agent's distance traveled, trajectory time, or energy usage. Second, in Section 4.2.2 we describe a method that enumerates paths using obstacle surface facets directly. The second method is useful when the heuristic-based approach fails, e.g. due to the heuristic becoming trapped in an obstacle crevasse. Many other problem-specific edge enumeration strategies are possible. For example, it may be desired that an agent travel with a constant direction along a surface. In this case, paths may be formed by enumerating neighboring landmark regions along a predetermined direction.

Before describing methods for enumerating connections between regions to form paths, we define a *grounded path*.

**Definition 4.1.** A *grounded path* is a sequence of pairs consisting of a region and waypoint state, i.e. $\mathcal{P}_{grounded} = [(\mathcal{I}, \omega_{\mathcal{I}}), (\mathcal{L}_1, \omega_{\mathcal{L}_1}), (\mathcal{L}_2, \omega_{\mathcal{L}_2}), \dots]$. Each waypoint is constrained to lie within the corresponding region.

The purpose of the grounded path is to maintain an estimate of where the optimal trajectory travels through a region. As described in the following section, the waypoints on a grounded path serve as points to use when computing a heuristic cost estimate of the path. To simplify notation, $\mathcal{P}$ is used to refer to a grounded path and path, the difference being clear from context.

#### 4.2.1 Heuristic-Based Polytope to the Goal

The purpose of the heuristic approach is to determine landmark regions that likely lie on shortest paths between the initial region and goal region or between goal regions. It is inspired by the Rapidly-Exploring Random Tree strategy of sampling from the goal state and attempting to connect the present search tree to the goal (LaValle & Kuffner Jr., 2000). The approach consists of shooting a polytope to the goal (abbreviated $PtG$) and detecting collisions along this polytope. If there are collisions, the landmark regions incident to the sub-facets of the colliding obstacle facet are enumerated and new paths created by appending the new landmark regions to the parent path. The PtG is an approximation of the state distribution describing the agent's position if it were to travel directly to the goal. For example, if the agent's position uncertainty grows proportionally to the distance traveled $D$ and the state standard deviation $\sigma$, the width of the PtG's base may be proportional to $D\sigma$.

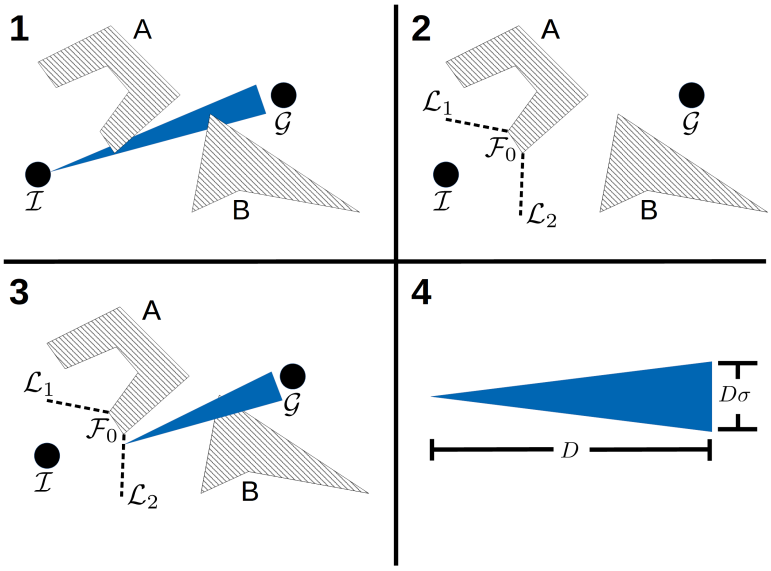



Figure 7: An illustration of the heuristic-based Polytope to Goal method. 1.) A polygon is directed at the goal and a collision checker collects collisions along this polygon. 2.) Landmark regions (i.e. rays in 2D) are enumerated at the sub-facets of the closest colliding facet, in this case $\mathcal{F}_0$. 3.) Based on a heuristic cost estimate, the process repeats using a point on $\mathcal{L}_2$ as the parent. 4.) The shape of the polygon approximates state uncertainty, e.g. a multiple of the state uncertainty's standard deviation.

Specifically, given a grounded path $\mathcal{P}$, let the final pair on the grounded path be denoted $(\mathcal{R}_f, \omega_f)$. Let $\mathcal{G}$ be the current goal to which the agent is navigating. A polytope is generated with vertex at $\omega_f$ and directed towards the goal. The geometry of the polytope reflects belief about the agent's position uncertainty. In 2D, a triangle is used to model the PtG while in 3D workspaces a tetrahedron is used. More complex shapes are possible depending on the accuracy with which one approximates the agent's state distribution. Collisions are collected along the PtG. Ignoring degenerate collisions, the PtG will collide with a set of facets $\boldsymbol{\mathcal{F}}$. Denote the closest colliding facet to $\omega_f$ as $\mathcal{F}_0$. Let $\boldsymbol{\mathcal{S}}_{\mathcal{F}_0}$ be the set of sub-facets associated with $\mathcal{F}_0$ and $\boldsymbol{\mathcal{L}}$ be the set of landmark regions associated with each sub-facet in $\boldsymbol{\mathcal{S}}_{\mathcal{F}_0}$. New paths are generated by appending $\mathcal{L}_{new} \in \boldsymbol{\mathcal{L}}$ to the original grounded path $\mathcal{P}$. For example, in 3D, if the PtG collides with a triangulated surface, three new paths are created. If the original grounded path is $\mathcal{P} = [(\mathcal{I}, \omega_{\mathcal{I}}), (\mathcal{L}_1, \omega_1)]$, three new paths of the form $\mathcal{P} = [(\mathcal{I}, \omega_{\mathcal{I}}), (\mathcal{L}_1, \omega_1), (\mathcal{L}_{new}, \omega_{new})]$ are generated. Point $\omega_{new}$ represents an estimate of where the optimal trajectory traverses $\mathcal{L}_{new}$. It may be generated by performing a partial trajectory optimization or simply by taking the closest point on $\mathcal{L}_{new}$ to $\mathcal{L}_1$. While the latter method is inaccurate, it is likely much simpler to compute.

Importantly, the process can be repeated by selecting one of the children as the new parent, re-shooting a polytope to the goal (using $\omega_{new}$ as the vertex), and collecting collisions along this polytope. The selection of children is covered more fully when discussing the

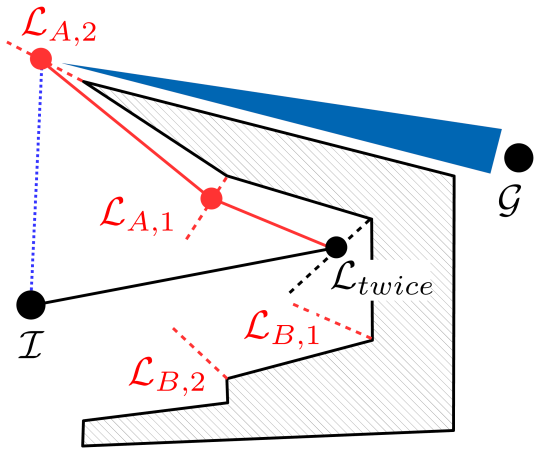



Figure 8: Using only the PtG approach, $\mathcal{L}_{twice}$ will be inserted onto the path twice; the black line represents the original path. The algorithm attempts to avoid $\mathcal{L}_{twice}$ by forming paths along the surface, shown in red as paths $A$ and $B$. Climbing a single vertex from $\mathcal{L}_{twice}$ finds $\mathcal{L}_{A,1}$ and $\mathcal{L}_{B,1}$; however, both paths will result in $\mathcal{L}_{twice}$ being part of another path loop. On forming a second loop, the algorithm enumerates the neighbors of $\mathcal{L}_{A,1}$ and $\mathcal{L}_{B,1}$, finding paths to $\mathcal{L}_{A,2}$ and $\mathcal{L}_{B,2}$. The path $[\mathcal{I}, \mathcal{L}_{twice}, \mathcal{L}_{A,1}, \mathcal{L}_{A,2}]$ is able to progress to the goal. The dotted blue line from $\mathcal{I}$ to $\mathcal{L}_{A,2}$ indicates how the path may be rewired.

region graph search strategy in Section 5. The PtG method is illustrated on a 2D toy problem in Fig. 7.

#### 4.2.2 Avoiding Path Loops

If the proposed algorithm uses only the heuristic-based PtG, path loops could result and the algorithm may not terminate. This is because the PtG could become trapped in an obstacle surface feature and continuously intersect the same facet. The search must reach a new landmark region where the PtG can continue making progress to the goal. To do this, a loop check is performed; this checks whether a landmark region has been added twice to a path. If a loop is detected, the algorithm attempts to move away from the landmark region that generated the loop by traversing the obstacle surface. The idea is illustrated in Fig. 8. Landmark region $\mathcal{L}_{twice}$ has been inserted into the path twice. Two paths are then generated each ending with a landmark region that neighbors $\mathcal{L}_{twice}$: $\mathcal{L}_{A,1}$ and $\mathcal{L}_{B,1}$. If $\mathcal{L}_{twice}$ causes another loop, new paths are generated which end in the neighbors of $\mathcal{L}_{A,1}$ and $\mathcal{L}_{B,1}$ (in this case including $\mathcal{L}_{A,2}$, from which progress can be made to the goal). A visited list of landmark regions keeps track of which $\mathcal{L}$ have been added to paths attempting to avoid $\mathcal{L}_{twice}$; this prevents the procedure from generating a path loop itself. The subroutine is described in Alg. 4, *AvoidPathLoop*, in Section 5.1.4.

A caveat of using neighboring facets to avoid loops is that the resulting path may be very poor (consider $[\mathcal{I}, \mathcal{L}_{twice}, \mathcal{L}_{A,1}, \mathcal{L}_{A,2}]$ in Fig. 8). Because of this, before the region planner passes a path to the trajectory planner, an attempt is made to re-wire it. Specifically, given a path $\mathcal{P} = [\mathcal{I}, \mathcal{L}_1, ..., \mathcal{L}_{n-1}, \mathcal{G}]$, a subroutine finds a subset of $\mathcal{P}$ that begins at $\mathcal{I}$ and ends

at $\mathcal{G}$, which passes collision checks between each $\mathcal{L}_n$. The rewiring is shown as the blue line in Fig. 8 connecting $\mathcal{I}$ to $\mathcal{L}_{A,2}$.

## 5. First Feasible Hybrid Search

Given the region graph's vertices and methods for enumerating its edges, we now specify the search that generates candidate paths through the region graph and passes them to the trajectory planner. First Feasible Hybrid Search has the objective of generating good, feasible paths quickly in complex environments. The key innovation behind FFHS is integrating the region and trajectory planners in a way that accomplishes this task. Because trajectory optimization calls are expensive, the search seeks to minimize them.

FFHS solves the goal-directed trajectory planning problem stated in Section 3 sequentially. Because the input goal sequence $\boldsymbol{\mathcal{G}}$ is ordered, the task of FFHS is to determine a sequence of regions that help navigate the agent between successive goals. For example, it determines a path from goal $\mathcal{G}_{g-1}$ to $\mathcal{G}_g$ and then from goal $\mathcal{G}_g$ to $\mathcal{G}_{g+1}$. Because the algorithm operates on pairs of goals, we use the term *current goal sequence* to denote the pair of goal regions between which FFHS is attempting to generate a path. The current goal sequence consists of parent goal $\mathcal{G}_g$ and child goal $\mathcal{G}_{g+1}$. To simplify the exposition, we assume that the initial region $\mathcal{I}$ is included as the first element of the sequence $\boldsymbol{\mathcal{G}}$ such that $\mathcal{G}_0$ is equal to $\mathcal{I}$. Given this formulation, we define the notions of a *complete* path, *prefix* path, and *suffix* path.

**Definition 5.1.** Given the current goal sequence $[\mathcal{G}_g, \mathcal{G}_{g+1}]$, a *prefix path* $\mathcal{P}_{pre}$ is a grounded path that contains parent goal $\mathcal{G}_g$ but does not contain the child goal region $\mathcal{G}_{g+1}$ or any goal region $g'$ for $g' > g$, i.e. $\mathcal{P}_{pre} = \left[(\mathcal{I}, \omega_{\mathcal{I}}), \ldots, \left(\mathcal{G}_g, \omega_{\mathcal{G}_g}\right), \ldots, (\mathcal{R}_n, \omega_n)\right]$, where no other goal region follows $\left(\mathcal{G}_g, \omega_{\mathcal{G}_g}\right)$.

**Definition 5.2.** Given the current goal sequence $[\mathcal{G}_g, \mathcal{G}_{g+1}]$, a *suffix path* $\mathcal{P}_{suf}$ is a grounded path that contains the current child goal region $\mathcal{G}_{g+1}$ but does not contain the parent goal region $\mathcal{G}_g$ or any goal region $g'$ for $g' < g$, i.e. $\mathcal{P}_{suf} = \left[\ldots, \left(\mathcal{G}_{g+1}, \omega_{\mathcal{G}_{g+1}}\right), \ldots, (\mathcal{G}_G, \omega_G)\right]$, where no other goal region precedes $\left(\mathcal{G}_{g+1}, \omega_{\mathcal{G}_{g+1}}\right)$.

**Definition 5.3.** A grounded path is *complete* with respect to the current goal sequence $[\mathcal{G}_g, \mathcal{G}_{g+1}]$ if it contains both $\mathcal{G}_g$ and $\mathcal{G}_{g+1}$.

**Definition 5.4.** A grounded path is *mission complete* if it contains all goals in the sequence $\boldsymbol{\mathcal{G}}$ in their proper order: $\mathcal{P}_{mc} = \left[(\mathcal{I}, \omega_{\mathcal{I}}), \ldots, \left(\mathcal{G}_g, \omega_{\mathcal{G}_g}\right), \ldots, \left(\mathcal{G}_{g+1}, \omega_{\mathcal{G}_{g+1}}\right), \ldots, \ (\mathcal{G}_G, \omega_{\mathcal{G}_G})\right]$

The insight behind First Feasible Hybrid Search is that only *mission complete* paths are passed to the optimizer to reduce the number of calls to the trajectory planner. FFHS generates a mission complete path using heuristics to compare path costs without performing full trajectory optimizations. Once a mission complete path is found, the path is passed to the trajectory planner for optimization.

Regarding the chance constraint, a caveat of this approach is that it may be necessary to ensure the trajectory is feasible with respect to risk post-optimization. This is the case when the chance constraint model formulates constraints directly from the path $\mathcal{P}$. Because the path $\mathcal{P}$ is generated using deterministic heuristics, it may differ substantially from the

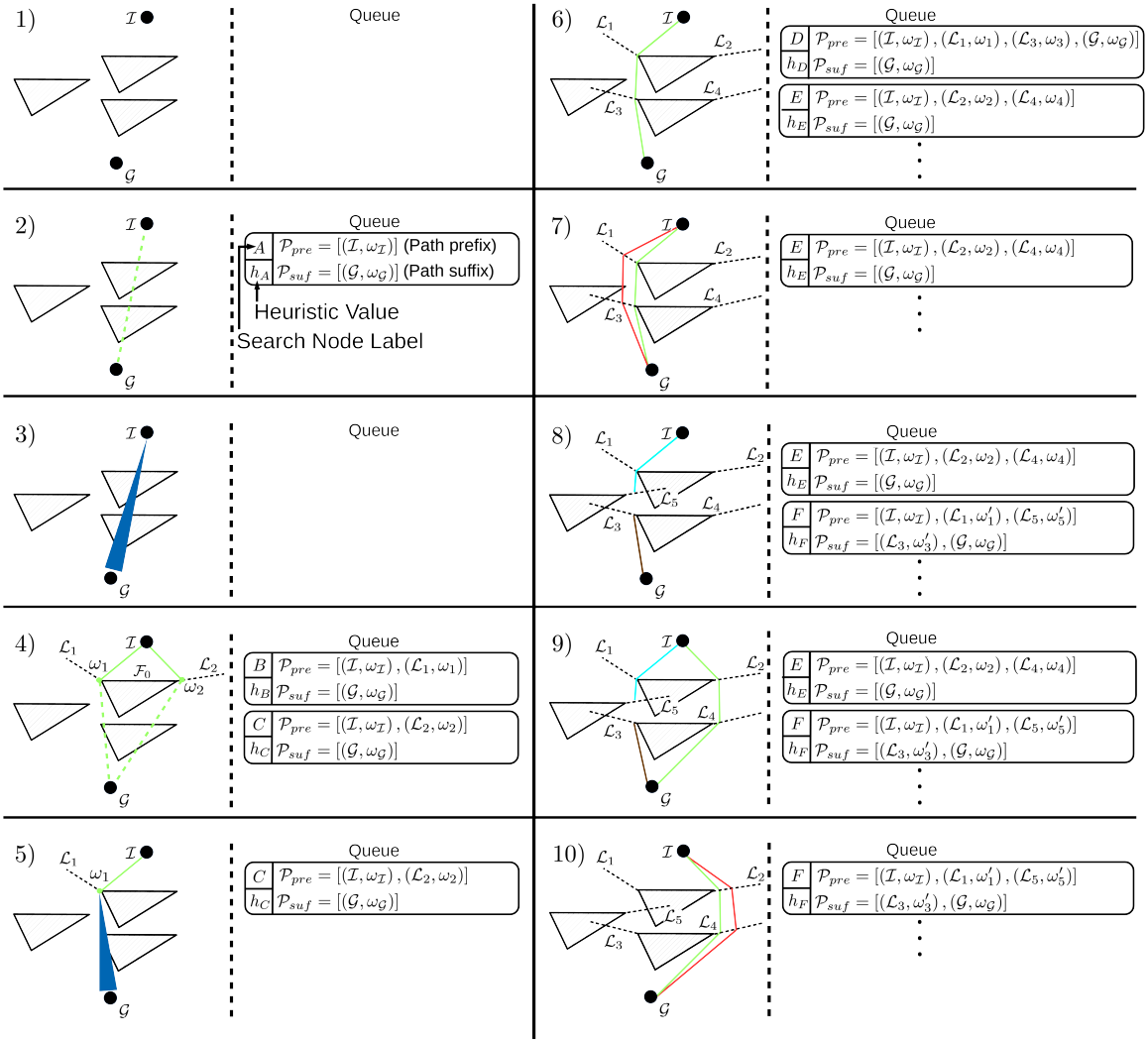



Figure 9: Illustration of key steps of First Feasible Hybrid Search on a toy problem. 1) The agent must navigate from initial region $\mathcal{I}$, i.e. the current goal sequence is $[\mathcal{I}, \mathcal{G}]$. 2) The initial search node $A$ contains a prefix with only the initial region and a suffix with only the goal region; the heuristic cost is the distance between them. 3) $A$ is popped from the queue and a polygon is shot to the goal. 4) Edge $\mathcal{F}_0$ is the closest intersection; two new search nodes are created, $B$ and $C$, with landmark regions $\mathcal{L}_1$ and $\mathcal{L}_2$ added to the respective prefix paths. Heuristic costs $h_B$ and $h_C$ are estimated as $f = g_{pre} + h_{pre,suf} + g_{suf}$ where $g_{pre}$ is the heuristic cost estimate of the prefix path (a solid green line), $h_{pre,suf}$ is the cost estimate from the final prefix region to the initial suffix region (dashed green line), and $g_{suf}$ is the heuristic estimate of the suffix path. 5) Node $B$ has a lower heuristic value, $h_B < h_C$, and is popped from the queue; a second polytope to the goal is generated from $\omega_1$. 6) The prefix path of node $D$ is mission complete and passed to the trajectory optimizer. 7) The risk-bound trajectory optimization produces a trajectory (red) that is further from the obstacles than the heuristically-generated path (green). However, it is invalid because it intersects an obstacle unseen to the heuristic-based method. 8) The trajectory is split into a new prefix (cyan) and suffix (brown) based on the conflicting obstacle. The node is re-added to the queue (node $F$). 9) Node $E$ now has the lowest heuristic value. 10) The trajectory optimized from node $E$'s path is validated by subroutine $ValidateTrajectory$.

**Algorithm 1:** First Feasible Hybrid Search

**Input:** A ccTPM planning problem
**Output:** A trajectory $\mathcal{T}$ that satisfies the ccTPM

$PriorityQueue.Initialize(\mathcal{N}_{root})$
**do**
  $\mathcal{N} \leftarrow PathGenerator(PriorityQueue.pop())$
  **if** $\mathcal{N}.IsMissionComplete()$ **then**
    $TrajectoryPlanner\,(\mathcal{N})$
    **if** $TrajectoryPlanner.Success$ **then**
      $ValidateTrajectory\,(\mathcal{N})$
**while** $\mathcal{N}.IsMissionComplete() \wedge \neg\mathcal{N}.IsValidated()$
**if** $\mathcal{N}.IsMissionComplete() \wedge \mathcal{N}.IsValidated()$ **then**
  **return** success
**else**
  **return** failure

**Algorithm 2:** PathGenerator

**Input:** The current candidate node $\mathcal{N}$
**Output:** A mission complete path $\mathcal{P}$ that satisfies all ccTPM goals

**if** $\mathcal{N}.IsMissionComplete()$ **then**
  $RewirePath\,(\mathcal{N})$
  **return** $\mathcal{N}$
**if** $\mathcal{N}.IsPathComplete()$ **then**
  $\mathcal{G}_{g+1} \leftarrow SelectNextUnassignedGoal(\mathcal{N})$
  $\mathcal{N}_c \leftarrow InitNodeStartingNextPath\,(\mathcal{N}, \mathcal{G}_{g+1})$
  $PathGenerator\,(\mathcal{N}_c)$
**else**
  $\mathcal{N}_c \leftarrow GenerateNextBestChild\,(\mathcal{N})$
  $PathGenerator\,(\mathcal{N}_c)$
**end**

risk-bound trajectory post-optimization. The optimized risk-bound trajectory may intersect obstacles not included on $\mathcal{P}$. The validation step ensures the optimized trajectory is feasible; if it is not, it is split into new prefix and suffix paths at the infeasible segment.

### 5.1 Components of FFHS

FFHS is described in detail in Algorithms 1 - 7 and in the following paragraphs. A step-by-step overview of FFHS is given in Fig. 9 on a 2D problem. To focus on the search, this section treats the trajectory optimizer as a black box optimizer; it is described in detail in the second half of this article and the appendices.

**Algorithm 3:** GenerateNextBestChild

**Input:** A parent search node $\mathcal{N}_p$, the current goal $\mathcal{G}_g$
**Output:** A child search node $\mathcal{N}_c$

**if** $CheckForPathLoops\,(N_p)$ **then**
  $\mathcal{N}_c \leftarrow AvoidPathLoop\,(\mathcal{N}_{p,pre})$
  **return** $\mathcal{N}_c$
**else**
  $PtG \leftarrow GenPtG\,(\mathcal{N}_{p,pre}, \mathcal{N}_{p,suf}, \mathcal{G}_g)$
  $ClosestFacet \leftarrow CollisionChecker\,(\mathcal{N}_{p,pre}, PtG)$
  $Children \leftarrow GenChildren\,(ClosestFacet)$
  $PriorityQueue.Insert(Children)$
  **return** $PriorityQueue.pop()$
**end**

**Algorithm 4:** AvoidPathLoop

**Input:** A prefix path $\mathcal{P}_{pre}$ with a loop, a landmark region $\mathcal{L}_{twice}$ that was inserted into the current path twice.
**Output:** An updated path $\mathcal{P}'$ that attempts to lead away from the sub-facet generating the loop or failure

$\boldsymbol{\mathcal{L}}_{visited} \leftarrow GetVisitedList(\mathcal{L}_{twice})$
$\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}} \leftarrow GetSurfacePaths(\mathcal{L}_{twice})$
$PriorityQueue_{\mathcal{L}} \leftarrow GetPriorityQueue(\mathcal{L}_{twice})$
$\mathcal{P}'_{pre} \leftarrow \mathcal{P}_{pre,\mathcal{I}:\mathcal{L}_{twice}}$
$\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}} \leftarrow ExpandNeighbors(\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}, \boldsymbol{\mathcal{L}}_{visited})$
**if** $\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}} = \emptyset$ **and** $PriorityQueue_{\mathcal{L}}.Empty()$ **then**
  **if** $\vec{v}_{Pt_{closest},\mathcal{G}} \cdot \hat{n}_{Pt_{closest}} < 0$ **then**
    **return** Failure
  $\mathcal{P}_{avoid} \leftarrow \mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$
**else**
  **for** $\mathcal{P} \in \boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$ **do**
    **for** $\mathcal{F} \in \mathcal{P}.Last_{\boldsymbol{\mathcal{F}}}$ **do**
      **if** $Dist(\mathcal{F}, \mathcal{G}) < Dist(Pt_{closest}, \mathcal{G})$ **then**
        $\mathcal{L}_{closest} \leftarrow InitializeLandmarkRegion\,(\mathcal{F}_{closest})$
        $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}} \leftarrow Concatenate\,(\mathcal{P}, \mathcal{L}_{closest})$
    **end**
    $PriorityQueue_{\mathcal{L}}.insert(\mathcal{P})$
  **end**
  $\mathcal{P}_{avoid} \leftarrow PriorityQueue_{\mathcal{L}}.Pop()$
$\mathcal{P}' \leftarrow Concatenate(\mathcal{P}'_{pre}, \mathcal{P}_{avoid})$
**return** $\mathcal{P}'$

**Algorithm 5:** ExpandNeighbors

**Input:** A set of paths $\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}$ starting at $\mathcal{L}_{twice}$, a visited list $\boldsymbol{\mathcal{L}}_{visited}$ containing all landmark regions in $\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}$

**Output:** An updated $\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$ and $\boldsymbol{\mathcal{L}}_{visited}$

```
1  P'_{L_twice} ← ∅
2  for P ∈ P_{L_twice} do
3  |  L_nbrs ← GetNeighbors(P.Last)
4  |  for L ∈ L_nbrs do
5  |  |  if L ∉ L_visited then
6  |  |  |  L_visited.Insert(L)
7  |  |  |  P_new ← P.Append(L)
8  |  |  |  P'_{L_twice}.Insert(P_new)
9  |  end
10 end
11 return P'_{L_twice}, L_visited
```

**Algorithm 6:** RewirePath

**Input:** A mission complete path $\mathcal{P}_{mc}$.

**Output:** An updated path $\mathcal{P}'_{mc}$ with equal or lower heuristic cost than $\mathcal{P}_{mc}$

```
1 for P_comp ∈ P_mc do
2 |  R ← P_comp.GetRegions()
3 |  G_g ← P_comp.InitialState()
4 |  G_{g+1} ← P_comp.GoalState()
5 |  P'_comp ← AStar(G_g, G_{g+1}, R)
6 |  P_mc.Replace(P_comp, P'_comp)
7 end
```

### 5.1.1 FFHS

The high level search, First Feasible Hybrid Search, is provided in Algorithm 1. The search begins by initializing a priority queue with the root node, $\mathcal{N}_{root}$. Each node on the search tree is a tuple $\mathcal{N} = (\mathcal{P}_{pre}, \mathcal{P}_{suf}, h)$ where $\mathcal{P}_{pre}$ is a grounded prefix path, $\mathcal{P}_{suf}$ is a grounded suffix path, and $h$ is a heuristic estimate of the node's cost. The root node is initialized with a prefix path of $\mathcal{I}$ (i.e. $\mathcal{G}_0$) and suffix path of $\mathcal{G}_1$. The priority queue stores nodes ordered by their heuristic cost. The top of the priority queue is popped and it is input to the *PathGenerator*. The path generator attempts to return a mission complete path. If it is successful, the node is passed to the trajectory planner for optimization. It may be the case that the path generator was unable to find a complete path (as in the case of an infeasible problem), which necessitates the check $\mathcal{N}.IsMissionComplete()$ (Line 4 of Alg. 1). In this case, the trajectory planner is unable to run and the algorithm returns failure. After trajectory optimization, the resulting trajectory may be validated (Line 7) depending on the risk model.

**Algorithm 7:** ValidateTrajectory

**Input:** A trajectory $\mathcal{T}$ post-optimization, the node that generated the trajectory, $\mathcal{N}$, with grounded complete path $\mathcal{P}$

**Output:** An assignment to the node's validation flag, $\mathcal{N}.IsValidated$, or failure

```
1  IsPathValid ← True
2  for (x_k, x_{k+1}) ∈ T do
3      if IsTrajectoryInfeasible(x_k, x_{k+1}) then
4          if (L_n, L_{n+1}) ∈ L_visited then
5              return Failure
6          else
7              L_visited.Insert(L_n, L_{n+1})
8          end
9          N_pre = P_{n≤k}
10         N_suf = P_{n>k}
11         IsPathValid ← False
12         PriorityQueue.Insert(N)
13 end
14 N.IsValidated ← IsPathValid
```

### 5.1.2 Path Generator

The goal of the *PathGenerator*, Alg. 2, is to receive a node and attempt to generate a mission complete path. There are three cases to consider for *PathGenerator*. First, if the node's prefix path is already mission complete (Line 1 of Alg. 2), an attempt is made to rewire the path and the path is returned. Second, if the path is complete but not mission complete, it means that the algorithm has found a path between the current goal sequence $[\mathcal{G}_{g-1}, \mathcal{G}_g]$ and must find a path to goal $g+1$. In this case, the algorithm generates a new node $\mathcal{N}_c$ where $\mathcal{N}_c$ has a prefix path equal to that of node $\mathcal{N}$ and a suffix path containing $\left(\mathcal{G}_{g+1}, \omega_{\mathcal{G}_{g+1}}\right)$ (Line 6). The third case is that $\mathcal{N}$ contains neither a mission complete nor a complete path. In this case, the algorithm attempts to generate a path between current goal sequence $\mathcal{G}_g$ and $\mathcal{G}_{g+1}$ and calls *GenerateNextBestChild*.

### 5.1.3 GenerateNextBestChild

The purpose of the subroutine, *GenerateNextBestChild*, Alg. 3, is to complete the path between $\mathcal{G}_g$ and $\mathcal{G}_{g+1}$ by adding landmark regions to the prefix path of the parent node $\mathcal{N}_p$. The methods in *GenerateNextBestChild* that enumerate landmark regions are those described in Section 4.2. First, a loop check is performed on the prefix path of the parent node (Line 1). If a loop is found, an attempt is made to climb from the loop using the strategy described in Section 4.2.2 (Alg. 4, *AvoidPathLoop*). If no loop is found, the method of shooting a polytope to the goal is used. As described in the previous section, a polytope is shot to the goal and collisions are collected along this polytope. Child nodes are generated for each sub-facet of the closest colliding facet. These children are then added to the priority queue. This is the same priority queue initialized in Line 1 of Alg. 1. The best candidate is returned from the queue (*PriorityQueue.pop*()).

### 5.1.4 Avoiding Path Loops

The subroutine to avoid path loops is described in Alg. 4. The core idea behind *AvoidPathLoop* is to generate paths extending from the landmark region that caused the loop, $\mathcal{L}_{twice}$, that run along the obstacle surface until the PtG is once again able to make progress to the goal. First, the algorithm retrieves a number of data structures associated with $\mathcal{L}_{twice}$, Lines 1 - 3. $\boldsymbol{\mathcal{L}}_{visited}$ is a set of landmark regions *AvoidPathLoop* has previously explored attempting to avoid $\mathcal{L}_{twice}$. $\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}$ is a set of previously generated paths running along surface $\mathcal{E}$, where $\mathcal{E}$ is the polytope obstacle surface to which $\mathcal{L}_{twice}$ is incident. $\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}$ is initialized with a single path consisting of only $\mathcal{L}_{twice}$ the first time $\mathcal{L}_{twice}$ causes a loop. $PriorityQueue_{\mathcal{L}}$ is a queue of paths *AvoidPathLoop* relies on to avoid $\mathcal{L}_{twice}$; it is initialized as empty. In Line 4, $\mathcal{P}'_{pre}$ is initialized as the path from the first element on $\mathcal{P}_{pre}$, $\mathcal{I}$, to the first occurrence of $\mathcal{L}_{twice}$. In Line 5, a new set of candidate paths is generated that attempt to lead away from $\mathcal{L}_{twice}$ that travel along $\mathcal{E}$. The subroutine $ExpandNeighbors$ is given in Alg. 5. If the new set of paths, $\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$, is not empty, then each path in $\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$ is added to $PriorityQueue_{\mathcal{L}}$, Line 17. In this case, the top of the queue is popped, Line 19, and the path leading along the surface, $\mathcal{P}_{avoid}$, is concatenated with $\mathcal{P}'_{pre}$, Line 20. This path is then returned. Certain edge cases must be considered. If $ExpandNeighbors$ finds no new paths and $PriorityQueue_{\mathcal{L}}$ is empty, *AvoidPathLoop* sets $\mathcal{P}_{avoid}$ to $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$, the path running along $\mathcal{E}$ from $\mathcal{L}_{twice}$ to $\mathcal{L}_{closest}$. $\mathcal{L}_{closest}$ is the landmark region incident to the point on $\mathcal{E}$ that is closest to the current goal, $Pt_{closest}$. Path $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$ is determined via a check on Line 13. *AvoidPathLoop* iterates over all facets $\mathcal{F}$ neighboring the final landmark region on $\mathcal{P}$, Line 12, where $\mathcal{P} \in \boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$. For each facet, a check is performed to determine if any point on the facet is closer to the goal than the obstacle surface incident to the current $\mathcal{L}_{closest}$. If it is, $Pt_{closest}$ and path $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$ are replaced. The closet point on $\mathcal{F}$ to $\mathcal{G}$ may not lie on a subfacet of $\mathcal{E}$, i.e. it may lie on an interior point of $\mathcal{F}$. In this case, a landmark region is initialized that is incident to the obstacle surface at the closest point, Line 14. $Pt_{closest}$ may also be used as a certificate of infeasibility once all $\mathcal{F}$ on $\mathcal{E}$ have been explored. A check is performed on the sign of the dot product between vectors, $\vec{v}_{Pt_{closest},\mathcal{G}}$ and $\hat{n}_{Pt_{closest}}$, Line 7. Vector $\vec{v}_{Pt_{closeset},\mathcal{G}}$ represents the vector from the closest point on obstacle surface $\mathcal{E}$ to goal $\mathcal{G}$. Vector $\hat{n}_{Pt_{closest}}$ is the normal vector of the obstacle surface at $Pt_{closest}$. If the dot product between these vectors is negative, the algorithm fails. This method of checking facet visibility to the goal is inspired by the backface culling technique in computer graphics (Breen, Regli, & Peysakhov, 2020).

Subroutine $ExpandNeighbors$ is given in Alg. 5. Given a set of paths $\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}$, each with initial landmark region $\mathcal{L}_{twice}$, the purpose of the subroutine is to generate a new set of paths, $\boldsymbol{\mathcal{P}}'_{\mathcal{L}_{twice}}$ which are not in visited set $\boldsymbol{\mathcal{L}}_{visited}$. This is done by iterating over each path and retrieving the neighbors of its final landmark region, Line 3. If the neighbor $\mathcal{L}$ is not a member of $\boldsymbol{\mathcal{L}}_{visited}$, a new path is created by appending $\mathcal{L}$ to $\mathcal{P}$ to create a new path $\mathcal{P}_{new}$. The set of new paths and updated visited list is returned.

### 5.1.5 Rewiring Complete Paths

An additional optimization regarding the region planner is rewiring the path after a mission complete path has been generated. After exiting *AvoidPathLoop*, the new path may be poor because it travels along the obstacle surface. An example is shown in Fig. 8 where the path

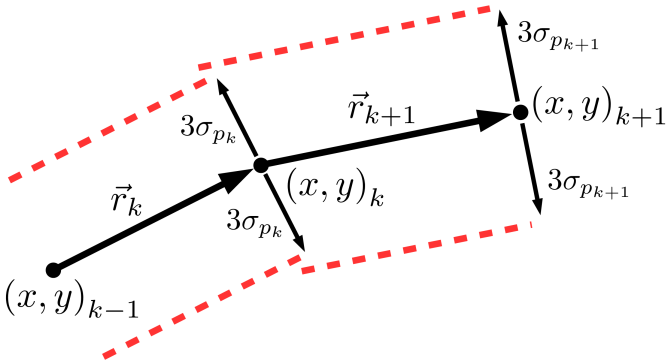



Figure 10: Illustration of the geometry used to validate a trajectory with respect to the chance constraint where the state distribution is assumed Gaussian. The trajectory is checked for collisions within three standard deviations on each side of the $(x, y)$ states. In general, $\sigma_p$ increases with each trajectory knot $k$.

could be improved by removing a number of landmark regions. To prevent poor paths from being passed to the trajectory planner, an attempt is made to rewire complete paths (Line 2 of Alg. 2). The simple subroutine is given in Alg. 6. The routine iterates over all complete paths $\mathcal{P}_{comp}$ in the mission complete $\mathcal{P}_{mc}$. For each complete path, the following components are enumerated: the path's starting region $\mathcal{G}_g$, ending region $\mathcal{G}_{g+1}$, and the set of regions connecting the start and end, $\boldsymbol{\mathcal{R}}$. A* is performed to connect $\mathcal{G}_g$ to $\mathcal{G}_{g+1}$ constrained to the set of regions $\boldsymbol{\mathcal{R}}$. A difference from the vanilla A* is that two regions $\mathcal{R}$ and $\mathcal{R}'$ are considered neighbors if it is possible to generate a collision-free or probabilistically feasible path from $\mathcal{R}$ to $\mathcal{R}'$ (this is checked using a variant of $IsTrajectoryInfeasible$). Consequently, the original path $\mathcal{P}_{comp}$ is replaced with $\mathcal{P}'_{comp}$ where $\mathcal{P}'_{comp}$ has equal or lower heuristic cost (due to having equal or a fewer number of regions connecting $\mathcal{G}_g$ to $\mathcal{G}_{g+1}$).

### 5.1.6 Trajectory Validation

Validation is required for certain chance constraint models to ensure the path is feasible. Specifically, if the chance constraint calculation depends on the path's regions, trajectory validation is necessary post trajectory optimization. The reason is because $PathGenerator$ forms $\mathcal{P}$ using deterministic heuristics; it may be that the risk-bound trajectory is significantly different from the deterministic trajectory. In other words, a trajectory optimized with respect to the constraints implied by $\mathcal{P}$ may not be feasible because the sequence $\mathcal{P}$ was not created with regard to risk. This is true for the CDF-based model presented in Section 7; risk is only evaluated at landmark regions along the path.

The validation subroutine is provided in Alg. 7, *ValidateTrajectory*. The subroutine relies on pairwise validations; the trajectory is iterated over and each pair of states is checked for feasibility. The subroutine $IsTrajectoryInfeasible\,(\boldsymbol{x}_k, \boldsymbol{x}_{k+1})$ checks feasibility between states $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k+1}$. The simplest validation is to check for collisions between the waypoint states $\boldsymbol{\omega}_k$ and $\boldsymbol{\omega}_{k+1}$. Collision checking can also be used to validate stochastic trajectories if a bound on the agent's state distribution is known at future times. For example, consider

the case of a linear dynamical system with Gaussian noise where the discrete-time dynamics are given by:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k = B\mathbf{u}_k + \mathbf{w} \tag{10}$$

where $\mathbf{w}$ is a vector of random variables, $\mathbf{w} \sim \mathcal{N}(0, \mathbf{\Sigma})$. As shown in (Blackmore, Li, & Williams, 2006), the distribution of $\mathbf{x}$ is known at all future times; it is Gaussian with mean $\boldsymbol{\mu}_k$ and covariance matrix $\mathbf{\Sigma}_k$ where

$$\boldsymbol{\mu}_k = \sum_{i=0}^{k-1} A^{k-i-1} B\mathbf{u}_i + A^k \mathbf{x}_0 \tag{11}$$

$$\mathbf{\Sigma}_k = \sum_{i=0}^{k-1} A^{k-i-1} \mathbf{\Sigma} \left(A^T\right)^{k-i-1} \tag{12}$$

Eq. (12) can be used to bound the state distribution at all time steps $k$ and enable trajectory validation. For example, consider the problem of an agent navigating a 2D workspace. In this case, the agent travels between waypoint states $\boldsymbol{\omega}_k = (x, y)_k$ and $\boldsymbol{\omega}_{k+1} = (x, y)_{k+1}$. Let $\vec{r}_k$ be the vector in the direction of travel and $\vec{p}_k$ be a unit vector perpendicular to $\vec{r}_k$. The standard deviation perpendicular to the direction of travel is given by:

$$\sigma_{p_k} = \sqrt{\vec{p}_k^T \mathbf{\Sigma}_k \vec{p}_k} \tag{13}$$

Given $\sigma_{p_k}$, the state distribution can be bounded and validated via collision checking. Because the Gaussian distribution has infinite support, it is practical to bound the trajectory to some constant multiple of $\sigma_{p_k}$. The geometry of the validation process for a system with a Gaussian state distribution is illustrated in Fig. 10.

If a collision is found between trajectory knots $k$ and $k+1$, the complete path $\mathcal{P}$ is split into new prefix and suffix paths, Lines 9 and 10 of Alg. 7. The notation $\mathcal{N}_{pre} = \mathcal{P}_{n \leq k}$ indicates that all regions prior to and including knot $k$ are placed onto the prefix; $\mathcal{N}_{suf} = \mathcal{P}_{n>k}$ indicates that all regions following knot point $k$ are placed onto the suffix. The node is marked as invalid and inserted into the global priority queue, i.e. the queue in Line 1 of Alg. 1.

In addition to discrete-time linear dynamics of the form Eq. 10, it is possible to validate other system types using this strategy if there is an efficient method to compute a bound on the agent's state distribution. This often means making some type of simplifying assumptions. For example, for nonlinear dynamical systems, there is seldom an analytic form for the state distribution. For continuous-time systems, one must decide if the underlying stochastic process is best modeled using discrete or continuous time. Bounds can be generated for certain simple continuous time stochastic processes such as Brownian motions. Rather than rely on Alg. 7 to determine whether a trajectory is feasible with respect to the chance constraint, another possibility is to rely on a better chance constraint model that models risk between regions on $\mathcal{P}$ such as the sampling-based methods in Sections 8 and 9.

### 5.2 Discussion of the Algorithm's Properties

#### 5.2.1 Soundness

For general motion planning problems under uncertainty, soundness with respect to the chance constraint is difficult to prove because of the difficulty modeling the agent's state

distribution and environment geometry. As shown in Section 5.1.6, when the state is linear between collocation steps and stochasticity is additive Gaussian noise, the trajectory can be validated to within a small number $\delta$ of the chance constraint. This follows from the assumption that the state distribution is known at any future time and a bound on its support can be computed. A small number $\delta$ is necessary because the Gaussian has infinite support; even checking a large region around the path does not mean all probability mass has been captured.

### 5.2.2 Termination

Termination of FFHS is described fully in Appendix C. In summary, it is necessary to show that *AvoidPathLoop* does not generate infinitely many loops. Appendix C shows that calls to *AvoidPathLoop* due to the same $\mathcal{L}_{twice}$ will eventually return path $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$. Once this occurs, landmark region $\mathcal{L}_{twice}$ is said to be explored. We then show that under certain geometric constraints on the Polytope to Goal, no two explored landmark regions will form loops. In the case of infeasible problems, the region planner is guaranteed to eventually find the closest point on the polytope obstacle surface that separates $\mathcal{I}$ from $\mathcal{G}$, which causes the check on Line 7 of Alg. 4 to return true and the algorithm to fail. This shows that the region planner will terminate by failing or returning a candidate path to the trajectory planner. A further edge case is whether trajectory planner failure causes the same infeasible path to be passed between the region and trajectory planners such as in the case of a feasible deterministic problem but infeasible stochastic problem. To prevent this, either the trajectory planner or $ValidateTrajectory$ maintains a cache of which paths have caused failure.

### 5.2.3 Completeness

For the deterministic case of a point robot navigating from an initial state to a goal state the algorithm is complete. For stochastic problems, completeness in the general case is very difficult to achieve because path feasibility is dependent on the assumptions underlying the stochastic model. In the deterministic case, completeness follows from the fact that given a set of polytope obstacle surfaces $\boldsymbol{\mathcal{E}}$ that separate $\mathcal{I}$ from $\mathcal{G}$, *AvoidPathLoop* will eventually find the closest point on each $\mathcal{E} \in \boldsymbol{\mathcal{E}}$ to the goal if it does not first fail or find the goal. The landmark region incident to the closest point will be added and popped from the *PriorityQueue*. The search will continue until Alg. 3 pops a path from the queue whose final element is able to shoot unobstructed to the goal. We discuss completeness and provide a proof in Appendix C.

### 5.2.4 Complexity

We analyze the complexity of First Feasible Hybrid Search treating the trajectory planner, Line 5 of Alg. 1, as a black box. The complexity of the trajectory planner is dependent on the underlying trajectory optimization routine and model of uncertainty. Our focus is on Alg. 2, *PathGenerator*. *PathGenerator* shares worst case complexity with state space search, $\mathcal{O}\left(b^d\right)$, where $b$ is the branching factor and $d$ is path depth. For 2D environments composed of polygon obstacles, the branching factor is two. Each node, whether expanded via the *PolytopeToGoal* or *AvoidPathLoop* methods has two potential children. For 3D

environments with triangulated polyhedron obstacles, the branching factor is three when a child is expanded via the *PolytopeToGoal* method. Because search is exponential in the depth of the tree, longer paths should be avoided. This is the motivation for the *PolytopeToGoal* method, which attempts to bypass irrelevant landmark regions. Subroutine *AvoidPathLoop* potentially creates longer paths. In the case of triangulated polyhedral surfaces, we can write a simple bound for the number of paths originating at $\mathcal{L}_{twice}$ and added to $PriorityQueue_{\mathcal{L}}$. At the first level, $\mathcal{L}_{twice}$ has four neighbors. At every subsequent call to $ExpandNeighbors$, Line 5, each landmark region has at most two neighbors that are not already in $\boldsymbol{\mathcal{L}}_{visited}$. This results in an upper bound on the number of paths:

$$\sum_{k=1}^{Dist} 2^{k+1} \tag{14}$$

where $Dist$ is the maximum distance (number of facets) separating $\mathcal{L}_{twice}$ from any other obstacle sub-facet sharing the same polytope surface. For a general polyhedron, $Dist$ is difficult to compute but can be bounded via Hirsch's Conjecture as $|\boldsymbol{\mathcal{F}}| - d$, the number of facets minus the polytope's dimension (Santos, 2012). For triangulated polyhedra, the number of landmark regions is bounded by $3|\boldsymbol{\mathcal{F}}|$ and $d = 3$. Therefore, the number of paths originating from $\mathcal{L}_{twice}$ via calls to *AvoidPathLoop* is bounded by:

$$|\boldsymbol{\mathcal{P}}_{\mathcal{L}_{twice}}| = \sum_{k=1}^{3|\boldsymbol{\mathcal{F}}|-3} 2^{k+1} \tag{15}$$

where $\boldsymbol{\mathcal{F}}$ in Eq. 15 refers to facets on the obstacle polyhedron to which $\mathcal{L}_{twice}$ belongs. Eq. 15 shows the algorithm's dependence on $|\boldsymbol{\mathcal{F}}|$; the algorithm will be challenged by missions with many obstacle concavities (i.e. *AvoidPathLoop* is called often) where concavities have complex surface geometry, e.g. $|\boldsymbol{\mathcal{F}}|$ is large.

#### 5.2.5 Optimality

First feasible hybrid search is greedy and does not provide a guarantee on finding the optimal solution. The goal of the algorithm is to compute good trajectories quickly. If the use case demands higher quality solutions than the greedy solution, it is trivial to make FFHS into an anytime algorithm where the best solution is recorded as an incumbent and the search continues until the priority queue is exhausted. We provide experimental results in Section 10.3.3 that illustrate FFHS's performance against an interleaved hybrid search that takes a strategy akin to A*, i.e. for every new region added to a path, a trajectory optimization is performed to evaluate the cost of each partial path.

Similar to completeness, optimality is difficult to prove for motion planning under uncertainty because of the required simplifications when computing expected cost. Most approaches consider only deterministic cost functions (Blackmore et al., 2011) or simple cost functions for which the expected cost is straightforward to compute, i.e. the linear quadratic Gaussian (LQG) controller (Ono et al., 2013). Even if simple cost functions are assumed, the effect of colliding with obstacles between collocation knots is almost always ignored.

## 6. The Trajectory Planner

Given a path from the region planner, the trajectory planner computes a continuous state and time trajectory that satisfies the ccTPM. Because it deals with continuous decision variables rather than discrete, this problem is solved using a different strategy than the region planner. Whereas the region planner relies on graph search, the trajectory planner makes use of off-the-shelf optimization libraries.

The difference between the problem statement in Section 3 and the problem solved by the trajectory planner is the addition of the region constraints:

$$\boldsymbol{\omega}_n \in \mathcal{R}_n \ \ \forall \ \ \mathcal{R}_n \in \mathcal{P} \tag{16}$$

These constraints, generated by the region planner, make it easier to formulate and solve the trajectory planner's underlying trajectory optimization. In the case of linear dynamical systems and Gaussian uncertainty, Constraints (16) allow the trajectory planning problem to be formulated as a convex program. For nonlinear dynamical systems with more complex risk models, the trajectory planning problem is formulated as a nonlinear program. Even in these cases, Constraints (16) aid in generating feasible guesses for the decision variables, which are often necessary for solver convergence.

The process of transforming a continuous state trajectory optimization into a mathematical program with a finite number of decision variables is termed transcription. The trajectory optimizations in this article are transcribed into nonlinear programs and solved as a sequential quadratic program (SQP). SQPs enable us to consider a wide variety of dynamical systems and solve the three chance constraint models in Sections 7 - 9. In spite of their expressiveness, SQPs are not without shortcomings. Specifically, SQPs do not, in general, reach global optima as convex programs do. Because our strategy is to seek good feasible solutions quickly rather than optimal ones, this is not a major drawback. Second, SQPs require initial guesses for their decision variables, which must typically be feasible or close to feasible. This is a more significant drawback as initial guesses are oftentimes hard to generate for complex systems. Strategies to overcome this include maintaining a cache of previously solved trajectories and using these as guesses for future optimizations. This article relies on the SNOPT SQP solver (Gill, Murray, Saunders, & Wong, 2017). While certain details of our implementation are specific to SNOPT, most are shared among SQP libraries. The user defines an SQP by providing the solver with access to a function that evaluates the objective and constraints and (optionally) a function that evaluates the Jacobian matrices of the objective and the constraints. The Jacobian is the gradient of the objective and constraints with respect to each of the decision variables. While the Jacobians may be approximated via finite difference, in general, the algorithm's efficiency is improved if analytic expressions for them are explicitly provided.

To solve the ccTPM via hybrid search, we must transcribe the chance constraint into a form that adequately models the agent's real world trajectory risk while still being tractable for the underlying trajectory optimizer. In Sections 7 through 9, we present three methods of approximating Constraint (3) of increasing complexity. Section 7 details a CDF-based model appropriate for linear systems with Gaussian uncertainty. Section 8 describes a sampling-based method when agent geometry is important to consider and Section 9 presents Shooting Method Monte Carlo, which is appropriate for nonlinear and non-Gaussian systems. The

CDF-based and sampling-based models of Sections 7 and 8 rely on collocation trajectory optimization. Collocation is a general method for trajectory optimization that approximates a trajectory via discretization at knot points. Trajectories $\mathcal{T}$ take the form:

$$\mathcal{T}_{collocation} = [\mathbf{x}_0, \mathbf{u}_0, h_0, \mathbf{x}_1, \mathbf{u}_1, h_1, \ldots, \mathbf{x}_k, \mathbf{u}_k, h_k, \ldots, \mathbf{x}_K, \mathbf{u}_K] \tag{17}$$

where states $\mathbf{x}_k$, control inputs $\mathbf{u}_k$, and time steps $h_k$ are decision variables indexed by collocation knot $k$. The third model, Shooting Method Monte Carlo, relies on the shooting method. Shooting methods involve simulating a choice of control inputs and iteratively correcting the resulting trajectory until it converges to a desired goal. Shooting trajectories take the form:

$$\mathcal{T}_{shooting} = [\mathbf{u}_0, h_0, \mathbf{u}_1, h_1, \ldots, \mathbf{u}_k, h_k, \ldots, \mathbf{u}_K] \tag{18}$$

where states $\mathbf{x}_k$ are determined via integrating the dynamics given $\mathbf{u}_k$ and $h_k$. Because the focus of this article is on the chance constraint, we leave other details of our trajectory optimization to Appendix A for collocation trajectory optimization and Appendix B for the shooting method.

Recall from Subsection 3.1 that the trajectory planner receives a path $\mathcal{P}$ from the region planner. It also requires specification of the agent's equations of motion, an objective function, and a risk bound $\epsilon$. The trajectory planner outputs a trajectory $\mathcal{T}$ and an objective value, $J$. These are passed back to the region planner for trajectory validation and possibly comparison with other trajectories.

## 7. CDF-Based Chance Constraints

The chance constraint models the probability that the solution to the ccTPM fails in execution. This article focuses on failure due to collision with an obstacle. Mathematically, this is described by Constraint (3), repeated here:

$$P\left(\bigwedge_{k \in K} \mathbf{x}_k \in \mathcal{W}_{free}\right) \geq 1 - \epsilon \tag{19}$$

The constraint is difficult to compute because it involves the integration of a probability distribution over possibly many dimensions as the state evolves in time. Simplifications are almost always made concerning the underlying stochastic process, the agent's state distribution, and the agent and environment's geometric complexity. This section focuses on CDF-based chance constraints. This is a simplification appropriate to agents with linear dynamics whose geometry is unimportant when computing collision risk, i.e. point robots. Although the agent's waypoint state distribution is assumed Gaussian, this method is applicable for any distribution with an easy-to-compute cumulative distribution function. While the notion of easy-to-compute is somewhat arbitrary, the inclusion of the single and multivariate Gaussian CDF in many popular numerical libraries and the speed with which it is computed enables us to classify it in this category (Jones, Oliphant, Peterson, et al., 2001).

The CDF-based model is most appropriate for longer trajectories where the emphasis is on computing trajectories quickly that coarsely approximate the true collision risk. Computing the mission paths for underwater gliders provide a good motivation for this strategy.

In certain circumstances, there is a very poor knowledge of the probability distributions followed by ocean currents. A coarse model of risk can be as accurate as a very refined model whose underlying stochastic process does not match the system's true stochastic process. The coarse model is much less computationally-intensive.

Eq. (19) is termed a *joint* chance constraint because it involves the state at various time steps. The first simplification for CDF-based methods is to make an assumption about the stochastic process' interdependence at different time steps. This allows the intractable joint chance constraint to be broken into simpler individual chance constraints. One method to approximate the joint chance constraint is through independence; under independence assumptions the probability of trajectory success is the product of the individual probabilities. Similar models have been discussed in other work such as (van den Berg et al., 2011). In the formulation used in our present work, the probability of success at landmark region $n$ is modeled by conditioning on the success of past transitions and the law of total probability:

$$P\left(S_n\right) = P\left(S_n | S_{n-1}, \ldots, S_1\right) P\left(S_{n-1} | S_{n-2}, ..., S_1\right) \ldots P\left(S_1\right) \tag{20}$$

where $S_n$ is the event of the agent succeeding at landmark region $n$. Each term is assumed Markovian: the probability of success at region $n$ is conditionally independent of success at $n-2$ given its success at $n-1$. Using this fact, (20) can be simplified:

$$P\left(S_N\right) = P\left(S_n | S_{n-1}\right) P\left(S_{n-1} | S_{n-2}\right) \ldots P\left(S_1\right) \tag{21}$$

The independence assumption is used to simplify Constraint (19) into a product involving each success event:

$$P\left(S_N\right) = \prod_{n=1}^{N} P\left(S_n | S_{n-1}\right) \geq 1 - \epsilon \tag{22}$$

Eq. (22) simplifies the joint chance constraint computation into the computation of individual chance constraints. The individual chance constraint is the probability of success at a specific point in time. Mathematically, it is given by the integral:

$$P\left(S_n\right) = \int_{\ell} f\left(\mathbf{x}_n\right) d\mathbf{x}_n \tag{23}$$

where $f\left(\mathbf{x}_n\right)$ is a distribution of the agent's state at region $n$ and $\ell$ represents the limits of integration. In general, Eq. (23) must be computed using numerical methods; however, the central assumption of this section is that we have access to a subroutine that can quickly evaluate the integral. What must be specified are the limits of integration $\ell$.

An advantage of the region planner's landmark regions is that they form not only constraints on waypoint states but also enable a geometric simplification for computing CDF-based chance constraints. Without a geometric simplification, the limits of integration to compute Eq. (23) are often non-convex. Consider the bivariate state distribution in the left pane of Fig. 11; the distribution encompasses the triangular obstacle. Instead of computing the integral with respect to the non-convex space, it is simplified by projecting the distribution onto the landmark region. This model simplifies the chance constraint; in place of computing $P\left(\mathbf{x}_n \in \mathcal{W}_{free}\right)$, the approximation $P\left(\mathbf{x}_n \in \mathcal{L}_n\right)$ is computed. The latter term is easier to compute because $\mathcal{L}_n$ is convex.

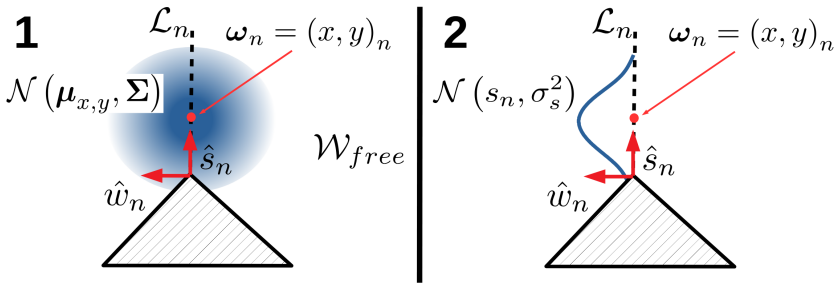



Figure 11: Illustration of the variables, projection, and bounds relevant to the risk computation at each waypoint for a CDF-based chance constraint in a 2D workspace. The left image represents the bivariate state distribution centered at the waypoint $\boldsymbol{\omega}_n$. The right image depicts the projection of the distribution onto the landmark region $\mathcal{L}_n$. To model the risk at the waypoint, the single variate distribution $\mathcal{N}(s_n, \sigma_n^2)$ is integrated from $s_n = 0$ to $s_n = \infty$.

The integral is computed as follows. Recall from Section 4.1 in 2D, the waypoint states $\boldsymbol{\omega} = (x, y)$ are projected onto a landmark region $\mathcal{L}_n$ with orthogonal basis $(\hat{s}, \hat{w})$. Coordinate $\hat{s}$ runs parallel the landmark region ray and $\hat{w}$ is orthogonal to the ray. The distribution of the agent's waypoint states is projected onto the landmark region and integrated along the $\hat{s}$-coordinate. The mean of the projected distribution is the value of the $s_n$ coordinate, introduced in Section 4.1. Intuitively, $s_n$ represents the agent's distance from the obstacle.

The projected distribution is integrated along the landmark region from $s_n = 0$ to $\infty$. This construction is illustrated in Fig. 11. This idea of computing the chance constraint by integrating with respect to the agent's distance from an obstacle is used in a similar manner in (Blackmore et al., 2011). The integration required to evaluate the chance constraint at a single waypoint, the probability of success $P(S_n)$, is:

$$P(S_n) = \frac{1}{\sqrt{2\pi}\sigma_s} \int_0^\infty \exp\left(\frac{-(s - s_n)^2}{2\sigma_s^2}\right) ds = 1 - \Phi\left(\frac{-s_n}{\sigma_s^2}\right) \tag{24}$$

where $\Phi$ is the cumulative distribution function of the standard normal. Eq. (24) is used with Eq. (22) to evaluate the joint chance constraint.

### 7.1 Multivariate Gaussians in 3D Workspaces

Under the independence assumption discussed above, the joint chance constraint for 3D workspaces can be transformed into individual chance constraints:

$$\bigwedge_{n \in N} P(\mathbf{x}_n \in \mathcal{W}_{free}) = \prod_n \Phi_{\Sigma,\boldsymbol{\ell}} \geq 1 - \epsilon \tag{25}$$

where $\Phi_{\Sigma,\ell}$ is the cumulative distribution function of the multivariate Gaussian distribution with correlation matrix $\Sigma$ integrated with respect to bounds $\boldsymbol{\ell}$. While this integral is difficult to evaluate, very efficient numerical integration routines have been implemented such as those described in (Genz, 1992). In terms of $(r, s)$ variables, the constraint may be

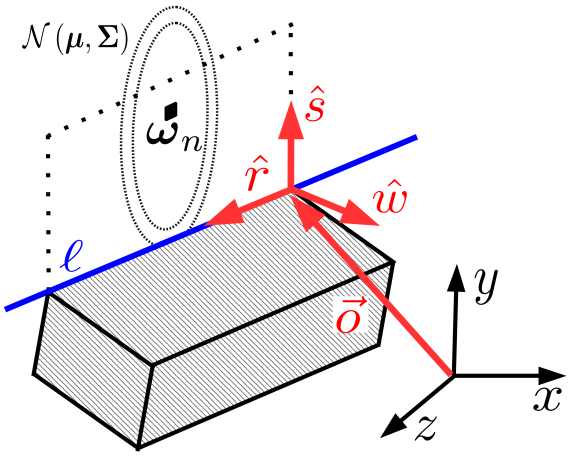



Figure 12: Illustration of the projection of the state distribution onto the landmark region for a 3D workspace. The dotted ellipses indicate the projection of the multivariate Gaussian onto the plane. The distribution is then integrated from the line $\ell$.

written using the bivariate normal distribution as (Weisstein, 2018):

$$\Phi_{\Sigma,\boldsymbol{\ell}} = \int_{-\infty}^{\infty}\int_{0}^{\infty} \frac{1}{2\pi\sigma_s\sigma_r\sqrt{1-\rho^2}} \exp\left(-\frac{z}{2\left(1-\rho^2\right)}\right) dsdr \tag{26}$$

$$z = \frac{\left(s-s_n\right)^2}{\sigma_s^2} - \frac{2\rho\left(s-s_n\right)\left(r-r_n\right)}{\sigma_s\sigma_r} + \frac{\left(r-r_n\right)^2}{\sigma_r^2} \tag{27}$$

where $\rho$ is the correlation coefficient. In this expression $s_n$ and $r_n$ are the decision variables determined by the optimizer (i.e. the waypoints) after being transformed from the global coordinates $\mathbf{x}_n = (x, y, z)_n$ using transformation Eqs. (6) - (8). Similar to the 2D case; the lower bound of the inner integral is at $s = 0$, i.e. the line $\ell$ as shown in Fig. 12.

## 8. Sampling-Based Chance Constraints

While the approach presented in the previous section provides a simple method to approximate trajectory risk, it is in many ways inadequate to model more complex systems. There was assumed access to a quickly-evaluated cumulative distribution function. It was also assumed that the agent was modeled well as a point robot. In this section, we discuss an approach that relaxes these assumptions. We propose a method that models trajectory risk for agents with rigid bodies and for which we do not have access to an easy-to-evaluate cumulative distribution $F$ but only a distribution $f$ from which we can draw samples. To do this, a Monte Carlo sampling approach is used.

Computing the collision probability involves integrating a multivariate density with respect to limits $\ell$, which are a function of the agent and obstacles' geometry. Computing these limits is challenging because it is non-trivial to compute whether a given agent configuration is in collision with an obstacle. Consider the rectangular polygon that models a simple car as shown in Fig. 13. The configuration states of the rectangle are $q = (x, y, \theta)$.

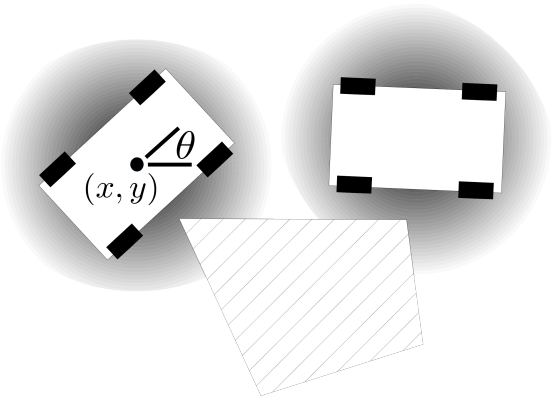



Figure 13: Computing the collision risk of even a simple rectangular shape is challenging. A possible distribution of the rectangular car's state is shown in gray. Not only is there uncertainty in states $(x, y)$ but also orientation $\theta$. An integration to compute the collision risk must be performed over all configurations that collide.

Additionally, assume that the probability distribution describing these states is uniform, $f(q) \sim \mathcal{U}$, in some set $(0 \leq x \leq a, 0 \leq y \leq b, 0 \leq \theta < 2\pi)$:

$$f(x, y, \theta) \begin{cases} \frac{1}{2\pi ab}, & \text{for } 0 \leq x < a, 0 \leq y < b, 0 \leq \theta < 2\pi \\ 0, & \text{otherwise} \end{cases} \tag{28}$$

In this case, the probability of success at time step $k$ is:

$$P(S_k) = \frac{1}{2\pi ab} \int_{\ell} \mathbb{1}_{q \in \mathcal{C}_{free}} dq \tag{29}$$

where $\mathcal{C}_{free}$ is the free region of the agent's configuration space. The integral in Eq. (29) is equivalent to determining the volume of the free region of the agent's configuration space over the support of the distribution $f$. In general, this is a very difficult region to determine, not to mention compute its volume (LaValle, 2006).

Instead of attempting to analytically compute the limits of integration, our approach combines collision checking and Monte Carlo simulation. We model trajectory risk for agents with rigid body geometry $\mathcal{B}$ by sampling from its configuration states along a trajectory, determining the number of samples in collision, and using this number to approximate trajectory failure. While Monte Carlo methods have previously been used to model trajectory risk, our innovation is integrating the method with our trajectory planner and enable its use with nonlinear optimizers, such as SQP solvers. Although many efficient algorithms exist for collision checking, (Ericson, 2005), integrating the proposed method with a trajectory optimization subroutine is nontrivial. Collision checks are step functions as a function of the agent's state: for a given state, an agent is either in collision or not. The derivative of a step function is a Dirac delta function and is non-zero only at the point of collision. Because SQP solvers rely on information from the Jacobian matrix in order to satisfy the problem's constraints, simply using a step function in an optimizer does not work well. Rather than using the collision check directly as an indicator of whether an agent configuration is in

collision, we propose the intersecting region $\mathcal{W}_{int}$ between the agent and obstacle. This region provides a continuous measure of whether an agent configuration is in collision and allows us to approximate the chance constraint with a constraint with a non-zero Jacobian over a larger region.

We explain the model in the following subsections. First, we describe the chance constraint model that is a function of the intersecting region between the agent and obstacles. Second, we describe how the intersecting region $\mathcal{W}_{int}$ is computed for 2D and 3D workspaces. Third, we describe how the Jacobian of this constraint is computed. Finally, we mention a shortcoming of this model: the need for a distribution $f$ from which we can sample agent configurations.

### 8.1 Approximating Trajectory Risk via Intersecting Region

We seek to approximate the original chance constraint, Eq. (3), for agents with rigid body geometry:

$$P\left(\bigwedge_{k\in K} \mathcal{W}_{a,k} \in \mathcal{W}_{free}\right) \geq 1-\epsilon \tag{30}$$

where $\mathcal{W}_{a,k}$ represents the region of the workspace occupied by an agent with rigid body $\mathcal{B}$ at trajectory step $k$. Given a trajectory $\mathcal{T}$, we draw samples from this trajectory that approximate the agent's configuration state distribution. How these samples are drawn is discussed in Section 8.4. For now, we assume that we are able to generate a sequence of agent body configurations $\mathcal{W}_{a,k,s}$. Variable $s$ indexes the sampling scenario. If $\mathcal{W}_{obs}$ represents the workspace occupied by the obstacles, let $\mathcal{W}_{int,s}$ be the intersecting region between the agent and obstacles summed over the entire trajectory scenario. Specifically:

$$\mathcal{W}_{int,s} = \sum_k \mathcal{W}_{a,k,s} \cap \mathcal{W}_{obs} \tag{31}$$

To model the collision probability, an indicator variable is set to one if the intersecting region is greater than zero, and zero otherwise:

$$P\left(\text{Collision}\right) \approx \frac{1}{S}\sum_s \mathbb{1}_{|\mathcal{W}_{int,s}|>0} \tag{32}$$

where $|\mathcal{W}_{int,s}|$ is the size of the intersecting region of scenario $s$ and $S$ is the total number of scenarios. The Jacobian of the indicator function is a Dirac delta function; Eq. (32) has a non-zero gradient only at the point of collision between the agent and the obstacle. This provides the optimizer with little information about how to satisfy the constraint and would likely prevent the optimizer from succeeding. Instead, this paper approximates the indicator variable through a sigmoid function. The sigmoid has seen prior use in non-convex optimization problems to approximate indicator or barrier functions (Freund, 2004). The general form of the sigmoid is:

$$\text{Sig}_\alpha\left(s\right) = \frac{1}{1+\exp\left(-\alpha s\right)} \tag{33}$$

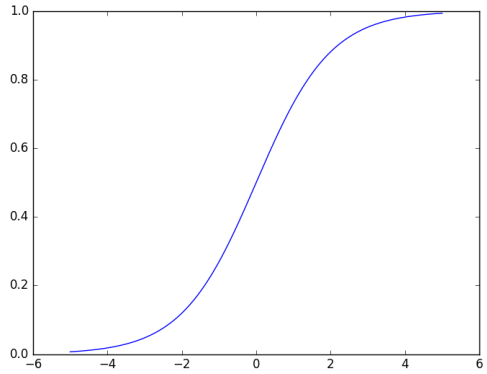



Figure 14: The shape of a general sigmoid function

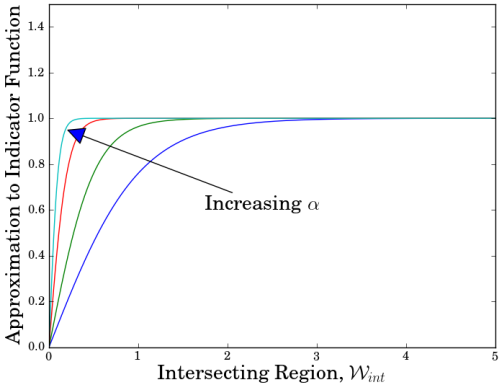



Figure 15: The modification of the sigmoid function used to approximate $\mathbb{1}_{|\mathcal{W}_{int}|>0}$. Increasing $\alpha$ improves the approximation.

where $\alpha$ is a parameter that dictates the steepness of the slope. A plot of this function is shown in Fig. 14. The figure illustrates how the sigmoid approximates an indicator variable; negative arguments model the case $\mathbb{1}_A = 0$ and positive arguments model the case $\mathbb{1}_A = 1$. The sigmoid becomes a better approximation to the indicator function as $\alpha$ grows large.

Eq. (33) must be modified to model the collision constraint. The intersecting region cannot be negative and the function should be zero at $|\mathcal{W}_{int}| = 0$. This suggests using a sigmoid of the form:

$$\text{Sig}_\alpha(s) = A_2\left(\frac{1}{1+\exp(-\alpha s - A_1)} - A_0\right) \tag{34}$$

where $A_0$, $A_1$, and $A_2$ are constants that provide the required offset and scaling. This function is shown in Fig. 15. The indicator function of Eq. (32) is approximated as:

$$\mathbb{1}_{|\mathcal{W}_{int}|>0} \approx \text{Sig}_\alpha(|\mathcal{W}_{int}|) = A_2\left(\frac{1}{1+\exp(-\alpha|\mathcal{W}_{int}| - A_1)} - A_0\right) \tag{35}$$

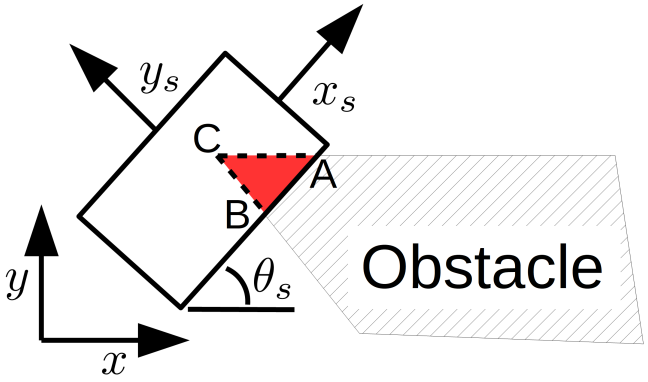



Figure 16: The intersecting region $\mathcal{W}_{int}$ is the triangle defined by $ABC$. The dashed lines represent interface facets; they are used to compute the partial derivatives of the change in intersection area as a function of the decision variables.

and the collision probability approximated as:

$$P\left(\text{Collision}\right) \approx \frac{1}{S}\sum_{s}\text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \tag{36}$$

The overall chance constraint, denoted $C_{cc}$, is written:

$$\epsilon - \frac{1}{S}\sum_{s}\text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \geq 0 \tag{37}$$

An advantage of the sigmoid function is that it is straightforward to differentiate when computing the Jacobian matrix. Specifically,

$$\frac{d\text{Sig}_{\alpha}\left(s\right)}{ds} = \text{Sig}_{\alpha}\left(s\right)\left(1-\text{Sig}_{\alpha}\left(s\right)\right) \tag{38}$$

The chain rule is used to derive the gradients with respect to the configuration states. The Jacobian of the chance constraint $C_{cc}$ with respect to a configuration state $x_n$ is:

$$\frac{\partial C_{cc}}{\partial x_n} = \frac{d\text{Sig}_{\alpha}\left(\mathcal{W}_{int}\right)}{d\mathcal{W}_{int}}\frac{\partial \mathcal{W}_{int}}{\partial x_n} \tag{39}$$

One downside to the sigmoid approximation is tuning $\alpha$. As shown in Fig. 15, increasing $\alpha$ enables a better approximation to the collision check. However, this also makes the derivative nonzero over a smaller region and makes it more difficult to integrate into a sequential quadratic program. We discuss the impact of $\alpha$ on the chance constraint model in Section 8.5.

### 8.2 Computing the Intersecting Region, $\mathcal{W}_{int}$

In two dimensions, the intersection of two polygons produces another polygon or, if the intersection occurs at only one point, a point. Computing the intersection is termed *clipping*. In general, computing clippings for general polygons is difficult. In spite of this,

many efficient algorithms have been proposed for general polygons such as Vatti's clipping algorithm, presented in (Vatti, 1992), and an implementation in (Johnson, 2014).

In three dimensions, computation of the intersecting volume is more complicated. Algorithms exist to compute the intersecting volume between two convex polyhedra such as described in (Powell & Abel, 2015). This clipping algorithm decomposes the subject and clipping polyhedra into planar graphs representing vertices and edges. Clipping occurs by searching through the vertices of the subject graph and clipping vertices that lie on one side of a clipping plane. Because requiring obstacles to be convex would limit our application, we discuss a method to approximate $\mathcal{W}_{int}^3$ for convex agent geometries but non-convex obstacles in Appendix D.

### 8.3 Generating the Collision Jacobians

SQP optimization routines are made more efficient by providing analytic derivations of the constraint Jacobians; i.e. the partial derivatives of each constraint with respect to the decision variables. An advantage of modeling the chance constraint using intersecting regions is that the constraint's gradients are relatively straightforward to compute using techniques from vector calculus. Given two intersecting polytopes, it is necessary to compute the change in the intersecting region with respect to the optimization's decision variables. Specifically, we must determine $\frac{\partial \mathcal{W}_{int}}{\partial x_n}$ in Eq. (39).

Only the configuration states affect the agent's position in the workspace. Namely, it is necessary to compute $\partial \mathcal{W}_{int}/\partial x_i$ where $x_i$ is a configuration state. For the 2D rectangular agent this means computing:

$$\frac{\partial A_{int}}{\partial x_k}, \frac{\partial A_{int}}{\partial y_k}, \frac{\partial A_{int}}{\partial \theta_k} \;\; \forall \;\; k \in K \tag{40}$$

where $A_{int}$ is the intersecting area. For the rigid body polyhedral agent whose orientation is described by Euler angles $(\psi, \theta, \phi)$, it is necessary to compute:

$$\frac{\partial V_{int}}{\partial x_k}, \frac{\partial V_{int}}{\partial y_k}, \frac{\partial V_{int}}{\partial z_k}, \frac{\partial V_{int}}{\partial \psi_k}, \frac{\partial V_{int}}{\partial \theta_k}, \frac{\partial V_{int}}{\partial \phi_k} \;\; \forall \;\; k \in K \tag{41}$$

where $V_{int}$ is the intersecting volume.

The insight into evaluating these gradients is that they can be computed by integrating the flux $\Phi$ over certain facets of $\mathcal{W}_{int}$. Flux is an idea from physics that models how much of a substance is flowing through a surface. The total amount of transfer over the surface is computed by integrating a vector field with respect to the surface (Anton, 1999). In this circumstance, flux can be used to compute the amount of incremental area (or volume) entering or leaving $\mathcal{W}_{int}$ due to an incremental change in one of the configuration states. This is exactly the quantity expressed by terms (40) and (41). Before describing the integration, we state a definition:

**Definition 8.1.** An *interface facet* is a facet of $\mathcal{W}_{int}$ that lies at the interface between regions $\mathcal{W}_{int}$ and $\mathcal{W}_a \setminus \mathcal{W}_{int}$

Intuitively, incrementally changing a configuration state could generate an incremental region flux across each interface facet. This produces a change in $\mathcal{W}_{int}$. This is shown for the rectangular agent in collision in Fig. 16. For each configuration state $(x, y, \theta)$, the

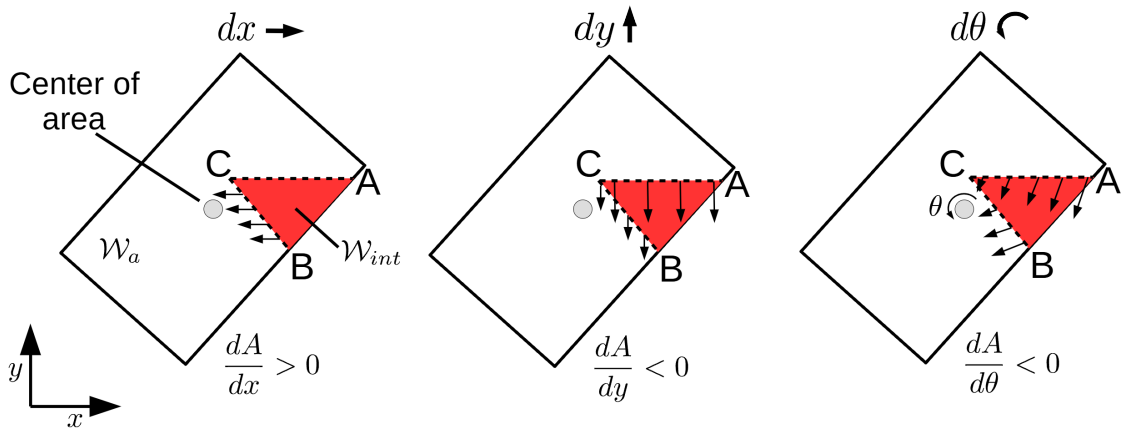



Figure 17: Examples of collision gradients for a 2D rectangular agent. Decision variables $(x, y)$ are assumed located at the center of area; rotations $\theta$ are around the center of area. Arrows indicate the direction that the interface facets $\overline{BC}$ and $\overline{AC}$ change due to an incremental change in one of the configuration states. They are opposite the flux. Arrows extending into the white region ($\mathcal{W}_a$) symbolize flux is heading *into* $\mathcal{W}_{int}$, i.e. the gradient $\partial\mathcal{W}_{int}/\partial$State is increasing. Arrows extending into the red region $\mathcal{W}_{int}$ indicate that flux is coming *out of* $\mathcal{W}_{int}$, i.e. the gradient $\partial\mathcal{W}_{int}/\partial$State is decreasing.

flux is computed using a line integral for each interface facet of the intersecting area. For example, the triangle $ABC$ in Fig. 16 represents the intersecting area. Only segments $\overline{BC}$ and $\overline{AC}$ represent interface facets. This is because segment $\overline{AB}$ is entirely embedded in $\mathcal{W}_{obs}$; an infinitesimal change in $\overline{AB}$ does not affect $\mathcal{W}_{int}$. On the other hand, $\overline{BC}$ and $\overline{AC}$ separate $\mathcal{W}_{int}$ and $\mathcal{W}_a \setminus \mathcal{W}_{int}$. An infinitesimal change in their position will cause an infinitesimal change in $\mathcal{W}_{int}$. The total flux (the total gradient) is the sum of the fluxes over each interface facet $i$:

$$\frac{\partial \mathcal{W}_{int}}{\partial x} = \Phi_x = \sum_i \Phi_{x,i} \tag{42}$$

where $\Phi_x$ means the flux generated by an incremental change in state $x$. Flux is computed through integration. In $\mathcal{W} \in \mathbb{R}^3$, this is a surface integral:

$$\Phi_i = \int\int_i \mathbf{F}(x, y, z) \cdot \mathbf{n}(x, y, z)\, ds \tag{43}$$

where $\mathbf{F}$ is a function that describes the rate of change of points $(x, y, z) \in \mathcal{W}_{int}$ due to changes in the configuration state. The surface normal $\mathbf{n}(x, y, z)$ is constant across the facets of $\mathcal{W}_{int}$. The integral is carried out over the interface facet $i$. For $\mathcal{W} \in \mathbb{R}^2$, the flux is computed in a similar manner except that Eq. (43) is a line integral. The following sections provide some examples computing Eq. (43) for the test cases.

### 8.3.1 Generating gradients for $\mathcal{W}^2$

Computing the collision gradients for the polygonal 2D agent requires integrating Eq. (43) as a line integral over edges that are interface facets. An illustration of these computations is shown in Fig. 17.

The $x$ and $y$ gradients are straightforward to compute:

$$\Phi_{x,i} = \left.\frac{dA}{dx}\right|_i = -\int_i \hat{n}_x di = -\hat{n}_x \ell_i \tag{44}$$

$$\Phi_{y,i} = \left.\frac{dA}{dy}\right|_i = -\int_i \hat{n}_y di = -\hat{n}_y \ell_i \tag{45}$$

where subscript $i$ represents the $i^{\text{th}}$ interface facet. Variables $\hat{n}_x$ and $\hat{n}_y$ represent the $x$ and $y$ components of outward facing, unit-length normals. Because neither $\hat{n}_x$ nor $\hat{n}_y$ change over the edge, the normals may be taken outside the integrals. This results in the integral simplifying to the normal multiplied by the length of the edge, $\ell_i$.

The gradient with respect to $\theta$ is more difficult to compute because the flux due to rotation varies over the edge. The gradient with respect to a change in $\theta$ is:

$$\left.\frac{dA}{d\theta}\right|_i = \int_i \left( (c_y - y)\,\hat{n}_y + (c_x - x)\,\hat{n}_x \right) ds \tag{46}$$

where $(c_x, c_y)$ represent the agent's center of rotation, $(x, y)$ represent $x$ and $y$ points along the line, and $s$ is a parameter that denotes position along the interface facet. To evaluate the line integral along the interface facet $i$, it is necessary to rewrite $x$ and $y$ in terms of $s$. If the vertices of edge $\overline{AC}$ are $(x_0, y_0)$ and $(x_1, y_1)$, $x$ and $y$ can be written in terms of parameter $s$ as: $x(s) = x_0 + s(x_1 - x_0)$ and $y(s) = y_0 + s(y_1 - y_0)$ where $0 \leq s \leq 1$. This parameterization allows the integral to be rewritten in terms of $s$:

$$\left.\frac{dA}{d\theta}\right|_i = \int_0^1 \left(c_y - y_0 - s y_d + c_x - x_0 - s x_d\right) \sqrt{(x_d)^2 + (y_d)^2}\, ds \tag{47}$$

where $x_d = x_1 - x_0$ and $y_d = y_1 - y_0$ and the fact was used that $di = \sqrt{(dx/ds)^2 + (dy/ds)^2}\, ds$. The integral in Eq. (47) is straightforward to evaluate.

Given the gradients with respect to each facet, Eqs. (44), (45), and (47), the total gradient is given by their sum over all interface facets:

$$\frac{dA}{dx} = \sum_i \left.\frac{dA}{dx}\right|_i \tag{48}$$

$$\frac{dA}{dy} = \sum_i \left.\frac{dA}{dy}\right|_i \tag{49}$$

$$\frac{dA}{d\theta} = \sum_i \left.\frac{dA}{d\theta}\right|_i \tag{50}$$

### 8.3.2 Generating gradients for $\mathcal{W}^3$

The strategy for determining the gradients for the three dimensional case is similar, although computations are more complicated. The intersecting region $\mathcal{W}_{int}$ is now a polyhedron and the facets between the obstacle and the agent are polygons. Surface integrals must be used to compute the flux. In the case of a polyhedron, gradients must be computed with respect to each of the six states in (41). Similar to the 2D case, the position gradients are simpler

to compute than the orientation gradients. Let $\mathcal{F}_i$ represent the interface facet. The $x$ gradient is:

$$\left.\frac{dV_{int}}{dx}\right|_i = -\int_{\mathcal{F}_i} \hat{n}_x ds = -\hat{n}_x A_i \tag{51}$$

where the left-hand side indicates the change in volume $V_{int}$ of $\mathcal{W}_{int}$ with respect to the change in $x$, $\hat{n}_x$ is the outward facing normal of the $i^{\text{th}}$ interface facet, and $A_i$ is the facet's area. As in the 2D case, the flux is constant over the surface so $\hat{n}_x$ can be pulled outside the integral. Similar equations hold for the $y$ and $z$ directions.

For orientation, the surface integrals require more effort. We want to compute the following quantities with respect to interface facet $\mathcal{F}_i$:

$$\left.\frac{\partial V_{int}}{\partial \psi}\right|_i, \left.\frac{\partial V_{int}}{\partial \theta}\right|_i, \left.\frac{\partial V_{int}}{\partial \phi}\right|_i \tag{52}$$

The strategy behind our approach is to derive an expression for the rate of change of points on $\mathcal{F}_i$ due to changes in Euler angles $(\psi, \theta, \phi)$ and integrate this expression over the surface $\mathcal{F}_i$. To derive the rate of change of points on $\mathcal{F}_i$, we begin with the equation for the velocity of a point on a rigid body due to a rotation $\vec{\omega}$: $\vec{v}_p = \vec{\omega} \times \vec{r}$. This allows us to write the integral, Eq. (43):

$$\Phi_i = \int\int_i \mathbf{F}(x,y,z) \cdot \mathbf{n}(x,y,z)\, ds \tag{53}$$

$$= \int\int_i \vec{v}_p \cdot \mathbf{n}(x,y,z)\, ds \tag{54}$$

$$= \int\int_i (\vec{\omega} \times \vec{r}) \cdot \mathbf{n}(x,y,z)\, ds \tag{55}$$

In the following, we specify $\vec{\omega}$ and $\vec{r}$, and provide an expression to evaluate the integral.

Assume $\mathcal{W}_{int}$ has been triangulated such that $\mathcal{F}_i$ is a triangle. Let $(u, v)$ be a coordinate system embedded in $\mathcal{F}_i$ so that the position vector from the agent polyhedron's ($\mathcal{W}_a$) center of volume to a point on $\mathcal{F}_i$ is:

$$\vec{r} = \langle r_{0,x} - r_{c,x} + u\hat{u}_x + v\hat{v}_x, r_{0,y} - r_{c,y} + u\hat{u}_y + v\hat{v}_y, r_{0,z} - r_{c,z} + u\hat{u}_z + v\hat{v}_z \rangle \tag{56}$$

where $r_0 = (r_{0,x}, r_{0,y}, r_{0,z})$ is the point in global coordinates representing the origin of $\mathcal{F}_i$'s local $(u, v)$ coordinate system. Point $r_c = (r_{c,x}, r_{c,y}, r_{c,z})$ is the agent polyhedron's center of volume. Unit vectors $\hat{u} = \langle \hat{u}_x, \hat{u}_y, \hat{u}_z \rangle$ and $\hat{v} = \langle \hat{v}_x, \hat{v}_y, \hat{v}_z \rangle$ are bases for the $(u, v)$ coordinate system, expressed in global coordinates.

Given this expression for $\vec{r}$, the precise form of $\vec{\omega}$ depends on the axis of rotation and the order of rotation. For example, if rotations are performed in order $\phi$, $\theta$, $\psi$ about the $z$ then $y$ then $x$ axes, $\vec{\omega}$ terms take the form:

$$\vec{\omega}_\phi = [0, 0, 1]^T \tag{57}$$

$$\vec{\omega}_\theta = R_\psi^T [0, 1, 0]^T \tag{58}$$

$$\vec{\omega}_\psi = R_\theta^T R_\psi^T [1, 0, 0]^T \tag{59}$$

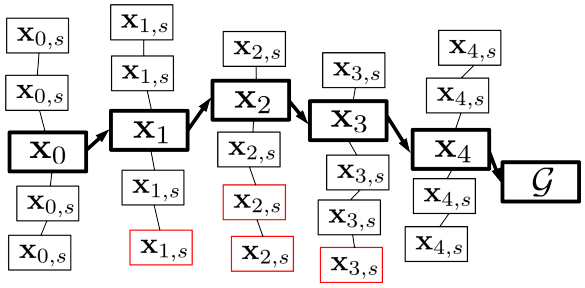



Figure 18: One possible sampling scheme where samples are drawn from the nominal trajectory (shown in bold). Red outlines indicate samples in collision.

where $R_\psi$ and $R_\theta$ are appropriate rotation matrices as defined in (Triantafyllou, 2004). Using these expressions and a simplification discussed in (Anton, 1999), the surface integral can be written:

$$\left.\frac{dV_{int}}{d\psi}\right|_i = \int\int (\vec{\omega}_\psi \times \vec{r}) \cdot (\vec{r}_u \times \vec{r}_v)\, dudv \tag{60}$$

with similar expressions for gradients with respect to angles $\phi$ and $\theta$. The second cross product, $\vec{r}_u \times \vec{r}_v$, is due to the parameterization of $\mathcal{F}_i$ with $(u, v)$, where $\vec{r}_u$ and $\vec{r}_v$ are the partial derivatives of $\vec{r}$ with respect to $u$ and $v$. To simplify the limits of integration, we use the fact that $\mathcal{F}_i$ is a triangle such that (60) becomes:

$$\left.\frac{dV_{int}}{d\psi}\right|_i = \int_0^{v_1}\int_0^{mv+b} (\vec{\omega}_\psi \times \vec{r}) \cdot (\vec{r}_u \times \vec{r}_v)\, dudv \tag{61}$$

where $v_1$, $m$, and $b$ are constant geometric properties of the triangle, i.e. $(u, v)$ are integrated over the regions: $0 \leq u \leq mv + b$ and $0 \leq v \leq v_1$. Eqs. (59) and (56) can be plugged into (61), for a double integral that is straightforward to evaluate. In order to ensure the gradients are computed quickly, the integral can be symbolically computed before run time.

### 8.4 Sampling from the Collocation Knots

The approach presented in this section approximates the chance constraint by sampling from a probability distribution of the configuration states along a trajectory and computing which samples are in collision. We have not yet specified which distribution is used for sampling. One simple method is to assume there is some nominal state trajectory given by the collocation knots. Given this trajectory, samples can be drawn from each knot using some distribution $f$, from which it is possible to draw samples. For example, assume $\mathbf{q}_k$ are the nominal configuration states at a collocation knot $k$ and the uncertainty is approximated by a multivariate Gaussian distribution, $f = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Samples are generated at knot $k$ by sampling from $f$ and adding the samples $\mathbf{Z}_s$ to $\mathbf{q}_k$:

$$\mathbf{q}_{k,s} = \mathbf{q}_k + \mathbf{Z}_s \;\; \forall \;\; s \in \mathcal{S} \tag{62}$$

This scheme is illustrated in Fig. 18. While straightforward, this method relies on knowledge of $f$: a distribution that we can sample from that describes the state distribution along the trajectory. Aside from simple linear systems with Gaussian uncertainty, the true state

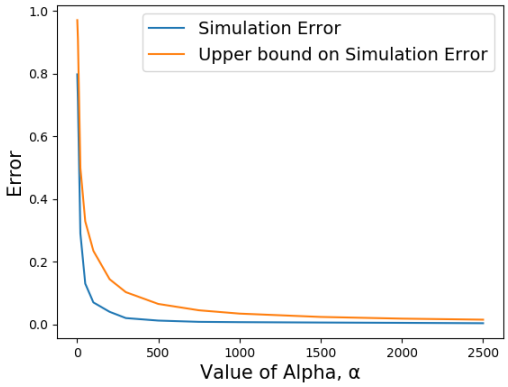



Figure 19: Comparison of the chance constraint error in simulation versus the upper bound on error, Eq. (67). The error is for a single trajectory knot.

distribution is often unknown and complex. Even for linear systems with Gaussian uncertainty, the state distribution is not truly Gaussian because configurations in collision must be removed. The crux of the issue is that the method presented in this section still focuses on modeling $P(S_k)$, i.e. the focus is on computing the individual chance constraint at each collocation knot rather than the joint chance constraint. The independence assumptions made in Section 7 to simplify the joint chance constraint do not hold for many systems. We address these issues in Section 9; rather than focusing on computing risk at individual indices, we compute risk over entire trajectories.

### 8.5 Selecting the Sigmoid Parameter $\alpha$

To approximate the chance constraint using a sigmoid function, sigmoid parameter $\alpha$ must be selected. In this section, we discuss a simple bound on the simulation error as a function of $\alpha$. Our method approximates the chance constraint via equation (32), repeated here:

$$P(\text{Collision}) \approx \frac{1}{S} \sum_{s} \mathbb{1}_{|\mathcal{W}_{int,s}|>0} \tag{63}$$

Eq. (63) represents a Monte Carlo method, which converges as $\mathcal{O}\left(S^{-1/2}\right)$ (Owen, 2013). However, the chance constraint approximation relies on approximating Eq. (63) with Eq. (36), repeated:

$$P(\text{Collision}) \approx \frac{1}{S} \sum_{s} \text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \tag{64}$$

The impact of the sigmoid is a bias in the chance constraint approximation:

$$\frac{1}{S} \sum_{s} \mathbb{1}_{|\mathcal{W}_{int,s}|>0} \geq \frac{1}{S} \sum_{s} \text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \tag{65}$$

As $\alpha$ increases, the approximation improves and the RHS of Eq. (65) approaches the LHS. Denote the bias as $B$ such that:

$$\frac{1}{S}\sum_{s}\mathbb{1}_{|\mathcal{W}_{int,s}|>0} = \frac{1}{S}\sum_{s}\text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) + B \tag{66}$$

Bias $B$ is a random variable dependent on the state samples $\mathbf{Z}_s$; each scenario contributes more to the bias if the intersecting region $|\mathcal{W}_{int,s}|$ is greater than zero but $\text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) < 1$. Let $r_B$ be the region of the workspace for which $0 < \text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \leq 1-\delta$, where $\delta$ is some infinitesimally small number. Intuitively, if a sample lands in $r_B$, it will cause a greater bias and the sigmoid will be a poor approximation to the indicator function. The expected value of $B$ can be written $\mathbb{E}\left[B\right] = \mathbb{E}\left[B_s\right]\mathbb{E}\left[S_{r_B}\right]$. The first term $\mathbb{E}\left[B_s\right]$ represents the expected bias per sample. It is challenging to compute exactly because it represents the expected value of the sigmoid as a function of $\mathcal{W}_{int}$ integrated over $r_B$. It may be upper bounded by 1. The second term, $\mathbb{E}\left[S_{r_B}\right]$, represents the expected number of scenarios that fall into $r_B$. This term can be written

$$\mathbb{E}\left[S_{r_B}\right] = SF_{r_B} \tag{67}$$

where $F_{r_B}$ is the CDF of $f$ integrated across $r_B$. Note that as $\alpha \to \infty$, the region $r_B$ goes to 0. For Gaussian $f$, $r_B \to 0$ means $F_{r_B} \to 0$ and consequently the bias $B$ goes to 0. The upper bound is depicted graphically in Fig. 19; the blue line is the error of the simulated trajectory; the orange line is the result of Eq. (67).

## 9. Shooting Method Monte Carlo

Up to now, we have relied on a simplification of the joint constraint into individual chance constraints and focused on computing the individual constraints. In doing this, we ignored the joint distribution of states between different time steps. Additionally, we assumed that the distribution at each time step was known and we either had access to its cumulative distribution function $F$ or could sample from its distribution $f$. In the case of nonlinear dynamics or non-Gaussian noise, $f$ is unlikely to be known.

In this section, we propose a method that computes the joint chance constraint by forward simulating realizations of entire trajectories. We then determine which scenarios fail and use this information to compute the overall trajectory risk. To do this, we incorporate a different method of trajectory optimization: the shooting method. In the deterministic case, the shooting method works by forward simulating a trajectory and updating the control inputs until the final simulated state matches the goal state, pictured in Fig. 20. The insight underlying this section is that the shooting method of trajectory optimization is similar to a single Monte Carlo simulation of a stochastic process. We combine the two strategies into the Shooting Method Monte Carlo (SMMC) algorithm.

Shooting Method Monte Carlo proceeds as follows. Decision variables are control inputs $\mathbf{u}$ and time steps $\mathbf{h}$; unlike the collocation method states $\mathbf{x}$ are not explicit decision variables but are determined via simulation of the equations of motion. First, a set of samples is drawn from the distribution describing the additive noise in the dynamical equations of motion. The number of samples is $SMK$ where $S$ is the total number of scenarios, $M$ is the number of dynamics states, and $K$ is the planning horizon. In the case that the mathematical program is solved via a sequential quadratic program, the algorithm proceeds iteratively.

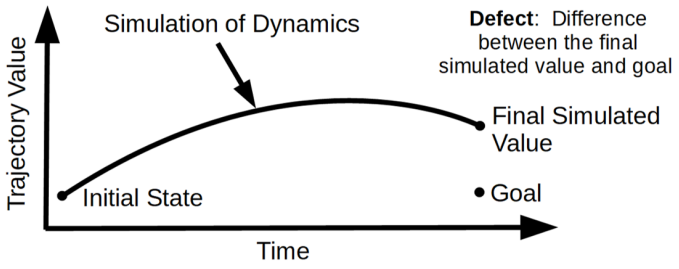



Figure 20: An illustration of the shooting method where a set of control inputs is simulated forward in time.

At each iteration, the SQP optimizer assigns values to the decision variables, $\mathbf{u}$ and $\mathbf{h}$, and asks for an evaluation of the objective and constraint functions or the Jacobian matrix of the objective and constraints. Given these decision variable assignments, the agent's trajectory is simulated across $S$ scenarios by integrating the equations of motion. Importantly, we assume $\mathbf{u}$ and $\mathbf{h}$ are deterministic; they are the same across all scenarios. However, each trajectory scenario $\mathcal{T}_s$ relies on a different sequence of samples $\mathbf{Z}_s$ such that the set of scenarios $\boldsymbol{\mathcal{T}}$ approximate the agent's trajectory distribution $f$.

Given the results of the simulations, a defect $d_i$ is calculated for each constraint $C_i$. In the deterministic case, the defect is measured as:

$$C_i - \overline{C}_i(\mathbf{u}, \mathbf{h}) = d_i \tag{68}$$

where $C_i$ is the desired value of constraint $i$ and $\overline{C}_i(\mathbf{u}, \mathbf{h})$ is the value of the constraint resulting from simulating the dynamics using decision variables $(\mathbf{u}, \mathbf{h})$. For example, if the desired goal at state $\mathbf{x}_K$ is $\mathbf{x}_{goal}$ and the simulated state at step $K$ is $\overline{\mathbf{x}}_K$, the defect is $d_i = \mathbf{x}_{goal} - \overline{\mathbf{x}}_K$. The task of the trajectory optimization routine is to reduce defects to some small tolerance.

In the case of Shooting Method Monte Carlo, the trajectory is simulated $S$ times and there is no single $\overline{C}_i(\mathbf{u}, \mathbf{h})$. Instead, SMMC approximates the expected value of each constraint $C_i$ that results from simulating the trajectory with respect to decision variables $(\mathbf{u}, \mathbf{h})$ and samples $\mathbf{Z}_s$. SMMC's defect is expressed as:

$$C_i - \mathbb{E}_{\hat{f}}\left[\overline{C}_i(\mathbf{u}, \mathbf{h})\right] = d_i \tag{69}$$

where $\hat{f}$ is the trajectory's simulated empirical distribution. The expected value in Eq. (69) is:

$$\hat{C}_i = \mathbb{E}_{\hat{f}}\left[\overline{C}_i(\mathbf{u}, \mathbf{h})\right] = \frac{1}{S}\sum_s \overline{C}_{i,s}(\mathbf{u}, \mathbf{h}) \tag{70}$$

where $\hat{C}_i$ is used to represent the constraint's expected value. For SMMC, two types of constraints rely on Eq. (70): the chance constraint and the region constraints. Approximating the objective follows a similar scheme. The value of the objective is computed for each scenario, $J_s$, and the expected cost is approximated as:

$$\hat{J} = \mathbb{E}_{\hat{f}}[J] = \frac{1}{S}\sum_s J_s \tag{71}$$

**Algorithm 8:** Shooting Method Monte Carlo

**Input:** An agent $A$, environment $E$, sequence of regions $\mathcal{P}$, objective $J$, and chance constraint $\epsilon$

**Output:** A sequence of control inputs $\mathbf{u}$ and time steps $\mathbf{h}$

1 $\mathbf{Z} \leftarrow$ Draw $SMK$ standard normal samples
2 **while** Optimizer tolerance is not met **do**
3  **for** $s \in \mathcal{S}$ **do**
4  $\mathcal{T}_s \leftarrow SimulateTrajectory\left(\mathbf{u}, \mathbf{h}, \mathbf{Z}_s\right)$
5  **if** Optimizer requires a constraint evaluation **then**
6  $J_s \leftarrow EvaluateTrajectoryCost\left(\mathcal{T}_s\right)$
7  $C_{cc,s} \leftarrow EvaluateTrajectoryRisk\left(\mathcal{T}_s\right)$
8  $C_{r,s} \leftarrow EvaluateRegionConstraints\left(\mathcal{T}_s\right)$
9  **end**
10  **if** Optimizer requires a Jacobian evaluation **then**
11  $\mathbf{J}_s\left(\mathbf{u}, \mathbf{h}\right) \leftarrow EvaluateJacobians\left(\mathbf{u}, \mathbf{h}\right)$
12  **end**
13  **end**
14  $\hat{J}, \hat{C}, \hat{\mathbf{J}} \leftarrow ComputeExpectedValues\left(J_{\forall s \in \mathcal{S}}, \mathbf{C}_{\forall s \in \mathcal{S}}, \mathbf{J}_{\forall s \in \mathcal{S}}\right)$
15  $\mathbf{u}, \mathbf{h} \leftarrow Optimizer.UpdateDecisionVariables\left(\hat{J}, \hat{C}, \hat{\mathbf{J}}\right)$
16 **end**

An overview of the SMMC algorithm is shown in Alg. 8. The algorithm proceeds until the optimizer reaches some optimality and feasibility tolerance set by the user, Line 2. Next, the algorithm loops over all scenarios $s \in \mathcal{S}$. The inner loop begins by simulating the trajectory forward in time, Line 4. The trajectory $\mathcal{T}_s$ is dependent on $MK$ random variable samples specific to scenario $s$ as well as the control inputs and time steps, $(\mathbf{u}, \mathbf{h})$. Depending on what is required by the optimizer, the algorithm evaluates either the objective and constraints or the Jacobian matrix of the objective and constraints. In the case that a constraint and objective evaluation is required, the trajectory cost $J_s$, chance constraint $C_{cc,s}$, and region constraints $C_{r,s}$ are evaluated for each scenario, Lines 6 - 8. Each evaluation is dependent on the simulated trajectory $\mathcal{T}_s$. If a Jacobian evaluation is required, the Jacobian matrix $\mathbf{J}$ specific to scenario $s$ is computed. An important feature of SMMC is that the entire inner loop, Lines 3 - 13, can take advantage of hardware parallelization such as provided by a GPU. This is discussed further in the results section. Line 14 relies on Eqs. (70) and (71) to compute the expected value of the objective and constraints. Notation $J_{\forall s \in \mathcal{S}}$, $\mathbf{C}_{\forall s \in \mathcal{S}}$, $\mathbf{J}_{\forall s \in \mathcal{S}}$ represents the fact that $ComputeExpectedValues$ computes the expected value across all scenarios in $\mathcal{S}$. Note that we use a slight overload of notation; $J$ represents the objective function while boldface $\mathbf{J}$ represents the Jacobian matrices. Finally, Line 15 passes the evaluations of the objective, constraints, and the Jacobians to the underlying optimization subroutine. This routine returns an updated set of control variables and time steps, $(\mathbf{u}, \mathbf{h})$.

Evaluation of the chance constraint at each time step proceeds very similarly to the method in Section 8. The probability of failure is approximated by taking the sigmoid of

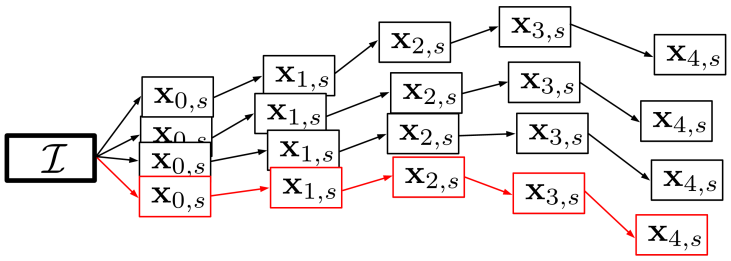



Figure 21: Illustration of the trajectory sampling and simulation strategy used by SMMC. The red trajectory is in collision and its $\mathcal{W}_{int,s}$ is positive.

the total intersecting region over an entire scenario:

$$P\left(\text{Collision}\right) \approx \frac{1}{S} \sum_{s \in \mathcal{S}} \text{Sig}_{\alpha}\left(|\mathcal{W}_{int,s}|\right) \tag{72}$$

An important difference between the method in this section and Section 8 is how the agent's configuration state at knot $k$ under scenario $s$, i.e. $\mathcal{W}_{a,k,s}$, is calculated. In the collocation case, this is done by using the nominal trajectory $\mathcal{T}$ and drawing samples at each configuration state $\mathbf{q}_k$ by sampling from a distribution $f$. In contrast, for SMMC, each $\mathcal{W}_{a,k,s}$ is generated directly from each trajectory scenario $\mathcal{T}_s$, which is itself the result of forward integrating the stochastic dynamics. The difference between the sampling strategies of the collocation method and SMMC is shown by contrasting Figs. 18 and 21. The collocation method samples at each collocation knot and ignores dependencies between time steps in the stochastic process. SMMC better approximates these dependencies by simulating entire trajectories.

Other details of Shooting Method Monte Carlo follow directly from the standard shooting method. Because the focus of this article is on the model of risk and not the underlying trajectory optimization, the details are left to Appendix B.

## 10. Results

Four sets of test cases are included. First, a test case is run comparing the hybrid search's ability to quickly compute feasible paths for linear systems in 2D workspaces using CDF-based chance constraints. Second, a case is performed for a linearized AUV model exploring the Kolumbo undersea volcano. Third, a nonlinear Slocum glider model is explored to illustrate the collocation sampling-based chance constraint approach. Finally, Shooting Method Monte Carlo is illustrated on a Dubins car. The first two cases in Sections 10.2 and 10.3 showcase First Feasible Hybrid Search's ability against other planners in terms of time to reach an initial solution and trajectory cost. The latter tests, Sections 10.4 and 10.5, show the performance of the various chance constraint models in simulation.

### 10.1 Computing Environment

Benchmarks were performed on a machine with an Intel i7-4790 CPU at 3.60 GHz with 16 GB RAM. GPU tests utilized an Nvidia GeForce 1080 Ti. The algorithms are implemented in C++ and, for GPU-based parallelization, Nvidia's Cuda. SNOPT Version 7.6 was used as an optimizer for the trajectory planner (Gill et al., 2017).

### 10.2 Linear Models with CDF-Based Chance Constraints

This section describes a number of examples and benchmarks that illustrate First Feasible Hybrid Search (FFHS) on agents with linear dynamics. The examples model trajectory risk via the CDF-based approximation described in Section 7. All test cases in this section rely on double-integrator linear dynamics of the form:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} + \mathbf{w} \tag{73}$$

where $\mathbf{w} \sim \mathcal{N}(0, 5)$ and

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}, A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \tag{74}$$

For 2D environments, Eq. (24) was used to model risk and for 3D environments, Eqs. (25) and (26) were used. For each test, the risk was set to $\epsilon = 0.20$, e.g. a 20% probability of failure. Minimizing the trajectory's Euclidean distance was used for an objective.

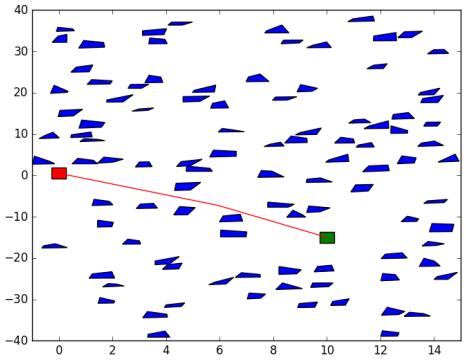
The algorithm was benchmarked against two other approaches from literature that also rely on linear models and CDF-based chance constraints. First, a benchmark was performed against the chance-constrained rapidly-exploring random tree (CC-RRT) approach discussed in (Luders et al., 2010). This algorithm extends the popular RRT planning algorithm to the probabilistic case. In brief, points are sampled from the state space and a set of nearest neighbors computed, which currently lie on the search tree. Paths are extended from the neighbors in an attempt to connect with the new sample. The probabilistic version requires not only that paths are collision free but also respect a chance constraint. The reader is referred to the innovative work in (Luders et al., 2010) for full details.

A second benchmark was performed against a disjunctive linear programming (DLP) approach similar to that presented in (Blackmore, 2008). This work considered convex, two dimensional obstacles. In this setting, $\mathcal{W}_{free}$ can be described by a conjunction of disjunctions:

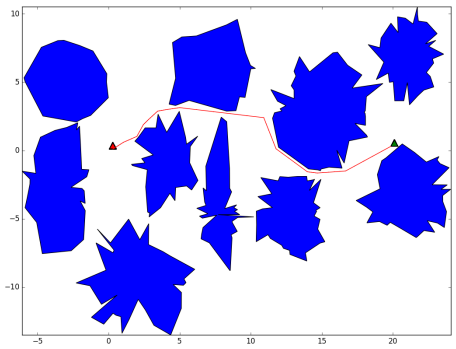
$$\mathbf{x}_k \in \mathcal{W}_{free} \ \forall \ k \iff \bigwedge_k \bigvee_l \mathbf{a}_{jl}\mathbf{x}_k \geq b_{jl} \tag{75}$$

where $l$ represents the $l^{\text{th}}$ linear constraint of the $j^{\text{th}}$ obstacle and $\mathbf{a}$ and $b$ are constants relating to surface geometry. Because of the disjunction, an integer decision variable $z_{j,k,l}$ is required that encodes whether the $j^{\text{th}}$ obstacle has constraint $l$ active at time step $k$:

$$\mathbf{a}_{jl}\mathbf{x}_k - \mathbf{b}_{jl} + Mz_{j,k,l} \leq 0 \tag{76}$$

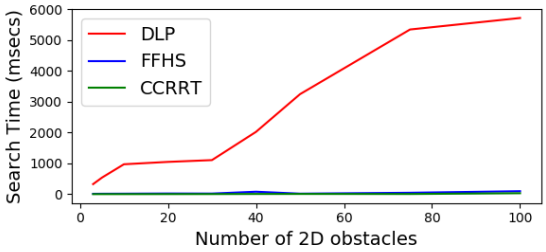


(a) Example 100 trapezoidal obstacle test case

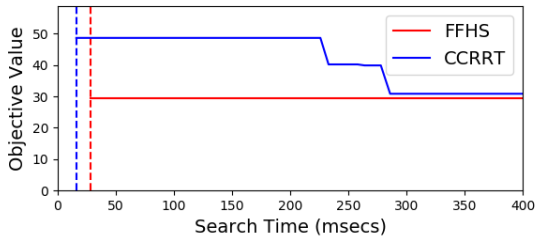


(b) FFHS can handle non-convex obstacles, which the disjunctive linear program cannot.

Figure 22: Example environments for the 2D linear models




(a) Comparison of the time taken to find initial feasible solutions for environments with various numbers of 2D convex obstacles.




(b) Objective values for FFHS and CC-RRT on a 2D obstacle test cases with 100 obstacles. Vertical dashed lines indicate the time to find the first feasible solution. The objective is the path's Euclidean distance.

Figure 23: Results for the 2D linear models

where $M$ is a big positive constant. Constraint (76) necessitates the use of a MILP solver; our benchmarks relied on (Gurobi Optimization, Inc., 2014).

Example trajectories produced by the FFHS algorithm are shown in Figs. 22a and 22b. Benchmarking plots are provided in Figs. 23a - 23b. As shown in Fig. 23a, CC-RRT and FFHS outperformed the DLP considerably in terms of speed. One reason for this is that, as presented in (Blackmore, 2008), the DLP lacks a heuristic to dictate which obstacle constraints may be relevant. This means that the DLP must solve a MILP which accounts for all obstacle constraints in the environment. Large obstacle fields will generate enormous mixed integer linear programs even when only a small number of the constraints are relevant.

Fig. 23a shows that CC-RRT and FFHS are close in performance with CC-RRT having a slight advantage in time taken to find an initial feasible solution. This is illustrated on a single 100-obstacle test case in Fig. 23b where CC-RRT is able to generate an initial feasible solution faster. On the other hand, FFHS performs well when comparing the quality

of solutions. Typically, FFHS can find a solution almost as fast as CC-RRT but the solution is often much better. The results show that there is value in the hybrid search and using a trajectory optimizer to improve paths. The hybrid search often leads to considerably better paths in a similar amount of time as the RRT-based approach.

## 10.3 Exploring the Kolumbo Undersea Volcano

A test case was performed on a Slocum Glider AUV exploring the undersea volcano Kolumbo off the coast of Greece. In this scenario, a team of scientists uses the motion planner to design trajectories for science goals; the planner should be fast to provide quick feedback whether a set of goals is feasible for the glider and minimize the time and energy usage required. There are certain regions of interest on the seafloor such as a ridge at the caldera's northeast corner.

This section, Section 10.3, describes a linearized model that relies on CDF-based chance constraints and is appropriate for long missions; Section 10.3.1 describes the simplified dynamical model, Section 10.3.2 describes the objective used to approximate glider energy usage, and Section 10.3.3 describes experimental results. The next section, Section 10.4, models risk using sampling-based chance constraints; this is more computationally intensive and is better for shorter trajectories when a more precise estimate of trajectory risk is needed.

### 10.3.1 Linearized "Sawtooth" Dynamics

The undersea Slocum glider has unique dynamics that enable high energy efficiency and long duration missions. This section develops a simple linearized model that captures the glider's basic sawtooth trajectory pattern. A thorough description of underwater glider design is provided in (Davis, C. Eriksen, & P. Jones, 2002).

The glider's motion is driven by a buoyancy engine. This means that rather than relying solely on propellers for thrust like a typical sea vessel, the glider can change its buoyancy and produce lift and drag similar to an airplane. The glider alternates between modes of low buoyancy and high buoyancy, which cause the vehicle to move between peaks and troughs, *inflection* points. Its motion through the sea resembles a sawtooth pattern. The glider is also characterized by difficulty localizing its position in the caldera. It is unable to use GPS underwater and must rely on dead reckoning, depth sensors, and sonar sensors that output the distance to the seafloor. Traveling through open ocean may take the glider away from the seafloor and out of range of its sonar sensors; it may be more advantageous to travel along the seafloor even if it results in a longer trajectory.

The results presented in this section model the glider's dynamics as linear between inflection points. Inflection points are modeled as regions $\mathcal{R}$ in the hybrid search. The glider operates in a depth band $d$ and travels at a pitch angle $\beta$ to the horizontal. Glider inflection regions are generated via a simple geometric procedure; they are inserted between every pair of landmark regions by alternating between peaks and troughs. This construction is illustrated in Fig. 24. Trajectory collocation knots are constrained to lie within each inflection region, i.e. the trajectory planner picks where in each region the glider will inflect. The size of the region is dictated by the range of pitch angles one allows; this is typically 15-25°.

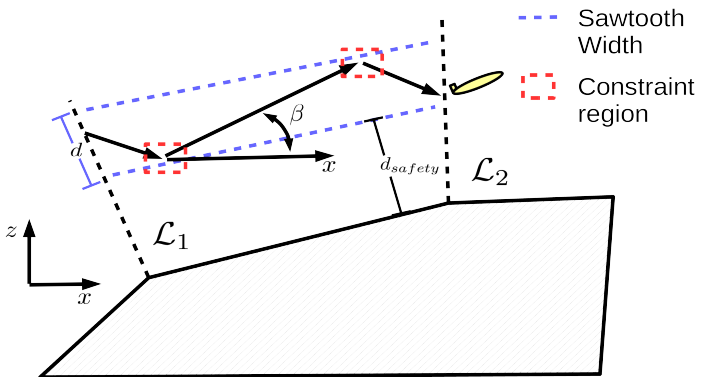



Figure 24: Integrating the glider's sawtooth trajectory with the region planner's $\mathcal{L}$. The glider travels at an angle $\beta$ to the horizontal in depth band $d$.

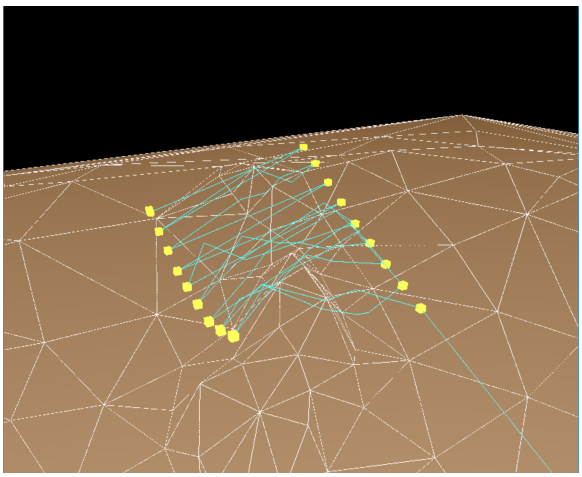


Figure 25: Illustration of the lawnmower motion primitive where the glider explores the ridge at the caldera's northeast corner. The trajectory is denoted by cyan lines with lawnmower goals in yellow.

### 10.3.2 Objective Function

Because mission range is important to the glider, its energy consumption must be considered. To simplify the problem, this test case assumes rules of thumb for energy consumption, i.e. how much energy a certain maneuver will require. There are two important maneuvers: inflecting and changing the glider's yaw. The first maneuver is much more energy intensive because it requires using the buoyancy engine; changing the glider's buoyancy at large depths necessitates operating a pump at high pressures. To model both of these effects and the trajectory's time, a multi-term cost function is assumed:

$$J = c_t \sum_{k=0}^{K} h_k + c_{inflection} \sum_{k=0}^{K} \mathbb{1}_{inflection,k} + c_{yaw} \sum_{k=0}^{K-1} \Delta_{yaw,k,k+1} \tag{77}$$

| | Open Ocean | Multi-Goal | Constrained to Seafloor | Lawnmower Pattern Small | Lawnmower Pattern Large |
|---|---|---|---|---|---|
| FFHS | 55 | 133 | 69 | 102 | 556 |
| CC-RRT | 11 | 148 | 1064 | 100 | 992 |
| CC-RRT-Connect | 11 | 104 | 388 | 97 | 754 |
| Interleaved | 37 | 90,440 | 17,437 | 4,018 | Timeout |

Table 1: Comparison of search times for different search strategies on five test cases. Times listed are the times taken to generate an initial feasible solution. Times are in milliseconds. A timeout was set at 5 minutes.

The first term represents a penalty on the trajectory's time; $c_t$ is a constant that penalizes time usage. The second term, $c_{inflection}$, represents an estimate of the energy used by an inflection and the indicator function $\mathbb{1}_{inflection,k}$ represents whether there is an inflection at knot point $k$. The final term relates to the energy usage by changes in yaw; $c_{yaw}$ is the estimated energy usage per change in degree of yaw and $\Delta_{yaw,k,k+1}$ represents the change in glider yaw from trajectory knot $k$ to $k+1$. For this study, $c_t$ was set to 100, $c_{inflection}$ set to $10,000$ to account for the approximately 10kJ used for a change in the buoyancy, and $c_{yaw}$ set to 2 (Rudnick, Davis, Eriksen, Fratantoni, & Perry, 2004). For a practical discussion of the glider's propulsion and energy usage, the reader is referred to (Cooney, 2016).

### 10.3.3 Results for Various Missions

Five missions were used as test cases. A basic "Open Ocean" case requires motion through the open ocean above the caldera. A "Multi-Goal" case requires the glider to reach three randomly placed goals in the caldera. Thirdly, "Constrained to Seafloor" forces the glider to travel within 50m of the seafloor for better localization. Finally, "Lawnmower Pattern Small" and "Lawnmower Pattern Large" call for lawnmower patterns that explore the ridge along the caldera's northeast corner. An example lawnmower pattern is shown in Fig. 25. Lawnmower patterns are generated by creating a grid of goals (yellow in Fig. 25). The grid is generated for a square defined by minimum and maximum latitude-longitude coordinates.

First Feasible Hybrid Search (FFHS) was benchmarked against CC-RRT and two additional approaches. We modified the RRT-Connect algorithm as described in (J. Kuffner & Lavalle, 2000) to handle chance constraints. RRT-Connect expands on vanilla RRT by growing trees from both start and goal states. Our modified algorithm, "CC-RRT-Connect," expands and connects trees similar to RRT-Connect, but calculates trajectory feasibility in terms of the chance constraint, similar to CC-RRT. A final benchmark was performed against a variant of hybrid search. One of the characteristics of hybrid search is that there are many ways in which the path planner and trajectory planners can interact. One possibility is that the trajectory planner is called after every successor is generated by the region planner. This is similar to the strategy used by ScottyPath in (Gonzalez, 2018). This is termed an "Interleaved" hybrid search because of how the region planner and trajectory

| | Open Ocean | Multi-Goal | Constrained to Seafloor | Lawnmower Pattern Small | Lawnmower Pattern Large |
|---|---|---|---|---|---|
| FFHS | 56 | 299 | 122 | 187 | 924 |
| CC-RRT | 116 | 2,817 | 617 | 2,257 | 50,750 |
| CC-RRT-Connect | 116 | 1,710 | 599 | 647 | 5,306 |
| Interleaved | 56 | 234 | 81 | 165 | Timeout |

Table 2: Comparison of the value of the cost function, Eq. (77), for test cases. Values listed are the objective value for the initial feasible solution rounded to the nearest thousand. A timeout was set at 5 minutes.

planners are tightly interleaved. This contrasts to the depth first strategy of First Feasible Hybrid Search.

For the RRT-based benchmarks,"CC-RRT" and "CC-RRT-Connect," the glider's sawtooth motion was generated by extending trajectories at feasible pitch angles in the direction of newly sampled points. Trajectories were extended until they no longer satisfied the chance constraint or arrived at the same latitude-longitude position as the sampled point. While RRT-based algorithms are good at exploring the search space through sampling, they must be modified to account for constrained trajectories such as those that must travel near a surface, i.e the "Constrained to Seafloor" use case. To perform the benchmarks described here, the sampling was unmodified but paths that did not travel within a certain distance to the seafloor for a pre-specified distance were pruned.

Results of the benchmarks are presented in Tables 1 and 2. Comparing against Interleaved Hybrid Search, FFHS produces solutions much more quickly. FFHS search times can be measured in milliseconds while times for Interleaved Hybrid Search are better measured in seconds. This is because the environment is complex and contains a huge number of potential constraints; there are a large number of potential paths that could be passed to the trajectory planner. While trajectory planner calls are typically measured in hundreds of milliseconds, calling the trajectory planner hundreds of times over the course of one planning instance (such as in the case of the Interleaved Hybrid Search) will greatly slow the overall search. Because FFHS is greedy, the Interleaved Hybrid Search on completion produces solutions of better quality than FFHS, shown in Table 2. The Interleaved Hybrid Search's solutions were typically less than 20% better than FFHS. Whether this improvement in solution quality is worth the wait is at the user's discretion; it may be the case that the user has a long time to generate plans before execution. On the other hand, in cases where the user may want to test a large number of missions to determine which best suits their needs, the time to generate a trajectory may be a more desirable property.

FFHS also compares well against CC-RRT and CC-RRT-Connect. Regarding time to find an initial solution, Table 1, FFHS produced trajectories in times comparable to the RRT-based methods. For certain cases, "Constrained to Seafloor" and "Lawn Mower Pattern Large," FFHS is faster. FFHS shows a large performance improvement when considering trajectory cost, Table 2. This is likely due to the environments being large; CC-

RRT and CC-RRT-Connect often found initial solutions far off course. The performance of both would likely be improved by allowing sampling to continue or performing some post-processing step. An advantage of FFHS is that its trajectory planner is similar to a risk-bound post-processing step for the region planner; while variants of RRT exist that provide smoothing, to the best of our knowledge, they have not been extended to the risk-bound case. Another advantage that FFHS has over the RRT-based implementations used in these tests is the use of motion primitives. For the "Constrained to Seafloor" test, FFHS relies on a motion primitive to compute the path along the seafloor; this is quicker to generate than the modification of CC-RRT and CC-RRT-Connect that simply prunes paths that do not travel along the seafloor. It is likely possible to generate a better sampling scheme for CC-RRT and CC-RRT-Connect, such as only sampling points from regions close to obstacles.

### 10.4 Navigating a Ridge in Kolumbo

This use case illustrates hybrid search applied to a nonholonomic system where agent geometry is important to modeling collision risk. The case showcases the advantage of Section 8's sampling-based model versus the simpler CDF-based model presented in Section 7 and other prior work. The planner is tasked with computing glider trajectories that run along a ridge at the undersea volcano's northeast corner. The planner must reason about uncertain ocean currents that may push the glider into the ridge. Unlike previous cases, a more accurate nonlinear model is used for the glider's dynamics.

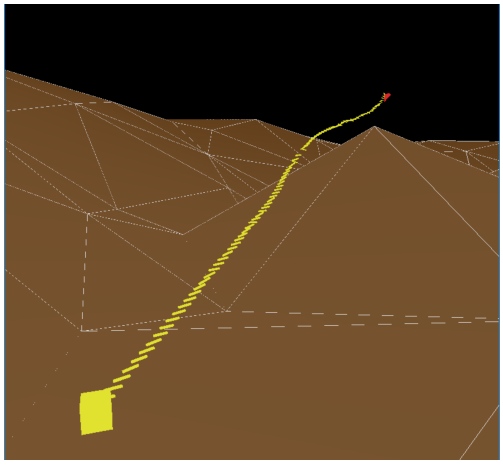
#### 10.4.1 Nonlinear Slocum Glider Model

This use case closely follows the glider model in (Leonard & Graver, 2001) where the glider's equations of motion are derived using conservation of energy and momentum. The chief difference between the model used in this article and in (Leonard & Graver, 2001) are the control inputs. The model in (Leonard & Graver, 2001) assumes the glider has access to two control inputs: a stationary point mass whose mass value can be varied and a movable point mass. The stationary point mass models the buoyancy engine; the glider's overall buoyancy is altered by changing its mass. Moving the movable point mass affects the center of buoyancy and consequently the glider's pitch. To simplify the model, we assume the glider has a variable mass but uses adjustable fins to change pitch and yaw rather than a movable mass. This removes the movable point mass's state from the equations of motion and allows for a slightly simpler model.

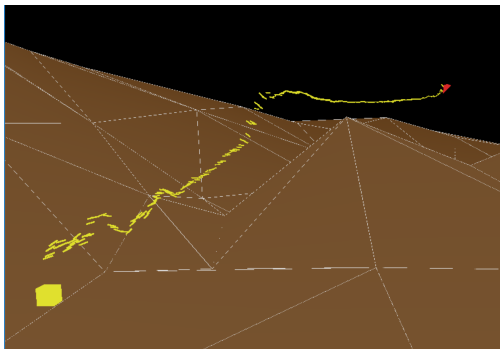
While (Leonard & Graver, 2001) focuses on control of a 2D test case, this work uses a full 3D dynamical model as a test case. It relies on thirteen states and three control inputs. The states are:

$$[x_i, y_i, z_i, v_{x,b}, v_{y,b}, v_{z,b}, \psi, \theta, \phi, \omega_x, \omega_y, \omega_z, m_b] \tag{78}$$

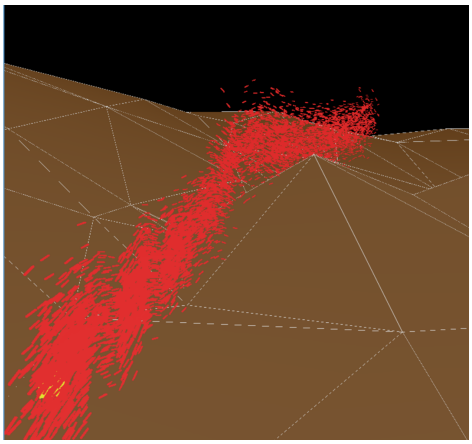
States $(x_i, y_i, z_i)$ are the glider's waypoint states in the inertial reference frame, while states $(v_{x,b}, v_{y,b}, v_{z,b})$ are velocity states in the glider's body reference frame. Angles $(\psi, \theta, \phi)$ are Euler angles that transform the inertial frame to the body frame by a rotation $\phi$ about the original $z$ axis, a rotation $\theta$ about the new $y$ axis, and finally a rotation $\psi$ about the new $x$ axis as described in (Triantafyllou, 2004). States $(\omega_x, \omega_y, \omega_z)$ are the rates of rotation around the $x$, $y$, and $z$ axes in the body frame. $m_b$ is the mass of the buoyancy engine.



(a) A trajectory under the assumption of small ocean currents and little uncertainty about the currents.



(b) A trajectory under larger uncertainty in the ocean currents. The trajectory optimizer generates a significantly different trajectory than the case with low uncertainty.



(c) Depiction of the samples used to compute the trajectory risk.

Figure 26: Various simulations of the nonlinear glider model avoiding a ridge. The glider's geometry is modeled as a polyhedron.

We assume three control inputs: $u_{pitch}$, $u_{yaw}$, and $u_{m_{buoyancy}}$. Inputs $u_{pitch}$ and $u_{yaw}$ model input fin angles; $u_{m_{buoyancy}}$ is the change in mass of the buoyancy engine.

### 10.4.2 Results

Example trajectories are shown in Figs. 26a and 26b. Fig. 26a presents a trajectory where there is little uncertainty that the ocean currents are small. In this case, a multivariate Gaussian was added to the configuration states $(x, y, z, \psi, \theta, \phi)$ with $\boldsymbol{\mu} = [0, 0, 0, 0, 0, 0]$ and $\sigma_x = 0.05$, $\sigma_y = 0.05$, $\sigma_z = 0.05$, $\sigma_\psi = 0$, $\sigma_\theta = 0$, $\sigma_\phi = 0$ (all off-diagonal terms in the covariance matrix are assumed zero). As shown in Fig. 26a, the small amount of uncertainty

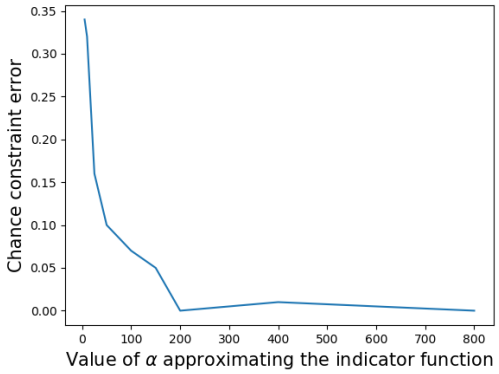
**Algorithm 9:** Collocation Simulation

**Input:** A chance-constrained trajectory planning mission, ccTPM
**Output:** $\epsilon'$, the simulated risk of trajectory $\mathcal{T}$

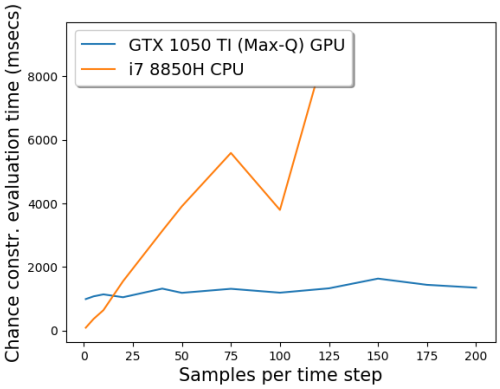
1 $\mathcal{T} \leftarrow FFHS\,(ccTPM)$
2 $total\_failures \leftarrow 0$
3 **for** $s \in total\_simulations$ **do**
4  **for** $\boldsymbol{q}_k \in \mathcal{T}$ **do**
5  $\mathbf{Z}_s \leftarrow DrawSample\,(f)$
6  $\boldsymbol{q}_{k,s} \leftarrow \boldsymbol{q}_k + \mathbf{Z}_s$
7  $\mathcal{W}_{a,k,s} \leftarrow ComputeBodyLocation\left(\boldsymbol{q}_{k,s}\right)$
8  **if** $InCollision\,(\mathcal{W}_{a,k,s})$ **then**
9  $total\_failures \leftarrow total\_failures + 1$
10  Go to next scenario
11  **end**
12 **end**
13 $\epsilon' \leftarrow total\_failures/total\_simulations$

means the optimizer can place the trajectory very close to the ocean floor; the generated trajectory is smooth with a slight curve to compensate for the ridge. The second example, Fig. 26b, presents a trajectory generated under more uncertainty in the ocean currents. For this example, $\boldsymbol{\mu} = [-5.0, 0, -5.0, 0, 0, 0]$ and $\sigma_x = 5.0$, $\sigma_y = 0$, $\sigma_z = 5.0$, $\sigma_\psi = 0$, $\sigma_\theta = 0$, $\sigma_\phi = 0$ (once again, all off-diagonal terms in the covariance matrix were zero). As depicted in Fig. 26b, this changes the output trajectory considerably. The optimizer produces a solution where the glider is kept at approximately the same depth until abruptly descending after passing the ridge. Both Figs. 26a and 26b show the nominal glider trajectory, i.e. they plot the knot points of the collocation trajectory optimization. As described in Section 8, the chance constraint is computed via sampling from distributions describing the configuration states and adding the samples to the collocation decision variables. A plot of these samples is shown in Fig. 26c. This test case was for 100 samples per collocation knot; for clarity, only every third sample is shown in the figure.

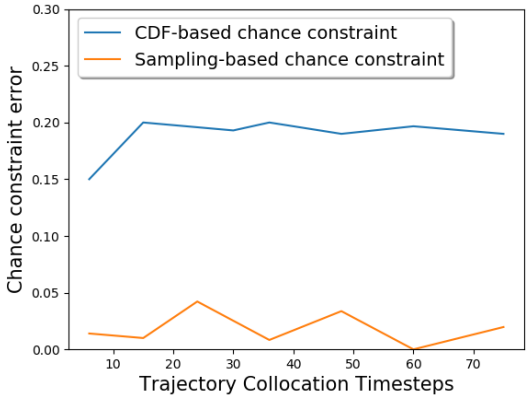
An essential question for all planners that deal with uncertainty is how well the returned plan matches the risk bound when executed in the real world. Unfortunately, most real world systems are prohibitively expensive to risk losing in order to test algorithmic performance. Therefore, simulation is used with various degrees of precision possible. The results in this section were generated using Collocation Simulation, Alg. 9. First, a trajectory $\mathcal{T}$ is generated using FFHS or possibly another planner. Second, the trajectory is simulated to determine whether the trajectory optimizer has produced a plan that meets the risk bound, $\epsilon$. The simulation mimics the collocation trajectory optimization in Section 8 by assuming a nominal configuration state $\boldsymbol{q}_k$ exists, sampling from state distribution $f$, and producing the configuration state under scenario $s$, $\boldsymbol{q}_{k,s}$, Line 6 of Alg. 9. Using $\boldsymbol{q}_{k,s}$, the agent's geometry in the workspace, $\mathcal{W}_{a,k,s}$, is generated. The fraction of simulations in collision




(a) The ability to meet the chance constraint in simulation as a function of $\alpha$




(b) Comparison of using sampling to approximate a chance constraint on a CPU vs. a GPU




(c) Error of a CDF-based chance constraint model vs. a sampling-based model in simulation. The sampling-based model assumed $\alpha = 200$ and used 800 scenarios.

Figure 27: Numerical results of the sampling-based chance constraint model using collocation trajectory optimization

provides the simulated trajectory risk, $\epsilon'$. Algorithm performance is measured by how well the simulated trajectory risk $\epsilon'$ matches the desired trajectory risk $\epsilon$ of the ccTPM.

Numerical results are presented in Figs. 27a - 27c. Figure 27a presents the ability of the sampling-based chance constraint model to approximate the trajectory risk in simulation as a function of the $\alpha$ parameter of Eq. (36). The approximation improves as $\alpha$ increases, which means the risk of the planned trajectory more closely matches the risk of the simulated trajectory. A significant advantage of sampling-based methods to approximate trajectory risk is that they are typically straightforward to parallelize. Collision checks and the corresponding Jacobian computations can be parallelized across the threads of a GPU. The improvement is significant as shown in Fig. 27b, which compares the glider ridge navigation computation on a GPU vs. a CPU. There is a cost to transferring data to the GPU; instances with few samples per time step perform better on a CPU. However, as the number of scenarios becomes large, there is a clear advantage to the GPU implementation.

**Algorithm 10:** Shooting Method Simulation

**Input:** A chance-constrained trajectory planning mission, ccTPM

**Output:** $\epsilon'$, the simulated risk of trajectory $\mathcal{T}$

$\mathcal{T} \leftarrow FFHS\left(ccTPM\right)$
$total\_failures \leftarrow 0$
**for** $s \in total\_simulations$ **do**
  $\mathbf{x}_{0,s} \leftarrow \mathbf{x}_0$
  **for** $(\mathbf{u}_k, \mathbf{h}_k) \in \mathcal{T}$ **do**
    $\mathbf{x}_{k+1,s} \leftarrow SimulateTrajectory\left(\mathbf{x}_{k,s}, \mathbf{u}_k, \mathbf{h}_k\right)$
    $\mathcal{W}_{a,k,s} \leftarrow ComputeBodyLocation\left(\mathbf{x}_{k+1,s}\right)$
    **if** $InCollision\left(\mathcal{W}_{a,k,s}\right)$ **then**
      $total\_failures \leftarrow total\_failures + 1$
      Go to next scenario
  **end**
**end**
$\epsilon' \leftarrow total\_failures/total\_simulations$

Figure 27c shows the advantage of using a sampling-based chance constraint model versus a simpler CDF-based model. Both models assume the same nonlinear glider dynamics. The CDF-based model assumes the glider's geometry is concentrated at a single point and the methods of Section 7 used to compute trajectory risk when solving the ccTPM. The sampling-based model uses the method in Section 8 and specifically 8.4 to model trajectory risk. Each output trajectory generated using the different chance constraint models is then simulated using Alg. 9. The error in Fig. 27c is the difference between the ccTPM's desired risk level $\epsilon$ and the simulated risk level, $\epsilon'$. As is shown in the figure, the sampling-based chance constraint matches the trajectory risk in simulation much better than the CDF-based model. For all tests, the chance constraint value was $\epsilon = 0.20$. Typically, the CDF-based model computed a trajectory with an active chance constraint (i.e. $P\left(\text{Fail}\right) = \epsilon$), but the corresponding simulated trajectory had little risk. The reason is that the CDF-based model does not account for the agent's body shape $\mathcal{B}$; without this, it is impossible to accurately model trajectory risk.

### 10.5 Shooting Method Monte Carlo with a Dubins Car

To illustrate the SMMC algorithm, a Dubins car model is used. The Dubins car is one of the simplest possible nonlinear models, which models a car in the xy-plane with steering angle $\theta$ (LaValle, 2006). To illustrate the trajectory planner's ability to model the risk of agents with non-trivial geometry, we model the car's geometry as a rectangle with length and width $d_x$ and $d_y$.

The algorithm was tested with the Dubins car navigating around an obstacle. An example set of simulations is shown in Figs. 28 and 29. Benchmarks are shown in Figs. 30, 31, and Table 3. The simulation in Fig. 28 shows simulations of Dubins car trajectories given an initial guess for the decision variables $(\mathbf{u}, \mathbf{h})$. As mentioned previously, the initial

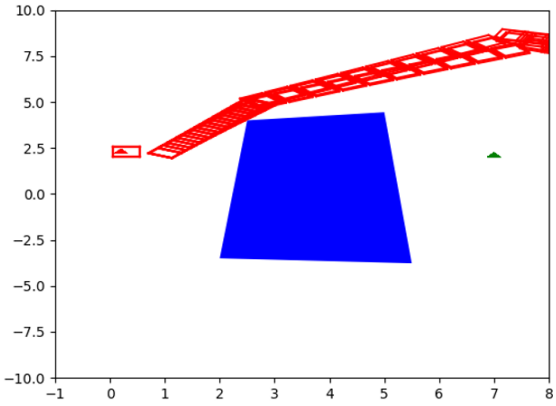



Figure 28: Simulations of trajectories using an initial guess for the control inputs $\mathbf{u}$

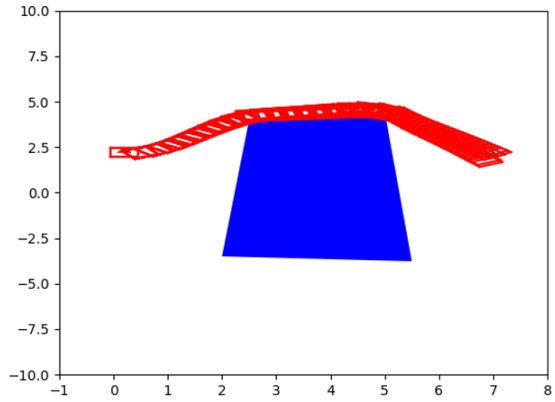



Figure 29: Simulations of trajectories using the optimized control inputs

trajectory should be feasible and is, in general, non-trivial to generate. Fig. 29 shows the resulting set of trajectories with $(\mathbf{u}, \mathbf{h})$ optimized via SMMC, which respect the chance constraint.

A Dubins car benchmark was performed testing the three risk models presented in this article for their ability to match the chance constraint in simulation. For this test, trajectories were produced using the respective model: the CDF-based method of Section 7, the collocation with sampling method of Section 8, and SMMC in Section 9. Unlike the results presented in the previous section that compare sampling-based collocation versus a CDF-based method (shown in Fig. 27c), these use a different simulation described by Alg. 10. Rather than assume a nominal configuration state $\boldsymbol{q}_k$ as in Alg. 9, Alg. 10 forward integrates the dynamics using the methods of Appendix B.1. The chief difference is Line 6, $SimulateTrajectory$, which forward integrates the stochastic dynamics one step. Alg. 10 is a more realistic simulation than Alg. 9 because it does not assume the existence of a nominal state at each step $k$. Using this form of simulation, Fig. 30 plots the three models' chance constraint simulation error for trajectories of various length. SMMC is able to best match the chance constraint in simulation. The collocation with sampling method shows

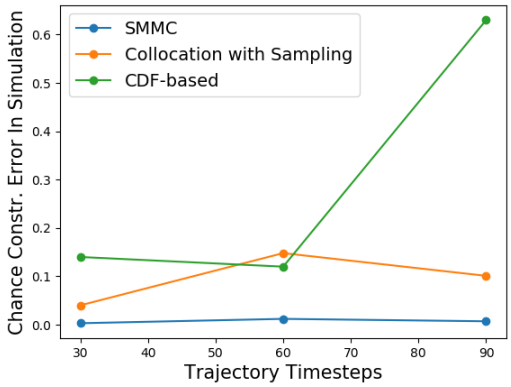



Figure 30: Comparison of the three chance constraint models' ability to approximate the chance constraint in simulation for the Dubins car

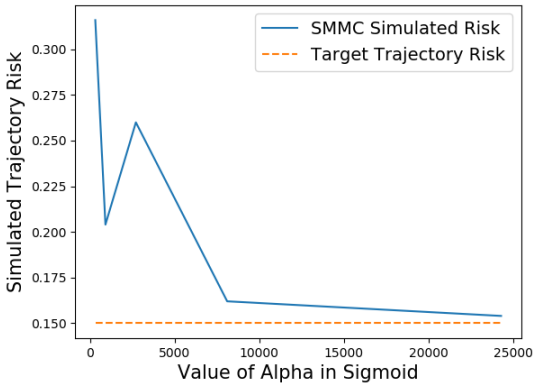



Figure 31: Plot depicting SMMC's ability to approximate the chance constraint in simulation as a function of the sigmoid's $\alpha$ parameter

worse performance because it assumes a nominal trajectory and draws Gaussian samples from this trajectory. Due to the agent's nonlinear dynamics, the true state distribution is likely not Gaussian. The CDF-based method performs poorly as it models neither the agent's true distribution nor geometry. For these benchmarks, the chance constraint, $\epsilon$, was set to 0.15. For the two sampling-based methods, $\alpha$ was set to 5e6 and the number of scenarios $S$ set to 2500. Uncertainty in the agent's dynamics, represented by $\sigma_x$, $\sigma_y$, $\sigma_\theta$ in Eqs. (97) - (99) was set to 0.1 for all variables.

Fig. 31 shows how increasing the value of $\alpha$ in the sigmoid function improves the accuracy of SMMC in respecting the chance constraint. As $\alpha$ increases, the sigmoid function approaches the indicator function that indicates whether a scenario is in collision or not. This plot was also generated using Shooting Method Simulation, Alg. 10. Finally, Table 3 provides examples of how parallelization speeds up SMMC. The second column shows the time taken to run SMMC using only a CPU without parallelization. The third column shows

| Number of Scenarios | CPU (secs) | GPU (secs) | Hot Start |
|---|---|---|---|
| 100 | 28.2 | 16.1 | No |
| 137 | 6.4 | 2.3 | Yes |
| 149 | 9.3 | 3.1 | Yes |
| 3732 | 182.2 | 24.3 | Yes |

Table 3: Table illustrating the improvement enabled by parallelization of SMMC on a GPU

the algorithm run on the same test cases but using parallelization to simulate trajectories and determine whether each is in collision. The final column, *Hot Start*, indicates whether the optimization was initialized with a solution from a previous optimization. The table shows that parallelization reduces the algorithm run time with the largest speed up coming when the number of scenarios is large. The makes sense; when there are a fewer number of scenarios to simulate, the time spent in the nonlinear optimizer dominates the overall run time. When there is a large number of scenarios, the time spent simulating each scenario dominates the run time.

## 11. Conclusion

In this article we presented an approach to hybrid search that solves the risk-bound motion planning problem for a richer set of environments, agent dynamics, agent geometries, and underlying stochastic processes than previously possible. We frame the problem as a ccTPM, a chance-constrained trajectory planning mission. The ccTPM models an agent that must accomplish a sequence of goals while constrained to a probability of mission failure. We solve the problem via hybrid search. There are two components of the hybrid search, a region planner and a trajectory planner. The job of the region planner is to produce candidate paths through the environment that achieve a sequence of goals. To achieve this, we introduce the notion of a landmark region which allows for the construction of an implicit region graph. A search method is introduced, First Feasible Hybrid Search, which greedily searches the region graph in order to produce good solutions quickly in complex environments. The region planner's paths are passed to a trajectory planner, whose job is to produce a trajectory that is dynamically feasible, minimizes a cost function, and is risk-bound. We presented three approaches to modeling trajectory risk, i.e. the chance constraint of a trajectory. We presented a fast approximation that relies on the availability of a cumulative distribution function to model the agent's state distribution. Second, we developed a sampling-based collocation method that models the risk of collision for agents with rigid bodies. Third, Shooting Method Monte Carlo models the joint chance constraint via simulating large numbers of trajectories. Different chance constraint models may be appropriate for different applications. The computationally simpler CDF-based method is appropriate for long trajectory missions. Sampling based-methods are important when the agent's geometry is relevant to computing risk and SMMC provides a more computationally intensive method that allows for modeling of non-Gaussian state distributions.

The benchmarks of hybrid search show its promise against state-of-the-art approaches. The method is able to handle much more complex environments than past planning approaches such as a disjunctive linear program, due to its use of a heuristic. The method also performs well against sampling-based chance-constrained RRT; it performs almost as quickly as CC-RRT but typically produces paths that are of much higher quality. Shooting Method Monte Carlo is able to better approximate the true collision risk of agents in simulation versus other models of trajectory risk.

## Acknowledgments

This work was partially supported by the Exxon Mobil Corporation and the Toyota Motor Corporation. We would like to thank Richard Camilli for his invaluable advice regarding the undersea use case. We would also like to thank the members of the MIT MERS lab and in particular the Scouts Team for their support developing this work. Finally, the journal reviewers' feedback was essential to help focus the article.

## Appendix A. Collocation Trajectory Optimization

The next two appendices detail the trajectory optimization used by the trajectory planner. Appendix A describes the collocation trajectory optimization. Appendix B describes the shooting method optimization underlying Shooting Method Monte Carlo.

The CDF-based and sampling-based chance constraint models of Sections 7 and 8 both rely on trajectory optimization via direct collocation. As mentioned in Section 6, collocation methods involve discretizing trajectories at knot points. *Direct* collocation means the objective is minimized directly rather than the necessary conditions of optimality, i.e. an *indirect* method. Typically, trajectory optimization via direct collocation requires decision variables of states $\mathbf{x}$, control inputs $\mathbf{u}$, and (optionally) time steps $\mathbf{h}$. For an overview of trajectory optimization the reader is referred to (Betts, 2009) and (Kelly, 2017).

An overview of the direct collocation trajectory optimization used in this article is shown in Fig. 32. Initial guesses for the decision variables are input as well as configuration state samples used to model the chance constraint (i.e. $\mathbf{Z}_s$ in Eq. (62)). The trajectory optimization proceeds by the SQP solver iterating between function and Jacobian evaluations and updating the decision variables. While the chance constraint models are described earlier in the article, the following sections describe the path constraints, the objective function, and the dynamics constraints.

### A.1 Path Constraints

The path passed by the region planner to the trajectory planner is compiled into sets of constraints. The form of the constraints depends on the region type. In the case of landmark regions, the waypoint states $\boldsymbol{\omega}$ of the agent are constrained to lie on the hyperplane. Recall from Section 3.1 we assume the agent has a unique set of waypoint states that characterize its position in the workspace, i.e. $\boldsymbol{\omega} = (x, y)$ or $\boldsymbol{\omega} = (x, y, z)$. To ensure the waypoint states at knot $K_n$ are constrained to the landmark region $\mathcal{L}_n$, the waypoint states are transformed

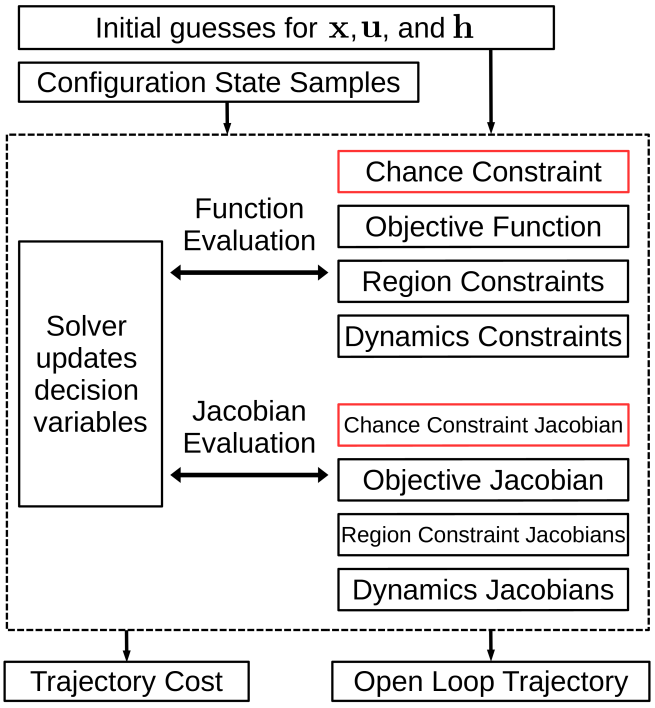



Figure 32: An overview of the collocation trajectory optimization inputs, outputs, and function evaluations. The box in red relates to the sampling-based chance constraint described in Section 8. It can be parallelized on a GPU.

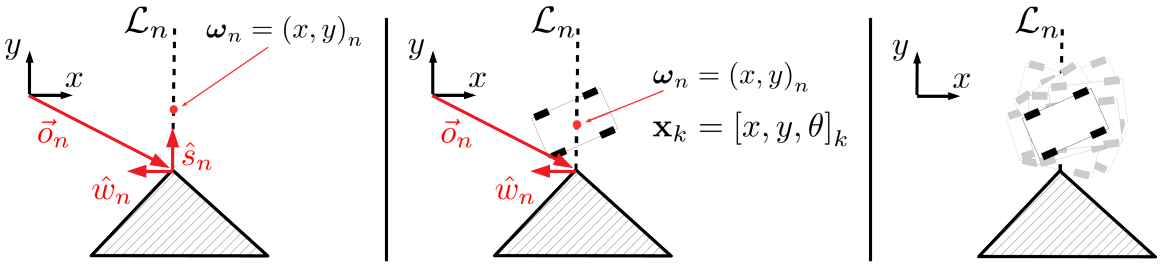



Figure 33: The above three panes illustrate how agents with geometry are integrated into the region planner's $\mathcal{L}_n$. The pane on the left depicts the landmark region with a point robot; the point robot is constrained to lie on the ray at $\hat{w}_n = 0$ and the risk evaluated as a function of the point's distance along the axis $\hat{s}_n$ as described in Section 7. The middle pane shows how waypoints are handled for agents with geometry (such as a Dubins car modeled as a rectangle); only the states $(x, y)$ are relevant to determining whether the agent lies on the ray $\mathcal{L}_n$. A similar construction holds for 3D workspaces where the waypoint states are $(x, y, z)$. For this construction, the $\hat{s}_n$ axis is no longer required because the risk is computed via sampling rather than an evaluation of a CDF. This is illustrated in the pane on the right; samples are generated using the decision variables at the $k^{th}$ collocation knot. The sampling method is described in Section 8.

into the coordinate $w_n$ using Eq. (8), restated here for clarity:

$$w_n = (\boldsymbol{\omega}_n - \vec{o}_n) \cdot \hat{w}_n \tag{79}$$

To constrain the waypoint states to the hyperplane, the constraint is necessary:

$$w_n = 0 \tag{80}$$

Fig. 33 depicts how point robots and agents with non-trivial body geometry are constrained to the landmark waypoint.

If the region is not a landmark region but a convex polytope of dimension $d$ for $\mathcal{W}^d$, the waypoint states are constrained to lie within the polytope by considering the planes that support the polytope's facets. For example, if the 3D waypoint states $\boldsymbol{\omega}_n = (x, y, z)_{k_n}$ are constrained to lie inside a polyhedral region, the following set of constraints are used:

$$ax + by + cz + d \geq 0 \;\; \forall \;\; hs \in HS \tag{81}$$

where $HS$ is a set of half spaces, the intersection of which describe the polyhedron. Constants $a$, $b$, $c$, $d$ define the plane that bounds half space $hs$. The normal vector of this half space points inside the polyhedron. This type of region might represent a "stay in" goal region.

Finally, it is possible to constrain a set of waypoint states to a single point. This might be useful to constrain an initial state value. In this case, the constraint is trivial:

$$\boldsymbol{\omega}_{n=0} = (x, y, z)_{k=0} = (x, y, z)_{\mathcal{I}} \tag{82}$$

where $(x, y, z)_{\mathcal{I}}$ are user-dictated initial state values.

### A.2 Objective Function

A wide variety of objective functions are possible including minimizing the distance traveled, minimizing time, or minimizing the control input. A simple objective minimizing the distance traveled, in the $\mathcal{W}^2$ case, is:

$$J = \sum_{k}^{K-1} \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2} \tag{83}$$

with an added $z$ state for the $\mathcal{W}^3$ case.

### A.3 Dynamic Constraints

Part of the challenge of transcribing the optimal control problem as a nonlinear optimization problem is transforming the dynamics from continuous time to a finite number of variables and constraints. This involves approximating the state derivatives $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$ with a numerical quadrature rule. To do this, there are many possibilities depending on the accuracy and complexity of the desired approximation. The simplest possible approximation is the Euler method:

$$\dot{x} \approx \frac{x_{k+1} - x_k}{h_k} = f(x_k, u_k) \tag{84}$$

Because of its simplicity, Euler's method has been widely used in previous motion planning under uncertainty and motion planning algorithms (Gonzalez, 2018), (Ono et al., 2013), (Blackmore et al., 2011). While simple, Euler's method is order $h$, which means its error scales as a function of the time step. This is relatively poor and typically higher order approximations are used.

For more complex dynamical systems such as the 3D Slocum glider model, this work relies on the trapezoidal method and Hermite-Simpson method. The trapezoidal method relies on the trapezoidal rule to integrate the dynamics:

$$x_{k+1} - x_k \approx \frac{1}{2} h_k \left( f\left(x_{k+1}, u_{k+1}\right) + f\left(x_k, u_k\right) \right) \tag{85}$$

Because both the left and right-hand sides are defined in terms of $x_{k+1}$ and it is generally difficult to solve for $x_{k+1}$ explicitly, the trapezoidal method is *implicit*. The trapezoidal rule has error of order $h^2$. In general, reducing $h_k$ by a factor of two will reduce the error by a factor of four. The Hermite-Simpson quadrature rule is slightly more complex:

$$x_{k+1} - x_k \approx \frac{1}{6} h_k \left( f\left(x_{k+1}, u_{k+1}\right) + 4f\left(x_{k+\frac{1}{2}}, u_{k+\frac{1}{2}}\right) + f\left(x_k, u_k\right) \right) \tag{86}$$

where the $k + \frac{1}{2}$ index indicates that the method requires a knot at the midpoint between knots $k$ and $k + 1$. While more complex than trapezoidal collocation, Hermite-Simpson quadrature has fourth order error, $\mathcal{O}\left(h^4\right)$ (Betts, 2009).

## Appendix B. Shooting Method Trajectory Optimization

Shooting methods are a general approach to trajectory optimization in the deterministic setting. They contrast with collocation by integrating the equations of motion given a set of control inputs. The inputs are then updated according to the error between the integrated trajectory and the desired trajectory. This idea is depicted in Fig. 20 in Section 9. A multiple shooting method has multiple simulations and defects along a single trajectory. Each defect measures the amount that a trajectory segment mismatches the following segment. In general, Jacobian matrices for shooting methods are more complex to generate because the state at each step $k$ relies on control inputs at time steps 0 to $k - 1$.

In this appendix, we first discuss how we simulate trajectories subject to additive Gaussian noise. Next, we discuss calculations required for the Jacobian matrix. Finally, we present our formulations for the objective function and region constraints.

### B.1 Simulating Dynamics via a Continuous Time Stochastic Process

The CDF-based risk model in Section 7 assumes risk is only computed at landmark regions; the sampling-based model of Section 8 samples and computes risk at collocation knots. Prior work such as (Ono et al., 2013) assumes a discrete time stochastic process. These models are disadvantageous for a couple of reasons. First, real world dynamical systems are described by continuous time differential equations. Using a discrete time stochastic process inherently requires some form of time discretization. Second, many of the numerical integration routines for nonlinear systems require precise specification of the time step. Many trajectory optimization routines (such as this article's collocation method) make the

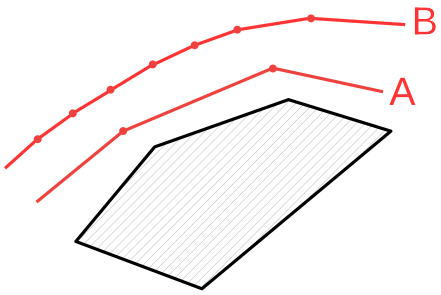



Figure 34: Illustration of how trajectories can become more conservative simply by decreasing time step size when the stochastic process variance is held constant. The diagram assumes the probability of failure is evaluated only at trajectory knots (red dots). For the same variance, trajectory A has fewer waypoints (a larger time step) and therefore fewer evaluations of path risk. This enables the optimizer to bring it closer to the obstacle while maintaining the overall chance constraint.

time step $h$ a decision variable that is changed by the optimizer. These issues raise the question of how to vary the stochastic model as the time step is changed for models that rely on a discrete time stochastic process. The simplest approach is to assume the stochastic model has a constant variance, i.e. the variance is not a function of the time step. This assumption creates the following contradiction: to satisfy the same chance constraint, a trajectory with a greater number of shorter time steps must be more conservative than a trajectory with a smaller number of longer time steps. This problem is illustrated in Fig. 34.

To solve these issues, for SMMC we model uncertainty in the agent's dynamics using a continuous time stochastic process. This allows us to connect the agent's continuous time dynamics with continuous time stochasticity, rather than employing a discrete time approximation. It also allows us to model how the stochastic process's variance varies with the integration time step $h$.

The continuous time stochastic model used in SMMC is a Wiener process. A Wiener process (Brownian motion) is a continuous time stochastic process that can be thought of as the limit of a random walk as the step size becomes small (Kloeden & Platen, 2011). A Wiener process $W$ is defined by three properties: $W_0 = 0$, $W$ as a function of time $t$ is continuous with probability 1, and $W$ has independent increments such that $W_{t+s} - W_s \sim \mathcal{N}(0, t)$ (Brzezniak & Zastawniak, 1999). The Wiener process is useful because it typically serves as the source of randomness in a stochastic differential equation (SDE). This SDE can be used to model the uncertain evolution in time of a dynamical system.

The general form of a stochastic differential equation (SDE) written in differential form is:

$$dX(t) = f(t, X(t)) + B(t, X(t))\, dW(t) \tag{87}$$

where $X$ is a random variable and $dW(t)$ represents a Wiener process. Variable $B$ represents a diffusion coefficient. For this work, only diffusion terms are considered where $B(t, X(t))$ is constant. Eq. (87) is written only symbolically to mean the integral equation (Kloeden

& Platen, 2011):

$$X_t = X_{t_0} + \int_{t_0}^{t} f\left(s, X\left(s\right)\right) ds + \int_{t_0}^{t} B\left(s, X\left(s\right)\right) dW_s \tag{88}$$

In general, analytic solutions to Eq. (88) exist only in the simplest cases and numerical routines must be used. Unfortunately, the same numerical integration procedures used for deterministic differential equations do not, in general, have analogs for their stochastic counterparts because of the different underlying calculuses. Fortunately, the simplest method, the Euler method, does have an analog for stochastic differential equations. For SDEs, it is known as the Euler-Maruyama method (Kloeden & Platen, 2011).

The Euler-Maruyama method approximates Eq. (88) via:

$$X_{k+1} = X_k + h_k f\left(t, X_k\right) + B\Delta W_k \tag{89}$$

where $\Delta W_k = W_{k+1} - W_k$, the change in the Brownian motion over the time interval $h_k = t_{k+1} - t_k$. Variable $\Delta W_k$ is a random variable with $\Delta W_k \sim \mathcal{N}\left(0, h_k\right)$. This enables the original SDE, Eq. (87), to be simulated in the following manner. At each time step, a normal random variable is sampled with $\mu = 0$ and $\sigma^2 = h_k$. Then the state is propagated forward using:

$$X_{k+1} = X_k + h_k f\left(t, X_k\right) + BZ \tag{90}$$

where $Z$ is the sampled random variable. Eq. (90) is used as the basis for the numerical simulation of the dynamics in the shooting method trajectory optimization.

We provide the Dubins car as an example, which is used to benchmark Shooting Method Monte Carlo in Section 10.5. Its equations of motion are:

$$\dot{x} = \cos\left(\theta\right) \tag{91}$$
$$\dot{y} = \sin\left(\theta\right) \tag{92}$$
$$\dot{\theta} = u_\theta \tag{93}$$

The stochastic differential equations for the Dubins car are:

$$dX_t = \cos\left(\theta\right) + \sigma_x dW_t \tag{94}$$
$$dY_t = \sin\left(\theta\right) + \sigma_y dW_t \tag{95}$$
$$d\Theta_t = u_\theta + \sigma_\theta dW_t \tag{96}$$

The equations are discretized using Eq. (90):

$$X_{k+1} = X_k + h_k \cos\left(\Theta_k\right) + \sigma_x Z_x \tag{97}$$
$$Y_{k+1} = Y_k + h_k \sin\left(\Theta_k\right) + \sigma_y Z_y \tag{98}$$
$$\Theta_{k+1} = \Theta_k + u_\theta + \sigma_\theta Z_\theta \tag{99}$$

where $Z_x$, $Z_y$, and $Z_\theta$ are samples drawn from a normal distribution with $\mu = 0$ and $\sigma^2 = h_k$. Eqs. (97) - (99) along with samples $Z_x$, $Z_y$, and $Z_\theta$ are used in the function *SimulateTrajectory*, Line 4 of Alg. 8.

## B.2 Formulation of the SMMC Mathematical Program

We illustrate how the objective and region constraints are formulated for SMMC. We first describe how the Jacobian matrix is calculated because it is used to derive Jacobians for the objective and region constraints. We derive equations for the 2D case; it is straightforward to extend them for 3D workspaces. Many of the calculations, particularly deriving the Jacobians, follow directly from the shooting method for deterministic systems.

### B.2.1 Computing the SMMC Jacobian Matrix

The Jacobian matrix is more complex to compute than in the collocation case because of the need to account for the effect of control inputs from time $t_0$ to $t_K$ on state $\mathbf{x}_{k,s}$. The Jacobian requires computing the derivative of the objective and constraint functions with respect to the decision variables, i.e. the control inputs and time steps:

$$\mathbf{J}(\mathbf{u}, \mathbf{h}) = \begin{bmatrix} \frac{\partial J}{\partial u_{0,0}} & \frac{\partial J}{\partial u_{1,0}} & \cdots & \frac{\partial J}{\partial h_K} \\ \frac{\partial C_0}{\partial u_{0,0}} & \frac{\partial C_0}{\partial u_{1,0}} & \cdots & \frac{\partial C_0}{\partial h_K} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial C_I}{\partial u_{0,0}} & \frac{\partial C_I}{\partial u_{1,0}} & \cdots & \frac{\partial C_I}{\partial h_K} \end{bmatrix} \tag{100}$$

where $J$ is the objective and $C_i$ is the $i^{th}$ constraint. Because the states $\mathbf{x}$ are no longer decision variables but are determined through simulation, they are functions of $\mathbf{u}$ and $\mathbf{h}$, i.e. $\mathbf{x}_{k,s}(\mathbf{u}_{0:k-1}, \mathbf{h}_{0:k-1})$. This means the value of constraint $i$ at time step $k$, $C_{i,k}$, is possibly dependent on control inputs at all prior time steps. Fortunately, the Jacobian computation may be simplified by propagating certain gradients forward in time. Suppose we want to compute $\frac{\partial C_{i,k'}}{\partial u_{p,k}}$, the change in constraint $C_{i,k'}$ due to a change in control input $u_{p,k}$ where $k' > k$. The gradient $\frac{\partial C_{i,k'}}{\partial u_{p,k}}$ may be simplified using the chain rule of differentiation as:

$$\frac{\partial C_{i,k'}}{\partial u_{p,k}} = \sum_{s}^{S} \sum_{m}^{M} \frac{\partial C_{i,k'}}{\partial x_{m,k',s}} \frac{\partial x_{m,k',s}}{\partial u_{p,k}} + \sum_{p}^{P} \frac{\partial C_{i,k'}}{\partial u_{p,k'-1}} \tag{101}$$

The first term in the first sum on the right-hand side, $\frac{\partial C_{i,k'}}{\partial x_{m,k',s}}$, denotes the change in the constraint due to the change in the state at $k'$ for scenario $s$. Its precise form depends on the constraint $C_{i,k'}$ and is covered in the following sections. The second term in the first sum represents the change in the state value $x_{m,k',s}$ due to the change in control input $u_{p,k}$ where $k' > k$. In the case $k' - 1 = k$, $x_{m,k',s}$ may be an explicit function of $u_{p,k}$ and the derivative is straightforward to compute using the dynamics. In the case $k' - 1 > k$, $x_{m,k',s}$ is not an explicit function of $u_{p,k}$, and it is necessary to compute this using the chain rule. Namely,

$$\frac{\partial x_{m',k',s}}{\partial u_{p,k}} = \sum_{m}^{M} \frac{\partial x_{m',k',s}}{\partial x_{m,k'-1,s}} \frac{\partial x_{m,k'-1,s}}{\partial u_{p,k}} \quad , k' - 1 > k \tag{102}$$

Note the recursive nature of Eq. (102). The RHS computes $\frac{\partial x_{m',k',s}}{\partial u_{p,k}}$ in terms of $x_{m,k'-1,s}$ while $\frac{\partial x_{m',k',s}}{\partial x_{m,k'-1,s}}$ can be computed from the dynamics. This suggests an approach for computing the gradients over the entire time horizon by maintaining terms $\frac{\partial x_{m',k',s}}{\partial u_{p,k}}$. Specifically, the dynamics equations are used to compute the initial terms:

$$\frac{\partial x_{m,1,s}}{\partial u_{p,0}} \ \forall \ m,p,s \tag{103}$$

Subsequently, these terms are used to compute $\frac{\partial x_{m,2,s}}{\partial u_{p,0}}$ using (102). This approach can be used to compute any $\frac{\partial x_{m,k',s}}{\partial u_{p,k}}$ for $k' > k$. As an example of this computation, consider the Dubins car dynamics using an Euler-Maruyama integration scheme as described in Eqs. (97) - (99):

$$X_{k+1} = X_k + h_k \cos(\Theta_k) + \sigma_x \sqrt{h_k} Z \tag{104}$$

$$Y_{k+1} = Y_k + h_k \sin(\Theta_k) + \sigma_y \sqrt{h_k} Z \tag{105}$$

$$\Theta_{k+1} = \Theta_k + u_\theta + \sigma_\theta \sqrt{h_k} Z \tag{106}$$

Eqs. (104) - (106) are equivalent to Eqs. (97) - (99) except Z is sampled from a standard normal distribution, i.e. $Z \sim \mathcal{N}(0,1)$. The partial derivatives of the states with respect to the decision variables are straightforward to compute:

$$\frac{\partial X_{k+1}}{\partial h_k} = \cos(\Theta_k) + \frac{\sigma_x Z}{2\sqrt{h_k}} \tag{107}$$

$$\frac{\partial Y_{k+1}}{\partial h_k} = \sin(\Theta_k) + \frac{\sigma_y Z}{2\sqrt{h_k}} \tag{108}$$

$$\frac{\partial \Theta_{k+1}}{\partial h_k} = \frac{\sigma_\theta Z}{2\sqrt{h_k}} \tag{109}$$

$$\frac{\partial \Theta_{k+1}}{\partial u_\theta} = 1 \tag{110}$$

These gradients can then be used with Eq. (102) to compute the general $\frac{\partial x_{m',k',s}}{\partial u_{p,k}}$.

Computation of the objective Jacobian proceeds similarly. Rather than compute the Jacobian for each of $I$ constraints, there is only one objective. Eq. (101) may be rewritten as:

$$\frac{\partial J}{\partial u_{p,k}} = \sum_{s}^{S} \sum_{m}^{M} \frac{\partial J}{\partial x_{m,k',s}} \frac{\partial x_{m,k',s}}{\partial u_{p,k}} + \sum_{p}^{P} \frac{\partial J}{\partial u_{p,k'-1}} \tag{111}$$

The method of computing gradients and propagating them forward is shown in Alg. 11.

### B.2.2 Objective

Similar to the collocation case in Appendix Section A.2, the Euclidean distance may be used as an objective function. In terms of states $(x, y)$, this is:

$$J(\mathbf{x}) = \sum_{k=0}^{K-1} \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2} \tag{112}$$

**Algorithm 11:** Method for computing the Jacobian matrix $\mathbf{J}$, Eq. (100), by propagating the gradients forward. $\mathbf{K}$ is a list containing all transition gradients, $\frac{\partial x_{m,k}}{\partial u_{p,k'}}$ for $k > k'$.

**Input:** A set of $I$ dynamics equations describing the evolution of the system state.
**Output:** The Jacobian matrix, $\mathbf{J}$

```
for s ← 1 to S do
    for k ← 1 to K do
        for x_m ∈ x_s, u_p ∈ u do
            Compute ∂x_{m,k,s}/∂u_{p,k-1} using Eq.(102) and store in K
        end
        for k' ← 1 to k do
            for i_k ← 1 to I_k do
                ∂C_{i_k,k}/∂u_{p,k'} ← ∂C_{i_k,k}/∂u_{p,k'} + Σ_m^M ∂C_{i,k}/∂x_{m,k-1,s} ∂x_{m,k-1,s}/∂u_{p,k'}
            end
        end
    end
end
```

For the shooting method, the dependence on the control inputs may be made explicit, because each $(x_k, y_k)$ state is computed as the result of a simulation dependent on $\mathbf{u}_{0:k-1}$:

$$J(\mathbf{u}) = \sum_{k=0}^{K-1} \sqrt{\left(x_{k+1}(\mathbf{u}_{0:k}) - x_k(\mathbf{u}_{0:k-1})\right)^2 + \left(y_{k+1}(\mathbf{u}_{0:k}) - y_k(\mathbf{u}_{0:k-1})\right)^2} \tag{113}$$

An advantage of SMMC is that the expected cost is straightforward to compute after performing the trajectory simulations. The expected cost is approximated by summing over all scenarios, from Eq. (71):

$$\mathbb{E}_{\hat{f}}[J(\mathbf{u})] = \frac{1}{S} \sum_{s}^{S} \sum_{k=0}^{K-1} \sqrt{\left(x_{k+1,s} - x_{k,s}\right)^2 + \left(y_{k+1,s} - y_{k,s}\right)^2} \tag{114}$$

### B.2.3 Objective Jacobian

Computing the Jacobian matrix requires the approach outlined in subsection B.2.1. What remains is to compute $\frac{\partial J}{\partial x_{m,k'}}$, the derivative of the objective with respect to the states. Given the objective in (114), there are four derivatives that must be computed: the derivative of the objective with respect to $x_{k+1}$, $x_k$, $y_{k+1}$, and $y_k$:

$$\frac{\partial J}{\partial x_{k+1,s}} = \frac{x_{k+1,s}}{\sqrt{\left(x_{k+1,s} - x_{k,s}\right)^2 + \left(y_{k+1,s} - y_{k,s}\right)^2}} \tag{115}$$

$$\frac{\partial J}{\partial x_{k,s}} = \frac{-x_{k,s}}{\sqrt{\left(x_{k+1,s} - x_{k,s}\right)^2 + \left(y_{k+1,s} - y_{k,s}\right)^2}} \tag{116}$$

$$\frac{\partial J}{\partial y_{k+1,s}} = \frac{y_{k+1,s}}{\sqrt{\left(x_{k+1,s} - x_{k,s}\right)^2 + \left(y_{k+1,s} - y_{k,s}\right)^2}} \tag{117}$$

$$\frac{\partial J}{\partial y_{k,s}} = \frac{-y_{k,s}}{\sqrt{\left(x_{k+1,s} - x_{k,s}\right)^2 + \left(y_{k+1,s} - y_{k,s}\right)^2}} \tag{118}$$

These equations are used along with Eq. (111) to compute the objective gradients.

#### B.2.4 Region Constraints

Similar to the collocation method, SMMC receives as input a sequence of regions, which is compiled into constraints on the waypoint states. As mentioned in the introduction to SMMC, unlike the collocation method, there is not a deterministic set of knot points or a nominal trajectory that can be constrained to lie on or in each region. Rather than constrain the waypoint state to lie on a region, the expected value of the waypoint state is required to lie in or on the region. For example, a landmark region constraint is written as $\mathbb{E}_{\hat{f}}\left[w_n\right] = 0$. In terms of the waypoint states $\boldsymbol{\omega}_{n,s} = (x, y)_{k_n,s}$, the constraint can be written:

$$\mathbb{E}_{\hat{f}}\left[w_n\right] \approx \frac{1}{S}\sum_{s}^{S}\left(\boldsymbol{\omega}_{n,s}\left(\mathbf{u}, \mathbf{h}\right) - \vec{o}_n\right) \cdot \vec{w}_n = 0 \tag{119}$$

Recall that $\vec{o}_n$ and $\vec{w}_n$ are functions of the landmark region's geometry; $\boldsymbol{\omega}_{n,s}$ is dependent on the simulated trajectory. The gradients are straightforward to compute:

$$\frac{\partial w_n}{\partial x_{k=n,s}} = \vec{w}_{n,x} \tag{120}$$

$$\frac{\partial w_n}{\partial y_{k=n,s}} = \vec{w}_{n,y} \tag{121}$$

These equations can be substituted in Eq. (101) for term $\frac{\partial C_{i,k'}}{\partial x_{m,k',s}}$ to compute the full gradients.

## Appendix C. Completeness and Termination of First Feasible Hybrid Search

We discuss termination and completeness of FFHS. Recall that we assume the environment consists of a set of closed continuous polytope obtacles $\boldsymbol{\mathcal{E}}$. The obstacles are without holes or self-intersections.

### C.1 Termination

We show termination of FFHS by considering the following cases: a feasible problem fails to terminate in the region planner, an infeasible problem fails to terminate in the region planner, and the trajectory planner's failure causes FFHS to fail to terminate. First, we state a number of lemmas.

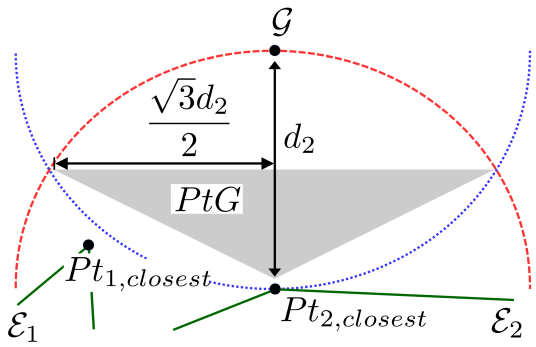



Figure 35: Geometry illustrating the width of the Polytope to Goal required for guaranteed termination. Similar geometry holds for a tetrahedron PtG in 3D.

**Lemma C.1.** Given two obstacle surfaces $\mathcal{E}_1$ and $\mathcal{E}_2$ with closest points to the goal $Pt_{1,closest}$ and $Pt_{2,closest}$ where the distance from $Pt_{2,closest}$ to the goal is less than the distance from $Pt_{1,closest}$ to the goal, a simplex Polytope to the Goal (PtG) shot from $Pt_{2,closest}$ will not return $\mathcal{E}_1$ as the closest collision if its base is less than $2\sqrt{3}$ times its height.

*Proof.* Consider the 2D geometry depicted in Fig. 35. Given two closed polytope obstacle surfaces, $\mathcal{E}_1$ and $\mathcal{E}_2$, each with closest points to the goal $Pt_{1,closest}$ and $Pt_{2,closest}$ where the distance from $Pt_{2,closest}$ to the goal, $d_2$, is less than $d_1$. In the figure, a PtG shown in gray is directed from $Pt_{2,closest}$ to the goal. We assume the PtG is a simplex, i.e. triangle in 2D and tetrahedron in 3D. The red dashed semicircle indicates the region within distance $d_2$ to $Pt_{2,closest}$. The blue dotted semicircle indicates points within distance $d_2$ to $\mathcal{G}$. $Pt_{1,closest}$ must be outside of the blue semicircle because $d_2 < d_1$. Line 6 of Alg. 3, *CollisionChecker*, returns the closest collision to $Pt_{2,closest}$. The subroutine will return $Pt_{1,closest}$ if it is intersected by the PtG and if $Pt_{1,closest}$ is closer to $Pt_{2,closest}$ than $\mathcal{G}$. This is not possible if the PtG remains within the intersection of the red and blue semicircles as pictured in Fig. 35, i.e. the PtG's width is less than $\sqrt{3}d_2$ at the intersection of the semicircles. If the triangular PtG's width is less than $2\sqrt{3}$ its height, the closest collision to $Pt_{closest,2}$ is guaranteed to be $\mathcal{G}$. □

**Lemma C.2.** Given a landmark region $\mathcal{L}_{twice}$ that repeatedly causes path loops and triggers calls to *AvoidPathLoop*; the algorithm will eventually reach a state when all executions of *AvoidPathLoop* due to $\mathcal{L}_{twice}$ return $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$ or, in the case of infeasibility, failure.

*Proof.* Lemma C.2 follows from the fact that *AvoidPathLoop* maintains a visited list of landmark regions added to its queue. Given landmark region $\mathcal{L}_{twice}$, which triggers a call to *AvoidPathLoop* and is incident to obstacle surface $\mathcal{E}$, and the set of all $\boldsymbol{\mathcal{L}}$ also incident to $\mathcal{E}$, the visited list $\boldsymbol{\mathcal{L}}_{visited}$ ensures that each $\mathcal{L} \in \boldsymbol{\mathcal{E}}$ will be popped from priority queue $PriorityQueue_{\mathcal{L}}$ at most once as the final landmark region on a path. Therefore, there are a finite number of calls to *AvoidPathLoop* due to $\mathcal{L}_{twice}$ until $PriorityQueue_{\mathcal{L}}$ is empty. After this, $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$ will always be returned if *AvoidPathLoop* does not fail on Line 8. That *AvoidPathLoop* will eventually find the closest point on $\mathcal{E}$ to the goal, $Pt_{closest}$,

follows from continuity of the polytope and that all $\mathcal{L}$ incident to $\mathcal{E}$ will eventually be enumerated. □

**Definition C.3.** If *AvoidPathLoop* has popped all landmark regions sharing surface $\mathcal{E}$ with $\mathcal{L}_{twice}$ and returns $\mathcal{P}_{\mathcal{L}_{twice}:\mathcal{L}_{closest}}$, landmark region $\mathcal{L}_{twice}$ is said to be *explored*.

**Lemma C.4.** Given the set of polytope obstacle surfaces $\boldsymbol{\mathcal{E}}$ intersected by a line drawn from $\mathcal{I}$ to $\mathcal{G}$, if it does not first fail nor find the goal, FFHS will eventually find the closest point on each surface $\mathcal{E} \in \boldsymbol{\mathcal{E}}$ to the goal.

*Proof.* This follows from Lemmas C.1 and C.2. By induction, assume that there is exactly one surface $\mathcal{E}_1$ intersected by a line from $\mathcal{I}$ to $\mathcal{G}$ . Lemma C.2 shows that, if FFHS does not first find the goal, *AvoidPathLoop* will eventually find $Pt_{1,closest}$. For the inductive step, assume Lemma C.4 holds for $n$ surfaces and that an $n+1^{th}$ surface is inserted between the closest point to the goal on the $n^{th}$ surface and the goal. Lemma C.1 shows that a PtG satisfying certain geometric constraints will intersect the $n+1^{th}$ surface as the closest collision. Lemma C.2 implies that *AvoidPathLoop* will eventually find the closest point on this surface if it does not first terminate. □

### C.1.1 Termination of the Region Planner for Feasible Problems

In the case of feasible problems with a PtG that satisfies the conditions of Lemma C.1, the only way that the region planner would fail to terminate and return a candidate path to the trajectory planner is if there were a sequence of explored landmark regions $\boldsymbol{\mathcal{L}}$, each incident to different obstacles surfaces $\boldsymbol{\mathcal{E}}$, that generated a loop. We use induction to show this is impossible. Assume the environment consists of two closed polytope surfaces $\mathcal{E}_1$ and $\mathcal{E}_2$ with $\mathcal{E}_2$ closest to the goal, i.e. $d_2 < d_1$. Assume the algorithm has reached a state where $\mathcal{L}_1$ incident to $\mathcal{E}_1$ and $\mathcal{L}_2$ incident to $\mathcal{E}_2$ are both explored landmark regions. Assume a PtG from $Pt_{closest,1}$ intersects $\mathcal{L}_2$, which triggers a call to *AvoidPathLoop* and returns $\mathcal{P}_{\mathcal{L}_2:\mathcal{L}_{closest}}$, because $\mathcal{L}_2$ is explored. The algorithm would not terminate if a PtG directed from $Pt_{closest,2}$ to the goal potentially intersected $\mathcal{L}_1$ as the closest collision. However, Lemma C.1 shows that for a PtG less than a certain width, this is impossible. A similar argument is used for the inductive step. Assume the environment consists of $n$ surfaces for which the algorithm terminates. Assume an $n+1^{th}$ surface is added which has a point $Pt_{closest,n+1}$ closer to $\mathcal{G}$ than any other $Pt_{closest,n}$. A PtG shot from $Pt_{closest,n+1}$ will not return as its closest intersection any of the surfaces in $\boldsymbol{\mathcal{E}}_n$ by Lemma C.1. This means that no loop will form between any of the explored landmark regions. Hence, the algorithm will terminate.

Due to Lemma C.1, termination is only guaranteed for PtG's less than a certain width. Because the PtG is a heuristic that seeks to approximate the agent's state distribution, it is possible to reduce its size to meet the geometric constraint in Lemma C.1. However, systems that exceed this bound suggest very high levels of trajectory uncertainty that may make planning unrealistic.

### C.1.2 Termination of the Region Planner for Infeasible Problems

Let a separating surface be a polytope surface that divides the workspace with $\mathcal{I}$ on one side of this surface and $\mathcal{G}$ on the other. Let $\mathcal{E}_{separating}$ be the first separating surface that

a line from $\mathcal{I}$ to $\mathcal{G}$ encounters. FFHS will fail on the closest point of this surface to the goal. That the algorithm will find the closest point follows from Lemma C.4. To terminate, *AvoidPathLoop* checks the sign of the dot product between the surface normal at this point and a vector from this point to the goal, Line 7 of Alg. 4. For a separating surface, this dot product will be negative because the closest point on $\mathcal{E}_{separating}$ will not be visible from the goal.

#### C.1.3 Termination in Cases of Trajectory Planner Failure

If the deterministic problem is feasible but the stochastic problem is infeasible such that the trajectory planner fails, termination depends on the chance constraint model. If the trajectory is validated post-optimization, a check is made in *ValidateTrajectory* to determine if the pair of landmark regions $(\mathcal{L}_n, \mathcal{L}_{n+1})$ surrounding infeasible trajectory states $(\boldsymbol{x}_k, \boldsymbol{x}_{k+1})$ has previously caused an infeasibility, Line 4 of Alg. 7. If so, the algorithm returns failure. For sampling-based chance constraints, which do not rely on $ValidateTrajectory$, a check should be performed if the same path has been optimized twice. While this check can be complex, trajectory optimization via sampling-based chance constraints typically dominates the algorithm's runtime compared to the region planner. In both of these cases, the algorithm is guaranteed to terminate.

### C.2 Completeness

We prove completeness of FFHS for the deterministic case and a point robot. We avoid the stochastic case because trajectory feasibility is dependent on assumptions regarding stochasticity. In the deterministic case, the PtG in Alg. 3 collapses to a line; Lemma C.1 always holds. We assume the point robot may travel infinitesimally close to the obstacle surface. Waypoint states $\boldsymbol{\omega}$ constrained to lie in landmark region $\mathcal{L}$ may be also be placed infinitesimally close to the subfacet incident to $\mathcal{L}$, e.g. in 2D they lie infinitesimally close to the obstacle vertex supporting $\mathcal{L}$. In 3D, we assume $\boldsymbol{\omega}$ lie at the point on $\mathcal{L}$ closest to $\mathcal{G}$. For the discussion of completeness, we assume the trajectory planner will always succeed and $ValidateTrajectory$ return true if the subroutine receives an obstacle-free path.

In this simplified case, completeness follows from Lemma C.4 and the fact that, for feasible problems, FFHS will eventually find the closest point on each surface that intersects a line drawn from $\mathcal{I}$ to $\mathcal{G}$ if it does not first find the goal. FFHS would only fail prematurely if there were a surface between $\mathcal{I}$ and $\mathcal{G}$ where the closest point on this surface was not visible to the goal, i.e. a separating surface between $\mathcal{I}$ and $\mathcal{G}$. However, we have assumed the deterministic problem is feasible so there can be no such surface. It is then straightforward to show that if FFHS finds the closest point to the goal on each of these surfaces, it will find the goal via a PtG satisfying Lemma C.1.

## Appendix D. Computing $\mathcal{W}_{int}$ for Non-Convex Obstacles

In this appendix, we describe a technique that allows approximation of $\mathcal{W}_{int}$ for 3D environments with non-convex obstacles. It is possible to transform arbitrary obstacles into a set of convex obstacles; however, the process of finding a good convexification is a hard problem (Deits & Tedrake, 2015). A popular technique is to transform environments into

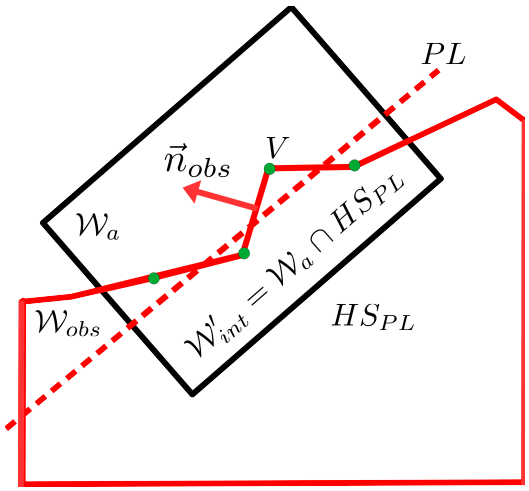



Figure 36: Illustration of the method to compute an approximation of the intersecting region $\mathcal{W}_{int}$ for non-convex obstacles and a convex agent $\mathcal{W}_a$. The key is to approximate the obstacle surface (outlined in red) that lies inside the agent with a plane, $PL$. How this plane is constructed depends on the geometry; in the image, a set of obstacle vertices $\boldsymbol{V}$ (shown in green) lie inside $\mathcal{W}_a$. The vertices' positions and outward normals are averaged together to generate $PL$; the normal of $PL$ creates the halfspace $HS_{PL}$. The intersecting volume $\mathcal{W}'_{int}$ is computed as the intersection of halfspace $HS_{PL}$ with $\mathcal{W}_a$.

collections of cuboids or voxels (Jetchev & Toussaint, 2010); however, a priori insight is required to properly size the cuboids. We propose a different approach to approximate $\mathcal{W}_{int}$ for convex $\mathcal{W}_a$ but possibly non-convex $\mathcal{W}_{obs}$. The insight is that $\mathcal{W}_{int}$ does not need to be precisely computed. Instead, we compute a proxy for $\mathcal{W}_{int}$, denote this as $\mathcal{W}'_{int}$, that has the following properties. First, $\mathcal{W}'_{int} = 0$ if the sample agent configuration is not in intersection and $\mathcal{W}_{int}\prime > 0$ if it is. Second, $\mathcal{W}'_{int}$ should increase as the agent and obstacles become more in collision and should decrease as they become less in collision. These criteria suggest the following approach to modeling $\mathcal{W}'_{int}$. Rather than compute the polyhedron $\mathcal{W}_{int}$ via clipping $\mathcal{W}_a$ with possibly non-convex $\mathcal{W}_{obs}$, the facets of $\mathcal{W}_{obs}$ lying inside $\mathcal{W}_a$ are approximated via a plane $PL$. The intersection of convex $\mathcal{W}_a$ with $PL$ is convex and its volume can be computed via a standard clipping algorithm. The method is illustrated graphically in Fig. 36.

Specifically, $\mathcal{W}_a$ is the convex polyhedron describing the region of the workspace occupied by the agent with rigid body $\mathcal{B}$ that is in intersection with the possibly non-convex obstacle polyhedron $\mathcal{W}_{obs}$. Let $\boldsymbol{\mathcal{F}}$ be the triangular facets of $\mathcal{W}_{obs}$ which lie inside $\mathcal{W}_a$ at least at one point. In order to specify the plane $PL$, we must specify a point in the plane and its normal vector. There are three cases to consider.

1. At least one vertex $V$ of $\boldsymbol{\mathcal{F}}$ lies inside $\mathcal{W}_a$. If this is the case, all vertices lying inside $\mathcal{W}_a$ are enumerated. Given this set of vertices $\boldsymbol{V}$, the approximating plane $PL$ is generated by averaging the position and inward pointing normal vectors of $\boldsymbol{V}$.

2. No vertices of $\boldsymbol{\mathcal{F}}$ lie inside $\mathcal{W}_a$ but the edge of at least one facet in $\boldsymbol{\mathcal{F}}$ does. Let the set of all edges of $\boldsymbol{\mathcal{F}}$ that lie in $\mathcal{W}_a$ be $\boldsymbol{E}$. Define a new line segment $E'$ which is

created by averaging each endpoint of the edges in $\boldsymbol{E}$. $E'$ is guaranteed to intersect $\mathcal{W}_a$. A point on $E'$ defines a point on $PL$. The normal vector of $PL$ is generated by averaging the inward pointing normal vectors of the facets adjacent to edges in $\boldsymbol{E}$; this attempts to approximate the surface of $\mathcal{W}_{obs}$.

3. Finally, neither vertices nor edges of $\boldsymbol{\mathcal{F}}$ lie in $\mathcal{W}_a$. If this is the case, $\boldsymbol{\mathcal{F}}$ must be of cardinality 1; subsequently, the plane that supports this facet is used as $PL$.

The normal vector of $PL$ (pointing inside the obstacle surface) defines a halfspace with which $\mathcal{W}'_{int}$ can be computed via a clipping algorithm. Specifically, $\mathcal{W}'_{int} = \mathcal{W}_a \cap HS_{PL}$ where $HS_{PL}$ is the halfspace defined by $PL$. This region, $\mathcal{W}'_{int}$, can then be used in Eq. (31) to approximate the chance constraint.

## References


Anton, H. (1999). *Calculus: A New Horizon*. John Wiley & Sons.

Berg, M. d., Cheong, O., Kreveld, M. v., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications* (3rd edition). Springer-Verlag TELOS, Santa Clara, CA, USA.

Betts, J. T. (2009). *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming* (2nd edition). Cambridge University Press, New York, NY, USA.

Bhatia, A., Kavraki, L. E., & Vardi, M. Y. (2010). Sampling-based motion planning with temporal goals. In *2010 IEEE International Conference on Robotics and Automation*, pp. 2689–2696.

Bhatia, A., Maly, M. R., Kavraki, L. E., & Vardi, M. Y. (2011). Motion planning with complex goals. *IEEE Robotics Automation Magazine*, *18*(3), 55–64.

Blackmore, L. (2008). Robust path planning and feedback design under stochastic uncertainty. In *American Institute of Aeronautics and Astronautics Guidance, Navigation and Control*.

Blackmore, L., Li, H., & Williams, B. S. (2006). A probabilistic approach to optimal robust path planning with obstacles. *2006 American Control Conference*, 7 pp.

Blackmore, L., Ono, M., Bektassov, A., & Williams, B. C. (2010). A probabilistic particle-control approximation of chance-constrained stochastic predictive control. *IEEE Trans. Robotics*, *26*(3), 502–517.

Blackmore, L., Ono, M., & Williams, B. C. (2011). Chance-constrained optimal path planning with obstacles. *IEEE Trans. Robotics*, *27*(6), 1080–1094.

Breen, D., Regli, W., & Peysakhov, M. (2020). Backface culling, z-buffering, ray casting, ray tracing radiosity. `https://www.cs.drexel.edu/~david/Classes/CS430/Lectures/L-15_CullingZbufRays.pdf`.

Brzezniak, Z., & Zastawniak, T. (1999). *Basic Stochastic Processes*. Springer-Verlag, London.

Choset, H. (2010). Robotic motion planning: Configuration space. `https://www.cs.cmu.edu/~motionplanning/lecture/Chap3-Config-Space_howie.pdf`.

Cooney, L. (2016). Expanding the capabilities of the Slocum glider. *OCEANS 2016 MTS/IEEE Monterey*, 1–5.

Cui, M. L., Harabor, D. D., & Grastien, A. (2017). Compromise-free pathfinding on a navigation mesh. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pp. 496–502. AAAI Press.

Davis, R., C. Eriksen, C., & P. Jones, C. (2002). Autonomous buoyancy-driven underwater gliders. In Griffiths, G. (Ed.), *The Technology and Applications of Autonomous Underwater Vehicles*. Taylor and Francis.

Deits, R., & Tedrake, R. (2015). *Computing Large Convex Regions of Obstacle-Free Space Through Semidefinite Programming*, pp. 109–124. Springer International Publishing, Cham.

Elfes, A. (1989). Using occupancy grids for mobile robot perception and navigation. *Computer*, *22*(6), 46–57.

Ericson, C. (2005). *Real-Time Collision Detection*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco.

Fainekos, G., Kress-Gazit, H., & Pappas, G. (2005). Temporal logic motion planning for mobile robots. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 2020–2025.

Faust, A., Ramirez, O., Fiser, M., Oslund, K., Francis, A., Davidson, J., & Tapia, L. (2018). PRM-RL: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane*, pp. 5113–5120, Brisbane, Australia.

Fernandez-Gonzalez, E., Williams, B. C., & Karpas, E. (2018). ScottyActivity: Mixed Discrete-Continuous Planning with Convex Optimization. *Journal of Artificial Intelligence Resesearch (JAIR)*, *62*, 579–664.

Freund, R. M. (2004). Issues in Non-Convex Optimization. `https://ocw.mit.edu/courses/sloan-school-of-management/15-094j-systems-optimization-models-and-computation-sma-5223-spring-2004/lecture-notes/non_convex_prob.pdf`.

Genz, A. (1992). Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, *1*(2), 141–149.

Gill, P. E., Murray, W., Saunders, M. A., & Wong, E. (2017). User's guide for SNOPT 7.6: Software for large-scale nonlinear programming. Center for Computational Mathematics Report CCoM 17-1, Department of Mathematics, University of California, San Diego, La Jolla, CA.

Gonzalez, E. F. (2018). *Generative multi-robot task and motion planning over long horizons*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA 02139.

Gurobi Optimization, Inc. (2014). Gurobi optimizer reference manual. `gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf`.

Hrabar, S. (2008). 3D path planning and stereo-based obstacle avoidance for rotorcraft UAVs. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 807–814.

Ichter, B., Schmerling, E., Agha-mohammadi, A., & Pavone, M. (2017). Real-time stochastic kinodynamic motion planning via multiobjective search on GPUs. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore*, pp. 5019–5026.

J. Kuffner, Jr., J., & Lavalle, S. M. (2000). RRT-Connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 995–1001.

Janson, L., Schmerling, E., & Pavone, M. (2018). *Monte Carlo Motion Planning for Robot Trajectory Optimization Under Uncertainty*, pp. 343–361. Springer International Publishing, Cham.

Jasour, A. M., Hofmann, A., & Williams, B. C. (2018). Moment-sum-of-squares approach for fast risk estimation in uncertain environments. In *2018 IEEE Conference on Decision and Control (CDC)*, pp. 2445–2451.

Jetchev, N., & Toussaint, M. (2010). Trajectory prediction in cluttered voxel environments. In *2010 IEEE International Conference on Robotics and Automation*, pp. 2523–2528.

Jimenez, P., Thomas, F., & Torras, C. (1998). Collision detection algorithms for motion planning. In Laumond, J.-P. (Ed.), *Robot Motion Planning and Control*, chap. 6, pp. 305–343. Springer.

Johnson, A. (2010-2014). Clipper - an open source freeware library for clipping and offsetting lines and polygons. `http://www.angusj.com/delphi/clipper.php`.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. `http://www.scipy.org/`.

Kelly, M. (2017). An introduction to trajectory optimization: How to do your own direct collocation. *SIAM Review*, *59*, 849–904.

Kettner, L. (2018). Halfedge data structures. In *CGAL User and Reference Manual* (4.13 edition). CGAL Editorial Board.

Kloeden, P., & Platen, E. (2011). *Numerical Solution of Stochastic Differential Equations*. Stochastic Modelling and Applied Probability. Springer Berlin Heidelberg.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press, New York, NY, USA.

LaValle, S. M., & Kuffner Jr., J. J. (2000). Rapidly-exploring random trees: Progress and prospects. `http://msl.cs.uiuc.edu/~lavalle/papers/LavKuf01.pdf`.

Léauté, T. (2005). Coordinating agile systems through the model-based execution of temporal plans. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA 02139.

Leonard, N. E., & Graver, J. G. (2001). Model-based feedback control of autonomous underwater gliders. *IEEE Journal of Oceanic Engineering*, *26*(4), 633–645.

Li, H. X. (2010). *Kongming: A Generative Planner for Hybrid Systems with Temporally Extended Goals*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA 02139.

Luders, B., Kothari, M., & How, J. P. (2010). Chance-constrained RRT for probabilistic robustness to environmental uncertainty. In *AIAA Guidance, Navigation, and Control Conference (GNC)*, Toronto, Canada.

Ono, M., Williams, B. C., & Blackmore, L. (2013). Probabilistic planning for continuous dynamic systems under bounded risk. *Journal of Artificial Intelligence Research (JAIR)*, *46*, 511–577.

Owen, A. B. (2013). *Monte Carlo theory, methods and examples*. Stanford University.

Patil, S., van den Berg, J., & Alterovitz, R. (2012). Estimating probability of collision for safe motion planning under Gaussian motion and sensing uncertainty. In *IEEE International Conference on Robotics and Automation, ICRA 2012, St. Paul, Minnesota, USA*.

Plaku, E., Kavraki, L. E., & Vardi, M. Y. (2009). Hybrid systems: from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*, *34*(2), 157–182.

Powell, D., & Abel, T. (2015). An exact general remeshing scheme applied to physically conservative voxelization. *Journal of Computational Physics*, *297*, 340 – 356.

Prékopa, A. (1995). *Stochastic Programming*. Kluwer Academic Publishers Group, Dordrecht; Norwell, MA.

Rudnick, D. L., Davis, R. E., Eriksen, C. C., Fratantoni, D. M., & Perry, M. J. (2004). Underwater gliders for ocean research. In *Marine Technology Society Journal*, pp. 48–59.

Santos, F. (2012). Diameter of polytopes and the Hirsch conjecture. `https://www3.math.tu-berlin.de/combi/MDS/summerschool12-material/santos_1.pdf`.

Sucan, Ioan A. and Kavraki, Lydia E. (2010). *Kinodynamic Motion Planning by Interior-Exterior Cell Exploration*. Springer Berlin Heidelberg.

Triantafyllou, M. (2004). Kinematics of moving frames. `https://ocw.mit.edu/courses/mechanical-engineering/2-154-maneuvering-and-control-of-surface-and-underwater-vehicles-13-49-fall-2004/lecture-notes/lec1.pdf`.

van den Berg, J., Abbeel, P., & Goldberg, K. Y. (2011). LQG-MP: Optimized path planning for robots with motion uncertainty and imperfect state information. *International Journal of Robotics Research*, *30*(7), 895–913.

Vasile, C. I., & Belta, C. (2014). Reactive sampling-based temporal logic path planning. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4310–4315.

Vatti, B. R. (1992). A generic solution to polygon clipping. *Commun. ACM*, *35*(7), 56–63.

Weisstein, E. W. (2018). Bivariate normal distribution. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/BivariateNormalDistribution.html`.

Ziegler, J., & Stiller, C. (2009). Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1879–1884.