Trajectory (re)planning of a quadcopter in indoor environments with unknown obstacles

DIPLOMA THESIS

Conducted in partial fulfillment of the requirements for the degree of a

Diplom-Ingenieur (Dipl.-Ing.)

supervised by

Univ.-Prof. Dr. techn. A. Kugi
M. N. Vu, M.Eng.

submitted at the

TU Wien
Faculty of Electrical Engineering and Information Technology
Automation and Control Institute

by
Martin Zimmermann
Matriculation number 01604997

Vienna, October 2022

Complex Dynamical Systems Group
A-1040 Wien, Gußhausstr. 27–29, Internet: https://www.acin.tuwien.ac.at

Preamble

At this point, I would like to thank all those who have contributed to the success of this master thesis through their professional and personal support.

My special thanks go to my supervisor Minh, for offering this exciting topic and his exceptional support, be it in the form of setting up software, mathematical issues or guidance to practical solutions. I will always be grateful for his flexibility, motivation in desperate times and advices for the future.

I would also like to thank Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Kugi for the mediation of this work, for his commitment as a lecturer in the introductory and in-depth courses of automation and control engineering, in order to give each student a profound understanding of this subject.

Furthermore, I cannot thank Dr. Dr. Werner Mach enough for always supporting this work with patience and dedication as part of my student work placement at Siemens AG.

A special thanks goes to my family, first and foremost my parents, who supported me in every possible way throughout my studies. Also, a heartfelt thanks to my girlfriend Melanie for her continued personal support during my studies and for proofreading this thesis. In this context I would also like to thank my siblings Doris, Carola and Gerhard, who also agreed to proofread this work.

Vienna, October 2022

Abstract

In recent decades, the development of unmanned aerial vehicles (UAVs), driven by research and industry, has been playing an important role in various fields. In this thesis, the trajectory planning and obstacle avoidance of a quadcopter in indoor environments with unknown obstacles is presented.

The quadcopter is equipped with a single-board computer and an autopilot program. The single-board computer executes the Robot Operating System (ROS), which is used to specify the trajectory as well as to process data from sensors of the quadcopter. The autopilot program takes over the hardware-related commands and calculates the motor control outputs based on the computed trajectory and data from sensors.

The proposed trajectory planning consists of two following steps. First, a collision-free path consisting of several waypoints is computed by using the optimal rapidly exploring random tree (RRT*) algorithm. Note that the line-of-sight (LOS) algorithm is employed to smooth and reduce as much as possible the number of waypoints of the computed path. Second, a constrained quadratic program is utilized to calculate the control inputs by exploiting the differential flatness property of the quadcopter.

In addition, a depth camera is used to detect unknown obstacles in the environment. Thereby, the quadcopter can decide whether to activate the replanning algorithm or to follow a computed one.

Finally, the proposed methods are validated in simulations and experiments with the real quadcopter.

Kurzzusammenfassung

In den letzten Jahrzehnten spielen unbemannte Luftfahrzeuge in Forschung und Industrie in verschiedenen Bereichen eine immer wichtigere Rolle. Die vorliegende Arbeit befasst sich mit der Trajektorienplanung und Hindernisvermeidung eines Quadcopters in Innenumgebungen mit unbekannten Hindernissen.

Der verwendete Quadcopter stellt einen Einplatinencomputer und einen Autopiloten bereit. Der Einplatinencomputer betreibt das Robot Operating System (ROS), welches die Vorgabe der Trajektorie des Quadcopters erlaubt und des Weiteren Messdaten des Quadcopters zur Verfügung stellt. Der Autopilot übernimmt die hardwarenahen Befehle und berechnet basierend auf der gegebenen Trajektorie und den Sensordaten die Ausgangsgrößen für die Motoransteuerung.

Die Trajektorienplanung erfolgt auf Basis der differentiellen Flachheit der Dynamik des Quadcopters. Mit Hilfe des RRT* Algorithmus wird ein kollisionsfreier Pfad vom vorgegebenen Start- zum Zielpunkt im dreidimensionalen Raum ermittelt. Anschließend wird dieser Pfad geglättet und die Stellgrößen über ein quadratisches Programm ermittelt.

Die Tiefenkamera wird verwendet, um neue unbekannte Objekte in der Umgebung zu erfassen. Dadurch wird entschieden, ob der Algorithmus zur Neuplanung aktiviert oder der berechneten Trajektorie weiterhin gefolgt wird.

Abschließend wird eine Validierung der vorliegenden Methoden sowohl in Simulationen als auch in Experimenten mit dem realen Quadcopter durchgeführt.

Contents

List of Figures

List of Tables

1 Introduction

In recent decades, unmanned aerial vehicles (UAVs) have attracted great interest and have undergone extensive development in industry and academia. There are numerous applications for UAVs, ranging from aerial photography [1], aerial manipulation [2], search and rescue observation [3], to structural inspection and in agriculture as well as geography, e.g. [4],[5]. In general, UAVs can operate in high-risk areas that are inaccessible to humans, such as contaminated or flammable environments, without endangering human lives or requiring expensive tools for exploration. In addition, compared to traditional ground-based robots, UAVs are versatile and agile as well as capable of flying at various altitudes. This allows them to navigate inaccessible environments.

There are various configurations of UAVs that have been developed in recent years, see, e.g., [6]. These UAV configurations, which diversify in weight, flexibility, speed, size, and cost, are designed for low-budget systems for the regular consumer to high-end systems for military customers. A detailed classification of UAV configurations can be found in [7]. This work focuses on quadcopters, which belong to the group of vertical takeoff and landing (VTOL) aircrafts. The quadcopter has four motors whose axe of rotation are aligned with the vertical axis of the main frame. Unlike other VTOL vehicles, such as helicopters, quadcopters can hover in one place because two motors rotate clockwise and two rotate counterclockwise to cancel each others momentum out, without additional measures such as a tail-rotor. This results in a more simplified mechanical design and high maneuverability. In indoor environments with several obstacles, a quadcopter is often equipped with four rotor guards to avoid possible collisions with the environment.

UAVs are directly controlled by a human, called UAV's pilot. However, manual control requires a highly skilled pilot and limits the range of operations. Furthermore, the pilot's position must be actively changed if the drone is flown behind obstacles in cluttered environments. Many systems allow flying in first-person perspective (FPV), which requires a stable wireless video connection that can be problematic in certain environments, especially for indoor flights. These constraints are pushing forward the development of autonomous aircraft. Most applications require UAVs to navigate safely and quickly in various unknown environments, detecting and avoiding obstacles as they traverse these areas. This work focuses on the development of a trajectory (re)planning algorithm that helps to navigate the quadcopter autonomously from a takeoff position to a target in an indoor environment with unknown obstacles.

The presented thesis is structured as follows. In Chapter 2, approaches for modeling, path planning, and obstacle detection are briefly reviewed. Chapter 3 provides insight into the hardware and software used in this thesis. The chosen setup allows the quadcopter to be controlled from a ground control station (GCS) and to detect unknown obstacles during the flight. In Chapter 4, the mathematical model is derived. This serves as the basis for trajectory planning in Chapter 5. Thereby, a path from the start position to the target position is computed by the optimal raplidly exploring random tree (RRT*). Then, an optimized smooth trajectory through all the path segments are computed by using quadratic programming (QP). To this end, the proposed replanning algorithm automatically recomputes the trajectory when unknown obstacles are detected that could potentially collide with the drone. Simulation and experiment results in different scenarios are demonstrated in Chapter 6. Finally, Chapter 7 summarizes this work and provides an outlook on further improvements.

2 Literature Review

There has been exciting progress in UAV research in recent times. The large number of applications in research and industry brought a multitude of different solutions, some of which are listed in this chapter that are relevant to this work. First, some contributions to the mathematical modeling of quadrotors are listed in Section 2.1, followed by an overview of different solutions concerning autonomous navigation in Section 2.2, with the core modules “Path and Trajectory Planning” in Section 2.2.1 and “Obstacle Detection” in Section 2.2.2.

2.1 Modeling

In order to realize autonomous robots, complex algorithms are necessary, which usually require knowledge of the mathematical model of the system. There are several models for quadrotors varying in complexity. A common model neglecting friction forces and aerodynamic effects is presented in [8] which is proven to be a differentially flat system. In [9], the model is extended with first order drag effects, which leads to reduced tracking errors for higher speeds or in case of wind disturbances. The work presented in [10] shows that the extended dynamical model of a quadrotor, which considers rotor drag, is also a differentially flat system.

2.2 Autonomous Navigation

Autonomous navigation of a quadcopter in a known or unknown environment requires two main capabilities: First, the quadcopter must be able to plan its movement towards targets. Second, the quadcopter has to locate its position and efficiently avoid possible obstacles during the motion.

For autonomous navigation of quadcopters, there are various methods that can be applied. In [11], the authors utilized a laser range finder for mapping the environment, serving as a basis for a frontier-based waypoint planner in 2D to explore its environment. In [12], a two-step approach for motion planning is proposed, consisting of a path planning utilizing the Dijkstra search algorithm. The path computed in [12] is further processed into a B-spline trajectory, in combination with a stereo camera setup. In [13], an initial path is obtained via RRT*, followed by a minimal jerk trajectory optimization to guarantee a smooth motion of the quadcopter, whereas a laser range finder was used to scan the environment.

2.2.1 Path and Trajectory Planning

In path and trajectory planning, a classical two-step approach, see, e.g., [12–14], is commonly employed. In the first step, a quick planning to the desired target position is performed with different path planning methods without considering the system dynamics, yielding a piecewise linear path, see, e.g., [12], [13]. Secondly, a dynamically feasible trajectory is computed by considering this piecewise linear path using various methods, see, e.g., [15], [14].

In the first step of the two-step approach, various graph search algorithms such as Dijkstra [16] and A* [17] are used to find a feasible path in a discretized occupancy map. On the other hand, sampling-based algorithms such as the probabilistic roadmap (PRM) [18] and the rapidly exploring random tree (RRT) [19] are used to compute a path in 3D or in the state space of the system.

The RRT* algorithm [20], also called optimal RRT algorithm, finds the path from an initial state towards a final state by randomly sampling states and building a planning tree. In this approach, rewiring steps are considered to reconstruct a tree, not only when adding new states, but also considering them as surrogate nodes for existing states in the tree. This helps the RRT* algorithm to provably converge to the optimal solution when the planning time is infinite, see [20]. There are several improvements of RRT* in the literature, such as, e.g., RRT*-SMART [21], which adds heuristics in the sampling process to speed up the convergence rate. In [22], the authors introduced the Informed-RRT* algorithm. Therein, after finding a first solution using the standard RRT* [20], samples of the system state are only allowed in the subspace created by the first solution. This reduction of the search space increases the probability to quickly find a better solution. Nonetheless, sampling-based motion planning is outperformed by a search-and-optimize approach in lower dimensions, since it is impractical to perform optimization for vehicles with nonlinear 12-DOF dynamics in the state space [14].

Once the initial path is obtained, it proved to be practical to assign a polynomial to each segment of the path, due to the differential flatness property of the dynamics of the quadrotor. These polynomials are optimized to yield a smooth trajectory via a quadratic program (QP) or gradient descent method, considering constraints on the endpoints of each segment, see, e.g., [8], [13], [14], [23].

Another class of methods used in motion planning are based on motion primitives [24]. Discrete short-duration inputs are used to discretize the state space of the system into a state lattice with motion primitives forming edges in the graph. Then graph search algorithms, e.g., A*, are used to find a feasible solution trough this graph [25], [24]. A drawback of this approach is the need to discretize both of the workspace and the state space. For example, motion primitives can only be generated for a finite number of start and end velocities. Therefore, the performance of the algorithm is tightly linked to the number of generated motion primitives [25]. Additionally, each motion primitive requires the integration of the system equations, which is unfavorable for vehicles with non linear dynamics.

Recently, Vu et al. [26, 27] introduced a two-stage approach in which an offline trajectory database is created in the first stage and linear constrained quadratic programming (LCQP) is applied in the second (online) stage. In this method, the offline trajectory database plays an important role as it provides reference trajectories for an LCQP and helps to speed up the computing time of this LCQP. Although this two-stage approach can be applied to other applications, see, e.g., [28] and [29], the ability to deal with dynamic obstacles is still a challenge.

2.2.2 Obstacle Detection

Obstacle detection and avoidance has been implemented on various robot platforms with several sensors, such as 2D-Lidar sensors [30], ultrasonic sensors [31], laser range finder [11], [13], stereo cameras [12], and RGB-D cameras [32]. Most of these sensors have in common that they deliver a depth image in form of point clouds. A general survey on point cloud segmentation is given in [33], which is the basis to recognize separate objects. An excellent contribution is described in [34], where a technique for distinguishing dynamic and static obstacles based on the input of one RGB-D camera is described.

3 Setup and Equipment

This chapter gives an overview of the used drone and the firmware architecture. Section 3.1 contains the essential information of the hardware and software of the quadcopter. Then, insights into the software architecture, which allows to control the drone from another PC, are presented. Section 3.2 briefly introduces the simulation environment. Finally, an overview of the simulation and experimental setup is presented in Section 3.3.

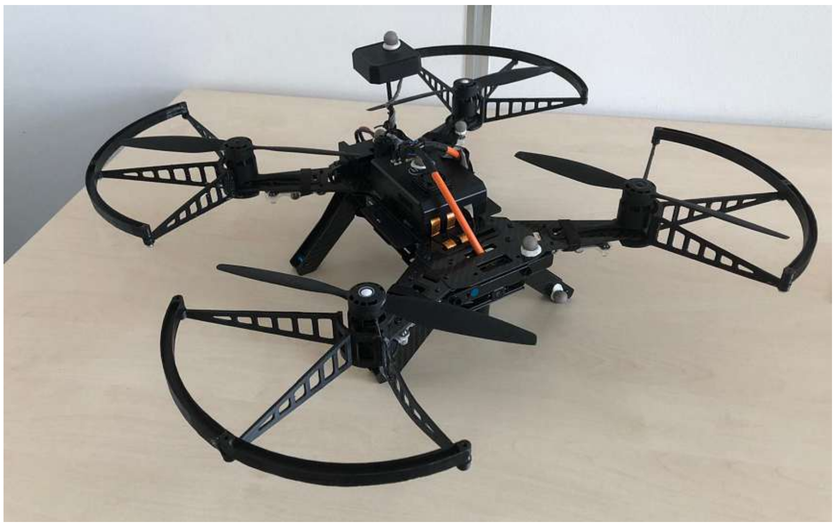


Figure 3.1: Intel Aero ready-to-fly (RTF) used in the thesis [35].

3.1 Intel Aero RTF drone

The Intel Aero ready-to-fly (RTF) drone is used in this thesis, as shown in Figure 3.1. Since indoor space is limited and there are various obstacles of different sizes and shapes, small and agile systems are better suited for maneuvering. On the other hand, sensors, for perception of the environment and the required computations of complex algorithms, e.g., computer vision task, trajectory planning task, require powerful hardware, which is costly. The development platform in this thesis offers a good trade-off between agility and the cost of the hardware.

The Intel Aero RTF consists of two main components, i.e. the Intel Aero compute board and the Intel Aero flight controller board. The communication protocol is processed via a high-speed FPGA interface. Additionally, middleware, i.e. Robot Operating System (ROS), is installed on the compute board and takes care of the high-level tasks, e.g. computer vision task, communications from/to a ground control station. On the other hand, a firmware, i.e. PX4 Autopilot, is installed on the Intel flight controller to handle the low-level tasks, e.g. reading IMU, gyroscopes, barometers, GPS and controlling brushless motors. More details on the components of the Intel Aero RTF drone are presented in the following.

3.1.1 Intel Aero compute board

The Intel Aero compute board contains the Intel Atom x7-Z8750 1.60 GHz processor with integrated graphics processing unit and 4 GB of RAM. An Ubuntu distribution and the open-source Robot Operating System (ROS) 18.04 Melodic are installed in the compute board. In addition, a wifi module, allowing connection to external devices, is integrated in the compute board. This offers the possibility to configure and control the drone via a remote PC, known as ground control station (GCS). Ideally, the obstacle avoidance module and the trajectory optimization can be processed onboard if the computing power of the compute board is sufficient. However, since the Intel Atom X7-Z8750 has limited computing power, only the data fusion of the extended Kalman filter (EKF) [36] is processed onboard. The obstacle avoidance module and the proposed trajectory (re)planning are computed by a companion PC, which is connected to the Intel Aero compute board via wifi.

3.1.2 Intel Aero flight controller board

The Intel Aero flight controller comprises the STM32F427VI that executes the PX4 Autopilot program [37], which is an open source flight controller for several kinds of vehicles from drones to ground vehicles. In addition, this flight controller connects with the compute board via the MAX 10 FPGA.

The PX4 autopilot firmware has a modular structure that consists of parts such as internal communication, integration of sensor data, and motor control. A very relevant part for further considerations is the controller architecture of the PX4 autopilot, which is shown in Figure 3.2. The flight controller is a cascaded controller, consisting of P and PID controllers for tracking the position, velocity, orientation and angular rate. This cascaded controller takes into account desired setpoints and state estimations, which are computed by the EKF that fuses data from sensors, e.g., IMUs, barometers, accelerometers, GPS and RGB-D camera. The outputs of the controller are PWM signals for the motors via electronic speed controllers (ESCs).

The Intel Aero flight controller communicates with the Intel Aero compute board via a fast serial link with the MAVLINK protocol [38]. This protocol is a very lightweight messaging protocol for communicating with drones and between onboard drone components [39]. Data streams are handled as topics in a publish-subscribe pattern, configuration sub-protocols are implemented point-to-point. Although a variety of standard messages are available, MAVLINK enables developers to add custom message types via XML files. The open-source ROS node, named MAVROS, running on the compute board, represents the missing connection between MAVLINK and ROS, handling all message conversion in both directions for the operator.

The PX4 Autopilot provides different flight modes. These modes are divided into two main categories: manual modes, i.e., the user has control of the drone via a remote control (RC), and autonomous modes, i.e., the autopilot controls the drone without user intervention. In this thesis, the autonomous mode, named "offboard mode", is employed, which allows a ground control station to control the drone. More specifically, several ROS topics are implemented to generate the desired trajectory for position, velocity, acceleration, attitude, and attitude rate, which are then fed into the cascaded controller in the flight controller. In our specific application, the trajectory (re)planning for the obstacle-free and dynamically feasibile trajectory is computed in 6 Hz from a ground station PC. Then, this planned trajectory is sent to the compute board via wifi and is executed by the drone flight controller.

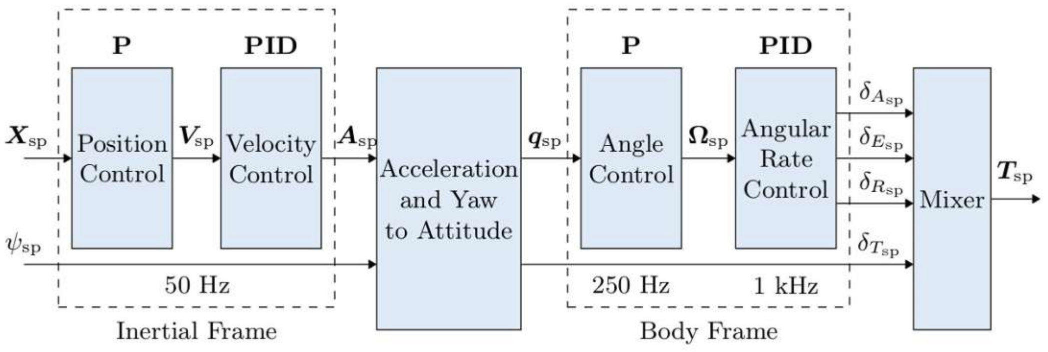

P
PID
P
PID
$\boldsymbol{X}_{\mathrm{sp}}$
Position Control
$\boldsymbol{V}_{\mathrm{sp}}$
Velocity Control
$\boldsymbol{A}_{\mathrm{sp}}$
Acceleration and Yaw to Attitude
$\boldsymbol{q}_{\mathrm{sp}}$
Angle Control
$\boldsymbol{\Omega}_{\mathrm{sp}}$
Angular Rate Control
$\delta_{A_{\mathrm{sp}}}$
$\delta_{E_{\mathrm{sp}}}$
$\delta_{R_{\mathrm{sp}}}$
$\delta_{T_{\mathrm{sp}}}$
Mixer
$\boldsymbol{T}_{\mathrm{sp}}$
ψ_{sp}
50 Hz
250 Hz
1 kHz
Inertial Frame
Body Frame

Figure 3.2: Cascaded Multicopter Control Architecture of the PX4 Autopilot [40].

3.1.3 Sensors

The Intel Aero RTF comes with several sensors to perceive its environment, which are directly connected to either the compute board or the flight controller. The system used in this work has an accelerometer and gyroscope combined in an inertial measurement unit (IMU), a magnetometer, an altitude sensor, a GPS module with a compass, a downward-facing monochrome camera, and a forward-facing RGB-D camera.

The latter camera is an Intel RealSense R200 camera with a full HD RGB color image and a VGA-resolved depth image produced by a stereo array of infrared cameras. To increase the accuracy of the depth estimation, an infrared projector is also used. This camera is connected to the Intel Aero compute board via USB and transmits data at 30 Hz at full resolution or at 60 Hz at reduced resolution. Note that in this work, only reduced resolution depth image information is employed to implement the obstacle avoidance function.

The PX4 Autopilot uses an EKF to estimate the system state from different sensors. In a cluttered environment, some sensors, e.g., GPS and magnetometers, may be severely disturbed or non-functional at all, which makes a solid position estimation impossible. PX4 warns the operator in case of poorly calibrated sensors and prevents the drone from taking off. To overcome this problem, a reliable source of the droneÂ´s position can be obtained with the help of a motion capture system. In our setup, the OptiTrack motion capture system is used [41]. This system emits infrared light and receives the reflections of specific markers attached to the object to be tracked with multiple cameras. It enables a sampling rate of up to 250 Hz with the accuracy of less than 1 mm and 1°. The estimation of the OptiTrack system is fed into the EKF module of the PX4 autopilot program of the drone.

3.2 Simulation environment

Simulation is very useful to test specific modules separately during the development process. There are many reasons why simulation or digital twins become more and more popular. First and foremost, the development and extensive tests can be carried out without a real system, which above all drastically increases the safety for people and machines especially in dangerous applications. The case of failure on a real system during a test run may lead to a considerably longer development time and exploding costs. However, in simulation only a simple restart is necessary. Furthermore, simulation environments yield debugging information without the need of additional sensors or hardware.

In this work, the open-source simulator Gazebo is set up on the ground control station. This physics engine based simulator offers an excellent connection to ROS and enables the software-in-the-loop (SITL) simulation of the PX4 flight stack, which allows the PX4 firmware used in the simulation to be transferred to the real flight controller. In addition, a library model of the RGB-D camera, IMU, and other sensors can be simulated in the Gazebo simulation environment. Overall, the Gazebo simulator represents an excellent development tool for this application and enables almost direct sim-to-real transfer.

3.3 Overview of the simulation and experiment setup

This section gives an overview of the simulation and experimental setup. The overall system architecture is illustrated in Figure 3.3.

The compute board and the flight controller board are illustrated in the blue and grey block, respectively. These two blocks communicate with each other using the MAVLINK protocol. The trajectory planning and obstacle avoidance module is implemented in a ground station PC. This module connects with the ROS node in the compute board via WLAN.

The overall processes of simulations as well as experiments are listed in the following paragraph. First, an obstacle-free optimal trajectory from the given starting point to a target point is computed by the proposed trajectory planning algorithm in the ground control station. Then, this desired trajectory is sent to the ROS node in the compute board via wifi. The point cloud data, captured by the RGB-D camera on the compute board, is updated to the obstacle detection block in the ground station PC. Thereby, the proposed algorithm can detect if there is a new obstacle and proceeds with the trajectory replanning scheme to generate a new obstacle-free trajectory.

In experiments, the OptiTrack provides highly precise pose information of the drone that is fused with the EKF on the compute board. This yields a very robust real-time positioning of the drone by fusing data of other sensors in the GPS-denied environment. Finally, the cascaded controller is able to generate the desired PWM signals for the ESCs of the motors.

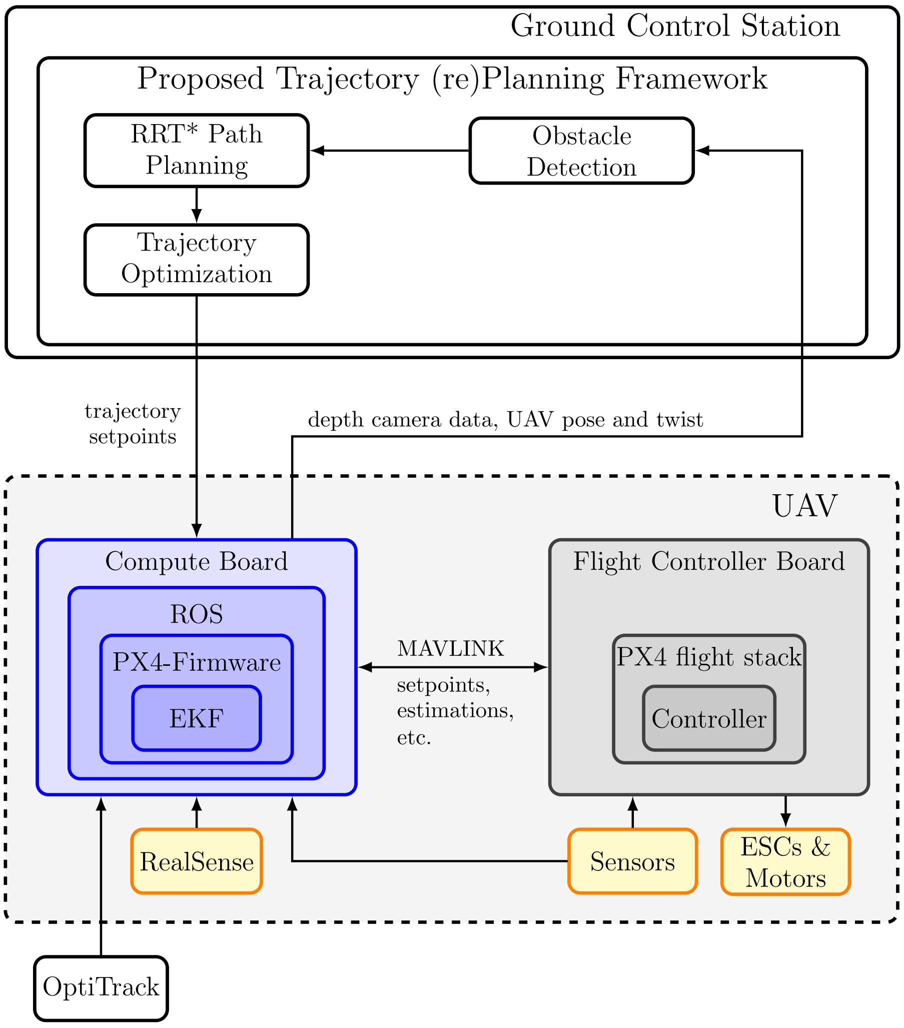


Figure 3.3: Overview of the simulation and experiment setup.

4 Mathematical Modeling

This section presents the mathematical model of the quadrotor, which serves as a basis for the subsequent sections. First, coordinate systems, used to derive the mathematical model of the quadrotor, are introduced. Then, the system dynamics is derived using Euler's equations. Finally, the differential flatness property of the quadrotor is exploited, see, e.g., [15], where all the system states and inputs can be parameterized by flat outputs and their time derivatives. These flat outputs are then used in the proposed trajectory replanning algorithm.

4.1 Coordinate Frames

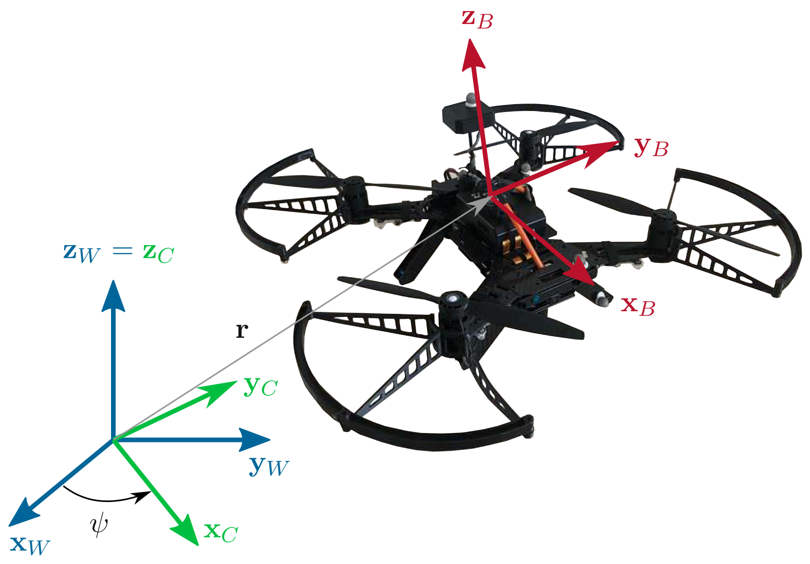


Figure 4.1: The world coordinate, intermediate coordinate, and the body coordinate frame are denoted as $\mathcal{I}_W$, $\mathcal{I}_C$, and $\mathcal{I}_B$, respectively.

Three coordinate frames that are used to derive the mathematical modeling of the drone are defined in Figure 4.1. The fixed world coordinate frame $\mathcal{I}_W$ is defined by three unit vectors x_W, y_W, and z_W. To track the movement of the quadrotor, the body coordinate frame $\mathcal{I}_B$ of three unit vectors x_B, y_B, and z_B, located at the center of mass of the drone, is illustrated in Figure 4.1. Note that the unit vector x_B and z_B are pointing toward the flight direction and upward. The position of the center of mass in the world frame is given by vector $\mathbf{r} = [x, y, z]^{\mathrm{T}}$. The intermediate coordinate frame $\mathcal{I}_C$ is obtained by rotating the world frame around the vector z_W with the angle ψ. This intermediate coordinate frame is used for deriving the differential flatness property of the drone in Section 4.3. *Z*-*X*-*Y* Euler-angle convention is used to describe the rotation of the quadrotor in the world coordinate system $\mathcal{I}_W$. Thereby, the rotation matrix from $\mathcal{I}_B$ to $\mathcal{I}_W$ is given by

$$ {}^{W}\mathbf{R}_B = \mathbf{R}_{z,\psi}\mathbf{R}_{x,\phi}\mathbf{R}_{y,\theta} = \begin{bmatrix} c\psi c\theta - s\psi s\phi s\theta & -s\psi c\phi & c\psi s\theta + s\psi s\phi c\theta \\ s\psi c\theta + c\psi s\phi s\theta & c\psi c\phi & s\psi s\theta - c\psi s\phi c\theta \\ -c\phi s\theta & s\phi & c\phi c\theta \end{bmatrix}, \tag{4.1} $$

where $\mathbf{R}_{i,\alpha}, i \in \{x, y, z\}, \alpha \in \{\psi, \phi, \theta\}$ denotes the rotation around the axis i with the angle α. Note that $s\alpha$ and $c\alpha$ are the compact form of $sin(\alpha)$ and $cos(\alpha)$. Using the quadcopter center of mass position $\mathbf{r}$ in the world frame, a vector ${}^{B}\mathbf{p}$ in the body-fixed coordinate system can be transformed to the world frame by

$$ {}^{W}\mathbf{p} = {}^{W}\mathbf{R}_B {}^{B}\mathbf{p} + \mathbf{r}. \tag{4.2} $$

The angular velocity of the body frame in the world frame ${}^{W}\boldsymbol{\omega}_B$ yields

$$ {}^{W}\boldsymbol{\omega}_B = \begin{bmatrix} {}^{W}\mathbf{x}_B & {}^{W}\mathbf{y}_B & {}^{W}\mathbf{z}_B \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{4.3} $$

where p, q and r in (4.3) are computed from

$$ \begin{bmatrix} p \\ q \\ r \end{bmatrix} = (\mathbf{R}_{x,\phi}\mathbf{R}_{y,\theta})^{\mathrm{T}} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + \mathbf{R}_{y,\theta}^{\mathrm{T}} \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} = \begin{bmatrix} c\theta & 0 & -c\phi s\theta \\ 0 & 1 & s\phi \\ s\theta & 0 & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = {}^{B}\mathbf{W} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}. \tag{4.4} $$

The latter equation describes the transformation matrix ${}^{B}\mathbf{W}$ from the angular velocities of the Euler angles in the world frame to the angular velocity of the body frame. Since (4.4) is expressed in the body frame and by applying the Z-X-Y Euler convention, the transformation matrix ${}^{B}\mathbf{W}$ can be written as

$$ {}^{B}\mathbf{W} = \begin{bmatrix} c\theta & 0 & -c\phi s\theta \\ 0 & 1 & s\phi \\ s\theta & 0 & c\phi c\theta \end{bmatrix} = \begin{bmatrix} {}^{B}\mathbf{x}_C & {}^{B}\mathbf{y}_B & {}^{B}\mathbf{z}_W \end{bmatrix}. \tag{4.5} $$

Additionally, the transformation of ${}^{B}\mathbf{W}$ to the world frame yields

$$ {}^{W}\mathbf{W} = {}^{W}\mathbf{R}_B {}^{B}\mathbf{W} = \begin{bmatrix} c\psi & -s\psi c\phi & 0 \\ s\psi & c\psi c\phi & 0 \\ 0 & s\phi & 1 \end{bmatrix} = \begin{bmatrix} {}^{W}\mathbf{x}_C & {}^{W}\mathbf{y}_B & {}^{W}\mathbf{z}_W \end{bmatrix}. \tag{4.6} $$

The angular velocity of the intermediate frame $\mathcal{I}_C$ in the world frame ${}^W\boldsymbol{\omega}_C$ reads as

$$
{}^W\boldsymbol{\omega}_C = \begin{bmatrix} {}^W\mathbf{x}_C & {}^W\mathbf{y}_C & {}^W\mathbf{z}_C \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = \dot{\psi}\mathbf{z}_W. \tag{4.7}
$$

Note that $\mathbf{z}_C = \mathbf{z}_W$, since the intermediate frame C is created by only rotating with the yaw angle ψ around the $\mathbf{z}_W$ axis.

4.2 System Dynamics

In this section, the system dynamics of the quadrotor is presented in detail. In Figure 4.2,

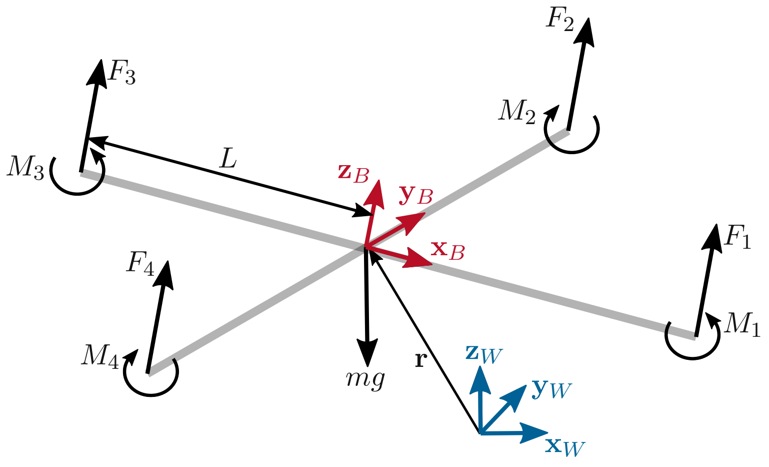


Figure 4.2: Forces and moments acting on quadcopter frame. The body coordinate frame $\mathcal{I}_B$ which is aligned with the rotor axes of the quadrotor, other than in Figure 4.1, is only used in this section for easier derivation.

a simplified quadcopter model is illustrated. Therein, the world coordinate frame $\mathcal{I}_W$ and the body frame $\mathcal{I}_B$ are aligned along the rotor axes. F_i and M_i denote the force and moment exerted by the four rotors of the quadcopter. The position vector $\mathbf{r}$ denotes the center of mass of the UAV in the world frame. Moreover, L is the distance from the center of mass to the axis of rotation of each rotor and ω_i is the angular rate of the i-th motor, which produces a force F_i in z_B direction. A moment M_i is in the opposite direction of the rotation of its blades. F_i and M_i are computed in the form

$$
F_i = k_F \omega_i^2, \quad M_i = k_M \omega_i^2, i \in \{1, ..., 4\}, \tag{4.8}
$$

with the constant parameters $k_F, k_M > 0$.

In Figure 4.2, the rotor 1 and 3 rotate in the opposite of the z_B-direction while the motor 2 and 4 rotate in z_B-direction. Thus, the moment M_1 and M_3 act in z_B direction whereas M_2 and M_4 act in the opposite direction. Note that this rotation setting is necessary for the quadcopter to cancel out its yaw momentum while hovering. Since the gravity is in z_W-direction, by adding up all forces of the rotors in the system, the acceleration of the center of mass is computed as

$$m\ddot{\mathbf{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + {}^W\mathbf{R}_B \begin{bmatrix} 0 \\ 0 \\ \sum F_i \end{bmatrix}. \tag{4.9}$$

With the known moment of inertia matrix $\mathbf{I}$ of the quadcopter with respect to the body frame $\mathcal{I}_B$, the angular acceleration of the quadcopter in the body frame $\mathcal{I}_B$ is derived by using the Euler equations in the form

$$\mathbf{I}\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \mathbf{I}\begin{bmatrix} p \\ q \\ r \end{bmatrix}. \tag{4.10}$$

As the motor dynamics are fast compared to the rigid-body dynamics and aerodynamics, it is assumed that the desired rotor speeds ω_i can be reached instantaneously on demand. Therefore, the system inputs u_1 and $\mathbf{u}_2$ are computed as

$$u_1 = \sum F_i, \qquad \mathbf{u}_2 = \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix}, \tag{4.11}$$

where u_1 and $\mathbf{u}_2 = [u_2, u_3, u_4]^{\mathrm{T}}$ are the net thrust and moment vector in the body coordinate frame, respectively.

Combining (4.9), (4.10) and (4.11), the quadcopter system dynamics in state-space form reads as

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= -g\mathbf{z}_W + \frac{u_1}{m}{}^W\mathbf{z}_B \\ {}^W\dot{\mathbf{R}}_B &= {}^W\mathbf{R}_B\boldsymbol{\omega}_B \\ \dot{\boldsymbol{\omega}}_B &= \mathbf{I}^{-1}\left[-\boldsymbol{\omega}_B \times \mathbf{I}\boldsymbol{\omega}_B + \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix}\right], \end{aligned} \tag{4.12}$$

where $\mathbf{v}$ is the linear velocity of the quadcopter center of mass. Note that the system state of (4.12) can be combined in the form

$$\mathbf{x} = \left[x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, p, q, r\right]^{\mathrm{T}}. \tag{4.13}$$

Combining (4.9), (4.10), (4.12), and (4.13), the system dynamics of the quadcopter are expressed in the form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \quad \text{with } \mathbf{u}^{\mathrm{T}} = \left[u_1, \mathbf{u}_2^{\mathrm{T}}\right]. \tag{4.14}$$

4.3 Differential Flatness

This section presents the differential flatness property of the quadcopter in a compact form. For more details, the reader is referred to [8], [10]. The differential flatness property allows that the state $\mathbf{x}$ and input $\mathbf{u} = [u_1, \mathbf{u}_2^{\mathrm{T}}]^{\mathrm{T}}$ of (4.14) can be parameterized by a flat output and its time derivatives. This simplifies to compute the control inputs of the quadcopter for a given trajectory of the flat outputs. For easier readability, all vectors in this section are considered in the world frame $\mathcal{I}_W$ without additional notation. The flat output is chosen to be the quadcopter center of mass position $\mathbf{r} = [x, y, z]^{\mathrm{T}}$ and the yaw angle ψ (heading), combined in the vector

$$\boldsymbol{\sigma} = \left[x, y, z, \psi\right]^{\mathrm{T}}. \tag{4.15}$$

In the following, the smooth map

$$(\mathbf{x}, \mathbf{u}) = \Phi(\sigma, \dot{\sigma}, \ddot{\sigma}, \dddot{\sigma}, \ddddot{\sigma}) \tag{4.16}$$

is introduced. The position, velocity, and acceleration of the system are simply defined by the first three terms of $\boldsymbol{\sigma}$, $\dot{\boldsymbol{\sigma}}$ and $\ddot{\boldsymbol{\sigma}}$. Considering (4.9) and the desired accelerations of the flat output in the form

$$m\begin{bmatrix} \ddot{\sigma}_1 \\ \ddot{\sigma}_2 \\ \ddot{\sigma}_3 + g \end{bmatrix} = m\mathbf{t} = u_1 \mathbf{z}_B, \quad \mathbf{t} = \left[\ddot{\sigma}_1, \ddot{\sigma}_2, \ddot{\sigma}_3 + g\right]^{\mathrm{T}}. \tag{4.17}$$

The z-axis of the body frame $\mathbf{z}_B$ and the net body force u_1 are defined as

$$\mathbf{z}_B = \frac{\mathbf{t}}{\|\mathbf{t}\|} \qquad u_1 = m\|\mathbf{t}\|. \tag{4.18}$$

Note that the input u_1 is defined by the second derivatives of the position of the quadcopter. Since the yaw angle $\sigma_4 = \psi$, the x-axis of the intermediate frame $\mathbf{x}_C$ in Figure 4.1 can be written as

$$\mathbf{x}_C = \begin{bmatrix} \cos\sigma_4 & \sin\sigma_4 & 0 \end{bmatrix}^{\mathrm{T}}. \tag{4.19}$$

Then, $\mathbf{x}_B$ and $\mathbf{y}_B$ can be derived by utilizing the local Z-X-Y Euler angle parameterization

$$\mathbf{y}_B = \frac{\mathbf{z}_B \times \mathbf{x}_C}{\|\mathbf{z}_B \times \mathbf{x}_C\|}, \quad \mathbf{x}_B = \mathbf{y}_B \times \mathbf{z}_B. \tag{4.20}$$

This equation holds true, as long as the singularity where $\mathbf{x}_C$ and $\mathbf{z}_B$ are parallel is not encountered. The rotation matrix from body frame to world frame is defined as

$$^W\mathbf{R}_B = \begin{bmatrix}\mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B\end{bmatrix}. \tag{4.21}$$

Therefore, with (4.21) and the flat output σ and its derivatives, the Euler angles ϕ, θ and ψ can be determined. By taking the first derivative of (4.9) in the form

$$m\dot{\mathbf{a}} = \dot{u}_1\mathbf{z}_B + \boldsymbol{\omega}_B \times u_1\mathbf{z}_B, \quad \dot{\mathbf{a}} = \ddot{\mathbf{v}} = \dddot{\mathbf{r}} \tag{4.22}$$

and projecting it along $\mathbf{z}_B$, the derivative of the input u_1 yields

$$\dot{u}_1 = \mathbf{z}_B \cdot m\dot{\mathbf{a}} \tag{4.23}$$

Substituting (4.23) in (4.22), the vector $\mathbf{h}_\omega$, which is the projection of $\frac{m}{a}\dot{\mathbf{a}}$ onto the $x_B - y_B$ plane, reads as

$$\mathbf{h}_\omega = \boldsymbol{\omega}_B \times \mathbf{z}_B = \frac{m}{u_1}(\dot{\mathbf{a}} - (\mathbf{z}_B \cdot \dot{\mathbf{a}})\mathbf{z}_B). \tag{4.24}$$

Considering (4.3) and (4.24), the body frame components p and q of the angular velocity are computed in the form

$$p = -\mathbf{h}_\omega \cdot \mathbf{y}_B, \quad q = \mathbf{h}_\omega \cdot \mathbf{x}_B. \tag{4.25}$$

To find the component r of the angular velocity, the mapping between the derivatives of the Euler angles and the angular velocity in the body frame, see (4.4), (4.5) and (4.6),

$$\boldsymbol{\omega}_B = p\mathbf{x}_B + q\mathbf{y}_B + r\mathbf{z}_B = \begin{bmatrix}\mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B\end{bmatrix}\begin{bmatrix}p\\q\\r\end{bmatrix} = \begin{bmatrix}\mathbf{x}_C & \mathbf{y}_B & \mathbf{z}_W\end{bmatrix}\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix}, \tag{4.26}$$

is taken into account. Using (4.25) and the given $\dot{\psi}$, the $\mathbf{z}_B$ component r of the angular velocity $\boldsymbol{\omega}_B$ can be computed from (4.26). Therefore, the full state $\mathbf{x}$ of the system can be determined by the flat outputs and their derivatives.

Taking the second derivative of (4.9), we have

$$\begin{aligned}
m\frac{\mathrm{d}\dot{\mathbf{a}}}{\mathrm{d}t} &= \frac{\mathrm{d}\boldsymbol{\omega}_B}{\mathrm{d}t} \times u_1\mathbf{z}_B + \boldsymbol{\omega}_B \times \frac{\mathrm{d}(\mathbf{z}_B u_1)}{\mathrm{d}t} + \frac{\mathrm{d}\mathbf{z}_B}{\mathrm{d}t}\dot{u}_1 + \ddot{u}_1\mathbf{z}_B \\
m\ddot{\mathbf{a}} &= \boldsymbol{\alpha}_B \times u_1\mathbf{z}_B + \boldsymbol{\omega}_B \times \left(u_1\frac{\mathrm{d}\mathbf{z}_B}{\mathrm{d}t} + \dot{u}_1\mathbf{z}_B\right) + \boldsymbol{\omega}_B \times \dot{u}_1\mathbf{z}_B + \ddot{u}_1\mathbf{z}_B \\
m\ddot{\mathbf{a}} &= \boldsymbol{\alpha}_B \times u_1\mathbf{z}_B + \boldsymbol{\omega}_B \times (\boldsymbol{\omega}_B \times u_1\mathbf{z}_B) + 2\boldsymbol{\omega}_B \times \dot{u}_1\mathbf{z}_B + \ddot{u}_1\mathbf{z}_B,
\end{aligned} \tag{4.27}$$

with $\boldsymbol{\alpha}_B$ denoting the derivative of the angular velocity $\boldsymbol{\omega}_B$ of the body frame. Note that the time derivative $u_1 \frac{\mathrm{d}\mathbf{z}_B}{\mathrm{d}t} = \boldsymbol{\omega}_B \times u_1 \mathbf{z}_B$ in the second line of (4.27) is orthogonal to $\boldsymbol{\omega}_B$ and $\mathbf{z}_B$. Thereby, projection of (4.27) along the vector $\mathbf{z}_B$ results in

$$\ddot{u}_1 = \mathbf{z}_B \cdot m\ddot{\mathbf{a}} - \mathbf{z}_B \cdot (\boldsymbol{\omega}_B \times (\boldsymbol{\omega}_B \times u_1 \mathbf{z}_B)). \tag{4.28}$$

The components of the angular acceleration along $\mathbf{x}_B$ and $\mathbf{y}_B$ are derived by computing

$$\mathbf{h}_\alpha = \boldsymbol{\alpha}_B \times \mathbf{z}_B, \tag{4.29}$$

leading to

$$\dot{p} = -\mathbf{h}_\alpha \cdot \mathbf{y}_B, \quad \dot{q} = \mathbf{h}_\alpha \cdot \mathbf{x}_B. \tag{4.30}$$

Calculating the first derivative of (4.26) considering the angular velocities of the intermediate frame and the body frame results in

$$\begin{aligned}
\dot{p}\mathbf{x}_B + p\frac{\mathrm{d}\mathbf{x}_B}{\mathrm{d}t} + \dot{q}\mathbf{y}_B + q\frac{\mathrm{d}\mathbf{y}_B}{\mathrm{d}t} + \dot{r}\mathbf{z}_B + r\frac{\mathrm{d}\mathbf{z}_B}{\mathrm{d}t} &= \ddot{\phi}\mathbf{x}_C + \dot{\phi}\frac{\mathrm{d}\mathbf{x}_C}{\mathrm{d}t} + \ddot{\theta}\mathbf{y}_B + \dot{\theta}\frac{\mathrm{d}\mathbf{y}_B}{\mathrm{d}t} + \ddot{\psi}\mathbf{z}_W \\
\begin{bmatrix}\mathbf{x}_B \\ \mathbf{y}_B \\ \mathbf{z}_B\end{bmatrix}^{\mathrm{T}} \begin{bmatrix}\dot{p} \\ \dot{q} \\ \dot{r}\end{bmatrix} + \boldsymbol{\omega}_B \times \boldsymbol{\omega}_B &= \boldsymbol{\omega}_C \times \dot{\phi}\mathbf{x}_C + \boldsymbol{\omega}_B \times \dot{\theta}\mathbf{y}_B + \begin{bmatrix}\mathbf{x}_C \\ \mathbf{y}_B \\ \mathbf{z}_W\end{bmatrix}^{\mathrm{T}} \begin{bmatrix}\ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi}\end{bmatrix} \\
\begin{bmatrix}\mathbf{x}_B \\ \mathbf{y}_B \\ \mathbf{z}_B\end{bmatrix}^{\mathrm{T}} \begin{bmatrix}\dot{p} \\ \dot{q} \\ \dot{r}\end{bmatrix} &= \boldsymbol{\omega}_C \times \dot{\phi}\mathbf{x}_C + \boldsymbol{\omega}_B \times \dot{\theta}\mathbf{y}_B + \begin{bmatrix}\mathbf{x}_C \\ \mathbf{y}_B \\ \mathbf{z}_W\end{bmatrix}^{\mathrm{T}} \begin{bmatrix}\ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi}\end{bmatrix}.
\end{aligned} \tag{4.31}$$

The $\mathbf{z}_B$ component of the angular acceleration is obtained from the third scalar equation of (4.31) with a given value of $\ddot{\psi}$.

Considering (4.17) - (4.31), it is obvious that the angular velocity $[p, q, r]^{\mathrm{T}}$ and angular acceleration $[\dot{p}, \dot{q}, \dot{r}]^{\mathrm{T}}$ are functions of the flat outputs and their derivatives. Since (4.10) can be utilized to compute the net body moments

$$\mathbf{u}_2 = \begin{bmatrix}u_2 \\ u_3 \\ u_4\end{bmatrix} = \mathbf{I}\begin{bmatrix}\dot{p} \\ \dot{q} \\ \dot{r}\end{bmatrix} + \begin{bmatrix}p \\ q \\ r\end{bmatrix} \times \mathbf{I}\begin{bmatrix}p \\ q \\ r\end{bmatrix}, \tag{4.32}$$

$\mathbf{u}_2$ is a function of the flat outputs σ and their derivatives.

5 Trajectory (re)planning

In this chapter, a trajectory planning framework is proposed for the quadcopter in environments with unknown obstacles. Furthermore, this proposed framework is capable of replanning in case the environment has changed, e.g., new obstacles appear. This is also the main focus of this work. Various scenarios with different settings are investigated to verify the effectiveness of the proposed algorithm.

The remainder of this chapter is structured as follows. In Section 5.1, an overview of the simulation and experimental setup is briefly introduced. Then, the optimal Rapidly Exploring Random Tree algorithm (RRT*) and the Line of Sight (LOS) algorithm are presented in Section 5.2 and 5.3, respectively. Using the obstacle-free path from RRT* and LOS, the trajectory generation for the quadcopter is introduced in Section 5.4. Finally, the (re)planning capabality of the proposed algorithm is presented in Section 5.5.

5.1 Overview of simulation and experimental setup

Without loss of generality, the UAV can be treated as a point located at the quadcopter's center of mass. In this work, obstacles, modeled as cuboids, are inflated in each dimension for safety purposes. An example of the flight environment is illustrated in Figure 5.1, including two inflated obstacles. The admissible flight range is defined as the set $\mathcal{X}$. For the sake of simplicity and safety, all obstacles in the set $\mathcal{O}$ are considered as convex obstacles, i.e., cuboids. By excluding obstacles, the obstacle-free space is denoted as $\mathcal{X}_{free} = \mathcal{X} \setminus \mathcal{O}$.

The overview of the proposed trajectory (re)planning framework consisting of the offline and the online trajectory planning block is depicted in Fig. 5.2. The offline block on the left-hand side of Fig. 5.2 is explained in the following. First, the RRT* algorithm [42] is utilized to compute a piecewise linear collision free path from an initial location to a target location. Since only the path of the quadcopter is computed by the RRT*, constraints on the system dynamics (4.14) are neglected in this path planning step. Second, to reduce the complexity of the path computed by RRT*, the Line of Sight (LOS) algorithm is implemented to prune the trajectory tree. This helps to reduce the number of optimization variables and computation time. The Gilbert-Johnson-Keerthi (GJK) algorithm [43] is applied to compute Euclidean distances between trajectory points and the obstacles. Thereby, the proposed algorithm can check whether the points on the current path are collision free or not. Then, further actions such as sampling a new 3D point in the RRT* algorithm and activating the (re)planning can be taken into account.

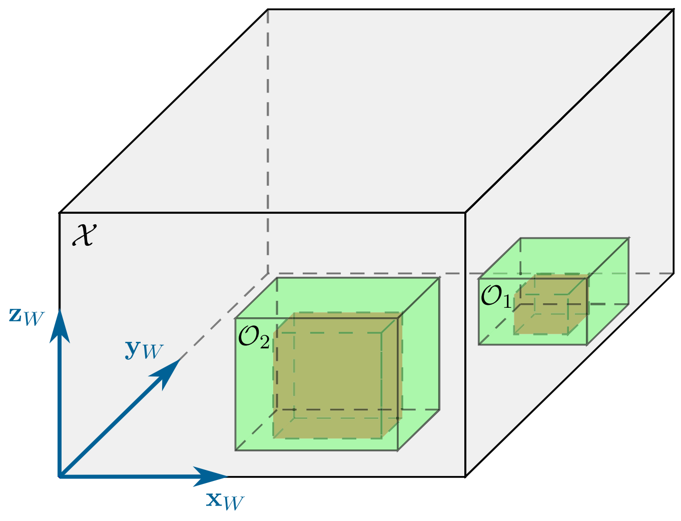


Figure 5.1: The visualization of the environment with the admissible flight space $\mathcal{X}$ in gray area and examples of possible obstacles $\mathcal{O}$. The obstacles (in red color) are inflated (in green color) due to safety reasons.

After computing the collision free path consisting set of path points, also called waypoints, this path is divided into a sequence of polynomial segments between waypoints which will be optimized into smooth trajectories by utilizing the differential flatness property of the quadrotor, see, e.g., [15] and [14].

Once having the first offline trajectory from the offline block, the online block on the right side of Fig. 5.2 is subsequently executed. During the flight, the RGB-D camera is employed to check for unknown obstacles in the environment. Once new obstacles are detected, a feasibility check of the pre-computed trajectory is initiated, leading to the rerouting of the tree structure from RRT* and regeneration of the trajectory. Note that the Online – ROS Loop block is iteratively executed as a ROS node to detect and avoid obstacles during flight. Matlab Timer Callback is utilized to send the actual trajectory via ROS to the quadcopter at 6 Hz.

5.2 RRT/RRT* - Rapidly Exploring Random Tree

The Rapidly Exploring Random Tree (RRT) is the most popular path planning algorithm which was originally introduced by Lavalle et al. [19]. Later, a variant of the RRT, i.e., RRT*, is developed by Karaman et al. [20]. This path planning algorithm guarantees to compute asymptotically optimal solutions. In the followings, a brief introduction of the RRT* is presented.

A Rapidly Exploring Random Tree $\mathcal{G}$ consists of a set of nodes or vertices $\mathcal{V}$ and a set

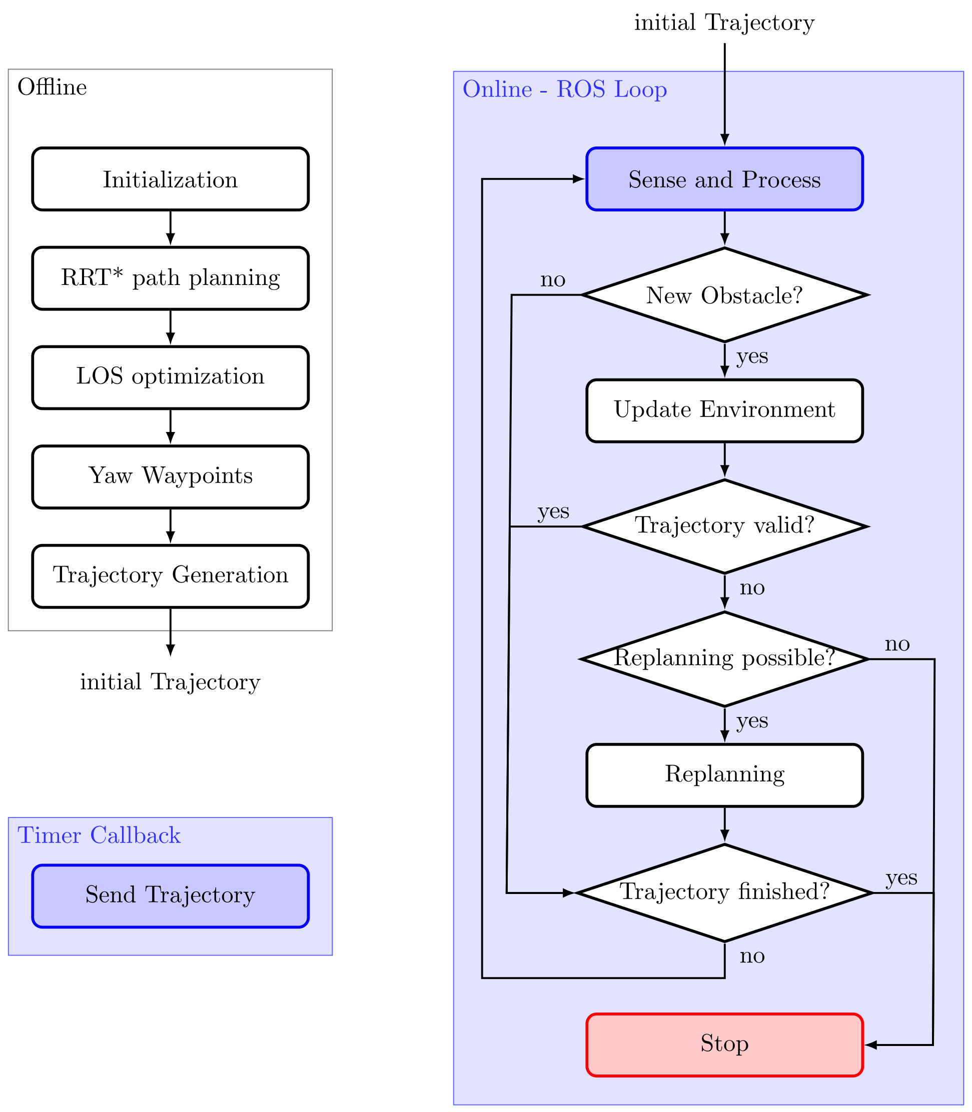


Figure 5.2: Flow chart of the overall trajectory planning process. The offline process is run prior to the flight. Processes in light blue background are executed online during the flight. All tasks are executed on the GroundControl station in MATLAB and blue blocks "Send Trajectory" and "Sense and Process" indicate a data exchange with MAVROS.

of edges $\mathcal{E}$. The pseudocode of the RRT* algorithm is illustrated in Alg. 1. Note that some basic functions utilized in Alg. 1 are introduced below.

- Function SampleFree($\mathbb{X}_{free}$) returns a random point from the obstacle-free space $\mathbb{X}_{free}$.
- Function AddNode : $(\mathbf{x}, \mathcal{G}, \mathcal{O}, \epsilon, \rho) \rightarrow \mathcal{G}$ adds the vertex $\mathbf{x}$ to the tree $\mathcal{G}$
- Function Nearest :$(\mathcal{G}, \mathbf{x}) \rightarrow \mathbf{v} \in \mathcal{V}$ provides the vertex in $\mathcal{V}$ with the smallest Euclidean distance to $\mathbf{x}$.
- The function Near :$(\mathcal{G}, \mathbf{x}, \rho) \rightarrow \mathcal{V}' \subseteq \mathcal{V}$, with the positive real number $\rho \in \mathbb{R}_{\geq 0}$, returns the vertices that are contained in a ball of radius ρ with its center at $\mathbf{x}$.
- Steering function Steer :$(\mathbf{x}, \mathbf{y}, \epsilon) \rightarrow \mathbf{z}$, with $\mathbf{x} \in \mathbb{X}$, yields a point $\mathbf{z} \in \mathbb{X}$ that linearly extends the tree from vertex $\mathbf{x}$ towards $\mathbf{y}$ in the three dimensional space $\mathbb{R}^3$ with the user-defined parameter ϵ. If $\mathbf{y}$ lies within the ϵ-sphere of $\mathbf{x}$, the length of the edge is not altered, and $\mathbf{z} = \mathbf{y}$.
- The boolean function CollisionFree :$(\mathbf{x}, \mathbf{y}, \mathcal{O})$ returns true if the line segment between $\mathbf{x}$ and $\mathbf{y}$ lies in $\mathbb{X}_{free}$.
- Considering two nodes $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$, Line$(\mathbf{x}, \mathbf{y}) : [0, s] \rightarrow \mathbb{X}$ denotes the straight-line path from $\mathbf{x}$ to $\mathbf{y}$.
- The function Parent :$(\mathbf{v}) \rightarrow \mathbf{u}$ maps a vertex $\mathbf{v} \in \mathcal{V}$ to the unique vertex $\mathbf{u} \in \mathcal{V}$, such that $(\mathbf{u}, \mathbf{v}) \in \mathcal{E}$.
- If $\mathbf{v}_0 \in \mathcal{V}$ is the root vertex of the tree, $\mathbf{v}_0 = \text{Parent}(\mathbf{v}_0)$.
- Cost :$(\mathbf{v} \rightarrow \mathbb{R}_{\geq 0})$ maps a vertex $\mathbf{v} \in \mathcal{V}$ to the cost of the unique path from the root of the tree to $\mathbf{v}$, whereas the function C :$(\mathbf{x}, \mathbf{y}) \rightarrow \mathbb{R}_{\geq 0}$ delivers the cost of the edge from $\mathbf{x}$ to $\mathbf{y}$ as Euclidean distance between these vertices.

In lines $1-2$ of Alg. 1, the set of vertices is initialized with the goal position. The stopping criteria, i.e., the maximum of the size of $\mathcal{V}$, is set. The function AddNode is presented in the sub-algorithm Alg. 2.

At each iteration, a point $\mathbf{x}_{rand} \in \mathbb{X}_{free}$ is sampled (in line 4, Alg. 1). Then, this point is handed over to the Algorithm 2 in line 5. The nearest vertex $\mathbf{x}_{nearest} \in \mathcal{V}$ to the sampled point is computed (in line 1, Alg. 2). The steering function extends the tree from vertex $\mathbf{x}_{nearest}$ towards $\mathbf{x}_{rand}$, yielding vertex $\mathbf{x}_{new}$ (in line 2, Alg. 2). In the next step, the algorithm checks whether the resulting path between $(\mathbf{x}_{nearest}, \mathbf{x}_{new})$ violates the obstacle constraints (line 3, Alg. 2). If so, the vertex $\mathbf{x}_{new}$ is dropped and a new iteration is initiated. In the collision free case, the vertex is added to the tree, setting $\mathbf{x}_{nearest}$ to be the parent node of $\mathbf{x}_{new}$ for the moment, also computing the cost c_{min} to reach $\mathbf{x}_{new}$ (lines $4-6$, Alg. 2). As preparation for the rerouting steps, in line 7, Alg. 2, the near vertices $\mathcal{X}_{near}$ in $\mathcal{V}$ inside a ball of radius ρ centered at $\mathbf{x}_{new}$ are computed. Each vertex in the set $\mathcal{X}_{near}$ is then considered, from which vertex yields a shorter path to $\mathbf{x}_{new}$ (lines $8-14$, Alg. 2). If so, the corresponding node is set to be the parent node $\mathbf{x}_{min}$ and the value of c_{min} is updated, leading to the minimum-cost path to the recently added vertex (lines $11-12$, Alg. 2). After the first rerouting step, the edge $(\mathbf{x}_{min}, \mathbf{x}_{new})$ is added to the tree $\mathcal{G}$ in line 15, Alg. 2. In the second rerouting step at lines $16-22$ of Alg. 2, all vertices in $\mathcal{X}_{near}$ are reconsidered to search for a vertex that reached over a shorter route due to the newly added vertex $\mathbf{x}_{new}$. If one vertice $\mathbf{x}_{near}$ can be reached via $\mathbf{x}_{new}$ and the overall length of this path shows up to be shorter (lines $17-18$, Alg. 2), the existing edge $(\mathsf{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near})$ is dropped and replaced by $(\mathbf{x}_{new}, \mathbf{x}_{near})$ (line 20, Alg. 2). In Alg. 1, lines $3-6$ are repeated until the desired number of nodes $numNodes$ in the tree is reached. Next, the start node is added to the tree $\mathcal{G}$ in the same manner as

```
   Input: $\mathbb{X}_{free}, (\mathbf{x}_{start}, \mathbf{x}_{goal}) \in \mathbb{X}_{free}, \mathcal{O}, numNodes, \epsilon, \rho$
   Output: $\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathbf{wp}_{pos}$
1  $\mathcal{V} \leftarrow \{\mathbf{x}_{goal}\}; \mathcal{E} \leftarrow \emptyset$
2  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
3  while $size(\mathcal{V}) \leq numNodes$ do
4      $\mathbf{x}_{rand} \leftarrow \mathsf{SampleFree}(\mathbb{X}_{free})$
5      $\mathcal{G} \leftarrow \mathsf{AddNode}(\mathbf{x}_{rand}, \mathcal{G}, \mathcal{O}, \epsilon, \rho)$
6  end
7  $\mathcal{G} \leftarrow \mathsf{AddNode}(\mathbf{x}_{start}, \mathcal{G}, \mathcal{O}, \epsilon, \rho)$
8  $\mathbf{x}_{temp} \leftarrow \mathbf{x}_{start}$;
9  $\mathbf{wp}_{pos} \leftarrow \{\mathbf{x}_{temp}\}$
10 while $\mathsf{Parent}(\mathbf{x}_{temp}) \neq \mathbf{x}_{goal}$ do
11     $\mathbf{x}_{temp} \leftarrow \mathsf{Parent}(\mathbf{x}_{temp})$
12     $\mathbf{wp}_{pos} \leftarrow \mathbf{wp}_{pos} \cup \{\mathbf{x}_{temp}\}$
13 end
14 $\mathbf{wp}_{pos} \leftarrow \mathbf{wp}_{pos} \cup \{\mathbf{x}_{goal}\}$
```

Algorithm 1: RRT* algorithm

the other vertices (line 7, Alg. 1). At this stage of the RRT* algorithm, the tree G holds the goal position as root and the start position as general vertex. To obtain a collision free path from the algorithm, a temporary point and a set of position waypoints $\mathbf{wp}_{pos}$ is initialized with the starting point (lines $8-9$, Alg. 1). In lines $10-13$, Alg. 1, the tree is traversed by utilizing the Parent function and adding them to the set of waypoints. This is repeated until the goal node is reached. Finally, in line 14, Alg. 1, the goal node is added to the set of position waypoints $\mathbf{wp}_{pos}$.

The RRT* yields the collision free shortest path consisting of waypoints $\mathbf{wp}_{pos}$ from start to goal. However, this generated path is not smooth due to the randomness of the

```
Input: $\mathbf{x}_{rand}, \mathcal{G}, \mathcal{O}, \epsilon, \rho$
Output: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
1  $\mathbf{x}_{nearest} = \text{Nearest}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathbf{x}_{rand})$
2  $\mathbf{x}_{new} \leftarrow \text{Steer}(\mathbf{x}_{nearest}, \mathbf{x}_{rand}, \epsilon)$
3  if $\text{CollisionFree}(\mathbf{x}_{nearest}, \mathbf{x}_{new}, \mathcal{O})$ then
4      $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{x}_{new}\}$
5      $\mathbf{x}_{min} \leftarrow \mathbf{x}_{nearest}$
6      $c_{min} \leftarrow \text{Cost}(\mathbf{x}_{nearest}) + \text{C}(\text{Line}(\mathbf{x}_{nearest}, \mathbf{x}_{new}))$
7      $\mathcal{X}_{near} \leftarrow \text{Near}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathbf{x}_{new}, \rho)$
8      for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ do
9          if $\text{CollisionFree}(\mathbf{x}_{near}, \mathbf{x}_{new}, \mathcal{O}) \wedge \text{Cost}(\mathbf{x}_{near}) +$
10                 $\text{C}(\text{Line}(\mathbf{x}_{near}, \mathbf{x}_{new})) < c_{min}$ then
11             $\mathbf{x}_{min} \leftarrow \mathbf{x}_{near}$
12             $c_{min} \leftarrow \text{Cost}(\mathbf{x}_{near}) + \text{C}(\text{Line}(\mathbf{x}_{nearest}, \mathbf{x}_{new}))$
13         end
14     end
15     $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{x}_{min}, \mathbf{x}_{new}\}$
16     for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ do
17         if $\text{CollisionFree}(\mathbf{x}_{new}, \mathbf{x}_{near}, \mathcal{O}) \wedge \text{Cost}(\mathbf{x}_{new}) +$
18                 $\text{C}(\text{Line}(\mathbf{x}_{new}, \mathbf{x}_{near})) < \text{Cost}(\mathbf{x}_{near})$ then
19             $\mathbf{x}_{parent} \leftarrow \text{Parent}(\mathbf{x}_{near})$
20             $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{(\mathbf{x}_{parent}, \mathbf{x}_{near})\}) \cup \{(\mathbf{x}_{new}, \mathbf{x}_{near})\}$
21         end
22     end
23 end
```

Algorithm 2: Function AddNode of the RRT* algorithm

generated tree of the RRT* algorithm. Hence, in the next subsection, the line of sight (LOS) is utilized to remove the unnecessary points of the generated path.

5.3 Line of Sight Optimization

The Line of Sight optimization removes redundant points from the set of position waypoints $\mathbf{wp}_{pos}$. The pseudocode for the Line of Sight optimization is given in Alg. 3. In addition, a simplified scenario is depicted in Figure 5.3. Starting with the first waypoint as temporary vertice $\mathbf{x}_{temp}$ in green (line 1, Alg. 3), the algorithm begins to first check for the longest possible straight-lined, collision free path from $\mathbf{x}_{temp}$. The LOS first checks for an open connection to the last waypoint (Figure 5.3(a)). If there is a collision, the counting index i is increased (line 7, Alg. 3) to find a collision free path to the parent of the previous waypoint, see Figure 5.3(b). This procedure is repeated (lines 3-11, Alg. 3), until a collision free path is found. Since the initial set of waypoints is already collision free, there is always one path found which contains less waypoints w.r.t. the path in Figure 5.3(a). For example in Figure 5.3(d), a collision free path is found.

This algorithm minimizes the number of waypoints, as well as the total distance of the computed path by RRT*. The reduced set of waypoints is then processed to a dynamically feasible trajectory, which is explained in Section 5.4.

```
   Input: $\mathbf{wp}_{pos}, \mathcal{O}$
   Output: $\mathbf{wp}_{pos}$
1  $\mathbf{x}_{temp} \leftarrow \mathbf{wp}_{pos}(1)$
2  $i = 0$
3  while $\mathbf{x}_{temp} \neq \mathbf{wp}_{pos}(end)$ do
4      if CollisionFree($\mathbf{x}_{temp}, \mathbf{wp}_{pos}(end - i), \mathcal{O}$) then
5          $\mathbf{wp}_{pos} \leftarrow$ DeleteWaypointsBetween($\mathbf{x}_{temp}, \mathbf{wp}_{pos}(end - i)$
6          $\mathbf{x}_{temp} \leftarrow \mathbf{wp}_{pos}(end - i)$
7          $i = 0$
8          continue
9      end
10     $i = i + 1$
11 end
```

Algorithm 3: Line of Sight Optimization (LOS)

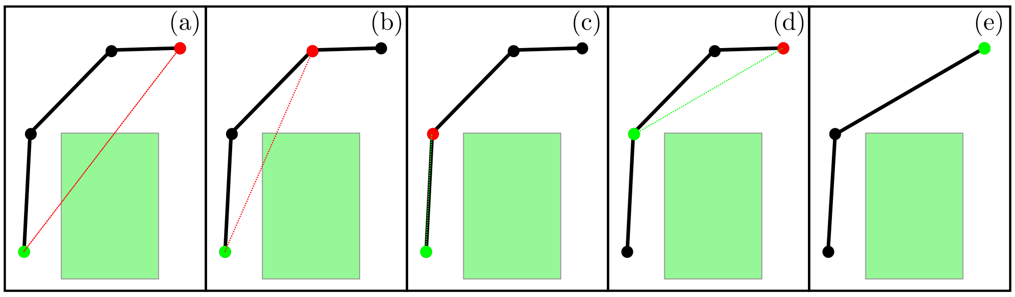


Figure 5.3: Simple scenario of the LOS optimization.

5.3.1 Yaw angle planning

The flat outputs of the quadcopter are the position of the center of mass $\mathbf{r} = [x, y, z]^{\mathrm{T}}$ and its yaw angle ψ. There is still a missing yaw angle definition, since RRT* and LOS only define the path of the position $\mathbf{r} = \left[x, y, z\right]$. Hence, the yaw path has to be taken into account.

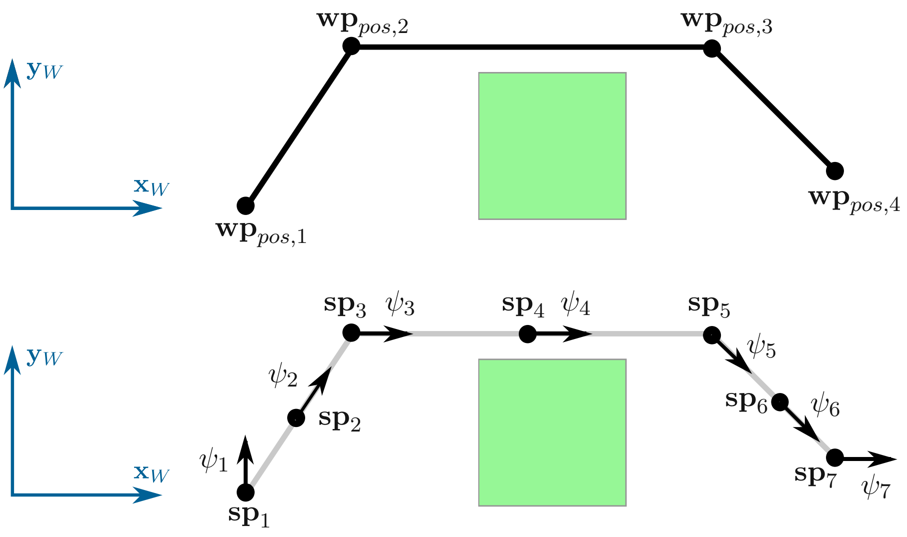


Figure 5.4: Example of the yaw trajectory waypoint creation.

Note that it is important to plan the yaw trajectory in a way that the front-facing camera can capture unknown obstacles in the direction of flight. This is accomplished by first inserting an additional point between vertices of the computed LOS path to create a fine grid of support points $\mathbf{sp}_i$ along the path, as visualized in Figure 5.4. This leads to a number of $Y = (2M - 1)$ support points and $(2M - 2)$ yaw trajectory segments respectively, with M as the number of position waypoints $\mathbf{wp}_{pos}$ obtained from the

LOS algorithm and $M-1$ as the number of position trajectory segments. A yaw angle waypoint ψ_i is computed in the form

$$\psi_i = \arctan \frac{\mathbf{sp}_{i+1,y} - \mathbf{sp}_{i,y}}{\mathbf{sp}_{i+1,x} - \mathbf{sp}_{i,x}}, i = 2, ..., Y-1, \tag{5.1}$$

whereas the first yaw angle is fixed by the initial pose of the drone and the last one is set by the user to ensure the drone will reach a specific final pose at the end of the trajectory.

To this end, the set of yaw waypoints

$$\mathbf{wp}_{yaw} = \{\psi_1, \psi_2, \dots, \psi_Y\} \tag{5.2}$$

will be used to generate a meaningful yaw trajectory.

5.3.2 Gilbert-Johnson-Keerthi Distance Algorithm

In this subsection, the Gilbert-Johnson-Keerthi distance algorithm [43], utilized for computing the distance between two convex sets, is introduced. The GJK is widely used in applications, e.g., robotics, rigid-body dynamics, computer graphics, physics, and computational mechanics [44]. Furthermore, this algorithm can be utilized to determine collisions of convex sets. In this work, the GJK algorithm is used in the RRT*, LOS, and to check whether the path is collision free or not. A simplified description of the GJK algorithm is given below.

The GJK algorithm substantially relies on the concept of the Minkowski sum. Considering the convex shapes $\mathcal{A}$ and $\mathcal{B}$, the Minkowski sum of $\mathcal{A}$ and $\mathcal{B}$ is the addition of all points of $\mathcal{A}$ to all points of $\mathcal{B}$ in the form

$$\mathcal{C} = \mathcal{A} + \mathcal{B} = \{\mathbf{a} + \mathbf{b} | \mathbf{a} \in \mathcal{A}, \mathbf{b} \in \mathcal{B}\}, \tag{5.3}$$

yielding a convex shape $\mathcal{C}$. The subtraction $\mathcal{C}$ in the Minkowski sum, referred as Minkowski difference

$$\mathcal{C} = \mathcal{A} - \mathcal{B} = \{\mathbf{a} - \mathbf{b} | \mathbf{a} \in \mathcal{A}, \mathbf{b} \in \mathcal{B}\}, \tag{5.4}$$

is again a convex set. Performing a Minkowski difference of intersecting shapes $\mathcal{A}$ and $\mathcal{B}$, the obtained convex shape $\mathcal{C}$ contains the origin $\{0\}$. This core statement is crucial in the implementation of the algorithm. The problem of determining the intersection of two shapes is transformed into the task, to check whether the origin is inside the Minkowski difference of these shapes.

The GJK algorithm does not actually compute the Minkowski difference, but iteratively builds a polygon inside the Minkowski difference called simplex that attempts to enclose the origin. The simplex is therefore a subset of the Minkowski difference. If the simplex encloses the origin, the Minkowski difference also contains it, since this is a property of convex sets. The simplex is constructed with the help of a support function s. Support functions map a search direction $\mathbf{d}$ to the furthest point of a shape $\mathcal{S}$ in that direction, the so-called support point $\mathbf{p}$.

$$\mathbf{p} = s_{\mathcal{S}}(\mathbf{d}) = \arg\max_{\mathbf{v}\in\mathcal{S}} \mathbf{v}^{\mathrm{T}}\mathbf{d} \tag{5.5}$$

With a support point of shape $\mathcal{A}$ in direction $\mathbf{d}$, and a support point of shape $\mathcal{B}$ in $(-\mathbf{d})$-direction, a support point $\mathbf{p}$ of the Minkowski difference of (5.4) is obtained as

$$s_C(\mathbf{d}) = s_A(\mathbf{d}) - s_B(-\mathbf{d}) = \mathbf{p} \in \mathcal{C}. \tag{5.6}$$

The usage of support function increases the performance of the algorithm, as the simplex with the maximum area is constructed via support points. This fact increases the possibility that the simplex encloses the origin.

In Figure 5.5, the basic principle of the GJK algorithm is explained with a 2D example. The shape $\mathcal{A}$ and the shape $\mathcal{B}$ are illustrated in green and blue color, respectively. The Minkowski difference in (5.4) is depicted as red region in Figure 5.5(a). The first point of the simplex is initialized with a random search direction $\mathbf{d}$, indicated as arrow inside each shape in Figure 5.5(b). Note that the search directions of shape $\mathcal{A}$ and $\mathcal{B}$ are in opposite orientations. This yields the first vertex of the simplex. The next search direction is chosen to be towards the origin, as the algorithm attempts to build a polygon that encloses the origin, see Figure 5.5(c). Using this direction in (5.6), the second point of the simplex is obtained, see Figure 5.5(c). The third search direction $\mathbf{d}$ is then chosen to be the normal vector on the connection of the first two points of the simplex, again towards the origin, see Figure 5.5(d). Applying this direction to (5.6), the third point is obtained. The simplex after the initialization is given in violet color in Figure 5.5(d). The examination, whether the origin is inside the simplex or not, is performed by a series of line tests in 2D using cross and dot products. As the simplex does not contain the origin, a new search direction is constructed based on the cross and dot products. The new search direction is again the normal vector towards the origin, from the side of the simplex closest to the origin. The third point is removed from the simplex, and the new one is added, see Figure 5.5(e). The obtained simplex encloses the origin. Thus, an intersection of $\mathcal{A}$ and $\mathcal{B}$ is detected. In the non-intersection case, an end-criterion for the algorithm is a constant simplex over two iterations, which does not contain the origin.

The same concept applies to the 3D case, with the difference that the simplex has the shape of a tetrahedron instead of a triangle. Also, more elaborate plane tests must be performed instead of line tests to determine the location of the origin relative to the simplex.

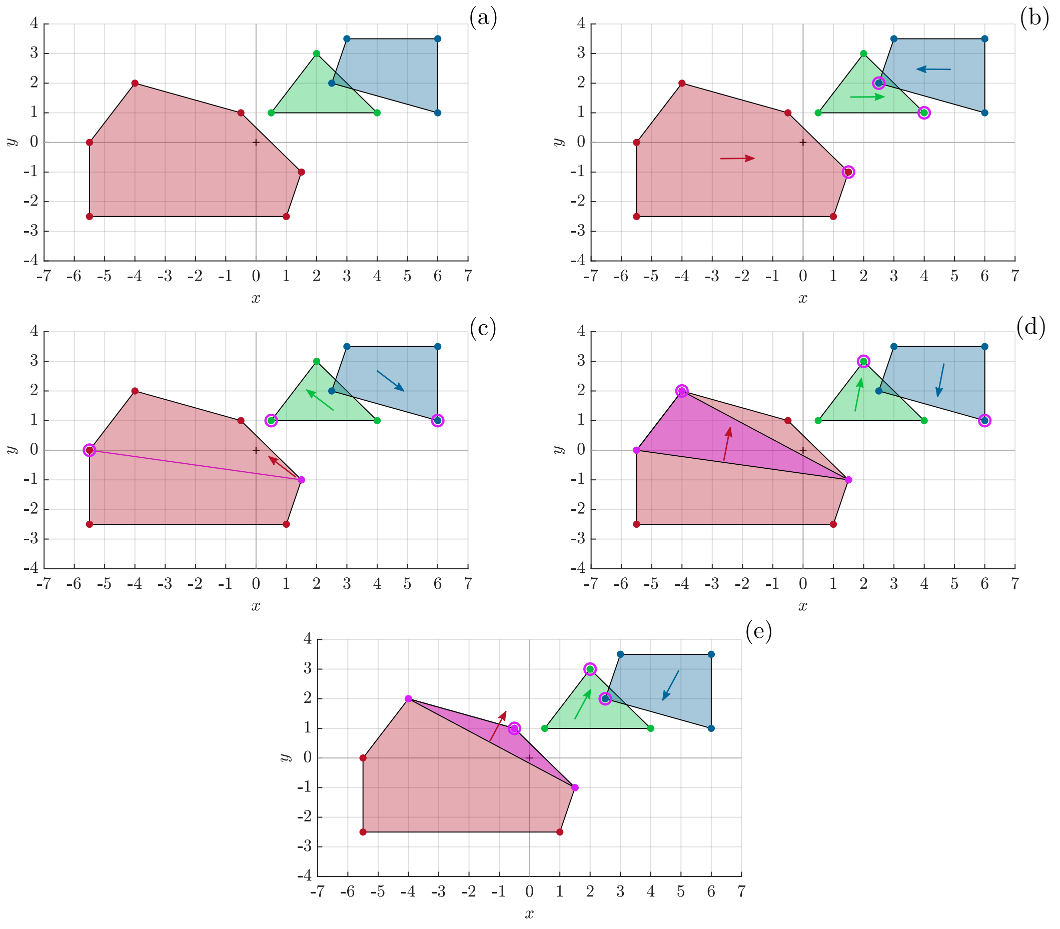


Figure 5.5: Example for the working principle of the GJK algorithm in 2D.

5.4 Trajectory Generation

Combining the results from Section 5.3 and Section 5.3.1, the path from the initial position to the target position is given as a set of position waypoints in the form

$$\mathbf{wp}_{pos} = \{\mathbf{wp}_{pos,1}, \mathbf{wp}_{pos,2}, ..., \mathbf{wp}_{pos,M}\}, \tag{5.7}$$

with $\mathbf{wp}_{pos,i} = [x_i, y_i, z_i]^{\mathrm{T}}$ and M as the number of position waypoints. In addition, a set of yaw angle waypoints

$$\mathbf{wp}_{yaw} = \{\psi_1, \psi_2, ..., \psi_Y\}, \tag{5.8}$$

with $Y = 2M - 1$ as the number of yaw angle waypoints is defined. The trajectory generation can be separated into four independent optimization problems for each flat output, see, e.g., [45], [8]. In the following, the trajectory generation process is explained for a position trajectory. The same principle applies to the yaw trajectory, differences to position trajectories are noted accordingly.

Since the system state $\mathbf{x}$ and the control input $\mathbf{u}$ can be straightforwardly computed in (4.14), the trajectory is parameterized by the time t in the following. For each optimization problem, a flat output trajectory, considered as a piecewise polynomial function, reads as

$$P(t) = \begin{cases} P_1(t) = p_{10} + p_{11}t + p_{12}t^2 + \cdots + p_{1N}t^N & t_0 \le t < t_1, m = 1 \\ P_2(t) = p_{20} + p_{21}t + p_{22}t^2 + \cdots + p_{2N}t^N & t_1 \le t < t_2, m = 2 \\ \quad\vdots & \\ P_M(t) = p_{M0} + p_{M1}t + p_{M2}t^2 + \cdots + p_{MN}t^N & t_{m-1} \le t < t_m, m = M-1 \end{cases}$$

where t_m is the time segment, $P(t)$ is a flat output trajectory and N is the order of the trajectory polynomial. For position trajectories, i.e. $x(t)$, $y(t)$ and $z(t)$, the number of segments is $m = M - 1$. On the other hand, for the yaw trajectory $\psi(t)$, the number of segments results in $m = Y - 1$. Note that more yaw angle waypoints than position waypoints are imposed in the independent optimization problems, as it is important to ensure the front-facing camera can capture obstacles in direction of flight. To generate a smooth trajectory with snap minimization, the order of the time parameterized polynomial $P(t)$ is $N = 7$. The proof of this optimization is given in the Appendix.

Since the cost function (5.9) depends on the segment time $T_m = t_m - t_{m-1}$, a priori selection of this travel time is needed. This segment time T_m can be computed trivially by dividing the length of the segment over the average speed of the quadcopter [14]. Another approach is to utilize the time-optimal trajectory planning [45], taking into account the bounds on velocity, acceleration, and jerk of the quadcopter. A precise analysis of the time-optimal trajectory planning algorithm is out of scope of this thesis, only the basic idea is presented below. The time-optimal trajectory planning algorithm, also called the 15-step algorithm, sequentially computes the time-optimal trajectory for each flat output. Considering constant values for the snap $(s_{max}, 0, -s_{max})$, a flat output is parameterized as a 4^{th}-order polynomial in time. In Figure 5.6, the 15 stages of the snap s over the time t are presented. In each step, at least one derivative reaches its limit. If the velocity limit is reached with a specific set of limitations for each derivative, then the retrieved trajectory is considered as time optimal. However, if the maximum speed is not achieved due to the application or constraints, the received trajectory is not time optimal. This algorithm is executed for the three positions $\sigma_1 = x$, $\sigma_2 = y$, $\sigma_3 = z$ and each segment m, $m = 1, \dots, M-1$, yielding three different segment times per segment. To ensure that the final trajectory can be followed by the quadcopter, the longest time estimation of each position segment m will be further used in the quadratic program (QP) for every flat output as T_m, which allows to reach the waypoints synchronously in every dimension.

As more yaw angle trajectory segments than position trajectory segments are obtained with the implemented approach, these segment times have to be adjusted accordingly for yaw angle trajectory generation. In Section 5.3.1, for construction of the yaw trajectory support points each position path segment was bisected, see Figure 5.4. Therefore, the segment times T_m of the position segment are also bisected for the yaw angle trajectory generation process.

The cost function of the m^{th} segment of (5.9) is defined as

$$J_m(\mathbf{w}, T_m) = \int_0^{T_m} w_1 P'_m(t)^2 + w_2 P''_m(t)^2 + \dots + w_N P_m^{(N)}(t)^2 dt = \mathbf{p_m}^T \mathbf{Q_m}(T_m)\mathbf{p_m}, \quad (5.9)$$

where $\mathbf{p}_m$ is the vector of the $N+1$ coefficients of the polynomial $P_m(t)$ and $w_i > 0, i = 1, \dots, N$ is the the weight of the i^{th} derivative, denoted by $()', ()', \dots, ()^N$. The construction of the Hessian matrix $\mathbf{Q}_m(T_m)$ is shown by a simple example in the following. Assuming to generate a minimum velocity trajectory for a time segment of the order $N = 4$, the basic polynomial function $P(t)$ with the corresponding velocity $P'(t)$ reads as

$$P(t) = p_0 + p_1 t + p_2 t^2 + p_3 t^3 + p_4 t^4 \quad (5.10)$$
$$P'(t) = p_1 + 2p_2 t + 3p_3 t^2 + 4p_4 t^3. \quad (5.11)$$

The cost function for a minimum velocity trajectory of one segment m with weight $w_1 = 1$ results in

$$J_{min_{vel}} = \int_0^{T_m} P'_m(t)^2 dt. \quad (5.12)$$

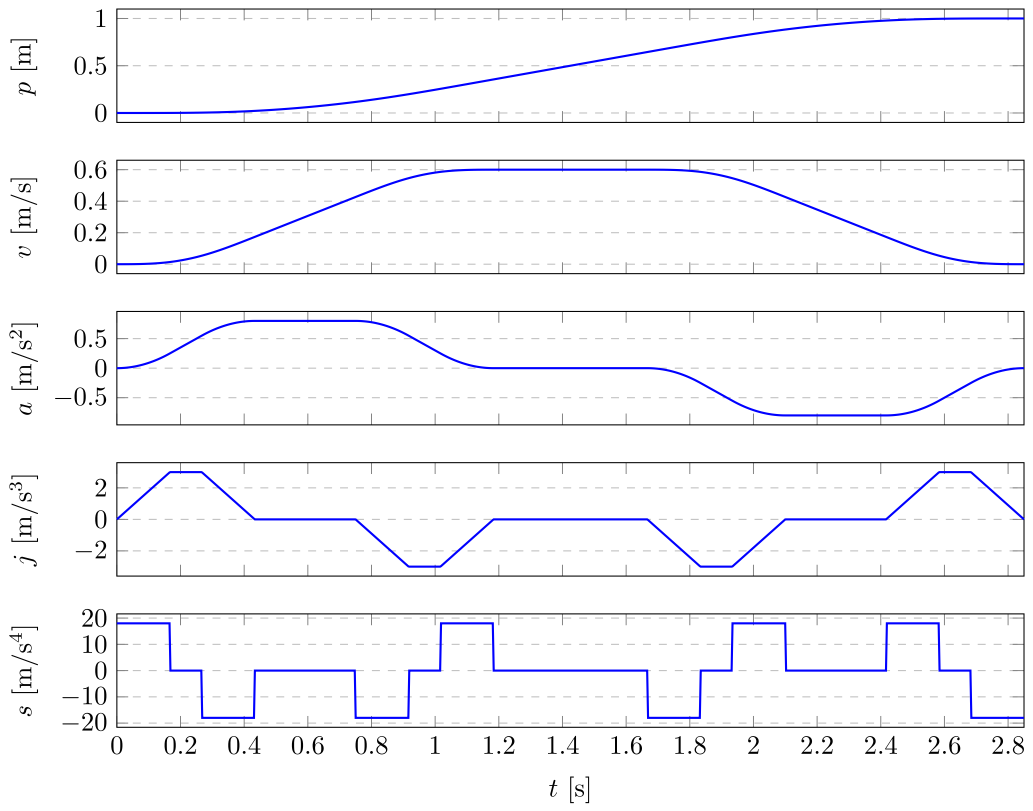


Figure 5.6: Example for a time-optimal 15-step trajectory, where p,v,a,j and s denote the position, velocity, acceleration, jerk and snap. The steps can be clearly seen in the trend of the snap s over time t.

The square of $P'_m(t)$ is expressed in the form

$$
\begin{array}{cccccccc}
\multicolumn{8}{l}{P'_m(t)^2 = (p_1 + 2p_2t + 3p_3t^2 + 4p_4t^3)(p_1 + 2p_2t + 3p_3t^2 + 4p_4t^3) =} \\
(& p_1^2 & + & 2p_1p_2t & + & 3p_1p_3t^2 & + & 4p_1p_4t^3 \quad + \\
 & 2p_1p_2t & + & 4p_2^2t^2 & + & 6p_2p_3t^3 & + & 8p_2p_4t^4 \quad + \\
 & 3p_1p_3t^2 & + & 6p_2p_3t^3 & + & 9p_3t^4 & + & 12p_3p_4t^5 \quad + \\
 & 4p_1p_4t^3 & + & 8p_2p_4t^4 & + & 12p_3p_4t^5 & + & 16p_4t^6 \quad).
\end{array}
$$

Hence, the cost function (5.12) reads as

$$J_{min_{vel}} = \int_0^{T_m} P_m'(t)^2 dt =$$

$$\begin{array}{cccccccccc} (& p_1^2 t & + & 2p_1p_2\frac{t^2}{2} & + & 3p_1p_3\frac{t^3}{3} & + & 4p_1p_4\frac{t^4}{4} & + \\ & 2p_1p_2\frac{t^2}{2} & + & 4p_2^2\frac{t^3}{3} & + & 6p_2p_3\frac{t^4}{4} & + & 8p_2p_4\frac{t^5}{5} & + \\ & 3p_1p_3\frac{t^3}{3} & + & 6p_2p_3\frac{t^4}{4} & + & 9p_3\frac{t^5}{5} & + & 12p_3p_4\frac{t^6}{6} & + \\ & 4p_1p_4\frac{t^4}{4} & + & 8p_2p_4\frac{t^5}{5} & + & 12p_3p_4\frac{t^6}{6} & + & 16p_4\frac{t^7}{7} &)|_{t=0}^{t=T_m}, \end{array}$$

For a compact notation, (5.12) can be expressed in matrix form

$$J_{min_{vel}} = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 & p_4 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & T_m & T_m^2 & T_m^3 & T_m^4 \\ 0 & T_m^2 & 4\frac{T_m^3}{3} & 6\frac{T_m^4}{4} & 8\frac{T_m^5}{5} \\ 0 & T_m^3 & 6\frac{T_m^4}{4} & 9\frac{T_m^5}{5} & 12\frac{T_m^6}{6} \\ 0 & T_m^4 & 8\frac{T_m^5}{5} & 12\frac{T_m^6}{6} & 16\frac{T_m^7}{7} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \mathbf{p}^T \mathbf{Q}(T_m)\mathbf{p}. \tag{5.13}$$

Note that if more than one derivative are considered in the cost function, the same procedure can be applied and the Hessian matrices are summed up.

Combining the cost matrices of each segment, the total cost function of $M-1$ polynomial segments used by the QP can be written in the form

$$J_{tot} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix}^T \begin{bmatrix} \mathbf{Q}_1(T_1) & & \\ & \ddots & \\ & & \mathbf{Q}_M(T_{M-1}) \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix}. \tag{5.14}$$

In order to guarantee the continuity of the trajectory, constraints on the endpoints of each segment are considered. This allows an endpoint of a trajectory segment to be equal to an initial point of the subsequent trajectory segment, which helps to create smooth transitions between segments [14].

These constraints are obtained via a mapping matrix $\mathbf{A}_m$ between the polynomial coefficients $\mathbf{p}_m$ and the endpoint derivatives $\mathbf{d}_m$ for each segment m in the form

$$\mathbf{A}_m \mathbf{p}_m = \mathbf{d}_m, \tag{5.15}$$

where $\mathbf{A}_m = [\mathbf{A}_0, \mathbf{A}_T]_m^{\mathrm{T}}$ and $\mathbf{d}_m = [\mathbf{d}_0, \mathbf{d}_T]_m^{\mathrm{T}}$. Here, $\mathbf{d}_0$ and $\mathbf{d}_T$ denote the corresponding values of the derivatives of the start points and endpoints. If there are no specific values for the endpoint derivatives, continuity constraints must be introduced to enforce a smooth transition between the segments. To do so, the derivatives at the end of segment m are enforced to be equal to the derivatives at the beginning of the $(m+1)^{st}$ segment.

$$\mathbf{A}_{T,m}\mathbf{p}_m = \mathbf{A}_{0,m+1}\mathbf{p}_{m+1} \tag{5.16a}$$

$$\mathbf{A}_{T,m}\mathbf{p}_m - \mathbf{A}_{0,m+1}\mathbf{p}_{m+1} = \mathbf{0} \tag{5.16b}$$

Thus, the constraint at the start of segment m of (5.15) reads as

$$\begin{aligned} P'_m(0) &= p_{m,1} + 2p_{m,2}t + 3p_{m,3}t^2 + 4p_{m,4}t^3|_{t=0} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{m,0} \\ p_{m,1} \\ p_{m,2} \\ p_{m,3} \\ p_{m,4} \end{bmatrix} = \mathbf{A}_{0,m}\mathbf{p}_m = \mathbf{d}_{0,m} = \mathbf{0}, \end{aligned}$$

and the constraint at the end of segment $m+1$ reads as

$$\begin{aligned} P'_{m+1}(T_{m+1}) &= p_{m+1,1} + 2p_{m+1,2}t + 3p_{m+1,3}t^2 + 4p_{m+1,4}t^3|_{t=T_{m+1}} \\ &= \begin{bmatrix} 0 & 1 & 2T_{m+1} & 3T_{m+1}^2 & 4T_{m+1}^3 \end{bmatrix} \begin{bmatrix} p_{m+1,0} \\ p_{m+1,1} \\ p_{m+1,2} \\ p_{m+1,3} \\ p_{m+1,4} \end{bmatrix} = \mathbf{A}_{T,m+1}\mathbf{p}_{m+1} = \mathbf{d}_{T,m+1} = \mathbf{0}. \end{aligned}$$

Additionally, the continuity condition (5.16b) is constructed

$$\begin{aligned} P'_m(T_m) - P'_{m+1}(0) = 0 \\ &= \{p_{m,1} + 2p_{m,2}t + 3p_{m,3}t^2 + 4p_{m,4}t^3\}|_{t=T_m} - \\ &\quad \{p_{m+1,1} + 2p_{m+1,2}t + 3p_{m+1,3}t^2 + 4p_{m+1,4}t^3\}|_{t=0} \\ &= \begin{bmatrix} 0 & 1 & 2T_m & 3T_m^2 & 4T_m^3 \end{bmatrix} \begin{bmatrix} p_{m,0} \\ p_{m,1} \\ p_{m,2} \\ p_{m,3} \\ p_{m,4} \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{m+1,0} \\ p_{m+1,1} \\ p_{m+1,2} \\ p_{m+1,3} \\ p_{m+1,4} \end{bmatrix} \\ &= \mathbf{A}_{T_m,m}\mathbf{p}_m - \mathbf{A}_{0,m+1}\mathbf{p}_{m+1} = \mathbf{0}. \end{aligned}$$

All constraints for M segments can be written into a single set of equality constraints in the form of

$$\mathbf{A}_{tot} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{M-1} \end{bmatrix}. \tag{5.17}$$

Combining (5.14) and (5.17), the constrained optimization problem reads as

$$\begin{aligned} \min_{P_1,\dots,P_{M-1}} \quad & \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix}^T \begin{bmatrix} \mathbf{Q}_1(T_1) & & \\ & \ddots & \\ & & \mathbf{Q}_M(T_{M-1}) \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix} \\ \text{s.t.} \quad & \mathbf{A}_{tot} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{M-1} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{M-1} \end{bmatrix}. \end{aligned} \tag{5.18}$$

5.5 Obstacle Avoidance and Replanning

This section explains the online ROS loop in Figure 5.2. This helps the quadcopter to detect new obstacles and to compute an online collision free trajectory.

In Figure 5.2, the *Sense and Process* block interacts with the ROS master, fetching the latest position, orientation and depth camera image of the quadcopter. The depth camera point cloud is processed and transformed online in order to detect new obstacles. Two different approaches are presented to detect new obstacles. The first approach mainly depends on the computer vision toolbox of MATLAB utilizing point cloud functionalities. The second approach relies on the GJK algorithm. Once a new obstacle is detected, the environment information is updated and a decision is made whether replanning is necessary or not. If the drone is predicted to collide with new obstacles, the replanning will be activated using previously introduced procedures, i.e., RRT* and LOS for finding the collision free waypoints and the constrained trajectory generation. The current trajectory is sent via a ROS publisher to the drone, constantly initiated by a callback function. If the trajectory is finished or a replanning is not possible in time, the drone is landed immediately.

5.5.1 Sense and Process

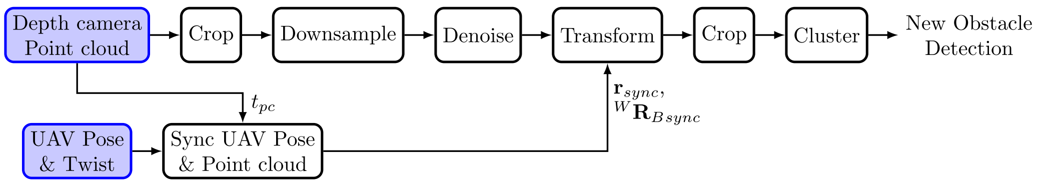


Figure 5.7: Detailed workflow of the *Sense and Process* block.

A detailed visualization of the *Sense and Process* block [34] in Figure 5.2 is shown in Figure 5.7. To improve the computational efficiency, the crop and downsample processes are taken into account. Therefore, the raw input is first cropped to remove cloud points which are too far away from the quadcopter's position. Then, the downsampling process is employed to keep a single point in a voxel of size 0.1 m. Since the RGB-D camera is sensitive to noise, filtering processes are necessary to pre-process raw point clouds. Depth camera data is represented in the body frame, which has to be transformed by using (4.2) to the world coordinate frame. In general, the position and depth camera data are not acquired at the exact same time. Thus, there is a time difference $t_{diff} = t_{pos} - t_{pc}$ between these two events.

When the quadcopter moves very quickly, the motion blur could cause wrong point cloud outputs, leading to a failure of a collision free trajectory replanning. To solve this issue, similar to [34], we only accept the filtered point cloud if the angular velocity of the three Euler angles $\mathbf{e} = \{\phi, \theta, \psi\}$ is lower then a limit ω_{max}. Before transformation, the pose message from the UAV should be aligned with the point cloud timestamp t_{pc} by the estimation in (5.19), where $\mathbf{r}_0$ and $\mathbf{e}_0$ denote the position and the Euler angles measurement at timestamp t_{pos}. The translational and rotational velocities are approximated to be constant during t_{diff}

$$\begin{bmatrix} \mathbf{r}_{sync} \\ \mathbf{e}_{sync} \end{bmatrix} = \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{e}_0 \end{bmatrix} - \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega}_{BW} \end{bmatrix} t_{diff}. \tag{5.19}$$

After the synchronisation, the depth data is cropped again to the boundaries of the available flight space $\mathbb{X}_{free}$. The obtained point cloud is then clustered taking a distance threshold $thresh_{pcsegdist}$ to spatially distinguish multiple clusters. Since depth camera data is very noisy, especially in highly enlighted and reflective environments, a cluster is considered as valid if it contains a minimum number of data points. Finally, to determine if one cluster of the depth camera information represents a new obstacle, the two following obstacle detection approaches are applied.

5.5.2 Obstacle Detection

Point Cloud Approach

When using this method, a point cloud of the previously known surroundings, denoted by pc_{world}, is stored serving as a reference. This data may be retrieved from previous flights or can be generated beforehand. The pseudocode of this approach is given in Alg. 4, whereas the function convertPc2Box(pc) takes a point cloud pc and returns its hit box with the maximum length in each dimension of pc. The algorithm takes the current environment pc_{world}. The obtained point cloud clusters from the depth camera are denoted by $clusters$. Also, a threshold value for differentiating known and new point clouds has to be set. An updated pc_{world} and a set of new obstacles $\mathcal{O}_{new}$ regarding trajectory replanning are returned. Each cluster is checked, whether it constitutes a new obstacle or not (lines $2-9$, Alg. 4). For this purpose, the MATLAB function findPointsInROI is utilized in line 3 of Alg. 4. The spatial area of the cluster pc_O from the well-known point cloud pc_{world}, which is named pc_W, is used to check their similarity. A search for the nearest neighbor in pc_W for each point of pc_O is conducted utilizing the MATLAB function knnsearch, see line 4 of Alg. 4. Once this task is completed, a *rmse* of all distances is calculated, which serves as a measure for determining if that cluster represents an unknown obstacle, see line 5 of Alg. 4. The distinction between a known and an unknown obstacle is performed by a threshold value $thresh_{rmse}$, which can be adjusted depending on the application or object size. When a cluster is detected to be unknown or new, its bounding box is treated as a new object in the further process of replanning, see line 6 of Alg. 4 and the *Update Environment* block in Figure 5.2. In order to update the overall world point cloud pc_{world}, it is merged with pc_O by using the MATLAB function pcmerge in line 7 of Alg. 4.

```
  Input: $pc_{world}, clusters, thresh_{rmse}$
  Output: $\mathcal{O}_{new}, pc_{world}$
1 $\mathcal{O}_{new} \leftarrow \{\}$
2 for $pc_O \in clusters$ do
3     $pc_W \leftarrow$ findPointsInROI($pc_{world}, pc_O$)
4     $\mathbf{d} \leftarrow$ knnsearch($pc_O, pc_W$)
5     if rmse($\mathbf{d} > thresh_{rmse}$ then
6         $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \cup$ convertPc2Box($pc_O$)
7         $pc_{world} \leftarrow$ pcmerge($pc_{world}, pc_O$)
8     end
9 end
```

Algorithm 4: Point cloud approach.

Eight-Corner Approach

This approach reduces each point cloud cluster to a cuboid-shaped box with the maximum length of the cluster in each dimension, leading to a convex set of eight corner points representing one cluster or obstacle. The pseudocode of this approach is listed in Alg. 5. Some functions used in Alg. 5 are introduced in the following.

- Function getDistanceMatrix($\mathbf{C}, \mathbf{O}$) returns a distance matrix, containing the minimum distance between each box $\mathbf{C} \in \mathcal{C}$ and each known obstacle $\mathbf{O} \in \mathcal{O}$. Therefore, the GJK algorithm is utilized.
- Function mergeBoxes($\mathbf{A}, \mathbf{B}$) merges two cuboid hit boxes $\mathbf{A}$ and $\mathbf{B}$ to one hit box.
- Function getCornerDistances($\mathbf{A}, \mathbf{B}$) returns the minimum distance of each corner of the hit box $\mathbf{A}$ to the closest known obstacle $\mathbf{B}$.

The information about the environment is provided by a set of obstacles $\mathcal{O}$, whereby each one is represented by its eight corner point coordinates. As parameters, Alg. 5 also gets the detected point cloud clusters and a distance threshold $thresh_{dist}$. The outputs are the newly detected obstacles $\mathcal{O}_{new}$ as well as the set of known obstacles $\mathcal{O}$.

At first, all point cloud clusters are converted to a set of cuboid boxes in line 2 of Alg. 5. If new obstacles are detected, they will be added to the set $\mathcal{O}$, see lines $3-6$ of Alg. 5. The vector $\mathbf{D}$ holds the minimum distance of each box to the closest known obstacle, $\mathbf{I}$ holds the corresponding index idx of the closest obstacle in $\mathcal{O}$ w.r.t. $\mathbf{C}$, see line 7 of Alg. 5. Each box $\mathbf{C} \in \mathcal{C}$ undergoes checking processes in lines $8-24$ of Alg. 5, whether it represents a new obstacle or not. If the smallest distance d between the found box $\mathbf{C}$ and the closest known object is greater than the threshold value $thresh_{dist}$, the found object $\mathbf{C}$ is recognized as a new object, see lines $10-13$ of Alg. 5. In lines $14-18$ of Alg. 5, a check for intersecting boxes is made, which is reflected by a distance of zero. If known obstacles and found obstacles are intersecting, both are merged and updated as new obstacles (line 15, Alg. 5). The previously known obstacle is removed in line 16, as it would be redundant. In line 19 of Alg. 5, the distance of each corner of the newly detected hit box $\mathbf{C}$ to the closest known obstacle $\mathcal{O}(idx)$ is obtained. If the minimum distance d of the convex set $\mathbf{C}$ to $\mathcal{O}(idx)$ is smaller than $thresh_{dist}$, and one corner point of $\mathbf{C}$ is further away than the threshold, both hit boxes are merged, see lines $21-22$ of Alg. 5. This condition allows the assembly of several clusters, obtained from one big real obstacle, to one hit box for trajectory replanning even with bad transformations of the depth camera data. This avoids detecting redundant objects.

5.5.3 Trajectory Validation

The trajectory validation block conducts a validity check of the current UAV trajectory, if unknown obstacles are detected. In the first step of this verification, newly discovered obstacles are inflated by a safety margin in every dimension to guarantee no collisions. After that, intersection checks of all straight-line path segments of the current trajectory with the new obstacles are conducted. The penalizing weights in (5.9) ensure, that the final position trajectory is reasonably close to the straight-line path obtained from the RRT* planner. Therefore, only small deviations are expected. This simplification leads to a faster processing of the validity check, since we save multiple evaluations of the polynomials in the $x-$, $y-$ and $z-$direction. If all new obstacles do not yield an intersection with the current path, no action is performed and the current trajectory will further be followed by the quadcopter.

Note that an estimation whether a trajectory replanning is feasible during flight is made when there are possible collisions with new obstacles. Trajectory replanning is initiated, if the following conditions are fulfilled:

- The target position is still reachable. In other words, the goal is not occupied by an object.

```
Input: $\mathcal{O}, clusters, thresh_{dist}$
Output: $\mathcal{O}_{new}, \mathcal{O}$
$\mathcal{O}_{new} \leftarrow \{\}$
$\mathcal{C} \leftarrow \mathsf{convertPc2Box}(clusters)$
if $\mathsf{size}(\mathcal{O}) == 0 \;\&\; \mathsf{size}(\mathcal{C}) > 0$ then
    $\mathcal{O}_{new} \leftarrow \mathcal{C}$
    return
end
$\mathbf{D}, \mathbf{I} \leftarrow \mathsf{min}(\mathsf{getDistanceMatrix}(\mathcal{C}, \mathcal{O}))$
for $\mathbf{C} \in \mathcal{C}$ do
    $d \leftarrow \mathbf{D}(i); idx \leftarrow \mathbf{I}(i);$
    if $d > thresh_{dist}$ then
        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \cup \mathbf{C}$
        continue
    end
    if $d == 0$ then
        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \cup \mathsf{mergeBoxes}(\mathbf{C}, \mathcal{O}(idx))$
        $\mathcal{O} \leftarrow \mathcal{O} \setminus \mathcal{O}(idx)$
        continue
    end
    $\mathbf{d}_{corner} \leftarrow \mathsf{getCornerDistances}(\mathbf{C}, \mathcal{O}(idx))$
    if $\mathsf{max}(\mathbf{d}_{corner}) > thresh_{dist} \;\&\; d < thresh_{dist}$ then
        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \cup \mathsf{mergeBoxes}(\mathbf{C}, \mathcal{O}(idx))$
        $\mathcal{O} \leftarrow \mathcal{O} \setminus \mathcal{O}(idx)$
    end
end
```

Algorithm 5: Eight corner approach.

- There is a minimum distance between the quadcopter and newly detected obstacles. This minimum distance is determined by the maximum velocity of the drone and the maximum trajectory replanning time.

If one of the two conditions is not met, the drone will immediately land at its position and the program will be aborted. Otherwise, the trajectory replanning is executed.

5.5.4 Replanning

This block represents the trajectory replanning. This helps the UAV to still reach the desired goal. Figure 5.8 briefly depicts the overall workflow of the replanning procedure, mainly consisting of previously described algorithms.

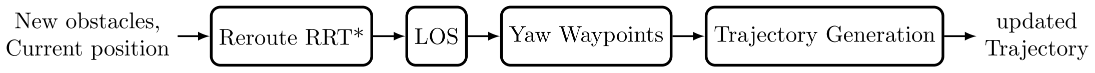


Figure 5.8: Detailed workflow of the *Replanning* block.

The first task is the rerouting of the consisting tree from RRT* considering newly detected obstacles, which has also been conducted in [46] in a similar way. The pseudocode of the rerouting step is given in Alg. 6.

Every edge $\mathcal{E}_v = (\mathsf{Parent}(\mathbf{v}), \mathbf{v})$ leading to each vertex $\mathbf{v}$ of the tree $\mathcal{G}$, except for the goal node $\mathbf{x}_{goal}$, is checked for freedom of collisions (line 2, Alg. 6). If one edge is interrupted by a new obstacle, it is marked to be removed from the tree (lines $3-4$, Alg. 6). If the considered vertex $\mathbf{v}$ is included by an obstacle, it is marked to be cropped from the tree since it is not reachable anymore (lines $6-8$, Alg. 6). Otherwise, an attempt to find a new parent node for vertex $\mathbf{v}$ is made, based on the first replanning step of the RRT* algorithm in Alg. 2 (lines $8-26$, Alg. 6). As no valid parent node is known, the minimum cost is set to a very high value in order to allow for the first collision free connection to be valid (lines $9-11$, Alg. 6). With $\mathcal{X}_{near}$, all nodes inside a ball of radius ρ centered at $\mathbf{v}$ are found (line 12, Alg.6). Each vertex $\mathbf{x}_{near} \in \mathcal{X}_{near}$ is then investigated, whether it yields a collision free and shorter path (lines $13-20$, Alg. 6). After all near vertices $\mathcal{X}_{near}$ have been checked and are valid, a collision free path could be found and the (repairing) edge $(\mathbf{x}_{min}, \mathbf{v})$ is added to the tree $\mathcal{G}$ (lines $21-22$, Alg. 6). When $\mathbf{v}$ is not reachable due to the new obstacle, the corresponding node will be marked to be removed from the tree (lines $23-25$, Alg. 6). In line 29 of Alg. 6, all lost connection-free nodes are removed from the set of nodes $\mathcal{V}$.

Once every node of the tree is examined, the tree $\mathcal{G}$ is collision free. To find the minimum-cost path from the estimated position after replanning to the goal, the estimated position is added to the tree $\mathcal{G}$. By traversing the tree from this vertex with the Parent function, a sequence of waypoints $\mathbf{wp}$ describing the shortest piecewise linear path to the goal is obtained. The following steps from Figure 5.8, e.g., LOS, yaw waypoints definition, and trajectory generation, are introduced in Section 5.3 and Section 5.4.

```
Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{O}, \rho$
Output: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
$\mathcal{V}_{lost} \leftarrow \{\}$
for $\mathbf{v} \in \mathcal{V} \setminus \{\mathbf{x}_{goal}\}$ do
    $\mathbf{x}_{parent} \leftarrow \mathsf{Parent}(\mathbf{v})$
    if not $\mathsf{CollisionFree}(\mathbf{x}_{parent}, \mathbf{v}, \mathcal{O})$ then
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \{(\mathbf{x}_{parent}, \mathbf{v}\}$
        if $\mathsf{InsideObstacle}(\mathbf{v}, \mathcal{O})$ then
            $\mathcal{V}_{lost} \leftarrow \mathcal{V}_{lost} \cup \{\mathbf{v}\}$
        else
            $c_{min} \leftarrow 1e15$
            $x_{min} \leftarrow []$
            $valid \leftarrow \mathsf{false}$
            $\mathcal{X}_{near} \leftarrow \mathsf{Near}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathbf{v}, \rho)$
            for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ do
                if $\mathsf{CollisionFree}(\mathbf{x}_{near}, \mathbf{v}, \mathcal{O}) \wedge \mathsf{Cost}(\mathbf{x}_{near}) +$
                                  $\mathsf{C}(\mathsf{Line}(\mathbf{x}_{near}, \mathbf{v})) < c_{min}$ then
                    $\mathbf{x}_{min} \leftarrow \mathbf{x}_{near}$
                    $c_{min} \leftarrow \mathsf{Cost}(\mathbf{x_{near}}) + \mathsf{C}(\mathsf{Line}(\mathbf{x}_{near}, \mathbf{v}))$
                    $valid \leftarrow true$
                end
            end
            if $valid$ then
                $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{x}_{min}, \mathbf{v})\}$
            else
                $\mathcal{V}_{lost} \leftarrow \mathcal{V}_{lost} \cup \{\mathbf{v}\}$
            end
        end
    end
end
$\mathcal{V} \leftarrow \mathcal{V} \setminus \mathcal{V}_{lost}$
```

Algorithm 6: Reroute RRT*.

6 Experimental Validation

In this chapter, the proposed trajectory (re)planning concept is evaluated in both simulation and experiments. Simulations are performed in MATLAB (R2021b, Version 9.11.0.1809720) on the Notebook with an Intel i7-10750H, 2.6 GHz and 32 GB RAM.

In simulations, the flight simulator in Gazebo and the PX4 autopilot program are executed on a different PC. This imitates the real-world scenario where the trajectory (re)planning is executed on a ground control station. The Gazebo simulator computes precise information, e.g., position, velocity, and orientation of all objects in the simulation including the quadcopter at any time instance. This information is published in ROS topics. For real life experiments, the Intel Aero RTF compute board runs the Linux Distribution Ubuntu 18.04 with ROS Kinetic. Similar to the simulation environment, the PX4 Firmware version 1.9 is used in the flight controller for a seamless sim-to-real transfer.

The first part of this chapter deals with the influence of various parameters that affect the result of the trajectory planning, specifically the path planning with RRT* and the replanning approach in Section 6.1. Next, the two presented obstacle detection approaches in Section 5.5.2 are evaluated utilizing depth camera data from a simulated flight in Section 6.2. Section 6.3 shows a simulated flight that contains the smooth replanning of the trajectory. Finally, real life experiments concerning online trajectory replanning are demonstrated in Section 6.4.

6.1 Influence of Parameters on Trajectory Planning

Since trajectory replanning is performed autonomously during flight, choosing a suitable set of parameters for the proposed algorithms is very important. In this section, the influence of the different parameter sets used in the trajectory planning is investigated. More specifically, we vary the steering distance ϵ, the rerouting radius ρ as well as the number of nodes *numNodes* of the RRT* algorithm and discuss their impact on the resulting tree structure. Furthermore, for a given set of waypoints obtained from RRT*, we examine the influence of the penalty weights w_i in the optimization problem (5.9) on the final trajectory of the quadcopter. All experiments in this section are executed in MATLAB on the GroundControl Station.

The parameters of the considered environment for the following investigation are listed in Table 6.1.

Parameter	Value [m]
x_{max}	10
y_{max}	10
z_{max}	5
start position	$\begin{bmatrix} 2 & 2 & 0.5 \end{bmatrix}$
target position	$\begin{bmatrix} 9 & 9 & 2.5 \end{bmatrix}$

Table 6.1: Environment parameters.

	Set 1	Set 2	Set 3	Set 4
$numNodes$	500	500	2000	500
ϵ	0.5 m	0.5 m	0.25 m	0.25 m
ρ	1.5 m	2.5 m	0.75 m	0.75 m

Table 6.2: Parameter sets of the RRT* algorithm.

6.1.1 Variation of RRT* Parameters

Four different sets of parameters are listed in Table 6.2. Monte Carlo simulations are performed with all sets of parameters.

In Figure 6.1, the obtained tree structures from the RRT* algorithm drastically vary among the sets of parameters. Specifically, for the parameter set number 4, the small number of nodes and the short steering distance ϵ provide a tree structure that only occupies the flight space to a small extent. Due to the low population and the short rerouting distance ρ, there are only a few main branches towards the target position. They may be cut off by new obstacles, which makes arriving at the destination impossible. Figure 6.1(3) depicts a tree with the same ϵ and ρ, however, with a significantly higher number of nodes compared to the set 2 and 3. This leads to an extraordinarily good exploration of the flight space. However, many turning paths due to the short rerouting distance ϵ are obtained. The runtime for creating and including so many nodes in the tree structure is significantly higher compared to other parameter sets. Since replanning has to be accomplished online, such a large number of nodes is not desirable. Figure 6.1(1) and (2) show a tree with the same number of nodes and steering distance, but with different ϵ. These sets result in the same proper exploration of the flight space, which is benefitial for the avoidance of new obstacles. Furthermore, it is clear that subsequent edges in the

	Set 1	Set 2	Set 3	Set 4
avg. run time [s]	0.743	0.764	10.411	0.725

Table 6.3: Runtimes of parameter variation of the RRT* algorithm.

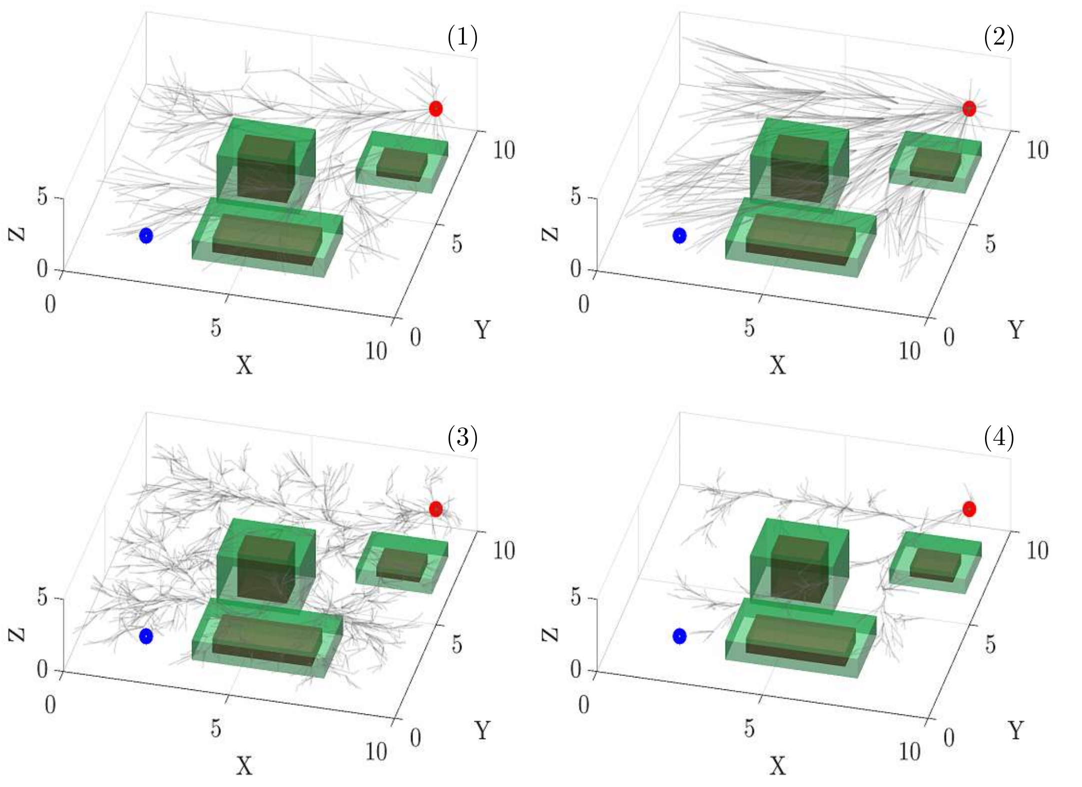


Figure 6.1: Evolutions of RRT* algorithm with different sets of parameters. The number on the top right of each subfigure indicates the set number from Table 6.2. The red and blue dot indicate the target and start position, respectively.

Parameter	Value
$numNodes$	500
ϵ	0.7 m
ρ	2.1 m

Table 6.4: Parameters of the RRT* Rerouting example.

Parameter	Initial tree	Rerouted tree
$numNodes$	500	483
number of waypoints	8	9
path length	10.46 m	12.11 m

Table 6.5: Result of the RRT* Rerouting example.

set 2 appear much more straightforward, which is well suited for path planning of the quadcopter. A path straightening is subsequently accomplished with the Line of Sight (LOS) optimization, which solves the problem of a poorly chosen rerouting parameter ρ. The difference in run times of parameter sets 1 and 2 is negligible for this small number of vertices. In conclusion, to create an appropriate population of the nodes in the flight space, the number of nodes and the steering distance ϵ have to be investigated thoroughly. In addition, the rerouting parameter ρ leads to longer edges of the tree. However, the LOS optimization removes redundant vertices of the obtained path afterwards.

6.1.2 RRT* Rerouting

This section gives an overview of the rerouting procedure, which processes a sub RRT* algorithm to detour the initial trajectory in case of the appearance of new obstacles. The same environmental setup as in the previous examples is reused with two known obstacles. If a new obstacle is detected that intersects the initial path, the rerouting algorithm is executed.

All parameters for this example of RRT* rerouting are given in Table 6.4 including the parameters ($numNodes$,ϵ,ρ) for the RRT* algorithm in Alg. 1,2, to generate the initial tree, and for the RRT* rerouting algorithm in Alg. 6. The search radius ρ of the rerouting algorithm is chosen to be the same as for the basic RRT* algorithm, because the RRT* rerouting algorithm is mainly based on the first rerouting step in lines $5-14$, Alg. 2.

Note that the run time of the RRT* algorithm heavily depends on the number of nodes in the tree. Therefore, the algorithm is executed multiple times with a different number of nodes in the inital tree, leading to the average run times listed in Table 6.6.

Figure 6.2(a) depicts the initial tree with the shortest collision free path. After detecting a larger object, the rerouting procedure is executed. The rerouted path is illustrated in

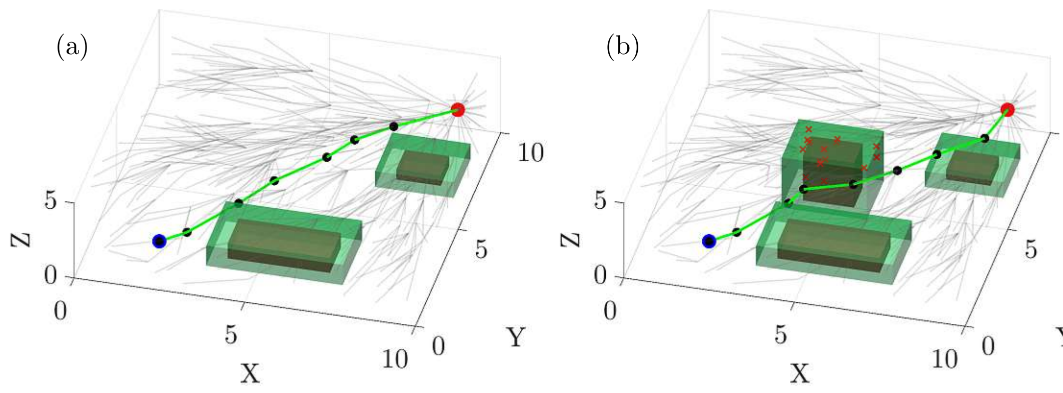


Figure 6.2: Example for the RRT* rerouting. The left picture depicts the initial tree, the right shows the rerouted tree with the new obstacle. The green lines indicate the shortest collision free path from the start to the target point. The black nodes refer to the waypoints. The red crosses mark the lost vertices.

numNodes	100	250	500	1000	2000
avg. run time [ms]	10.5	46.6	136.0	416.0	1942

Table 6.6: Run times of RRT* rerouting for different number of nodes in the tree.

Figure 6.2(b). The main findings of this example are listed below:

- When obstacles are recognized, tree nodes occupied by these new obstacles are discarded. This reduces the total number of nodes in the tree. For this reason, when parameterizing the RRT* algorithm, we have to ensure that the entire flight space is evenly populated in order to avoid possible dead ends.
- In Table 6.5, the increase of the number of waypoints leads to an increase in the corresponding path lengths. Additionally, more waypoints lead to an increased set of optimization variables, which implies a higher computation time for the optimization. These consequences are partially mitigated by the LOS optimization afterwards.
- Another important statement that can be made is the high run time dependence of replanning on the number of nodes in the tree, see Table 6.6. Due to the need to check every connection in the tree, and the fact that in the event of a cut edge to a specific vertex more surrounding nodes have to be checked for possible connections the necessary run time increases exponentially. This fact suggests building a tree with a few nodes, but this undermines the probability of finding a collision free and short path.

In conclusion, the parameterization of the RRT* algorithm, especially the choice of the number of nodes, is a trade-off decision which depends on the scenario and application. For instance, in an application with the need for aggressive trajectories and fast trajectory replanning, a smaller number of nodes can be chosen. If one needs a higher probability to find the shortest path for different sizes of objects, a higher number of nodes is helpful.

6.1.3 Variation of Trajectory Generation Weights

The tree obtained by using RRT* is used to create a dynamically feasible trajectory for the quadcopter. The coefficients of the polynomial of the trajectory are computed with quadratic programming. In this subsection, the effect of different user-defined weights of the optimization problem (5.9) is investigated. Two different sets of user-defined weights $\mathbf{w}_{pos}$, with $N = 4$, and $\mathbf{w}_{yaw}$, with $N = 2$, according to (5.9) are given in Table 6.7. The two time evolutions of the trajectories are depicted in Figure 6.3. In addition, a three-dimensional representation of the position trajectories is visualized in Figure 6.4.

As can be seen in Figure 6.3 and Figure 6.4, the two trajectories differ considerably despite the same waypoints and segment times due to the different user-defined weights in the optimization. The segment times are defined by the 15-step time-optimal trajectory generation algorithm according to Section 5.4 utilizing a set of dynamic constraints of the quadcopter in Table 6.11. Set 1 in Table 6.7, with lower value of the low derivative orders and higher value of the high derivative orders, leads to a curved and smooth trajectory, see Figure 6.3(a). However, it has a high deviation from the collision free path, which

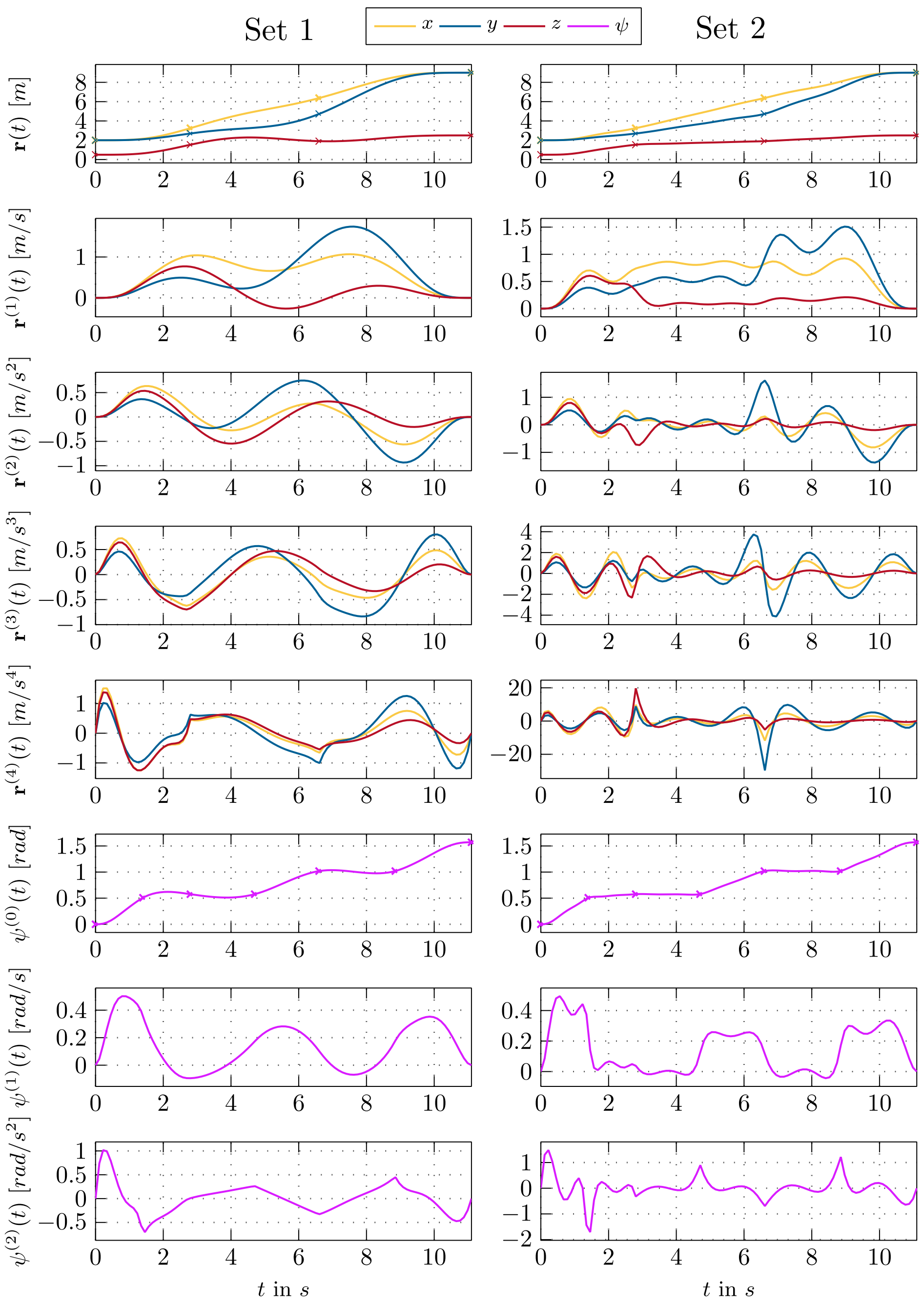


Figure 6.3: Different penalizing weights for the position and yaw trajectory for equal waypoints, indicated with marks. The left shows the obtained trajectory for parameter set 1, the right depicts set 2.

	Set 1	Set 2
$\mathbf{w}_{pos}$	$\begin{bmatrix}1 & 0 & 0 & 10\end{bmatrix}$	$\begin{bmatrix}1e5 & 0 & 0 & 10\end{bmatrix}$
$\mathbf{w}_{yaw}$	$\begin{bmatrix}0.1 & 1e3\end{bmatrix}$	$\begin{bmatrix}1e4 & 100\end{bmatrix}$

Table 6.7: Two sets of penalizing weights for the position and yaw trajectory.

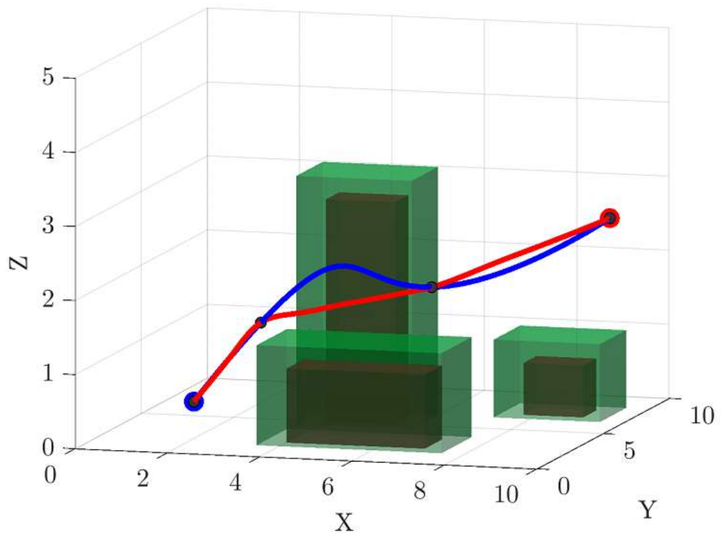


Figure 6.4: Position trajectory for different sets of weights from Table 6.7. The blue and red line show the trajectories computed with parameter set 1 and parameter set 2, respectively.

makes the trajectory potentially unsafe, see Figure 6.4. Thus, further verification would be required to ensure that the trajectory is collision free. Note that if the yaw angle varies significantly, it is not guaranteed that the drone will be oriented in the direction of flight to detect new objects. In general, higher weight values for the higher derivative orders as in set 1, show a smoother trajectory.

On the other hand, with set 2, a strong guidance of the position and facing direction along the path is achieved, which is desirable in the context of safety, see Figure 6.4. This increases the probability of recognizing dangerous objects during flight. However, the drawback is to have aggressive inputs over time, which could be infeasible for the system due to actuator constraints. At this point, it should be noted that depending on the application the shape of the trajectory can be determined with the choice of the penalizing weights in (5.9). A trade-off regarding the deviation from the collision free straight lined path and smoothness of the trajectory has to be made with respect to dynamic constraints.

6.2 Simulation - Comparison Obstacle Detection Approaches

This section evaluates and compares the two proposed approaches for detecting new obstacles from Section 5.5.2. The first approach is based on point cloud processing methods provided by the computer vision toolbox from MATLAB. The second approach relies mainly on the GJK algorithm. The depth camera data used in this section are taken from Gazebo. The position, velocity, and orientation data are obtained from the PX4 estimator. The status of the estimator is based on various sensor data, see Section 3.1.3. This leads to slight deviations between the real and the estimated positions and orientations. Since an external vision system is not available for every application, e.g. exploring a tunnel or in disaster environments, the object detection should be robust against measurement and estimation inaccuracies.

To evaluate the two obstacle detection approaches, the scenery in Figure 6.5 is used. This scenario simulates a workshop or production hall with shelves or cabinets. The new obstacle is illustrated in red color. It should be noted that no walls and ceilings are considered in the simulation, since they are cut off during the object detection procedure. Due to the large objects, there will be a lot of depth points per cluster. The parameter settings for both methods are listed in Table 6.8. The depth camera sent 62 different point clouds with a resolution of 320 x 240 during the flight experiment, which were processed with both approaches. Figure 6.8 depicts a box plot of the run times of the obstacle detection process, the statistics is given in Table 6.9.

An insight into the evolution of the captured environments is given in Figure 6.6. The left column shows the point cloud representation of the current known environment in purple, whereas newly detected clusters are visualized in green. The middle column depicts the obtained environment with the point cloud approach, the right one results

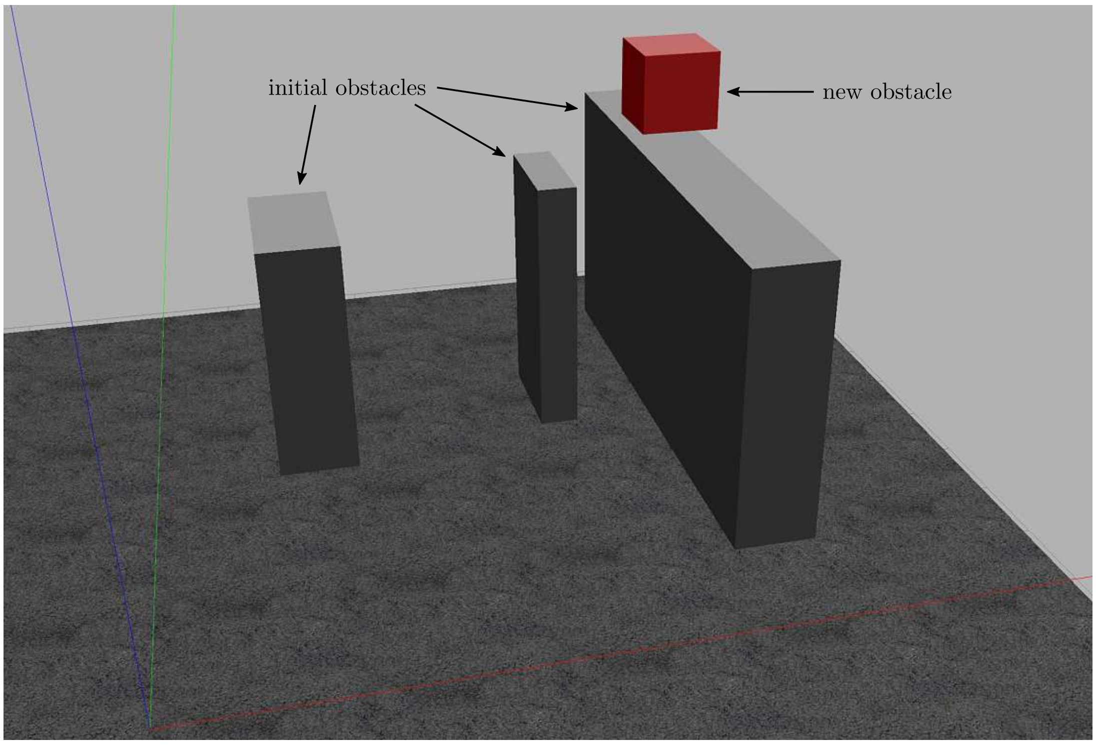


Figure 6.5: Environment of the simulation flight. The three bigger obstacles on the ground are considered during trajectory planning, the unknown red object intersects with the shortest path from the start to the target position.

Parameter	Value
ω_{max}	1 rad/s
voxel size	8 cm
min. cluster points	8
$thresh_{pcsegdist}$	0.15 m
$thresh_{rmse}$	0.15 m

Table 6.8: Parameters for the obstacle detection approaches for comparison.

	point-cloud	eight-corner
avg. run time [ms]	160.6	131.4
median [ms]	143.2	123.8
min. run time [ms]	120.0	111.8
max. run time [ms]	368.3	194.8
75^{th} percentile [ms]	183.2	139.7
outliers	2	5

Table 6.9: Run times statistics of the obstacle detection approaches.

from the eight-corner approach on the same data set. The inital environment with all known obstacles is illustrated at the top of column two und and three. The environments are updated from top to bottom over time. The magenta cross and arrow indicate the position and heading of the drone at the particular time instance.

The two methods give different results due to the inaccuracies of the position estimation. In Figure 6.6, the side surface of one object is considered with both approaches as a new object. The depth camera points deviate distinctly after the transformation with the inaccurate pose information from the real position of the object. Obviously, the $rsme$ value of the comparison of each cluster point (green) to its closest point from the known obstacle (purple) is bigger than the treshold value, yielding a new obstacle with the point cloud approach, see Figure 6.6 (2b). The point clouds of the known world and the cluster describing the new obstacle are merged, see Subfigure (3a). Considering the eight-corner approach in Figure 6.6 (2c), the closest obstacle is merged together with the cluster, yielding one bigger, outer-left obstacle. Subfigure (3a) depicts the first snippet of an unknown obstacle, captured by the depth camera, in green. The point cloud approach yields one small yellow obstacle in Figure 6.6 (3b), wheres the eight-corner approach merges the small cluster with the closest obstacle in Subfigure (3c). The same principle applies in Figure 6.6(4a). The full front of the new obstacle is detected with the depth camera. This leads to a separate, bigger obstacle with the point cloud approach Figure 6.6(4b). The previously detected yellow obstacle, see Figure 6.6(3b), shows a too big difference to the green depth point cluster in Subfigure (4a). On the other side, the eight-corner method merges both, as they are intersecting, see Figure 6.6(4c). The final configurations of both approaches are depicted in Figure 6.7.

The main differences of both methods are summarized in the following. The point cloud approach does not alter the size of an existing obstacle. If a cluster, obtained from depth camera data, happens to deviate more than a predefined measure from the closest known obstacle, it will be considered as a separate new obstacle. This may lead to redundant obstacle detections, since obstacles are from the same real object. Note that the unnecessary higher number of modeled obstacles slow down the collision-check and the replanning procedure in later processes. On the other hand, the eight-corner

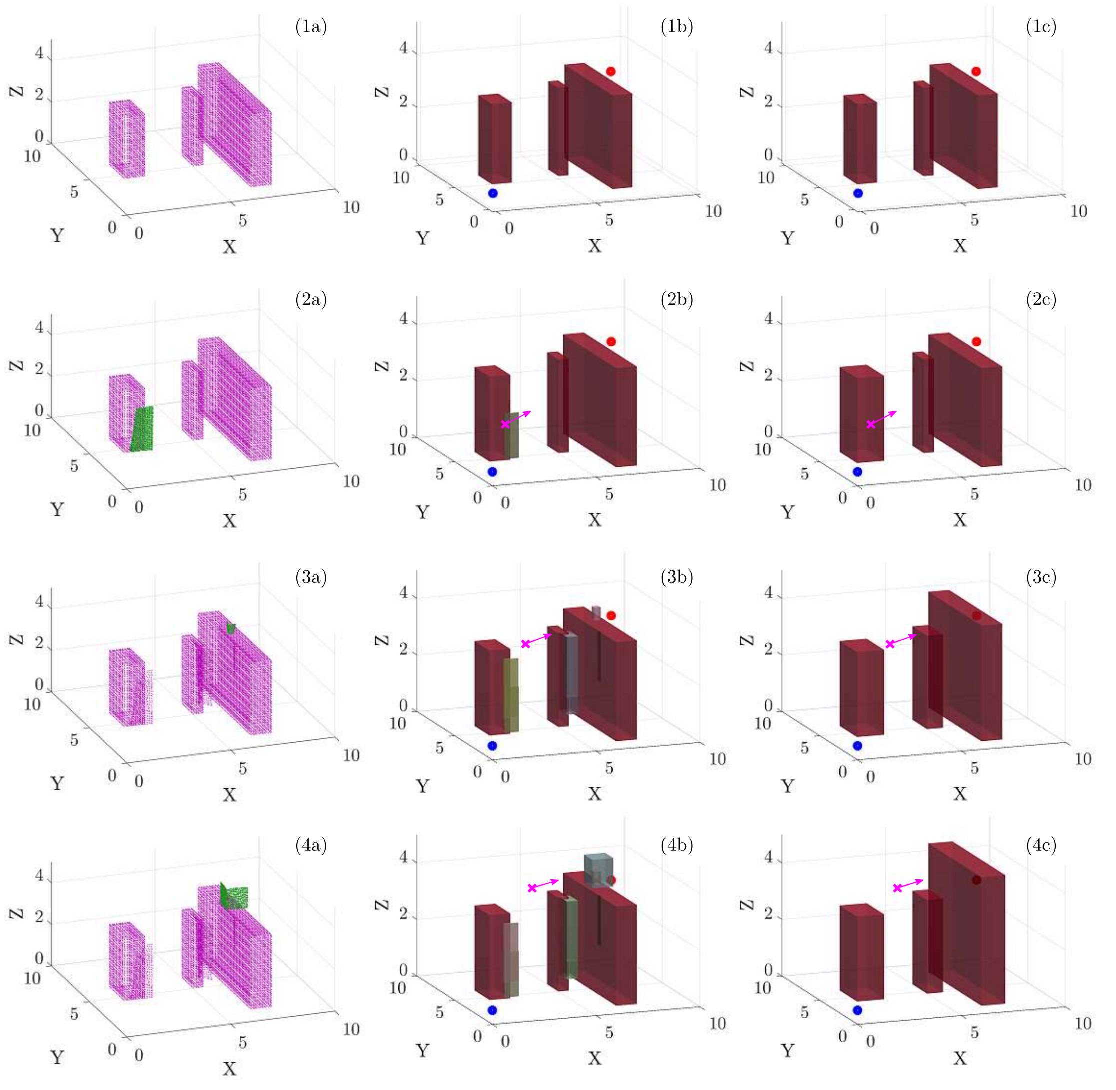


Figure 6.6: Evolution of the environment with two different obstacle detection approaches. The column (a) shows the current environment in purple and the new cluster in green as point cloud, column (b) depicts the obtained obstacles from the point cloud based approach. The column (c) is obtained with the eight-corner approach. The position and facing direction is indicated in purple.

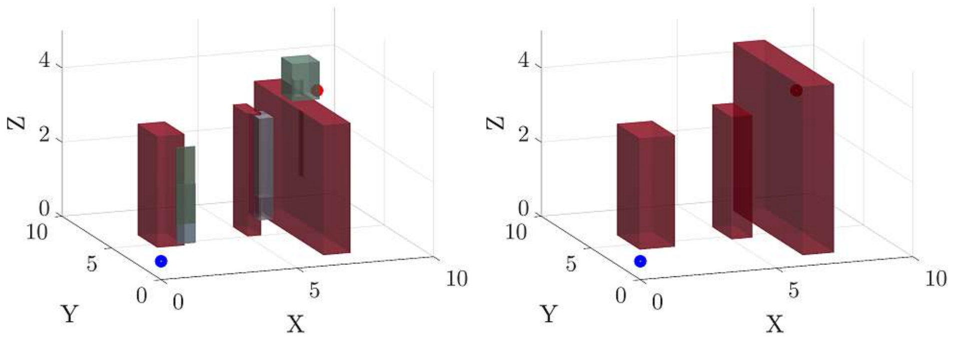


Figure 6.7: Final environments of the point cloud and eight-corner approach.

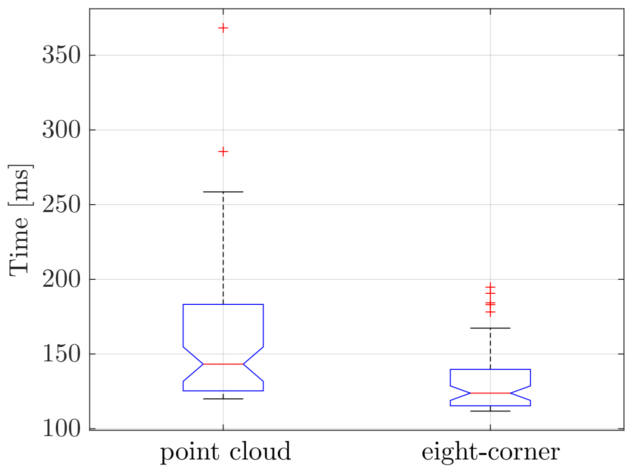


Figure 6.8: Box plot of run times of obstacle detection approaches.

approach considers only the hit box of the closest known obstacle and cluster. If both intersect or are very close, they are merged together to one bigger box. This could lead to clearly oversized models of obstacles, especially if the real obstacle may be non-convex. Furthermore, close obstacles, which are in fact further apart than the segmentation distance, may be merged to one obstacle due to poor position estimation. This may lead to very unfavorable constellations, such as the ones in Figure 6.6 (4c). This is definitely a drawback compared to the point cloud approach, which allows to describe obstacles in more detail.

In Figure 6.8, the run times of the two approaches are depicted. The eight-corner approach is faster than the point cloud approach. The median of the eight-corner method is approximately the minimum time of the point cloud approach. Although the box plot of the eight-corner approach has 5 outliers, the absolute span of the distribution of 83 ms is also much smaller, compared to 248 ms of the point cloud method.

The smaller number of final obstacles outweighs the disadvantage of a possible false fusion of multiple objects when having a poor position estimation. Since we use OptiTrack in the real-life test for precise position and orientation measurements, the eight-corner method is used to extract new obstacles in the following flight experiments.

6.3 Simulation - Replanning

The previous chapters show the necessary building blocks for the trajectory generation, the replanning if a new obstacle appears in the flight space, and the methods for detecting new obstacles. The following simulation combines all modules to demonstrate the trajectory (re)planning strategy. In this simulation, a flight is performed in a larger environment with known obstacles using Gazebo. The initial trajectory is planned to avoid the three known objects on the ground. However, a fourth unknown object appears in the flight space, which requires a replanning of the trajectory.

6.3.1 Environment and Parameters

The scenery of the simulation flight is illustrated in Figure 6.5 with the environment parameters in Table 6.10. The parameter set for the RRT* and rerouting algorithm is given in Table 6.12, system constraints used for estimating segment times are presented in Table 6.11. Penalizing weights for the trajectory optimization (5.9) and parameters for the obstacle detection with the eight-corner approach are listed in Table 6.13 and Table 6.14, respectively.

6.3.2 Results

The trajectory planning process is depicted in Figure 6.9. Figure 6.9(a) shows a simulated 3D scenario, while Figure 6.9(b) presents the initial position trajectory in yellow. After

Parameter	Value $[m]$
x_{max}	10
y_{max}	10
z_{max}	5
inflation distance (x,y)	0.5
inflation distance (z)	0.35

Table 6.10: Flight environment parameters of the simulation experiment.

Parameter	Value
v_{max}	1 m/s
a_{max}	5 m/s
j_{max}	8 m/s
s_{max}	20 m/s

Table 6.11: Dynamic constraints for the trajectory planning of the simulation experiment.

Parameter	Value
$numNodes$	500
ϵ	1 m
ρ	3 m

Table 6.12: Parameters of the RRT* algorithm of the simulation experiment.

	weights
$\mathbf{w}_{pos}$	$\begin{bmatrix} 1e4 & 0 & 0 & 10 \end{bmatrix}$
$\mathbf{w}_{yaw}$	$\begin{bmatrix} 1e3 & 10 \end{bmatrix}$

Table 6.13: Penalizing weights for the trajectory generation of the simulation experiment.

Parameter	Value
ω_{max}	1 rad/s
voxel size	8 cm
min. cluster points	8
$thresh_{pcsegdist}$	0.15 m

Table 6.14: Parameters for the eight-corner approach of the simulation experiment.

5.66 s in Subfigure Figure 6.9(c), the unknown obstacle is detected by the eight-corner approach. Thereby, the trajectory replanning is activated. Note that the newly detected obstacle is inflated due to safety reasons. In Figure 6.9(d), the replanned trajectory is depicted in green. The time evolution of the computed trajectory is given in Figure 6.10.

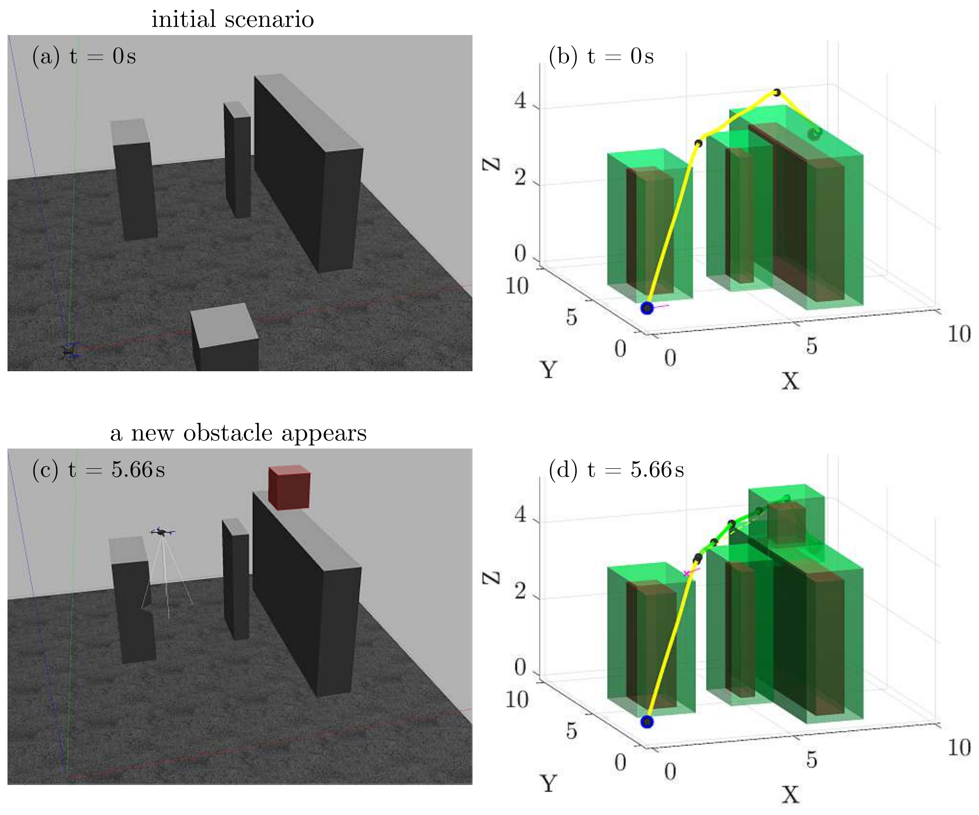


Figure 6.9: Trajectory planning process of the simulation experiment. Snapshots of the simulation at start and at replanning can be seen in the left, the planned trajectories with obstacles in the right column. The first row depicts the initial setting at $t = 0\,\mathrm{s}$, the second row the updated environment and the replanned trajectory at $t = 5.66\,\mathrm{s}$

The overall trajectory is composed of two partial trajectories, which are divided by a dashed vertical line and steadily merge into each other.

The tracking error of the trajectory is defined as

$$\mathbf{e_r}(t) = \mathbf{r}_{des}(t) - \mathbf{r}(t), \qquad e_\psi(t) = \psi_{des}(t) - \psi(t), \tag{6.1}$$

from the desired position $\mathbf{r}_{des}(t)$ and the real position $\mathbf{r}(t)$ at time t, the same applies to the yaw angle ψ. This simulation shows a successful test flight. The drone started

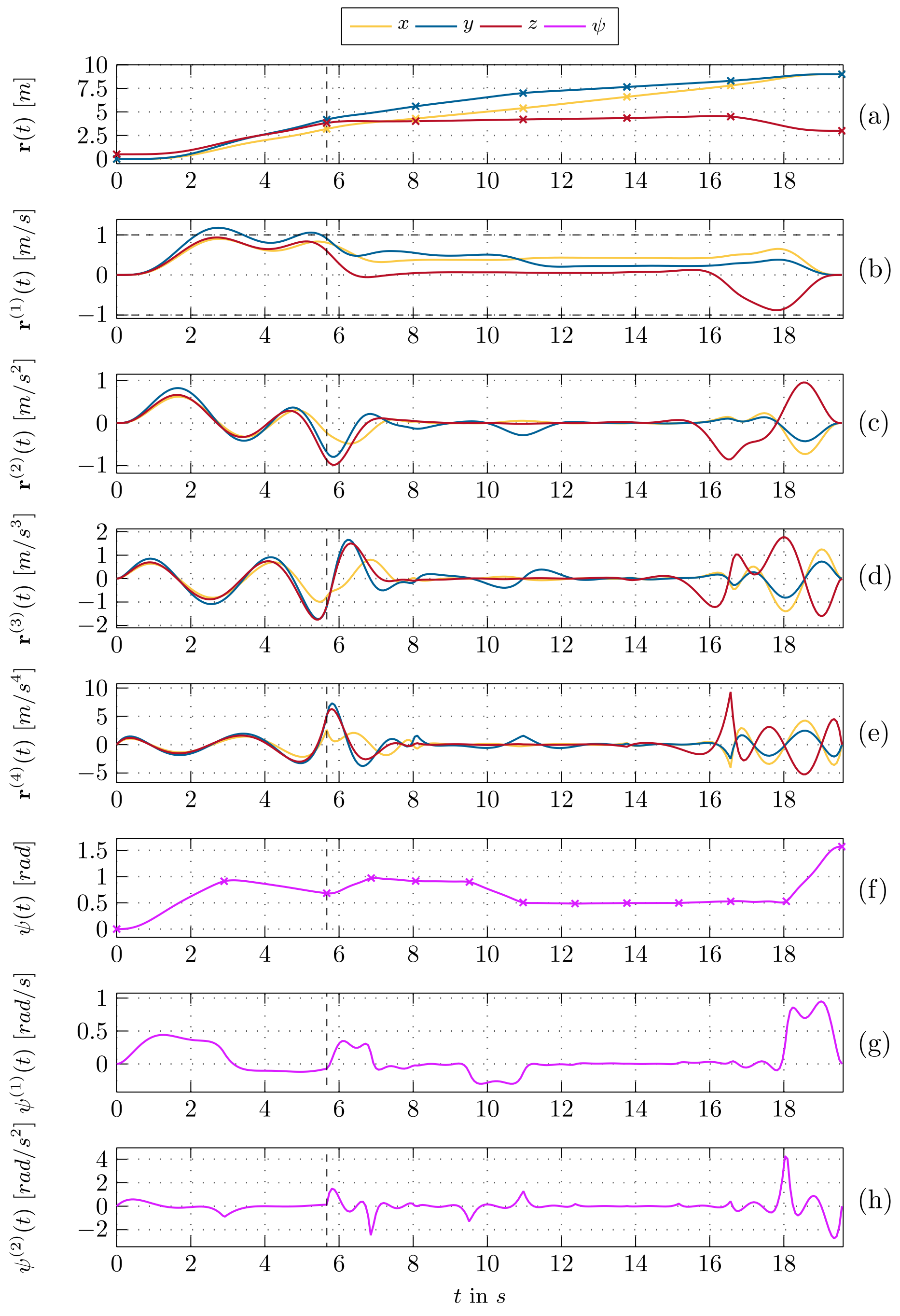


Figure 6.10: Resulting trajectory of the simulated flight. The vertical dashed line at 5.66 s indicates a trajectory replanning, the small marks show waypoints.

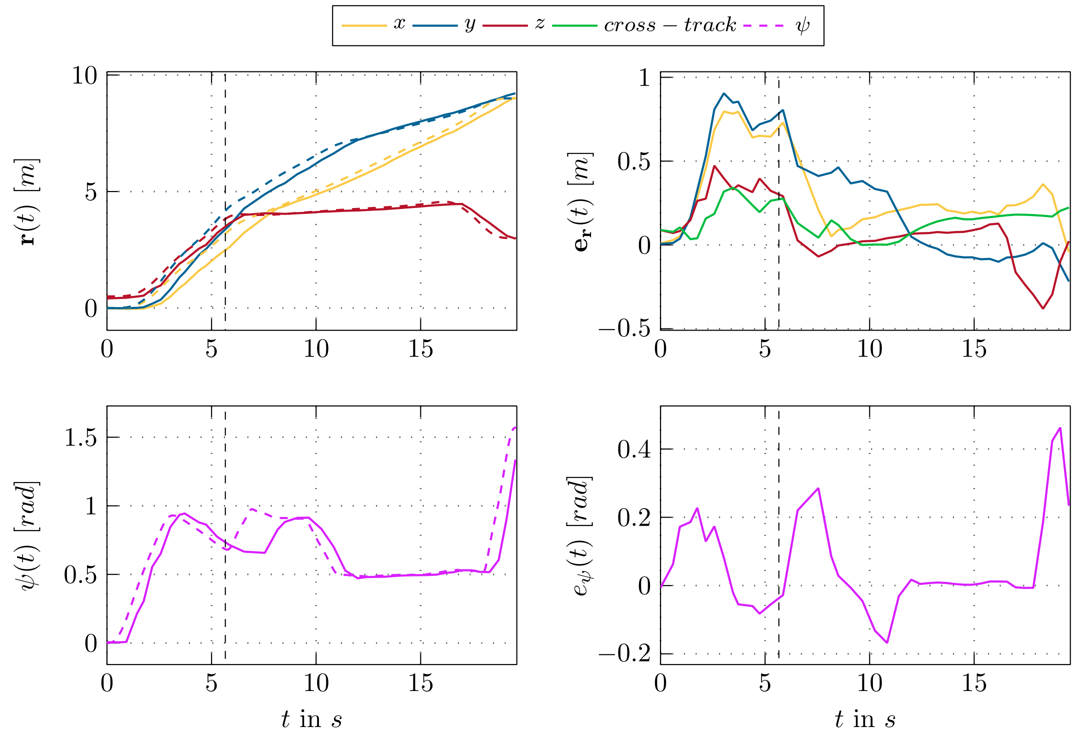


Figure 6.11: Trajectory tracking error of the simulated flight. The dashed lines depict the desired trajectory, whereas the full lines indicate the ground truth data. The green line shows the cross-track error stated in (6.2), neglecting the tangential component of the trajectory tracking error.

from the bottom left corner of the flight space, travelled along the initial trajectory and the obstacle detection recognized the unknown object in the mid-flight. Therefore, a trajectory replanning was activated, leading to a safe trajectory towards the target, i.e. the green path in Figure 6.9(d).

In Figure 6.10, the time evolution of the trajectory is visualized. When considering the higher derivatives of the position $\mathbf{r}(t)$ and the yaw angle $\psi(t)$, we see a steady transition at the time instance of replanning, i.e., at $t = 5.66\,\mathrm{s}$ in Figure 6.10(e) and Figure 6.10(h). Another noticeable feature are the peaks that occur at the waypoints in the higher derivatives, which are within the defined limits. In Figure 6.10(b), the desired maximum velocity is slightly surpassed in y-direction in the first part of the trajectory, which is possible due to the polynomial-based approach.

The trajectory tracking errors for the position $\mathbf{r}(t)$ and the yaw angle $\psi(t)$ according to (6.1) are shown in Figure 6.11. It can be stated that the position tracking errors in each dimension are not satisfactory, since they are larger than the inflation distance of the obstacles. The position and orientation control of the drone is based on P and PID controllers, see Section 3.3. The drone is always slightly lagging behind in the tangential direction of the trajectory, which results in a larger tracking error at certain times t. Therefore, the cross-track error $e_{cross-track}$ is introduced in [15] to compute the closest point on the trajectory $\mathbf{r}_{des}$ to the current position $\mathbf{r}$. At each time instance of the position trajectory, a unit tangent vector $\mathbf{t}$ is defined. For $\mathbf{t}$, a normal vector $\mathbf{n}$ and a binormal vector $\mathbf{b}$ can be determined. The cross-track error reads as

$$e_{cross-track} = \|((\mathbf{r}_{des} - \mathbf{r}) \cdot \mathbf{n})\mathbf{n} + ((\mathbf{r}_{des} - \mathbf{r}) \cdot \mathbf{b})\mathbf{b}\|. \tag{6.2}$$

Here, the position error in tangential direction is ignored, since we are more concerned about reducing the cross-track error rather than the error in the tangential direction of the trajectory [15]. As mentioned above, when neglecting the tangential tracking error, a maximum cross-track error of 29 cm after approximately $t = 3\,\mathrm{s}$ is obtained, which is also slightly too high. Moreover, in the worst case, it can mean a collision with an object. This can be attributed to poor position estimation and high speeds of the quadcopter at that time and the controller architecture. The yaw angle error of the drone peaks at the very end, where a sharp turn towards the end-facing direction happens. However, it stays within reasonable bounds to perceive the environment. Also, right after rerouting, a rather long time of approximately 2 s went past, until the UAV orients towards the desired direction. The tracking errors may be improved by lower trajectory weights, longer segments times or tuning the PID-control parameters more aggressively. Since the main focus of this thesis is on the (re)planning method, no further efforts were made to reduce the trajectory tracking error by improving the existing trajectory tracking controller of PX4 [40].

Parameter	Value $[m]$
x_{max}	3.5
y_{max}	1.3
z_{max}	1.6
inflation distance (x,y)	0.5
inflation distance (z)	0.35
start position	$\begin{bmatrix} 0.3 & 0 & 1 \end{bmatrix}$
target position	$\begin{bmatrix} 3.3 & -0.25 & 0.5 \end{bmatrix}$

Table 6.15: Flight environment parameters for the real-life experiment.

6.4 Experiment - Replanning

Simulations presented in the previous section show the functionality of the trajectory planning and obstacle avoidance. For safety reasons, trajectory constraints are tightened, which lead to slower and less aggressive trajectories. This allows the operator in case of a failure to instantaneously initiate the quadcopter to hover or land via RC. In this scenario, a flight environment with one fixed obstacle on the ground is set up, which is not considered when planning an initial trajectory. Since this obstacle is located on the line of sight between start and target position, a first replanning step is necessary. After the UAV started to follow the trajectory, the human operator steps into the planned path to force the trajectory replanner to compute a collision free trajectory.

6.4.1 Environment and Parameters

The experiment is conducted inside a laboratory, shown in Figure 6.12, with a flight space of 3.5 m x 1.3 m x 1.6 m. Note that safety margins from surrounding obstacles are already considered. Each dimension of the detected obstacles will be inflated with values from Table 6.15 in all directions. An OptiTrack motion capture system with six cameras is used to localize the drone. These cameras are placed around the flight space such that each camera captures the maximum possible area. In order to increase safety in case of failure and to avoid damage, safety nets were installed. The parameter set for the RRT* algorithm and the rerouting is given in Table 6.16, system state constraints used for estimating segment times are presented in Table 6.17. Penalizing weights for the trajectory optimization (5.9) and parameters for obstacle detection with the eight-corner approach are listed in Table 6.18 and Table 6.19.

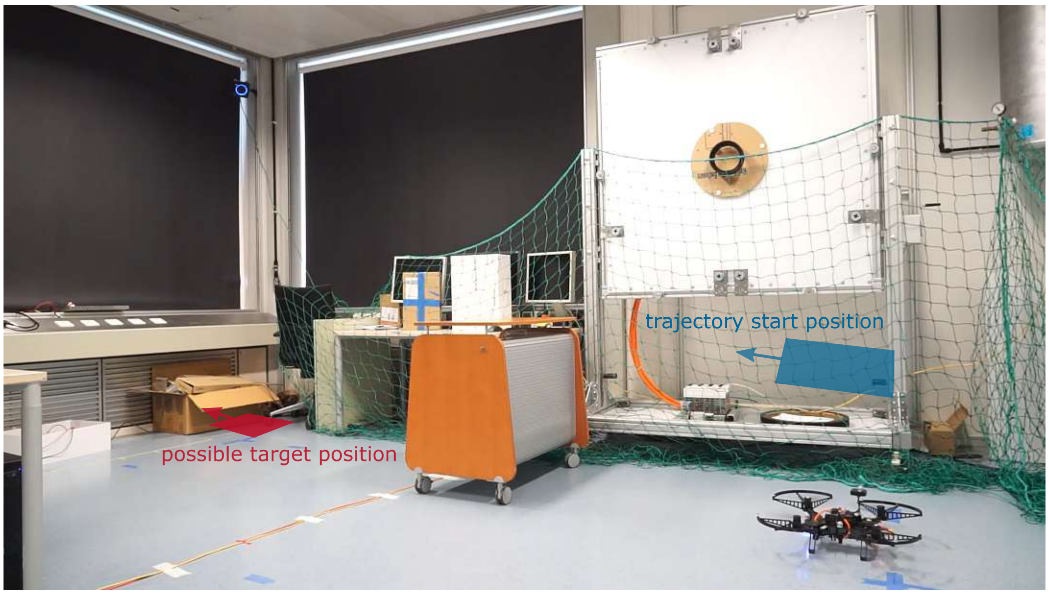


Figure 6.12: Real-life experiments are executed in this laboratory. The blue marks on the floor indicate the boarders of the flight space. Flights are conducted with various obstacles.

Parameter	Value
$numNodes$	500
ϵ	50 cm
ρ	150 cm

Table 6.16: Parameters of the RRT* algorithm for the real-life experiment.

Parameter	Value
v_{max}	0.5 m/s
a_{max}	5 m/s
j_{max}	8 m/s
s_{max}	10 m/s

Table 6.17: Dynamic constraints for trajectory planning for the real-life experiment.

	weights
$\mathbf{w}_{pos}$	$\begin{bmatrix} 1e4 & 0 & 0 & 5 \end{bmatrix}$
$\mathbf{w}_{yaw}$	$\begin{bmatrix} 1e3 & 10 \end{bmatrix}$

Table 6.18: Penalizing weights of trajectory generation of the real-life experiment.

Parameter	Value
ω_{max}	0.5 rad/s
voxel size	5 cm
min. cluster points	8
$thresh_{pcsegdist}$	0.15 m

Table 6.19: Parameters of the eight-corner approach for the real-life experiment.

6.4.2 Results

The trajectory planning process is visualized in Figure 6.13. The dashed yellow line indicates the initial trajectory for an obstacle-free scenario. In the very first step, the fixed obstacle is detected. Immediately the trajectory replanning is activated, which yields the trajectory in green color in Figure 6.13(b). A short time after the quadcopter started to follow the trajectory in green, another unknown obstacle, i.e. the operator, interferes the planned flight path and is partly detected. This replanning step yields the trajectory in blue in Figure 6.13(d). The final trajectory in red is obtained after merging depth camera data to the correct size of the human in Figure 6.13(e) and (f).

A time evolution of the obtained trajectory is given in Figure 6.14. The overall trajectory is composed of four partial trajectories, which are divided by dashed vertical lines and steadily merge into each other.

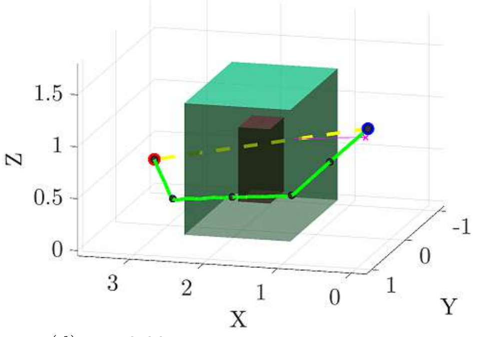
The tracking error of the trajectory is depicted in Figure 6.15. The tracking error in (6.1) results from the desired position and the real position at time t. The same procedure is applied to the yaw trajectory. The cross-track error according to (6.2) is depicted in green, which neglects the error in tangential direction.

The real test proves the performance of the proposed trajectory planning and the obstacle avoidance method. After detecting the fixed obstacle in the middle of the flight space in the first depth camera frame, the trajectory replanning was activated. Since the quadcopter was not in motion at the very beginning of the experiment, the replanned trajectory, depicted in green color in Figure 6.13, almost starts from the same position as the initial one in yellow color. After the UAV started to follow the trajectory, the operator stepped into the planned path, yielding another replanned trajectory in blue. The operator was not fully detected, therefore, after reducing the angular speed following a sharp turn, the full size of the operator in the field of view of the depth camera was perceived, leading to a necessary third replanning step that creates the final trajectory towards the target position.

In Figure 6.14, it can be seen that all derivatives of each flat output are continuous at replanning times, guaranteeing continuous inputs to the system. Also the position trajectory stays within reasonable bounds for all derivatives. In the time between 3.33 s and 5 s, a very sharp maneuver in yaw happens, the drone has to turn around its own axis in a very short time to be aligned with the flight direction again. This leads to a high yaw

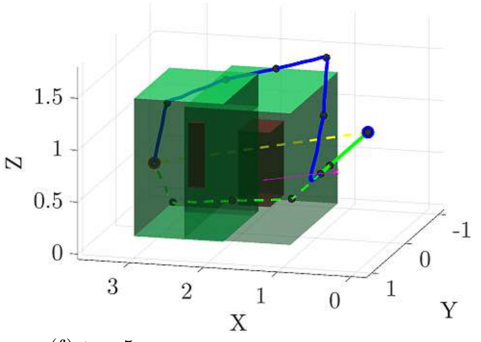
(a) t = 0.33 s

(b) t = 0.33 s

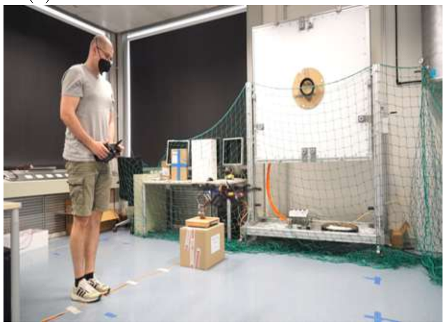


(c) t = 3.33 s

(d) t = 3.33 s



(e) t = 5 s



(f) t = 5 s

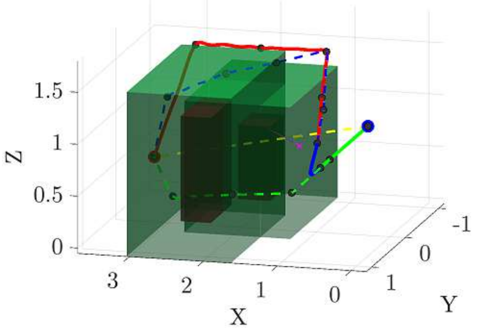


Figure 6.13: Trajectory planning process of the real-life experiment. Snapshots of the experiments at replanning timestamps can be seen in the left, the planned trajectories with obstacles in the right column. The first row depicts the planning process at $t = 0.33\,\mathrm{s}$, the second at $t = 3.33\,\mathrm{s}$ and the last one at $t = 5\,\mathrm{s}$.

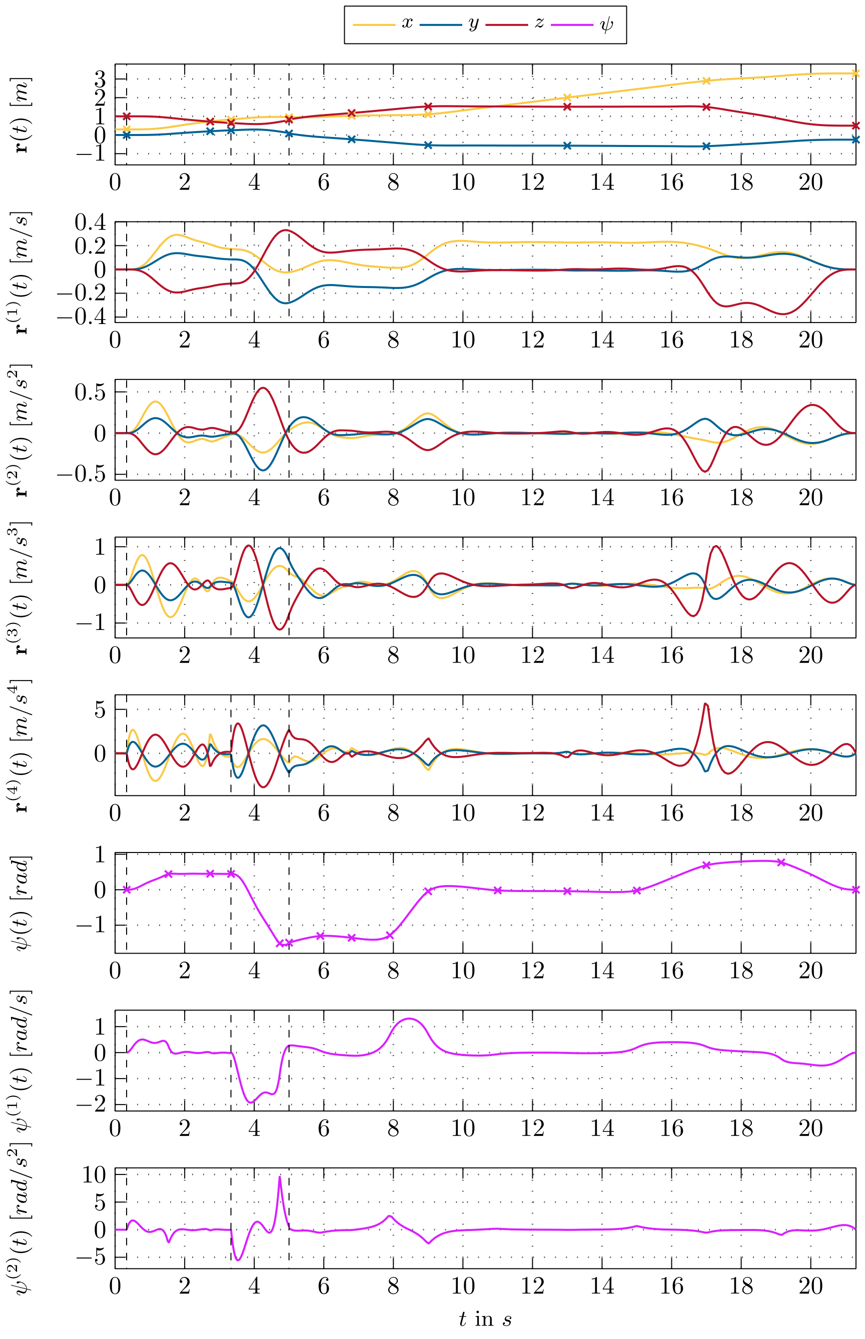


Figure 6.14: Resulting trajectory of the real-life experiment. The vertical dashed lines at 0.33 s, 3.33 s and 5 s indicate a trajectory replanning, the small marks show waypoints.

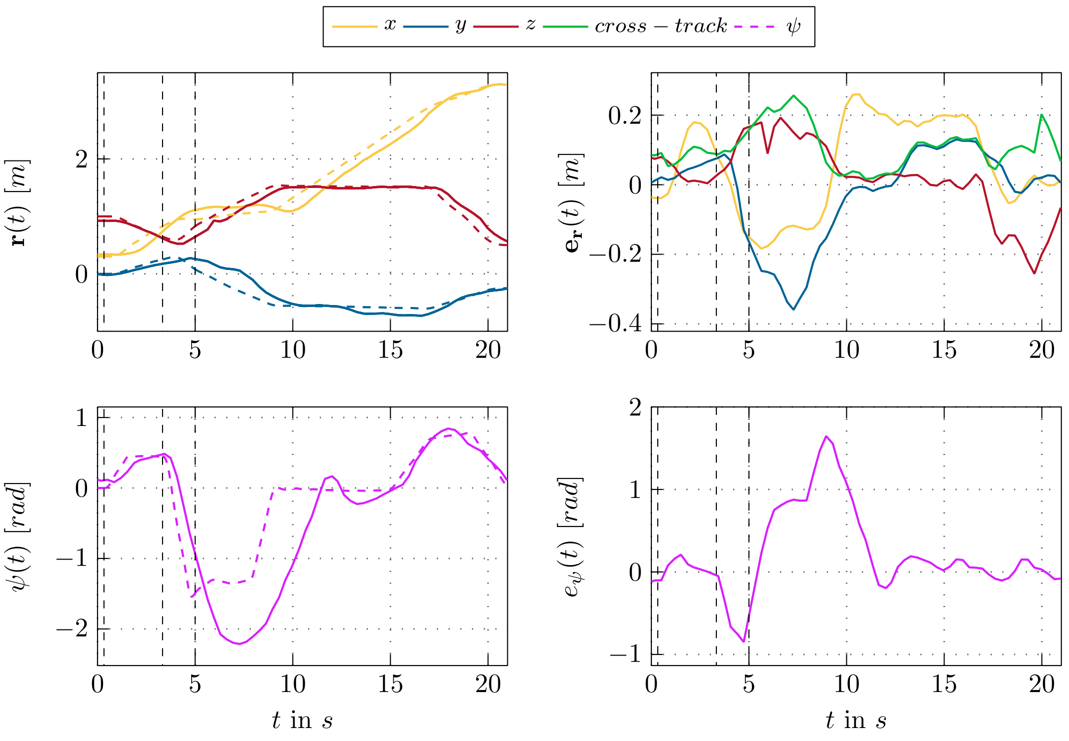


Figure 6.15: Trajectory tracking error of the real-life experiment. The dashed lines depict the desired trajectory, whereas the full lines indicate the ground truth data. The green line shows the cross-track error stated in (6.2), neglecting the tangential component of the trajectory tracking error.

rate and sharp peaks in the angular acceleration. As the yaw angle is not included in the path cost in the RRT* and the segment time estimation for the trajectory optimization (5.9), the yaw trajectory could be very aggressive for certain scenarios. The yaw trajectory error in Figure 6.15 is jerky after this rapid maneuver, indicating that the replanned yaw trajectory was too aggressive for the settings of the PX4 flight controller. In fact, the PX4 flight controller limits the yaw rate in autonomous flight modes by default to increase safety. The tremendous error in the yaw trajectory suggests that reasonable dynamic specifications should also be made for the yaw trajectory for the segment time estimation. Another solution may be the implementation of additional depth camera and distance sensors to perceive the surroundings of the quadcopter. The cross-track error stays within 27 cm, which is the maximum distance of the rotor protectors to the center of mass, which means the trajectory is safe for all instances of time. A very important fact to note at this point is that the Intel Aero RTF in combination with the PX4 Autopilot struggles to steadily hover at low altitudes or close to other surroundings. Although we do not suffer from bad position estimation, the UAV has flight stability problems when approaching a specific location. A possible cause could be the drag effects, since the Intel Aero RTF has a respectable size and weight for indoor flights.

7 Conclusions

In this thesis, the topic of trajectory (re)planning and object avoidance is concerned, which is a central topic for higher-level applications, e.g. autonomous exploration, inspection or transportation. The focus is on the development of a trajectory (re)planning algorithm that helps navigating the quadcopter autonomously from a takeoff position to a target position in the presence of known and unknown obstacles.

In Chapter 3, the available hardware and software is introduced. The Intel Aero Ready-To-Fly (RTF) platform was used, since it has a reasonable size for indoor flights and offers various sensors to estimate its position and to perceive its surroundings. The Intel Aero compute board runs an Ubuntu distribution with ROS, which allows a modular combination of several tasks, and utilizes a wifi module for a connection to remote PCs. The Intel Aero flight controller with the PX4 flight stack takes care of low-level tasks like motor control and the position and attitude control. Both compute boards are connected via the MAVLink protocol and the MAVROS library. The quadcopter is commanded in autonomous mode via a remote PC, the so-called ground control station (GCS). The data exchange between the PC and the drone occurs via ROS topics. The front-facing RGB-D camera provides depth camera information, which is utilized for obstacle detection and avoidance at 3 Hz. The point cloud processing and trajectory planning is accomplished by the GCS, which sends setpoints of the collision-free optimal trajectory at a rate of 6 Hz. For development purposes, the open-source simulator Gazebo was set up on the GCS, which offers an excellent connection to ROS and enables the software-in-the-loop (SITL) simulation of the PX4 flight stack. This allows a seamless sim-to-real transfer. During the experiments, an OptiTrack motion capture system is utilized to retrieve a robust pose information of the drone in a GPS-denied environment.

Chapter 4 gives an insight into the mathematical modelling of the quadcopter. Based on the derivation of the equations of motion, the differential flatness property of the system, with the flat output as the position and yaw angle of the drone, was determined. This serves as a basis for the trajectory planning process proposed in Chapter 5.

Trajectories are planned in the flat output space with obstacles modeled as cuboids, which are inflated for safety reasons. The optimal Rapidly Exploring Random Tree (RRT*) algorithm is utilized to find the shortest piecewise linear collision free path from the start to the target position. The Gilbert-Johnson-Keerthi (GJK) distance algorithm is employed to compute distances and to check for collisions. The obtained path is truncated by a line-of-sight (LOS) optimization to straighten the path and to improve the trajectory optimization performance. Each segment of the path is assigned a polynomial, which are then all optimized to get a sufficiently smooth trajectory. The trajectory duration is estimated by the 15-steps near time-optimal trajectory planning. The shape of the trajectory can be adjusted with user-defined penalizing weights. Two different approaches are presented to decide whether a specific part of the depth camera point cloud represents a new obstacle. If an unknown obstacle, which may be detected during flight, happens to intersect the flight path, the tree structure of RRT* is rerouted and a replanning of the trajectory is initiated.

Each major module of the trajectory planning process as well as the overall planning procedures are evaluated in Chapter 6. The influence of the respective parameters on the RRT*, the trajectory optimization and the methods for detecting new objects based on the depth camera data were investigated. A test flight in simulation and in real life verified the functionality of the proposed trajectory planning and obstacle avoidance.

To increase the performance and quality in the future, different aspects may be improved. First and foremost, including dynamic constraints on the yaw angle in the estimation of segment times would guarantee sufficient time to align the drone with the flight path even without aggressive controller parameterization. Also, including the bending angle between consecutive edges in the cost of the RRT* algorithm would yield a more linear path from the start to the target position at first hand. An automated setting of parameters based on flight space dimensions, obstacle sizes and the respective application would eliminate the need for a skilled operator. Softening the assumption of convex objects in the flight space and a more precise way of modeling obstacles in the flight space would also be further directions of research. An attempt to streamline the algorithms, in particular for obstacle detection, could allow direct online on-board replanning.

A Minimum Snap Trajectory - Minimum Principle of Pontryagin

This chapter presents the proof for a minimum snap trajectory for position $\mathbf{r}(t)$ of the quadcopter to be a 7^{th}-order polynomial in time t using the minimum principle of Pontryagin. Since all variables are time dependent, for the sake of simplicity, we use $\mathbf{z}$ instead of $\mathbf{z}(t)$. First, we define the state $\mathbf{z}$ of the quadcopter in one dimension and its time derivative $\dot{\mathbf{z}}$ as

$$\mathbf{z} = \begin{bmatrix} p \\ v \\ a \\ j \end{bmatrix}, \qquad \dot{\mathbf{z}} = \begin{bmatrix} v \\ a \\ j \\ s \end{bmatrix} \tag{A.1}$$

with the position p, velocity v, acceleration a, jerk j and snap s of the quadcopter in one dimension.

In this work, trajectories are considered in the flat output space in the form of (5.9). Since the inputs u_2, u_3 and u_4 are functions of the fourth derivatives of the position, see Section 4.2, Section 4.3, we minimize the integral of the square of the snap s

$$J = \int\limits_{t_0=0}^{t_m=T} s^2 dt. \tag{A.2}$$

Next, the co-state $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 \end{bmatrix}^T$ is defined. Now (A.2) is rewritten with the system dynamics $\dot{\mathbf{z}} = \mathbf{f}$ and the co-state in the form

$$J = \int\limits_0^T s^2 + \boldsymbol{\lambda}^T(\mathbf{f} - \dot{\mathbf{z}}) d\tau. \tag{A.3}$$

Since

$$\begin{aligned}
\frac{\mathrm{d}\boldsymbol{\lambda}^T\mathbf{z}}{\mathrm{d}\tau} &= \frac{\mathrm{d}\boldsymbol{\lambda}^T}{\mathrm{d}\tau}\mathbf{z} + \boldsymbol{\lambda}^T\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\tau} \\
-\boldsymbol{\lambda}^T\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\tau} &= \frac{\mathrm{d}\boldsymbol{\lambda}^T}{\mathrm{d}\tau}\mathbf{z} - \frac{\mathrm{d}\boldsymbol{\lambda}^T\mathbf{z}}{\mathrm{d}\tau} \\
-\int\limits_0^T \boldsymbol{\lambda}^T\dot{\mathbf{z}}\mathrm{d}\tau &= -\int\limits_0^T \frac{\mathrm{d}\boldsymbol{\lambda}^T\mathbf{z}}{\mathrm{d}\tau}\mathrm{d}\tau + \int\limits_0^T \dot{\lambda}^T\mathbf{z}\mathrm{d}\tau,
\end{aligned}$$

(A.3) reads as

$$J = \boldsymbol{\lambda}_0^T \mathbf{z}_0 - \boldsymbol{\lambda}_T^T \mathbf{z}_T + \int_0^T (s^2 + \boldsymbol{\lambda}^T \mathbf{f}) - \dot{\boldsymbol{\lambda}}^T \mathbf{z} d\tau, \tag{A.4}$$

with the notation $\mathbf{z}_0 = \mathbf{z}(t=0)$ and $\boldsymbol{\lambda}_T^T = \boldsymbol{\lambda}^T(t=T)$. The first term of the integrand in (A.4) is defined as the Hamiltonian H. The optimal trajectory $P^*(T)$ (5.9) consists of the state $\mathbf{z}^*$ and the input $\mathbf{u}^*$. We define the state as $\mathbf{z} = \mathbf{z}^* + \epsilon\delta\mathbf{z}$ and the input as $\mathbf{u} = \mathbf{u}^* + \epsilon\delta\mathbf{u}$, where ϵ is a small number, so that the variation of $\mathbf{z}$ and $\mathbf{u}$ is small. This leads to $\frac{\partial \mathbf{z}}{\partial \epsilon} = \delta\mathbf{z}$ and $\frac{\partial \mathbf{u}}{\partial \epsilon} = \delta\mathbf{u}$. The minimum condition is defined as

$$\delta J = \frac{\partial J}{\partial \epsilon} = 0 \tag{A.5}$$

$$\begin{aligned}
\delta J &= \boldsymbol{\lambda}_0^T \frac{\partial \mathbf{z}_0}{\partial \epsilon} - \boldsymbol{\lambda}_T^T \frac{\partial \mathbf{z}_T}{\partial \epsilon} + \int_0^T \frac{\partial H}{\partial \epsilon} + \dot{\boldsymbol{\lambda}}^T \frac{\partial \mathbf{z}}{\partial \epsilon} d\tau \\
&= \boldsymbol{\lambda}_0^T \delta\mathbf{z}_0 - \boldsymbol{\lambda}_T^T \delta\mathbf{z}_T + \int_0^T \left(\frac{\partial H}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \epsilon} + \frac{\partial H}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \epsilon} \right) + \dot{\boldsymbol{\lambda}}^T \delta\mathbf{z} d\tau \\
&= \boldsymbol{\lambda}_0^T \delta\mathbf{z}_0 - \boldsymbol{\lambda}_T^T \delta\mathbf{z}_T + \int_0^T \left(\frac{\partial H}{\partial \mathbf{z}} \delta\mathbf{z} + \frac{\partial H}{\partial \mathbf{u}} \delta\mathbf{u} \right) + \dot{\boldsymbol{\lambda}}^T \delta\mathbf{z} d\tau \\
&= \boldsymbol{\lambda}_0^T \delta\mathbf{z}_0 - \boldsymbol{\lambda}_T^T \delta\mathbf{z}_T + \int_0^T \left(\frac{\partial H}{\partial \mathbf{z}} + \dot{\boldsymbol{\lambda}}^T \right) \delta\mathbf{z} + \frac{\partial H}{\partial \mathbf{u}} \delta\mathbf{u} d\tau.
\end{aligned} \tag{A.6}$$

Due to initial and final constraints, the first two terms in (A.6) are zero. To fulfill the minimum condition, each component of the integrand in (A.6) must be zero, leading to

$$\frac{\partial H}{\partial \mathbf{z}} + \dot{\boldsymbol{\lambda}}^T = \mathbf{0} \tag{A.7a}$$

$$\frac{\partial H}{\partial \mathbf{u}} = \mathbf{0}. \tag{A.7b}$$

Using (A.7a), we have

$$\frac{\partial H}{\partial \mathbf{z}} = \frac{\partial (s^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j + \lambda_4 s)}{\partial \mathbf{z}} = \begin{bmatrix} 0 \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = -\dot{\boldsymbol{\lambda}}^T = \begin{bmatrix} -\dot{\lambda}_1 \\ -\dot{\lambda}_2 \\ -\dot{\lambda}_3 \\ -\dot{\lambda}_4 \end{bmatrix}. \tag{A.8}$$

Thus,

$$\lambda_1 = -\alpha$$
$$\lambda_2 = \alpha t + \beta \tag{A.9}$$
$$\lambda_3 = -\alpha\frac{t^2}{2} - \beta t - \gamma \tag{A.10}$$
$$\lambda_4 = \alpha\frac{t^3}{6} + \beta\frac{t^2}{2} + \gamma t + \delta, \tag{A.11}$$

with the polynomial coefficients α, β, γ and δ. Using (A.7b), the optimal input $\mathbf{u}^*$ reads as

$$\frac{\partial H}{\partial \mathbf{u}} = \frac{\partial(s^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j + \lambda_4 s)}{\partial \mathbf{s}} = 2s + \lambda_4 = 0 \tag{A.12}$$
$$\mathbf{u}^* = s^* = -\alpha\frac{t^3}{12} - \beta\frac{t^2}{4} - \gamma\frac{t}{2} - \frac{1}{2}\delta. \tag{A.13}$$

Finally, the optimal trajectory $\mathbf{z}^*$ is obtained by integration, resulting in

$$\mathbf{z}^* = \begin{bmatrix} p^* \\ v^* \\ a^* \\ j^* \end{bmatrix} = \begin{bmatrix} \int_0^t v^* dt \\ \int_0^t a^* dt \\ \int_0^t j^* dt \\ \int_0^t s^* dt \end{bmatrix} = \begin{bmatrix} -\alpha\frac{t^7}{10080} - \beta\frac{t^6}{1440} - \gamma\frac{t^5}{240} - \delta\frac{t^4}{48} \\ -\alpha\frac{t^6}{1440} - \beta\frac{t^5}{240} - \gamma\frac{t^4}{48} - \delta\frac{t^3}{12} \\ -\alpha\frac{t^5}{240} - \beta\frac{t^4}{48} - \gamma\frac{t^3}{12} - \delta\frac{t^2}{4} \\ -\alpha\frac{t^4}{48} - \beta\frac{t^3}{12} - \gamma\frac{t^2}{4} - \delta\frac{t}{2} \end{bmatrix}. \tag{A.14}$$

Bibliography

[1] S. Angster, S. Wesnousky, W. Huang, G. Kent, T. Nakata, and H. Goto, "Application of uav photography to refining the slip rate on the pyramid lake fault zone, nevada," *Bulletin of the Seismological Society of America*, vol. 106, no. 2, pp. 785–798, 2016.

[2] T. Wang, K. Umemoto, T. Endo, and F. Matsuno, "Modeling and control of a quadrotor uav equipped with a flexible arm in vertical plane," *IEEE Access*, vol. 9, pp. 98 476–98 489, 2021.

[3] M. Silvagni, A. Tonoli, E. Zenerino, and ChiabergeMarcello, "Multipurpose uav for search and rescue operations in mountain avalanche events," *Geomatics, Natural Hazards and Risk*, vol. 8, no. 1, pp. 18–33, 2017.

[4] D. Kang and Y.-J. Cha, "Autonomous uavs for structural health monitoring using deep learning and an ultrasonic beacon system with geo-tagging," *Computer-Aided Civil and Infrastructure Engineering*, vol. 33, no. 10, pp. 885–902, 2018.

[5] C. Ju and H. I. Son, "Multiple uav systems for agricultural applications: Control, implementation, and evaluation," *Electronics*, vol. 7, no. 9, 2018.

[6] A. S. Saeed, A. B. Younes, S. Islam, J. Dias, L. Seneviratne, and G. Cai, "A review on the platform design, dynamic modeling and control of hybrid UAVs," in *International Conference on Unmanned Aircraft Systems*, 2015, pp. 806–815.

[7] L. R. Garcia Carrillo, A. Dzul, R. Lozano, and C. Pégard, *Quad Rotorcraft Control. Vision-Based Hovering and Navigation*. London: Springer, 2012.

[8] D. Mellinger and V. Kumar, "Minimum Snap Trajectory Generation and Control for Quadrotors," in *International Conference on Robotics and Automation (ICRA)*, 2011, pp. 2520–2525.

[9] J.-M. Kai, G. Allibert, M.-D. Hua, and T. Hamel, "Nonlinear feedback control of quadrotors exploiting first-order drag effects," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 8189–8195, 2017.

[10] M. Faessler, A. Franchi, and D. Scaramuzza, "Differential Flatness of Quadrotor Dynamics Subject to Rotor Drag for Accurate Tracking of High-Speed Trajectories," *IEEE Robotics & Automation Letters*, vol. 3, no. 2, pp. 620–626, 2018.

[11] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy, "Autonomous Navigation and Exploration of a Quadrotor Helicopter in GPS-Denied Indoor Environments," in *First Symposium on Indoor Flight*, 2009.

[12] K. Schmid, P. Lutz, T. Tomi'c, E. Mair, and H. Hirschmüller, "Autonomous Vision-Based Micro Air Vehicle for Indoor and Outdoor Navigation," *Journal of Field Robotics*, vol. 31, no. 4, pp. 537–570, 2014.

[13] S. Shen, "Autonomous navigation in complex indoor and outdoor environments with micro aerial vehicles," Ph.D. dissertation, University of Pennsylvania, 2014.

[14] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics Research*, Cham: Springer, 2016, pp. 649–666.

[15] D. W. Mellinger, "Trajectory Generation and Control for Quadrotors," Ph.D. dissertation, University of Pennsylvania, 2012.

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[18] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 566–580, 1996.

[19] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[20] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[21] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. M. Saeed, "RRT*-SMART: A rapid convergence implementation of RRT*," *International Journal of Advanced Robotic Systems*, vol. 10, no. 7, p. 299, 2013.

[22] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *International Conference on Intelligent Robots and Systems (IROS)*, 2014, pp. 2997–3004.

[23] M. Cutler and J. How, "Actuator Constrained Trajectory Generation and Control for Variable-Pitch Quadrotors," Minneapolis, Minnesota: American Institute of Aeronautics and Astronautics, 2012.

[24] S. Liu, N. Atanasov, K. Mohta, and V. Kumar, “Search-based motion planning for quadrotors using linear quadratic minimum time control,” in *International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 2872–2879.

[25] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran, “Continuous-time trajectory optimization for online uav replanning,” in *International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 5332–5339.

[26] M. Vu, P. Zips, A. Lobe, F. Beck, W. Kemmetmüller, and A. Kugi, “Fast motion planning for a laboratory 3d gantry crane in the presence of obstacles,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 9508–9514, 2020.

[27] M. N. Vu, A. Lobe, F. Beck, T. Weingartshofer, C. Hartl-Nesic, and A. Kugi, “Fast trajectory planning and control of a lab-scale 3d gantry crane for a moving target in an environment with obstacles,” *Control Engineering Practice*, vol. 126, p. 105 255, 2022.

[28] M. N. Vu, C. Hartl-Nesic, and A. Kugi, “Fast swing-up trajectory optimization for a spherical pendulum on a 7-dof collaborative robot,” in *International Conference on Robotics and Automation (ICRA)*, 2021, pp. 10 114–10 120.

[29] F. Grander, “Dynamisches greifen von 3d-objekten mittels robotischem system,” M.S. thesis, Automation and Control Institute (ACIN), TU Wien, 2021.

[30] Y. Peng, D. Qu, Y. Zhong, S. Xie, J. Luo, and J. Gu, “The obstacle detection and obstacle avoidance algorithm based on 2-d lidar,” in *International Conference on Information and Automation (ICIA)*, 2015, pp. 1648–1653.

[31] J. Azeta, C. Bolu, D. Hinvi, and A. A. Abioye, “Obstacle detection using ultrasonic sensor for a mobile robot,” *IOP Conference Series: Materials Science and Engineering*, vol. 707, no. 1, 2019.

[32] A. R. Vetrella, A. Savvaris, G. Fasano, and D. Accardo, “Rgb-d camera-based quadrotor navigation in gps-denied and low light environments using known 3d markers,” in *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 185–192.

[33] A. Nguyen and B. Le, “3d point cloud segmentation: A survey,” *International Conference on Robotics, Automation and Mechatronics*, pp. 225–230, 2013.

[34] H. Chen and P. Lu, “Real-time identification and simultaneous avoidance of static and dynamic obstacles on point cloud for uavs navigation,” *Robotics and Autonomous Systems*, vol. 154, 2022.

[35] D. Natter, “Quadrotor Navigation in GPS-Denied Environments using Multi-Sensor Data Fusion,” M.S. thesis, Automation and Control Institute (ACIN), TU Wien, 2020.

[36] L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys, "Pixhawk: A system for autonomous flight using onboard computer vision," in *International Conference on Robotics and Automation (ICRA)*, 2011, pp. 2992–2997.

[37] *PX4 User Guide*. [Online]. Available: `https://docs.px4.io/master/en/` (visited on 06/09/2022).

[38] A. Allouch, O. Cheikhrouhou, A. Koubâa, M. Khalgui, and T. Abbes, "Mavsec: Securing the mavlink protocol for ardupilot/px4 unmanned aerial systems," in *15th International Wireless Communications & Mobile Computing Conference*, 2019, pp. 621–628.

[39] *Introduction MAVLink Developer Guide*. [Online]. Available: `https://mavlink.io/en/` (visited on 06/09/2022).

[40] *Controller Diagrams | PX4 User Guide*. [Online]. Available: `https://docs.px4.io/v1.12/en/flight_stack/controller_diagrams.html` (visited on 06/09/2022).

[41] *Motion Capture Systems*, en. [Online]. Available: `http://optitrack.com/index.html` (visited on 06/09/2022).

[42] S. Vemprala, *2D/3D RRT* algorithm*. [Online]. Available: `https://de.mathworks.com/matlabcentral/fileexchange/60993-2d-3d-rrt-algorithm` (visited on 02/05/2022).

[43] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal on Robotics and Automation*, vol. 4, pp. 193–203, 1988.

[44] M. Montanari and N. Petrinic, "OpenGJK for C, C# and Matlab: Reliable solutions to distance queries between convex bodies in three-dimensional space," *SoftwareX*, vol. 7, pp. 352–355, 2018.

[45] A. Boeuf, "Kinodynamic Motion Planning for Quadrotor-Like Aerial Robots," Ph.D. dissertation, Institut National Polytechnique de Toulouse, 2017.

[46] J. Qi, H. Yang, and H. Sun, "MOD-RRT*: A Sampling-Based Algorithm for Robot Path Planning in Dynamic Environment," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 8, pp. 7244–7251, 2021.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct - Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, October 2022

Martin Zimmermann