ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Autonomous Systems Lab
Prof. Roland Siegwart

Master-Thesis

Path Planning for Dynamic Maneuvers with Micro Aerial Vehicles

Autumn Term 2014

Supervised by:
Markus Achtelik
Michael Burri

Author:
Fabian Neuschäfer

Contents

Abstract

The goal of this master thesis is to develop a numerical robust trajectory planning algorithm for dynamic multi-copter flights in dense environments. The trajectory generated by this algorithm is represented by polynomials which are jointly optimized. The cost function of the optimization consists of the total trajectory-time as well as the total quadratic snap (second derivation of the acceleration). The inclusion of the snap into the cost function guaranties a trajectory without abrupt or expensive control inputs.

Furthermore, the process of exploring the state space using the Rapidly-Exploring Random Tree (RRT) algorithm is embedded into the numerical robust algorithm. The sampling points oft the RRT (or RRT*) algorithm are then used as the nodes in the polynomial optimization.

Symbols

Terms and Definitions

jerk	Derivation of acceleration
snap	Derivation of jerk
vertex	Fixed sampling point of a polynomial trajectory

Acronyms and Abbreviations

ETH	Eidgenössische Technische Hochschule
QP	Quadratic Programming
UAV	Unmanned Aerial Vehicle
RRT	Rapidly-Exploring Random Tree
ROS	Robot Operating System

Chapter 1

Introduction

1.1 State of the Art

A lot of research has been performed in the field of Unmanned Aerial Vehicles (UAV) in the last years leading to a strong improvement in trajectory planning [1] as well as in control ([2], [3]). Another field of research is machine learning [4] which is suitable to enhance the performance of aerobatic maneuvers but seems to have a downside regarding motion planning and trajectory generation in dense environments.

Speaking of trajectory planning, there are two different strategies which are pursued. On the one hand, the geometric and the temporal planning are decoupled [5], on the other hand geometric and temporal information are coupled and the trajectory is the result of a minimization problem. For the coupled problem one can make use of the differential flatness of a quadrocopter to derive constraint on the trajectory. A cost function which could be the total trajectory-time [3] or the total snap [6] can be formulated.

Another aspect of planning is exploring the state space in the first place. A strong tool to do so are incremental search techniques as for instance the A* [7] or the RRT* algorithm [8]. The sampling points of the solution of the incremental search can then be used as the nodes for the polynomial optimization.

1.2 Quadratic Programming

As mentioned above, the snap can be used as the cost function in trajectory optimization. Regarding snap minimization, Quadratic Programming (QP) is a powerful tool.

1.2.1 Constrained Quadratic Programming

QP is a special case of an optimization problem in which a quadratic function $f(x)$ is optimized with respect to its optimization variables (which are represented with the vector x in equation 1.1)

$$f(x) = \frac{1}{2} \cdot x^T Q x + c^T x \tag{1.1}$$

The optimization can be performed under linear constraints on the optimization variables. The linear constraints can be divided in two groups.

For one thing there are the inequality constraints

$$Ax \leq b \tag{1.2}$$

where the vector b contains the inequality constraints. For another thing there are the equality constraints

$$Ex = d \tag{1.3}$$

where the vector d contains the equality constraints. In case there are only equality constraints, the solution to the QP is given by the linear system in equation 1.4 :

$$\begin{bmatrix} Q & E^T \\ E & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{1.4}$$

where λ is a set of Lagrange multipliers and c is the linear term of the cost function in equation 1.1.

The constrained QP gets ill-conditioned for a large number of optimization variables which lead to large matrices. The performance of the constraint QP deteriorates even more if the matrices are sparse. This particular case often appears in polynomial optimization for high order polynomials where some polynomial coefficients are close to zero.

To reduce the number of optimization variables, and therefore the size of the matrices, the constrained QP with equality constraints can be converted into a numerical robust unconstrained QP. This is one of the goals of this master thesis.

1.2.2 Unconstrained Quadratic Programming

For the unconstrained QP the equality constraints $E\mathbf{x} = \mathbf{d}$ resp. $\mathbf{x} = E^{-1}\mathbf{d}$ are embedded into the quadratic cost function from equation 1.1 resulting in equation 1.5:

$$f(d) = \frac{1}{2} \cdot d^T E^{-T} Q E^{-1} d + c^T E^{-1} d \tag{1.5}$$

Since the vector x is replaced by $E^{-1}d$ and E is a constant matrix, the new optimization variables are now stored in the vector d.

If the cost function defined in equation 1.1 does not have a linear term, i.e. the vector c is equal to zero, equation 1.5 can be simplified to:

$$f(d) = \frac{1}{2} \cdot d^T E^{-T} Q E^{-1} d \tag{1.6}$$

If we are not interested in the cost itself but only in the optimization variables stored in d, the constant multiplier $1/2$ can be dropped:

$$f(d) = d^T E^{-T} Q E^{-1} d \tag{1.7}$$

The theoretical derived equation 1.7 will be compared to to multidimensional cost function in equation 2.5. Equation 2.5 establishes a connection between the numerical advantages of a unconstrained QP and the polynomial coefficients representing a trajectory.

Chapter 2

Polynomial Trajectory Optimization

2.1 Polynomial Trajectory

Regarding the differentiability of polynomials, they are a profound choice to represent a trajectory, especially for the use in a differentially flat representation of the UAV dynamics. (Flatness in the proper sense of system theory means that all the states and inputs can be expressed in terms of the flat output and a finite number of its derivative). Furthermore, the differentiability of polynomials enables the possibility to check the derivatives of the trajectory for bounding violations to avoid input saturation. This saturation-check can be performed during trajectory optimization and therefore guarantees the feasibility of the resulting trajectory.

2.2 Optimization

The goal of this master thesis is to optimize a trajectory which passes through waypoints (also called vertices or nodes) which are defined in advance. This waypoints can be chosen manually or by a path-finding algorithm such as RRT* which will be discussed in chapter 3. Furthermore, not only the waypoints (i.e. the position) can be fixed in advance but also its derivatives (such as speed, acceleration etc.). The position and its derivatives are then utilized as the equality constraints for a QP (explained in Section 1.2).

2.2.1 Cost Function

Optimization for the purpose of trajectory planning means to minimize a cost function. The cost function in this case is a combination of temporal and geometric cost. The geometric cost penalizes the square of the derivatives of the trajectory. In this master thesis the geometric cost is represented by the squared snap which guarantees a trajectory without abrupt control inputs.
The temporal cost is simply the total trajectory time multiplied by a user chosen factor k_T which determines the aggressiveness of the resulting trajectory. The impact of k_T can be seen in equation 2.10 which represents the combined geometric and temporal cost.

To express the geometric cost in a compact way one can utilize the Hessian matrix Q. The Hessian matrix is defined as a squared matrix of second-order partial derivatives which follows from differentiation a function with respect to each of its coefficients,

in this instance the polynomial coefficients. The geometric cost function $J(T)$ for one segment with the duration T can now be written as

$$J(T) = p^T \cdot Q(T) \cdot p \tag{2.1}$$

where $Q(T)$ is the Hessian matrix for a fixed segment time T. p is the vector containing the coefficients of the polynomial trajectory.

If the trajectory consists of more than one segment the Hessian matrix has to be extended to a block-diagonal matrix. The geometric cost function for multiple segments with fixed but individual segment times T_i can be written as

$$J(T) = \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix}^T \cdot \begin{bmatrix} Q_1(T_1) & & \\ & \ddots & \\ & & Q_n(T_n) \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix} \tag{2.2}$$

2.2.2 Polynomial Optimization as a Constrained QP

The cost function in equation 2.2 has to be minimized under constraints since we want to fix things like start or end position. In a first intuitive approach the constraints on the endpoint derivatives are utilized in a constrained QP. Therefore, a mapping matrix E between endpoint derivatives and polynomial coefficients is needed. The resulting equality constraint for the i^{th} segment can be written as

$$E_i \cdot p_i = d_i \tag{2.3}$$

where p is the vector containing the polynomial coefficients and d is the vector containing the endpoint derivatives. Regarding the total number of segments of the trajectory, equation 2.3 can be written in matrix form:

$$\begin{bmatrix} E_1 & & \\ & \ddots & \\ & & E_n \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix} \tag{2.4}$$

The constrained QP is suitable for a small amount of segments but gets ill-conditioned for a large amount of segments and therefore large matrices. Especially if there are matrices which are close to singularity and have coefficients which are close to zero, the constrained QP can get numerical unstable.

2.2.3 Polynomial Optimization as a Unconstrained QP

To avoid the numerical instability of a constrained QP the optimization problem is converted into a unconstrained QP. To achieve this, the polynomial coefficients p_i from equation 2.2 have to be substituted by the endpoint derivatives d_i which are now the new optimization variables. The cost function of the unconstrained QP can now be written as

$$J = \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix}^T \cdot \begin{bmatrix} E_1 & & \\ & \ddots & \\ & & E_n \end{bmatrix}^{-T} \cdot \begin{bmatrix} Q_1 & & \\ & \ddots & \\ & & Q_n \end{bmatrix} \cdot \begin{bmatrix} E_1 & & \\ & \ddots & \\ & & E_n \end{bmatrix}^{-1} \cdot \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix} \tag{2.5}$$

where Q_i is the Hessian matrix according to the i^{th} segment time T_i.

As mentioned above, the endpoint derivatives are the new optimization variables. Due to the equality constraints some of the endpoint derivatives are already specified consequently reducing the number of optimizations variables. Expediently, the endpoint derivatives are divided in fixed derivatives d_f and unspecified derivatives d_p and then reordered using the matrix C which consists of zeros and ones. After reordering the endpoint derivatives equation 2.5 can be rewritten as

$$J = \begin{bmatrix} d_f \\ d_p \end{bmatrix}^T \underbrace{C^T E^{-T} Q E^{-1} C}_{\mathrm{R}} \begin{bmatrix} d_f \\ d_p \end{bmatrix} \tag{2.6}$$

where the product of the reordering matrix C, the mapping matrix E and the Hessian matrix Q can be expressed as a single Matrix R. The matrix R for his part can be divided into four submatrices according to the fixed and unspecified endpoint derivatives which modifies equation 2.6 as follows:

$$J = \begin{bmatrix} d_f \\ d_p \end{bmatrix}^T \begin{bmatrix} R_{ff} & R_{fp} \\ R_{pf} & R_{pp} \end{bmatrix} \begin{bmatrix} d_f \\ d_p \end{bmatrix} \tag{2.7}$$

Partially differenting equation 2.7 with respect to the unspecified derivatives d_p and equate it to zero yields the optimized/minimized unspecified derivatives d_p^*

$$d_p^* = -R_{pp}^{-1} \cdot R_{fp}^T \cdot d_f \tag{2.8}$$

as a function of the fixed derivatives d_f and two of the submatrices (R_{pp}, R_{fp}) of R. Equation 2.8 can now be used to compute the initial solution.

2.3 Initial Solution

As can be seen in equation 2.5, the Hessian matrix for the i^{th} segment Q_i depends on the segment time T_i. Thus, all the segment times have to be defined in advance. For the initial solution the segment times are calculated based on the Euclidean distance d_{norm} and on the user specified maximal speed (v_{max}) and maximal acceleration (a_{max}).

Basically, the segment time is determined by the term $d_{norm}/v_{max} \cdot 2$ which is twice the time the UAV would need for a segment by flying the whole distance at maximal speed. Although this is a good estimation for long segments, for shorter ones the time needed to accelerate gets significant. In order to incorporate acceleration time, a multiplier which is zero for long segments and larger than zero for short ones is added. The segment time T_i for the i^{th} segment can be computed according to

$$T_i = \frac{d_{norm_i}}{v_{max}} \cdot 2 \cdot \left(1 + 6.5 \cdot \frac{v_{max}}{a_{max}} \cdot \frac{1}{e^{\frac{d_{norm_i}}{v_{max}} \cdot 2}} \right) \tag{2.9}$$

where d_{norm_i} is the Euclidean distance of the i^{th} segment, v_{max} the user specified maximal velocity and a_{max} the user specified maximal acceleration. The fraction v_{max}/a_{max} gives an idea of how much time is needed to accelerate to maximum velocity whereas 6.5 is a empirical weighting factor.

The result of equation 2.9 is depicted in figure 2.1 whereat the x-axis represents the Euclidean distance d_{norm} and the y-axis represents the segment time T. For this plot the user specified limitation on speed has been set to $v_{max} = 3\frac{m}{s}$ and the limitation on acceleration has been set to $a_{max} = 5\frac{m}{s^2}$. The dashed green line represents the term $d_{norm}/v_{max} \cdot 2$ and serves as a reference. The blue graph is the exact representation of equation 2.9.

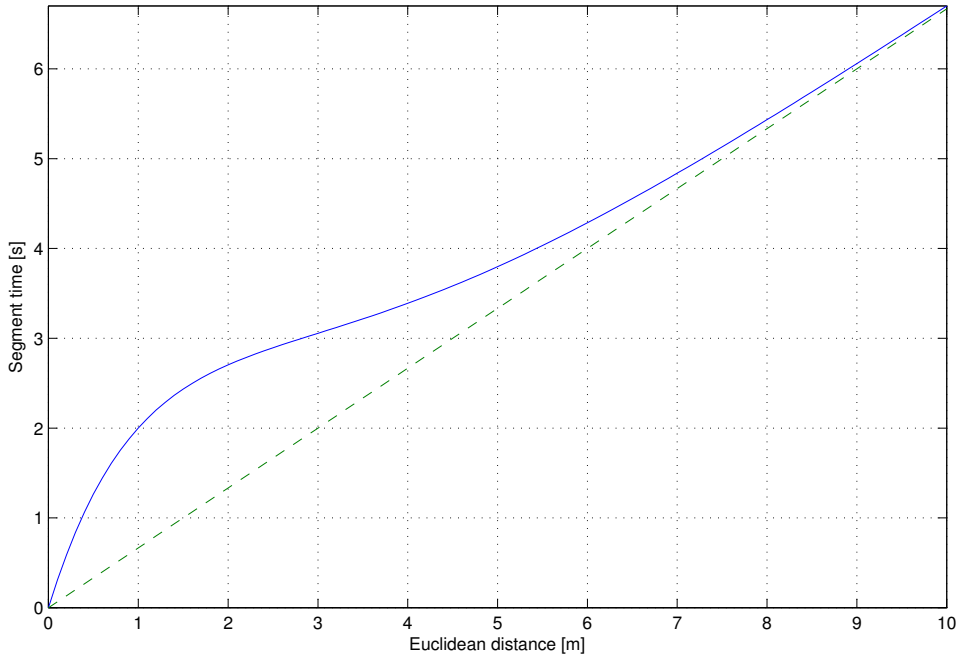

Figure 2.1: The segment time T depends on the Euclidean distance d_{norm} of a segment as well as on the max. velocity v_{max} and the max. acceleration a_{max}. Integration of acceleration time has an impact on segment time for short distances.

Once the segment times are calculated, the initial snap minimized solution can be computed according to equation 2.8. The initial solution for a 3 dimensional problem with 2 segments is depicted in figure 2.2. The start of the trajectory is the origin of the Cartesian coordinate system (0/0/0). For both, start and goal state, the velocity, the acceleration, the jerk and the snap are fixed and set to zero. For all the other sampling points (vertices) the derivatives are unspecified. The Cartesian coordinates of the sampling points are chosen manually and are listed in the following table. For the sake of convenience, a simple set of vertex with coordinates which increase permanently have been chosen:

Waypoint	x-coordinate	y-coordinate	z-coordinate
Start-Vertex	0	0	0
Vertex 1	1	2	5
Goal-Vertex	3	4	6

Table 2.1: 3 manually chosen vertices

The initial solution depicted in figure 2.2 is divided into 3 plots. Plot a) shows the position (i.e. the Cartesian coordinates) whereby each of the 3 dimensions is depicted as a single graph. The Cartesian coordinates from the above-noted table are depicted as circles. Plot b) shows the velocity of the individual direction as a solid graph. Additional, the velocity in the three-dimensional space (i.e. the Euclidean norm of the velocity vector) is depicted as a dashed graph. Plot c) depicts the acceleration in the individual directions (solid) and the acceleration in the three-dimensional space (dashed). Furthermore the limitation for velocity and acceleration ($v_{max} = 3\frac{m}{s}$ and $a_{max} = 4\frac{m}{s^2}$ for this problem) are depicted in plot b) resp. c) . The x-axis for all the 3 plots is the time.

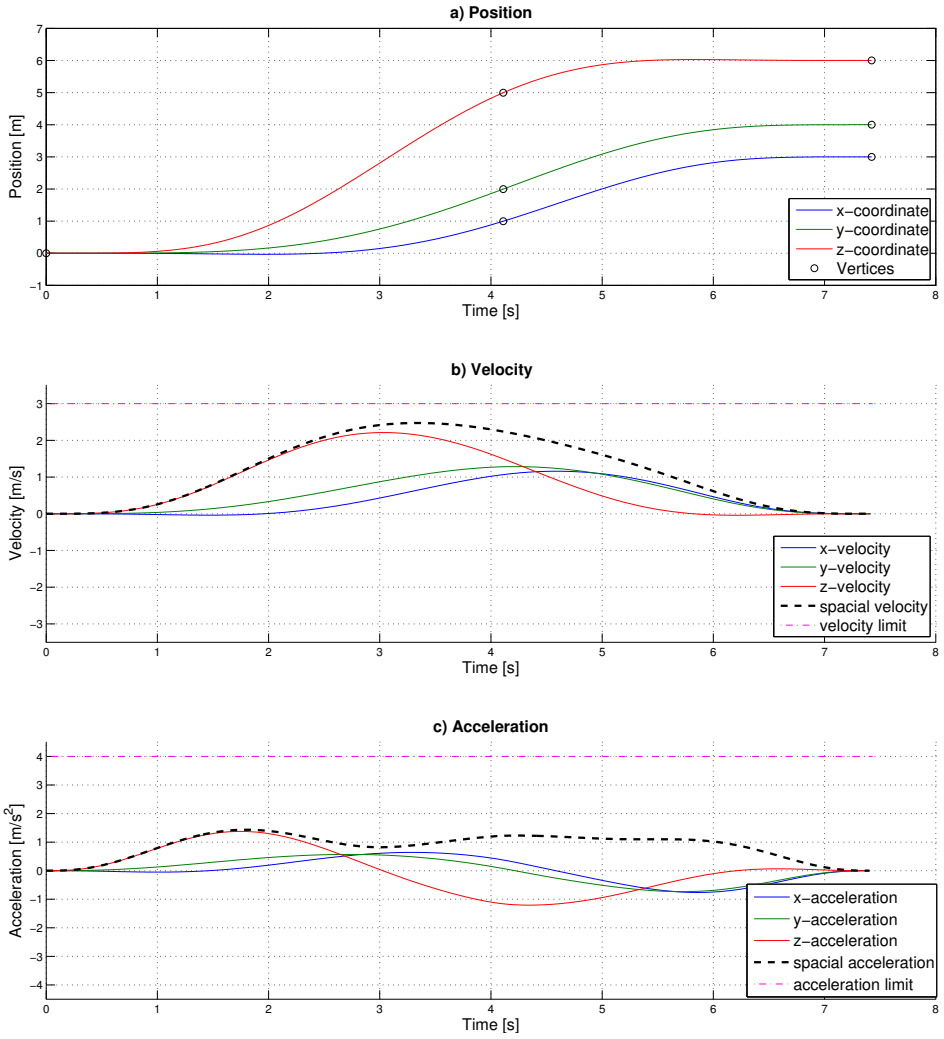

Figure 2.2: Initial solution of a trajectory with 2 segments: Plot a) shows the position (i.e. the Cartesian coordinates). Plot b) shows the velocity and plot c) the acceleration. A dashed graph represents the velocity respectively the acceleration in the three-dimensional space.

As can be seen in plot b) the maximal spacial velocity is less then the user specified maximal velocity $v_{max} = 3\frac{m}{s}$. The same applies to plot c) where the maximal acceleration is far below the user specified maximal acceleration of $a_{max} = 4\frac{m}{s^2}$. Hence, the trajectory could be more aggressive/faster without violating the limitations.

2.3.1 Drawbacks of the Initial Solution

The individual segment times and consequently the total time of the initial trajectory are defined by equation 2.9. Since equation 2.9 does not incorporate the circumstances from one segment to another it is likely to find a better trajectory (i.e. a trajectory with a smaller total cost) with the same total time but a different ratio between the segment time. Moreover, there is no possibility to adjust the aggressiveness of the initial solution since the segment times are calculated up-front.

In summary, the modification of the individual segment times and therefore the modification of the ratio of the segment times can lead to a solution with smaller cost. Supplementary, the modification of the individual segment times gives us the opportunity to adjust the aggressiveness of a trajectory.

2.4 Time Allocation

So fare, only the geometric cost (i.e. the squared snap) was included in the cost function defined in equation 2.7. Minimization of the geometric cost ensures a smooth trajectory without abrupt input signal but has no effect on the aggressiveness of a trajectory. Therefore equation 2.7 has to be extended by the temporal cost (i.e. the sum of the segment times) which results in the total cost J_{total}

$$J_{total} = \begin{bmatrix} d_f \\ d_p \end{bmatrix}^T \begin{bmatrix} R_{ff} & R_{fp} \\ R_{pf} & R_{pp} \end{bmatrix} \begin{bmatrix} d_f \\ d_p \end{bmatrix} + k_T \cdot \sum_{i=1}^{N} T_i \tag{2.10}$$

where k_T is a user specified weighting factor and T_i is the segment time of the i^{th} segment.

A hight value for k_T lays weight to the temporal cost and therefore leads to a trajectory with a short total trajectory time. A small value for k_T on the other hand lays little weight on the temporal cost, meaning the geometric cost gets more important. Since the geometric cost (i.e. the quadratic snap) decreases for long segment times with little changes in jerk, this leads to a trajectory with a long total trajectory time. Thus, the user specified weighting factor k_T enables the adjustment of the aggressiveness of a trajectory.

2.4.1 Nonlinear Optimization

The geometric cost function in equation 2.7 has only the unspecified endpoint derivatives d_p as optimization variables. This optimization problem can be solved analytically as performed in equation 2.8. The cost function in equation 2.10 has the segment times T_i as additional optimization variables, meaning the segment times T_i are directly represented in the cost function and not only indirectly via the Hessian matrices Q_{T_i}. Since the segment times are now optimization variables, the Hessian matrices Q_{T_i} are no longer defined in advance and the problem cannot be solved analytically. Due to that a nonlinear solver is used. In this master thesis NLopt, a open-source library for nonlinear optimization, is applied. The unspecified endpoint derivatives d_p and the segment times T_i of the initial solution are the initial values for the nonlinear solver. Meaning the initial solution has to be computed in advance to start a nonlinear optimization.

To illustrate the nonlinear optimization, the trajectory with the same 3 vertices (defined in the table 2.1) is reused. In this master thesis position, velocity, acceleration, jerk and snap are fixed for the start and for the goal vertex since we want to start from complete rest and want to end up in standstill. For the vertex in the middle the position is fixed but velocity, acceleration, jerk and snap are unspecified, resulting in the first 4 optimization variables. Since this is a 3 dimensional problem, the number of geometric optimization variables triples to 12. Together with the segment time of the two segments, the problem ends up with a total number of 14 optimization variables.

The result of the nonlinear optimization is depicted in figure 2.3. Plot a) depicts the position (i.e. the Cartesian coordinates) whereby each of the 3 dimensions is depicted as a single graph. Plot b) shows the velocity of the individual direction as a solid graph. Additional, the velocity in the three-dimensional space (i.e. the Euclidean norm of the velocity vector) is depicted as a dashed graph. Plot c) depicts the acceleration in the individual directions (solid) and the acceleration in the three-dimensional space (dashed). Furthermore the limitation ($v_{max} = 3\frac{m}{s}$ and $a_{max} = 4\frac{m}{s^2}$ for this problem) are depicted. In contrast to figure 2.2, the x-axis of figure 2.3 stops at $7s$ since the optimized trajectory is faster.

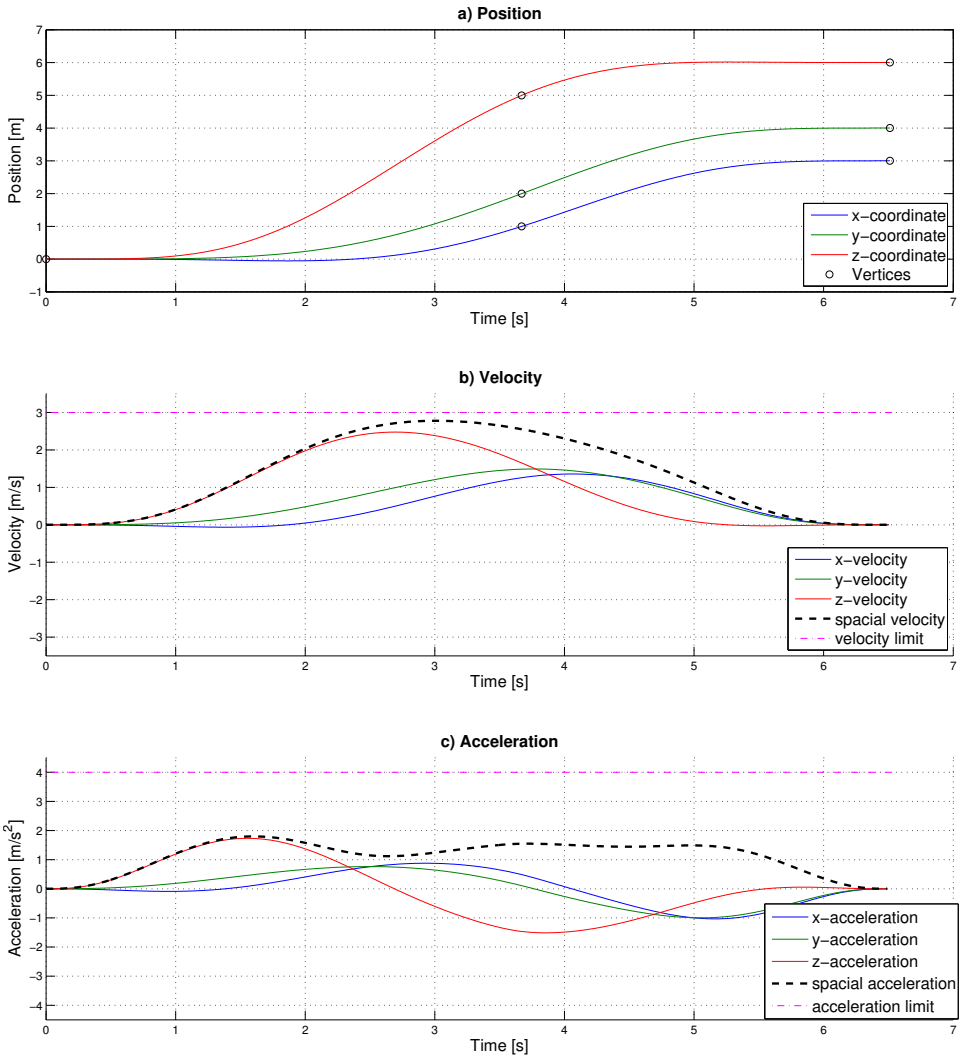

Figure 2.3: Optimized solution of a trajectory with 2 segments: Plot a) shows the position (i.e. the Cartesian coordinates). Plot b) shows the velocity and plot c) the acceleration. A dashed graph represents the velocity respectively the acceleration in the three-dimensional space. Weighting factor k_T was set to 100.

As can be seen in figure 2.3 a) the trajectory passes the same vertices as in figure 2.2. In this optimization the user specified weighting factor was set to $k_T = 100$. This rather small value for k_T leads to a trajectory which is neither affected by

the limitation on the velocity (the spacial velocity in plot b) is always below the velocity limit) nor by the limitation on acceleration(the spacial acceleration in plot c) is always below the acceleration limit). Nevertheless, the optimized trajectory with a duration of $6.52s$ is faster then the initial solution with a duration of $7.43s$.

To get a more aggressive trajectory the value for k_T has to be increased. In figure 2.4 the optimize trajectory for $k_T = 190$ is depicted. As can be seen in plot b) the limitation on the velocity comes into play, i.e. the spacial velocity is restricted to $v_{max} = 3\frac{m}{s}$. The duration of the trajectory depicted in figure 2.4 is $6.01s$. Although the maximal velocity and the maximal acceleration are higher than in figure 2.3, the pathway of the spacial velocity and of the spacial acceleration look very similar for $k_T = 100$ and $k_T = 190$.

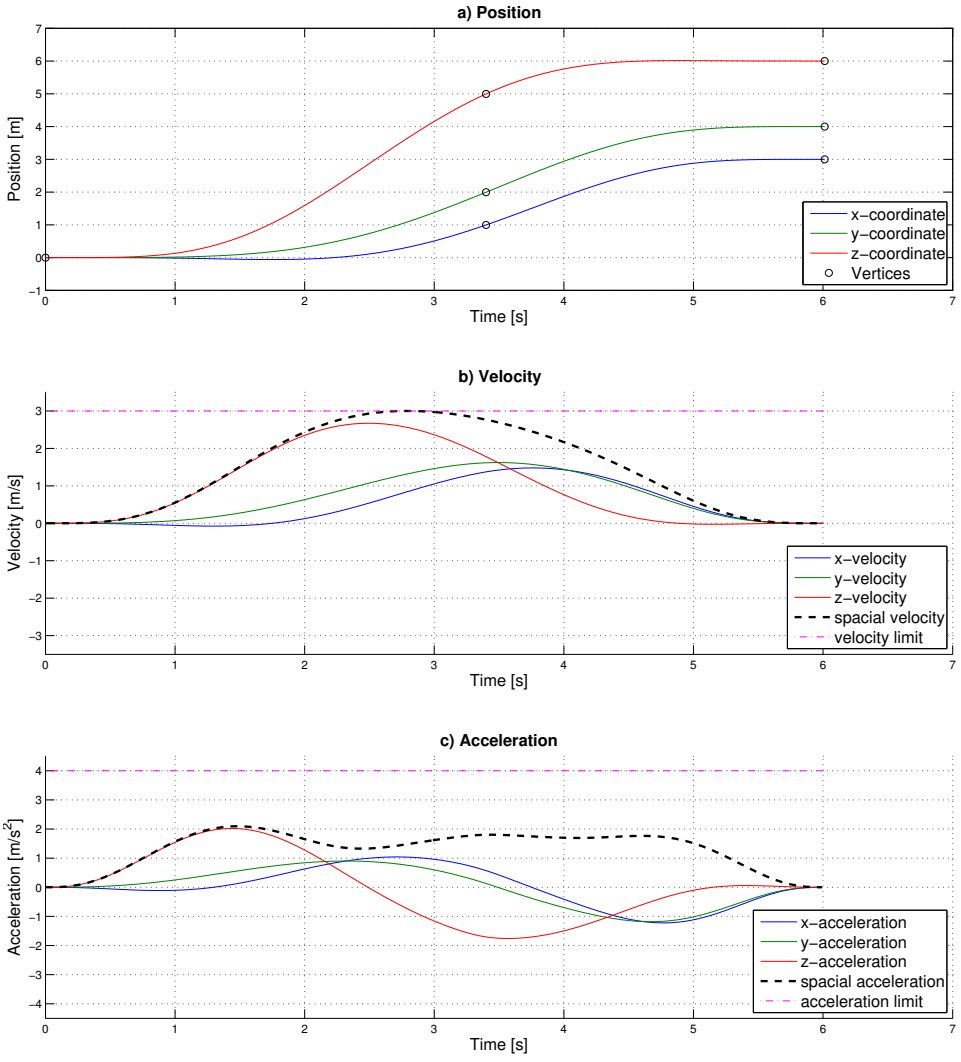

Figure 2.4: Optimized solution of a trajectory with 2 segments: Plot a) shows the position (i.e. the Cartesian coordinates). Plot b) shows the velocity and plot c) the acceleration. A dashed graph represents the velocity respectively the acceleration in the three-dimensional space. Weighting factor k_T was set to 190.

In a next step the value for k_T is increased even more and set to $k_T = 2000$. The resulting trajectory is depicted in figure 2.5. The duration of this aggressive trajectory is $5.16s$. The pathway of the spacial velocity looks now different than in the previous figures. The spacial velocity not only touches the velocity limit but stays at the limit for about one second.

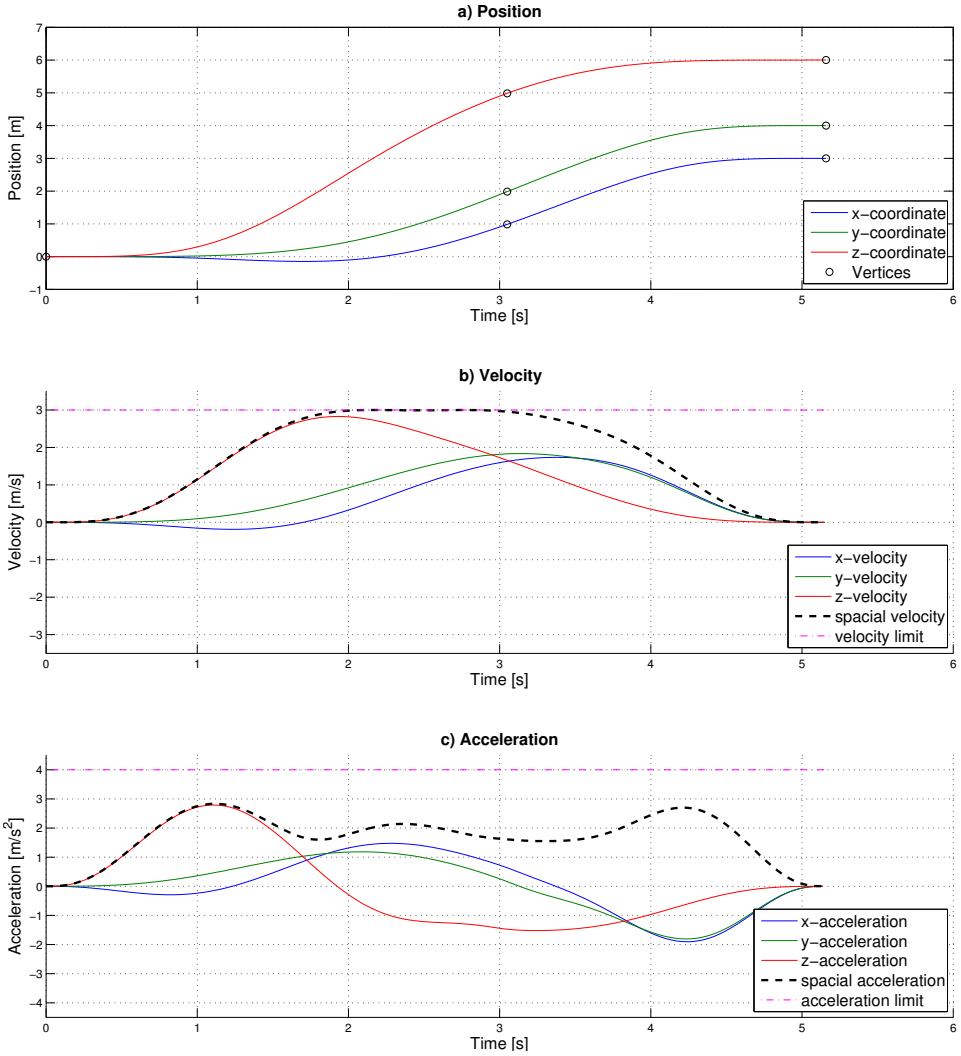

Figure 2.5: Optimized solution of a trajectory with 2 segments: Plot a) shows the position (i.e. the Cartesian coordinates). Plot b) shows the velocity and plot c) the acceleration. A dashed graph represents the velocity respectively the acceleration in the three-dimensional space. Weighting factor k_T was set to 2000.

Although figure 2.5 depicts an aggressive trajectory the spacial acceleration always remains under the acceleration limit. Of course this depends on the choice of the acceleration limit, i.e. the acceleration limit would get significant if a_{max} is set to less than $3\frac{m}{s^2}$. A factor which is independent from the user specified choice of k_T is the pathway of the trajectory. For a trajectory with little curves the acceleration has a smaller influence on the duration than the velocity. Since the coordinates in all 3 dimensions increase permanently, this is the case in figure 2.5.

To depict the features of a trajectory with more curves, a new set of vertices is chosen manually:

Waypoint	x-coordinate	y-coordinate	z-coordinate
Start-Vertex	0	0	0
Vertex 1	5	1	-2
Vertex 2	3	-2	1
Vertex 3	-1	2	3
Goal-Vertex	1	-1	-2

Table 2.2: 5 manually chosen vertices

The initial solution with 4 segments passing through the above-noted vertices is depicted in figure 2.6. Please note that the user specified limitation on the velocity is set to $v_{max} = 4\frac{m}{s^2}$ for this example.

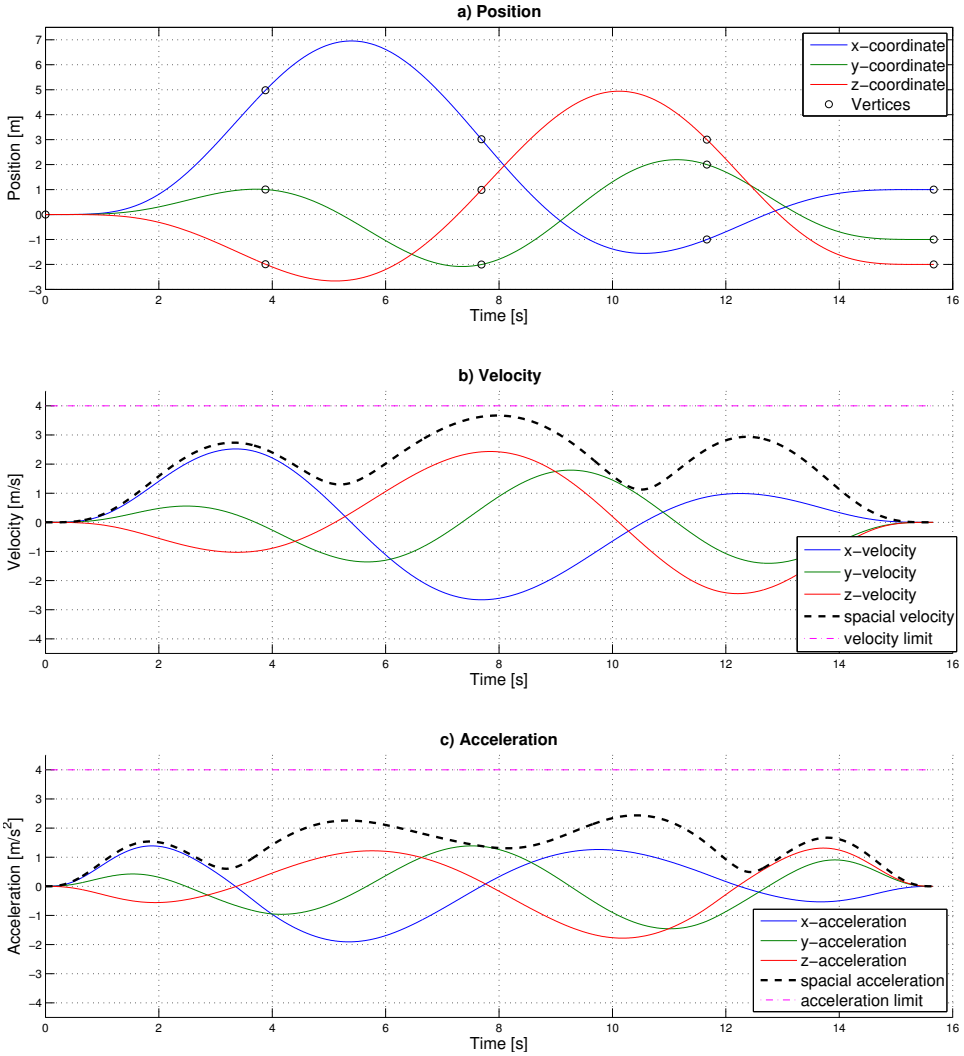

Figure 2.6: Initial solution of a trajectory with 4 segments: A dashed graph represents the velocity respectively the acceleration in the three-dimensional space.

The initial solution is required to define the initial values of the nonlinear optimization. As can be seen in plot b) and plot c) neither the limitation on the velocity nor the limitation on the acceleration are violated.

Using the unspecified enpoint derivatives d_P and the segment times T_i of the initial solution as initial values for the optimization variables the nonlinear optimization was performed with a weighting factor of $k_T = 2000$. The optimized trajectory is depicted in figure 2.7. The duration of the optimized trajectory is $10.02s$ compared to the initial trajectory duration of $15.67s$ depicted in figure 2.6.

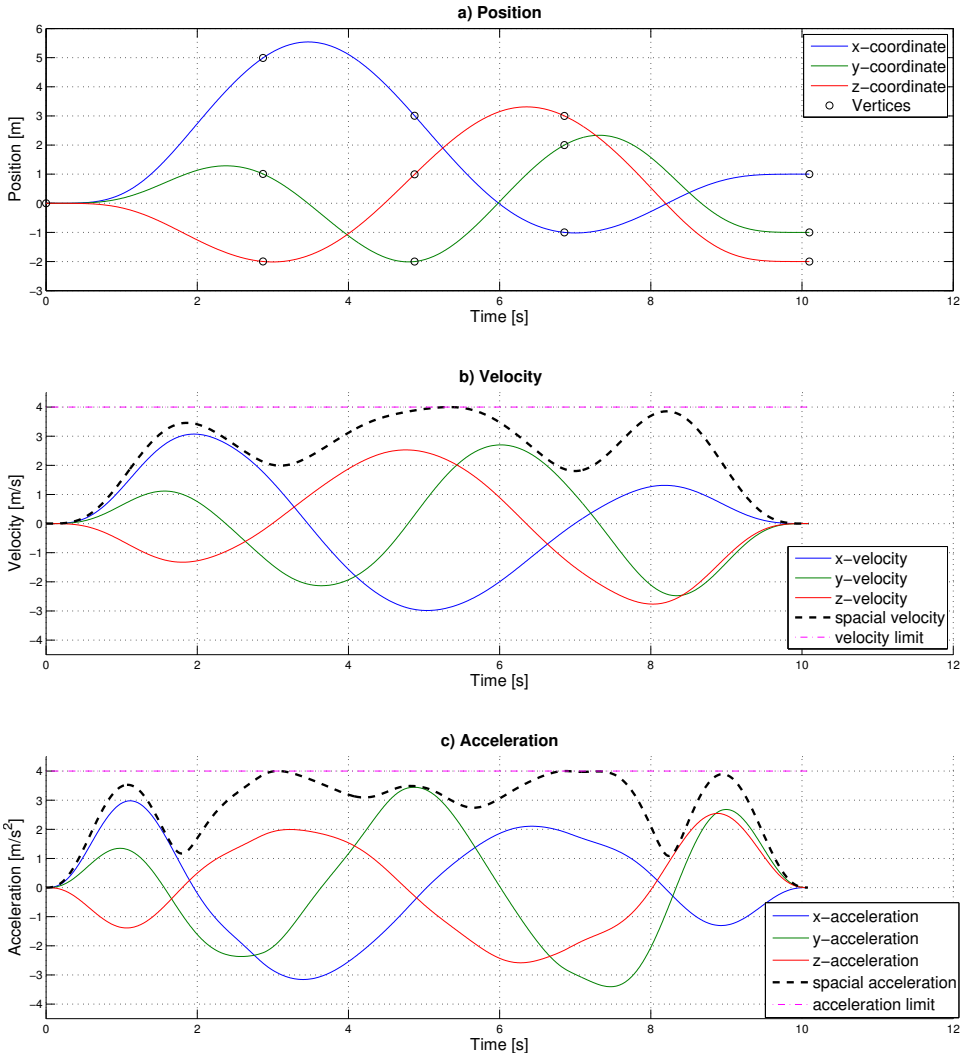

Figure 2.7: Optimized solution of a trajectory with 4 segments: A dashed graph represents the velocity respectively the acceleration in the three-dimensional space. Weighting factor k_T was set to 2000.

In contrast to the aggressive trajectory depicted in figure 2.5, the acceleration limit has an impact on the optimized winding trajectory in figure 2.7. As can be seen in plot c) the spacial acceleration touches the acceleration limit repeatedly and even stays at exact $4\frac{m}{s^2}$ for some time.

2.5 Pathway of the Trajectory

The pathway of a snap minimized trajectory is mainly determined by the ratio of the segment times and not by the segment times themselves or by the total trajectory time. As mentioned in section 2.3.1, the segment times from the initial solution do not consider the circumstances from one segment to an other. Hence, it is likely to find a better trajectory by changing the ratio between the segment times, even if the total time of the trajectory stays the same.

The process of optimizing the ratio of the segment times is implicitly performed during the process of nonlinear optimization. In other words, the nonlinear optimization not only changes the total trajectory time according to the weighting factor k_T but also optimizes the ratio of the segment times.

Such a change in the pathway is distinguishable between the initial trajectory in figure 2.6 a) and the optimized trajectory in figure 2.7 a). The easiest notable difference is the peak of the x graph with a peak value of 6.95 in the initial trajectory and a peak value of 5.58 in the optimized trajectory. However, the changes in the pathway are more apparent in a 3D plot.

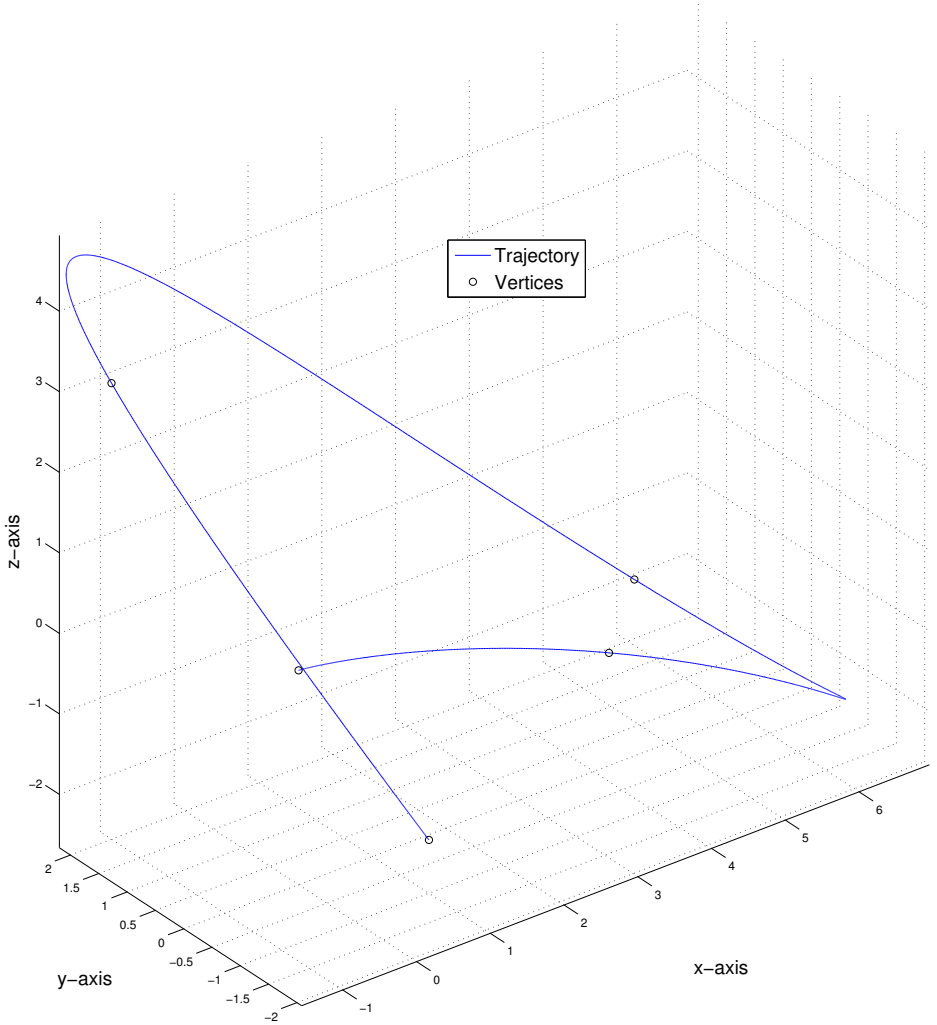

Figure 2.8: 3D plot of the initial trajectory with 4 segments. The start vertex is located at (0/0/0) and the trajectory proceeds counter-clockwise.

Figure 2.8 depicts the initial trajectory (from figure 2.6) in 3D space. Again, the vertices are marked as circles but now the time is no longer explicit readable. The start vertex is located at (0/0/0) and the trajectory proceeds counter-clockwise. The second segment (lower right corner) of the initial trajectory in figure 2.8 looks very sharp. Such sharp corners, accrued by the suboptimal segment time ratio, are undesirable if the UAV should fly a trajectory dynamically.

By optimizing the ratio of the segment times, which is implicitly performed during the nonlinear optimization, the pathway of the trajectory gets smoother, i.e. no sharp corners exist in the optimized trajectory.
Figure 2.9 depicts the optimized trajectory (from figure 2.7) in 3D space. Especially the second segment of the optimized trajectory is now more suitable for a dynamic flight. In addition, the pathway in the region of the 4. vertex (in the upper left corner) is improved. The initial trajectory made a detour before passing the 4. vertex. This detour, accrued by the suboptimal segment time ratio, vanishes in the optimized trajectory.

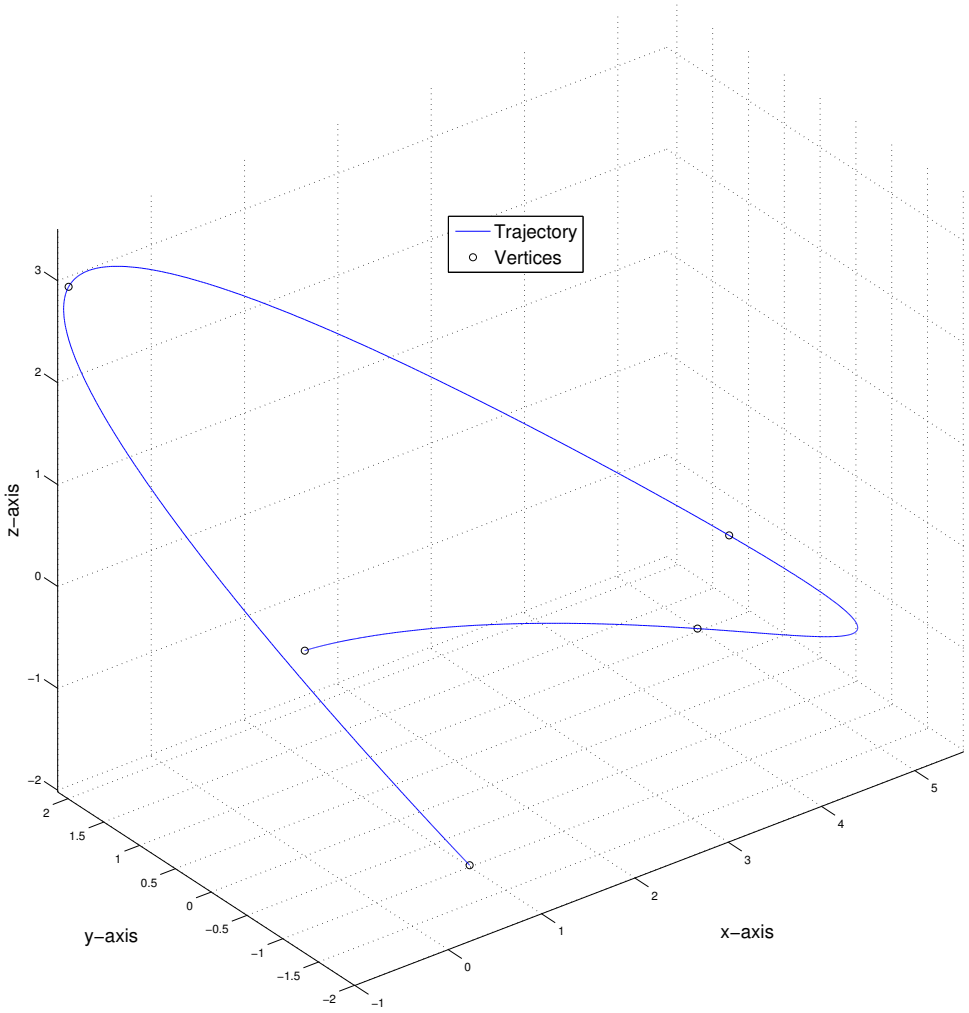

Figure 2.9: 3D plot of the optimized trajectory with 4 segments. The start vertex is located at (0/0/0) and the trajectory proceeds counter-clockwise.

2.6 NLopt

For all the nonlinear optimizations which were performed in the previous sections, NLopt [9], an open-source library for nonlinear optimization, was applied. The optimization algorithm tries to minimize the cost function every iteration and needs parameters for termination condition in order to know when to stop.
The termination condition of the optimization can be specified by the optimization variables as well as by the total cost. Generally, the termination conditions are formulated relative to the current value(s). For instance, if the relative termination condition for the total cost J_{rel} is set to 0.01 the optimization ends if the total cost changes less than one percent during an iteration. The relative termination condition for the optimization variables x_{rel} is only fulfilled if all of the optimization variables change less than the threshold in one iteration. Additionally, an absolute termination condition x_{abs} can be applied to the optimization variables. This termination condition is only needed if one or several optimization variables are close to zero and the relative criteria therefore does not work properly.

Two additional options for termination condition are the limitation of the number of optimization iterations and the limitation of the duration of the optimization. All of these termination condition can be set simultaneously and the first condition which is fulfilled stops the optimization. During the optimization the constraints on velocity and acceleration are checked every iteration.

For the previous optimization problems only J_{rel} was applied. This is the general case, since the other parameters are mostly used to intercept if the the cost function does not converge.

Chapter 3

RRT

3.1 General

The goal of this thesis is not only to generate a numerically stable, snap optimized polynomial trajectory but also to explore a densely packed (indoor) environment and plan an aggressive trajectory in between the obstacles. Hence, the Rapidly-Exploring Random Tree (RRT) algorithm is used to find a collision-free straight line solution through densely packed environments. The sampling points of the RRT (or RRT*) algorithm are then used as the vertices in the polynomial optimization.

3.2 RRT Algorithm

RRT is a computational efficient algorithm to find a path in a high dimensional space by randomly building a space-filling tree. The sampling points are drawn randomly from the sample space and the tree grows incrementally. For each new sample the algorithm attempts to build a collision-free connection to the nearest state in the tree. If a collision-free connection is possible, the sample and the connection are added to the tree.

An iteration of the RRT algorithm can be depicted schematically:

1. Generate a random sample
2. Find nearest state in tree
3. Try to build a collision-free connection to the nearest state
4. If feasible, add the sampled state and the connection to the tree

3.2.1 Goal State

As mentioned above, the RRT algorithm is based on random samples. Therefore it is very unlikely that a sampled state perfectly matches the desired goal state.

There are two different strategies to enable the RRT algorithm to get to the goal. One strategy is to define not only a goal state but a goal region. Every random sample which is located within the goal region is considered a goal state. As soon as a collision-free connection to a sample in the goal region is established, this trajectory is stored as the best trajectory. At this point the algorithm can be stopped or further iteration can be performed to find a better trajectory to the goal

region. Once an other state from the goal region is sampled and the cost of the path to the new state is lower then the cost of the best trajectory, the best trajectory is replaced by the new path.
Another strategy is to steer the RRT algorithm directly to the goal state. In addition to the randomly sampled states the exact goal state is added to the algorithm. The schematic description of an iteration of the RRT algorithm listed in section 3.2 can be modified to represent an iteration with the goal state:

1. Insert goal state
2. Find nearest state in tree
3. Try to build a collision-free connection to the nearest state
4. If feasible, add the sampled state and the connection to the tree

In all the cases where a direct collision-free connection between start and goal state is not possible the iteration with the goal state will not succeed in a first attempt. Hence the iterations with the randomly sampled states described in section 3.2 are needed to build the space filling tree.

Figure 3.1 depicts the straight line solution of the RRT algorithm with a fixed goal state. The figure is in bird's eye perspective and shows a crossing of different hallways. The blue cells represent the floor and the green cells represent the walls. The map was generated with a stereo camera and was not reworked. Therefore, some of the cells of the floor which should be occupied/blue are left free.

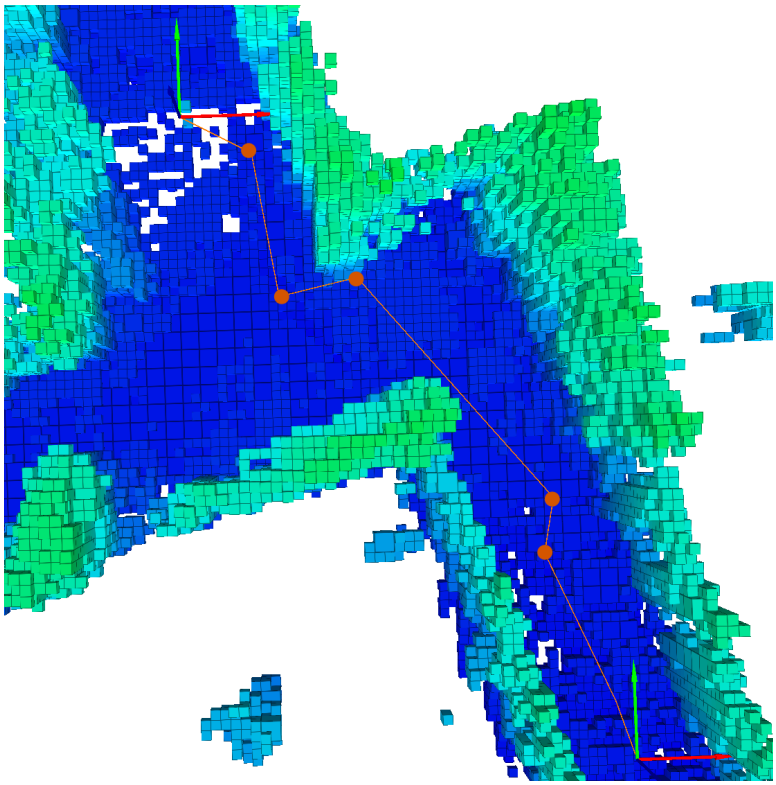

Figure 3.1: Straight line solution of the RRT algorithm with a fixed goal state. The start state (upper left corner) and the goal state (lower right corner) are each represented by a x and y-vector. The random sampled states are depicted as orange dots.

3.3 RRT* Algorithm

In contrast to the RRT algorithm, the RRT* (or RRT Star) algorithm not only tries to connect to the nearest state in the tree but to several states near the sampled state. The user can define a threshold (on the distance) which defines which states of the tree belongs to the "near states". If there is no state within the user specified range the algorithm attempts to build a collision-free connection to the nearest state in the tree just as the RRT algorithm.
As a first step, the sampled state is connected to the best state among the near states whereas "best" means minimum cost/distance. Once the sampled state is added to the tree, all the other states among the set of near states are connected to the sampled state. If the connection is collision-free and the cost of the total path is smaller than the cost of the existing path, the old path is replaced.

An iteration of the RRT* algorithm can be depicted schematically:

1. Generate a random sample
2. Define a threshold for a set of near states
3. Try to build a collision-free connection to best state among the near states
4. Add the sampled state and the connection to the tree
5. Try to connect all the other states from the set with the sampled state
6. Replace the old path if the new one has a smaller cost
7. If there is no near state within the threshold, apply the RRT algorithm

Because the RRT* algorithm tries to connect to several states each iteration, the procedure to find a path takes longer and is computationally more expensive. However, solutions with lower cost can be found which is more important for most real life applications.

3.3.1 Rewiring

The sequence of step 5 and step 6 of the RRT* algorithm is called "rewiring". The amount of rewiring is defined by a threshold which specifies the set of near states. The threshold, in our case the radius of a sphere, depends on an user specified parameter γ. I.e. all the near states are located within this sphere. The radius r can be calculated according to

$$r = \gamma * \left(\frac{ln(n+1)}{n+1} \right)^{1/d} \tag{3.1}$$

where n is the number of states which are already in the tree. The dimension of the state space d is a fixed parameter and ln represents the natural logarithm.

As mentioned in section 3.3, the RRT* algorithm is computationally more expensive but delivers trajectories with lower cost. Both aspects are caused by the rewiring and are therefore strongly influenced by the parameter γ. A large γ defines a large sphere, hence it is likely to have more states (which are already in the tree) to be located within the sphere. The rewiring of the near states tends to result in shorter trajectories with fewer segments.

Figure 3.2 depicts the straight line solution of the RRT* algorithm with a fixed goal state. Compared to figure 3.1, a superior trajectory is found due to rewiring of the states in the tree.

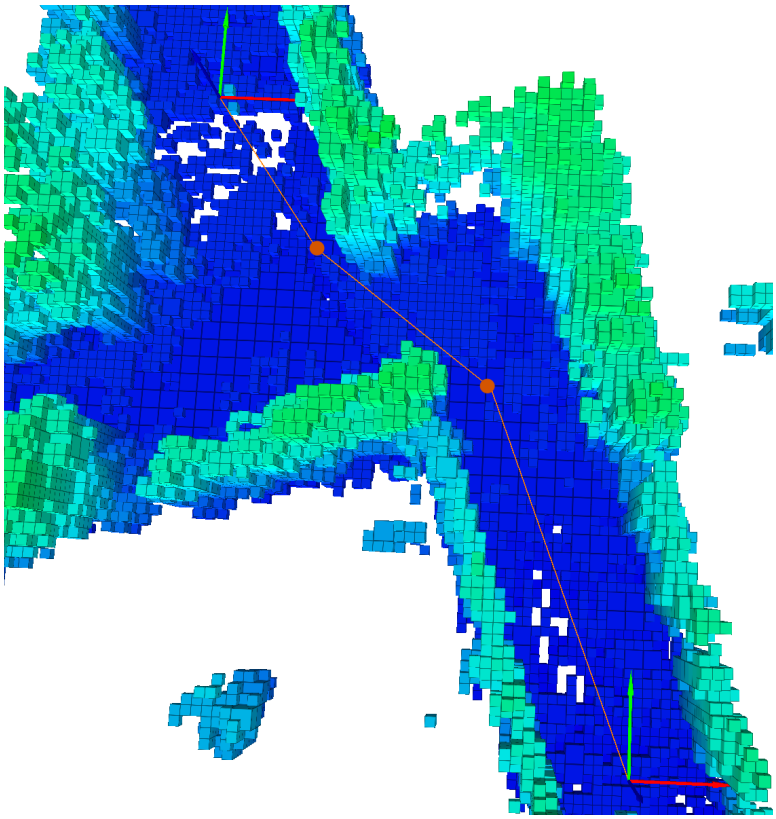

Figure 3.2: Straight line solution of the RRT* algorithm with a fixed goal state. The start state and the goal state are each represented by a x and y-vector. The random sampled states are depicted as orange dots. The γ parameter in this example was set to $\gamma = 1.5$.

3.3.2 Bounding Box

The straight line solution in figure 3.2 is collision-free but very close to the walls. In real life application not only a point mass but a object (in this master thesis a UAV) should follow the trajectory. Therefore a bounding box needs to be installed around the trajectory.
The bounding box is implemented as a cuboid. The 3 dimensions of the cuboid can be defined individually. The trajectory is then divided into a discrete trajectory and the bounding box is installed around the discrete points. If there is a obstacle in one of the bounding boxes the hole straight line is considered "in collision".

Figure 3.3 depicts the straight line solution of the RRT* algorithm with a bounding box. In contrast to figure 3.2, the trajectory is now located more central in the hallway because the bounding box makes it impossible to pass by the wall very close. Because the trajectory proceeds less direct from the start state to the goal state, the total distance of the trajectory increases.

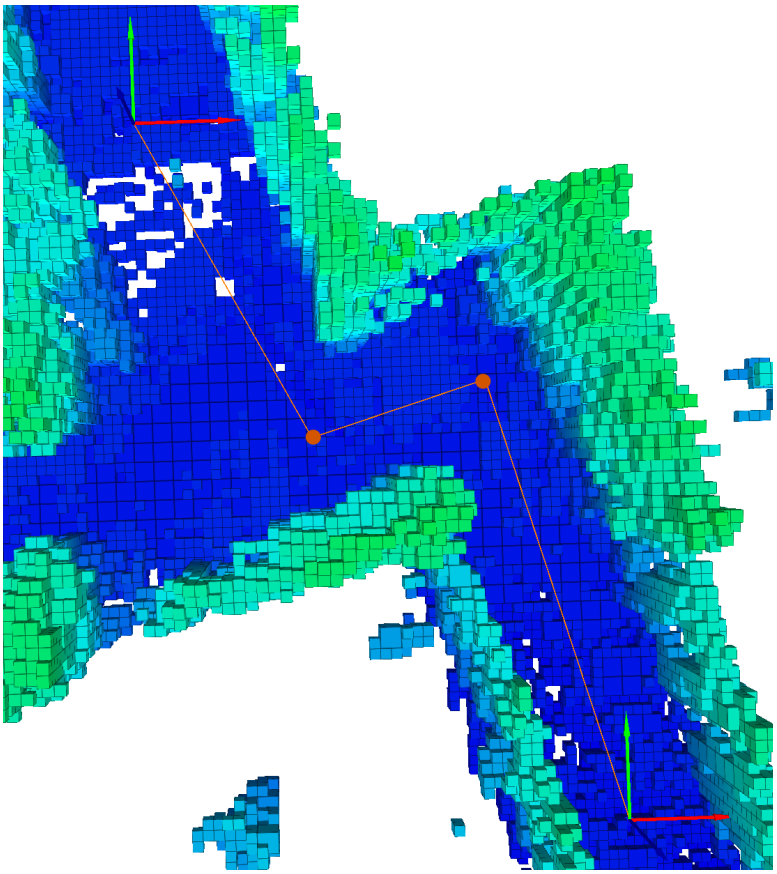

Figure 3.3: Straight line solution of the RRT* algorithm with bounding box. Due to the bounding box the trajectory is located more central in the hallway. The γ parameter in this example was set to $\gamma = 1.5$.

3.3.3 Ray Check

In the figures above, the map depicting the hallways is a so-called "OctoMap" [10]. In an OctoMap, the obstacles are stored as cells. Every cell has its individually key, which defines the location in the map. This keys are stored in a tree, named octree.

As discussed in section 3.3.2, a bounding box is used to check the trajectory for collisions. To do so, the trajectory is discretized and the bounding box is installed around every discrete trajectory-point. The bounding box is implemented as a cuboid with its edges aligned to the coordinate system of the map. Since the straight line connections can be situated arbitrary in space, a certain amount of overlap of the bounding boxes is required to guarantee that all the space around the trajectory is collision-free. As a downside of this overlap, several cells of the OctoMap are checked twice.

To avoid the unnecessary double-check of the cells, a new approach called "Ray Check" is implemented. The idea is to generate a ray from one point in the OctoMap to another. Then, all the keys of the cells through which the trajectory passes are stored. Subsequently, the keys are checked if one of them represent a occupied cell.

Since a single ray does not represent any volume, multiple rays have to be tested. This corridor of rays is implemented as a tube in this master thesis.

Figure 3.4 depicts the start of a straight line connection (red) which is situated arbitrary in space. Around the start point, a set of points is arranged on a circle. The same set of points have to be installed around the end point of the straight line connection (which is not depicted in figure 3.4). If all the points around the start point are connected via a ray to the corresponding point around the end point, a tube develops.

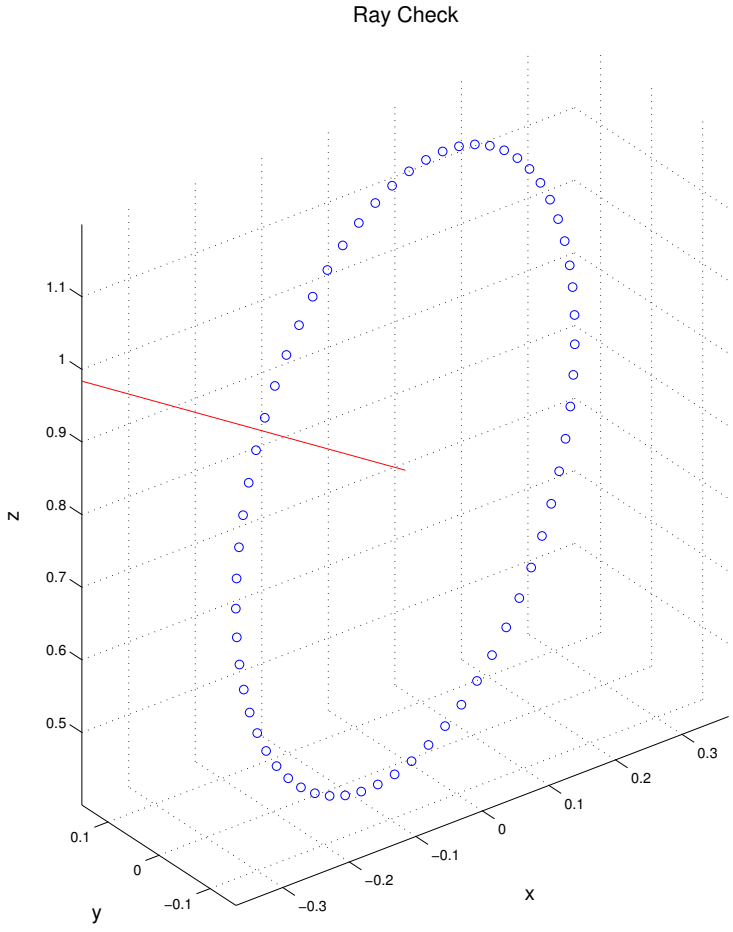

Figure 3.4: Points are arranged on a circle around the start point of a straight line connection. The points have to be connected to the corresponding point from the end point.

A problem with the Ray Check approach are floating objects. A object which is floating, i.e. has no connection to any wall, could be located inside the corridor without being touched by a ray. Generally, there are no floating objects in a indoor environment. Still, a lamp which hangs on a thin cable could lead to a floating object in the OctoMap if the cable is not detected.

To resolve the problem, the OctoMap has either to be reworked or more points on different circles, each with a different radius, have to be added. Both possibilities would need additional computational time. First, the simple approach depicted in figure 3.4 is compared to the Bounding Box approach to decide if a further development of the Ray Check approach makes sense.

The two approaches have been tested for the start and goal point depicted in figure 3.5. The RRT* algorithm has been stopped as soon as a collision-free straight line solution was found.

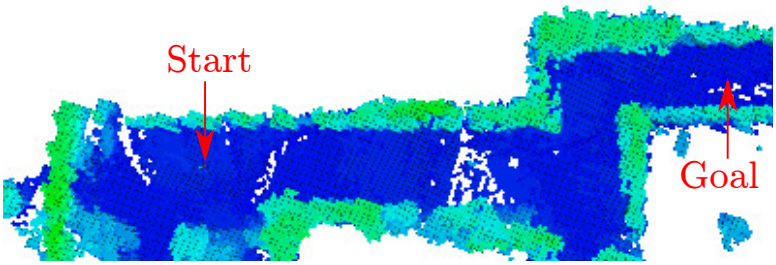

Figure 3.5: Bird's eye perspective on a hallway in the "ML" building of the ETH Zurich. The start and the end point are illustrated.

The size of the bounding box was $0.5m$ in all dimensions. The straight line connection was discretized in $0.3m$ steps. The radius for the Ray Check was $0.5m$ and the points were place every 0.2 radian. The comparison has been performed for $\gamma = 1$ and $\gamma = 1.5$. All the combinations have been executed 10 times and the average computational times are listed in the following table:

Approach	Time for $\gamma = 1$	Time for $\gamma = 1.5$
Bounding Box	0.343	0.643
Ray Check	0.473	1.233

Table 3.1: Average computational time for the two approaches "Bounding Box" and "Ray Check".

As can bee seen, the Bound Box approach performs better, independent of the γ parameter. Anyway, the difference gets more significant if γ and therefore the amount of rewiring increases.

The Bounding Box approach performs better according to table 3.1 and does not have any problems with floating objects. Therefore, only the Bounding Box approach is progressed.

Chapter 4

Path Plananing

4.1 Usage of the RRT* Vertices

Chapter 2 illustrates the optimization of a trajectory which passes through predefined vertices. From now on, the vertices are no longer chosen manually but are defined by the result of the RRT* algorithm. The computational efficiency of the RRT* algorithm combined with the positive features of the nonlinear optimization discussed in section 2.4.1 will lead to a optimized trajectory in densely packed environment.

4.1.1 Vertex Extension

Since the RRT* algorithm does not consider the dynamical behavior of a UAV there are still some challenges converting the RRT* straight line solution into a polynomial trajectory. In other words, there is the possibility that the polynomial trajectory passing through the vertices of the collision-free straight line solution is in collision with an obstacle. In this case a vertex, which is located on the straight line, has to be added.

The process of generating a collision-free optimized trajectory can be summarized in following steps:

1. Generate a collision-free straight line solution using the RRT* algorithm

2. Create an initial trajectory passing through the vertices of the straight line solution

3. If the initial solution is in collision, extend the existing vertices by a vertex located on the straight line solution. Compute the initial solution including the new vertex. Repeat this step until the trajectory is collision-free

4. Perform a nonlinear optimization on the collision-free trajectory

5. If the optimized solution is in collision, extend the existing vertices by a vertex located on the straight line solution and restart the nonlinear optimization. Repeat this step until the optimized trajectory is collision-free

This schematic description of the process is depicted step by step in the subsequent figures for a better understanding.

1. Generate a collision-free straight line solution using the RRT* algorithm

Figure 4.1 depicts the straight line solution of the RRT* algorithm, which is explained in section 3.3. The γ parameter, which is needed to calculate the radius r in equation 3.1, is set to $\gamma = 1.5$. Furthermore, the edge length of the bounding box is set to $0.5m$ for each dimension. Although the hight of a UAV is commonly smaller than the length and the width this is required if great roll and pitch angles are permitted.

Please note that, due to the randomness of the RRT* algorithm, the straight line solution depicted in figure 4.1 is different than the straight line solution depicted in figure 3.3 even though the parameters are the same.

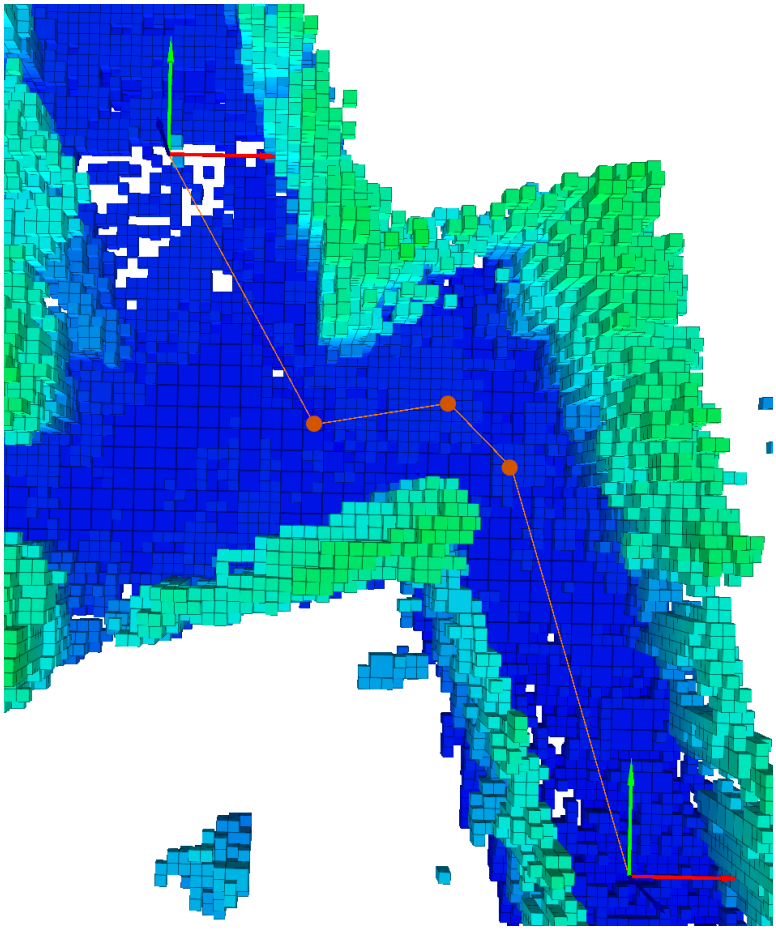

Figure 4.1: A collision-free straight line solution with 4 segments.

2. Create an initial trajectory passing through the vertices of the straight line solution

The start and goal vertex (each marked by a red and green arrow in figure 4.1) as well ass the 3 vertices (each marked by a orange dot in figure 4.1) defined by the solution of the RRT* algorithm are now used to generate the initial polynomial solution according to equation 2.8 and equation 2.9.

The initial trajectory passing through the vertices of the RRT* algorithm is depicted in figure 4.2. The start vertex is in the upper left corner and the goal vertex is in the lower right corner. In the second segment there is a collision between the trajectory and the wall of the hallway (represented by green boxes).

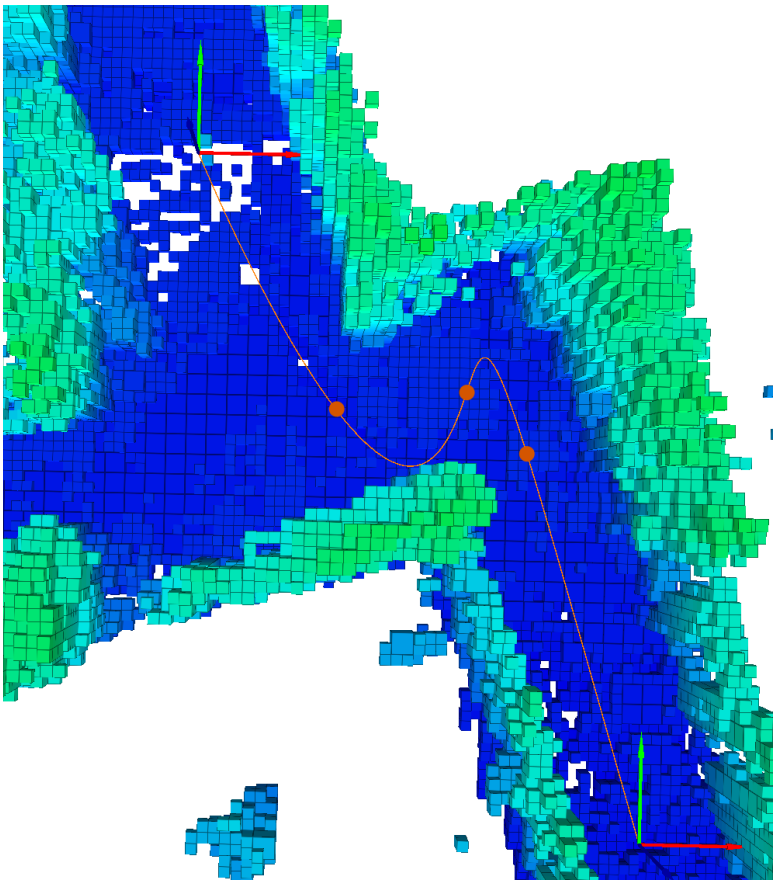

Figure 4.2: Initial trajectory with 4 segments. The start vertex is located in the upper left corner and the goal vertex is located in the lower right corner. There is a collision in the second segment of the trajectory.

3. If the initial solution is in collision, extend the existing vertices by a vertex located on the straight line solution. Compute the initial solution including the new vertex. Repeat this step until the trajectory is collision-free

To avoid the collision, a new vertex which is located on the straight line connection of the second segment has to be added. As a result, the trajectory is forced to stay closer to the straight line solution which is collision-free.

The trajectory after one cycle of vertex extension is depicted in figure 4.3. A new vertex, represented by a red dot, is placed on the straight line connection of the second segment.
As can be seen in figure 4.3 the new vertex is not placed in the middle of the straight line connection. Especially if the segment is long and the collision is close to the start or the end of the segment, placing the new vertex in the middle is not an appropriate approach.

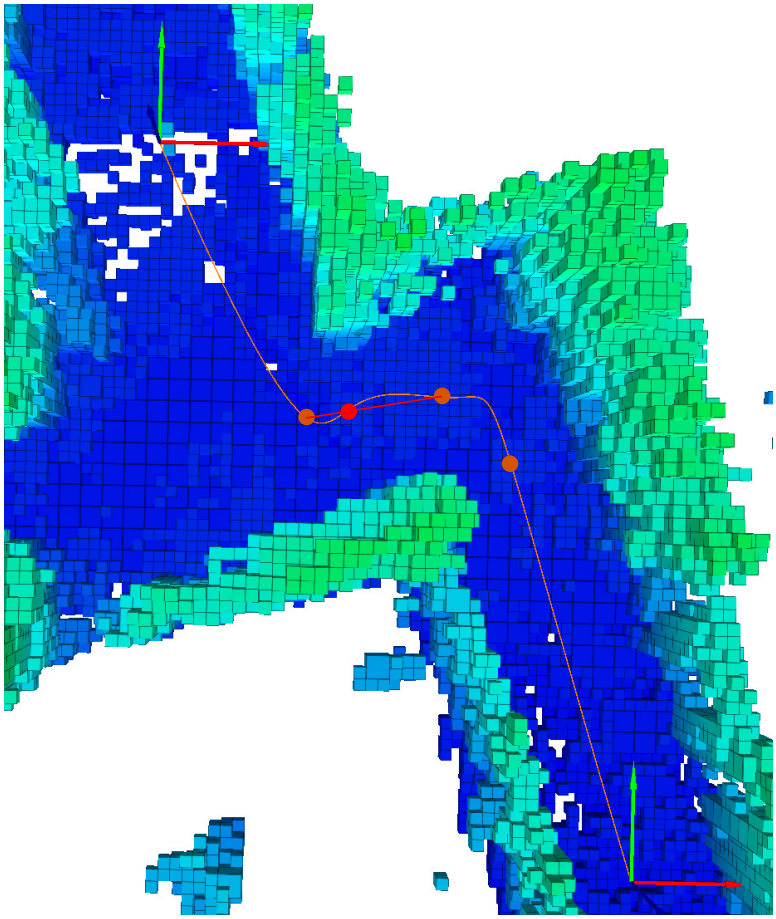

Figure 4.3: Polynomial trajectory after a vertex extension, whereas the new vertex is represented by a red dot. The red line is not part of the trajectory but depicts the straight line connection of this segment.

To get an idea in which part of the trajectory the collision takes place, the length of the trajectory from the start of the segment to the collision is compared to the total trajectory length in this segment. This ratio is then mapped to the straight line solution and the additional vertex is placed. As explained in section 3.3.2, the trajectory is in collision as soon as there is any obstacle inside the bounding box. Applied to the trajectory in figure 4.2 this means that the collision is detected in the first half of the second segment before the trajectory itself proceeds through the wall of the hallway. Hence, the new vertex in figure 4.3 is closer to the start of the second segment than to the end.

4. Perform a nonlinear optimization on the collision-free trajectory

Using the trajectory depicted in figure 4.3 as initial values a nonlinear optimization is performed. The optimized trajectory is depicted in figure 4.4.

As mentioned in section 2.5, the pathway of a snap minimized trajectory is mainly determined by the ratio of the segment times. Since the the ratio of the segment times of the trajectory in figure 4.3 is not optimal the pathway contains unnecessary detours.

After the nonlinear optimization, which includes an implicit optimization of the ratio of the segment times, the pathway is smoother and more suitable for a dynamic UAV flight.

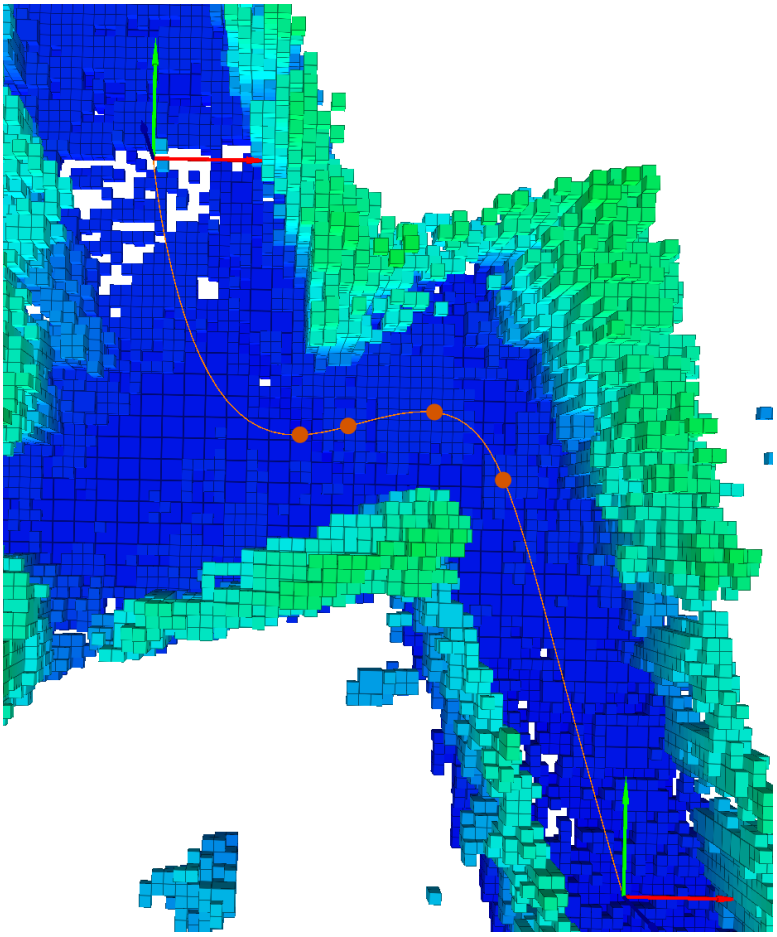

Figure 4.4: The optimized trajectory with 5 segments.

5. If the optimized solution is in collision, extend the existing vertices by a vertex located on the straight line solution and restart the nonlinear optimization. Repeat this step until the optimized trajectory is collision-free

In this example the optimized trajectory is collision free without any further vertex extensions. Since the optimization smooths the trajectory but does not change the pathway of the trajectory completely, this is the general case.

Nevertheless, if the optimized trajectory is in collision the vertex extension is equivalent to the procedure depicted in figure 4.3.

4.1.2 Enlargement of the Bounding Box

The initial trajectory passing through the vertices of the RRT* solution can be in collision although the straight line solution is collision free. This scenario is depicted in figure 4.2.

As demonstrated, the collision can be bypassed by a vertex extension. But an additional vertex means additional optimization variables (endpoint derivatives and segment times). Thus not only the nonlinear optimization takes longer but the trajectory is more restricted and looses the possibility of more efficient and dynamical maneuvers.

Regarding the drawbacks listed above, it is preferable to avoid the necessity of a vertex extension. Therefore the bounding box is enlarged during the RRT* algorithm and then reduced to its original size for the trajectory planning. The resulting enlarged security corridor reduces the possibility of a collision in the initial trajectory. This enlargement of the bounding box can not be arbitrary high since the RRT* algorithm would no longer be able to find any solution through a densely packed environment if the bounding box for the straight line solution is to high.

In this master thesis the individual dimension of the bounding box are enlarged by 20% for the straight line solution. This range reduces the necessity of a vertex extension but still enables the RRT* algorithm to find a solution through a densely packed environment.

4.2 RRT* Replanning

It may happen that the straight line solution in a specific segment is very unsuitable to convert into a polynomial solution. In this case, this specific segment is still in collision after several vertex extensions. If so, it is reasonable to do a RRT* straight line replanning of this segment. Due to de randomness of the RRT* algorithm, the straight line solution will change.

Figure 4.5 is a schematic depiction of a initial polynomial solution with 4 segments. The vertices are numbered and illustrated as dots initiating with the start vertex (vertex 0) on top. In this example, the collision is in the third segment (i.e. between vertex 2 and vertex 3). If this collision, marked by the red cross, can not be eliminated with multiple vertex extension a RRT* straight line replanning makes sense. In other words, vertex 2 and vertex 3 should be relocated. In order to achieve this, vertex 1 is used as the start vertex and vertex 4 is used as the goal vertex of a new RRT* straight line solution. This replanning is illustrated with a green arrow. As mentioned, the randomness of the RRT* algorithm leads to a new solution between vertex 1 and vertex 4.

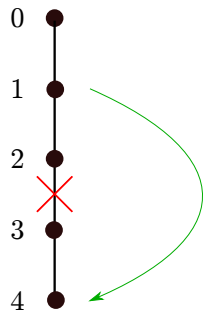

Figure 4.5: Schematic depiction of a initial polynomial solution with 4 segments. There is a collision in the third segment, marked as red cross. The replanning takes place between vertex 1 and vertex 4.

4.2.1 Threshold for the RRT* Replanning

There has to be a user specified threshold which defines after how many vertex extension a segment is considered a difficult segment. In this master thesis, the RRT* replanning is triggered after one unsuccessful vertex extensions.

Since one cycle of vertex extension defines a new vertex in every segment which is in collision, it is not trivial to find the difficult segment in the initial solution.

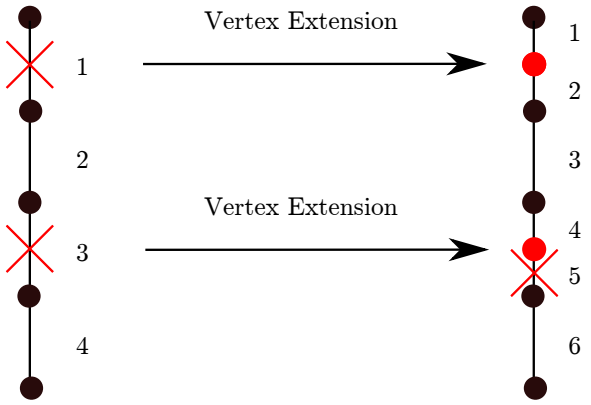

Figure 4.6: One cycle of vertex extension. In the initial solution on the left-hand side of the figure there is a collision in segment 1 and segment 3. After one cycle of vertex extension there are 6 segments.

In figure 4.6 the segments are numbered. The left-hand side depicts the initial solution with a collision in segment 1 and segment 3. For both segments, a new vertex is added an illustrated as a red dot in the right-hand side of figure 4.6. To find the difficult segment, 4 pairs of numbers have to be stored. The pairs consists of the initial segment which is in collision and of the corresponding segment after the vertex extension.

The four pairs resulting from figure 4.6 are listed in the following table:

Pair	Initial Segment	Extended Segment
i)	1	1
ii)	1	2
iii)	3	4
iv)	3	5

Table 4.1: Four pairs of numbers, representing possible difficult segments. Every initial segment is connected to two extended segments.

In the example depicted in figure 4.6 there is a collision in segment 5 after the vertex extension. Therefore, pair iv) gets significant and segment 3 (from the initial straight line solution) is exposed as a difficult segment.

Once the difficult segment is located, the RRT* replanning can be performed as illustrated in figure 4.5.

4.3 Overall Implementation

All the features which are discussed so far can now be combined to the final implementation:

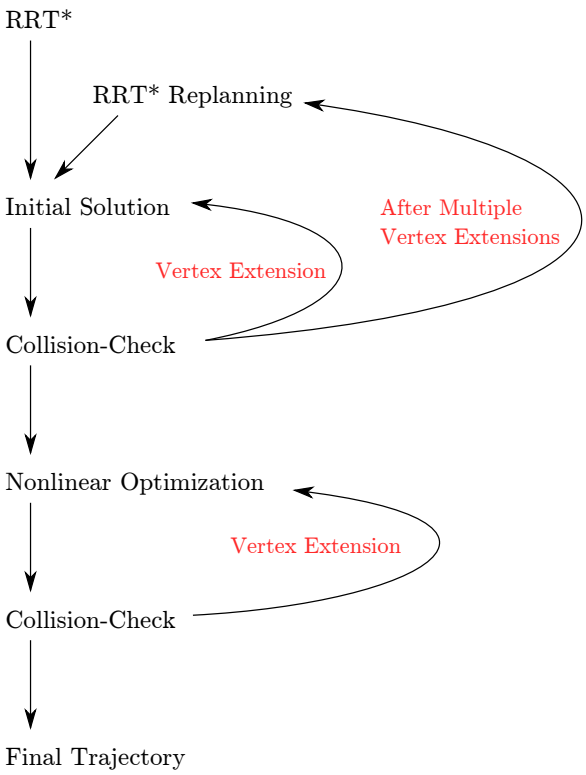

Figure 4.7: Schematic depiction of the final implementation of the path planning algorithm.

Figure 4.7 is simplified and does not describe all the necessary information but is useful to understand the basic process. For instance, the schematic depiction does not explain exactly how to proceed after the RRT* replanning and how many replannings should be allowed.

4.3.1 Store the Best Initial Solution

As mentioned in section 4.2.1 the RRT* replanning is triggered after one unsuccessful vertex extensions. Caution, if a vertex extension is performed and this leads to a collision in a different part of the trajectory but the segment which was in collision in the initial solution is now collision free, no RRT* replanning is executed.

A difficult segment in the initial solution does not mean that is impossible to find a collision free polynomial trajectory. Meaning the polynomial solution which is still in collision after one cycle of vertex extension, depicted in the right-hand side of figure 4.6, can be collision-free after additional vertex extensions. If possible, the initial RRT* straight line solution is converted into a collision free polynomial trajectory with multiple vertex extensions and the polynomial trajectory and the corresponding total cost of the trajectory is stored.

Not until then, the RRT* replanning is performed. The difficult section of the original straight line solution is then replaced by the replanned section and a polynomial trajectory passing trough the vertices is computed. The new trajectory on his part can be in collision and vertex extensions have to be performed until the trajectory is collision-free. Once the trajectory is collision-free, the total cost of the new trajectory is compared to the total cost of the first trajectory and the one with the lower cost is used for the nonlinear optimization.

4.4 Robot Operating System

The whole program is built using the Robot Operating System (ROS) [11]. To be exact, it is built as a ROS node. A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, services, and the Parameter Server. A very powerful feature of ROS is "Dynamic Reconfigure". This tool allows the user to change parameters during runtime. Runtime is the period of time when a program is running. It begins when a program is opened (or executed) and ends with the program is quit or closed.

4.4.1 Dynamic Reconfigure

In this master thesis, several imported parameters are modifiable by Dynamic Reconfigure. Two of them mainly determine the performance of the RRT* algorithm:

- γ
- *endRRT*

The γ parameter determines the amount of rewiring and is described in detail in section 3.3.1. *endRRT* defines how many additional RRT* iteration are performed after a collision-free straight line connection is found. Thus, $endRRT = 2000$ leads to 2000 additional iterations in which the algorithm can improve the straight lines solution via rewiring.

The next two parameters which are included into Dynamic Reconfigure are used to modify NLopt:

- J_{rel}
- k_T

J_{rel} is defined in section 2.6 and is a ending criteria for NLopt. The weighting factor k_T is defined in equation 2.10 in section 2.4 and specifies the aggressiveness of a trajectory.

The next parameter concerns the RRT* replanning. As explained in section 4.2, a RRT* replanning is beneficial if there are difficult passages in the initial straight line solution. It may happen, that the straight line solution with the replanned segments still has difficult passages.

- *maxReplanning*

maxReplanning defines the max. number of RRT* replannings. After this number of replannings, the optimization starts with the best solution so fare. If no collision-free solution has been found, the complete RRT* planning starts again.

The last parameter is bool parameter (either "true" or "false") affects the start vertex of a trajectory:

- *currentPose*

currentPose defines if the start vertex is either manually chosen or the current pose of the UAV. In this master thesis, the current pose is provided by Vicon [13].

Chapter 5

Results

5.1 Performance of the RRT* Algorithm

In this master thesis, the performance of the RRT* algorithm itself was not improved but the parameters were tuned to serve the nonlinear optimization in an optimal manner.
As mentioned in section 3.3, the calculation time of the RRT* algorithm is mainly defined by the "rewiring" and therefore by the user specified parameter γ (used in equation 3.1). A good straight line solution, whereas good means a small length of the straight line solution, does not necessarily lead to a good polynomial trajectory. The influence of the user specified parameter γ on the final trajectory and the corresponding calculation time is evaluated in this section.
Please call to mind that a small γ parameter means little rewiring and a big γ parameter means lots of rewiring. The impact of the γ parameter can be looked up in figure 3.1 and figure 3.2.

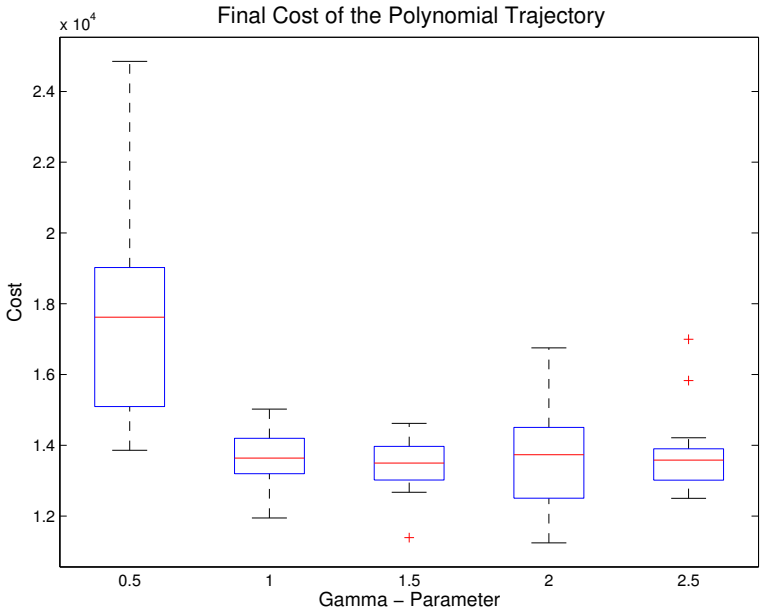

Figure 5.1: The x-axis depicts different γ parameters and the y-axis depicts the final cost of the polynomial trajectory. The red mark illustrates the median.

Figure 5.1 depicts the boxplot for different γ parameters. Each dataset consists of 15 measurements. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme datapoints the algorithm considers to be not outliers, and the outliers are plotted individually.

It becomes apparent that the small amount of rewiring associated with $\gamma = 0.5$ is to little to obtain a good polynomial trajectory. The 4 remaining datasets have a similar performance in terms of the final trajectory cost.

The total computational times for the five different γ parameters are depicted in figure 5.2. The total computational time is the combined duration of the RRT* algorithm and the nonlinear optimization. In contrast to the final cost, the total computational times for $\gamma = 1$, $\gamma = 1.5$, $\gamma = 2$ and $\gamma = 2.5$ are distinct.

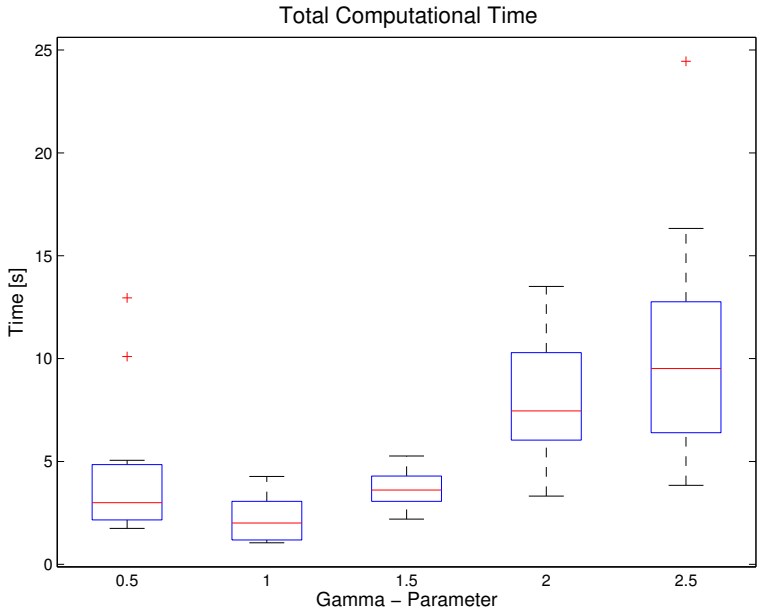

Figure 5.2: The x-axis depicts different γ parameters and the y-axis depicts the total computational time. The red mark illustrates the median.

Figure 5.2 shows that the computational time increases significant if γ is larger than 1.5. This is simply because the RRT* algorithm needs more time for rewiring. Furthermore, $\gamma = 0.5$ (which is not a good choice sine the final cost is too high) has 2 outliers. In this 2 cases the straight line solution is not enough target-orientated and multible vertex extensions are required to obtain a collision-free trajectory.

Combining the results from figure 5.1 and figure 5.2, a γ parameter in the range of 1 to 1.5 lead to the best performance.

5.2 Performance of NLopt

To test the performance of NLopt, a nonlinear optimization library, trajectories of different length were optimized. Figure 5.3 depicts a start vertex and 6 different goal vertices. The figure is in bird's eye perspective and shows a crossing of different hallways. The blue cells represents the floor and the green cells represents the walls.

In some of the hallways, there are objects blocking (part of) the way. This passages are tagged with "Bottleneck". Please note that other passages which may look like a bottleneck, such as the top right corner, are not blocked. The green boxes in this passage are part of the ceiling.

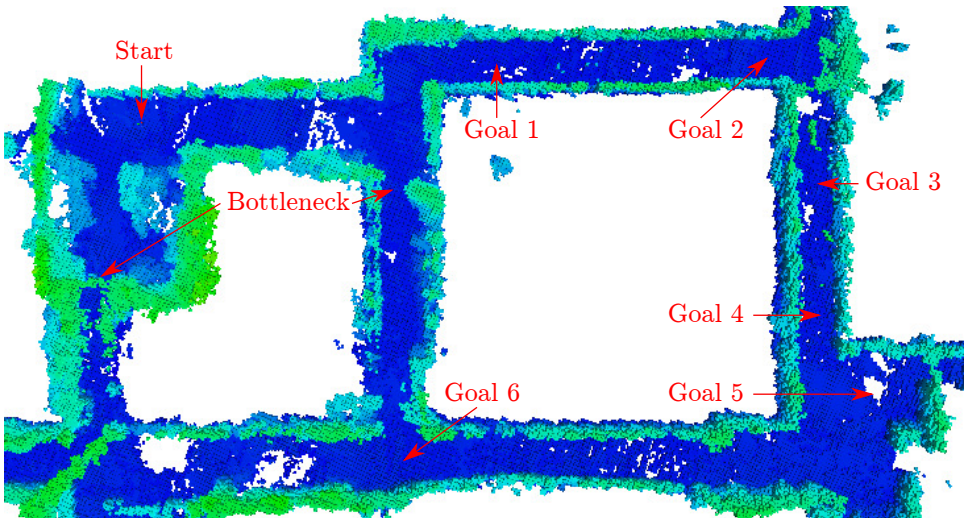

Figure 5.3: Bird's eye perspective on hallways in the "ML" building of the ETH Zurich. One start vertex and 6 different goal vertices are depicted

The bottleneck in the center of figure 5.3 gets significant for large bounding boxes. If the dimension of the bounding box are larger than $0.5m$ x $0.5m$ x $0.5m$ the trajectory is not able to pass the bottleneck. Hence, a trajectory with a large bounding box has to go all the way around to proceed from the start vertex to "Goal 6" as depicted in figure 5.4.

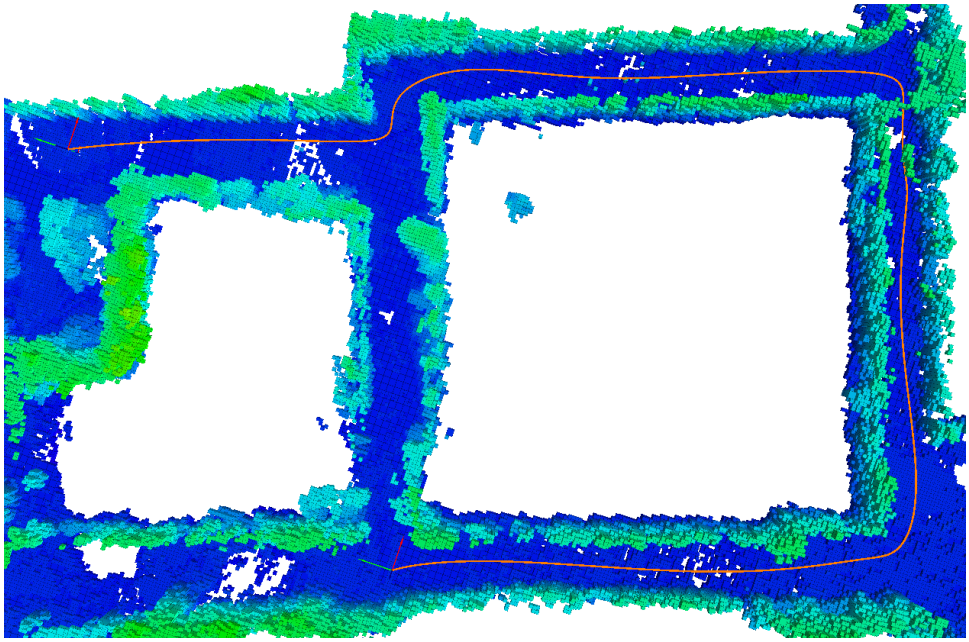

Figure 5.4: Trajectory from the start vertex to "Goal 6". Due to the bottleneck in the hallway in the middle, the trajectory has to proceed all the way around.

The trajectories from the start vertex to the 6 different goal vertices (depicted in figure 5.3) are examined for number of segments and the duration of NLopt. The only ending criteria for NLopt was the relative criteria on the total cost $f_{rel} = 0.02$. In addition to the number of segments, the number of corresponding optimization variables (endpoint derivatives d_p and segment times T_i) are calculated. For all the "free" vertices (which are neither the start nor the goal vertex) there are 12 optimization variables, namely the velocity, the acceleration, the jerk and the snap in the 3 dimensions. Furthermore, the segment times are optimization variables.

Since the number of free vertices is the number of segment n_{seg} minus 1, the equation for the number of optimization variables n_{var} is:

$$n_{var} = (n_{seg} - 1) \cdot 12 + n_{seg} \tag{5.1}$$

The results for the 6 different goal vertices are listed in the following table:

Goal Vertex	Segments	Optimization Variables	Optimization Time
Goal 1	5	53	$1.15s$
Goal 2	8	92	$2.94s$
Goal 3	11	131	$7.80s$
Goal 4	15	183	$42.96s$
Goal 5	19	235	$93.08s$
Goal 6	20	248	$347.62s$

Table 5.1: The number of segments of the collision free polynomial solution and the corresponding number of optimization variables. The time needed by NLopt with the ending criteria $f_{rel} = 0.02$.

Table 5.1 shows that the optimization time (i.e. the duration of the NLopt algorithm) increases significant for a large amount of optimization variables. Comparing the result of "Goal 5" to "Goal 6" it becomes clear, that the number of optimization variables is not the only factor which determines the optimization time. If the optimization values of the initial solution are close to the global minimum of the cost function, the optimization takes less time and vice versa. Still, the number of optimization variables is the main factor.

Please note that the RRT* algorithm was only executed once for every goal vertex. Due to the randomness of the RRT* algorithm, the number of segments could change for unchanged start and goal vertices.

5.3 Reduction of the Optimization Variables

The results in table 5.1 have shown, that the performance of NLopt decreases with a large amount of optimization variables. In cases where the optimization times is significant (i.e. online planning) the number of optimization parameters has to be reduced. The benefits and the drawbacks of a cost function with fewer optimization variables is discussed in the next section.

5.3.1 Cost Function Without Endpoint Derivatives

To call to mind, the cost function for the nonlinear optimization is

$$J_{total} = \begin{bmatrix} d_f \\ d_p \end{bmatrix}^T \begin{bmatrix} R_{ff} & R_{fp} \\ R_{pf} & R_{pp} \end{bmatrix} \begin{bmatrix} d_f \\ d_p \end{bmatrix} + k_T \cdot \sum_{i=1}^{N} T_i \tag{5.2}$$

where the optimization variables are the segment times T_i and the unspecified endpoint derivatives d_p. k_T is a user specified weighting factor and d_f is the vector containing the fixed endpoint derivatives. The 4 sub-matrices of R are containing the quadratic snap rearranged according to the fixed and unspecified endpoint derivatives.
As discussed in section 2.4.1, the optimum of this cost function can not be found analytically but with a nonlinear optimization. Only the snap minimized solution of a trajectory with fixed segment times can be computed analytically according to

$$d_p^* = -R_{pp}^{-1} \cdot R_{fp}^T \cdot d_f \tag{5.3}$$

where the optimal unspecified endpoint derivatives d_p^* are a function of the fixed derivatives d_f and two of the submatrices (R_{pp}, R_{fp}) of R.

Since the endpoint derivatives d_p can be found analytically once the segment times T_i are known, d_p can be excluded from the optimization variables. In other words, in every optimization steps the segment times are modified by NLopt. Then d_p^* is computed with the current segment times and the total cost is calculated according to equation 5.2.

Recapitulating the new cost function:

- The segment times T_i are the only optimization variables.
- The optimal unspecified endpoint derivatives d_p^* are computed analytically (equation 5.3) every optimization step with the current segment times.
- The cost function (equation 5.2) gets minimized.

5.3.2 Comparison of the Analytical Solvers

Before we can compare the performance of the approach with the full number of optimization variables (in the interests of simplification called approach A) to the approach with the reduced number of optimization variables (approach B) we have to examine the performance of the analytical solver which solves equation 5.3.

An analytical or linear solver is able to solve an single matrix equation:

$$A \cdot x = b \tag{5.4}$$

Indeed, equation 5.3 is a single matrix equation where $-R_{pp}$ is represented by A and x is the vector containing the optimization variables (i.e. d_p). Furthermore, $R_{fp}^T \cdot d_f$ can be taken together as vector b.

In this master thesis, three different strategies have been implemented to solve equation 5.3. All of them are part of "Eigen". Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. The definition of the implemented strategies can be found on the Eigen-homepage [12]:

FullPivLU

This class represents a LU decomposition of any matrix, with complete pivoting: the matrix A is decomposed as $A = PLUQ$ where L is unit-lower-triangular, U is upper-triangular, and P and Q are permutation matrices. This is a rank-revealing LU decomposition. The eigenvalues (diagonal coefficients) of U are sorted in such a way that any zeros are at the end.

$FullPivLU$ is a very accurate technique but rather slow.

HouseholderQR

This class performs a QR decomposition of a matrix A into matrices Q and R such that $A = QR$ by using Householder transformations. Here, Q a unitary matrix and R an upper triangular matrix. The result is stored in a compact way compatible with $LAPACK$. Note that no pivoting is performed. This is not a rank-revealing decomposition.

$HouseholderQR$ is a fast technique but in general less accurate than $FullPivLU$.

Inverse

For small fixed sizes up to $4x4$, this method uses cofactors. In the general case, this method uses class $PartialPivLU$. This class represents a LU decomposition of a square invertible matrix, with partial pivoting: the matrix A is decomposed as $A = PLU$ where L is unit-lower-triangular, U is upper-triangular, and P is a permutation matrix.

The performance of the inverse is compared experimentally to the other techniques.

First, the three techniques were tested with a long trajectory. The initial solution has been computed for a trajectory with 300 segments with random vertices. The identical set of vertices was used for all the three techniques and the results are listed in the following table:

Technique	Computational Time
FullPivLU	$19.52s$
HouseholderQR	$3.34s$
Inverse	$2.07s$

Table 5.2: Comparison of the three analytical solvers for a trajectory with 300 segments. Only the initial solution has been computed.

In a next step, the three techniques has been compared in the nonlinear optimization. The RRT* algorithm was executed to find a straight lines solution from the start vertex to "Goal 6" in figure 5.3. The collision-free polynomial trajectory had 23 segments. The trajectory was optimized using $FullPivLU$ but in every iteration $HouseholderQR$ and $Inverse$ has been computed as well and the total time of the individual techniques has been stored. The optimization terminated after 883 iterations.

The result of the three techniques are listed in the following table:

Technique	Computational Time
FullPivLU	$1.134s$
HouseholderQR	$0.704s$
Inverse	$0.285s$

Table 5.3: Comparison of the three analytical solvers for a trajectory with 23 segments. The nonlinear optimization was performed (883 iterations).

As mentioned above $FullPivLU$ is very accurate. In the two above noted tests (and also in other performed tests) the other two techniques led to the same result as $FullPivLU$, meaning there were no numerical instability issues. This is due to the fact, that R_{pp} is quadratic and a band matrix. A band matrix is a sparse matrix whose non-zero entries are confined to a diagonal band. Band matrices are beneficial from a computational point of view.

Since the numerical stability is given for all of the techniques the determining factor is the computational time. Therefore, the $Inverse$ is the preferable technique!

Please note that the $FullPivLU$ and $HouseholderQR$ have to solve equation 5.3 separate for each dimension. In contrast, the inverse has to be computed once and can then be applied to the three dimensions. This means that $HouseholderQR$ would be faster than $Inverse$ if only on dimension had to be optimized. However, in this master thesis with three dimension the inverse can display his advantages.

5.3.3 Comparison of the Different Approaches

Table 5.1 depicts the optimization time from the start vertex to "Goal 6" using the approach with the full number of optimization variables (approach A). Table 5.3 depicts the computational time for the same start and goal vertex using the approach with the reduced number of optimization variables (approach B). Caution, the time needed for the limit check v_{max} and a_{max} is not included in table 5.3.

Although the number of segments are slightly different (based on the randomness of RRT*) and table 5.3 does not include the time needed for the limit check it is obvious that approach B with a computational time of $0.285s$ is much faster than approach A with a optimization time of $347.62s$.

During the further procedure of comparing the two approaches, the focus is on the final trajectory cost. As a consequence of the reduction of the optimization variables, approach B is not able to reach the global minimum of the cost function in all cases. Approach A on the other hand, is theoretical allays able to reach the global minimum of the cost function. In practice, depending on the initial values and on the ending criteria, it happens that the optimization only finds a local minimum. The performance of the two strategies has therefore to be testes experimentally.

In addition, a third approach is tested. In one trajectory optimization, approach B is applied in the first place and this result is than used as the initial value for approach A. Thus, the third approach is a combination of approach A and approach B and is called approach BA because of the chronological order.

To start with, the simple set of vertices from table 2.1 is reused. As depited in figure 2.3, a weighting factor of $k_T = 100$ does not lead to any values near the limitation. In this case, the global minimum of the cost function 2.3 can be found by only manipulating the segment times T_i. The endpoint derivatives d_p as optimization variables are redundant. The only termination criteria for NLopt was $J_{rel} = 0.001$.

Approach	Optimization Time	Final Cost
A	$1.48s$	944.887
B	$0.01s$	944.319
BA	$0.04s$	944.318

Table 5.4: Optimization time and final cost for a trajectory with 2 segments. Weighting factor k_T was set to 100 and NLopt ending criteria J_{rel} was set to 0.001.

As explained above, the final cost depicted in table 5.4 is (beside some numerical differences) the same for the three approaches. Approach B is the fastest and approach A is by fare the slowest.

Now, the weighting factor k_T is increased to 2000 (corresponding to figure 2.5). This aggressive trajectory stays at the velocity limit for a certain time. Due to that, the segment times T_i are no longer sufficient to reach the global minimum of the cost function.

Approach	Optimization Time	Final Cost
A	$0.17s$	15295.9
B	$0.02s$	15994.5
BA	$0.12s$	15047.9

Table 5.5: Optimization time and final cost for a trajectory with 2 segments. Weighting factor k_T was set to 2000 and NLopt ending criteria J_{rel} was set to 0.001.

As can be seen in table 5.5, approach B is again the fastest one. Due to the velocity limitation, approach B is not able to reach the global minimum of the cost function. Comparing the cost of approach A (which could theoretically reach the global minimum) and approach BA, is seems surprising that approach A could not find a trajectory as good as approach BA did. The reason is that approach A got stuck in a local minimum.

But how could approach BA overcome this local minimum? Part A of approach BA only starts as soon as part B is terminated. Therefore, the initial values for part A are closer to the global minimum and the estimation of the initial step size for NLopt is improved. Both factors lead to a better overall performance of the NLopt algorithm and approach BA can find the best trajectory.

Interim Conclusion

As can bee seen in table 5.1, approach A is very slow if the number of segments is high. In addition, approach BA performs better in terms of final cost as can be seen in talbe 5.5. Taken together, approach A is the approach with the worst performance and is not used any longer.

So far, approach B and approach BA have been compared for a trajectory with 2 segments. In a next step, approach BA was used to find a trajectory from the start vertex to "Goal 1" in figure 5.3. For 5 iterations, the number of segments fluctuated between 6 and 9, i.e. between 55 and 105 optimization variables for approach A. The parameters for this examination were: $\gamma = 1$, $J_{rel} = 0.02$ and $k_T = 5000$. A high k_T guaranties that the limitation on velocity/acceleration are called into action. Otherwise, part B would have no effect at all. The optimization times for both parts are measured individually, i.e. optimization time A is the additional time after part B.

Segments	Opt. Time B	Final Cost B	Opt. Time A	Final Cost A
7	$0.54s$	51577.7	$2.79s$	51575
6	$0.09s$	54142.9	$1.72s$	53917.5
9	$0.38s$	56145.4	$8.53s$	56129.6
7	$0.49s$	57110.4	$1.47s$	57104.4
6	$0.03s$	56020.7	$1.76s$	55553.2

Table 5.6: Multiple trajectories from start to "Goal 1". Parameters were: $\gamma = 1$, $J_{rel} = 0.02$ and $k_T = 5000$.

Conclusion

Apparent from table 5.6, the benefit of connecting part A in series with part B is little. Furthermore, part B (and therefore approach B) is much faster than part A.

As a consequence, approach B has the best overall performance. Approach BA only makes sense if a trajectory with little segments has to be optimized to its limits.

Bibliography

[1] R. He, A. Bachrach, M. Achtelik, A. Geramifard, D. Gurdan, S. Prentice, J. Stumpf, and N. Roy: *On the design and use of a micro air vehicle to track and avoid adversaries*. The Int. Journal of Robotics Research, vol. 29, pp. 529-546, 2010.

[2] D. Colling, O. A. Yakimenko, J. F. Whidborne, and A. K. Cooke: *A prototype of an autonomous controller for a quadrotor UAV*. In Proceedings of the European Control Conference, Kos, Greece, 2007, pp. 1-8.

[3] M. Hehn and R. D'Andrea : *Quadrocopter trajectory generation and control*. In International Federation of Automatic Control (IFAC), World Congress 2011, 2011.

[4] S. Lupashin, A. Schollig, M. Sherback, and R. D'Andrea: *A simple learning strategy for high-speed quadrocopter multi-flips*. In Proc. of the IEEE Int. Conf. on Robotics and Automation, Anchorage, AK, May 2010, pp. 1642-1648.

[5] Y. Bouktir, M. Haddad, and T. Chettibi: *Trajectory Planning for a Quadrotor Helicopter*. In Mediterranean Conference on Control and Automation, Jun. 2008, pp. 1258-1263

[6] D. Mellinger and V. Kumar: *Minimum snap trajectory generation and control for quadrotors*. In International Conference on Robotics and Automation, 2011, pp. 2520-2525.

[7] M. Likhachev, G. Gordon and S. Thrun: *ARA*: Anytime A* with Provable Bounds on Sub-Optimality* . Advances in Neural Information Processing Systems, vol. 16, 2003

[8] C. Richter, A. Bry, and N, Roy: *Polynomial Trajectory Planning for Quadrotor Flight*. In International Conference on Robotics and Automation, 2013.

[9] NLopt: *NLopt, an open-source library for nonlinear optimization.*
http : //ab − initio.mit.edu/wiki/index.php/NLopt

[10] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard: *OctoMap: An efficient probabilistic 3D mapping framework based on octrees*. Autonomous Robots, 2013.

[11] ROS: *The Robot Operating System (ROS) is a set of software libraries and tools for robot applications.*
http : //www.ros.org/

[12] EIGEN: *A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.*
http : //eigen.tuxfamily.org/index.php?title = Main_Page

[13] VICON: *Vicon Tracker is a powerful object tracking solution providing unrivaled data accuracy for integration in to 3D applications.*
http : //www.vicon.com/