Danmarks Tekniske Universitet

DTU

Unmanned Autonomous Systems

June 2022

July 20, 2023

Authors

s183955 - Alan Mansour
s212075 - Martin Mikšík
s212502 - Wojciech Mysliwiec
s202711 - Zhicun Tan

Section	A.M.	M.M.	W.M	Z.T.
Ex 1	50%	50%	0%	0%
Ex 2	30%	0%	35%	35%
Ex 3	0%	0%	75%	25%
Ex 4	50%	50%	0%	0%
Ex 5	50%	50%	0%	0%
Ex 6	25%	25%	25%	25%
Ex 7	20%	20%	30%	30%

Table 1: Work commitment expressed in percentage.

Contents

Introduction

The Team

This is a report of a Project for Course 31390 Unmanned Autonomous Systems at Denmark Technical University.

The authors of this paper participated equally in the Project.

For the Simulation purposes, a Matlab framework [1] was used.

The Goal

This project assignment aims to practice the topics learned in the Course. We will apply linear and rotational transformations in Euler angles, show the possibilities of Quaternion applications, apply control theory linearization to build a SIMULINK controller, explore path trajectory planning algorithms and simulate the results in the Framework [1] as well as employ the work on real quadcopter to autonomously solve a maze and navigate with reflective-driven optical tracking sensors[2]

1 Frame transformations

In this exercise, we will look at different transformations and rotations in between frames in Euler angles as well as in quaternions.

1.1 Finding the rotation matrix

We will now find the rotation matrix corresponding to the set of Euler angles ZXZ and describe the procedure used to find the solution.

RPY (Roll-Pitch-Yaw angles) angles are a set of Euler angles following the order ZYX.

For a rotation about the x-axis, y - and z-coordinates are rotated, and the x-coordinate remains unchanged. Such rotation can be expressed by a rotation matrix in the following form:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{1}$$

The same logic applies on the z-axis. For a rotation about the z-axis, x - and y-coordinates are rotated, and the z-coordinate remains unchanged. Such rotation can be expressed by a rotation matrix in the following form:

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

To express the rotation matrix corresponding to ZXZ, we multiply the matrices for each axis using arbitrary angles: a rotation of ψ around the z-axis and a rotation of ϕ around the x-axis. It is worth to note that the three-dimensional rotations do not commute. Thus, the order of multiplication matters.

$$R_z(\Psi_2) \cdot (R_x(\phi) \cdot R_z(\Psi_1)) =$$

$$\begin{bmatrix} \cos(\Psi_2)\cos(\Psi_1) - \sin(\Psi_2)\cos(\phi)\sin(\Psi_1) & -\cos(\Psi_2)\sin(\Psi_1) - \sin(\Psi_2)\cos(\phi)\cos(\Psi_1) & \sin(\Psi_2)\sin(\phi) \\ \sin(\Psi_2)\cos(\Psi_1) + \cos(\Psi_2)\cos(\phi)\sin(\Psi_1) & -\sin(\Psi_2)\sin(\Psi_1) + \cos(\Psi_2)\cos(\phi)\cos(\Psi_1) & -\cos(\Psi_2)\sin(\phi) \\ \sin(\phi)\sin(\Psi_1) & \sin(\phi)\cos(\Psi_1) & \cos(\phi) \end{bmatrix} \tag{3}$$

1.2 Finding the inverse solution

Now we will discuss the inverse solution for the Euler angles ZYZ in case $s_\theta = 0$.

We first need to express the rotation matrix corresponding to ZYZ. We do so by following the same procedure from the previous exercise. For a rotation about the y-axis, x- and z-coordinates are rotated, and the y-coordinate remains unchanged. Such rotation can be expressed by a rotation matrix in the following form:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

To determine the set of Euler angles corresponding to a given rotation matrix, we introduce the inverse problem. We first abstract the above rotation matrices:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

From r_{13} and r_{23}, we can get Ψ_2 as following:

$$\frac{r_{23}}{r_{13}} = \frac{\sin(\psi_2)\sin(\theta)}{\cos(\psi_2)\sin(\theta)} = \frac{\sin(\psi_2)}{\cos(\psi_2)} = \tan(\psi_2)$$

$$\psi_2 = \arctan\left(\frac{r_{23}}{r_{13}}\right)$$

From r_{13}, r_{23} and r_{33}, we can get θ as following:

$$\frac{\sqrt{r_{13}^2 + r_{23}^2}}{r_{33}} = \frac{\sqrt{(\cos(\psi_2)\sin(\theta))^2 + (\sin(\psi_2)\sin(\theta))^2}}{\cos(\theta)}$$
$$= \frac{\sqrt{\sin^2(\theta)\left(\cos^2(\psi_2) + \sin^2(\psi_2)\right)}}{\cos(\theta)} = \frac{\sqrt{\sin^2(\theta)(1)}}{\cos(\theta)} = \frac{\sin(\theta)}{\cos(\theta)} = \tan(\theta)$$

$$\theta = \arctan\left(\frac{\sqrt{r_{13}^2 + r_{23}^2}}{r_{33}}\right)$$

From r_{31} and r_{32}, we can get ψ_1 as following:

$$\frac{r_{32}}{-r_{31}} = \frac{\sin(\theta)\sin(\psi_1)}{-(-\sin(\theta)\cos(\psi_1))} = \frac{\sin(\psi_1)}{\cos(\psi_1)} = \tan(\psi_1)$$

$\psi_1 = \arctan\left(\frac{r_{32}}{-r_{31}}\right)$

Now, when $s_\theta = 0, \theta$ must be either 0 or π in the unit circle, hence, in case of $\theta = 0$, the matrix becomes

$$R_z(\psi_2) \cdot (R_y(0) \cdot R_z(\psi_1)) = \begin{bmatrix} \cos(\psi_2)\cos(\psi_1) - \sin(\psi_2)\sin(\psi_1) & -\cos(\psi_2)\sin(\psi_1) - \sin(\psi_2)\cos(\psi_1) & 0 \\ \sin(\psi_2)\cos(\psi_1) + \cos(\psi_2)\sin(\psi_1) & \cos(\psi_2)\cos(\psi_1) - \sin(\psi_2)\sin(\psi_1) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

We can simplify the matrix using Ptolemy's identities by hand to the following form:

$$\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$
$$\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$$
$$\sin(\alpha - \beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta$$
$$\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$$

$$\begin{bmatrix} \cos(\psi_2+\psi_1) & -\sin(\psi_2+\psi_1) & 0 \\ \sin(\psi_2+\psi_1) & \cos(\psi_2+\psi_1) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In case of $\theta = \pi$, the matrix becomes

$$R_z(\psi_2)\cdot(R_y(\pi)\cdot R_z(\psi_1)) = \begin{bmatrix} -\cos(\psi_2)\cos(\psi_1)-\sin(\psi_2)\sin(\psi_1) & \cos(\psi_2)\sin(\psi_1)-\sin(\psi_2)\cos(\psi_1) & 0 \\ \cos(\psi_2)\sin(\psi_1)-\sin(\psi_2)\cos(\psi_1) & \sin(\psi_2)\sin(\psi_1)+\cos(\psi_2)\cos(\psi_1) & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{5}$$

By simplifying, we get

$$\begin{bmatrix} -\cos(\Psi_2-\psi_1) & -\sin(\Psi_2-\psi_1) & 0 \\ -\sin(\Psi_2-\Psi_1) & \cos(\Psi_2-\Psi_1) & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Considering both cases, we can only get the sum or difference of angles ψ_2 and ψ_1. So, from r_{12} and r_{22}, we can get $\psi_2+\psi_1$, when $\theta = 0$, as following:

$$\frac{-r_{12}}{r_{22}} = \frac{-(-\sin(\Psi_2+\psi_1))}{\cos(\Psi_2+\Psi_1)} = \frac{\sin(\Psi_2+\psi_1)}{\cos(\Psi_2+\psi_1)} = \tan(\psi_2+\psi_1)$$

$$\psi_2+\psi_1 = \arctan\left(\frac{-r_{12}}{r_{22}}\right)$$

When $\theta = \pi$, the same procedure applies:

$$\psi_2-\psi_1 = \arctan\left(\frac{-r_{12}}{r_{22}}\right)$$

1.3 The inverse solution for Roll-Pitch-Yaw

Now we will discuss the inverse solution for Roll-Pitch-Yaw angles in the case $c_\theta = 0$.

Similarly as in the previous exercise, when $c_\theta = 0, \theta$ must be either $-\frac{\pi}{2}$ or $\frac{\pi}{2}$ in the unit circle, hence, in case of $\theta = -\frac{\pi}{2}$

$$R_z(\psi)\cdot\left(R_y\left(-\frac{\pi}{2}\right)\cdot R_x(\phi)\right) = \\ = \begin{bmatrix} 0 & -\cos(\psi)\sin(\phi)-\sin(\psi)\cos(\phi) & -\cos(\psi)\cos(\phi)+\sin(\psi)\sin(\phi) \\ 0 & -\sin(\psi)\sin(\phi)+\cos(\psi)\cos(\phi) & -\cos(\psi)\sin(\phi)-\sin(\psi)\cos(\phi) \\ 1 & 0 & 0 \end{bmatrix} \tag{6}$$

We can simplify the matrix to the following form:

$$\begin{bmatrix} 0 & -\sin(\psi+\phi) & -\cos(\psi+\phi) \\ 0 & \cos(\psi+\phi) & -\sin(\psi+\phi) \\ 1 & 0 & 0 \end{bmatrix}$$

In case of $\theta = \frac{\pi}{2}$, the matrix becomes:

$$\begin{aligned} & R_z(\psi) \cdot \left(R_y\left(\frac{\pi}{2}\right) \cdot R_x(\phi) \right) = \\ & = \begin{bmatrix} 0 & \cos(\psi)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\cos(\phi) + \sin(\psi)\sin(\phi) \\ 0 & \cos(\psi)\cos(\phi) + \sin(\psi)\sin(\phi) & \sin(\psi)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -1 & 0 & 0 \end{bmatrix} \\ & = \begin{bmatrix} 0 & \sin(\psi - \phi) & \cos(\psi - \phi) \\ 0 & \cos(\psi - \phi) & -\sin(\psi - \phi) \\ -1 & 0 & 0 \end{bmatrix} \end{aligned} \tag{7}$$

Considering both cases, we can only get the sum or difference of angles ψ and ϕ.

So, from r_{22} and r_{23}, we can get $\psi + \phi$, when $\theta = -\frac{\pi}{2}$, as following:

$$\frac{-r_{23}}{r_{22}} = \frac{-(-\sin(\psi - \phi))}{\cos(\psi - \phi)} = \frac{\sin(\psi - \phi)}{\cos(\psi - \phi)} = \tan(\psi + \phi)$$
$$\psi + \phi = \arctan\left(\frac{-r_{23}}{r_{22}}\right)$$

When $\theta = \frac{\pi}{2}$, the same procedure applies:

$$\psi - \phi = \arctan\left(\frac{-r_{12}}{r_{22}}\right)$$

1.4 The minimal rotation needed

Given a pair of unit vectors v and w (v = from and w = to), we will find the minimal rotation that brings v in w.

The scalar product of two arbitrary vectors a and b is given by

$$a \cdot b = |a||b|\cos(\theta)$$

The cross product is given by

$$|a| \times |b| = |a| \cdot |b| \sin(\theta)$$

where θ is the angle between v and w.

In our case, both v and w are unit vectors. A unit vector has a magnitude of exactly 1.

By substituting v and w, we get the following:

$$v \cdot w = |v||w|\cos(\theta) = 1 \cdot 1 \cdot \cos(\theta) = \cos(\theta)$$

$$|v \times w| = |v||w|\sin(\theta) = 1 \cdot 1 \cdot \sin(\theta) = \sin(\theta)$$

We also know that

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}$$

Thus, the minimal rotation, which is expressed by the minimal angle of rotation, is given as

$$\theta = \arctan\left(\frac{|v \times w|}{v \cdot w}\right)$$

For any other vectors, we can apply normalization to get unit vector representations.

1.5 Quaternion representation

In this subsection, we will answer the following questions:

- What is the quaternion $q1$ that represents the rotation of 180 degree about the x-axis?
- What is the quaternion $q2$ that represents the rotation of 180 degree about the z-axis?
- What rotation is represented by composite quaternion $q = q1q2$? Answer by specifying its rotation angle and axis.

The quaternion $q1$ that represents the rotation of 180 degree about the x-axis is given as

$$q1 = \cos\left(\frac{\phi}{2}\right) + i\sin\left(\frac{\phi}{2}\right) = \cos\left(\frac{\pi}{2}\right) + i\sin\left(\frac{\pi}{2}\right) = i$$

where $\phi = 180$ degrees $= \pi$ radians.

The quaternion $q2$ that represents the rotation of 180 degree about the z-axis is given as

$$q2 = \cos\left(\frac{\psi}{2}\right) + i\sin\left(\frac{\psi}{2}\right) = \cos\left(\frac{\pi}{2}\right) + k\sin\left(\frac{\pi}{2}\right) = k.$$

The composite quaternion $q = q1q2$ is given as

$$q = i \cdot k = -j = \cos\left(\frac{\theta}{2}\right) + j\sin\left(\frac{\theta}{2}\right) = \cos\left(\frac{3\pi}{2}\right) + j\sin\left(\frac{3\pi}{2}\right)$$

The computed quaternion corresponds to rotation around the y-axis with a rotation angle $\theta = 3\pi$

Detailed explanation can be found in A Tutorial on Euler Angles and Quaternions [3]

1.6 Composition of rotation matrices and quaternions

We will now compare the number of additions and multiplications needed to perform the following operations:

- Compose two rotation matrices.
- Compose two quaternions.
- Apply a rotation matrix to a vector.
- Apply a quaternion to a vector (as in Exercise 4).

We will count a subtraction as an addition, and a division as a multiplication.

Composing two rotation matrices involves matrix multiplication. A rotation matrix is a 3×3 matrix. So, when multiplying two such matrices, we do $3^3 = 27$ multiplications and $3^2 \cdot 2 = 18$ additions, since for every element in the new matrix totalling $3^2 = 9$ we do 2 additions.

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} . \begin{bmatrix} N_{11} & N_{12} & N_{13} \\ N_{21} & N_{22} & N_{23} \\ N_{31} & N_{32} & N_{33} \end{bmatrix}$$

$$= \begin{bmatrix} M_{11}N_{11} + M_{12}N_{21} + M_{13}N_{31} & M_{11}N_{12} + M_{12}N_{22} + M_{13}N_{32} & M_{11}N_{13} + M_{12}N_{23} + M_{13}N_{33} \\ M_{21}N_{11} + M_{22}N_{21} + M_{23}N_{31} & M_{21}N_{12} + M_{22}N_{22} + M_{23}N_{32} & M_{21}N_{13} + M_{22}N_{23} + M_{23}N_{33} \\ M_{31}N_{11} + M_{32}N_{21} + M_{33}N_{31} & M_{31}N_{12} + M_{32}N_{22} + M_{33}N_{32} & M_{31}N_{13} + M_{32}N_{23} + M_{33}N_{33} \end{bmatrix}$$

Composing two quaterions takes $4^2 = 16$ multiplications, since each quaterion has 4 scalars and 15 additions.

$$\begin{aligned} &(q_0 + q_1 i + q_2 j + q_3 k)(p_0 + p_1 i + p_2 j + p_3 k) \\ &\quad = \left(q_0 p_0 + q_1 p_1 i^2 + q_2 p_2 j^2 + q_3 p_3 k^2\right) + \\ &\quad\quad (q_0 p_1 i + q_1 p_0 i + q_2 p_3 jk + q_3 p_2 kj) + \\ &\quad\quad (q_0 p_2 j + q_2 p_0 j + q_1 p_3 ik + q_3 p_1 ki) + \\ &\quad\quad (q_0 p_3 k + q_3 p_0 k + q_1 p_2 ij + q_2 p_1 ji) \\ &\quad = (q_0 p_0 - q_1 p_1 - q_2 p_2 - q_3 p_3) + \\ &\quad\quad (q_0 p_1 + q_1 p_0 + q_2 p_3 - q_3 p_2)\, i + \\ &\quad\quad (q_0 p_2 + q_2 p_0 - q_1 p_3 + q_3 p_1)\, j + \\ &\quad\quad (q_0 p_3 + q_3 p_0 + q_1 p_2 - q_2 p_1)\, k \end{aligned}$$

Applying a rotation matrix to a vector involves matrix-vector multiplication. A rotation matrix is a 3×3 matrix. So, when multiplying such matrix with a corresponding vector, we do $3^2 = 9$ multiplications and $3 \cdot 2 = 6$ additions, since for every element in the new vector totalling 3 we do 2 additions.

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \cdot \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} V_1 M_{11} + V_2 M_{12} + V_3 M_{13} \\ V_1 M_{21} + V_2 M_{22} + V_3 M_{23} \\ V_1 M_{31} + V_2 M_{32} + V_3 M_{33} \end{bmatrix}$$

2 Modeling

We will elaborate on control theory modelling basics of the quadcopter. We will use this theory to implement a functioning linearized system in SIMULINK.

2.1 Dynamic equation derivation with Euler angles

First, let's define the rotation matrix representing the orientation of the body-fixed frame w.r.t. the inertial frame.

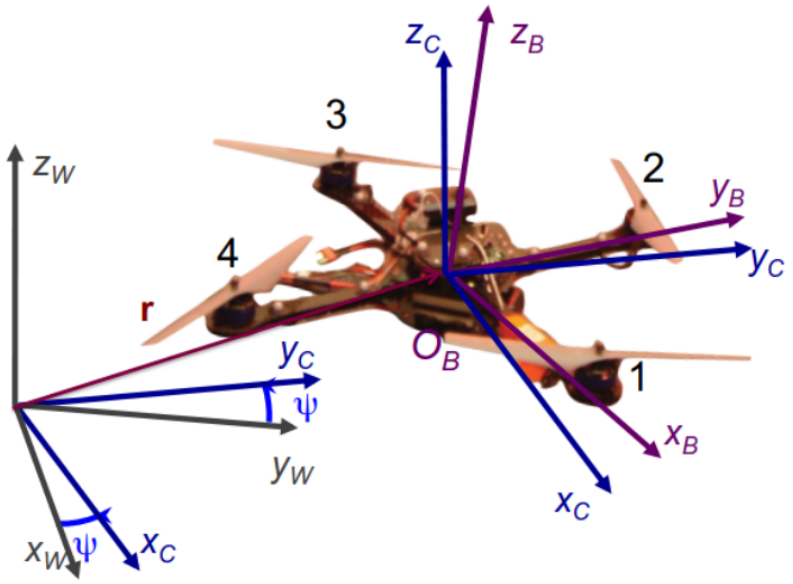


Figure 1: Quadcopter and inertial frame transformation

Let $\{\vec{x}, \vec{y}, \vec{z}\}$ be the three coordinate axis unit vectors without a frame of reference. Let {A} denote a right-hand inertial frame with unit vectors along the axes denoted by $\{\overrightarrow{a_1}, \overrightarrow{a_2}, \overrightarrow{a_3}\}$ expressed in {A}. Let {B} be a (right-hand) body-fixed frame for the airframe with unit vectors $\left\{\overrightarrow{b_1}, \overrightarrow{b_2}, \overrightarrow{b_3}\right\}$, where these vectors are the axes of frame {B} with respect to frame {A}. The orientation of the rigid body is given by a rotation matrix R modelled using RPY (Roll-Pitch-Yaw angles) angles following the order ZYX.

To get from {A} to {B}, we first rotate about a_3 by the angle ψ, and we call this the intermediary frame {E} with a basis $\{\overrightarrow{e_1}, \overrightarrow{e_2}, \overrightarrow{e_3}\}$ where $\overrightarrow{e_i}$ is expressed with respect to frame {A}. This is followed by a rotation about a_1 in the rotated frame through the angle φ, followed by a third rotation about a_2 through the angle θ that results in the body-fixed triad $\left\{\overrightarrow{b_1}, \overrightarrow{b_2}, \overrightarrow{b_3}\right\}$.

$$\mathrm{R}=\begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\theta)\sin(\phi)-\sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi)+\sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\theta) & \sin(\psi)\sin(\theta)\sin(\phi)+\cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi)-\cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

Now we will define the relation between the angular velocity $\dot{\Theta}$ and the rotational velocity of the body-fixed frame w.

$$\dot{\Theta} = R \cdot w$$

The linear and angular dynamic equation of the drone in compact form with each component shown explicitly is:

Propeller total thrust:

$$F_B = k \begin{bmatrix} 0 \\ 0 \\ \Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2 \end{bmatrix} \tag{8}$$

Viscous damping:

$$F_{\mathrm{D}} = \begin{bmatrix} -k_d \cdot x \\ -k_d \cdot y \\ -k_d \cdot z \end{bmatrix} \tag{9}$$

Linear dynamic equation of motion expressed in A frame in Figure 1

$$M\ddot{p} = M \cdot G + R \cdot F_B + F_{\mathrm{D}} \tag{10}$$

hence,

$$\begin{aligned} \ddot{p} &= M^{-1} \cdot (M \cdot G + R \cdot F_B + F_{\mathrm{D}}) = G + M^{-1} \cdot (R \cdot F_B + F_{\mathrm{D}}) \\ \ddot{p} &= \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \\ M &= \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \\ G &= \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} \end{aligned} \tag{11}$$

With plugged values, the equation is:

$$\ddot{p} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} \frac{(\cos(\psi)\sin(\theta)\cos(\phi)+\sin(\psi)\sin(\phi))k\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right)-k_d\cdot x}{m} \\ \frac{(\sin(\psi)\sin(\theta)\cos(\phi)-\cos(\psi)\sin(\phi))k\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right)-k_d\cdot y}{m} \\ \frac{-mg+\cos(\theta)\cos(\phi)k\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right)-k_d\cdot z}{m} \end{bmatrix} \tag{12}$$

Torque on x-axis of the rigid body:

$$\tau_\phi = Lk\left(\Omega_1^2 - \Omega_3^2\right)$$

Torque on y-axis of the rigid body:

$$\tau_\theta = Lk\left(\Omega_2^2 - \Omega_4^2\right)$$

Torque on z-axis of the rigid body:

$$\tau_\psi = b\left(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2\right)$$

Total body torque:

$$\tau_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}$$

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

Rotational dynamic equation of motion expressed in $\{B\}$ frame in Figure 1

$$I \cdot \dot{\omega} + \omega \times (I \cdot \omega) = \tau_B$$

$$\dot{\omega} = \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = \mathrm{I}^{-1} \cdot (\tau_B - \omega \times (\mathrm{I} \cdot \omega)) = \begin{bmatrix} \tau I_{xx}^{-1} \\ \tau_{\theta yy}^{-1} \\ \tau I_{zz}^{-1} \\ \tau_z \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix}$$

With plugged values:

$$\begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = \begin{bmatrix} \frac{\omega_z(I_{yy}-I_{zz})\omega_y + Lk(\Omega_1-\Omega_3)(\Omega_1+\Omega_3)}{I_{xx}} \\ \frac{-\omega_z(I_{xx}-I_{zz})\omega_x + Lk(\Omega_2-\Omega_4)(\Omega_2+\Omega_4)}{I_{yy}} \\ \frac{b\left(\Omega_1^2-\Omega_2^2+\Omega_3^2-\Omega_4^2\right)+\omega_x\omega_y(I_{xx}-I_{yy})}{I_{zz}} \end{bmatrix} \tag{13}$$

And with numbers:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0.02(\cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi))\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right) - 2k_d\dot{x} \\ 0.02(\sin(\psi)\sin(\theta)\cos(\phi) - 1.\cos(\psi)\sin(\phi))\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right) - 2k_d\dot{y} \\ -9.81 + 0.02\cos(\theta)\cos(\phi)\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right) - 2k_d\dot{z} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = \begin{bmatrix} 749.9\Omega_1^2 - 749.9\Omega_3^2 - 2.33\omega_y\omega_z \\ 749.9\Omega_2^2 - 749.9\Omega_4^2 + 2.33\omega_x\omega_z \\ 100.\Omega_1^2 - 100.\Omega_2^2 + 100.\Omega_3^2 - 100.\Omega_4^2 \end{bmatrix} \tag{14}$$

2.1.1 Simulation of a MATLAB/Simulink model of the drone

Three main scenarios were simulated where the speeds of the motors are as follows: [0, 0, 0, 0], [10000, 0, 10000, 0], [0, 10000, 0, 10000]. See figures 2, 3 and 4. The behaviour of the drone and a model can be explained by:

1. Scenario [0, 0, 0, 0]: There is no thrust, so drone will fall down to infinity, because of gravity. There are no torques acting on the drone either so it does not spin.
2. Scenario [10000, 0, 10000, 0]: Thrust is bigger than gravity force and creates upward acceleration (in z-direction) so drone goes upwards unbounded. (The drone itself is in theory stable in x-direction and y-direction) But, because the two motors are spinning in the same direction, and creating the torques of the same direction, the drone spins around the z-axis (in the negative direction).
3. Scenario [0, 10000, 0, 10000]: The same as above, but drone is spinning in the positive directio of z-axis.

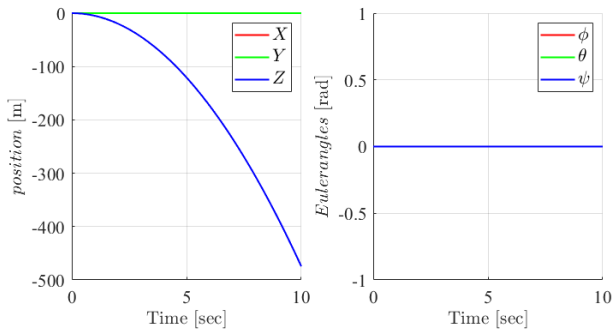


Figure 2: Simulation for the following speed of motors: [0, 0, 0, 0].

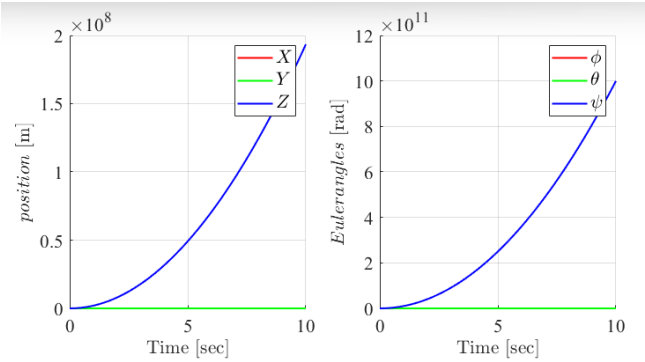


Figure 3: Simulation for the following speed of motors: [10000, 0, 10000, 0].

2.2 Dynamic equation derivation with quaternions

In order to save some space, the equations are sumarized in the form of a figure 5.

2.2.1 Simulation of a MATLAB/Simulink model of the drone

Three main scenarios were simulated where the speeds of the motors are as follows: [0, 0, 0, 0], [10000, 0, 10000, 0], [0, 10000, 0, 10000]. See figures 6, 7 and 8. The behaviour of the drone and a model are exactly the same as when using Euler angles. The only difference is that quaternion when being recalculated back to Euler angles has limits between $-\Pi/2$ and $\Pi/2$. This is because of a trigonometric functions.

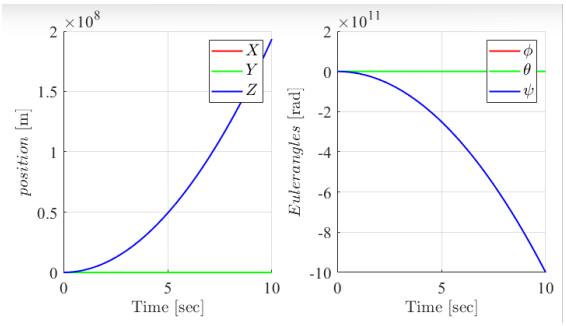


Figure 4: Simulation for the following speed of motors: [0, 10000, 0, 10000].

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} R T_B + \frac{1}{m} F_D$$

$$\dot{x}_3 = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix}^{-1} x_4$$

$$\dot{x}_4 = \begin{bmatrix} \tau_\phi {I_{xx}}^{-1} \\ \tau_\theta {I_{yy}}^{-1} \\ \tau_\psi {I_{zz}}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}} \omega_y \omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}} \omega_x \omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}} \omega_x \omega_y \end{bmatrix}$$

$$R = \begin{bmatrix} q_4^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2+q_3q_4) & 2(q_1q_3-q_2q_4) \\ 2(q_1q_2-q_3q_4) & q_4^2-q_1^2+q_2^2-q_3^2 & 2(q_2q_3+q_1q_4) \\ 2(q_1q_3+q_2q_4) & 2(q_2q_3-q_1q_4) & q_4^2-q_1^2-q_2^2+q_3^2 \end{bmatrix}$$

$$\dot{x}_3 = \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{pmatrix} = \frac{1}{2} \begin{bmatrix} q_1 & q_4 & -q_3 & q_2 \\ q_2 & q_3 & q_4 & -q_1 \\ q_3 & -q_2 & q_1 & q_4 \\ q_4 & -q_1 & -q_2 & -q_3 \end{bmatrix} \begin{pmatrix} 0 \\ \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} x_4$$

Figure 5: Dynamic equation using quaternions.

2.3 Simulation of a linearized MATLAB/Simulink model of the drone

Comparing results between non-linear and linearized system responses one can assess that the general trends are the same. The difference lies in the speed and magnitude of response. For example during the first 10 seconds a drone in non-linear system falls down 500 meters when the motors are shut down, where for the same test the linearized model claims it would be almost twice as much. See figures 9, 10, 11 .This imprecision might be explained by the fact that a linearized system is an approximation of a non-linear one and lacks its complexity.

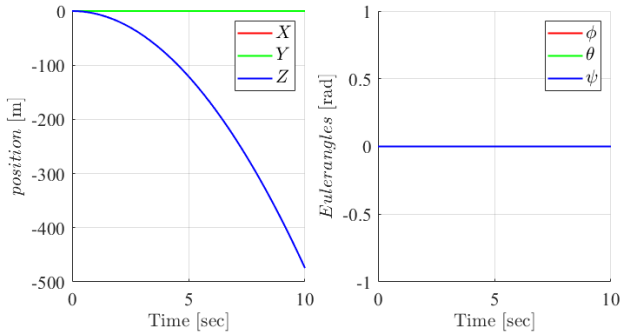


Figure 6: Simulation for the following speed of motors: [0, 0, 0, 0].

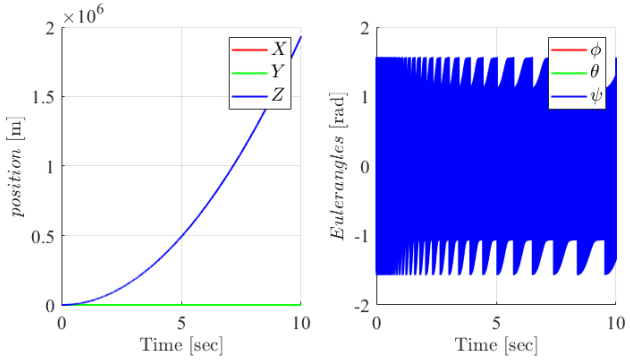


Figure 7: Simulation for the following speed of motors: [10000, 0, 10000, 0].

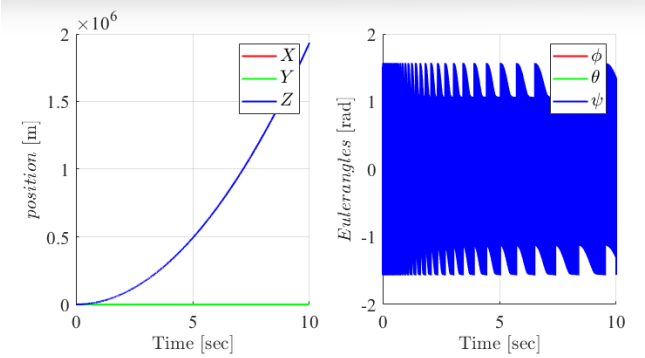


Figure 8: Simulation for the following speed of motors: [0, 10000, 0, 10000].

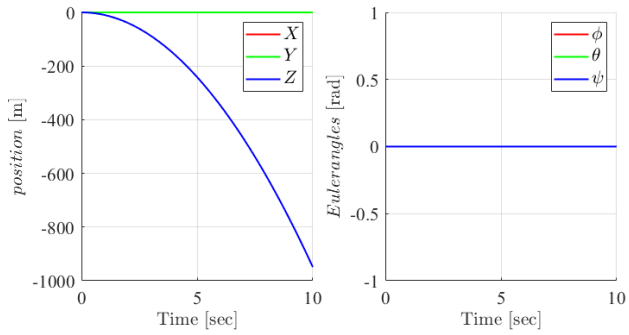


Figure 9: Simulation for the following speed of motors: [0, 0, 0, 0].

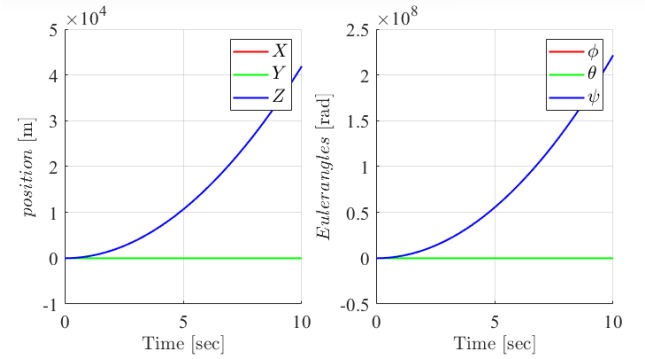


Figure 10: Simulation for the following speed of motors: [10000, 0, 10000, 0].

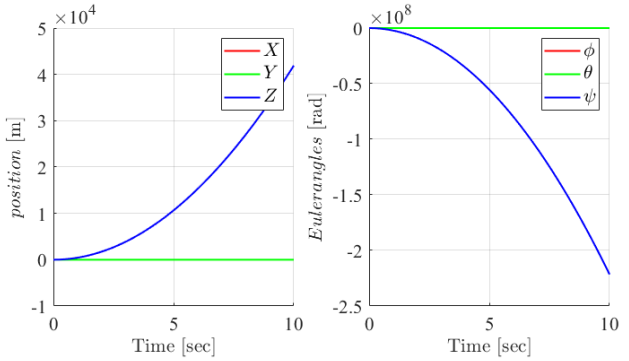


Figure 11: Simulation for the following speed of motors: [0, 10000, 0, 10000].

3 Control

3.1 Attitude controller tested on linearized system

Gains of a PID controller were tuned initially on a linearized system. In the figure 12 it is visible that all of the responses gave no overshoot and a steady state error remains below 1% (dash lines). It means that a closed-loop system is critically damped. It was achieved by using PD controller without an integral action. The gains were tuned minimizing a setteling time which in this case is around 0.5 sec for angular step references and around 1.5 sec for a position reference on z-axis.

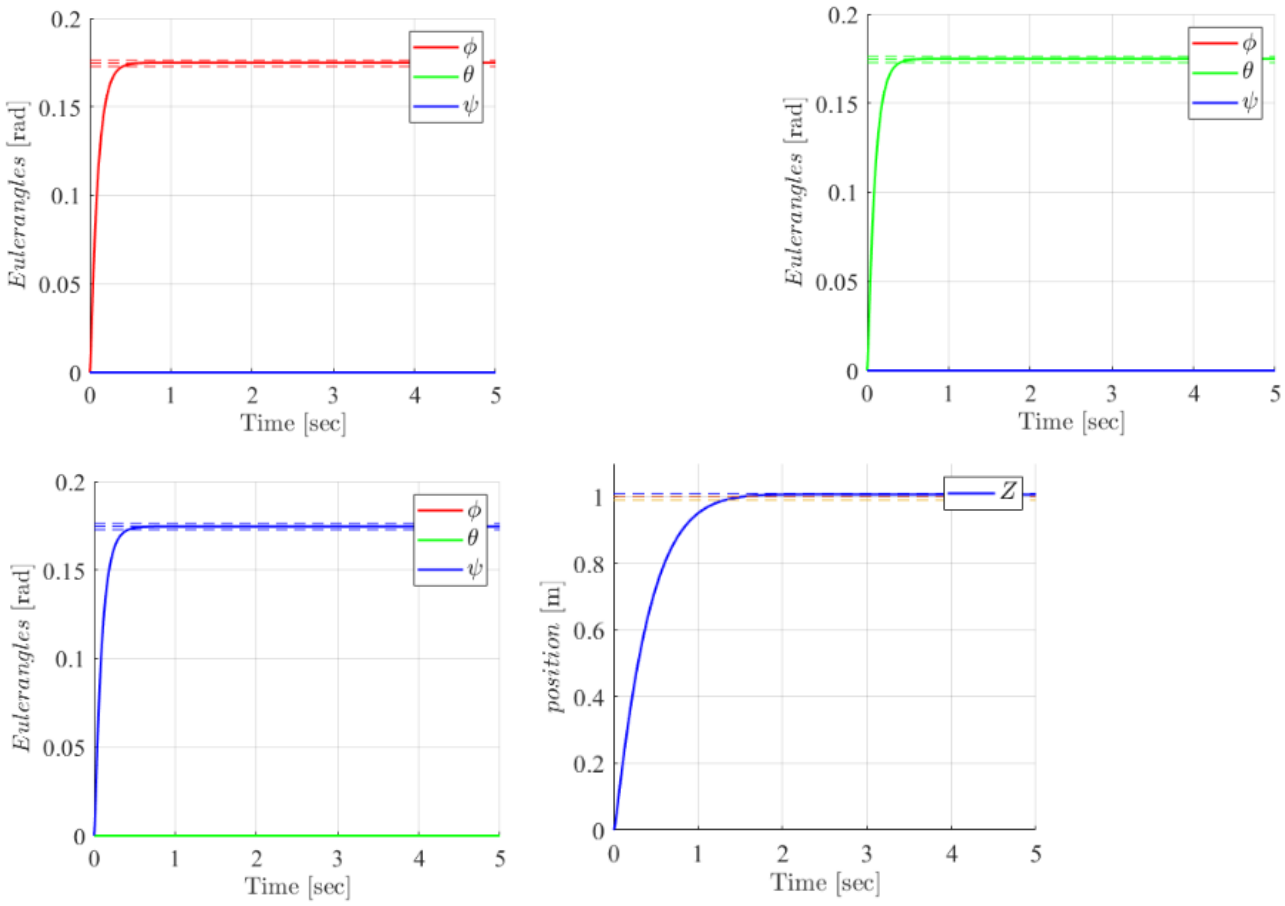


Figure 12: Step responses of a linearized system.

3.2 Attitude controller tested on non-linear system

Testing the same controller on a non-linear system we can see exactly the same behaviour for angular references. On the other hand the step response on a z-axis characterizes a huge overshoot, above 60%. See figure 13. Such a difference between two simulations can be explained by the fact that two models (it is linearized and non-liner one) are in principle different. Thus tuning a controller on linearized system does not guarantee a good performance on non-linear system. In other words, a linearized model is just a simplification of a non-linear one, which again is a poor model of a complex reality.

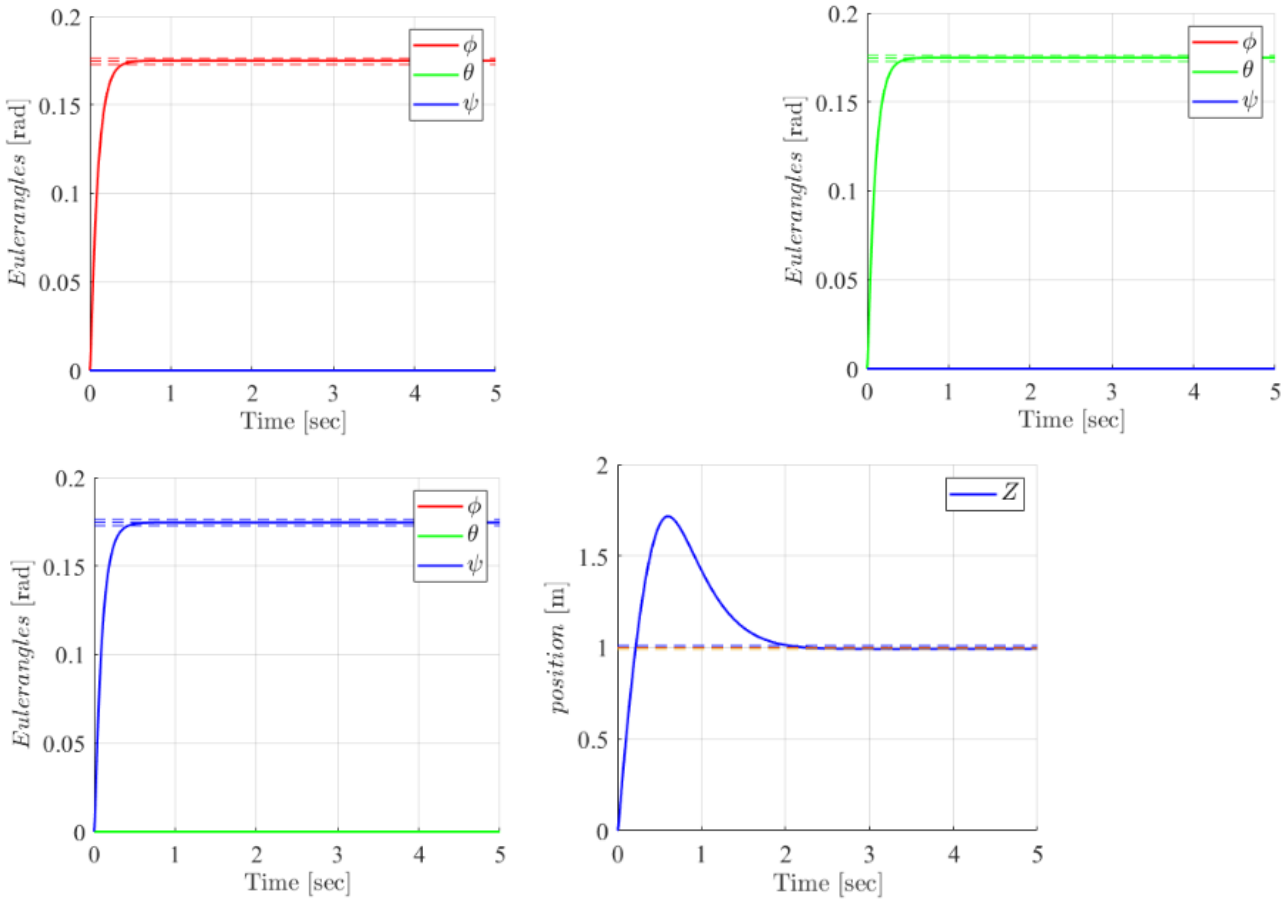


Figure 13: Step responses of a non-linear system.

3.3 Position controller tested on non-linear system

After closing a position control loop for x and y components the PID gains were tuned. Initially Ziegler–Nichols method was to be applied, but system gives an oscillatory response for all values of proportional gain. Thus making this method unusable in this case. Therefore the gains have been tuned empirically. The proportional gain was increased in order to make a response faster, while the D gain was also rising to ensure as little as possible overshoot. It was decided to use a PD controller, without integral action, because the latter caused a steady state error and slow oscillation around the reference. It is worth noting that this set-up might work in simulation, but in real life scenario, usage of an integral action might be necessary. In the figure 14 it can be seen that a position controller is slower than a attitude controller. It has an overshoot of around 7% and settling time of around 30 sec.

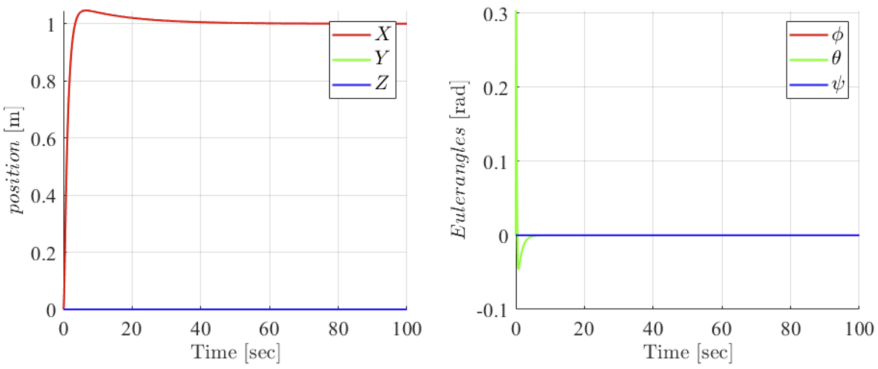


Figure 14: Step responses of a non-linear system. Testing the position controller gains.

4 Path Planning

In this exercise, we will explore methods for the quadrocopter path planning, using algorithms like DFS, BFS, Greedy and A* [4].

4.1 Depth-first search (DFS) algorithm

4.1.1 Stacking the lowest of the numbers

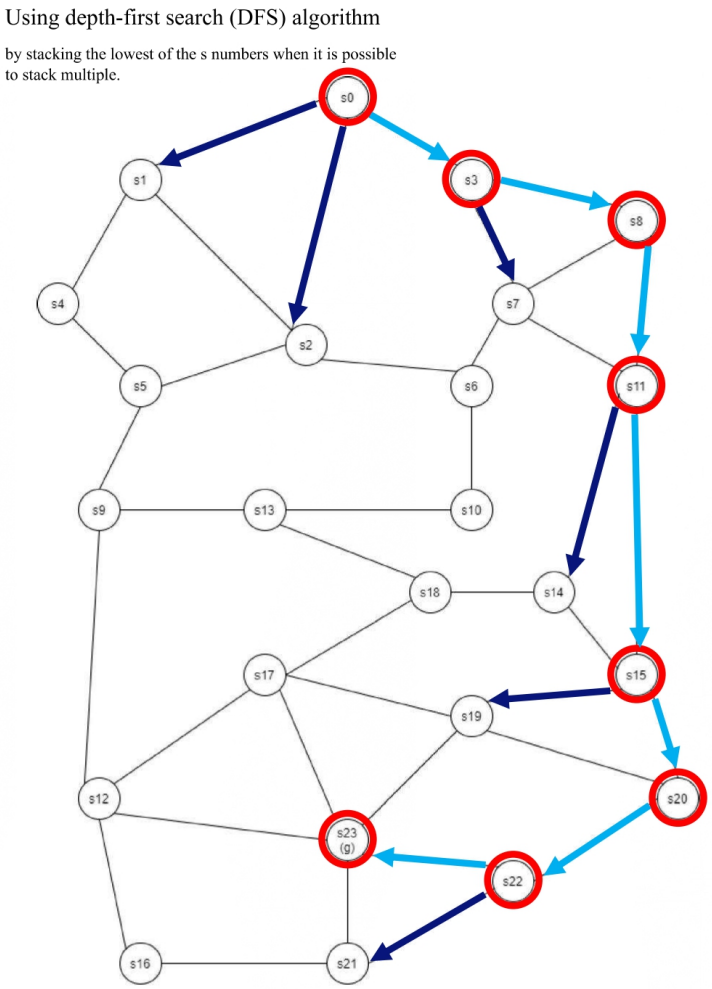


Figure 15: Running depth-first search (DFS) algorithm on a map of cities (s0-23) connected with roads

In the figure above, we show how we ran DFS to find a route from city s0 to s23. Blue edges point from expanded nodes, which are designated by red circles, to generated nodes. Cyan edges represent the path of the found route. Below is the process log after execution. It shows the steps taken by the algorithm, specifying which nodes were expanded as well as the status of the data structure after every expansion:

<s0>	[<s1>, <s2>, <s3>]
<s3>	[<s1>, <s2>, <s7>, <s8>]
<s8>	[<s1>, <s2>, <s7>, <s11>]
<s11>	[<s1>, <s2>, <s7>, <s14>, <s15>]
<s15>	[<s1>, <s2>, <s7>, <s14>, <s19>, <s20>]
<s20>	[<s1>, <s2>, <s7>, <s14>, <s19>, <s22>]
<s22>	[<s1>, <s2>, <s7>, <s14>, <s19>, <s21>, <s23>]
<s23>	[<s1>, <s2>, <s7>, <s14>, <s19>, <s21>]

Table 2: DFS process log after execution on the graph from Figure 15

The same scheme is used in all exercises. In this exercise, when it is possible to stack multiple nodes, we stack the lowest of the s numbers. The nodes are stacked in LIFO (Last In, First Out) order. The following table shows further information about the execution:

Total number of expanded nodes	8
Total number of generated frontiers	13
Length of the found path	7 (assuming each step costs 1)

Table 3: DFS execution results after execution on the graph from Figure 15

4.1.2 Stacking the highest of the numbers

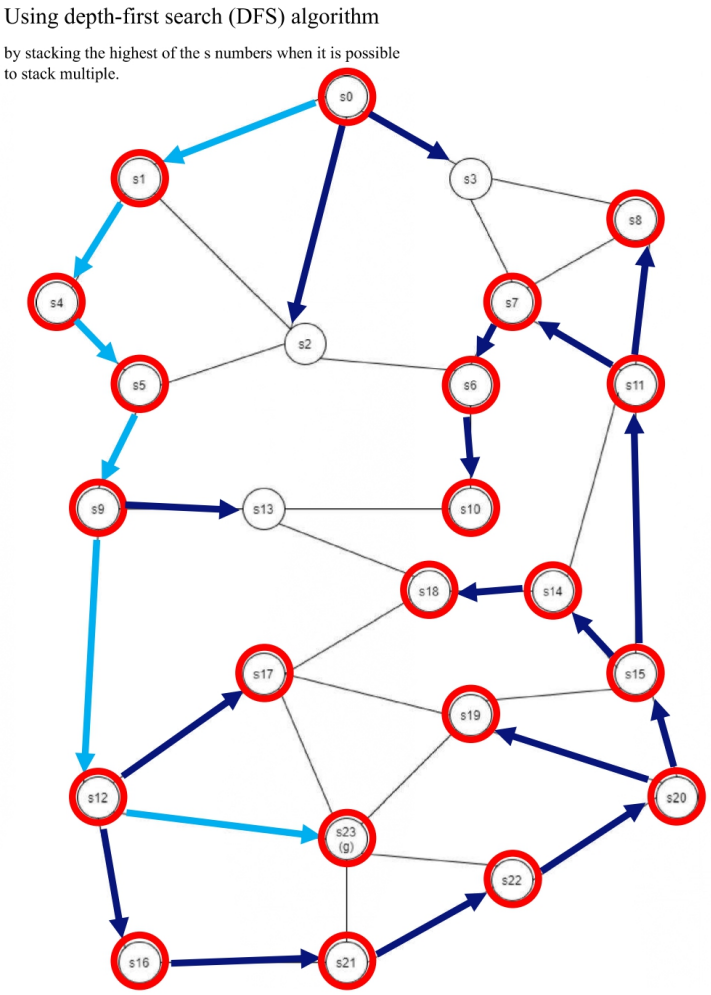


Figure 16: Running depth-first search (DFS) algorithm on a map of cities (s0-23) connected with roads

We do the same procedure here as well.

<s0>	[<s3>, <s2>, <s1>]
<s1>	[<s3>, <s2>, <s4>]
<s4>	[<s3>, <s2>, <s5>]
<s5>	[<s3>, <s2>, <s9>]
<s9>	[<s3>, <s2>, <s13>, <s12>]
<s12>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s16>]
<s16>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s21>]
<s21>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s22>]
<s22>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s20>]
<s20>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s15>]
<s15>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>, <s11>]
<s11>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>, <s8>, <s7>]
<s7>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>, <s8>, <s6>]
<s6>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>, <s8>, <s10>]
<s10>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>, <s8>]
<s8>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s14>]
<s14>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>, <s18>]
<s18>	[<s3>, <s2>, <s13>, <s23>, <s17>, <s19>]
<s19>	[<s3>, <s2>, <s13>, <s23>, <s17>]
<s17>	[<s3>, <s2>, <s13>, <s23>]
<s23>	[<s3>, <s2>, <s13>]

Table 4: DFS process log after execution on the graph from Figure 16

Total number of expanded nodes	21
Total number of generated frontiers	23
Length of the found path	6 (assuming each step costs 1)

Table 5: DFS execution results after execution on the graph from Figure 16

4.1.3 DFS summary

DFS is not optimal. The reason we get 1 route longer than the other is because we chose a different way to stack our nodes. DFS proceeds by expanding the deepest node until no successors are found. Nodes with no successors are dropped from the frontier once expanded, and the algorithm searches the next deepest node that still has unexplored successors. In the first run, we expanded through s3, whereas in the second run, we expanded through s1. Thus, we get 2 different routes.

4.2 Breadth-first search (BFS) algorithm

4.2.1 Queuing the lowest of the numbers

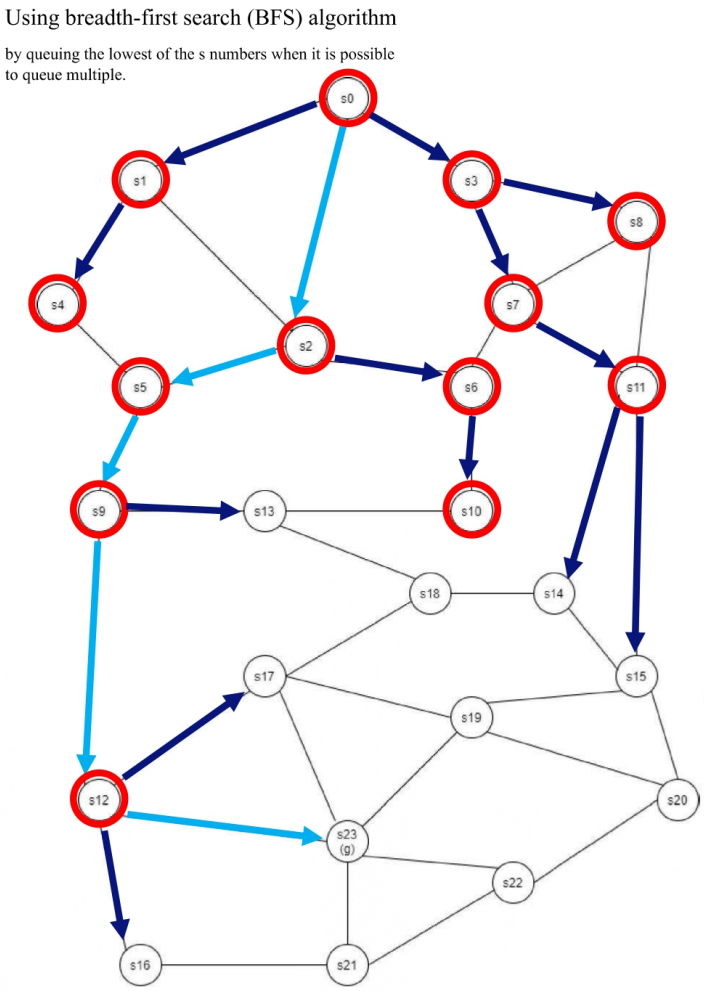


Figure 17: Running breadth-first search (BFS) algorithm on a map of cities (s0-23) connected with roads

<s0>	[<s1>, <s2>, <s3>]
<s1>	[<s2>, <s3>, <s4>]
<s2>	[<s3>, <s4>, <s5>, <s6>]
<s3>	[<s4>, <s5>, <s6>, <s7>, <s8>]
<s4>	[<s5>, <s6>, <s7>, <s8>]
<s5>	[<s6>, <s7>, <s8>, <s9>]
<s6>	[<s7>, <s8>, <s9>, <s10>]
<s7>	[<s8>, <s9>, <s10>, <s11>]
<s8>	[<s9>, <s10>, <s11>]
<s9>	[<s10>, <s11>, <s12>, <s13>]
<s10>	[<s11>, <s12>, <s13>]
<s11>	[<s12>, <s13>, <s14>, <s15>]
<s12>	[<s13>, <s14>, <s15>, <s16>, <s17>, <s23>]

Table 6: BFS process log after execution on the graph from Figure 17

The nodes are stacked in FIFO (First In, First Out) order.

Total number of expanded nodes	13
Total number of generated frontiers	18
Length of the found path	5 (assuming each step costs 1)

Table 7: BFS execution results after execution on the graph from Figure 17

4.2.2 BFS summary

BFS always finds a solution. It is optimal for unit step costs. Unfortunately, it has exponential time complexity. This means that it may not be able to find a solution quickly when working on large finite state spaces. DFS on the other hand may never find a solution. It is not optimal. Yet, it has linear space complexity. This means that it may find a solution even when working on large finite spaces, since it will require less memory. So, when working with robots, DFS is preferred, because the hardware will at least be able to cope with real search problems, and not obstructed by memory demands.

4.3 Dijkstra's algorithm

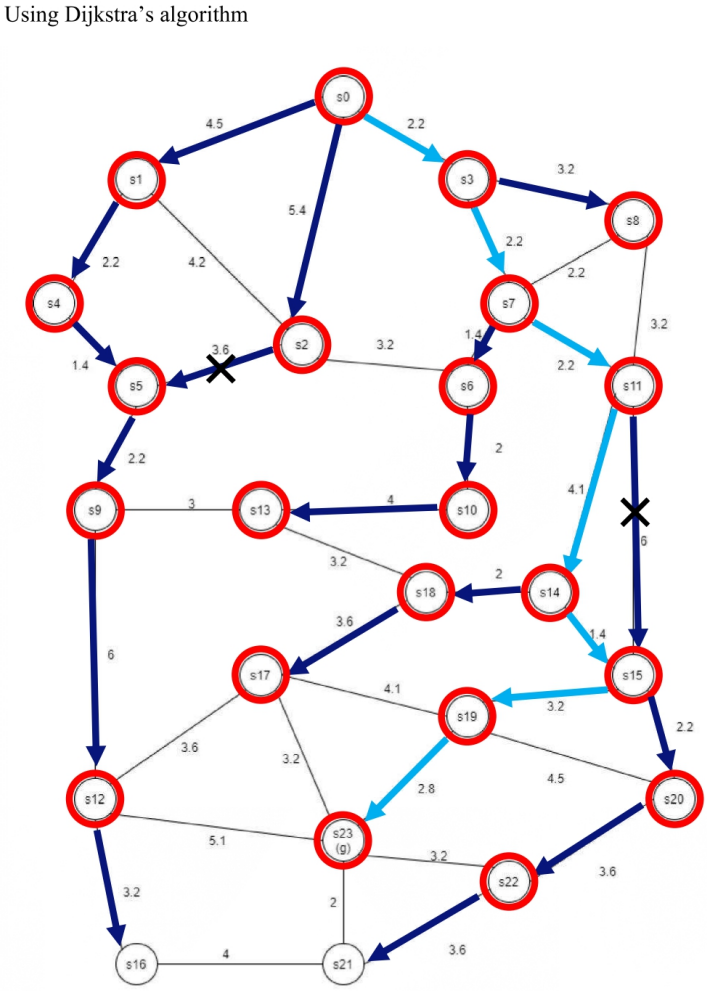


Figure 18: Running Dijkstra's algorithm on a map of cities (s0-23) connected with roads of various costs

With costs added to the graph, we attach to every generated node a corresponding accumulated cost. This is shown in the following table:

<s0>	[(2.2, <s3>), (5.4, <s2>), (4.5, <s1>)]
<s3>	[(4.4, <s7>), (5.4, <s2>), (4.5, <s1>), (5.4, <s8>)]
<s7>	[(4.5, <s1>), (5.4, <s2>), (5.4, <s8>), (5.8, <s6>), (6.6, <s11>)]
<s1>	[(5.4, <s2>), (5.8, <s6>), (5.4, <s8>), (6.6, <s11>), (6.7, <s4>)]
<s2>	[(5.4, <s8>), (5.8, <s6>), (6.7, <s4>), (6.6, <s11>), (9.0, <s5>)]
<s8>	[(5.8, <s6>), (6.6, <s11>), (6.7, <s4>), (9.0, <s5>)]
<s6>	[(6.6, <s11>), (7.8, <s10>), (6.7, <s4>), (9.0, <s5>)]
<s11>	[(6.7, <s4>), (7.8, <s10>), (9.0, <s5>), (10.7, <s14>), (12.6, <s15>)]
<s4>	[(7.8, <s10>), (8.1, <s5>), (12.6, <s15>), (10.7, <s14>)]
<s10>	[(8.1, <s5>), (10.7, <s14>), (12.6, <s15>), (11.8, <s13>)]
<s5>	[(10.3, <s9>), (10.7, <s14>), (12.6, <s15>), (11.8, <s13>)]
<s9>	[(10.7, <s14>), (11.8, <s13>), (12.6, <s15>), (16.3, <s12>)]
<s14>	[(11.8, <s13>), (12.7, <s18>), (12.1, <s15>), (16.3, <s12>)]
<s13>	[(12.1, <s15>), (12.7, <s18>), (16.3, <s12>)]
<s15>	[(12.7, <s18>), (14.3, <s20>), (15.3, <s19>), (16.3, <s12>)]
<s18>	[(14.3, <s20>), (16.3, <s12>), (15.3, <s19>), (16.3, <s17>)]
<s20>	[(15.3, <s19>), (16.3, <s12>), (16.3, <s17>), (17.9, <s22>)]
<s19>	[(16.3, <s12>), (17.9, <s22>), (16.3, <s17>), (18.1, <s23>)]
<s12>	[(16.3, <s17>), (17.9, <s22>), (18.1, <s23>), (19.5, <s16>)]
<s17>	[(17.9, <s22>), (19.5, <s16>), (18.1, <s23>)]
<s22>	[(18.1, <s23>), (19.5, <s16>), (21.5, <s21>)]
<s23>	[(19.5, <s16>), (21.5, <s21>)]

Table 8: Dijkstra's process log after execution on the graph from Figure 18

The nodes with the lowest path cost are expanded first.

Total number of expanded nodes	22
Total number of generated frontiers	25
Length of the found path	18.1

Table 9: Dijkstra's execution results after execution on the graph from Figure 18

4.4 Greedy best-first search (GBFS) algorithm

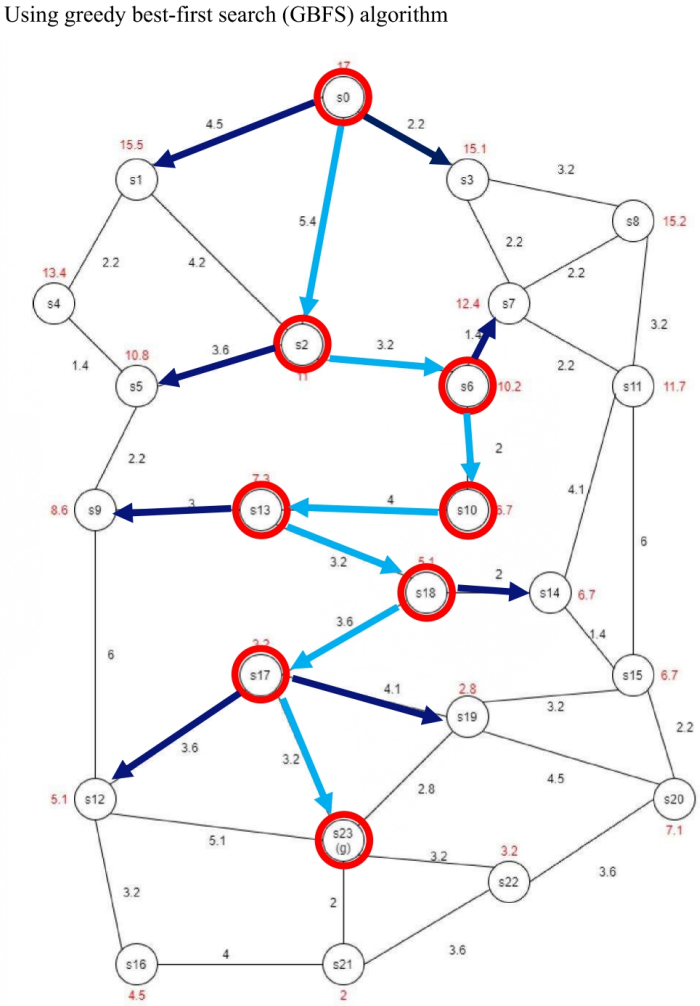


Figure 19: Running Greedy best-first search (GBFS) algorithm on a map of cities (s0-23) connected with roads of various costs

<s0>	[(11, <s2>), (15.5, <s1>), (15.1, <s3>)]
<s2>	[(10.2, <s6>), (10.8, <s5>), (15.1, <s3>), (15.5, <s1>)]
<s6>	[(6.7, <s10>), (10.8, <s5>), (15.1, <s3>), (15.5, <s1>), (12.4, <s7>)]
<s10>	[(7.3, <s13>), (10.8, <s5>), (15.1, <s3>), (15.5, <s1>), (12.4, <s7>)]
<s13>	[(5.1, <s18>), (10.8, <s5>), (8.6, <s9>), (15.5, <s1>), (12.4, <s7>), (15.1, <s3>)]
<s18>	[(3.2, <s17>), (10.8, <s5>), (6.7, <s14>), (15.5, <s1>), (12.4, <s7>), (15.1, <s3>), (8.6, <s9>)]
<s17>	[(0, <s23>), (2.8, <s19>), (6.7, <s14>), (5.1, <s12>), (12.4, <s7>), (15.1, <s3>), (8.6, <s9>), (15.5, <s1>), (10.8, <s5>)]
<s23>	[(2.8, <s19>), (6.7, <s14>), (5.1, <s12>), (12.4, <s7>), (15.1, <s3>), (8.6, <s9>), (15.5, <s1>), (10.8, <s5>)]

Table 10: GBFS process log after execution on the graph from Figure 19

The nodes with the lowest distance to the target are expanded first.

Total number of expanded nodes	8
Total number of generated frontiers	15
Length of the found path	24.6

Table 11: GBFS execution results after execution on the graph from Figure 19

4.5 A* search algorithm

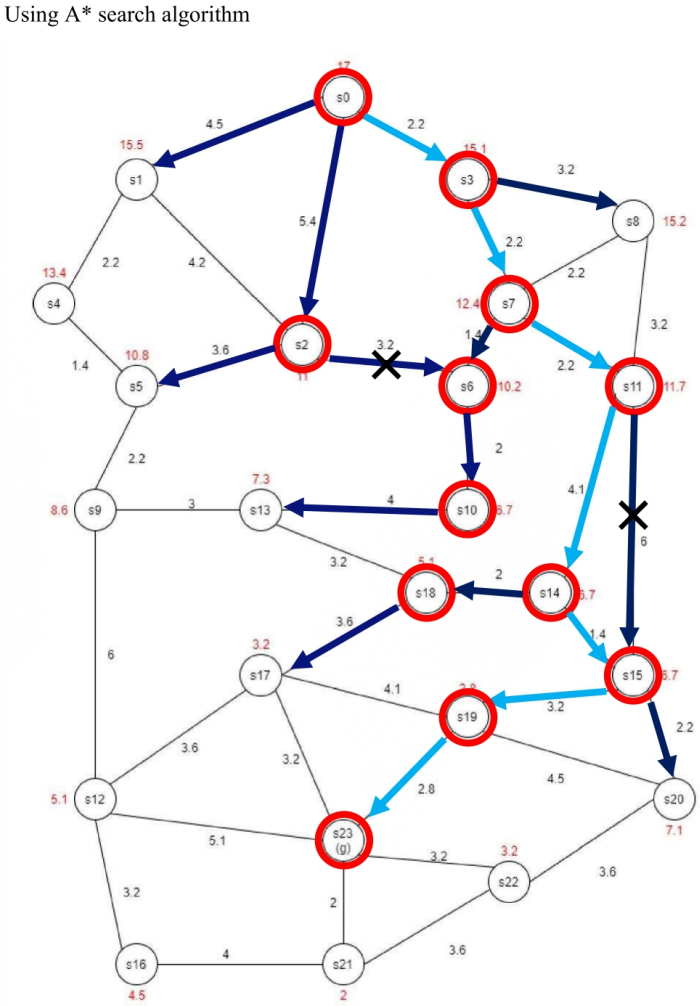


Figure 20: Running A* search algorithm on a map of cities (s0-23), where for each the distance to the goal is specified, connected with roads of various costs

<s0>	[(16.4, <s2>), (20.0, <s1>), (17.3, <s3>)]
<s2>	[(17.3, <s3>), (18.8, <s6>), (19.8, <s5>), (20.0, <s1>)]
<s3>	[(16.8, <s7>), (18.8, <s6>), (19.8, <s5>), (20.0, <s1>), (20.6, <s8>)]
<s7>	[(16.0, <s6>), (18.3, <s11>), (20.6, <s8>), (20.0, <s1>), (19.8, <s5>)]
<s6>	[(14.5, <s10>), (18.3, <s11>), (20.6, <s8>), (20.0, <s1>), (19.8, <s5>)]
<s10>	[(18.3, <s11>), (19.1, <s13>), (20.6, <s8>), (20.0, <s1>), (19.8, <s5>)]
<s11>	[(17.4, <s14>), (19.1, <s13>), (19.3, <s15>), (20.0, <s1>), (19.8, <s5>), (20.6, <s8>)]
<s14>	[(17.8, <s18>), (19.1, <s13>), (18.8, <s15>), (20.6, <s8>), (19.8, <s5>), (20.0, <s1>)]
<s18>	[(18.8, <s15>), (19.1, <s13>), (19.5, <s17>), (20.6, <s8>), (19.8, <s5>), (20.0, <s1>)]
<s15>	[(18.1, <s19>), (19.8, <s5>), (19.1, <s13>), (20.6, <s8>), (20.0, <s1>), (19.5, <s17>), (21.4, <s20>)]
<s19>	[(18.1, <s23>), (19.8, <s5>), (19.1, <s13>), (20.6, <s8>), (20.0, <s1>), (21.4, <s20>), (19.5, <s17>)]
<s23>	[(19.8, <s5>), (19.1, <s13>), (20.6, <s8>), (20.0, <s1>), (21.4, <s20>), (19.5, <s17>)]

Table 12: A* process log after execution on the graph from Figure 20

The nodes with the cheapest solution, which is a sum of the path cost and the distance to the target, are expanded first.

Total number of expanded nodes	12
Total number of generated frontiers	19
Length of the found path	18.1

Table 13: A* execution results after execution on the graph from Figure 20

Indeed, the route and the path length are similar to those found by Dijkstra's. The advantage of using A* however is that it expands less nodes and generates less frontiers. This can be important when working on large finite state spaces.

4.6 Search algorithms comparison

4.6.1 Greedy best-first advantages

One of the advantages of using GBFS is that it evaluates nodes by using only the heuristic function $h(n)$, the cost to get from a node to a certain goal. With a good heuristic, its space complexity can be reduced significantly. Thus, it is often efficient. However, it is not optimal, since the algorithm is greedy and seeks to get as close to the goal as possible. It is also incomplete, so we might unfortunately never find a solution.

4.6.2 A* advantages

Using the same heuristic as in GBFS, A* combines it with $g(n)$, the cost to reach a node from another, $f(n) = g(n) + h(n)$. This allows it to become optimal only when $h(n)$ is admissible — never

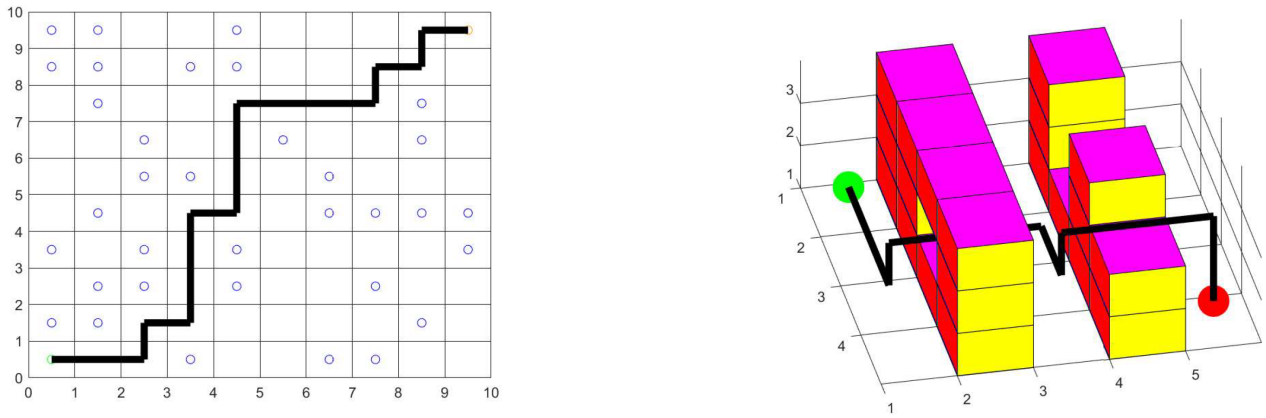
overestimates the cost to reach a certain goal. Unlike GBFS, it is also complete. Unfortunately, its space complexity is exponential in the length of the solution. Thus, it may not be efficient.

4.6.3 Selection for aerial robots

Obviously, we are interested in implementing efficient algorithms, since aerial robots would most likely have hardware limitations as they rely on embedded computers powered by limited energy sources. GBFS would therefore be a safe choice, espcially when such robots operate in large finite or infinite state spaces. Efficiency can be even improved by increasing the quality of the heuristic used. Nonetheless, A* can be great for small finite state spaces. Also, despite the hardware limitations, if there is an option to connect to high-performance computers, at that point all the computations can be conducted remotely, which allows for finding optimal solutions even in complex environments.

4.7 Implementing a 3D Greedy best-first search

We have upgraded the Greedy best-first search from 2D to 3D by expanding the given code with another dimension with a simple for cycle and additional changes and employed the code on generated 2D and 3D maze respectively.



((a)) 2D maze

((b)) 3D maze

Figure 21: Greedy algorithm

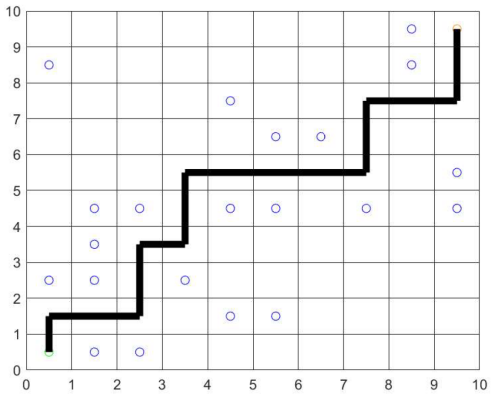
4.8 Implementing a 3D A*

We have upgraded the Greedy best-first to A* algorithm in 2D as well as in 3D. The changes were made in the in the cost function with added Heuristics function, taking into account the approximated cost of the distance to the goal. As the heuristics cannot overestimate the results, we have chosen the Euclidian distance, compared to Manhattan distance as a Heuristic measure.

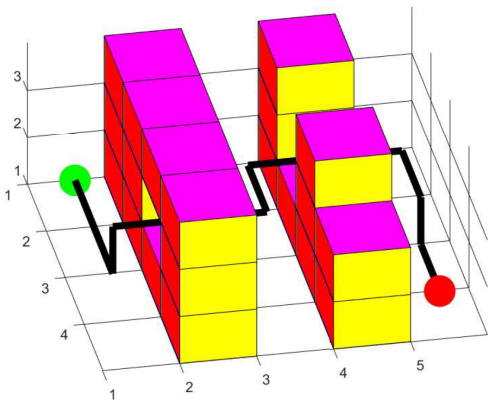
A* algorithm is an upgraded variance of the Dijkstra's algorithm[5] by the heuristic measure:

$$f(n) = h(n) + g(n) \tag{15}$$

having $f(n) = h(n) + g(n)$ or $g(n) = 0$ will therefore result in default Dijkstra's algorithm.



((a)) 2D maze



((b)) 3D maze

Figure 22: A* algorithm

As A* is complete, optimal, and has a time and space complexity of $\mathcal{O}(b^m)$ compared to Greedy BFS with polynomial space complexity. Therefore A* uses more memory than Greedy and can be much slower, although in our simple case, the difference was only in milliseconds.

5 Trajectory Planning

In this exercise, we will look into B-splines and trajectory planning

5.1 Quintic splines

A 1D quintic spline is defined as

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

By first and second derivation of the spline we can obtain the time matrix:

$$T_{time} = \begin{bmatrix} 1 & t_{in} & t_{in}^2 & t_{in}^3 & t_{in}^4 & t_{in}^5 \\ 0 & 1 & 2t_{in} & 3t_{in}^2 & 4t_{in}^3 & 5t_{in}^4 \\ 0 & 0 & 2 & 6t_{in} & 12t_{in}^2 & 20t_{in}^3 \\ 1 & t_A & t_A^2 & t_A^3 & t_A^4 & t_A^5 \\ 0 & 1 & 2t_A & 3t_A^2 & 4t_A^3 & 5t_A^4 \\ 0 & 0 & 2 & 6t_A & 12t_A^2 & 20t_A^3 \end{bmatrix}$$

Where t_{in} and t_{out} are the in and out durations of the segment. As we know that the segment takes 0-1 time units, we can obtain the time matrix as:

to get the velocity and acceleration of this trajectory we take the first and second derivative respectively given the following:

$$T_{time} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 2 & 6 & 12 & 20 \end{bmatrix} \tag{16}$$

$$\dot{x}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4$$
$$\ddot{x}(t) = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3$$

This can also be represented as a matrix and a vector

$$\begin{bmatrix} x(t) \\ \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} = \begin{bmatrix} 1 & t & t^2 & t^3 & t^4 & t^5 \\ 0 & 1 & 2t & 3t^2 & 4t^3 & 5t^4 \\ 0 & 0 & 2 & 6t & 12t^2 & 20t^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}$$

With the initial and final conditions which we will call X:

$$\begin{aligned} x(0) &= 0 \\ \dot{x}(0) &= 0 \\ \ddot{x}(0) &= 0 \\ x(1) &= 1 \\ \dot{x}(1) &= 0 \\ \ddot{x}(1) &= 0 \end{aligned}$$

And with the coeffitients $a_0 - a_5$ (we will call them A) we want to find, we can solve the system of equations by:

$$\begin{aligned} X &= T_{time} \cdot A \\ A &= T_{time}^{-1} \cdot X \end{aligned} \tag{17}$$

and we obtain the coefficients:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 10 \\ -15 \\ 6 \end{bmatrix} \tag{18}$$

5.2 Cox-de Boor smoothing

In order to fluently interpolate the trajectory for smooth quadcopter movement, we will use Cox-de Boor recursion formula [6]

The definition of a spline curve is given by:

$$P(u) = \sum_{k=0}^{n} p_k B_{k,d}(u) \tag{19}$$

where d is the order of the curve and the blending functions $B_{k,d}(u)$ are defined by the recursive Cox-deBoor equations:

$$\begin{aligned} B_{k,1}(u) &= \begin{cases} 1 & \text{if } u_k \leq u \leq u_{k+1} \\ 0 & \text{otherwise} \end{cases} \\ B_{k,\,\mathrm{d}}(u) &= \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u), d > 1 \end{aligned} \tag{20}$$

The generated curve is defined as being the part that is in the range of d blending functions of the form $B_{k,1}(u)$ and u values outside this range are not used. This means that all points in the curve are controlled by d (the order) control points.
Our goal is therefore to compute the two first-order basis functions $B_{0,1}(t)$ and $B_{1,1}(t)$ with $t_i = u_k = i$ and combine them linearly to interpolate between two points $(1, 2)$ and $(2, 3)$

```
%Cox-de Boor
T = [0,0,1,2,3,3,3]';
t = [];

for i = 1:(length(T)-1)
    temp = linspace(T(i),T(i+1), 101);
    t = [t, temp(1:(end-1))];
end

B01 = zeros(1,length(t));
for i = 1:length(B01)
    if t(i) >= 0 && t(i) < 1
        B01(i) = t(i);
    elseif t(i)>=1 && t(i) < 2
        B01(i) = 2-t(i);
    end
end

B11 = zeros(1,length(t));
for i = 1:length(B11)
    if t(i) >= 1 && t(i) < 2
        B11(i) = t(i)-1;
    elseif t(i)>=2 && t(i) < 3
        B11(i) = 3-t(i);
    end
end
```

Figure 23: Matlab computation of Cox-de Boor

We can then parameterize each of the time intervals, linearly combined them and visualize with the help of Script 23.

We can compute the two first-order basis functions from Equations (20) and linearly combine them:

$$\begin{aligned} B_{0,1}(u) &= u \cdot B_{0,0}(u) + (2-u) \cdot B_{1,0}(u) \\ B_{1,1}(u) &= (u-1) \cdot B_{1,0}(u) + (3-u) \cdot B_{2,0}(u) \\ B_{\text{comb}}(u) &= u \cdot B_{0,0}(u) + B_{1,0}(u) + (3-u) \cdot B_{2,0}(u) \end{aligned} \tag{21}$$

And further compute the spline curve by the Equation (19) and Equation (21)

$$P(u) = (2-u)(1,1) + (u-1)(2,2) \quad 1 \leq u \leq 2$$

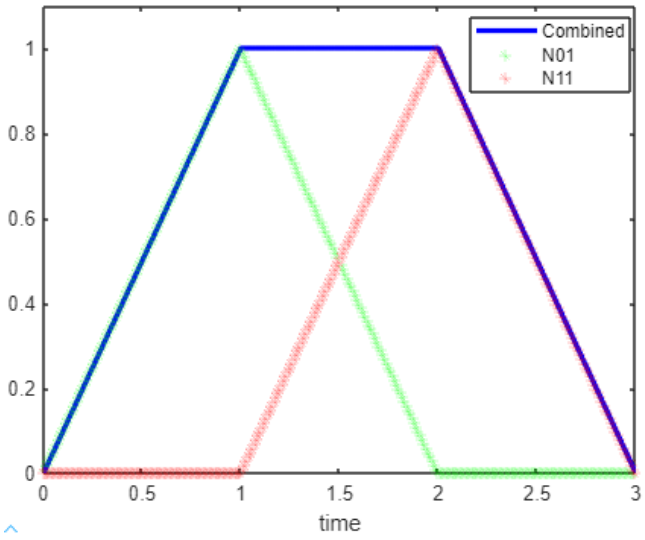


Figure 24: Visualization of the B-Splines and their linear combination

This can be done for all of the intervals.

5.3 Mellinger & Kumar Direction Cosine Matrix

In this subsection, we will have a look at Mellinger & Kumar Direction Cosine Matrix (DCM) [7] for minimum snap trajectory generation.

DCM is found by knowing the upward direction of the drone z_B and the yaw angle ψ :

$$R_B = \begin{bmatrix} x_{B_0} & y_{B_0} & z_{B_0} \\ x_{B_1} & y_{B_1} & z_{B_1} \\ x_{B_2} & y_{B_2} & z_{B_2} \end{bmatrix}, y_B = \frac{z_B \times x_C}{\|z_B \times x_C\|}, x_C = \begin{bmatrix} \cos\psi \\ \sin\psi \\ 0 \end{bmatrix}, x_B = y_B \times z_B \tag{22}$$

However, this is based on the Z-X-Y Euler angles convention, and the convention we are using in this course is Z-Y-X, i.e. $R_B = R_z(\psi)R_y(\theta)R_x(\phi)$ with the roll, pitch and yaw angles ϕ, θ and ψ (respectively), and

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \text{ and } R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{23}$$

Our goal is therefore to compute R_B based on z_B and ψ in the Z-Y-X Euler angles convention.

From the "Direction Cosine Matrix IMU: Theory" [8], we can construct Z-X-Y convention, with similar rules as in Part 1.1:

$$cos_r(\theta) = c_r$$
$$sin_r(\theta) = s_r$$

Where $r \in (x, y, z)$

$$R_z(\theta_z) R_x(\theta_x) R_y(\theta_y) = \begin{bmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{bmatrix} = R_B = \begin{bmatrix} x_{B_0} & y_{B_0} & z_{B_0} \\ x_{B_1} & y_{B_1} & z_{B_1} \\ x_{B_2} & y_{B_2} & z_{B_2} \end{bmatrix} \tag{24}$$

Now, we the knowledge of Equations (5.3) and Equation (24) we can find the similarities by factorizing the elements:

$$\begin{aligned} \phi &= \operatorname{asin}(y_{B_2}) \\ \psi &= \operatorname{atan}2(-y_{B_0}, y_{B_1}) \\ \theta &= \operatorname{atan}2(-x_{B_2}, z_{B_2}) \end{aligned} \tag{25}$$

$$R_B = R_z(\psi) R_y(\theta) R_x(\phi)$$

Which is our desired Z-Y-X convention.

6 Navigation

6.1 Navigating a 2D maze

First, from the maze used in the simulation environment, see Figure 28, we constructed a corresponding 2D map, a matrix consisting of 0s and 1s, where each 0 and 1 represent a free position and a wall, respectively.

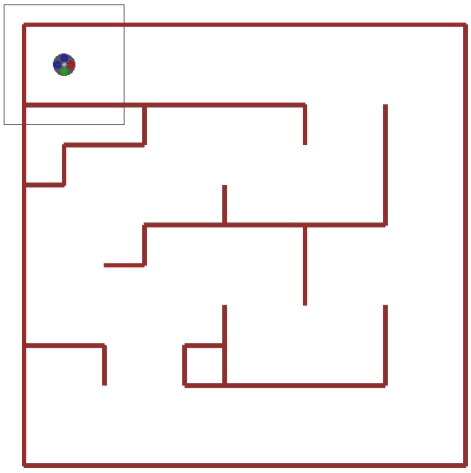


Figure 25: Top view of the maze used in the simulation environment

We then used the provided `greedy_2d` function from exercise 4 to find a route from points (0, 0) to (3, 5). Since there are no required movements in z-axis, which is why we considered the problem in 2D instead of 3D, we simply concatenated a vector of 1s to the resulting route. These 1s represent the path in z-axis. Thus, the route will consist of points in space representing a path from (0, 0, 1) to (3, 5, 1). All this is achieved using the following code:

```
map = [0  0  0  0  0  0  0  0  0  0;
       1  1  1  1  1  1  1  0  1  0;
       1  1  1  0  0  0  1  0  1  0;
       1  0  0  0  1  0  0  0  1  0;
       0  0  1  1  1  1  1  1  1  0;
       0  1  1  0  0  0  1  0  0  0;
       0  0  0  0  1  0  1  0  1  0;
       1  1  0  1  1  0  0  0  1  0;
       0  1  0  1  1  1  1  1  1  0;
       0  0  0  0  0  0  0  0  0  0];
start = [0, 0];
end_ = [3, 5];
route = greedy_2d(map, start+1, end_+1);
route = route(:, [2,1])
route = cat(2, route-1, ones(1,length(route))')
```

An X-Y plot of the achieved trajectory is given in the figure provided below:

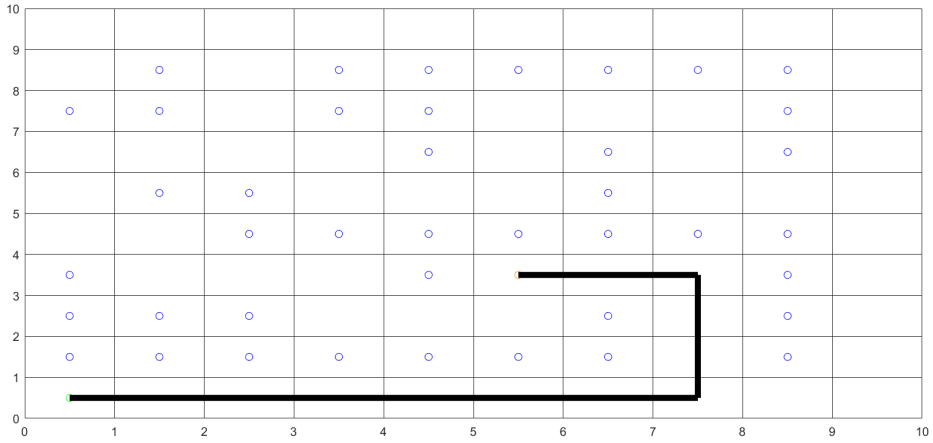


Figure 26: X-Y plot of the achieved trajectory

6.2 Re-implementing the position controller

6.3 Re-implementing the attitude controller

6.4 Aggressively navigating a 2D maze

In this exercise, we used the default position controller. After connecting the blocks from the purple area to the quadcopter, and commenting the unused blocks, we generated a trajectory that allows the drone to fly from (0,0,1) to (9,9,1) in less than 5s without touching any of the walls by introducing 2 corridor constraints. The trajectory passes through these corridors. We also contracted them to ensure that the trajectory would not be generated close to the walls along the way. An X-Y plot of the generated trajectory and plots of the time profile are given in the figures:

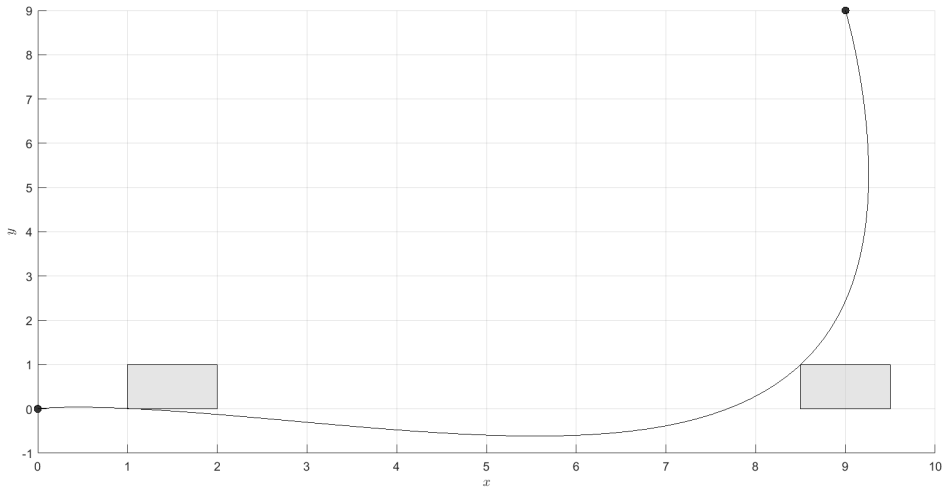


Figure 27: X-Y plot of the generated trajectory

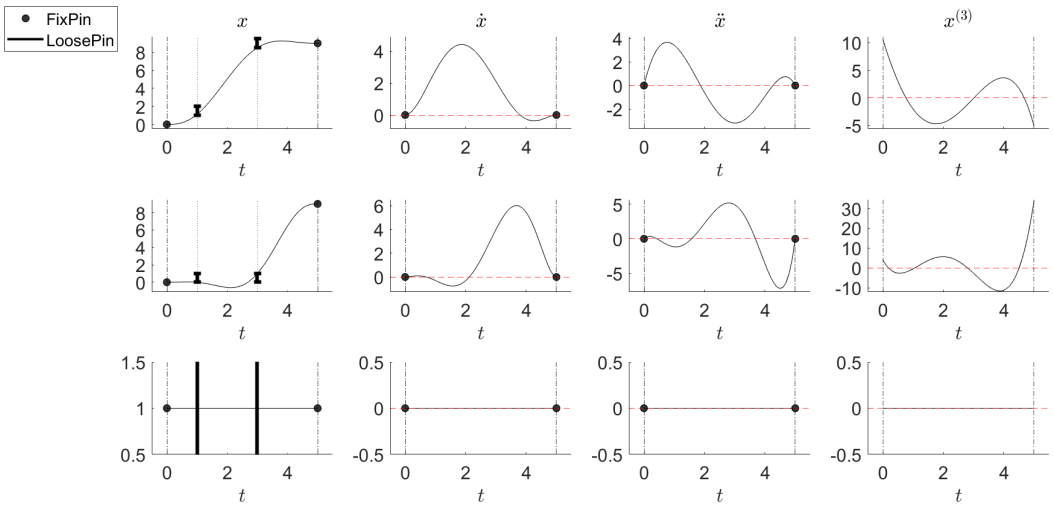


Figure 28: Trajectory's time profile

Indeed, according to such trajectory, our drone is able to reach the target in less than 5s.

All this was achieved by only adjusting corridors' coordinates in `uas trajectory.m`:

```
order = 7;
corridors.times = [1  3];
corridors.x_lower = [1  8.5];
corridors.x_upper = [2  9.5];
corridors.y_lower = [0  0];
corridors.y_upper = [1  1];
corridors.z_lower = [0  0];
corridors.z_upper = [2  2];
```

The rest remained untouched, where the waypoints were already specified as well as the time of reaching them. Open `6.4_flying.mp4` provided together with this report to see how the drone is able to fly quickly along the generated trajectory.

7 Deployment and Demonstration

This part will summarize the Project by utilizing the knowledge, gained in previous exercises, and applying it to real problem solving scenarios.

7.1 Flying through a 3D maze

We were given a text file symbolizing a 3D map of a maze, as well as the build maze in real life, with 10 minute time limit for our drone to fly from the start and land in the goal position.

We have therefore quickly created a 3D structure and maze with the given framework Script [1] and created the optimal path described in Part 4. We chose the A* algorithm 4.5, as it shall give the optimal path, therefore, there will be less possibilities for the drone to deviate from the trajectory, lose signal, or be too slow for the 10 minute time margin. As this task required navigation only by radio, we had to quickly re-calibrate coordinate margins, as well as the inertial´s frame off-sets on the go:

```
%% Scale the route
x_scale = 2.3/max_x;
y_scale = 2.6/max_y;
z_scale = 1.6/max_z;

x_offset = 0.35;
y_offset = 0.5;
z_offset = 0.45;

% Make a copy of the route
route_scaled = route;

% Scale the copy
route_scaled(:,1) = (route_scaled(:,1) - 1) * x_scale + x_offset;
route_scaled(:,2) = (route_scaled(:,2) - 1) * y_scale + y_offset;
route_scaled(:,3) = (route_scaled(:,3) - 1) * z_scale + z_offset;

route = route_scaled
```

Lastly, we have implemented the landing procedure.

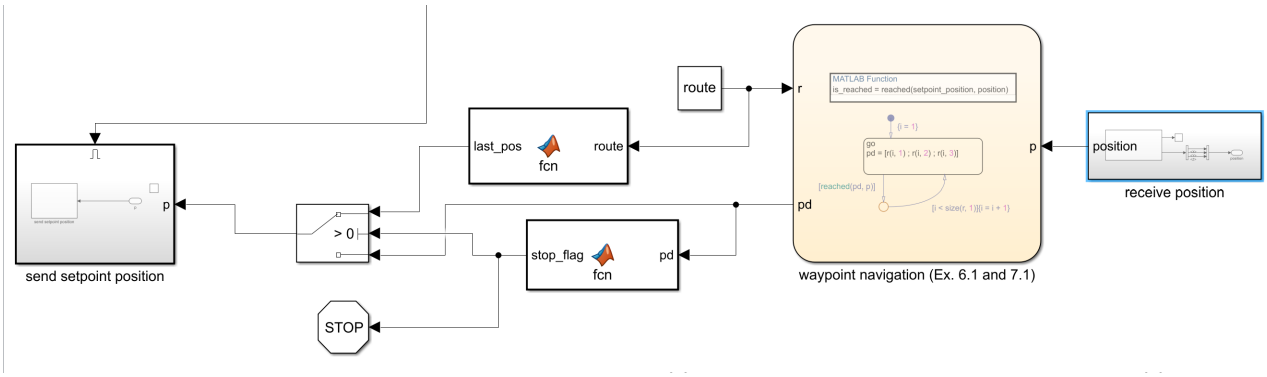


Figure 29: Landing procedure diagram

When the drone reaches the goal position, it shall decrease its altitude to a few centimeters above the ground. Then, a command shall be given to stop the motors.

For a robust landing solution, we have implemented Simulink blocks and Matlab functions shown in Figure 29 with the Switch, Function 1 (upper) and Function 2 (lower):

Function 1:

```
function last_pos = fcn(route)
last_pos = route(end-1,:)';
```

Function 2:

```
function stop_flag = fcn(pd)
stop_flag = 0;
if pd == [0;0;0]
    stop_flag = 1;
end
```

The landing procedure is as follows:

- First, we add a point $Landing_point$, which is in the Goal coordinates, but only a few centimeters above the ground and scale it accordingly.
- Then, we add point *Flag* [0; 0; 0] to the already scaled route. In this scenario, the drone shall never try to reach this point and, therefore, these coordinates are symbolizing a flag command to land.
- Function 2 reads the landing flag, which will result in the consequent switch to flip its input to Function 1, which will repeat an instruction to reach the $Landing_point$ position. This will ensure the drone will maintain stable hovering position few centimeters above the ground and still correct for errors.
- Simultaneously, the "STOP" instruction is given, which will stop the simulation, ground the drone, deactivate its motors and log the flight data.

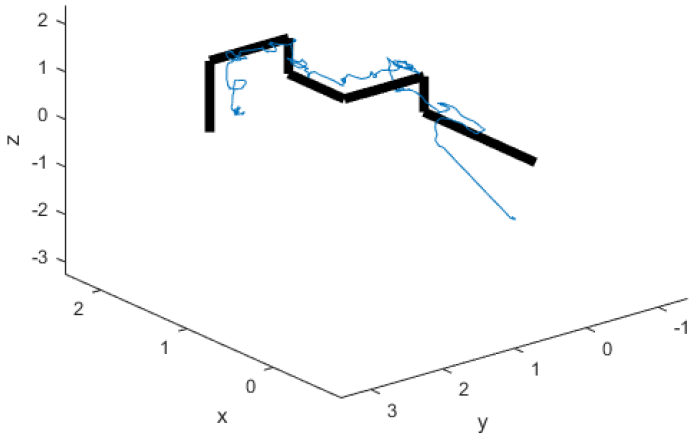


Figure 30: Desired trajectory vs. actual

With this procedure, the drone successfully completed the real 3D maze and we have succeeded in 6 minutes. We can conclude that the drone's behaviour was stable and with accordance to design. The trajectory fluctuation, seen in Figure 30 is due to the positional tracking system's uncalibration and interference, resulting in a worse trajectory log result.

7.2 Porting the position controller to the real system

7.2.1 Hardware set-up

Second and third parts of an experimental work were done using the Optitrack 3D motion tracking system. This positioning set-up uses the reflective markers for tracking. It identifies the configuration of at least 3 of them as a rigid body (object). That being said, in order to get a consistent and stable data from the system a quad-copter needed to characterize itself with a unique pattern of reflective trackers. To assure that the special bracket was designed, which can be seen in Figure 31. Parts were 3D printed and mechanically assembled. The mass of a construction is 15g, which gives 42g of mass in total for a drone with an additional payload.

7.2.2 Identification of thrust to PWM mapping

The position controller that we used outputs thrust, roll and pitch angles. This configuration was compatible with the attitude controller in the simulation. On the other hand the set-up used in experimental part makes use of a attitude controller on board the drone. The latter takes as inputs linear velocity (in inertial frame) and yaw rate commands instead. The linear velocity on z axis is expressed by a PWM command ranging from 0 to 60000. Thus a relationship between the thrust commands and PWM commands needed to be identified.

The course instruction suggests to use a joystick or a gamepad for manual control and in this way gather the data. During experimenting with this set-up we found it to be not good enough. It means

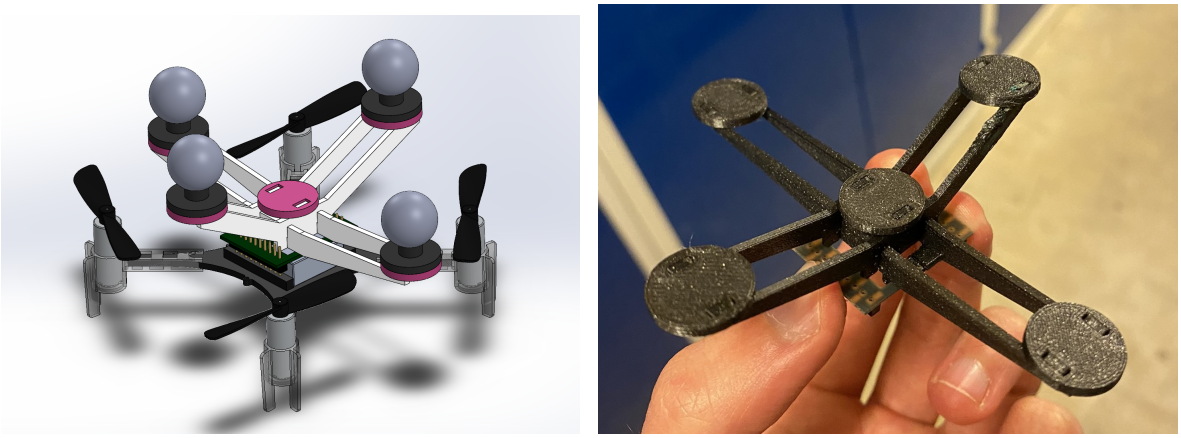


Figure 31: Bracket design for a reflective pattern recognition.

that gathered data was influenced by a lot of noise and uncertainty. On top of that measured acceleration was influenced by drag forces and other variables not modeled in the system (f.e. air flow and variable density in different parts of ASTA). To minimize these negative effects, the commands were sent as a step to attitude PWM input.

The experiment was done as follows. Firstly, a drone starts from the ground with zero initial velocity. The command is being set to a fixed PWM value. Secondly, the drone accelerates only in inertial z-axis direction (the same direction where gravity influences the measurements). Next, the data of acceleration are gathered from onboard IMU unit. Lastly, the drone returns to starting position and the experiment begins again with a different PWM value.

Analyzing data for each round we could see an initial spike in measured acceleration which was decreasing afterwards, as the drone had greater velocity. It can be explained by drag forces which are proportional to the velocity. This is why only the maximum value of measured acceleration was taken into account as a data point for curve fitting. Thrust can be then calculated knowing acceleration and total mass of a drone (In this experiment the Optitrack tracking system was not used at all, so the bracket was not mounted on the drone. Thus the mass was equal to 27 grams). First order polynomial which transforms thrust [N] to PWM was fitted. See figure 32.

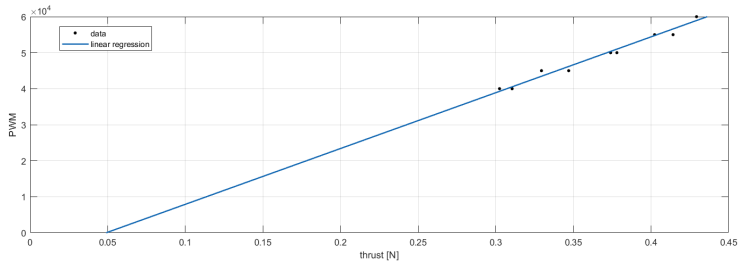


Figure 32: First order polynomial fitting between thrust [N] and PWM signal

Knowing the mass of a drone, the needed thrust to keep it hover can be easily calculated. We used the polynomial to calculate corresponding PWM signal and sent it directly to the drone to test if the

drone can keep its height. Due to measurement errors, as an acceleration, the fitting is normally not perfect and thus needs to be tuned. The final polynomial was calculated to be $f(x) = 149000x - 7600$.

Lastly we removed the mapping block (recalculating thrust in inertial frame to the force along z axis in body-fixed frame) from the default simulink configuration. This is because the velocity commands accepted by onboard attitude controller is expressed in inertial frame.

7.2.3 Design and tuning of a position controller

In principle a PID controller was used as a position controller[7]. P and I gains were implemented directly on the position error, while a D term was implemented on the velocity error. The latter was done because having a D gain separate from the positional error gives us an additional "degree of control". It means that we can specify not only a desired position, but also a reference velocity at the point.

PID gains parameters were initially determined through a simulation (done in previous exercises) and then tuned on the real drone. The simulation showed that PD controller might be sufficient to manage a drone. The practical experiment showed that we need to tune Ki parameter (integral gain) as well. It is due to model uncertainties and disturbances. To eliminate a steady state error we added integral action in the position controller, which can be seen in figure 33.

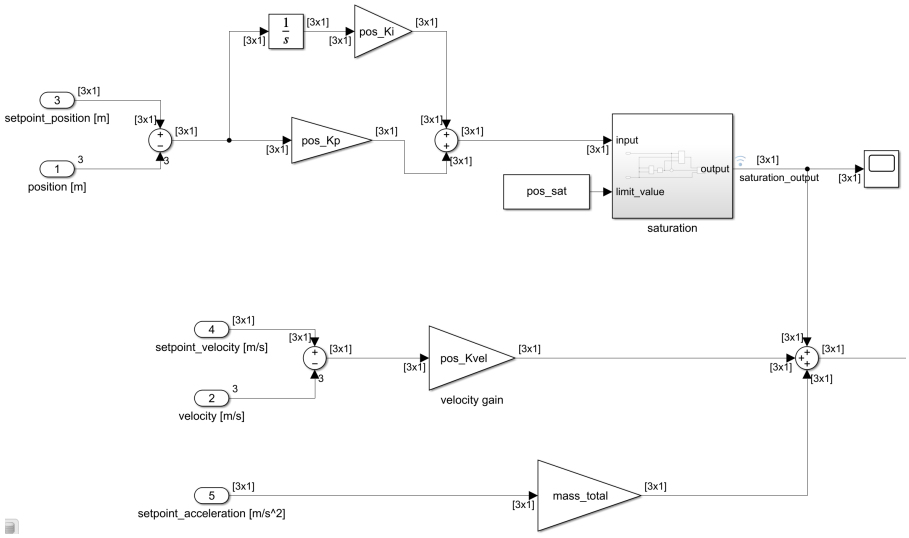


Figure 33: PID controller

The tuning procedure starts from eliminating the overshoot by increasing Kd. Then we add a very small Ki to eliminate steady-state error. However, I term will increase the overshoot again, thus we decrease Kp to reduce it. In the end we get a decent PID controller with almost no overshoot.

7.2.4 Hovering and keeping desired position

We combined the two experiments, described in the exercise 7.2 together. That is, from 2s to 14s the drone tries to hover stably at point (1, 1, 1) and from 14s to 20s the drone moves from setpoint (1, 1,

1) to (2, 1, 1). The results are shown in figures 34, 35 and 36. It can be seen that a drone reaches the desired position in few seconds and the steady states error remains below 10%. The video from the experiment can be fuound under the link: `https://youtube.com/shorts/mcR3SWO7yR8?feature=share`.

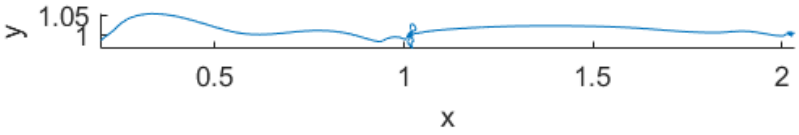


Figure 34: x-y position

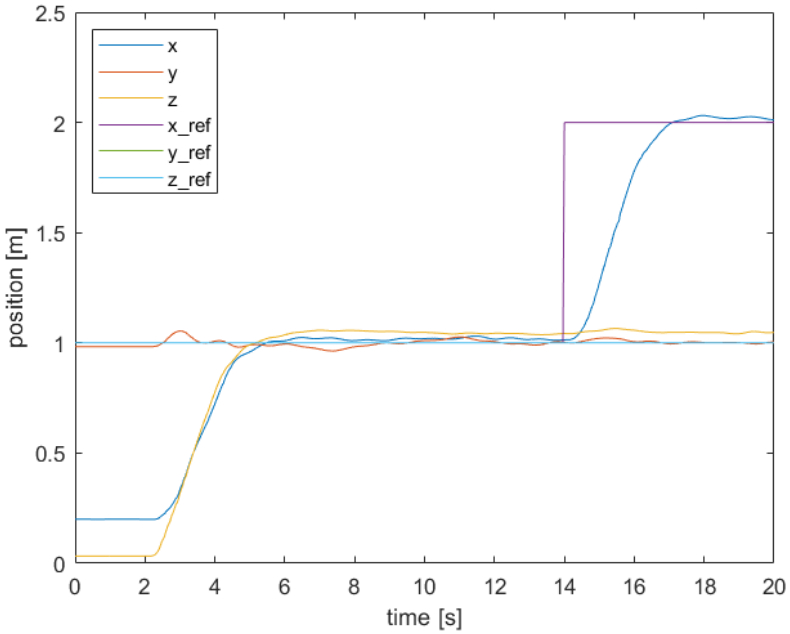


Figure 35: x-y-z position versus time (with references)

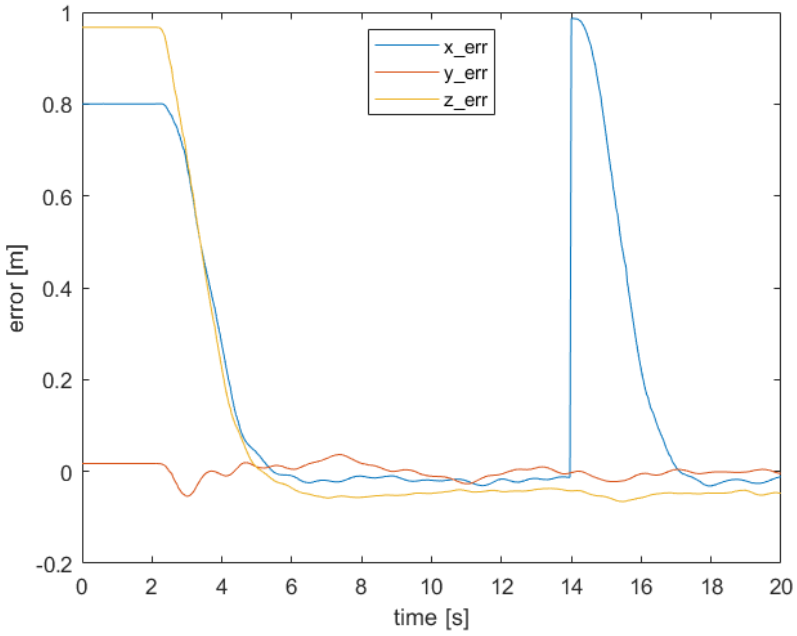


Figure 36: x-y-z error versus time (with references)

7.3 Aggressively navigating through hoops

Having a tuned controller from exercise 7.2 and the thrust to PWM mapping, the exercise 7.3 was more or less about creating a desired trajectory and testing it live. It must be said that at this point the controller was really stable which can be seen in a video where a drone goes flawlessly though the hoops: `https://youtube.com/shorts/-Fk5KHK6B-A`.

7.3.1 Trajectory generation

For trajectory generation the following files have been modified: uas_trajectory.m and uas_minimum_snap.m. Firstly the position and orientation of hoops was read from the Optitrack system. The strategy for trajectory generation was to use only 6 points (start, hoop1, hoop2, hoop3, hoop4, finish) and fit 7th order polynomial to them. The hoop's approach angle was reached though velocity constraints specified in uas_minimum_snap.m file (We didn't use the additional constraints such as corridors). Trajectory generation used by us worked based on a simple physics fact that velocity is always tangential to the path. Thus setting velocity at each hoop being perpendicular to its surface made a drone go perfectly through it. The only two parameters that needed tuning were: speed at the hoops' center and the time at which each hoop was reached. The first one was set to be 0.5 m/s for all 4 hoops and zero for start and finish. The timing, on the other hand, was set for 7 seconds between each waypoint. The result of a trajectory generating script can be seen in figures 37 and 38.

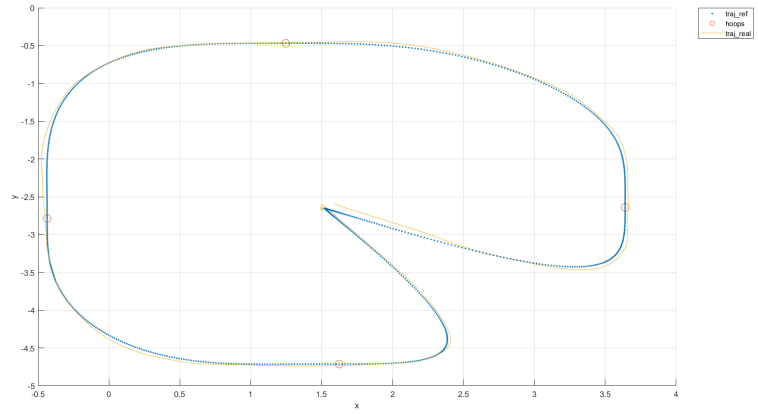


Figure 37: x-y position of the reference trajectory (in blue) and the real trajectory (in orange)

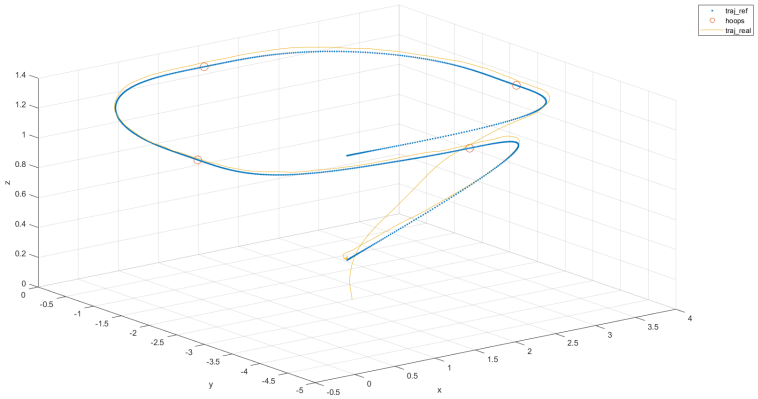


Figure 38: x-y-z position of the reference trajectory (in blue) and the real trajectory (in orange)

Judging by the plots of the reference trajectory superimposed with the achieved one it can be concluded that the drone followed path almost perfectly.

References

[1] D. Wuthier, "Dtu 31039 matlab/ros repository." https://github.com/DavidWuthier/uas-31390, 2022.

[2] "Optical tracking cameras." https://optitrack.com/. Accessed: 2022-06-06.

[3] M. Ben-Ari, "A tutorial on euler angles and quaternions." http://www.weizmann.ac.il/sci-tea/benari/, 2014.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. 01 2001.

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

[6] C. V. Beccari and G. Casciola, "A cox-de boor-type recurrence relation for c1 multi-degree splines," *Computer Aided Geometric Design*, vol. 75, p. 101784, 2019.

[7] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, pp. 2520–2525, 2011.

[8] W. Premerlani and P. Bizard, "Direction cosine matrix imu: Theory," *DIY DRONE: USA*, 01 2009.