Bendik Stuevold Eger

Trajectory Planning for Fixed-wing Unmanned Aerial Vehicles in Real World Terrain Data

TTK4900

Master's thesis in Cybernetics and Robotics
Supervisor: Lars Struen Imsland
January 2020

Bendik Stuevold Eger

Trajectory Planning for Fixed-wing Unmanned Aerial Vehicles in Real World Terrain Data

TTK4900

Master's thesis in Cybernetics and Robotics
Supervisor: Lars Struen Imsland
January 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

To my friends and family

Summary

This thesis presents an implementation of a system capable of planning airplane-like trajectories in real world terrain data intended for transmission mast inspection via fixed-wing unmanned aerial vehicles (UAV). A contribution made to the Open Motion Planning Library embedding Dubins airplane model as a state space class is also presented. This class serves as a local planning method in a sample-based planning context to give efficient closed-form solutions to the boundary value problem connecting UAV configuration pairs. The system uses the Flexible Collision Library combined with the OctoMap framework to model three different scenarios. The results of several simulations, using different planning algorithms, in all scenarios are presented. These simulations aim to find a good choice of planning algorithm for the implemented system. The results indicate that the Informed RRT* algorithm is a good choice when planning in the presented environments.

Sammendrag

Denne oppgaven presenterer en implementasjon av en system i stand til å planlegge fly-lignende baner i realistisk terreng data ment for inspeksjon av radiomaster og strømnettverk via *fixed wing*-drone. Et bidrag til rammeverket *Open Motion Planning Library* som implemeterer Dubins flymodell som en *state space*-klasse er også presentert. Denne klassen fungerer som en lokal planlegging metode i kontekst av *sample*-basert planlegging for å gi en effektiv analytisk løsning til randverdiproblemet som kobler sammen et konfigurasjoner for dronen. Systemet bruker rammeverkene *Flexible Collision Library* og *OctoMap* til å modellere tre ulike scenarioer. Resultater fra ulike simuleringer, med ulike planleggings-algoritmer, i alle scenarioene er presentert.Disse simuleringene har som mål å finne et godt valg av planleggings-algoritme for det implementerte systemet. Resultatene indikerer at *Informed RRT**-algoritmen er et godt valg for å planlegge i de presenterte miljøene.

Preface

The work presented in this thesis, and the thesis itself, collectively make the submission to my master thesis and final work regarding my masters degree at the Norwegian University of Science and Technology (NTNU). The topic of the project was suggested by the Norwegian robotics company KVS Technologies, but the work presented in this thesis has been performed by me independently. I am grateful for the encouraging conversations I have had with both my NTNU supervisor, Lars Struen Imsland, and my contact at KVS, Helge-André Langåker.

Table of Contents

Chapter 1

Introduction

Homo Sapiens – wise man – a name asserting the importance of our intelligence. Human's ability to think rational and reasonable is a trait we seek to recreate through the field of artificial intelligence. We have a unique ability to make complex decisions in a manner that seems almost trivial. E.g. we recognize faces and sounds when we interact with other people, we know where to place our feet to maintain balance while running and we plan how to spend our time days (if not months) in advance. For a machine to solve the same tasks each problem must be decomposed and translated to a logical form understandable for a computer. In the field of navigation and path planning, e.g. for an autonomous car, this involves sensing and comprehending the surrounding environment, understanding the dynamic restrictions of the car and being able to make the right decisions to bring the car to its destination is a safe manner.

The work presented in this thesis relates to these subjects, not through navigation of an autonomous car, but through path planning for fixed-wing unmanned aerial vehicles (UAVs). UAVs are widely used in today's world to perform tasks deemed too tedious or dangerous for humans to perform. This might be inspection of remotely located equipment such as transmission masts or grid towers, emergency aid deliveries in disaster struck areas or the more macabre scenario of military ”defense” systems – the latter being least motivational topic.

The project summarized in this thesis aim to contribute to a solution for automating long range trajectory planning for fixed-wing UAVs to perform inspection of remote transmission masts. The work is a spiritual successor of a previous project by Eger (2019). The previous project evaluated how the choice of heuristic function affected the use of a weighted A* algorithm to plan optimal paths in a grid like terrain representation. This project will rely on sample-based path planning to avoid explicitly modeling the environment all together. Path optimality will instead be implemented through using algorithms proven to be asymptotically optimal, meaning a resulting path will converge towards a cost minimum as the algorithm is given more iterations.

This thesis will present an implemented of a path planning system based on the Open Motion Planning Library (OMPL) (Şucan et al., 2012). OMPL is a robotic motion planning framework consisting of state-of-the-art sample-based planning methods. To build a planner capable of finding trajectories feasible for a fixed wing UAV a contribution integrating the Dubins airplane system into the base version of OMPL has been made. This is inspired by a similar work presented in a master thesis by Schneider (2016).

An outline for this project was developed in collaboration with the Norwegian robotics company KVS Technologies, and the list of requirements formulated for such a planning system is given below.

- The planner should take into account the system dynamics of the UAV and find a feasible path considering these dynamics.
- It should find an efficient path through the environment.
- The system should plan a trajectory through real wold terrain data.
- The planner should take into consideration eventual no-fly zones.
- The planner should maintain a safe volume around the UAV keeping a minimal distance to any terrain or no-fly zones.
- The planner should include several checkpoints in the trajectory. The checkpoints should be given as an ordered list of poses (coordinate and heading).

1.1 Outline

The remainder of the thesis is divided into the follow parts:

- **Theory**: Introduces the theoretical background and topics used throughout the rest of this thesis.
- **System implementation**: Presents the implemented system.
- **Results and simulations**: Presents what simulations were run to test the implemented system and determine what parameters are best suited for said system.
- **Discussion**: Discusses the results presented in the chapter prior. Also briefly mentions what topics should be of interest with regards to further work.
- **Conclusion**: Concludes the thesis.

Chapter 2

Background Theory

The following chapter will introduce the theoretical background used throughout this thesis. The book by LaValle (2006) is used as a main source for these topics, as well as the inspiration for most of the figures presented in this chapter.

The chapter will start off by introducing several topics within the field of motion planning needed to understand the implemented system. This includes:

- Planning with and without regarding the constraints sat by a dynamic system.
- Sample-based motion planning.
- Local planning methods and how this relates to Dubins system for a car and airplane.
- An introduction of several asymptotically optimal planning algorithms.

Lastly the chapter will briefly introduce the software frameworks used to implement the planning system.

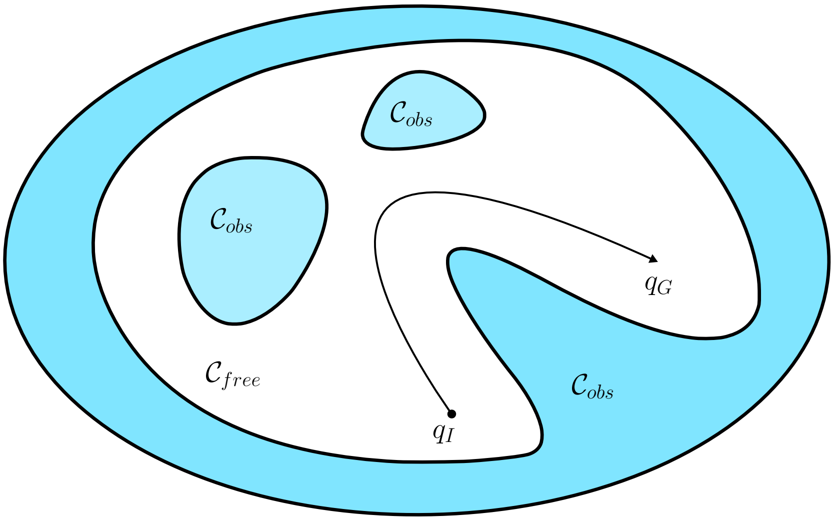
2.1 The piano mover's problem

This section will give a formal definition of the motion planning problem not including differential constraints given by a dynamic system. This definition only includes constraints sat by geometric and kinematic restrictions. This formulation is called the *piano mover's problem*.

First, lets define the *world* $\mathcal{W} = \mathbb{R}^3$ (can also be $\mathbb{R}^2$). The world generally contains two kinds of entities:

- **Obstacles:** Portions of the world that are permanently occupied. For example the walls of a building or terrain in nature.
- **Robots:** Bodies that are modeled geometrically and are controllable via a motion plan.

Figure 2.1 The basic motion planning problem is illustrated in this very simple conceptualization of C-spaces. The task is to find a path from q_I to q_G within $\mathcal{C}_{free}$. The content of the entire ellipse represents $\mathcal{C} = \mathcal{C}_{free} \cap \mathcal{C}_{obs}$.



Let the *obstacle region* $\mathcal{O}$ denote the set of all points located within obstacles in the world, $\mathcal{O} \subset \mathcal{W}$. Furthermore, let the geometry of a rigid body *robot* be defined by $\mathcal{A} \subset \mathcal{W}$. Let the *state space*, or *configuration space* as used by LaValle (2006), $\mathcal{C}$ define the set of rigid body transformations mapping the robot geometry, $\mathcal{A}$, into $\mathcal{W}$ via a function $h(q) : \mathcal{A} \to \mathcal{W}$, where $q \in \mathcal{C}$ defines a locational state called *configuration* of $\mathcal{A}$.

The set of configurations where the robot geometry intersects with the obstacle region is defined as:

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}.$$

The leftover set of configurations is called the *free space* and is defined as:

$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}.$$

Given an initial configuration, $q_I \in \mathcal{C}_{free}$, and a goal configuration, $q_G \in \mathcal{C}_{free}$, this is the set of configurations a motion planning algorithm must traverse to solve the motion planning problem. An illustration of the is given in Figure 2.1.

2.2 Motion planning with differential constrains

While solving the piano mover's problem is sufficient for finding a path, in some cases it puts a lot of responsibility in the robot control system's ability to follow this path. There is no consideration taken to account for the robots dynamics and the differential constrains of the system. In the case of a fixed wing UAV this can lead to problems, especially if the path is planned in low altitude and penetrating terrain. A motion planner not taking into consideration a minimum turning radius and maximum climb angle might plan a path unfeasible to navigate by the UAV's autopilot.

This calls for an expansion of the planning problem's formulation to take account for the dynamics of the system. The following is a problem formulation given by LaValle (2006):

Formulation 2.1: Motion planning with differential constrains

1. A *world* $\mathcal{W}$, a *robot* $\mathcal{A}$ an *obstacle region* $\mathcal{O}$ and a *configuration space* $\mathcal{C}$, which are defined the same as for the piano mover's problem.

2. An unbounded *time interval* $T = [0, \infty)$.

3. A smooth manifold X, called the *state space*, which may be $X = \mathcal{C}$ or it may be a phase space derived from $\mathcal{C}$ if dynamics are considered. Let $\kappa : X \to \mathcal{C}$ denote a function that returns the configuration $q \in \mathcal{C}$ associated with $x \in X$. Hence, $q = \kappa(x)$.

4. An obstacle region X_{obs} is defined for the state space. If $X = \mathcal{C}$ then $X_{obs} = \mathcal{C}_{obs}$. The notation $X_{free} = X \setminus X_{obs}$ indicates the states that avoid collision and satisfy any additional global constrains.

5. For each state $x \in X$, a bounded *action space* $U(x) \subseteq \mathcal{R}^m \cup x_T$, which includes the termination action u_T and m is some fixed integer called the *number of action variables*. Let U denote the union of $U(x)$ over all $x \in X$.

6. A system is specified by $\dot{x} = f(x, u)$, defined for every $x \in X$ and $u \in U$.

7. A state $x_I \in X_{free}$ is designated as the *initial state*.

8. A set $X_G \subset X_{free}$ is designated as the *goal region*.

9. A complete algorithm must compute an *action trajectory* $\tilde{u} : T \to U$, for which the state trajectory $\tilde{x}$ satisfies: 1) $x(0) = x_I$, and 2) there exists some $t > 0$ for which $u(t) = u_T$ and $x(t) \in X_G$.

This formulation can be combined with an optimization objective of the form

$$L(\tilde{x}_{t_F}, \tilde{u}_{t_F}) = \int_0^{t_F} l(x(t), u(t))dt + l_F(x(t_F)), \tag{2.1}$$

where $\tilde{x}_{t_F}$ is the state trajectory, $\tilde{u}_{t_F}$ is the action trajectory, t_F is the time at which the termination action is applied, $l(x, u)$ is the cost of a specific state x and input u and $l_F(x(t_F))$ is the terminal cost.

Formulation 2.1 can be interpreted as a *boundary value problem* (BVP) where the start state and goal region define the boundary conditions. The objective is to find a solution trajectory within the obstacle-free configuration space $\mathcal{C}_{free}$, satisfying the differential constrains.

2.3 Sample-based motion planning

One major challenge when solving a motion planning problem is representing $\mathcal{C}_{obs}$. Sample-based planning is a motion planning philosophy where one avoids explicitly modelling $\mathcal{C}_{obs}$ by instead probing $\mathcal{C}$ based on a sampling scheme and checking if the sampled configuration and robot geometry collides with the obstacle region, $\mathcal{O}$. This actually avoids requiring a discretization of the configuration space all together. The collision check can be done by a separate module and can be implemented as a black box.

A general framework for a sample-based planning algorithm is formulated by LaValle (2006) in the following way:

Formulation 2.2: Sample-based planning algorithm

1. **Initialization**: Let $\mathcal{G}(V, E)$ represent an undirected *search graph*, for which V contains at least one vertex and E contains no edges. Typically, V contains q_I, q_G or both.
2. **Vertex selection method**: Choose a vertex $q_{curr} \in V$ for expansion.
3. **Local Planning Method**: For some $q_{new} \in \mathcal{C}_{free}$ that may or may not be represented by a vertex in V, attempt to construct a path $\tau_s : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_{curr}$ and $\tau(1) = q_{new}$. Check to ensure τ_s does not cause a collision. If it fails to produce a collision-free path segment, then return to step 2.
4. **Insert an Edge in the Graph**: Insert τ_s into E, as an edge from q_{curr} to q_{new}. If q_{new} is not already in V, then insert it.
5. **Check for a Solution**: Determine whether $\mathcal{G}$ encodes a solution path.
6. **Return to step 2**: Iterate unless a solution has been found of some termination condition is satisfied, in which the algorithm reports failure.

A contrasting motion planning philosophy worth mentioning is *combinatorial motion planning*. These motion planning algorithms revolve around explicitly representing the planners input, such as the world, $\mathcal{W}$, and the obstacle region, $\mathcal{O}$, usually in a roadmap or grid-like structure. These algorithms are generally harder to implement and is very dependent on the application, but when done correctly they can be effective and offer *completeness*. A sampling-based algorithm, on the other hand, can only give *probabilistic completeness*. This means the probability that a sample-based algorithm will produce an answer approaches 1 as more time is spend, but they can never determine if no path actually exists. Whereas a combinatorial algorithm will yield a definite answer if no path exists.

2.4 Local planning methods

Local planning methods (LMP) refer to the planning method used to connect configuration-vertices when building a search tree, as used in step 3 in formulation 2.2. LaValle (2006) mentions two different local planning methods: geometric and control-based planning.

2.4.1 Geometric planning

Geometric planning accounts for the geometric and kinematic constraints of the robot, $\mathcal{A}$, and the obstacle region, $\mathcal{O}$. This can be used to solve the piano mover's problem described in the beginning of this chapter. Edges in the search tree are typically modeled as straight lines, but can be represented by any motion primitive, as we will see later.

2.4.2 Control-based planning

Control-based planning solves the problem given in problem formulation 2.1. This method creates edges by solving a BVP using a nearest neighbour node in the search tree and the sampled node as boundary conditions. Efficient BVP solvers are however not designed for handling global constraints such as obstacle regions. So when using a control-based local planning method a BVP is solved with no regard of $\mathcal{O}$. The solution path is then validated by a collision check module before adding it to $\mathcal{G}$.

2.4.3 Closed form optimal paths

The suggested approach for solving the path planning problem presented up until now is to use a sample-based planning method to expand a search graph where the vertices are viable robot configurations within the world. To model the obstacle region, $\mathcal{O}$, a collision detection module is used. A local planning method is used to connect vertices. Once a path is found connecting two configurations, configurations along this path is also checked for collision before adding it as an edge connecting the vertices.

The local planning methods mentioned in the previous section are geometric- and control-based. Control-based LPMs are capable of finding paths satisfying the differential constraints of a dynamic system. This is desirable when planning paths for fixed wing aerial vehicles as the system dynamics can't be ignored trivially, i.e. in the case of sharp turn angles or straight vertical climbs.

A control-based LPM can find a solution path by representing the path planning problem as a BVM and solving it. This is however very computationally expensive compared to its geometric counterpart, especially when combined with 2.1 to find an optimal trajectory. A better solution would be to geometrically model an optimal path connecting two robot configurations. The following section will present methods for doing so.

Dubins curves

LaValle (2006) presents several methods of finding shortest paths between configuration pairs for wheeled vehicles. One of these methods is called Dubins curves or the Dubins car, where optimal paths for a simple car model can be found. Chitsaz and LaValle (2007) expands this model further to include all dimensions in 3D, and thus gives the possibility to find optimal paths for a simple airplane model.

Symbol	Steering: u
S	0
L	1
R	-1

Table 2.1: The three motion primitives from which all optimal paths for the Dubins car can be constructed.

Dubins car

Given a model of a simple car with constant unit velocity, defined by

$$\begin{aligned}\dot{x} &= \cos\theta \\ \dot{y} &= \sin\theta \\ \dot{\theta} &= u,\end{aligned} \tag{2.2}$$

where the u is the cars turning rate chosen from the interval $[-\tan\phi_{max}, \tan\phi_{max}]$ and ϕ_{max} is the maximum turn angle of the car, it has been shown by Dubins (1957) that the shortest path between any two configurations of this system can be expressed as a combination of no more that three motion primitives. Each of these primitives can be applied by a constant action over a time interval. The only actions that are needed to traverse the shortest path are $u \in \{-1, 0, 1\}$. Each primitive is associated with a symbol: S, L and R. The primitives and their associated symbols can be seen in table 2.1. The S primitive drives the car straight ahead. The L and R primitives turn the car as sharply as possible to left and right, respectively.

Possible shortest paths can then be characterized as a sequence of these symbols that corresponds to the order in which the primitives are applied. Such a sequence is called a *word*. Dubins showed that only six words are possibly optimal:

$$\{LRL,\ RLR,\ LSL,\ RSR,\ LSR,\ RSL\} \tag{2.3}$$

To be more precise the duration of each of the primitives can also be specified. For L and R a subscript can denote the accumulated rotation angle during the application of the primitive. For S a subscript can denote the total distance traveled. Using these subscripts, the Dubins curves can be more precisely characterized as:

$$\{L_\alpha R_\beta L_\gamma,\ R_\alpha L_\beta R_\gamma,\ L_\alpha S_d L_\gamma,\ R_\alpha S_d R_\gamma,\ L_\alpha S_d R_\gamma,\ R_\alpha S_d L_\gamma\}. \tag{2.4}$$

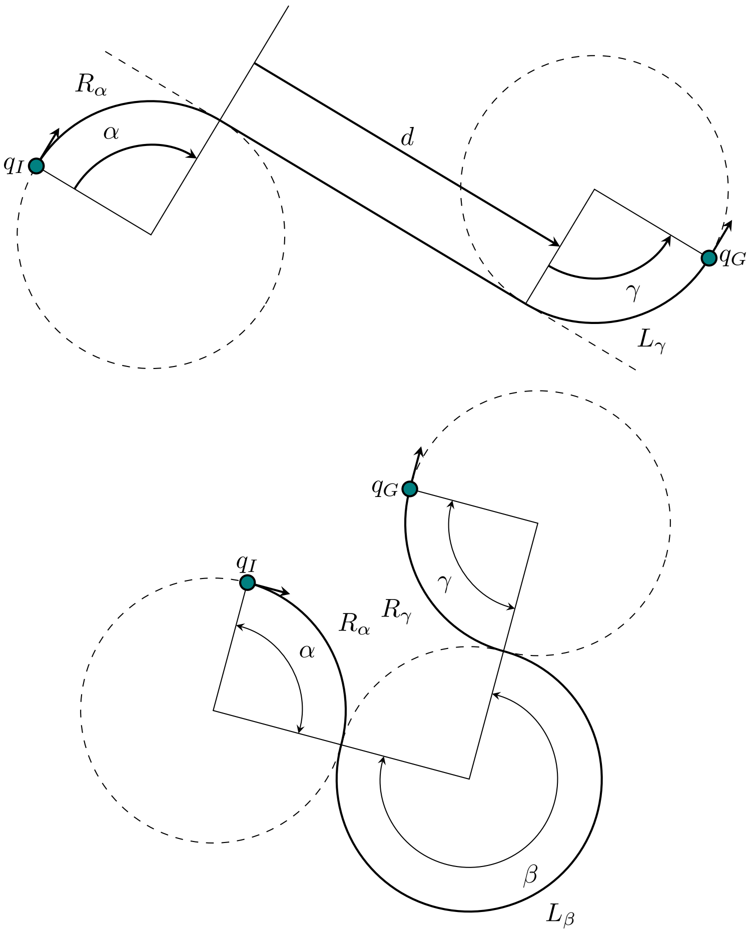
An illustration of two such paths can be seen in figure 2.2.

To use Dubins curves as an LPM two questions must be answered effieciently:

1. Which of the six words in 2.4 gives the shortest path between q_I and q_G?
2. What are the values of the subscripts α, γ, β and d?

A naive approach would be to find the Dubin curves for each word and select the shortest one. However, Shkel and Lumelsky (2001) propose a more method to classify the set of Dubins paths for two configurations based on the distance between them and their heading angle. Methods for finding the subscripts are also given by Shkel and Lumelsky (2001).

Figure 2.2 Example showing two Dubin car paths and the word of each trajectory. The words of each curve are RSL and RLR, shown in $\mathcal{W} = \mathbb{R}^2$



Dubins airplane

The Dubins airplane extends the Dubins car model into four dimensions by including a Z-axis. The system in 2.2 can then be expanded to:

$$\begin{aligned}\dot{x} &= \cos\theta \\ \dot{y} &= \sin\theta \\ \dot{z} &= u_z \\ \dot{\theta} &= u_\theta,\end{aligned} \tag{2.5}$$

where x, y and z is the airplanes position in euclidean space and θ is the orientation angle in the xy-plane with regards to the x-axis. The rate of which the plane can climb or sink is given by $u_z \in [-u_{z,max}, u_{z,max}]$. The yaw turning rate is given by $u_\theta \in [-u_{\theta,max}, u_{\theta,max}]$.

In many cases a closed form solution exists for finding an optimal path between configurations of the Dubins airplane, as shown by Chitsaz and LaValle (2007). When this is the case the problem can be reduced to finding the Dubins car path between projections of the configurations in the xy-plane. To explain this further the problem should be split into three cases. Two of which always will yield an optimal path through finding the projected Dubins car path.

Let q_I denote the initial configuration for the Dubins airplane system. Similarly let q_G denote the goal configuration. For simplicity the following cases will be explained in a scenario where the airplane needs to ascend to reach the goal configuration, but all cases can be inverted to where the airplane must descend. The projection of these configuration into the xy-plane, thus becoming configurations for the Dubins car, will be denoted with a subscript: $q_{I,car}$ and $q_{G,car}$. The path connecting the will be denoted as p_{car}.

Low altitude Let the airplane fly directly over the projected Dubins car path connecting $q_{I,car}$ and $q_{G,car}$ with a maximum climb rate $u_z = u_{z,max}$. If the altitude of the airplane's final configuration is higher than the altitude of q_G, or at the same height as q_G, the path is in the case of *low altitude*. An example of such a path can be seen in Figure 2.3.

The optimal Dubins airplane path can then be found by flying directly over the projected Dubins car path, p_{car}, with an adequate climb rate

$$u_z = \frac{z_G - z_I}{p_{car}.length}.$$

High altitude Let the airplane fly directly over the projected Dubins car path connecting $q_{I,car}$ and $q_{G,car}$ with a maximum climb rate $u_z = u_{z,max}$. Afterwards the airplane flies in a helix of 360° with a minimal turning radius r_{min} while maintaining a maximum climb rate. If the altitude of the airplanes final configuration is beneath, or at the same height, as the altitude of q_G the path is in the case of *high altitude*. An example of such a path can be seen in Figure 2.3.

The optimal Dubins airplane path can then be found by flying directly over the projected Dubins car path, p_{car}, with an maximum climb rate $u_z = u_{z,max}$. Followed by a helix of $n * 360°$ with adequate turning radius

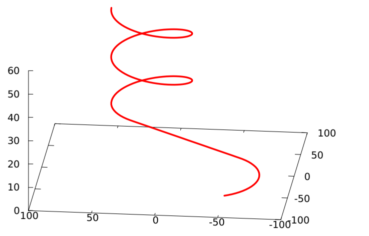
$$r = \frac{|z_G - z_I| - p_{car}.length * u_{z,max}}{2 * \pi * n * u_{z,max}} \geq r_{min}.$$

Where r_{min} is the minimum turn radius of the airplane and n is the maximum number of helices at where the airplanes altitude is still beneath or equal to the altitude of q_G.

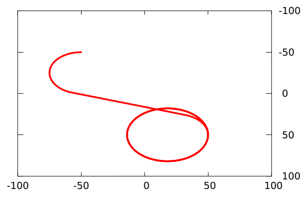
Medium altitude Let the airplane fly directly over the projected Dubins car path connecting $q_{I,car}$ and $q_{G,car}$ with a maximum climb rate $u_z = u_{z,max}$. Afterwards the airplane flies in a helix of 360° with a minimal turning radius while maintaining a maximum climb rate. If the altitude of the airplanes final configuration now is above as the altitude of q_G the path is in the case of *Medium altitude*.

This case can further be split into two categories:

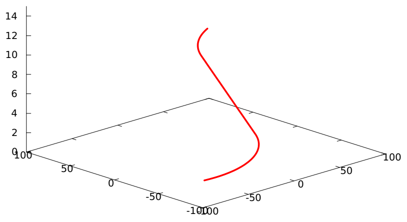
Figure 2.3 Examples of Dubins airplane paths showing a high and low type path.



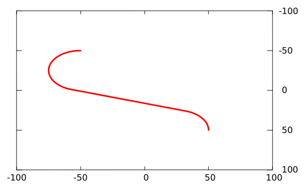
(a) Example of a high altitude Dubins airplane path.



(b) Projection in the xy-plane of the high altitude path shown in 2.3a.



(c) Example of a low altitude Dubins airplane path.



(d) Projection in the xy-plane of the low altitude path shown in 2.3c

1. If there exists a projected Dubins car path p_{car} from which the airplane can fly above with a maximum climb rate and reach q_G this trajectory will be optimal. p_{car} does not need to be an optimal path for the Dubins car, i.e. it can consist of curves with a turn radius greater than r_{min}.

2. If there does not exist such a path for the Dubins car the optimal trajectory for the airplane will consist of straight lines and turns and helices with minimal turning radii. The climb rate will not be saturated, $|u_z| \neq u_{z,max}$.

2.5 Planning algorithms

In contrast to the local planning methods mentioned in the previous section the term *planning algorithm* is used when addressing the algorithm building the search graph for a sample-based planner. This section will briefly mention the planning algorithms relevant for this thesis. The original publication for each algorithm will be listed, so for more information please refer to these.

All the algorithms presented are used for geometric planning with a cost optimization goal. These algorithms can be split into two classes: expanding tree algorithms and road map algorithms. Expanding tree algorithms work by growing a tree of states connected by valid motions. These planning algorithms differ in the heuristic they use to control where and how the tree is expanded. Some algorithms grow two trees: one from the start and one from the goal. Road map algorithms initially sample states and build a road map of the entire environment that is used for queries to find a path connecting a new configuration.

Informed RRT*

The Informed RRT* algorithm is an optimized extension of the RRT* algorithm. Here RRT stands for *rapid-expanding random tree*. The original RRT* algorithm provides asymptotically optimal paths through expanding a search tree and always finding the path with minimal cost whenever a new vertex is added to the tree. In doing so it also finds an asymptotically optimal path from the initial state to every vertex in the planning domain, this behavior is inefficient.

Informed RRT* adds to the original implementation by once a path is found connecting the initial state with the goal state it only samples states in an elliptical distribution around the this path. If a better path is found this ellipse is shrunk to narrow the sampling domain.

For more informati on both the RRT* and the Informed RRT* planning algorithms read the original paperes by Karaman and Frazzoli (2011) and Gammell et al. (2014), respectively.

FMT*

FMT* is an asymptotically-optimal sampling-based motion planning algorithm, which is guaranteed to converge to a shortest path solution. The algorithm is specifically aimed at solving complex motion planning problems in high-dimensional configuration spaces where obstacles are frequent. The FMT* algorithm essentially performs a lazy dynamic programming recursion on a set of probabilistically-drawn samples to grow a tree of paths, which moves steadily outward in cost-to-come space.

For more information please refer to the original publication by by Janson et al. (2015).

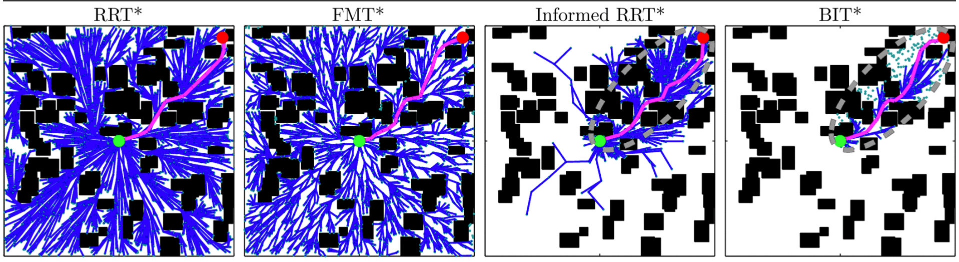
BIT*

Batch Informed Trees is a planning algorithm based on unifying graph- and sample-based planning techniques. The following is a informal explanation given in the original paper by Gammell et al. (2015).

BIT* works as follows: An initial random geometric graph (RGG) with implicit edges is defined by uniformly distributed random samples from the free space and the start and goal. An explicit tree is then built outwards from the start towards the goal by a heuristic search. This tree includes only collision-free edges and its construction stops when a solution is found or it can no longer be expanded. This concludes a batch.

To start a new batch, a denser implicit RGG is constructed by adding more samples. If a solution has been found, these samples are limited to the subproblem that could contain a better solution (e.g., an ellipse for path length). The tree is then updated using search

Figure 2.4 An illustration taken from the publication by Gammell et al. (2015) illustrating the behavior of the RRT*, FMT*, Informed RRT* and BIT* algorithms.
© 2015, IEEE



techniques that reuse existing information. As before, the construction of the tree stops when the solution cannot be improved or when there are no more collision-free edges to traverse. The process continues with new batches as time allows.

Figure 2.4 shows a comparison of how the RRT*, FMT* Informed RRT* and BIT* algorithms build their search graph and sample configurations.

BFMT*

Bi-directional FMT* is an expansion of the original FMT* algorithm including bi-directional search, expanding one tree from the initial configuration and one tree from the goal. More info can be found in the original paper by Starek et al. (2015).

PRM*

PRM* is an asymptotically optimal version of the Probabilistic Roadmap (PRM) algorithm. Kavraki et al. (1996) describes PRM in general terms as an algorithm split into two phases: a learning phase and query phase. In the learning phase the algorithm constructs a roadmap of the world by randomly sampling configurations and connecting them with a local planning method. In the querying phase the algorithm tries to connect the initial and goal configurations to this roadmap and searches for a connecting path.

The RPM* algorithm expands the original version to grow the roadmap in a way that converges to the optimal solution. More information on this algorithm can be found in the paper by Karaman and Frazzoli (2011).

2.6 Software frameworks

This section will serve as a brief introduction to the software frameworks used in the project. Each framework has a comprehensive online documentation and an associated paper. Please refer to these for more information.

Open Motion Planning Library

The Open Motion Planning Library (OMPL) is the main framework used in the project behind this thesis. It provides an abstract representation for all core concepts in motion planning. Such as:

- The configuration space, implemented through classes called state spaces.
- Collision detection is implemented through state validity checking callback.
- State-of-the-art sampling-based planning algorithms capable of both geometric and control-based planning. It embeds all the planning algorithms mentioned in Section 2.5 and many more.

This section will not serve the purpose of being an OMPL tutorial, for this please refer to their website (Şucan et al., 2012), but some core concepts should be mentioned. The OMPL framework includes a helper class, called `SimpleSetup`, for both geometric and control-based planning. Through this class one can access all necessary parts of the framework with minimal setup.

For geometric planning it is required to set what state space to use for planning. The library already includes state spaces embedding $\mathbb{R}^2$, $\mathbb{R}^3$, $\mathrm{SE}(2)$, $\mathrm{SE}(3)$ and much more. It even includes a state space embedding the Dubins car. For control-based planning an action space must also be set, along with a method describing the ODE of a system.

Once the state space is set a start and goal state must be given. It is also necessary to implement a method for state validation checking. This is implemented as a callback function used by the planning algorithm to check for state validity and collision. OMPL itself does not include any code related to collision checking. This is a deliberate choice by the developers so that OMPL is not tied to any particular framework dependency.

Flexible Collision Library

The Flexible Collision Library (FCL) is a collision and proximity detection library that integrate several techniques for fast and accurate collision and proximity computation. To read more about this library see the original publication by Pan et al. (2012) or see their GitHub.

The framework gives a high level abstraction layer to create collision objects consisting of geometric shapes, such as boxes, cylinders and spheres. The user can then apply transformations to these shapes to position them in three dimensional space. The framework also includes a class to group collision objects called collision managers. Collision checking is done through ”colliding” the collision objects or mangers and querying for collision points.

OctoMap

OctoMap is an integrated framework for representation of three-dimensional environments. For more information refer to the paper by Hornung et al. (2013) or their website.

The framework uses octrees to efficiently store information regarding structure and occupancy of 3D space. An octree is a hierarchical data structure for subdivision in 3D.

Each node is an octree represents the space contained in a cubic volume, called a voxel. This volume is recursively subdivided into eighth smaller voxels until a given minimum voxel size is reached. The minimum voxel size is determined by the resolution of the octree.

Octrees are widely used to generate 3D environments from sensor data such as LiDAR readings or point clouds. It stores a probabilistic occupancy value in each voxel that can be updated if new sensor data is available. This makes it possible to represent dynamic environments and expand the environment if the sensor is mobile.

Chapter 3

System Implementation

This chapter will present how the path planner, capable of planning efficient and collision free paths through real world terrain, is implemented. Chapter 1 introduced the following requirements for the system:

- The planner should take into account the system dynamics of the UAV and find a feasible path considering these dynamics.
- Should find an efficient path.
- Should plan a trajectory through real wold terrain data.
- Should take into consideration eventual no-fly zones.
- Should maintain a safe volume around the UAV keeping a minimal distance to any terrain or no-fly zones.
- Should include several checkpoints in the trajectory. The checkpoints should be given as an ordered list of poses (coordinate and heading).

Here efficiency is deemed with regard to path distance, aiming for minimal distance.

To simplify the problem the planner should not have to consider wind conditions and can use a uniform movement cost throughout the world. The planner should also only be concerned with finding a feasible trajectory and not calculate any control plan for navigating said trajectory.

To build such a system two software implementations are made. Firstly, a contribution is made to the Open Motion Planning Library (OMPL) implementing the Dubins airplane model local planning method. This makes it possible to use the sample-based planning framework included in OMPL to plan paths in three dimensional space adhering the dynamics of a fixed wing UAV. Secondly, a path planner system is implemented using this extended version of the OMPL framework. The planner is implemented as a Robot Operating System (ROS) node. All software implemented regarding OMPL and the planning system is written in C++.

The following chapter will explain how the contribution to the OMPL-framework is implemented, and also elaborate on the chosen method to represent terrain, do collision detection and avoid any eventual no-fly zones. Lastly, the chapter will explain how all this is joined together to create the implemented planning system. The choice of planning algorithm will be covered through experiments in Chapter 4.

3.1 Implementing the UAV dynamics

For the planner to generate paths feasible for a fixed-wing UAV to maneuver, the contribution to the OMPL-framework embeds the system given in Equation 2.5. This system represents the dynamics of a real world airplane well enough to find suitable trajectories given a reasonable choice of bounding climb and turn rates. Another advantage to this system is that there exists an almost optimal closed-form solution to the boundary value problem connecting configuration pairs. This closed-form solution to the BVP is implemented as a local planning method through two new state space classes: `SimpleSE3StateSpace` and `DubinsAirplaneStateSpace`.

`SimpleSE3StateSpace` embeds configurations consisting of coordinates along the x-, y- and z-axes, combined with rotation about the z-axis relative to the x-axis, denoted as $\theta \in [-\pi, \pi)$. This state space is implemented similarly to the class `SE2State-Space` already included in the base version of OMPL.

`DubinsAirplaneStateSpace` is implemented similarly to OMPL's own `Du-binStateSpace` which embeds the Dubins car system, but where the two-dimensional car system inherits its configuration space from `SE2StateSpace`, the implemented airplane state space inherits from `SimpleSE3StateSpace`, expanding it to three dimensions. Some of the member functions of `DubinStateSpace` are also extended to account for the added dimension. Naturally this includes the methods used to generate the closed from solution path for the Dubins airplane path (how this is implemented can be seen in Algorithm 1), but also methods to calculate path length and methods for interpolating along the path. The airplane state space is initialized with a bounding maximal climb/sink angle and a minimum turn radius. These values are related to the bounding climb/sink and turn rate in Equation 2.5 through:

$$u_{z,max} = \tan(\gamma_{max}) \tag{3.1}$$

$$u_{\theta,max} = \frac{1}{r_{min}}, \tag{3.2}$$

where γ_{max} is the bounding climb/sink angle and r_{min} is the minimum turn radius. The paths illustrated in Figure 2.3 are found using this implementation of Dubins airplane state space.

As mentioned in Section 2.4.3 an optimal path for the Dubins airplane can be found through a closed-form expression in the cases of a low and high altitude path. The implemented methods for finding paths in these cases can be seen in Algorithm 2 and 3, respectively. In the case of a medium altitude path a choice is made to use a bounded sub-optimal solution. The solution is to create a path the way its described in Section 2.4.3 (flying over the Dubins car path connecting the projected configuration pairs and flying in a helix at the end), but instead of using a saturated climb rate an adequate climb rate is chosen. This implementation can be seen in Algorithm 4 and is inspired by Schneider (2016) whom solved the case of medium altitude paths in a similar manner. Schneider (2016) showed through experiments that the sub-optimal path is bounded by:

$$path_{sub}.length - path_{opt}.length < \frac{2\pi r_{min}}{\cos(\gamma_{max})},$$

and claims this to be a more computationally efficient method than the one presented by McLain et al. (2014), among other advantages.

The method used to find the Dubins car path, in Algorithm 1, is identical to the method used in the original `DubinStateSpace`, which is based on Shkel and Lumelsky (2001).

3.2 Collision detection module

With a local planning method implemented as described is the previous section the next step in developing a sample-based motion planning system is to create a collision detection module. As explained in Section 2.3 the purpose of such a module is to disqualify any sampled configuration from being added to the search graph built by the planning algorithm if the geometry of a robot in this configuration intersects with the obstacle region. Considering the requirements listed in the beginning of this chapter an obvious choice was to model the robot geometry as the safe volume around the UAV and the obstacle region as a union of the terrain and any no-fly zones.

The terrain is modeled as an octree datatype from the OctoMap framework. Collision detection is done using the Flexible Collision Library (FCL). This library supports OctoMap's data types out of the box. The no-fly zones were modeled in FCL as vertical cylinders, with the center of the bottom circle at a given coordinate in the xy-plane, with a given cylinder radius and height. The obstacle region was then represented through registering all no-fly zone cylinders and the OctoMap terrain with one of FCL's collision managers. Figure 3.1 visualizes an OctoMap environment with a no-fly zone cylinder.

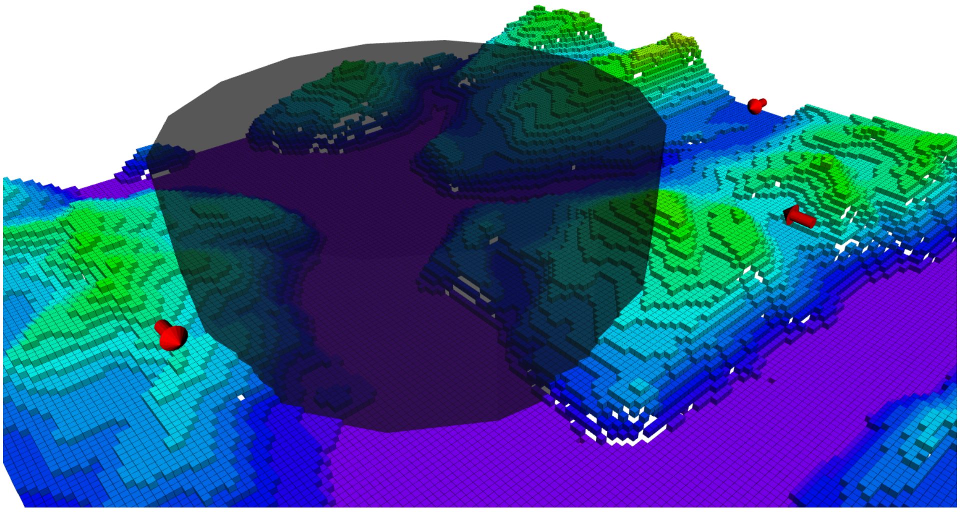
The robot geometry is implemented in FCL as a sphere with its center given by the x-, y- and z-coordinates of a configuration. The collision manager can then "collide" the sphere with its registered collision objects querying any collision points. In the existence of any collision points the configuration is discarded and not added to the search graph.

3.3 Planner implementation

The planner system is implemented following the suggestions from OMPL online documentation and examples. It uses the `ompl::geometric::SimpleSetup` class with the custom `DubinsAirplaneStateSpace` to allow for geometric planning while still adhering to the dynamics of a fixed-wing UAV. The bounds of the state space are initialized to the maximal and minimal values of the terrain Octree.

The collision detection module is implemented as described in the previous section and set to be the `SimpleSetup`'s state validity checker. The setup of the collision manager

Figure 3.1 A illustration of an OctoMap environment combines with a no fly zone cylinder. The arrows shown are markers for planning checkpoints.



is done initially. The OctoMap is published on a ROS topic from a `octomap_server_node` and the no-fly zones are stored as a list in the ROS parameter server.

A code snippet is included in Listing A.1 in Appendix A.2 showing the implementation of both the planner system and the collision detection module.

Other parameter are also stored in the ROS parameter server, such as the ordered list of checkpoints, the bounding turn radius and climb angle and the radius of the safe volume sphere surrounding the UAV.

The system handles the requirement of several checkpoints in a path by managing each pair of checkpoints as a separate planning task. It loops through all checkpoint pairs using the same collision manager, but initializes a new state space for every iteration.

The planner is called from the command line specifying what planning algorithm to use along with different search parameters. This can be run time to specify the time used to plan between two checkpoints and/or a sample batch size to be used by the planning algorithm. Once a path is found connecting all the specified checkpoints the system interpolates along the paths and writes the coordinates to a file. This allows for some flexibility when choosing the size of the interpolation step, giving control over how many points should be in a path and how often the UAVs autopilot need to update its target.

Algorithm 1 Algorithm for determining the type of a path connecting a configuration pair in the Dubins airplane state space. The function DUBINCARPATH returns the optimal path in the xy-plane connection the projected configurations.

```
function DUBINSAIRPLANEPATH($q_I$, $q_G$)
    $dz \leftarrow q_G.z - q_I.z$
    $carPath \leftarrow$ DUBINCARPATH($q_I$, $q_G$)
    $climbAngle \leftarrow \text{arctan2}(dz,\ carPath.length)$
    if $|climbAngle| \leq climbAngleLimit$ then
        return COMPUTELOWPATH($carPath$, $dz$)
    else
        $n \leftarrow \left\lfloor \frac{|dz|/\tan(climbAngleLimit) - carPath.length}{2*\pi*turnRadiusLimit} \right\rfloor$
        if $n > 0$ then
            return COMPUTEHIGHPATH($carPath$, $dz$)
        else
            return COMPUTEMEDIUMPATH($carPath$, $dz$)
        end if
    end if
end function
```

Algorithm 2 Algorithm showing how the low altitude paths are computed. The algorithm DUBINSAIRPLANEPATH returns an instance of a path consisting of a Dubins car path and a climb angle.

```
function COMPUTELOWPATH($carPath$, $dz$)
    $climbAngle \leftarrow \text{arctan2}(dz,\ carPath.length)$
    $airplanePath \leftarrow$ DUBINSAIRPLANEPATH($carPath$, $climbAngle$)
    return $airplanePath$
end function
```

Algorithm 3 Algorithm showing how the high altitude paths are computed. The algorithm DUBINSAIRPLANEPATH returns an instance of a path consisting of a Dubins car path and a climb angle. The COPYSIGN function returns the value the first argument with the sign of the second. The HELIX function returns a helix with given direction, radius r, climb angle and n turns.

```
function COMPUTEHIGHPATH($carPath,\ dz$)
    $climbAngle \leftarrow$ COPYSIGN($climbAngleLimit, dz$)
    $airplanePath \leftarrow$ DUBINSAIRPLANEPATH($carPath,\ climbAngle$)
    $n \leftarrow \left\lfloor \frac{|dz|/\tan(climbAngleLimit) - carPath.length}{2 * \pi * turnRadiusLimit} \right\rfloor$
    $r \leftarrow \frac{|dz| - carPath.length * turnRadiusLimit * \tan(climbAngleLimit)}{2 * \pi * n * \tan(climbAngleLimit)}$
    $direction \leftarrow carPath.lastSegmentDirection$
    $airplanePath.helix \leftarrow$ HELIX($direction,\ r,\ climbAngle,\ n$)
    return $airplanePath$
end function
```

Algorithm 4 This algorithm shows how the medium altitude paths are computed. The algorithm DUBINSAIRPLANEPATH returns an instance of a path consisting of a Dubins car path and a climb angle. The HELIX function returns a helix with given direction, radius r, climb angle and n turns.

```
function COMPUTEMEDIUMPATH($carPath,\ dz$)
    $climbAngle \leftarrow \arctan2(dz,\ carPath.length + 2 * \pi * turnRadiusLimit)$
    $airplanePath \leftarrow$ DUBINSAIRPLANEPATH($carPath,\ climbAngle$)
    $r \leftarrow turnRadiusLimit$
    $direction \leftarrow carPath.lastSegmentDirection$
    $airplanePath.helix \leftarrow$ HELIX($direction,\ r,\ climbAngle,\ 1$)
    return $airplanePath$
end function
```

Chapter 4

Simulations and Results

This chapter will present what simulations where run to evaluate the implemented system's performance using several of OMPL's out-of-the-box planning algorithms. The results from these simulations will be presented and discussed in order to determine which algorithm is best suited to be used when planning for fixed-wing UAV trajectories. While searching for a suitable algorithm some pitfalls were also discovered regarding some of the planning algorithm in conjunction with the implemented local planning method.

Path planning is a demanding computational task, especially when the world to plan in is large. It became apparent while testing the implemented planner that the system memory of the computer running the simulations limited how computationally intensive each simulations could be. For this reason the simulations were separated into two categories: a set of less resource demanding simulations to give a statistical analysis of the systems performance and a set of more thorough simulations to give a qualitative analysis of the more optimal paths. The statistical analysis will be composed of several simulations repeated multiple times to give a distribution of resulting path lengths. The qualitative analysis will focus on single more resource demanding simulations, yielding paths closer to the optimal solution.

All five planning algorithms introduced in Section 2.5 were originally tested, but only some of them were deemed suitable to be used in the system. The main reason for this being how local paths found using `DubinAirplaneStateSpace` are asymmetric, meaning a path from q_I to q_G will not always be equal to a path from q_G to q_I. This will be discussed further in Section 4.2. The algorithms at issue is RPM* and BFMT*.

As stated in the requirements listed in the beginning of chapter 3 the system should find an efficient path, where efficiency is comparable to a minimal length path. Therefore the main criteria for evaluating the system will be path length.

The algorithms were tested with different relevant parameters:

- IRRT* and BIT* are evaluated as a function of run time.
- FMT* is evaluated as a function of number of samples initially chosen.

Climb angle, γ_{max}	$0.1\,\mathrm{rad}$
Turn radius, r_{min}	$300\,\mathrm{m}$
Saftey radius	$150\,\mathrm{m}$

Table 4.1: The values for the initialization parameters for the `DubinAirplaneStateSpace` and the saftey radius used in the simulations.

The simulations also test the systems performance in different environments. The purpose of this is to capture the effect of introducing no-fly zones to the system, as well as the impact of the terrain size on the system. The first two sets of simulations were all performed in the same terrain environment, but one was without no-fly zones and one was with. Lastly simulations were run in a larger terrain environment without any no-fly zones. This last environment did however include a maximal height restriction to make the simulations a bit more demanding.

Common for all simulations is the initialization parameters of the `DubinAirplane-StateSpace`, the ordered list of checkpoints to be visited by the planner and the radius of the sphere representing the safety volume around the UAV. The values used for the state space' parameters and the chosen safety radius can be seen in Table 4.1. The checkpoints will be introduced in the following section.

4.1 Simulation environments

The simulation environments used in the test are based on OctoMaps generated from terrain taken from the municipality of Sykkylven, located along the west coast of Norway. The OctoMaps were generated in relation to a previous project by Eger (2019). The environment consists of diverse terrain, with both fjords and mountains with altitudes of over $1700\,\mathrm{m}$. For the OctoMap used in the two first sets of simulations the area covered by the model is roughly $50\,000\,\mathrm{m} \times 17\,000\,\mathrm{m}$. An elevation map of the environment can be seen in Figure 4.1. For the OctoMap used in the large scale experiment the model covered roughly $50\,000\,\mathrm{m} \times 100\,000\,\mathrm{m}$. Both environments had a voxel resolution of $50\,\mathrm{m}$.

As stated in the project thesis by Eger (2019) the OctoMap was generated by collecting elevation data from `hoydedata.no` and converted into a point cloud with a given density. The octree is then created by ray-casting from a ceiling, set to the highest z-value in the point cloud plus $550\,\mathrm{m}$, down to each point in the cloud.

For the first two simulation scenarios there are six checkpoints included, their location can be seen in Table 4.2. The distance between each pair of checkpoints can be seen in Table 4.3. For the simulations including no-fly zones, four no-fly zones were added. Their coordinates and radii can be seen in Table 4.4. All but one of the no-fly zones' altitudes reach the ceiling altitude of the OctoMap, meaning the planner could not fly over them. Z_2 is possible to fly over. An overview of the environment can be seen in Figure 4.2. Here the no-fly zones are visualized as dark cylinsers and the checkpoints are visualized with red arrows.

Checkpoint	$x,\ y,\ z,\ \theta\ \ (\mathrm{m}, \mathrm{rad})$
C_1	$42500.0,\ 600.0,\ 700.0,\ 1.6$
C_2	$42500.0,\ 14500.0,\ 1200.0,\ -3.1$
C_3	$39000.0,\ 5000.0,\ 1100.0,\ 0.0$
C_4	$20000.0,\ 2600.0,\ 750.0,\ -3.1$
C_5	$17000.0,\ 14000.0,\ 750.0,\ -3.1$
C_6	$1300.0,\ 11000.0,\ 600.0,\ 0.0$

Table 4.2: Coordinate and heading orientation for each of the checkpoints used in the simulations.

Checkpoint pair	Length
$p_{1,2}$	$13\,908.99\,\mathrm{m}$
$p_{2,3}$	$10\,124.72\,\mathrm{m}$
$p_{3,4}$	$19\,154.18\,\mathrm{m}$
$p_{4,5}$	$11\,788.13\,\mathrm{m}$
$p_{5,6}$	$15\,984.76\,\mathrm{m}$
Total	$70\,960.78\,\mathrm{m}$

Table 4.3: Euclidean distance between each of the checkpoint pairs and the total distance linking together all checkpoints. Terrain is ignored in this distance measure.

No-fly zone	$x,\ y,\ h,\ r\ (\mathrm{m})$
Z_1	$28000.0,\ 7500.0,\ 5000.0,\ 4000.0$
Z_2	$24000.0,\ 2500.0,\ 2000.0,\ 3000.0$
Z_3	$8000.0,\ 13000.0,\ 3000.0,\ 4000.0$
Z_4	$43000.0,\ 10000.0,\ 3000.0,\ 4000.0$

Table 4.4: Radius, x- and y coordinate of all no-fly zones.

Figure 4.1 An elevation map of the Sykkylven environment. Height is given in meters. The origin is located in the bottom right hand corner. The figure is taken from previous work by Eger (2019).

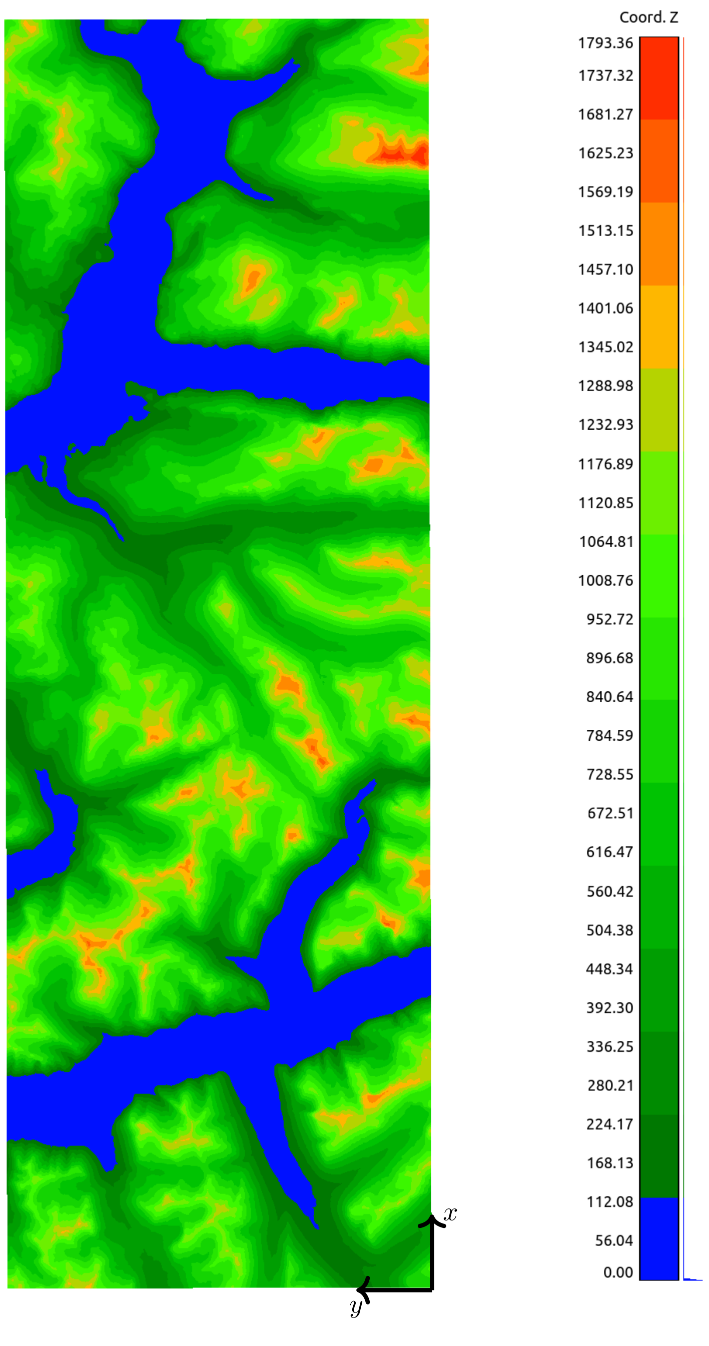
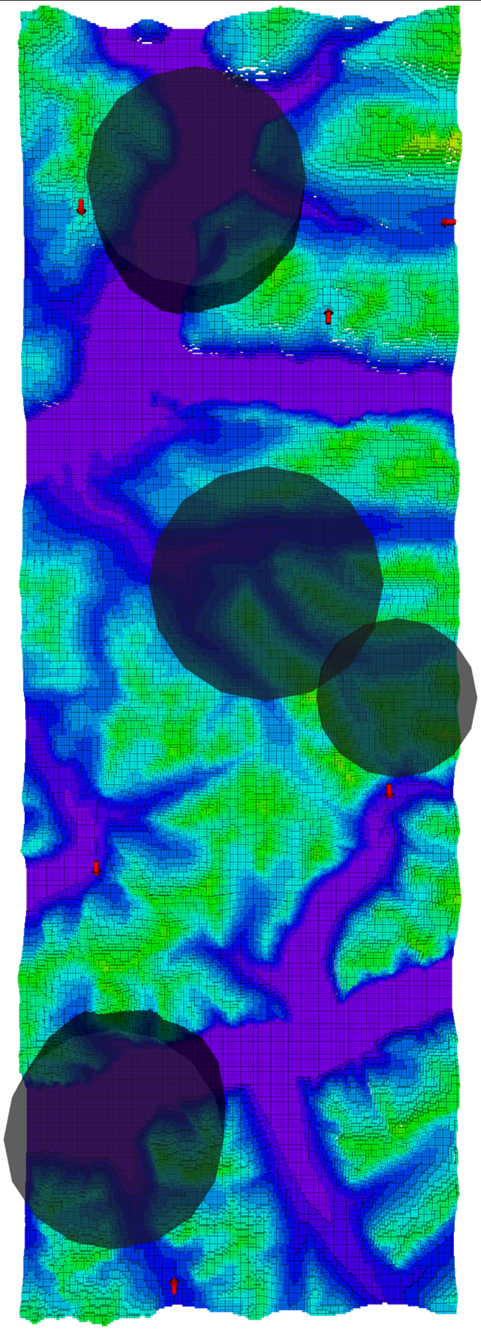


Figure 4.2 An overview of the Sykkylven environment visualizing both checkpoints as red arrows and no-fly zones as dark cylinders. The checkpoints are located as the tip of the arrows and have heading in the same direction as the arrow. The origin is located in the bottom right hand corner.



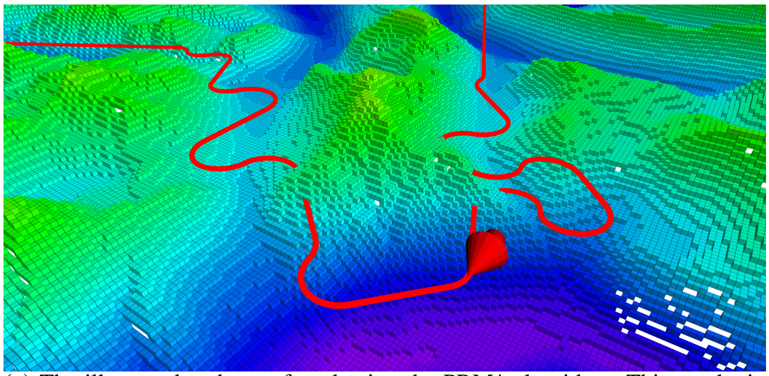
4.2 Issues using bidirectional and roadmap planners

As mentioned in the beginning of this chapter some pitfalls were discovered during the simulations while using the RPM* and the BFMT* algorithms. The issue being that using these planning algorithms combined with the implemented `DubinAirplaneState-Space` would cause the paths crashing into the terrain. Figure 4.3 shows how using these algorithms yielded invalid paths cutting through the terrain.

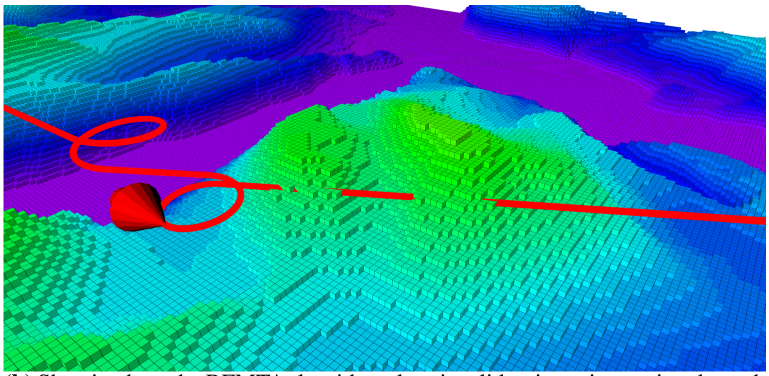
After researching the problem and coming across an issue post on OMPL's Github repository (Moll, 2016) it became apparent that it is not possible to use bidirectional or roadmap based planners on state spaces with asymmetric interpolation. This limitation is caused by the implementation of these planners. A custom implementation for these algorithm with an asymmetric state space might be possible, but falls outside the scope of this thesis.

Further presentation of the simulations will therefore not include any of the results from these algorithms.

Figure 4.3 Illustrating how using the BFMT* and RPM* planning algorithms yielded invalid paths cutting through terrain.



(a) The illustrated path was found using the PRM* algorithm. This results in invalid trajectories cutting through terrain.



(b) Showing how the BFMT* algorithm plans invalid trajectories cutting through terrain.

4.3 Simulations without no-fly zones

To set a baseline for the system's performance simulations were run in an environment without any no-fly zones.

The planning algorithms included in the statistical analysis are IRRT*, BIT* and FMT*. IRRT* was evaluated with regards to given run time. The simulations using BIT* differed in both run time and batch size. BIT* did however not always yield a solution. FMT* was evaluated with regards to the initial sample pool size. These results can be seen in Figures 4.4 to 4.7. Here all planning algorithms were run in 20 simulations each. Other than setting the mentioned parameters the algorithms used their default settings.

From these statistical results is apparent that the IRRT* algorithm yields paths with the shortest distance. By comparing the most demanding simulations of IRRT* and FMT* it can be seen that using the IRRT* with a 50 s run time between checkpoints yields a distribution of shorter paths than using FMT* with a sample pool size of 5000. For the simulations using FMT* the run time was varied, but the median total run time was approximately 700 s, giving an average run time per checkpoint of 140 s. The BIT* algorithm did not always find a solution within the given run time. A trend can be seen in how the sample batch size affects how many paths were found. Looking at the most demanding simulations for the BIT* algorithm the distribution of path lengths seem to become shorter as the batch size increases. However, using a large batch size with a small run time is not convenient when using the BIT* algorithm.

To approximate the path length of an optimal path in this scenario some more resource intensive simulations were run. This was done by running single simulations using the IRRT* and BIT* algorithms with a run time of 300 s between each checkpoint. BIT* was tested with batch sizes of 100, 300 and 1000. The best result was found using 100. This might be a coincidence given the previously mentioned trend in a slightly shorter path found with a bigger batch size for simulations with a run time of 50 s. The FMT* algorithm was run with an initial sample pool size of 15 000. This simulation lasted for a total of 2141 s, giving an average run time between checkpoints of 428.2 s. The resulting path lengths can be seen in Table 4.5. The shortest path was found using the IRRT* algorithm. This was however also the most memory demanding simulation. This simulation almost used all 16 GB of the available system memory.

The distance of the path found using IRRT* for 300 s is illustrated with a horizontal line in Figures 4.4 to 4.7. Comparing the resulting path lengths of the less demanding simulations it is clear that the IRRT* converges asymptotically to an optimal path faster than the other three in this given scenario.

An overview showing the shortest path from Table 4.5 alongside the median distance paths found using IRRT* with 50 s and FMT* with 5000 samples can be seen in Figure 4.8. The behavior of the same paths at checkpoints C_2, C_3, C_4, C_5 and C_6 can be seen in in Figure 4.9.

From Figure 4.8 and 4.9 it is easy to see that the most resource demanding simulation always chooses a more effective trajectory compared to the two less demanding simulations. From the overview in figure 4.8 it is however apparent that they do choose a similar trajectory, but the 300 s simulation minimizes the amount of turns and unnecessary helices.

Algorithm	Path length
Informed RRT*	$84\,204.03\,\mathrm{m}$
FMT*	$87\,233.55\,\mathrm{m}$
BIT* k_{100}	$91\,465.88\,\mathrm{m}$

Table 4.5: Resulting path lengths of three resource demanding simulations. IRRT* and BIT* where both run with $300\,\mathrm{s}$ runtime between checkpoints. BIT* used a batch size of 100. FMT* was wan with a sample pool size of $15\,000$, the entire simulation lasted for $2141\,\mathrm{s}$ for an average runtime of $428.2\,\mathrm{s}$ between checkpoints. All simulations were run in the small scale Sykkylven environment without no-fly zones.

Figure 4.4 Boxplot showing the resulting distributions of path lengths for the simulations using the IRRT* algorithm in the small scale Sykkylven environment without no-fly zones. 20 simulations were run at each run time. The horizontal line at $84\,204\,\mathrm{m}$ marks the shortest path found in this simulation setup, from Table 4.5. Read A.1 for how to interpret the boxplot.

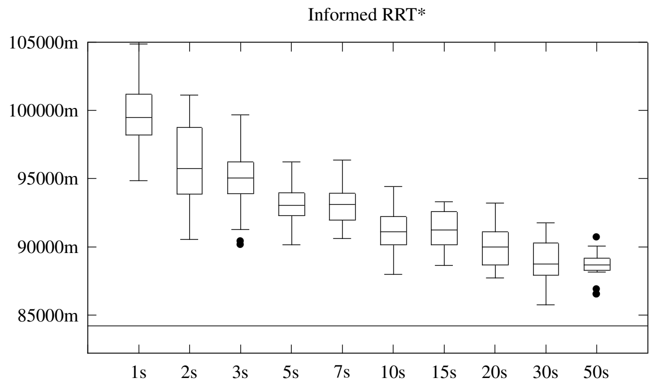
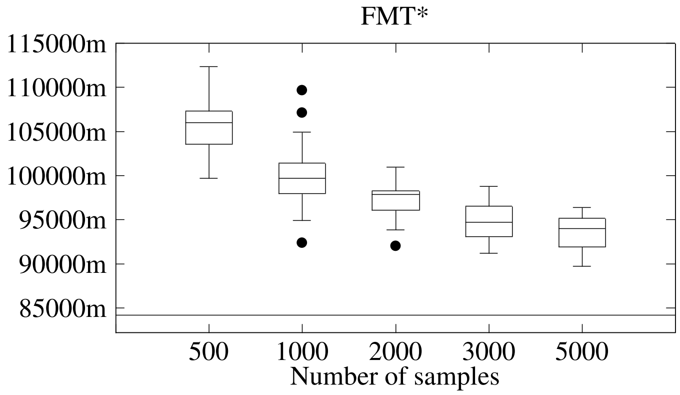
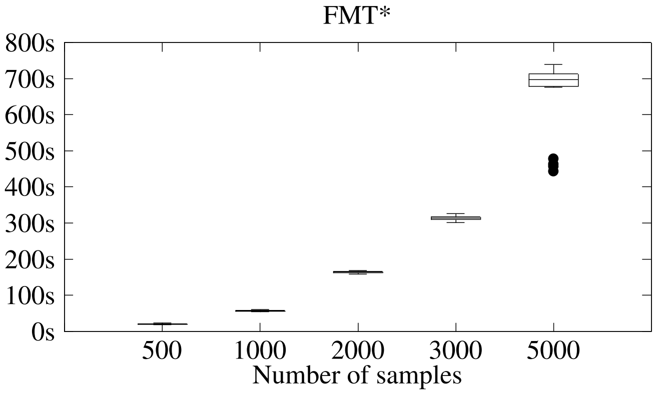


Figure 4.5 Boxplots showing the result for the FMT* planning algorithm, illustrating the distribution of path lengths and run times as function of number of samples chosen initially. The simulations were run in the small scale Sykkylven environment without no-fly zones. The algorithm ran for 20 simulations at each sample size. The horizontal line at 84 204 m marks the shortest path found in this simulation setup, from Table 4.5. Read A.1 for how to interpret the boxplot.

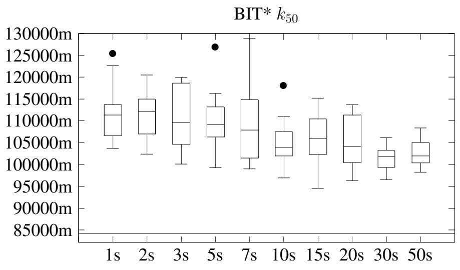


(a) Simulation results using FMT* in an environment without no-fly zones. Comparing the resulting path length to number of samples. The horizontal line at 84 204 m marks the shortest path found in this simulation setup, from Table 4.5.

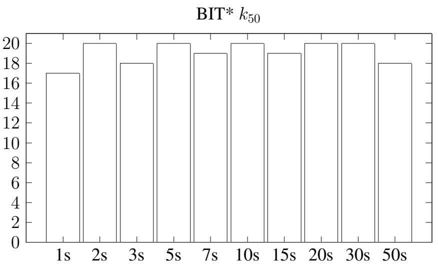


(b) Simulation results using FMT* in an environment without no-fly zones. Comparing the run time to the number of samples.

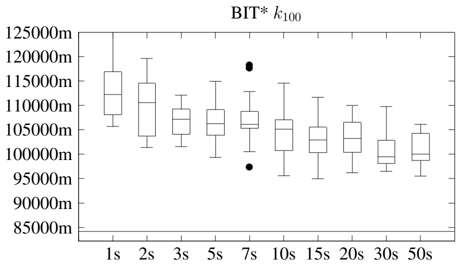
Figure 4.6 The boxplots show the distributions of path lengths at different batch size values and run times using the BIT* algorithm. The bar plot shows the amount of simulations resulting in a path at with the given parameters. The horizontal line at 84 204 m marks the shortest path found in this simulation setup, from Table 4.5. 20 simulations were run at each run time. All simulations were run in the small scale Sykkylven environment without no-fly zones. Read A.1 for how to interpret the boxplot.



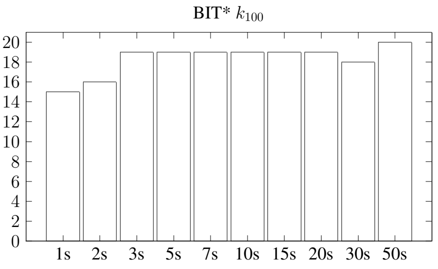
(a) Distribution of path lengths for simulations with a batch size of 50.



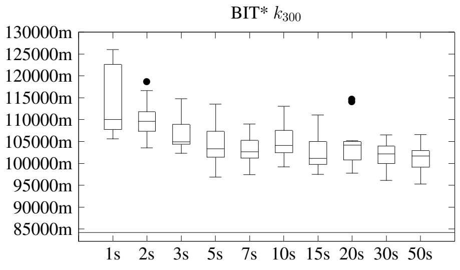
(b) Amount of solutions found by the simulations with a batch size of 50.



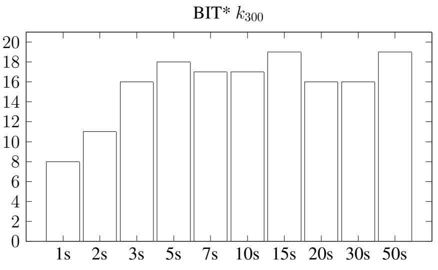
(c) Distribution of path lengths for simulations with a batch size of 100.



(d) Amount of solutions found by the simulations with a batch size of 100.

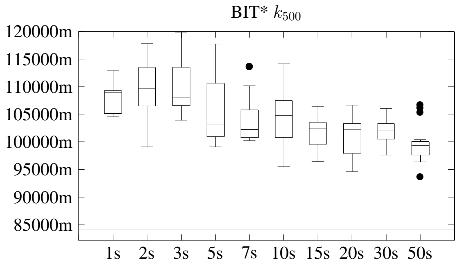


(e) Distribution of path lengths for simulations with a batch size of 300.

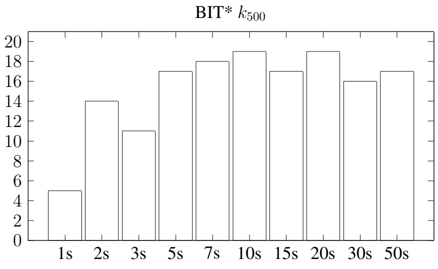


(f) Amount of solutions found by the simulations with a batch size of 300.

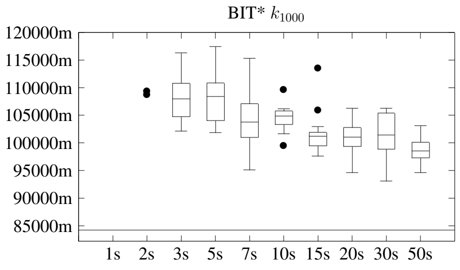
Figure 4.7 The boxplot shows the distribution of path lengths at different batch size values and run times using the BIT* algorithm. The bar plot shows the amount of simulations resulting in a path at with the given parameters. The horizontal line at $84\,204\,\mathrm{m}$ marks the shortest path found in this simulation setup, from Table 4.5. 20 simulations were run at each run time. All simulations were run in the small scale Sykkylven environment without no-fly zones. Read A.1 for how to interpret the boxplot.



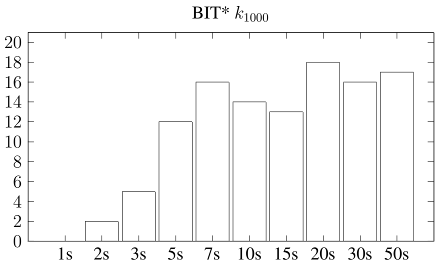
(a) Distribution of path lengths for simulations with a batch size of 500.



(b) Amount of solutions found by the simulations with a batch size of 500.



(c) Distribution of path lengths for simulations with a batch size of 1000.



(d) Amount of solutions found by the simulations with a batch size of 1000.

Figure 4.8 Overview of three paths found in the baseline scenario: with an OctoMap resolution of $50\,\mathrm{m}$ and without no-fly zones. The green path was found using the IRRT* algorithm with a run time of $300\,\mathrm{s}$, this is the shortest path found in this simulation scenario. The orange path was found using the IRRT* algorithm with a run time of $50\,\mathrm{s}$, this is the median result of the simulations with this configuration. The red path was found using the FMT* algorithm with a sample pool size of 5000, this is the median result of the simulations with this configuration.

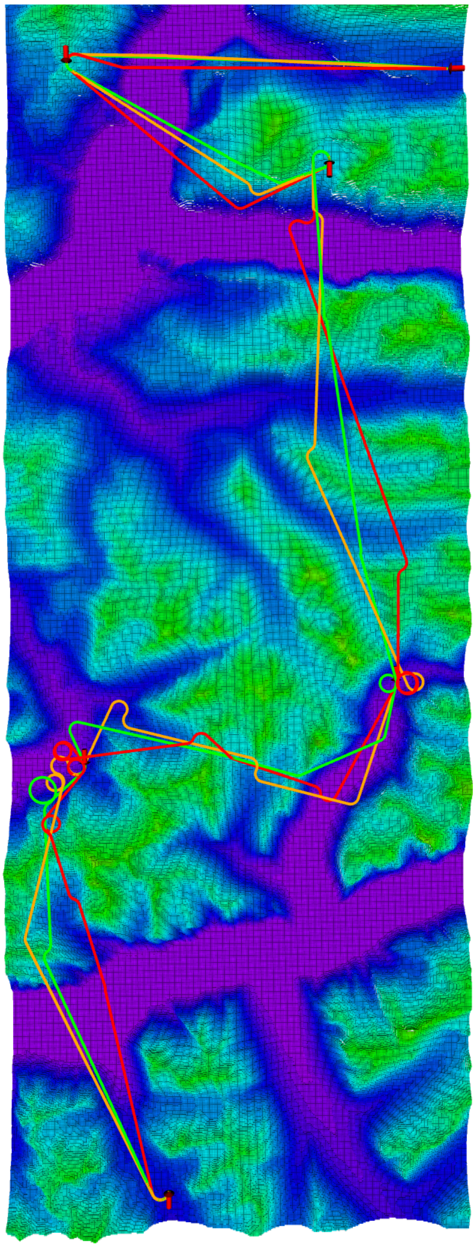
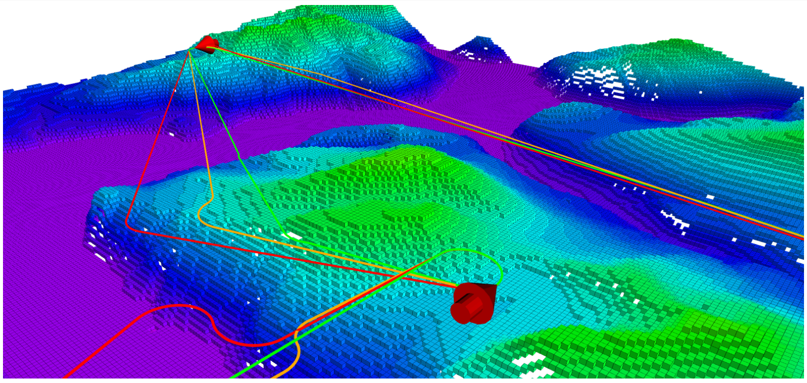
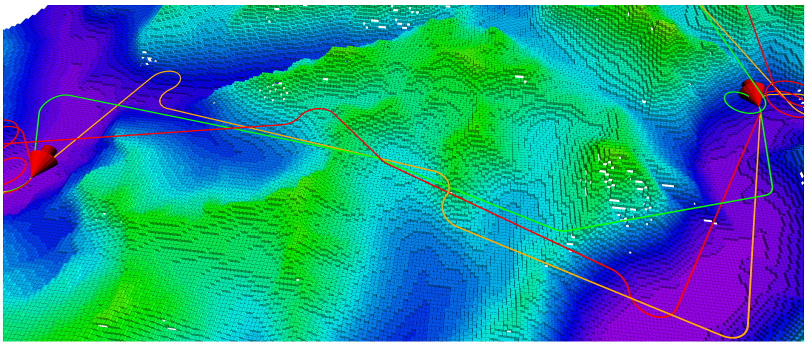


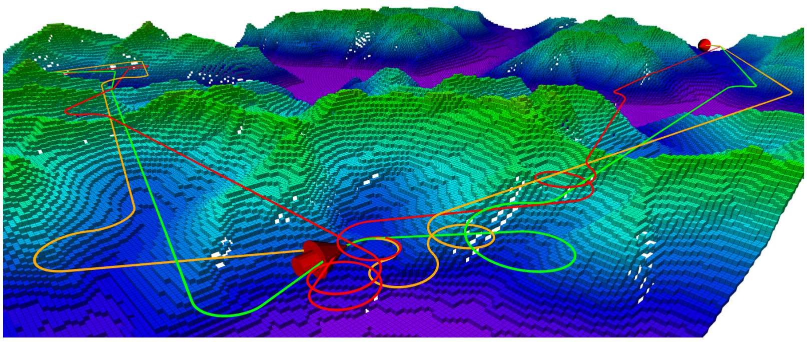
Figure 4.9 Comparisons of the behavior of three at different checkpoints. Found in the baseline scenario with an OctoMap resolution of $50\,\mathrm{m}$ and without no-fly zones. The green path was found using the IRRT* algorithm with a run time of $300\,\mathrm{s}$, this is the shortest path found in this simulation scenario. The orange path was found using the IRRT* algorithm with a run time of $50\,\mathrm{s}$, this is the median result of the simulations with this configuration. The red path was found using the FMT* algorithm with a sample pool size of 5000, this is the median result of the simulations with this configuration.



(a) Behavior at checkpoints C_2 (back) and C_3 (front).



(b) Behavior between checkpoints C_4 (right) and C_5 (left).



(c) Behavior at checkpoints C_5 (front) and C_6 (back right).

4.4 Simulations with no-fly zones

To determine if the implemented system could handle path planning in environments including no-fly zones simulations were run similar to the ones presented in the previous section, but this time the no-fly zones listed in Table 4.4 were included.

The planning algorithms used in these simulations are the same as in Section 4.3, the only difference being in the choice of batch sizes for the BIT* algorithm. For the simulations using BIT* batch sizes of 50, 300 and 1000 samples were used. The distribution of path lengths from the different algorithms can be seen in Figures 4.10 to 4.12. Similarly to the previous section 20 simulations were run for each parameter with every algorithm.

From the statistical results we see a similar trend in performance between the algorithms as in the scenario without no-fly zones. Here, of course, all paths are longer as the no-fly zones are placed so that they obscure any straight overhead line between checkpoints. Comparing simulations using IRRT* with a $50\,\mathrm{s}$ run time to simulations using FMT* with a sample pool size of 5000 the former yielded a distribution centered around a shorter path distance. The simulations using FMT* with a sample size of 5000 had a median total run time of $750\,\mathrm{s}$ giving an average run time between checkpoints of roughly $150\,\mathrm{s}$, triple the amount from used by IRRT*. For simulations with BIT* the distances do not converge towards the optimum as quickly as with the IRRT* algorithm. Similarly to the previous simulations increasing the batch size affects the amount of runs yielding a solution, especially at lower run times.

Some more resource demanding simulation were run to approximate an optimal minimum path length in this environment. A simulation using the IRRT* algorithm was run for $300\,\mathrm{s}$ as well as a simulation using the FMT* algorithm with a sample pool size of 10 000. The FMT* simulation lasted for a total of $1315\,\mathrm{s}$, giving an average run time between checkpoints of $263\,\mathrm{s}$. The BIT* algorithm was tested with different batch sizes with run times of both $300\,\mathrm{s}$ and $500\,\mathrm{s}$, the best result was found with a batch size equal $k = 50$ and a run time of $500\,\mathrm{s}$. The resulting path lengths can be seen in Table 4.6. Similarly to the previous simulations the simulation using IRRT* yielded the shortest path length. The shortest path found through these simulations are also marked in Figures 4.10 to 4.12 with a horizontal line.

Figures 4.13 and 4.14 show a comparison of paths found using the FMT* algorithm with different initial sample pool sizes in the environment. Specifically, paths found using 10 000 and 7000 samples, and the median path found using 5000 samples are shown. One noteworthy observations is how the path found using the most samples flew around no-fly zone Z_1, while the other two took a u-turn at checkpoint C_2. This seemed to be a common trend, where the more resource intensive simulations chose the path going around the no-fly zone. Another observation is how the path found using 5000 samples traverses over no-fly zone Z_2, this was one of the few inspected paths that chose this route.

Figures 4.15 and 4.16 compare paths found using the IRRT* and BIT* algorithm. The paths shown found by the IRRT* algorithm used run times of $300\,\mathrm{s}$, $50\,\mathrm{s}$ and $10\,\mathrm{s}$. For the prior the shortest path is shown for the latter two the median distance paths are shown. The paths found using the BIT* algorithm used run time $300\,\mathrm{s}$ and $500\,\mathrm{s}$.

Contrary to to the paths found by FMT* here both the shortest and longest path found by IRRT* flew around the first no-fly zone. This indicates the segments might be of close to equal length, and the choice of trajectory is more dependent on the random state sampling.

Algorithm	Path length
Informed RRT*	$103\,431.91\,\mathrm{m}$
FMT*	$110\,288.83\,\mathrm{m}$
BIT* k_{50}	$116\,267.80\,\mathrm{m}$

Table 4.6: Shortest path lengths found by three resource demanding simulations. The simulation using Informed RRT* ran for 300 s, the simulation using FMT* had a batch size of 10 000 and the simulation with BIT* ran for 500 s with a sample batch size of 50 samples.

Figure 4.10 Boxplot showing the distribution of path lengths found as a result for simulations using the Informed RRT* algorithm in the Sykkylven environment including no-fly zones. 20 simulations were run at each run time. The horizontal line marks the minimum path found in this setup from Table 4.6. See appendix A.1 for how to interpret the boxplot.

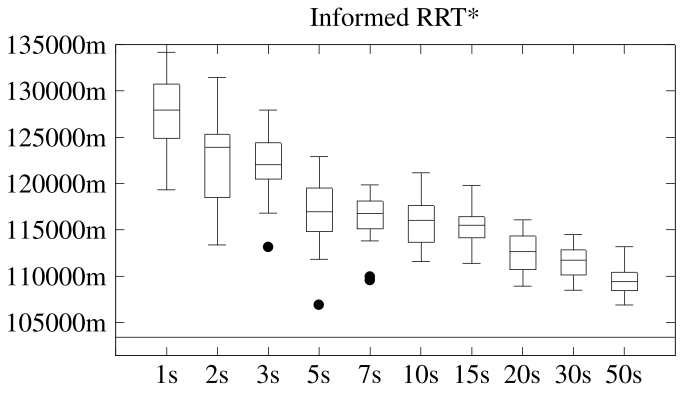
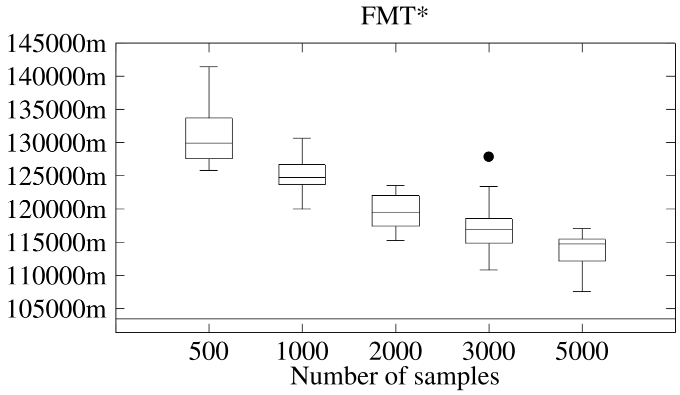
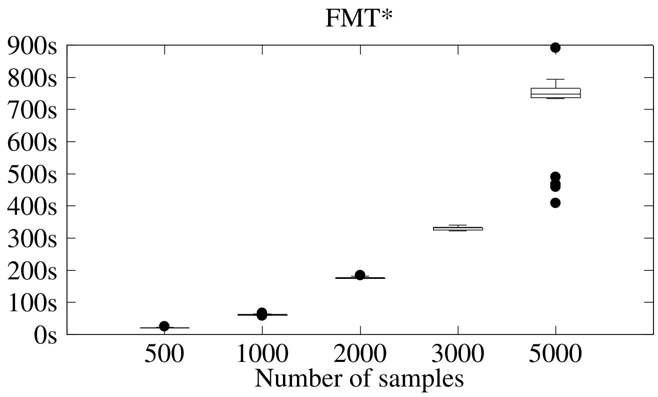


Figure 4.11 Boxplot showing the distribution of path lengths found and run times simulations using the FMT* algorithm in the Sykkylven environment including no-fly zones. 20 simulations were run at each run time. See appendix A.1 for how to interpret the boxplot.

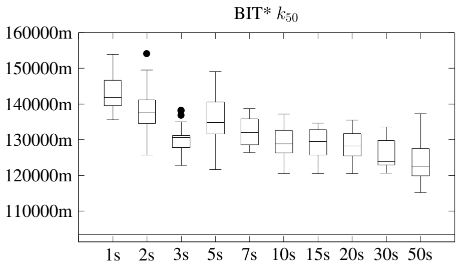


(a) Distribution of resulting path lengths in relation to number of samples initially chosen. The horizontal line marks the minimum path found in this setup from Table 4.6.

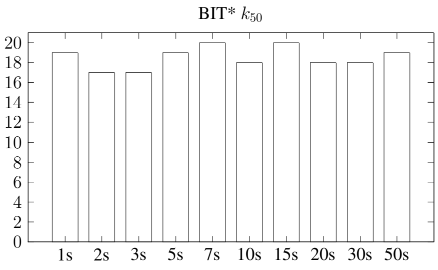


(b) Distribution of total run time in relation to number of samples initially chosen.

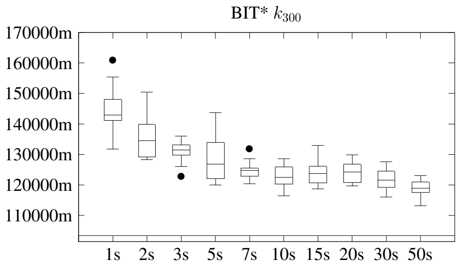
Figure 4.12 Boxplot showing the distribution of path lengths and number of paths found for simulations using the BIT* algorithm in the Sykkylven environment including no-fly zones. 20 simulations were run at each run time. See appendix A.1 for how to interpret the boxplot.



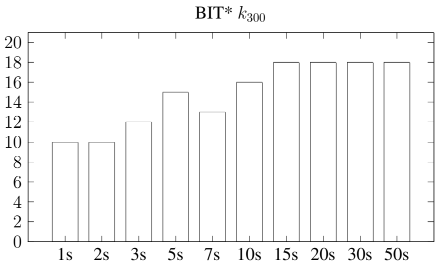
(a) Path lengths found using a batch size of 50.



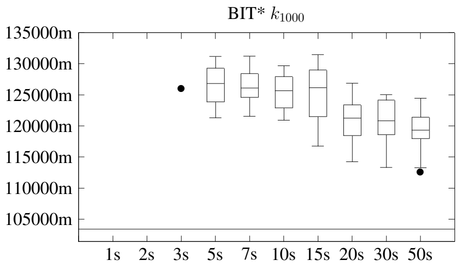
(b) Number of solutions found by the simulations with a batch size of 50.



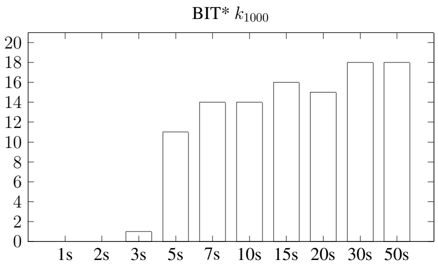
(c) Path lengths found using a batch size of 300.



(d) Number of solutions found by the simulations with a batch size of 300.

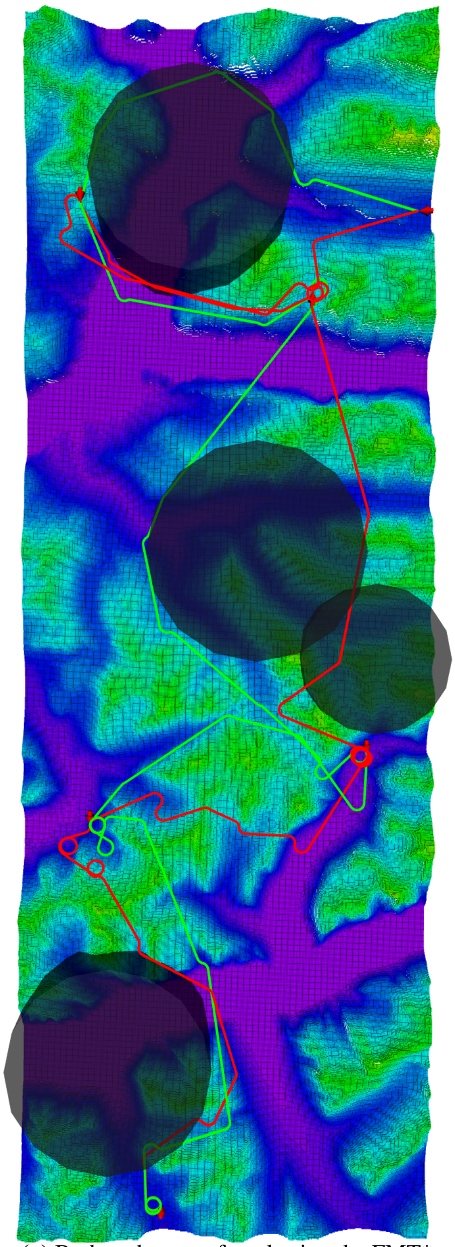


(e) Path lengths found using a batch size of 1000.

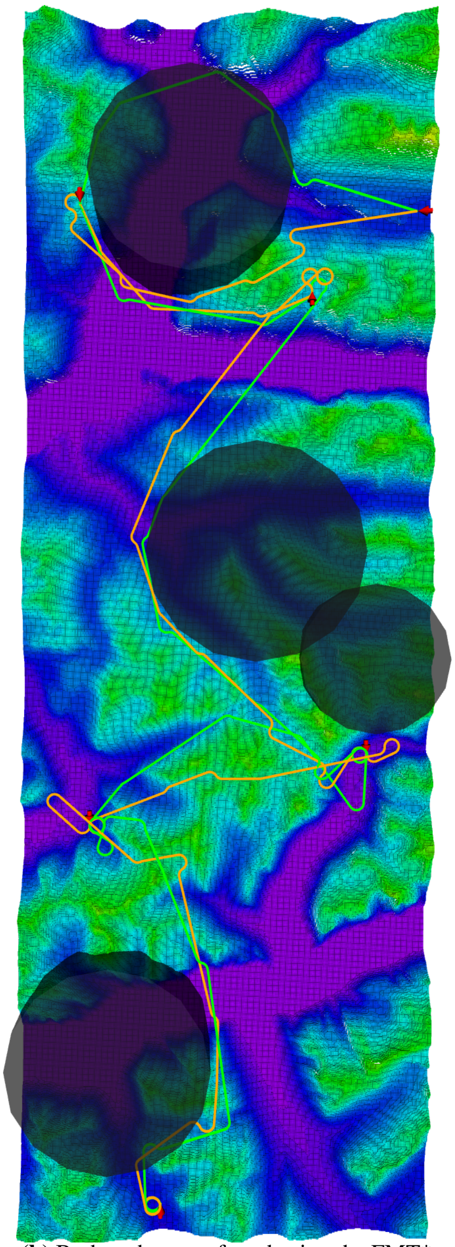


(f) Number of solutions found by the simulations with a batch size of 1000.

Figure 4.13 Overview of some paths found using the FMT* algorithm in the Sykkylven environment with a resolution of $50\,\mathrm{m}$ including no-fly zones.

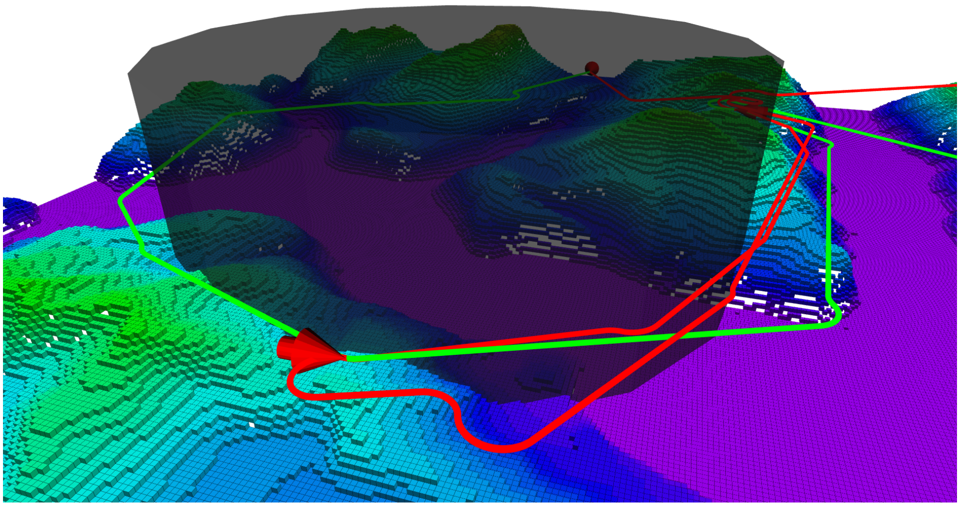


(a) Both paths were found using the FMT* algorithm. The green path used an initial sample pool size of 10 000 while the red path is the median path found using 5000 samples.

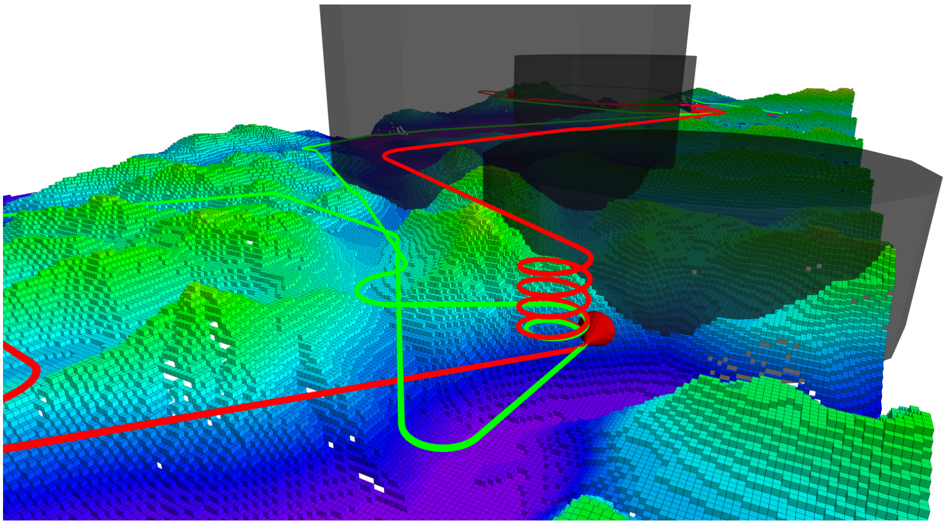


(b) Both paths were found using the FMT* algorithm. The green path had an initial sample pool size of 10 000, while the orange path had 7000.

Figure 4.14 A comparison of two paths found using the FMT* algorithm. The green path used an initial sample pool size of 10 000, while the red path is the median path found using 5000 samples.

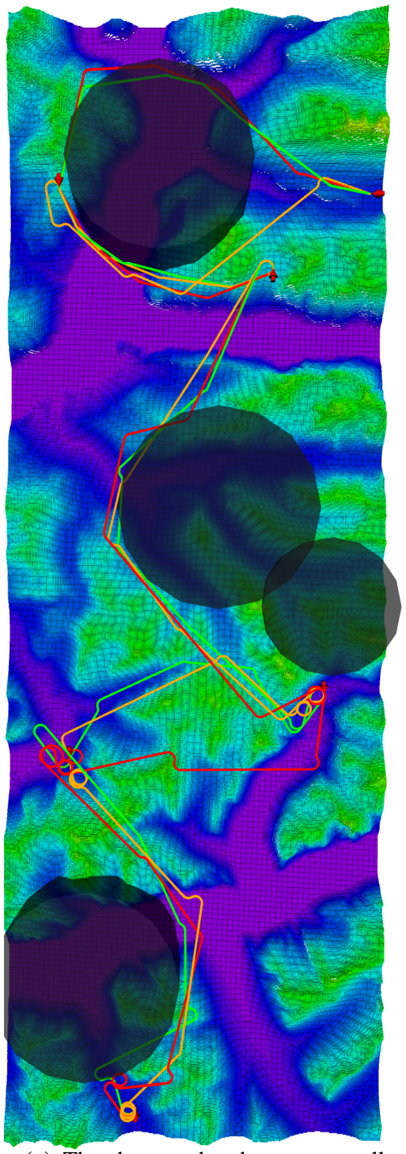


(a) The path found using more samples chose a different path around the first no-fly zone.

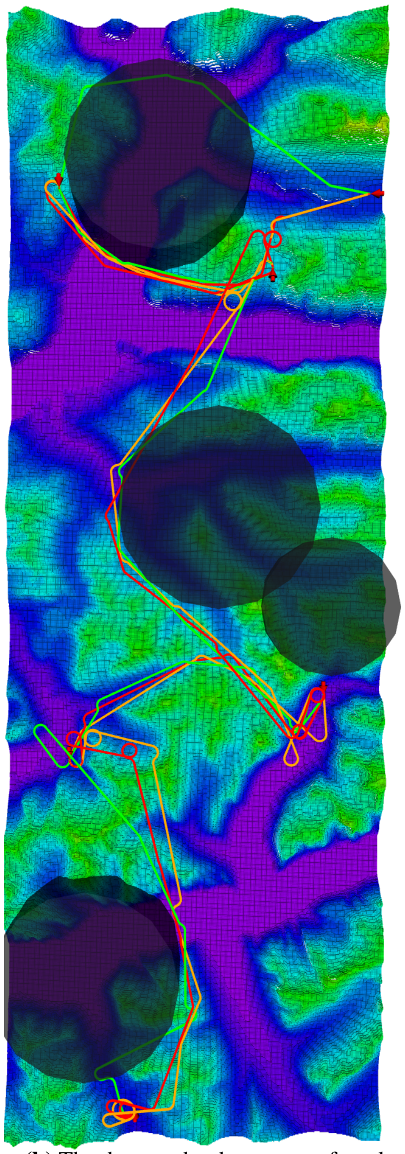


(b) The path found using 5000 samples chose to fly over the second no-fly zone.

Figure 4.15 Overview of paths found using the Informed RRT* and BIT* algorithms in the Sykkylven environment including no fly zones.

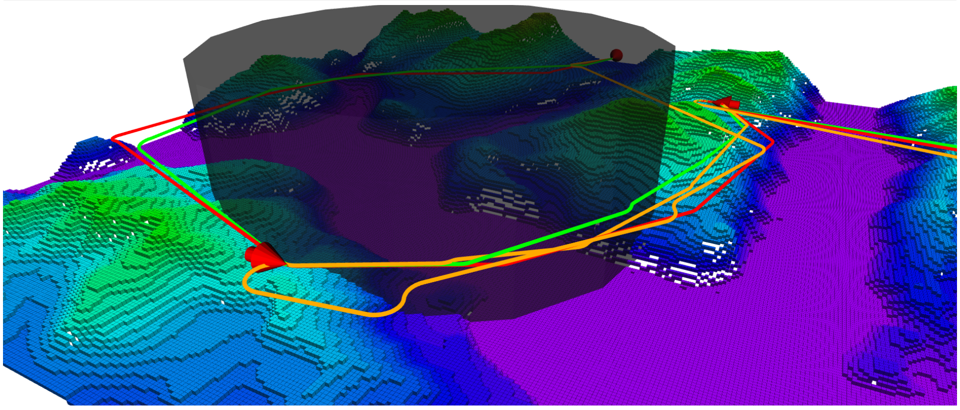


(a) The three paths shown were all found using the Informed RRT* algorithm. The green path used a run time of 300 s, the orange path is the median path found with run time 50 s and the red path is the median path found with run time 10 s.

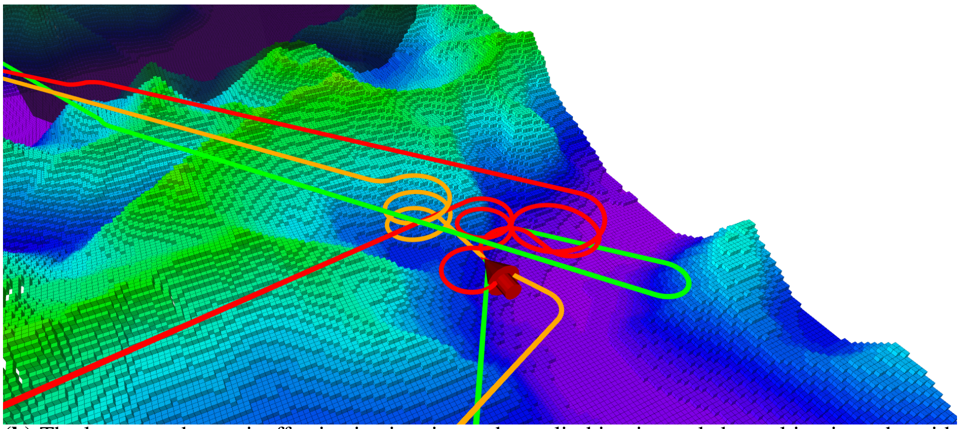


(b) The three paths shown were found using the Informed RRT* and BIT* algorithm. The green path used Informed RRT* with a run time of 300 s, the orange path used BIT* with 500 s and the red path used BIT* with 300 s. Both BIT* algorithms used a batch size of 50 samples.

Figure 4.16 A comparison of three paths found using the Informed RRT* algorithm. The green path used a run time of 300 s, the orange path is the median path found with run time 50 s and the red path is the median path found with run time 10 s.



(a) Both the shortest and longest path chose to fly around the first no-fly zone.



(b) The longer paths are ineffective in situations where climbing is needed, resulting in paths with long and unnecessary helices. The shortest path uses as few turns as possible.

4.5 Simulations in a large scale environment

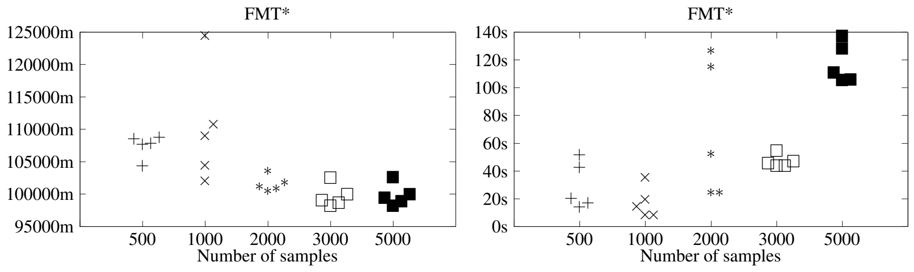
The results presented in the up until now indicates that the Informed RRT* algorithm is the best choice of planning algorithm if the scale of the environment and distance between goal and start configurations matches the first environment described in Section 4.1. The following section will present simulations run in an environment of a much larger scale. As mentioned in Section 4.1 this environment is also based on terrain from the Sykkylven municipality, but this model is a super set of the first environment presented. This larger environment spans an area of $50\,000\,\mathrm{m} \times 100\,000\,\mathrm{m}$, over 5 times the size of the smaller environment.

Due to the scale of this environment a minor tweak had to be made to the OMPL settings. The resolution at which state validity is checked had to be increased via `ompl::base::SpaceInformation::SetValidityCheckingResolution()`. Meaning state validity checks will happen more often. The default value of 0.01 was changed to 0.001. This value is specified as a fraction of the environments extent.

These simulations only included two checkpoints, a start and goal, with a Euclidean distance of $90\,077.74\,\mathrm{m}$ between them. The environment was not augmented by any no-fly zones, but a maximal height restriction of $700\,\mathrm{m}$ was added. This gave the environment a maze-like nature, where the planner could not traverse over the high terrain, but had to find a trajectory following the open water and the fjords to reach the goal.

The planning algorithms tested in these simulations were Informed RRT* and FMT*. Contrary to the previously presented results the FMT* algorithm came out on top. After running 5 simulation of the Informed RRT* algorithm with a run time of $300\,\mathrm{s}$ none of the results yielded a trajectory where the goal state was reached. With the FMT* algorithm, however, 5 simulations each at sample sizes of 500, 1000, 2000, 3000 and 5000 all yielded valid paths to the goal. This showcases the FMT* algorithm's strength is scenarios where collision checking is expensive and obstacles are frequent. The distribution of path lengths and run times can be seen in Figure 4.17. Figure 4.18 and 4.19 shows the median path found using the FMT* algorithm with an initial sample pool size of 5000 samples.

Figure 4.17 Scatter plot showing the paths lengths and run times given by simulations with the FMT* algorithm in the large scale Sykkylven environment. Five simulations were run at each sample pool size.



(a) Resulting path lengths in relation to number of samples initially chosen.

(b) Run times in relation to number of samples initially chosen.

Figure 4.18 The median distance path found using the FMT* algorithm with a initial sample pool size of 5000 in the large scale Sykkylven environment.

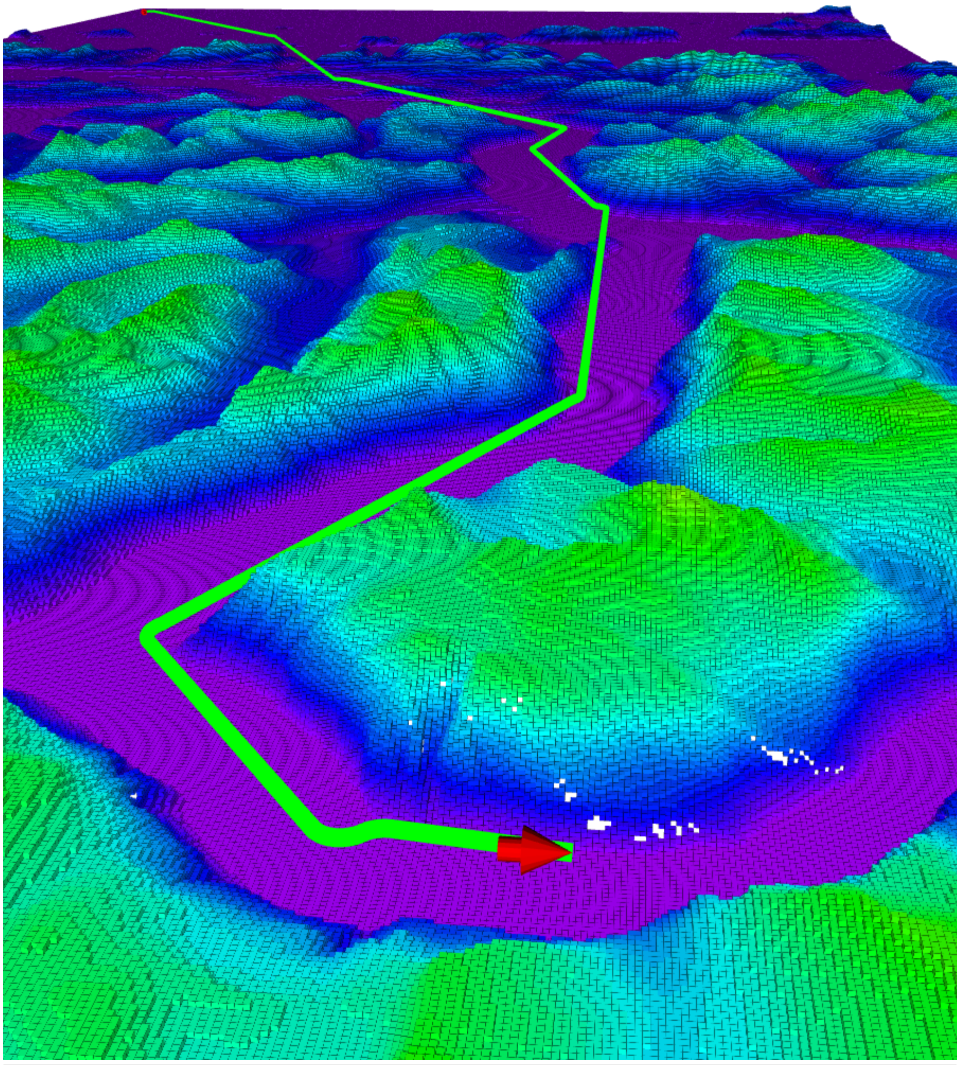
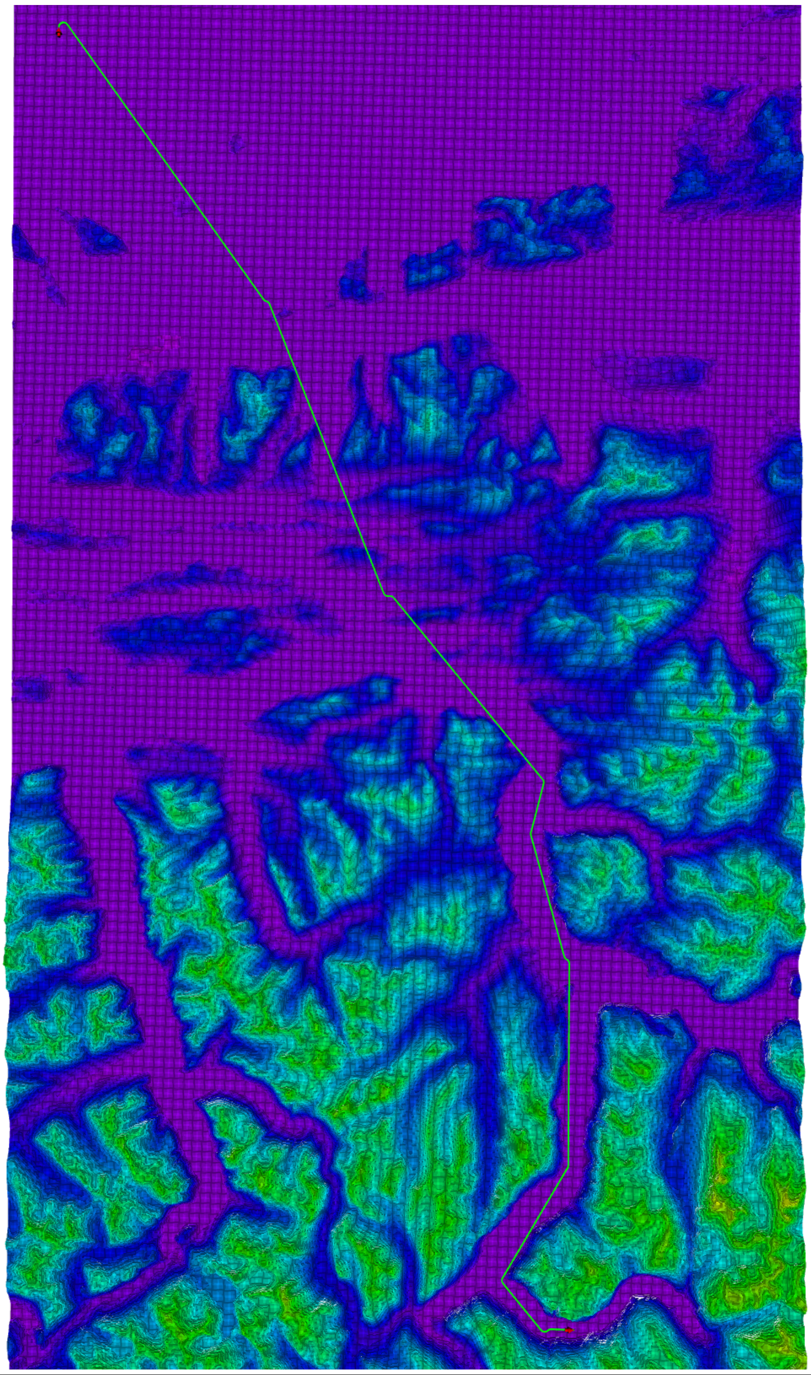


Figure 4.19 An overview of the median path found using the FMT* algorithm with a sample pool size of 5000 in the large scale Sykkylven environment.



Chapter 5

Discussion

The following chapter will discuss the results presented in the previous chapter and reflect upon choosing a planning algorithm for the system. Afterwards a summary of potential areas of improvement for the implemented planner and further work will be given.

By evaluating the results from Chapter 4 it is apparent that the implemented system meets the requirements presented in the beginning of Chapter 3. The resulting planned trajectories obey the dynamics of the Dubins airplane model through the use of the implemented `DubinsAirplaneStateSpace`. This model doea in turn mimics the dynamics of a fixed-wing UAV. The system has the capability to plan trajectories in real world terrain data and avoid any potential no-fly zones in this terrain. It also manages to keep a minimal safety distance between the trajectory and any obstacle, terrain or no-fly zone, in the environment. Through dividing the ordered list of checkpoints to be visited into separate individual planning problems and solving them in turn the system is able to plan paths linking together multiple target locations. Lastly, by evaluating the systems performance with different planning algorithms an opinion can be made about which is best suited for path efficiency.

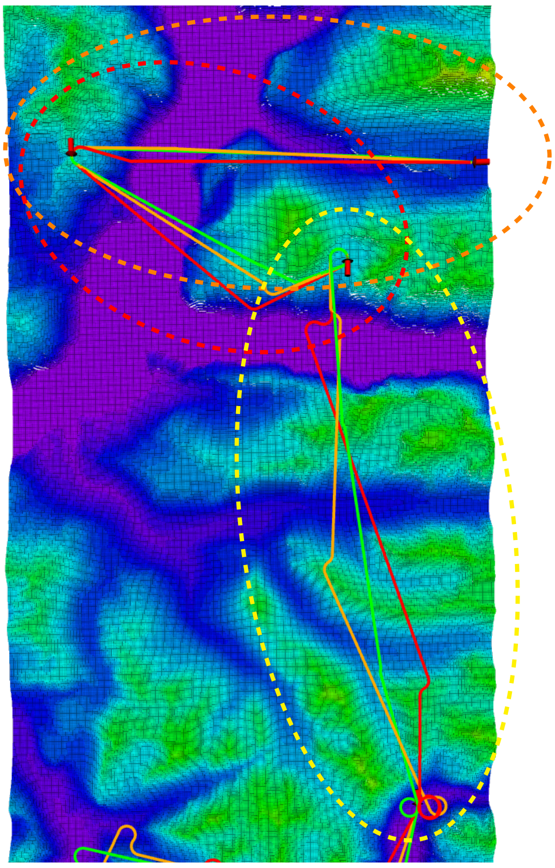
The results presented in Sections 4.3 and 4.4 indicate that using the Informed RRT* algorithm is a good choice when the environment is similar the one used described in Section 4.1. However, this algorithm fell short compared to the FMT* algorithm in the simulations using an environment of a larger scale, as shown in Section 4.5. This is not necessarily due to the size of this environment, but rather due to the maze-like nature caused by the maximal height restriction. This scenario showcased the strengths of the FMT* algorithm: being efficient in environments with many obstacles and where collision checking is expensive. In simulations without this height restriction both FMT* and the Informed RRT* algorithm found a path simply by flying over the terrain until the goal was reachable.

For the task of planning trajectories to be used for transmission mast and power grid inspection the Informed RRT* algorithm would be the best choice in most situations. In this scenario a height restriction, such as the one in Section 4.5, is unlikely. This choice of planning algorithm is consistent with the conclusion given by Schneider (2016). The different capabilities of the planning algorithms should, however, be taken into account when designing a planing problem, and the choice of planner might vary based on specific problems.

5.1 Further work

As mentioned in the beginning of Chapter 4 all the planning algorithms tested in the presented simulations were out-of-the-box. As the majority of the time spend working on this project went to implementing the dynamics of a fixed-wing UAV into in the Open Motion Planning Library there is a potential performance increase available by further tuning the planning algorithms. Better performance could also be made possible through customiz-

Figure 5.1 How the terrain model can be cropped to encircle only a selection of terrain relevant for path planning between two checkpoints. Each dashed ellipse represent the relevant terrain for an individual planning problem.



ing the algorithms to fit the implemented `DubinsAirplaneStateSpace` better. One possibility is to create a state sampling scheme that fits the system dynamics, as seen in the project presented by Schneider (2016).

Furthermore, the implemented system proved itself to be quite resource demanding, especially with regards to system memory. Little focus has been put into optimizing the implemented planner regarding resource usage, such as memory or cpu utilization, this might also be an area where better performance could be retained. When planning a path connecting two checkpoints it might be unnecessary to use a terrain model of an area much larger than whats relevant. E.g. looking at Figure 4.2 it is obvious that the OctoMap does not need to model the terrain around checkpoints C_4, C_5 and C_6 when planning a path to connect checkpoints C_1 and C_2. A system could be implemented to crop the terrain model to only include the necessary area encircling the two checkpoint. The idea of such a system is illustrated in Figure 5.1, where each dashed ellipse contain the terrain relevant for planning between two checkpoints. This, of course, relies heavily on assuming the terrain is not shaped in a way so that a more optimal path could lie outside the cropped boundary. It also assumes it is more effective to fly over terrain then around. There is also a possibility that two checkpoints will be separated by a no-fly zone, so some method must be implemented to make sure the cropped terrain is a continuous space. Such an implementation could free up resources to be put elsewhere, e.g. into bettering the resolution of the terrain OctoMap to give safer and more accurate trajectories, or giving the simulations more run time.

This could be extended even further by solving the planning problems in parallel. Once an adequate terrain model is generated between two checkpoints the planning problem could be handed off to a separate computing system and a terrain model for the next checkpoint pair can be generated, this could in turn be handed of to a second separate computing system. Such a parallel pipeline could be used to ensure an effective and, most importantly, safe path is found between all checkpoints while reducing computation time significantly.

Chapter 6

Conclusion

As a final conclusion to the work behind this thesis it is evident that using the Open Motion Planning Library as a framework in a planning system to find trajectories for fixed-wing UAVs is a solid choice. Sample-based planning is a standard approach to robotic motion planning as it gives asymptotically optimal solutions well suited for real world navigation.

As in the case of the implemented system using the Dubins airplane model and its closed form solutions to the boundary value problem connecting configurations with almost optimal paths gave reasonable results. By setting a conservative minimal turn radius and climb rate the outcome are paths comparable to that of a fixed-wing UAV. Through the results presented in Chapter 4 a suitable choice of planning algorithm is Informed RRT*. In some cases the FMT* algorithm should also be considered.

The implemented `DubinAirplaneStateSpace` embedded this behavior in an effective manner. Combined with the Flexible Collision Library and the OctoMap framework the implemented system is capable of finding airplane like trajectories in real world terrain while respecting cylinder shaped no fly zones.

Bibliography

Chitsaz, H., LaValle, S.M., 2007. Time-optimal paths for a dubins airplane, in: Proceedings of the IEEE Conference on Decision and Control, pp. 2379–2384. doi:10.1109/CDC.2007.4434966.

Dubins, L.E., 1957. On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents. American Journal of Mathematics 79, 497. doi:10.2307/2372560.

Eger, B.S., 2019. Long-range path planning for fixed-wing UAVs using A* search and OctoMap. Ph.D. thesis. The Norwegian University of Science and Technology.

Gammell, J.D., Srinivasa, S.S., Barfoot, T.D., 2014. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic, in: IEEE International Conference on Intelligent Robots and Systems, Institute of Electrical and Electronics Engineers Inc.. pp. 2997–3004. doi:10.1109/IROS.2014.6942976, arXiv:1404.2334.

Gammell, J.D., Srinivasa, S.S., Barfoot, T.D., 2015. Batch Informed Trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs, in: Proceedings - IEEE International Conference on Robotics and Automation, Institute of Electrical and Electronics Engineers Inc.. pp. 3067–3074. doi:10.1109/ICRA.2015.7139620, arXiv:1405.5848.

Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C., Burgard, W., 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Autonomous Robots 34, 189–206. doi:10.1007/s10514-012-9321-0.

Janson, L., Schmerling, E., Clark, A., Pavone, M., 2015. Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions. The International journal of robotics research 34, 883–921. URL: http://www.ncbi.nlm.nih.gov/pubmed/27003958http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4798023, doi:10.1177/0278364915577958.

Karaman, S., Frazzoli, E., 2011. Sampling-based Algorithms for Optimal Motion Planning URL: http://arxiv.org/abs/1105.1186, arXiv:1105.1186.

Kavraki, L.E., Švestka, P., Latombe, J.C., Overmars, M.H., 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation 12, 566–580. doi:10.1109/70.508439.

LaValle, S.M., 2006. Planning algorithms. volume 9780521862. Cambridge University Press, Cambridge. URL: https://www.cambridge.org/core/product/identifier/9780511546877/type/book, doi:10.1017/CBO9780511546877.

McLain, T., Beard, R., Owen, M., 2014. Implementing Dubins Airplane Paths on Fixed-wing UAVs. Faculty Publications URL: https://scholarsarchive.byu.edu/facpub/1900.

Moll, M.m., 2016. GeometricCarPlanning demo gives invalid solutions for dubins state space · Issue #329 · ompl/ompl. URL: https://github.com/ompl/ompl/issues/329.

Pan, J., Chitta, S., Manocha, D., 2012. FCL: A general purpose library for collision and proximity queries, in: Proceedings - IEEE International Conference on Robotics and Automation, Institute of Electrical and Electronics Engineers Inc.. pp. 3859–3866. doi:10.1109/ICRA.2012.6225337.

Schneider, D., 2016. Master Thesis Path Planning for Fixed-Wing Unmanned Aerial Vehicles URL: https://www.research-collection.ethz.ch/handle/20.500.11850/116625, doi:10.3929/ethz-a-010646508.

Shkel, A.M., Lumelsky, V., 2001. Classification of the Dubins set. Robotics and Autonomous Systems 34, 179–202. URL: https://www.sciencedirect.com/science/article/pii/S0921889000001275?via{%}3Dihub, doi:10.1016/S0921-8890(00)00127-5.

Starek, J.A., Gomez, J.V., Schmerling, E., Janson, L., Moreno, L., Pavone, M., 2015. An Asymptotically-Optimal Sampling-Based Algorithm for Bi-directional Motion Planning URL: http://arxiv.org/abs/1507.07602http://dx.doi.org/10.1109/IROS.2015.7353652, doi:10.1109/IROS.2015.7353652, arXiv:1507.07602.

Şucan, I.A., Moll, M., Kavraki, L., 2012. The open motion planning library. URL: https://ompl.kavrakilab.org/index.html, doi:10.1109/MRA.2012.2205651.

Appendices

Appendix A

A.1 Reading boxplots

The top and bottom edge of each box mark the edges for the 25th and 75th percentiles, respectively. The line in the middle of the box is the median observation. The whiskers extend to include the most distant observation within 1.5 times the interquartile range. Beyond this outliers are marked with dots.

A.2 Code snippets

Listing A.1: Implementation of the planner system. The code is cleaned for readability by removing imports, global variables, namespaces etc.

```
// Collision check module
bool isStateValid(SpaceInformation *si, State *state,
                  shared_ptr<OcTree> map,
                  shared_ptr<CollisionManagerd> manager)
{
    shared_ptr<Sphere> geom(new Sphere(safeVolumeRadius));
    Transform3d pose = Transform3d::Identity();
    Vector3d tran = Vector3d(s->getX(), s->getY(), s->getZ());
    pose.translation() = tran;

    CollisionObject* obj = new fcl::CollisionObject(geom, pose);
    CollisionData cdata;

    manager->collide(obj, &cdata, defaultCollisionFunction);

    return si->satisfiesBounds(state) && !cdata.result.isCollision();
}

// Planning algorithm
void plan(Checkpoint start,
          Checkpoint goal,
          shared_ptr<OcTree> map,
          shared_ptr<CollisionManagerd> manager,
          Planner planner,
          vector<PathGeometric> paths)
{
    StateSpacePtr space =
        make_shared<DubinsAirplaneStateSpace>(turnRadius, climbAngle);

    double xMax, yMax, zMax;
    map->getMetricMax(xMax, yMax, zMax);
    double xMin, yMin, zMin;
    map->getMetricMin(xMin, yMin, zMin);

    RealVectorBounds bounds(3);
    bounds.setHigh(0, xMax);
    bounds.setHigh(1, yMax);
    bounds.setHigh(2, zMax);

    bounds.setLow(0, xMin);
    bounds.setLow(1, yMin);
    bounds.setLow(2, zMin);

    space->as<SimpleSE3StateSpace>()->setBounds(bounds);

    SimpleSetup ss(space);
    SpaceInformation *si = ss.getSpaceInformation().get();

    // Set Collision check module.
    ss.setStateValidityChecker([si, map, manager](State *state) {
        return isStateValid(si, state, map, manager);
    });

    // Must be set on use, as its not done by the OMPL contrib.
    si->setMotionValidator(make_shared<DubinsAirplaneMotionValidator>(si));

    // Set start and goal checkpoints.
    ScopedState<SimpleSE3StateSpace> start(space);
    start->setXYZYaw(start.x, start.y, start.z, start.yaw);
    ScopedState<SimpleSE3StateSpace> goal(space);
    goal->setXYZYaw(goal.x, goal.y, goal.z, goal.yaw);
    ss.setStartAndGoalStates(start, goal);

    // Planner given as parameter and initialized outside this scope.
    ss.setPlanner(planner);
    ss.setup();

    // Solver run time is set to value from ROS parameter server.
    // Variable solverRunTime is defined globally.
    PlannerStatus solved = ss.solve(solverRunTime);

    if (solved)
    {
        ss.simplifySolution();
        PathGeometric path = ss.getSolutionPath();
        paths.push_back(path);
    }
    else
    {
        // Detect if no path is found by comparing
        // length of paths with number of checkpoints.
        cout << "No solution found" << endl;
        return;
    }
    return;
}

int main(char argc, char* argv)
{
    vector<Checkpoint> checkpoints;
    vector<NoFlyZone> zones;
    vector<PathGeometric> paths;
    StateSpacePtr space;
    Planner planner;

    // These functions also set global variables such as run time,
    // safety radius, turn radius, climb angle.
    readParameterServer(checkpoints, zones, planner);
    parseArgs(argc, argv);

    auto map = readOctomapTopic();
    auto manager = setupCollisionManager(map, zones);

    for (int i = 0; i < checkpoints.size() - 1; i++)
    {
        plan(checkpoints[i], checkpoints[i+1], map, manager, planner, paths);
    }

    if (paths.size() != checkpoints.size() - 1)
    {
        writeFail();
        return 0;
    }
    writePaths(paths);
    return 1;
}
```