Technical University of Denmark

Unmanned Autonomous Systems

Report

Written by

Ask Espensønn Øren s217109
Ole Batting s173894
Lukas Malek s212074
Ernestas Simutis s212571

Contents

Here you will find a table of contribution for the various parts:

Task	**Ask**	**Ernestas**	**Ole**	**Lukas**
1	30%	20%	50%	0%
2	30%	30%	35%	5%
3	60%	0%	40%	0%
4	0%	10%	0%	90%
5	0%	100%	0%	0%
6	40%	0%	30%	30%
7	26.7%	26.7%	26.7%	20%
Writing	30%	20%	35%	15%

1 Part I: Rotations

1.1 Exercise 1.1

The rotation matrix ZXZ can be defined by multiplying the following three matrices:

$$R_I^{v_1} = \begin{bmatrix} cos(-\psi) & sin(-\psi) & 0 \\ -sin(-\psi) & cos(-\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1a}$$

$$R_{v_1}^{v_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(-\theta) & sin(-\theta) \\ 0 & -sin(-\theta) & cos(-\theta) \end{bmatrix} \tag{1b}$$

$$R_{v_2}^{B} = \begin{bmatrix} cos(-\phi) & sin(-\phi) & 0 \\ -sin(-\phi) & cos(-\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1c}$$

In the following order:

$$R_B^I = R_I^{v_1} R_{v_1}^{v_2} R_{v_2}^{B} \tag{2}$$

While recalling that $cos(-x) = cos(x)$ and $sin(-x) = -sin(x)$, such that it yields:

$$R_B^I = \begin{bmatrix} cos(\psi)cos(\phi) - sin(\phi)cos(\theta)sin(\psi) & -cos(\psi)sin(\phi) - sin(\psi)cos(\theta)cos(\phi) & sin(\psi)sin(\theta) \\ sin(\psi)cos(\phi) + cos(\psi)cos(\theta)sin(\phi) & -sin(\psi)sin(\phi) + cos(\psi)cos(\theta)cos(\phi) & -cos(\psi)sin(\theta) \\ sin(\theta)sin(\phi) & sin(\theta)cos(\phi) & cos(\theta) \end{bmatrix} \tag{3}$$

This matrix thus brings the body frame to the inertial one.

1.2 Exercise 1.2

Using trigonometric identities for the two cases:

$$\begin{aligned} R_B^I\Big|_{\sin\theta=0,\ \cos\theta=1} &= \begin{bmatrix} cos(\psi)cos(\phi) - sin(\phi)sin(\psi) & -cos(\psi)sin(\phi) - sin(\psi)cos(\phi) & 0 \\ sin(\psi)cos(\phi) + cos(\psi)sin(\phi) & -sin(\psi)sin(\phi) + cos(\psi)cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} cos(\psi+\phi) & -sin(\psi+\phi) & 0 \\ sin(\psi+\phi) & cos(\psi+\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4}$$

When we get the inverse solution in axis-angle representation

$$\theta = \cos^{-1}(\cos(\psi+\phi) + \cos(\psi+\phi) + 1 - 1) = \psi + \phi$$

$$r = \frac{1}{2\sin(\psi+\phi)} \begin{bmatrix} 0 \\ 0 \\ 2\sin(\psi+\phi) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

we have a singularity if $\psi + \phi = 0$.

And the other case:

$$\begin{aligned} R_B^I\Big|_{\sin\theta=0,\ \cos\theta=-1} &= \begin{bmatrix} -cos(\psi)cos(\phi) - sin(\phi)sin(\psi) & cos(\psi)sin(\phi) - sin(\psi)cos(\phi) & 0 \\ -sin(\psi)cos(\phi) + cos(\psi)sin(\phi) & sin(\psi)sin(\phi) + cos(\psi)cos(\phi) & 0 \\ 0 & 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -cos(\psi-\phi) & -sin(\psi-\phi) & 0 \\ -sin(\psi-\phi) & cos(\psi-\phi) & 0 \\ 0 & 0 & -1 \end{bmatrix} \end{aligned} \tag{5}$$

When we get the inverse solution in axis-angle representation

$$\theta = \cos^{-1}(-\cos(\psi+\phi) + \cos(\psi+\phi) - 1 - 1) = \cos^{-1}(1) = 0$$

we then get a singularity regardless of ψ and ϕ.

1.3 Exercise 1.3

Starting from the rotation matrix:

$$R_B^I = \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\theta)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\theta) & \sin(\psi)\sin(\theta)\sin(\phi) + \cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

With $c_\theta = 0$ and using trigonometric identities as in 1.2 we get the following two cases:

$$R_B^I\Big|_{\sin\theta=0,\ \cos\theta=1} = \begin{bmatrix} 0 & -\sin(\psi-\phi) & \cos(\psi-\phi) \\ 0 & \cos(\psi-\phi) & \sin(\psi-\phi) \\ -1 & 0 & 0 \end{bmatrix}$$

When we get the inverse solution in axis-angle representation:

$$\theta = \cos^{-1}\left(\frac{0 + \cos(\psi-\phi) + 0 - 1}{2}\right) = \cos^{-1}\left(\frac{\cos(\psi-\phi) - 1}{2}\right)$$

we then have a singularity at $\psi - \phi = \pi$. And the other case:

$$R_B^I\Big|_{\sin\theta=0,\ \cos\theta=-1} = \begin{bmatrix} 0 & -\sin(\psi+\phi) & -\cos(\psi+\phi) \\ 0 & \cos(\psi+\phi) & -\sin(\psi+\phi) \\ 1 & 0 & 0 \end{bmatrix}$$

When we get the inverse solution in axis-angle representation:

$$\theta = \cos^{-1}\left(\frac{0 + \cos(\psi+\phi) + 0 - 1}{2}\right) = \cos^{-1}\left(\frac{\cos(\psi+\phi) - 1}{2}\right)$$

we then have a singularity at $\psi + \phi = \pi$.

1.4 Exercise 1.4

The axis is determined using the cross-product:

$$r = v \times w = \begin{bmatrix} v_2w_3 - v_3w_2 \\ v_3w_1 - v_1w_3 \\ v_1w_2 - v_2w_1 \end{bmatrix}$$

The angle is determined using:

$$\cos\theta = \frac{v \cdot w}{|v||w|}$$

since v and w are unit vectors:

$$\cos\theta = v \cdot w,\ \theta = \cos^{-1}(v \cdot w)$$

Using the quaternion encoding from slide 83:

$$\begin{pmatrix} a \\ b \\ c \\ a \end{pmatrix} = \begin{pmatrix} \cos\frac{\theta}{2} \\ r_x \sin\frac{\theta}{2} \\ r_y \sin\frac{\theta}{2} \\ r_z \sin\frac{\theta}{2} \end{pmatrix} = \begin{pmatrix} cos(\frac{1}{2}\theta) \\ (v_2w_3 - v_3w_2)sin(\frac{1}{2}\theta) \\ (v_3w_1 - v_1w_3)sin(\frac{1}{2}\theta) \\ (v_1w_2 - v_2w_1)sin(\frac{1}{2}\theta) \end{pmatrix}$$

1.5 Exercise 1.5

Answer the following questions with explanations:

1.5.1 What is the quaternion q1 that represents the rotation of 180 degree about the x-axis?

Starting with $R_{x180} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \rightarrow Trace(R_{x180}) = -1$, we then have:

$$|q_0| = \sqrt{\frac{-1+1}{4}} = 0 \tag{6a}$$

$$|q_1| = \sqrt{\frac{1}{2} + \frac{1+1}{4}} = 1 \tag{6b}$$

$$|q_2| = \sqrt{\frac{-1}{2} + \frac{2}{4}} = 0 \tag{6c}$$

$$|q_3| = \sqrt{\frac{-1}{2} + \frac{2}{4}} = 0 \tag{6d}$$

1.5.2 What is the quaternion q2 that represents the rotation of 180 degrees about the z-axis?

Starting with $R_{z180} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow Trace(R_{z180}) = -1$, we then have - following the same logic as for the x rotation:

$$|q_0| = 0 \tag{7a}$$

$$|q_1| = 0 \tag{7b}$$

$$|q_2| = 0 \tag{7c}$$

$$|q_3| = 1 \tag{7d}$$

1.5.3 What rotation is represented by composite quaternion q = q1q2? Answer by specifying its rotation angle and axis.

Given the two previous answers, we have:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0-0-0-0 \\ 0+0+0-0 \\ 0+0-1+0 \\ 0+0+0-0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} cos(\frac{\theta}{2}) \\ r_x sin(\frac{\theta}{2}) \\ r_y sin(\frac{\theta}{2}) \\ r_z sin(\frac{\theta}{2}) \end{bmatrix} \tag{8}$$

Which has the solutions:

$$cos^{-1}(0) = \pm\frac{\pi}{2} = \frac{\theta}{2} \longrightarrow \theta = \begin{bmatrix} \pi \\ -\pi \end{bmatrix} \tag{9}$$

Geometrically one can view this as turning 180 degrees around the y-axis.

1.6 Exercise 1.6

In this exercise, we shall compare the number of additions and multiplications needed to perform the following things.

- Compose two rotational matrices - For composing two rotational matrices we have $9(9M + 2A) = 27M + 18A$ where M is multiplications and A is additions.

- Compose two quaternions - For two quaternions, multiplying them together takes a total of $16M + 12A$ in accordance with the method presented in [1].
- Apply a rotation matrix to a vector - Applying a rotational matrix to a vector is the same as multiplying it with said vector. Hence it takes $9M + 6A$.
- Apply a quaternion to a vector - Finally, applying a quaternion takes $3(7A + 13M) = 39M + 21A$, given by the formula for the conjugation $q_B = q_R q_A q_R^*$.

2 Part II: Modelling

2.1 Exercise 2.1

2.1.1 Define the rotation matrix representing the orientation of the body-fixed frame w.r.t. the inertial frame.

This is a combination of three rotation matrices around each axis. However, the angles in the body frame should be negative to do an inverse mapping from the body-fixed frame to the inertial one.

$$R_B^I(\phi,\theta,\psi) = R_I^{v1}(-\psi)R_{v1}^{v2}(-\theta)R_{v2}^B(-\phi)$$

Individual rotation around the axes above are:

$$R_I^{v1}(\psi) = \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R_{v1}^{v2}(\theta) = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \quad R_{v2}^{B}(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix}$$

2.1.2 Define the relation between the angular velocity $\dot{\Theta}$ and the rotational velocity of the body-fixed frame ω.

This relation is given to us from the slides, and is defined as follows:

$$\vec{\omega} = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \dot{\vec{\theta}}$$

2.1.3 Write the linear and angular dynamic equation of the drone in a compact form and clearly show each component of the equation explicitly.

In general, UAV linear dynamics are described by:

$$m\ddot{\mathbf{x}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_B^I F_B + F_D$$

where m - mass in the CoG, first vector on the right-hand side represents gravity, F_B - the total thrust of propellers, multiplying F_B by R_B^I (same as in the previous question) changes the reference frame to a static world frame and F_D - viscous damping. Expanding each element explicitly:

$$m\ddot{\mathbf{x}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_B^I k \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^4 \Omega_i^2 \end{bmatrix} - D\dot{\mathbf{x}}$$

Angular dynamics:

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

we are interested in angular rates thus:

$$\dot{\omega} = \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = I^{-1}\left(\tau_B - \omega \times (I\omega)\right)$$

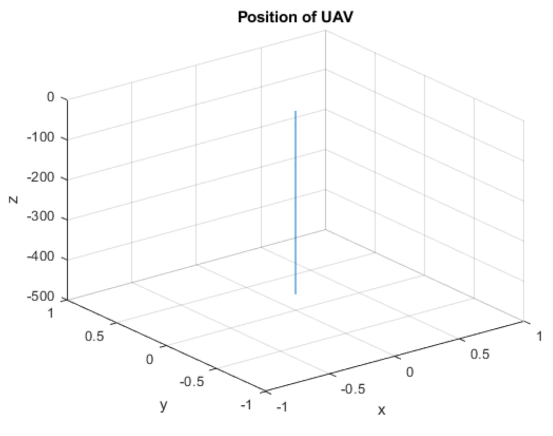
where I - moments of inertia matrix, τ - propeller generated torques, ω - angular speed vector. Expanding each element:

$$\dot{\omega} = \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = I^{-1}\left(\begin{bmatrix} Lk\left(\Omega_1^2 - \Omega_3^2\right) \\ Lk\left(\Omega_2^2 - \Omega_4^2\right) \\ b\left(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2\right) \end{bmatrix} - \omega \times (I\omega)\right)$$

$$\dot{\omega} = \begin{bmatrix} Lk\left(\Omega_1^2 - \Omega_3^2\right) I_{xx}^{-1} \\ Lk\left(\Omega_2^2 - \Omega_4^2\right) I_{yy}^{-1} \\ b\left(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2\right) I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix}$$

2.1.4 Make a MATLAB/Simulink model of the drone, given initial conditions $\mathbf{p}(0) = [0,0,0]^T$, $\dot{\mathbf{p}}(0) = [0,0,0]^T$, $\Theta(0) = [0,0,0]^T$ and $\dot{\Theta}(0) = [0,0,0]^T$, and report the following:

- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0,0,0,0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [10000,0,10000,0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0,10000,0,10000]^T$ and explain the result



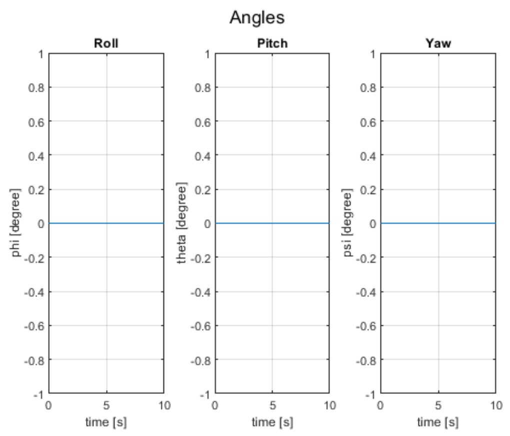
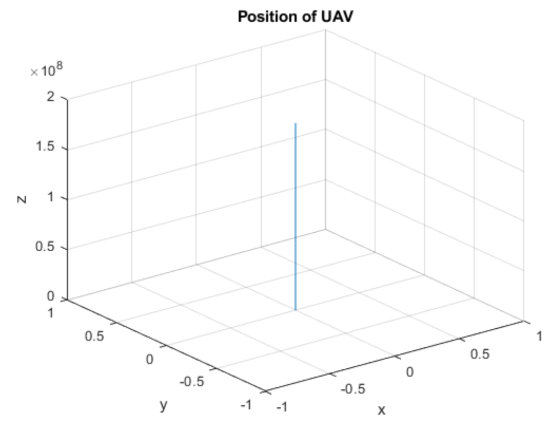


Figure 1: Position of UAV for $\Omega = [0,0,0,0]^T$

Above the situation in which no propellers are spinning is shown. Unsurprisingly, this leads to an unbounded drop in the z-position as gravity is the only force affecting the drone. Moreover, it is unbounded since we have not introduced, for example, a ground-level into which it would collide. The UAV does not spin as with all propellers forces being 0, and they are still equal - although one cannot precisely say they cancel each other out.



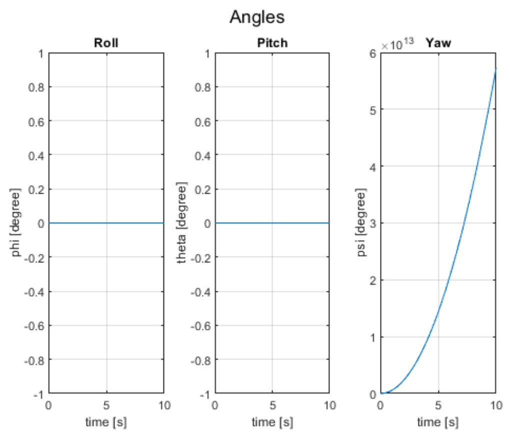


Figure 2: Position of UAV for $\Omega = [10000,0,10000,0]^T$

Introducing a high propeller speed on two diagonal propellers while keeping the other two zero causes two things to happen. Firstly, the combined upwards force generated by the propellers is far more significant than the downwards pull of gravity - which we will later see would give rotor speeds of roughly 11. Thus the UAV starts accelerating upwards. It does so until the drag forces cause the sum of forces to enter an equilibrium, in which case we have achieved terminal velocity. Moreover, the two rotors produce a torque on the body of the UAV, which causes it to spin around its z-axis. Under normal circumstances, this spin of the body would have accelerated until it equalled the torque generated from the propellers. Yet, this model does not actually take this into account, so it takes the form of an exponential equation with constantly accelerating rotations.

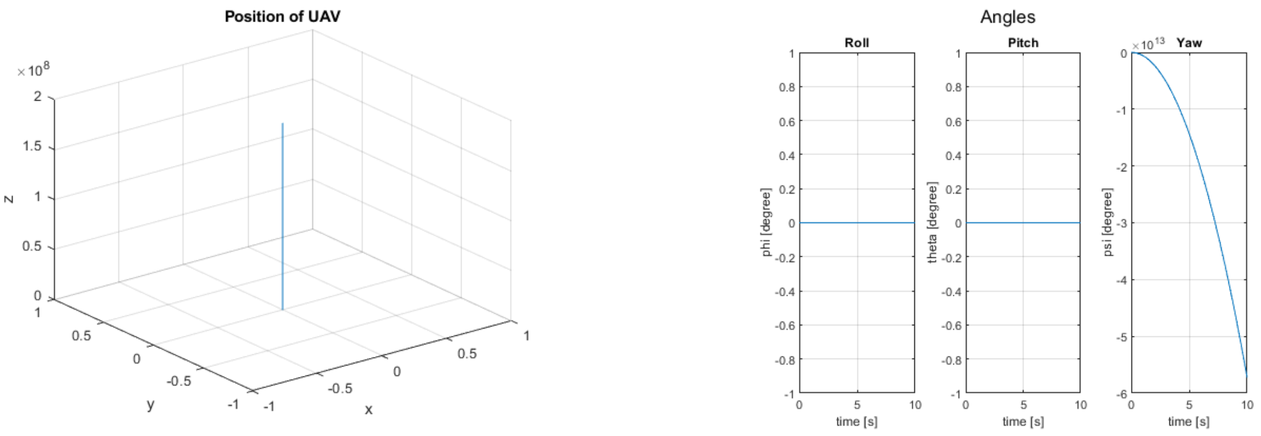


Figure 3: Position of UAV for $\Omega = [0, 10000, 0, 10000]^T$

Introducing the same situation, except on the other diagonal, produces the same result as before, except now the body spins in the opposite direction. This is why when all four propellers are spinning, there would ideally be no spin around the z-axis, as the torques produced by the diagonals would cancel each other out.

2.2 Exercise 2.2

2.2.1 Define the rotation matrix representing the orientation of the body-fixed frame w.r.t. the inertial frame

Having Euler angles, we can represent the rotation by quaternion by [1]:

$$q_\phi = \cos\frac{\phi}{2} + i\sin\frac{\phi}{2}$$
$$q_\theta = \cos\frac{\theta}{2} + j\sin\frac{\theta}{2}$$
$$q_\psi = \cos\frac{\psi}{2} + k\sin\frac{\psi}{2}$$

These represent rotation about each axis (x, y, z). It can combine these three distinct axis rotations into a single quaternion vector. The real parts should be added together and imaginary ones placed in the remaining elements of the vector. An important note is that quaternion must be normalized for square constants, to sum up to one. Resulting quaternion with four numbers (one real by summing real parts and three-unit vectors):

$$q = \cos\left(\frac{\phi}{2}\right) + \cos\left(\frac{\theta}{2}\right) + \cos\left(\frac{\phi}{2}\right) + i\sin\left(\frac{\phi}{2}\right) + j\sin\left(\frac{\theta}{2}\right) + k\sin\left(\frac{\psi}{2}\right)$$

2.2.2 Write the linear and angular dynamic equation of the drone in a compact form and clearly show each component of the equation explicitly.

We are only doing a rotation in the dynamical model equations in the linear case where the total thrust of propellers F_B is converted to the inertial frame of reference. Now working with quaternions, we can rotate a vector:

$$\mathbf{v}_B = \mathbf{q}_i^b \begin{pmatrix} 0 \\ \mathbf{v}_I \end{pmatrix} \left(\mathbf{q}_i^b\right)^{-1}$$

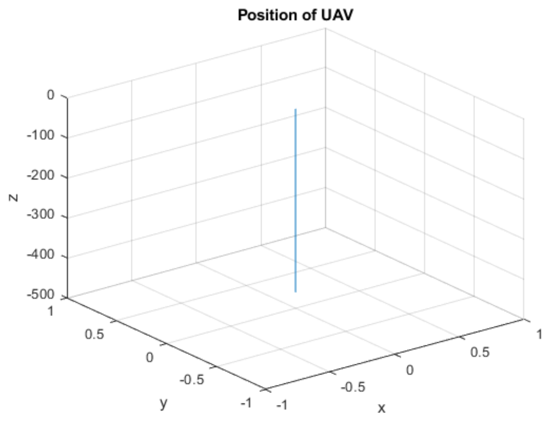
And the linear dynamic equation:

$$m\ddot{\mathbf{x}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \mathbf{q}_i^b \begin{pmatrix} 0 \\ \mathbf{F}_B \end{pmatrix} \left(\mathbf{q}_i^b\right)^{-1} + F_D$$

Rotational dynamics are unchanged because there are no rotations involved there.

2.2.3 Make a MATLAB/Simulink model of the drone, given initial conditions $\mathbf{p}(0) = [0,0,0]^T$, $\dot{\mathbf{p}}(0) = [0,0,0]^T, \Theta(0) = [0,0,0]^T$ and $\dot{\Theta}(0) = [0,0,0]^T$, and report the following:

- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0,0,0,0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [10000,0,10000,0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0,10000,0,10000]^T$ and explain the result



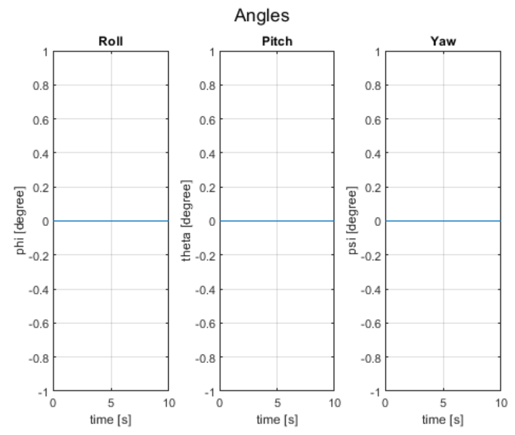


Figure 4: Position of UAV for $\Omega = [0,0,0,0]^T$

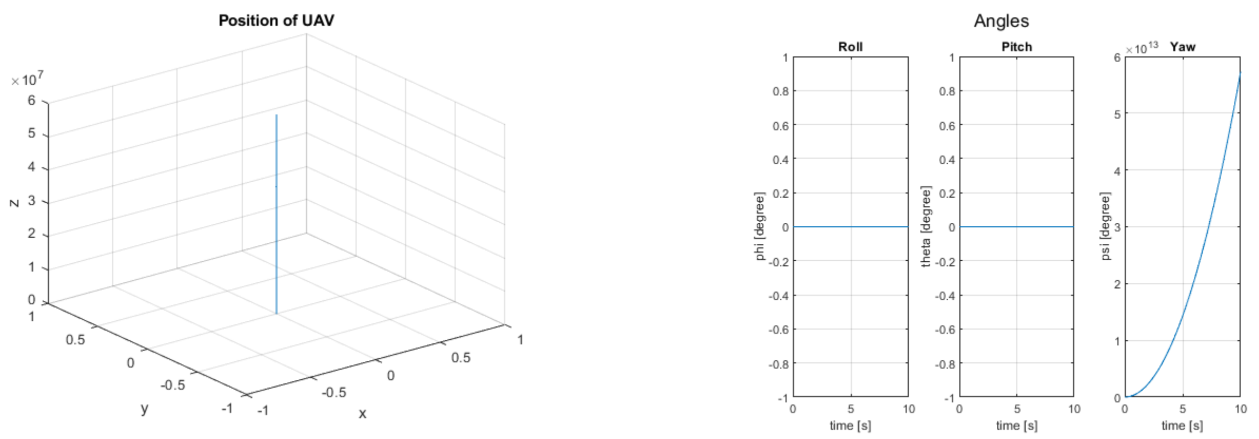


Figure 5: Position of UAV for $\Omega = [10000, 0, 10000, 0]^T$

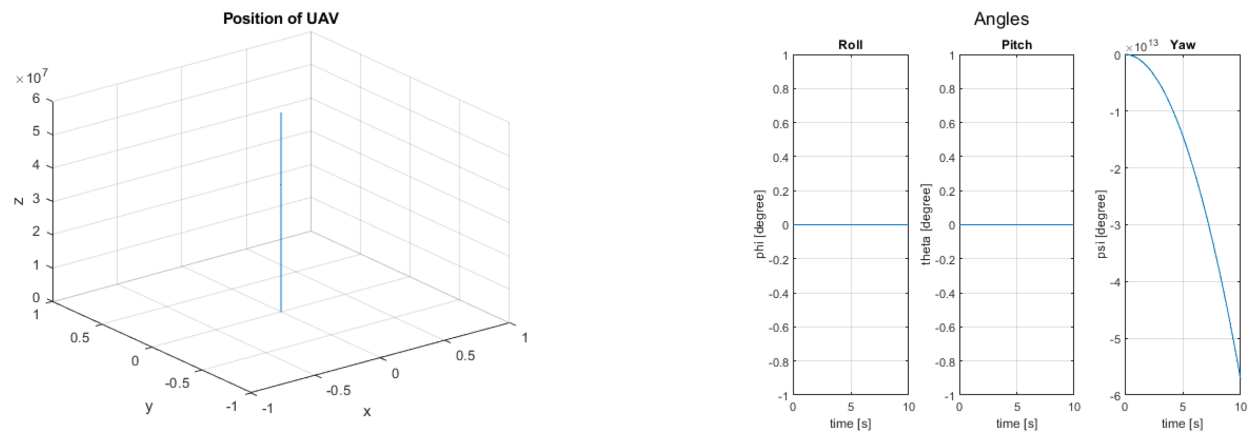


Figure 6: Position of UAV for $\Omega = [0, 10000, 0, 10000]^T$

2.3 Exercise 2.3

Using the model in Exercise 2.1 (or 2.2), linearize the dynamic model of the UAV in hovering conditions. Compare the linearized model with the non-linear one under the same input conditions as in previous exercises (2.1 and 2.2 if solved):

- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0, 0, 0, 0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [10000, 0, 10000, 0]^T$ and explain the result
- Make a plot of $\mathbf{p}$ and Θ, given $\Omega = [0, 10000, 0, 10000]^T$ and explain the result

We linearize the dynamics equations both on state and input variables on hovering conditions for the linear: the total force should be zero:

$$m\ddot{\mathbf{x}} = m\mathbf{g} + R \cdot k \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} \Omega_i^2 \end{bmatrix} - D\dot{\mathbf{x}} = \mathbf{0}$$

with the initial conditions $\mathbf{p}(0) = [0, 0, 0]^T$, $\dot{\mathbf{p}}(0) = [0, 0, 0]^T, \Theta(0) = [0, 0, 0]^T$ and $\dot{\Theta}(0) = [0, 0, 0]^T$ we get:

$$m\mathbf{g} + k \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} \Omega_i^2 \end{bmatrix} = \mathbf{0}$$

Given the equation for rotation, all rotors need to rotate at the same speed:

$$4k\Omega_0^2 = -mg_z, \Omega_0 = \frac{\sqrt{-kmg_z}}{2k} \approx 11.1$$

this is the initial condition for the rotor speeds.
With this we linearize with the first order taylor expansion. $\widehat{A}$ denoting a linearized A:

$$RF_B = \begin{bmatrix} \left(\cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi)\right) k \left({\Omega_1}^2 + {\Omega_2}^2 + {\Omega_3}^2 + {\Omega_4}^2\right) \\ \left(\sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi)\right) k \left({\Omega_1}^2 + {\Omega_2}^2 + {\Omega_3}^2 + {\Omega_4}^2\right) \\ \cos(\theta)\cos(\phi)\, k \left({\Omega_1}^2 + {\Omega_2}^2 + {\Omega_3}^2 + {\Omega_4}^2\right) \end{bmatrix}$$

$$\widehat{RF}_B = \begin{bmatrix} 4k\Omega_0^2\theta \\ -4k\Omega_0^2\phi \\ 2k\Omega_0(\Omega_1 + \Omega_2 + \Omega_3 + \Omega_4 - 2\Omega_0) \end{bmatrix}$$

For the rotational dynamics we have:

$$\dot{\omega} = I^{-1}(\tau_B - \omega \times (I\omega))$$

where the term with the cross product disappears with linearization:

$$I^{-1}\tau_B = \begin{bmatrix} Lk(\Omega_1^2 - \Omega_3^2)/I_{xx} \\ Lk(\Omega_2^2 - \Omega_4^2)/I_{yy} \\ b(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2)/I_{zz} \end{bmatrix}$$

$$\widehat{\dot{\omega}} = 2\Omega_0 \begin{bmatrix} Lk(\Omega_1 - \Omega_3)/I_{xx} \\ Lk(\Omega_2 - \Omega_4)/I_{yy} \\ b(\Omega_1 - \Omega_2 + \Omega_3 - \Omega_4)/I_{zz} \end{bmatrix}$$

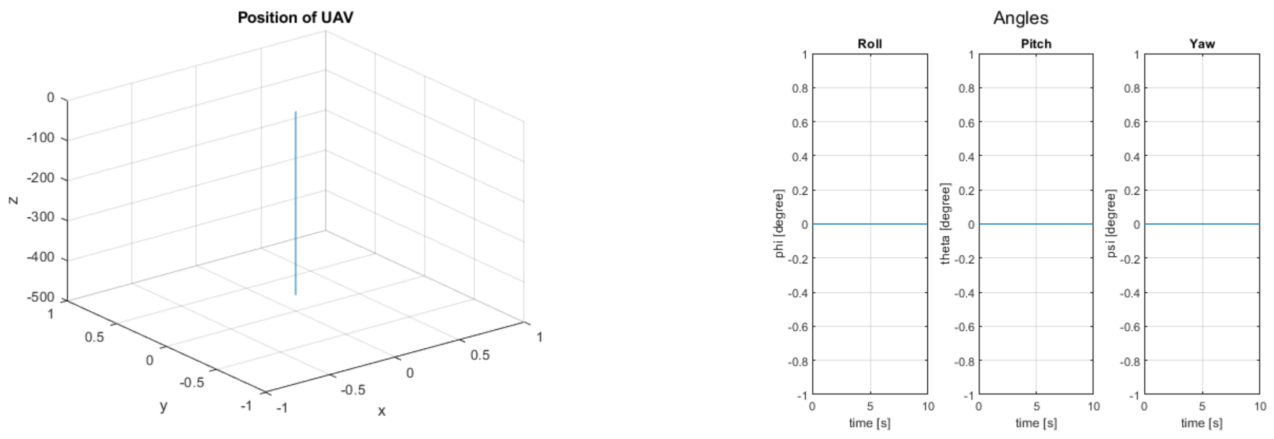


Figure 7: Position of UAV for $\Omega = [0, 0, 0, 0]^T$

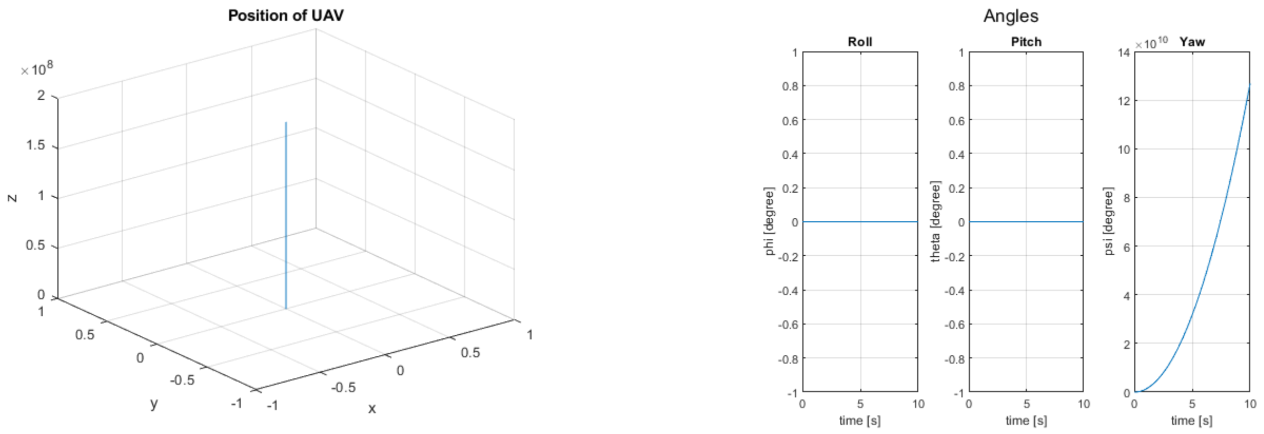


Figure 8: Position of UAV for $\Omega = [10000, 0, 10000, 0]^T$

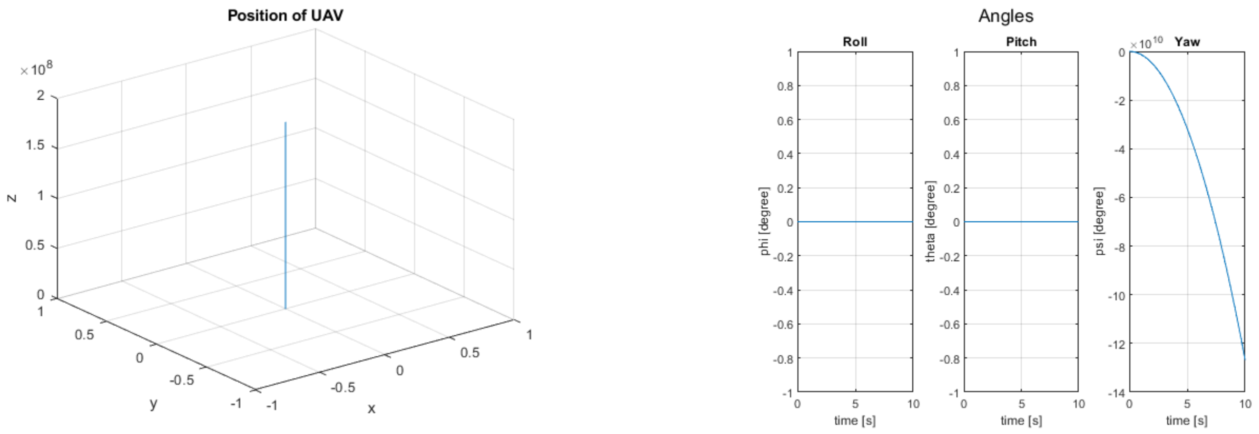


Figure 9: Position of UAV for $\Omega = [0, 10000, 0, 10000]^T$

3 Part III: Control

When simulating, we see an initial drop in z, which is assumed to represent a non-actuated system being dropped. For the tests in 3.1 and 3.2, we let the system settle into hovering equilibrium and then apply step reference values for the tests. The plots are limited only to show the step response.

3.1 Exercise 3.1

The implementation of the controller for the linearised system can be found in the appendix 54, but to give a short overview; the attitude references (roll, pitch, yaw and altitude) are passed through a step block. Then, the errors for said references are found and passed through the PID - which consists of a proportional (K_p) gain, as well as an integral and a derivative. These are then fed into a set of gains, where the first two simply adds mg and derivates by k and the cosines of pitch and roll to create the force we want. We then multiply by a matrix fo the inertias (as well as a 1 to keep the force as is), before retrieving our motor speeds with another multiplication - and reverting back to our torques by multiplying with the inverse of the same vector afterwars. We started with a Ziegler-Nichols tuning on the linearized system. According to the methodology of Ziegler-Nichols, we should begin by setting $K_i, K_d = 0$ and have K_p be small. Then, we gradually increase K_p until marginal stability is reached, i.e. we have standing waves. This value is then labelled K_c, and we approximate the period for this gain value before moving on. These can then be used to give a good initial point for tuning the entire system. Doing so, we got gain $K_c = 1$ and period $T_c = 1.34$, which gave the following gains:

$$K_p = 0.6 * K_c = 0.6$$

$$K_d = T_c/8 = 0.1675$$

$$K_i = 2/T_c = 1.4925$$

After further tuning we arrived at:

$$K_p = 1.5$$

$$K_d = 0.8$$

$$K_i = 0.1$$

3.1.1 Option 1: $\Theta^* = [10,\ 0,\ 0]^T\ [deg],\ z^* = 0[m]$

See figure 10. We see y decreases uncontrolled as expected with a constant roll. Rise time is about 1s. Also, it is worth noting that although we observe what seems to be an unstable ψ, the scale is that of 10^{14}, so the impact on the system is negligible. Yet, this kind of behaviour appears in a couple of different situations.

3.1.2 Option 2: $\Theta^* = [0,\ 10,\ 0]^T\ [deg],\ z^* = 0[m]$

See figure 11. We have similar results to the above, except now x rises uncontrolled, which is again expected with a constant pitch. Again we see that rise time is around 1s.

3.1.3 Option 3: $\Theta^* = [0,\ 0,\ 10]^T\ [deg],\ z^* = 0[m]$

See figure 12. We see a controlled rotation with no deviation in position or any other angle. Rise time is again around 1s.

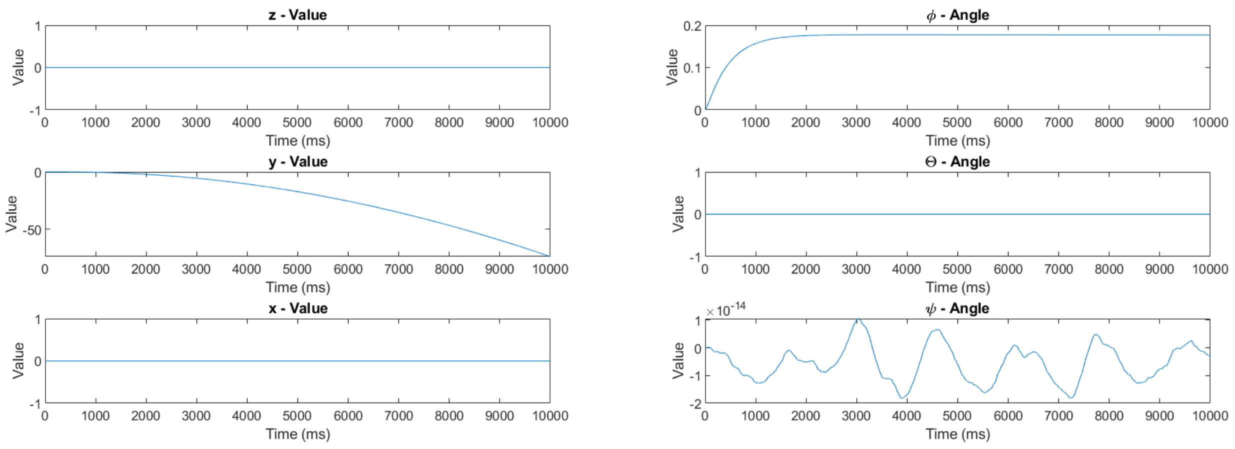


Figure 10: $\Theta^* = [10,\ 0,\ 0]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

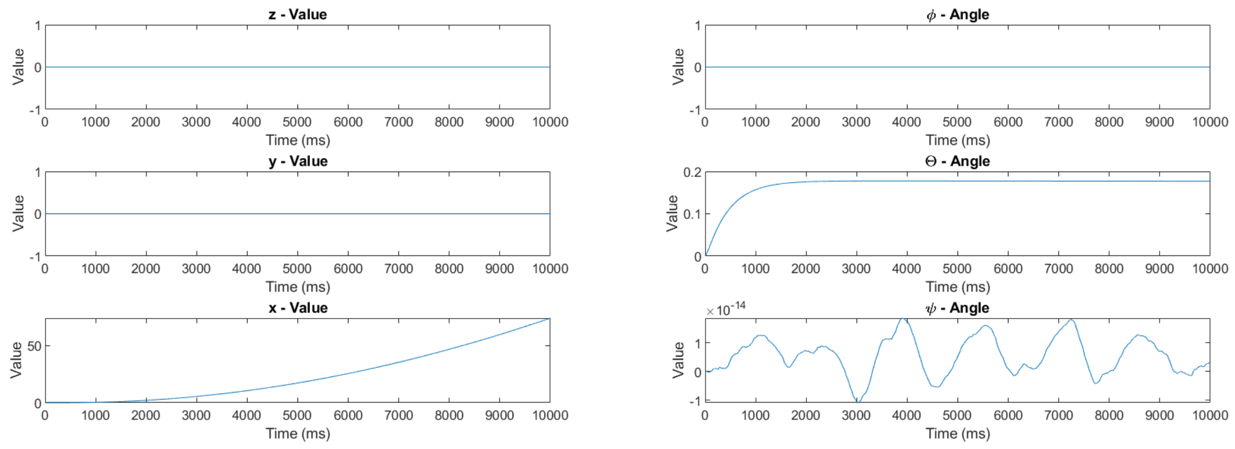


Figure 11: $\Theta^* = [0,\ 10,\ 0]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

3.1.4 Option 4: $\Theta^* = [0,\ 0,\ 0]^T\ [deg],\ z^*1[m]$

See figure 13. We see no rotation and no deviation in x and y. Rise time is around 1s.

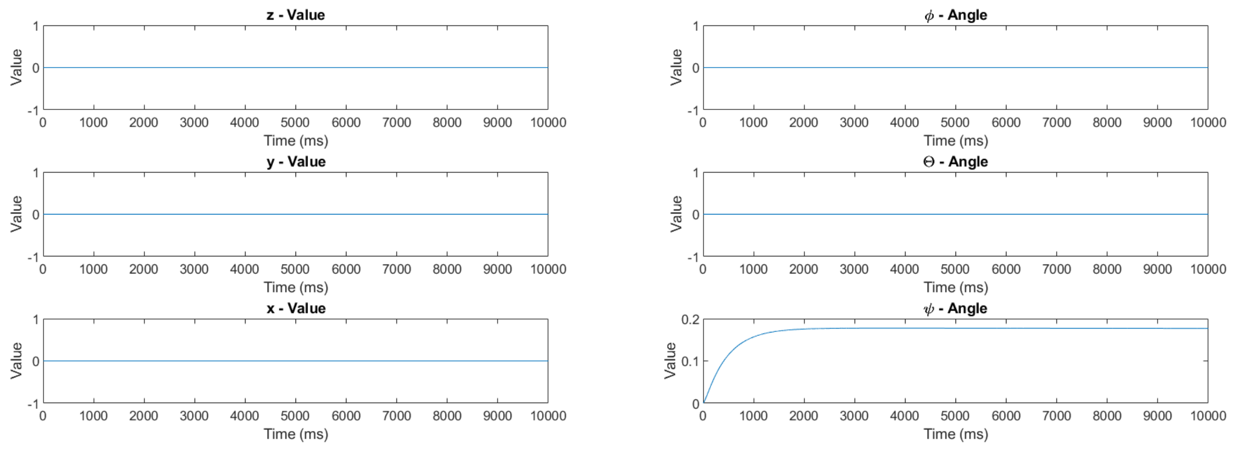


Figure 12: $\Theta^* = [0,\ 0,\ 10]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

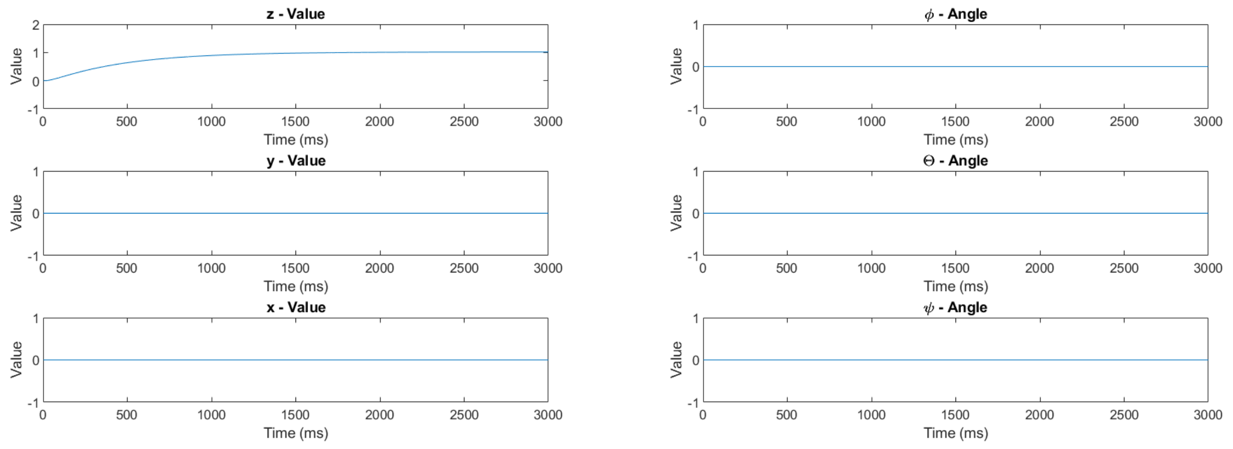


Figure 13: $\Theta^* = [0,\ 0,\ 0]^T\ [deg],\ z^* = 1[m]$. Note the difference in time unit.

3.2 Exercise 3.2

Now we concern ourselves with the non-linear model found in the appendix **??**.

3.2.1 Option 1: $\Theta^* = [10,\ 0,\ 0]^T\ [deg],\ z^* = 0[m]$

See figure 14. For the non-linear system, we see a rise time slightly over 1s and a settling time around 6-7s. We now know an overshoot.

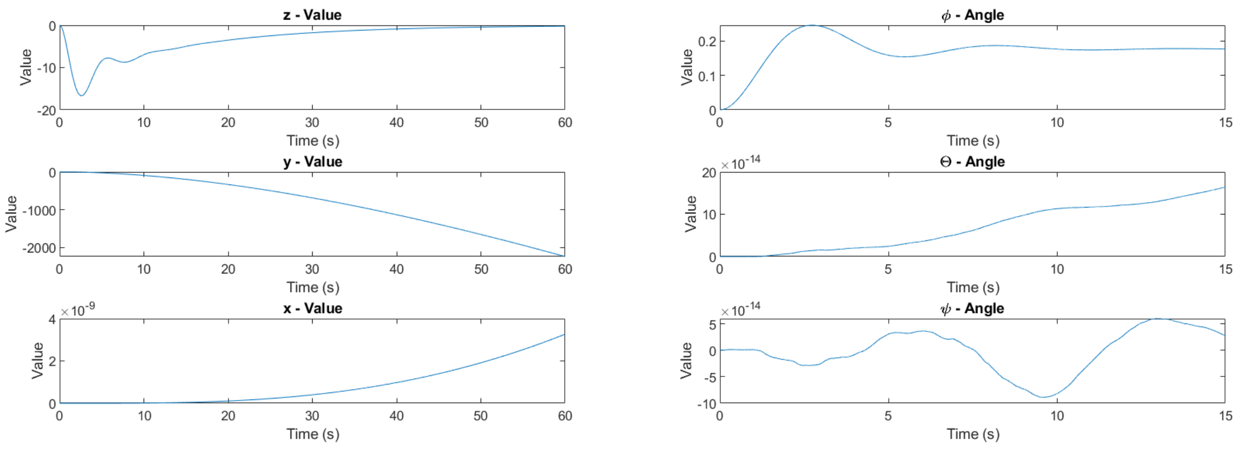


Figure 14: $\Theta^* = [10,\ 0,\ 0]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

3.2.2 Option 2: $\Theta^* = [0,\ 10,\ 0]^T\ [deg],\ z^* = 0[m]$

See figure 15. We get similar results to the roll control.

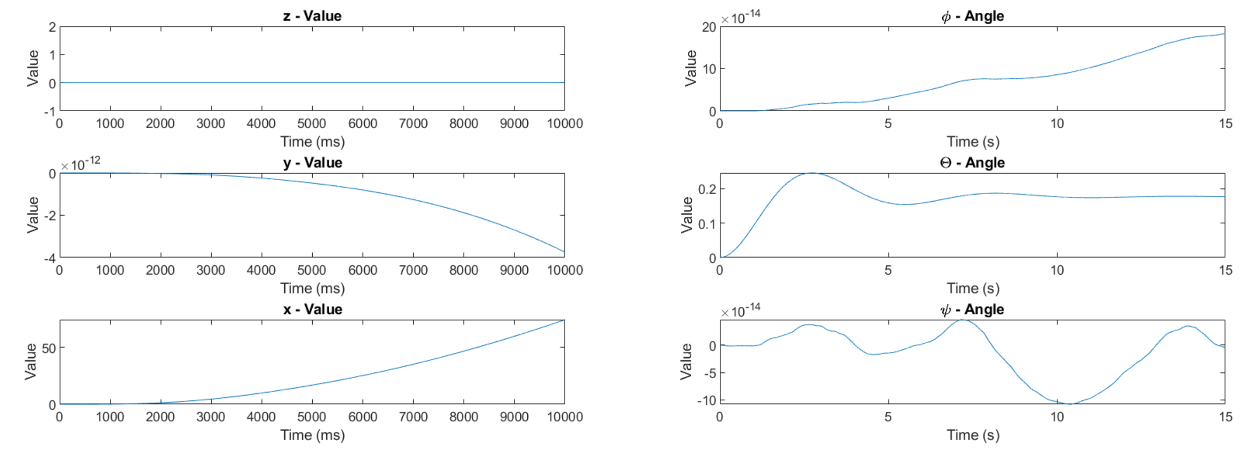


Figure 15: $\Theta^* = [0,\ 10,\ 0]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

3.2.3 Option 3: $\Theta^* = [0,\ 0,\ 10]^T\ [deg],\ z^* = 0[m]$

See figure 16. This is similar to the non-linear case with a rise time around 1s and no overshoot.

3.2.4 Option 4: $\Theta^* = [0,\ 0,\ 0]^T\ [deg],\ z^*1[m]$

See figure 17. Again very similar to the non-linear case with a rise time around 1s and no overshoot.

3.3 Exercise 3.3

We now want to integrate a position controller into the system, which we do as shown in the appendix 51 for the linearized version and 52 for the non-linear. Block diagram wise it functions largely the same way as the attitude controller, using a PID for the errors of the x and y positions. We then multiply the rotation matrix with a relation

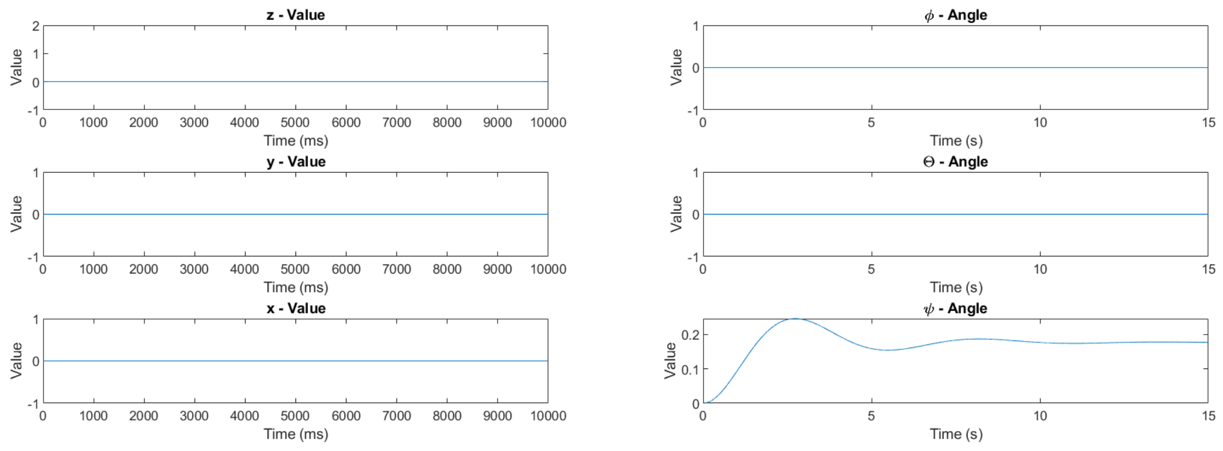


Figure 16: $\Theta^* = [0,\ 0,\ 10]^T\ [deg],\ z^* = 0[m]$. Note the difference in time unit.

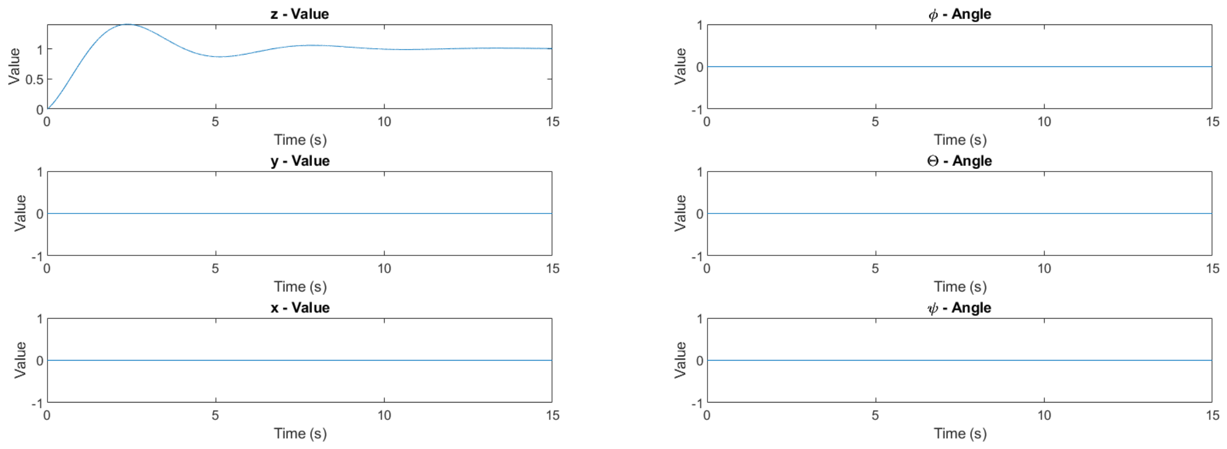


Figure 17: $\Theta^* = [0,\ 0,\ 0]^T\ [deg],\ z^* = 1[m]$. Note the difference in time unit.

matrix which shows the correlation between an error in the x-y position and the required roll-pitch references to fix it. This then turns to our reference signals used in the attitude controller.

We started with a Ziegler-Nichols tuning on the linearized system, as described in a previous section, where we got gain $K_c = 0.155$ and period $T_c = 5$, which gave the following gains:

$$K_p = 0.6 * K_c = 0.093$$

$$K_d = T_c/8 = 0.625$$

$$K_i = 2/T_c = 0.4$$

After further tuning we arrived at:

$$K_{p,xy} = 0.2$$

$$K_{d,xy} = 1.25$$

$$K_{i,xy} = 0.0$$

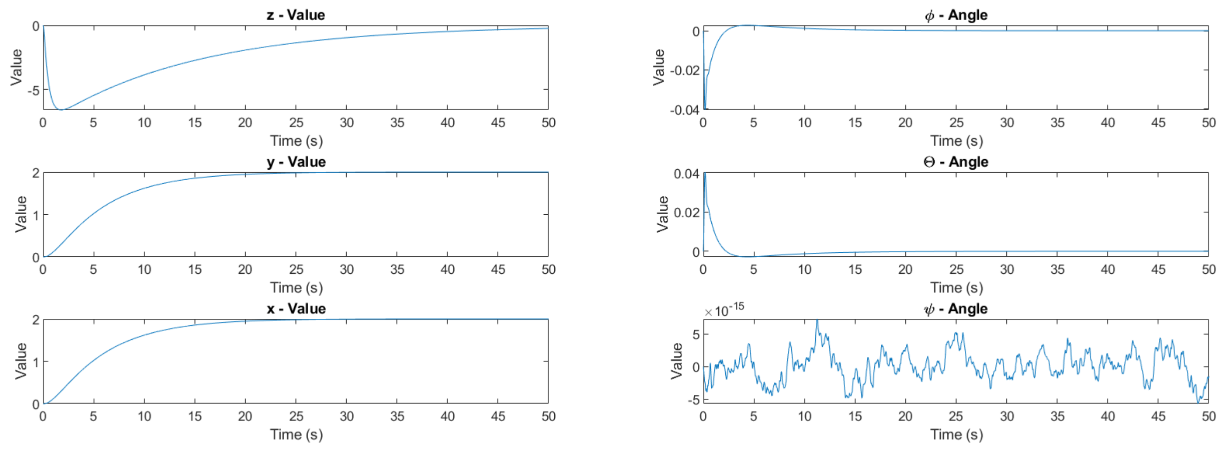
And we retuned the attitude controller to:

$$K_p = 1.0$$

$$K_d = 1.25$$

$$K_i = 0.1$$

In the figures 18-23 we see no position overshoot for the linearized and non-linear systems. We have a significant dip in z, caused by not letting the system reach hovering equilibrium. The rise time for x and y is around 10s.



The l

Figure 18: Linearized system controlling both x and y to reference point 2, all other references are set t0 0

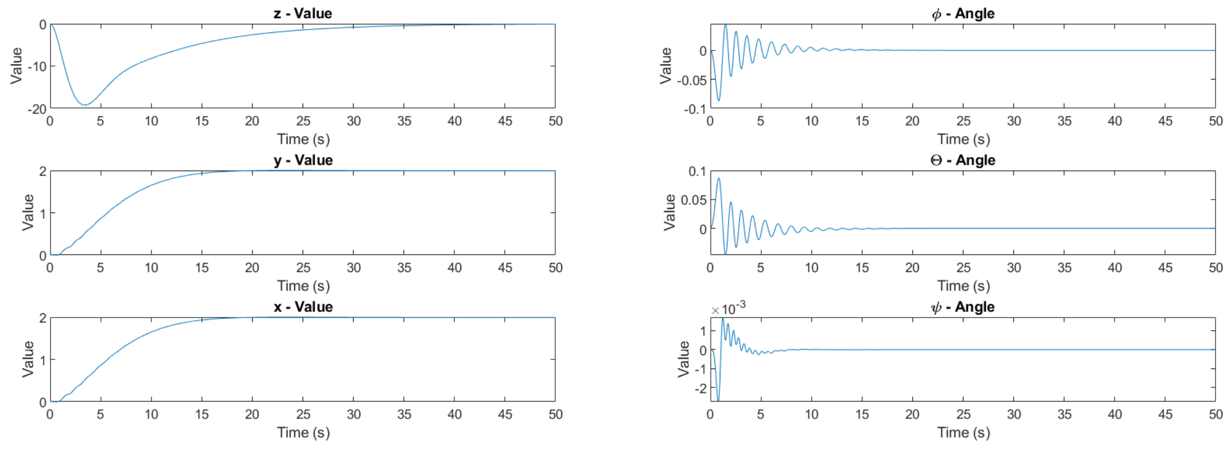
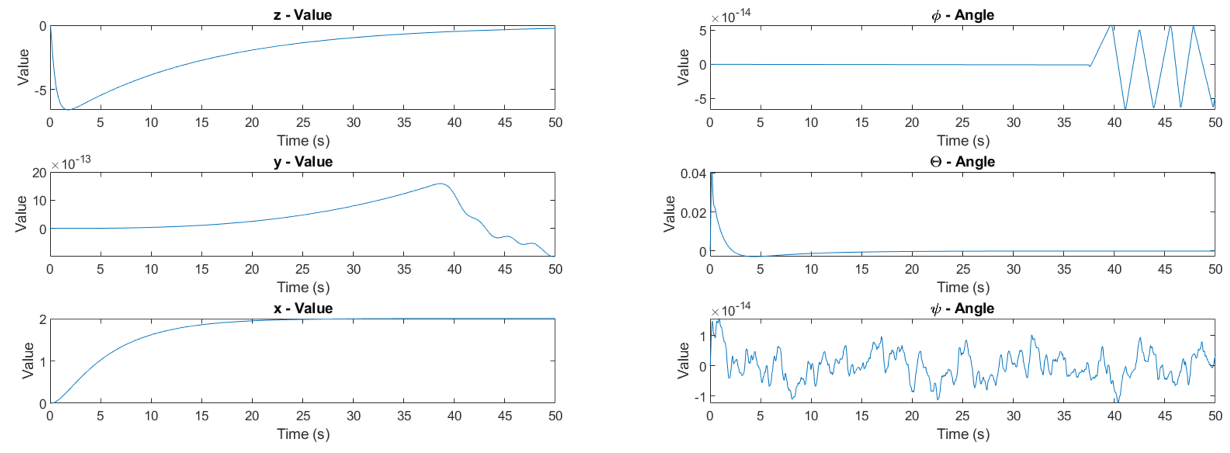


Figure 19: The non-linear system controlling x and y to reference 2, all other references are set to 0



The l

Figure 20: Linearized system controlling towards x = 2, while keeping the rest at 0

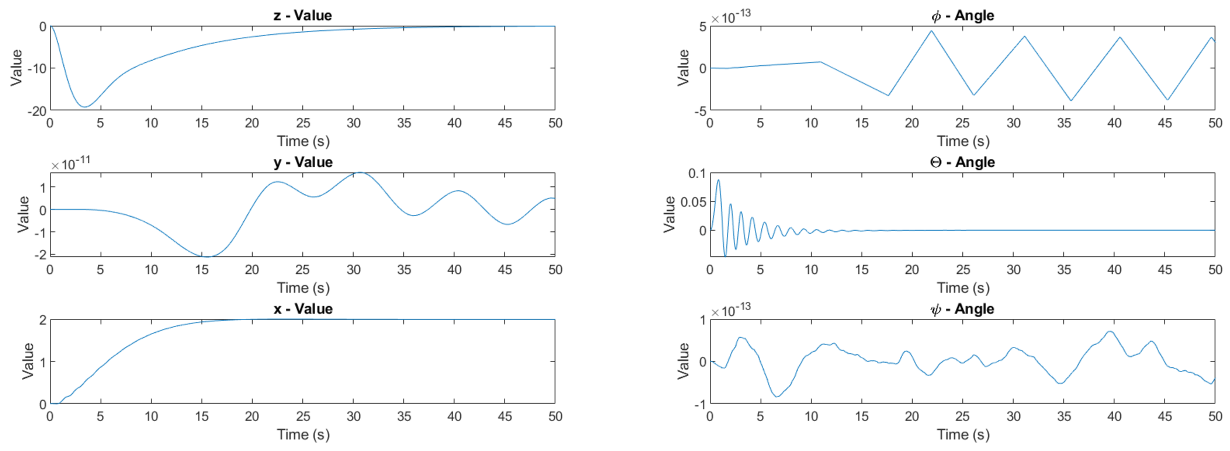
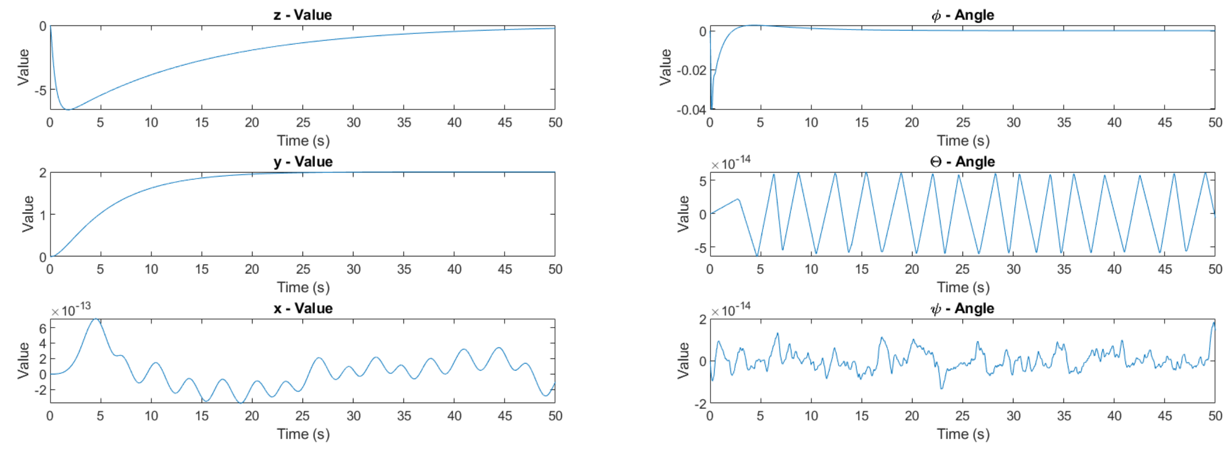


Figure 21: The non-linear system controlling towards x = 2, the other references are 0



The l

Figure 22: Linearized system controlling towards y = 2, the other references are 0

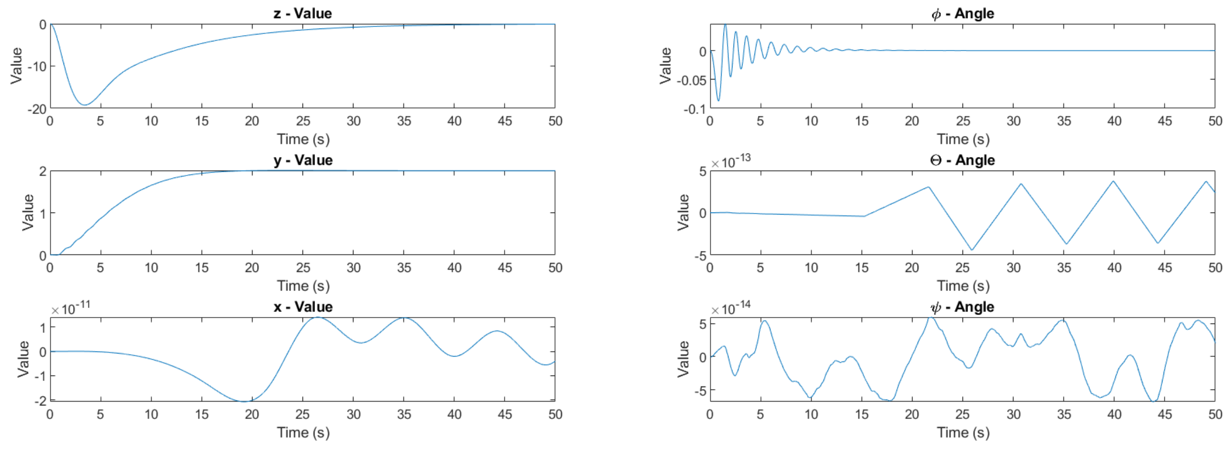


Figure 23: The non-linear system controlling towards y = 2, the other references are 0

4 Part IV: Path Planning

4.1 DFS - Depth First Seach

Start by stacking the lowest/highest of the s numbers when it is possible to stack multiple. Start with s0 as the first frontier in the stack. Write down the steps taken by the algorithm. In which order the nodes are expanded and what frontiers are queued at that point. When the goal is reached, draw the found route to the goal, write down the route from s0 to the goal, draw the layer numbers on each node, or specify to which layer each node belongs. Specify the total number of expanded nodes and the total number of frontiers generated. Show the path, as well as, the length of the path.

- In each iteration, we visit the node on the right side of the stack
- Once we reach the goal node, we stop the algorithm
- Nodes that are already visited are not entered again
- We store information only about the lowest (or highest in the second part) node
- In DFS, we use stack (LIFO)
- Both DFS and BFS are not complete and not optimal
- Stack: nodes that we consider if we reach a leaf or node that visited all of the neighbours
- Visited nodes: nodes that we visited, and we do not visit again (can cause looping)
- Terminal node: Either Leaf node or node where all other nodes were already visited
- Frontier: All nodes that we can expand (aka are not a terminal nodes)

4.1.1 Lowest number

Start by stacking the lowest of the s numbers when it is possible to stack multiple.

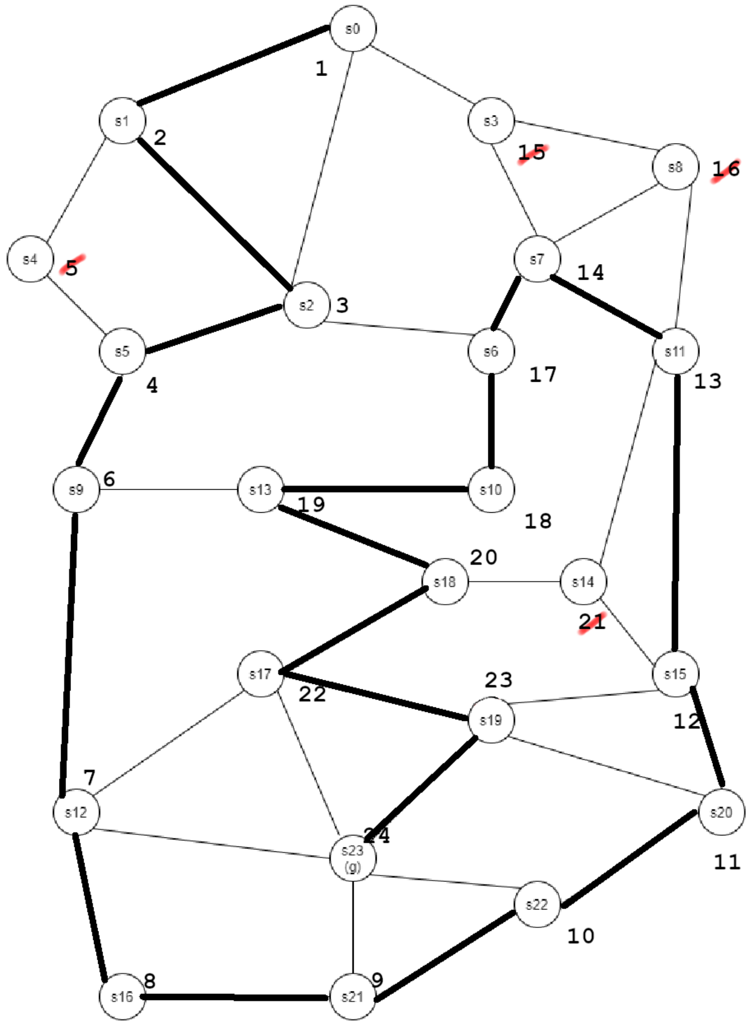


Figure 24: Drawing of DFS while expanding the lowest node

Step 0

- Stack: s0
- Visited Nodes: -
- Terminal Nodes: -

Step 1

- Stack: s0-s1
- Visited Nodes: s0
- Terminal Nodes: -

Step 2

- Stack: s0-s1-s2
- Visited Nodes: s0-s1
- Terminal Nodes: -

Step 3

- Stack: s0-s1-s2-s5
- Visited Nodes: s0-s1-s2
- Terminal Nodes: -

Step 4

- Stack: s0-s1-s2-s5-s4
- Visited Nodes: s0-s1-s2-s5
- Terminal Nodes: -

Step 5 (Expand stack from node s5)

- Stack: s0-s1-s2-s5-s9
- Visited Nodes: s0-s1-s2-s5-s4
- Terminal Nodes: s4 (s4 deleted from stack)

Step 6

- Stack: s0-s1-s2-s5-s9-s12
- Visited Nodes: s0-s1-s2-s5-s4-s9
- Terminal Nodes: s4

Step 7

- Stack: s0-s1-s2-s5-s9-s12-s16
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12
- Terminal Nodes: s4

Step 8

- Stack: s0-s1-s2-s5-s9-s12-s16-s21
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16
- Terminal Nodes: s4

Step 9

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21
- Terminal Nodes: s4

Step 10

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22
- Terminal Nodes: s4

Step 11

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20
- Terminal Nodes: s4

Step 12

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15
- Terminal Nodes: s4

Step 13

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11
- Terminal Nodes: s4

Step 14

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7
- Terminal Nodes: s4

Step 15

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3
- Terminal Nodes: s4

Step 16 (Expand stack from node s7)

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8
- Terminal Nodes: s4-s3-s8 (s3 and s8 deleted from stack)

Step 17

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6
- Terminal Nodes: s4-s3-s8

Step 18

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10
- Terminal Nodes: s4-s3-s8

Step 19

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13-s18
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10-s13
- Terminal Nodes: s4-s3-s8

Step 20

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13-s18-s14
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10-s13-s18
- Terminal Nodes: s4-s3-s8

Step 21

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13-s18-s17
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10-s13-s18-s14
- Terminal Nodes: s4-s3-s8-s14 (s14 deleted from stack)

Step 22

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13-s18-s17-s19
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10-s13-s18-s14-s17
- Terminal Nodes: s4-s3-s8-s14

Step 23

- Stack: s0-s1-s2-s5-s9-s12-s16-s21-s22-s20-s15-s11-s7-s6-s10-s13-s18-s17-s19-s23
- Visited Nodes: s0-s1-s2-s5-s4-s9-s12-s16-s21-s22-s20-s15-s11-s7-s3-s8-s6-s10-s13-s18-s14-s17-s19
- Terminal Nodes: s4-s3-s8-s14

Step 24

- Goal node is reached

```
Node Layer
s0
 s1
  s2
   s5
    s4
    s9
     s12
      s16
       s21
        s22
         s20
          s15
           s11
            s7
             s3
              s8
             s6
              s10
               s13
                s18
                 s14
                 s17
                  s19
                   s23
```

Number of expanded nodes: 24
Number of frontiers: 19 (24 total nodes - 1 goal state - 4 terminal states which could not be expanded)
Path length: 19 (66.4 using the cost of the edge)
Final path: $s0 \rightarrow s1 \rightarrow s2 \rightarrow s5 \rightarrow s9 \rightarrow s12 \rightarrow s16 \rightarrow s21 \rightarrow s22 \rightarrow s20 \rightarrow s15 \rightarrow s11 \rightarrow s7 \rightarrow s6 \rightarrow s10 \rightarrow s13 \rightarrow s18 \rightarrow s17 \rightarrow s19 \rightarrow s23$

4.1.2 Highest number

Start by stacking the highest of the s numbers when it is possible to stack multiple.

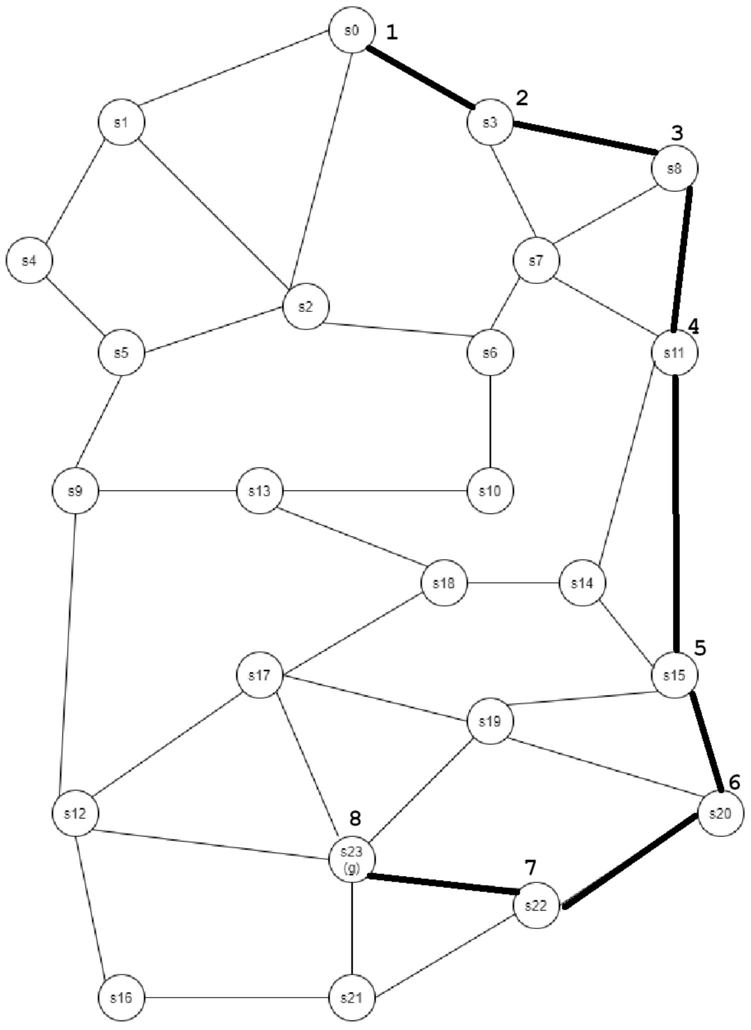


Figure 25: Drawing of DFS while expanding the highest node

Step 0

- Stack: s0
- Visited Nodes: -
- Terminal Nodes: -

Step 1

- Stack: s0-s3
- Visited Nodes: s0
- Terminal Nodes: -

Step 2

- Stack: s0-s3-s8
- Visited Nodes: s0-s3
- Terminal Nodes: -

Step 3

- Stack: s0-s3-s8-s11
- Visited Nodes: s0-s3-s8
- Terminal Nodes: -

Step 4

- Stack: s0-s3-s8-s11-s15
- Visited Nodes: s0-s3-s8-s11
- Terminal Nodes: -

Step 5

- Stack: s0-s3-s8-s11-s15-s20
- Visited Nodes: s0-s3-s8-s11-s15
- Terminal Nodes: -

Step 6

- Stack: s0-s3-s8-s11-s15-s20-s22
- Visited Nodes: s0-s3-s8-s11-s15-s20
- Terminal Nodes: - ; Frontiers: -

Step 7

- Stack: s0-s3-s8-s11-s15-s20-s22-s23
- Visited Nodes: s0-s3-s8-s11-s15-s20-s22
- Terminal Nodes: -

Step 8

- Goal node is reached

```
Node Layer
s0
 s3
  s8
   s11
    s15
     s20
      s22
       s23
```

Number of expanded nodes: 8
Number of frontiers: 7 (8 expanded nodes - 1 goal state which was not expanded)
Path length: 7 (23.6 using the cost of the edge)
Final path: $s0 \rightarrow s3 \rightarrow s8 \rightarrow s11 \rightarrow s15 \rightarrow s20 \rightarrow s22 \rightarrow s23$

4.1.3 Comparison

The results are significantly different, although we have used the same method in both cases. We want to reach the node with the highest number. Therefore prioritizing higher numbers will reach the goal much faster. If we choose the lower number in our graph, where each node has at least two ancestors, this will take much more time, as we need to discover nearly the whole graph before reaching our desired node.

4.2 BFS - Breadth-First Search

Run the same algorithm as for DFS, where each edge has the exact cost. In BFS, we are guaranteed to find the shortest path, but the algorithm is computationally heavy. In the following exercises, we will work with more efficient algorithms.

- Once we reach the goal node, we stop the algorithm
- Nodes that are already visited are not entered again
- We need to store the parent node
- We firstly expand the node with the lowest value
- In BFS, we use the queue (FIFO)
- Queue: nodes that we consider if we reach a leaf or node that visited all of the neighbours
- Visited nodes: nodes that we visited-we do not visit again (can cause looping)
- Terminal node: Either Leaf node or node where all other nodes were already visited
- Frontier: All nodes that we can expand (aka are not terminal nodes)

4.2.1 Algorithm

Expand the node with the lowest number. In the first row, we see the current node with its parent p() and in the second row, its successors.

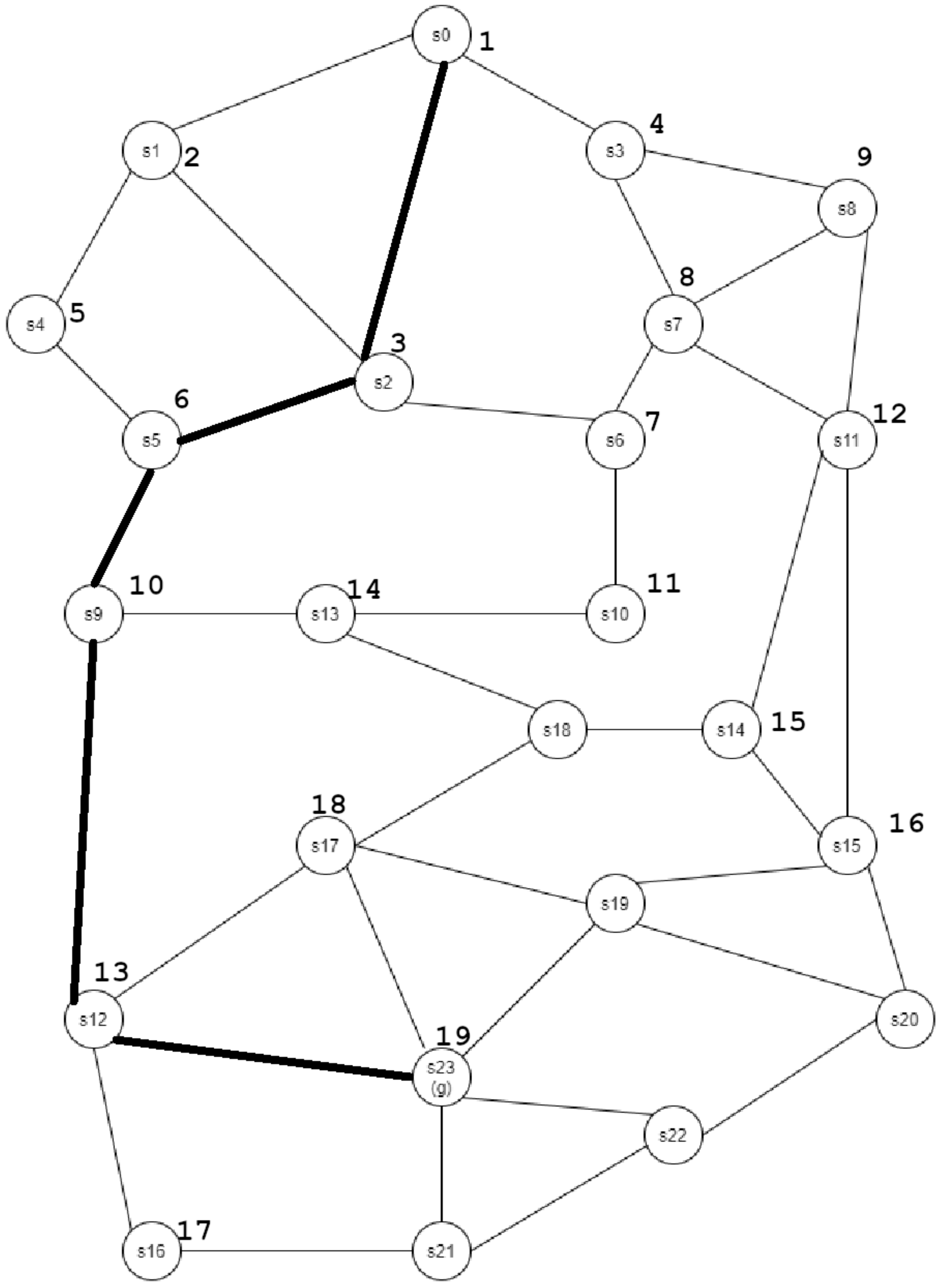


Figure 26: Drawing of BFS while expanding the lowest node

Step 1:

s0/p(-)		
s1	s2	s3

Step 2:

s1/p(s0)
s4

Step 3:

s2/p(s0)	
s5	s6

Step 4:

s3/p(s0)	
s7	s8

Step 5:

s4/p(s1)
-

Step 6:

s5/p(s2)
s9

Step 7:

s6/p(s2)
s10

Step 8:

s7/p(s3)
s11

Step 9:

s8/p(s3)
-

Step 10:

s9/p(s5)	
s12	s13

Step 11:

s10/p(s6)
-

Step 12:

s11/p(s7)	
s14	s15

Step 13:

s12/p(s9)		
s16	s17	s23

Step 14:

s13/p(s9)
s18

Step 15:

s14/p(s11)
-

Step 16:

s15/p(s11)	
s19	s20

Step 17:

s16/p(s12)
s21

Step 18:

s17/p(s12)
-

Step 19:

s23/p(12)
x

```
Node Layer
s0
 s1
 s2
 s3
  s4
  s5
  s6
  s7
```

```
s8
 s9
 s10
 s11
  s12
  s13
  s14
  s15
   s16
   s17
   s23
```

Number of expanded nodes: 19
Number of frontiers: 13 (19 Expanded nodes - 5 nodes without children - 1 goal state node)
Path length: 5 (22.3 using the cost of the edge)
Final path: $s0 \rightarrow s2 \rightarrow s5 \rightarrow s9 \rightarrow s12 \rightarrow s22$

4.2.2 Comparison between BFS and DFS

BFS	DFS
IS optimal, IS complete	NOT optimal, NOT complete
Guaranteed to find the goal	Possible never to find the goal
Memory demanding (all connected vertices are stores)	Consumes less memory
Finds the shortest (optimal) path	Fast in finding the goal
Uses queue (FIFO)	Uses stack (LIFO)

In our application, we need to consider the following:

DFS

- Consumes less memory
- If a robot chooses the correct node, it can be very fast
- A robot may never reach its destination
- Robots can fall into an infinite loop

BFS

- The robot finds the shortest path between vertices in the maze
- Requires a lot of memory
- It can be complex to run

Since the common goal for robots is to map the environment, BFS is more suitable for this use. BFS explores equally in all directions. This is a handy algorithm for common pathfinding and procedural map generation, flow field pathfinding, distance maps and other types of map analysis.

4.3 Dijkstra's Algorithm

Now the length of the roads has been added to the map. We are allowed to drive at the same speed on all roads. Use Dijkstra's algorithm to find the fastest route from city s0 to city s23.
Dijkstra's algorithm (Uniform Cost Search) lets us prioritize which paths to explore. Instead of exploring all paths equally as in BFS, it favours lower-cost paths. This can be helpful in navigation, setting lower costs on roads and higher in forests. This algorithm is used instead of BFS when the moving cost varies (in our case, the vertices have different costs between the nodes)

- Stops when we have the goal, and other nodes have a higher cost
- Instead of expanding the shallowest node, we expand the node with the smallest path cost
- Is optimal and complete
- The parent is not checked
- If an existing node is re-computed and the value is lower, we need to revisit the node

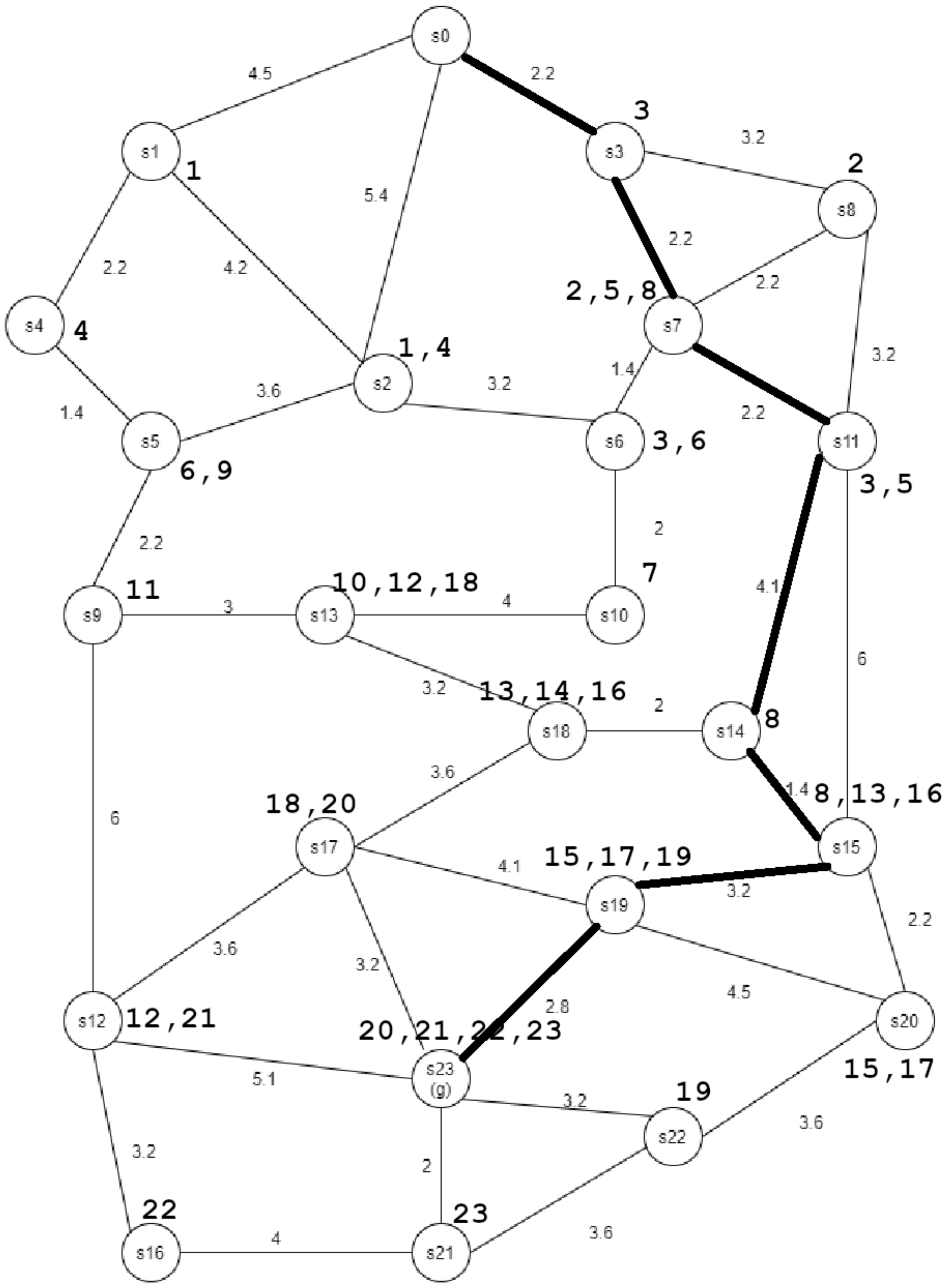


Figure 27: Drawing of DFS while expanding the highest node

In the steps below, we can see the iterative process. The first column is the node itself, the second one its parent and the last one its value. If the node's value was higher than the previous one, the value is not updated for that node and is represented by − in the table. Also, in the first row is the parent node, which we expand in the current step.

Step 1:

s0	s0	0
s1	s0	4.5
s2	s0	5.4
s3	s0	2.2

Step 2:

s3	s0	2.2
s7	s3	4.4
s8	s3	5.4

Step 3:

s7	s3	4.4
s6	s7	5.8
s11	s7	6.6
s8	-	-

Step 4:

s1	s0	4.5
s2	-	-
s4	s1	6.7

Step 5:

s8	s3	5.4
s7	-	-
s11	-	-

Step 6:

s2	s0	5.4
s6	-	-
s5	s2	9
s1	-	-

Step 7:

s6	s7	5.8
s10	s6	7.8
s2	-	-

Step 8:

s11	s7	6.6
s8	-	-
s14	s11	10.7
s15	s11	12.6

Step 9:

s4	s1	6.7
s5	s4	8.1

Step 10:

s10	s6	7.8
s13	s10	11.8

Step 11:

s5	s4	8.1
s9	s5	10.3
s4	-	-

Step 12:

s9	s5	10.3
s13	-	-
s12	s9	16.3

Step 13:

s14	s11	10.7
s15	s14	12.1
s18	s14	12.7

Step 14:

s13	s10	11.6
s18	-	-

Step 15:

s15	s14	12.1
s19	s15	15.3
s20	s15	14.1

Step 16:

s14	s11	10.7
s15	s14	12.1
s18	s14	12.7

Step 17:

s15	s14	12.1
s19	s15	15.3
s20	s15	14.1

Step 18:

s18	s14	12.7
s13	-	-
s17	s18	16.3
s11	-	-

Step 19:

s20	s15	14.1
s19	-	-
s22	s20	17.7

Step 20:

s19	s15	15.3
s23	s19	18.1
s17	-	-
s20	-	-

Step 21:

s17	s18	16.3
s23	-	-
s12	-	-
s19	-	-

Step 22:

s12	s9	16.3
s16	s12	19.5
s17	-	-
s23	-	-

Step 23:

s22	s20	17.7
s21	s22	21.3
s23	-	-

Step 24:

s23	s19	18.1

Number of expanded nodes: 22 (node s16 and s21 is not expanded)

Number of frontiers: 24
Path length: 7 (18.1 using cost of the edge)
Final path: $s0 \rightarrow s3 \rightarrow s7 \rightarrow s11 \rightarrow s14 \rightarrow s15 \rightarrow s19 \rightarrow s23$

4.4 Greedy Best-First Search

For Greedy search, the only thing we consider is the distance from the goal. Therefore the cost of the route (edge) can be omitted entirely.

- Greedy best-first only takes the distance to the goal into account. It ”greedily” tries to get closer to the goal each step without considering any other costs
- Expands the node that seems closest using heuristic (Manhattan distance, Euclidean distance to the goal etc.)
- Heuristic = function that estimates how close a state is to the goal
- NOT optimal
- It takes expensive ways that look good in the future

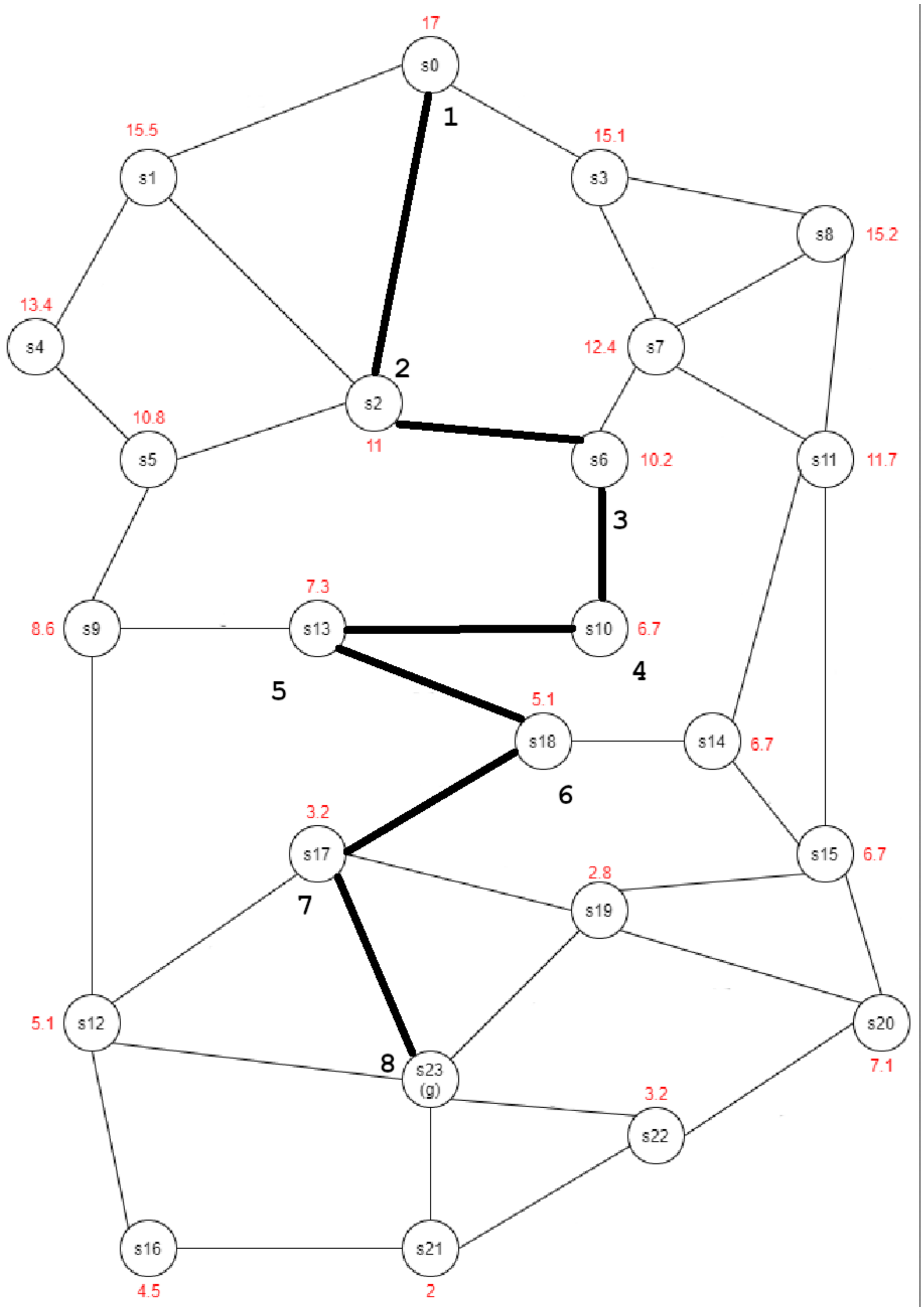


Figure 28: Drawing of DFS while expanding the highest node

The next node is on the right in the iteration process below. We should sort the frontiers after every iteration, but the operation is computationally heavy, and in this graph, the way is straightforward. Therefore sorting is not needed. We need to keep it in mind for more complex graphs or the A* algorithm. To find the final path, we backpropagate to the parents from the goal node in terminal nodes (the parent node is mentioned in the brackets). There are no value updates in this algorithm.

Step 0

- Frontiers: s0(17)
- Terminated Nodes: -

Step 1

- Frontiers: s1(15.5) - s3(15.1) - s2(11)
- Terminated Nodes: s0(-)

Step 2

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s6(10.2)
- Terminated Nodes: s0(-) - s2(s0)

Step 3

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s7(12.4) - s10(6.7)
- Terminated Nodes: s0(-) - s2(s0) - s6(s2)

Step 4

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s7(12.4) – s13(7.3)
- Terminated Nodes: s0(-) - s2(s0) - s6(s2) – s10(s6)

Step 5

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s7(12.4) – s9(8.6) – s18 (5.1)
- Terminated Nodes: s0(-) - s2(s0) - s6(s2) – s10(s6) – s13(s10)

Step 6

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s7(12.4) – s9(8.6) – s14(6.7) – s17(3.2)
- Terminated Nodes s0(-) - s2(s0) - s6(s2) – s10(s6) – s13(s10) – s18(s13)

Step 7

- Frontiers: s1(15.5) - s3(15.1) - s5(10.8) - s7(12.4) – s9(8.6) – s14(6.7) – s12(5.1) – s19(2.8) – s23(0)
- Terminated Nodes: s0(-) - s2(s0) - s6(s2) – s10(s6) – s13(s10) – s18(s13) – s17(s18)

Step 8

- Final node reached
- Terminated Nodes: s0(-) - s2(s0) - s6(s2) – s10(s6) – s13(s10) – s18(s13) – s23(s18)

Number of expanded nodes: 7 (final node is not expanded)
Number of frontiers: 8
Path length: 7 (24.6 using the cost of the edge)
Final path: $s0 \to s2 \to s6 \to s10 \to s13 \to s18 \to s17 \to s23$

4.5 A* search algorithm

A* is a modification of Dijkstra's algorithm optimized for a single destination. Dijkstra's algorithm can find paths to all locations. A* finds the path to one location or the closest of several locations. It prioritizes ways that seem to be leading closer to a goal.

- Combination of Dijkstra's and greedy algorithm
- It prioritizes paths that seem to be leading closer to a goal
- NOT optimal (the heuristic might be misleading sometimes)
- IS optimal only if h(n) is admissible (estimates h must be less than actual cost)
- Heuristic is admissible if $h(n) \leq h*(n)$, where h*(n) is the actual cost of going from n to the nearest goal
- When putting a new frontier to the list, we sort it in the ascending/descending order (using a linked list is the faster implementation, so we do not have to shift every single item in the list)

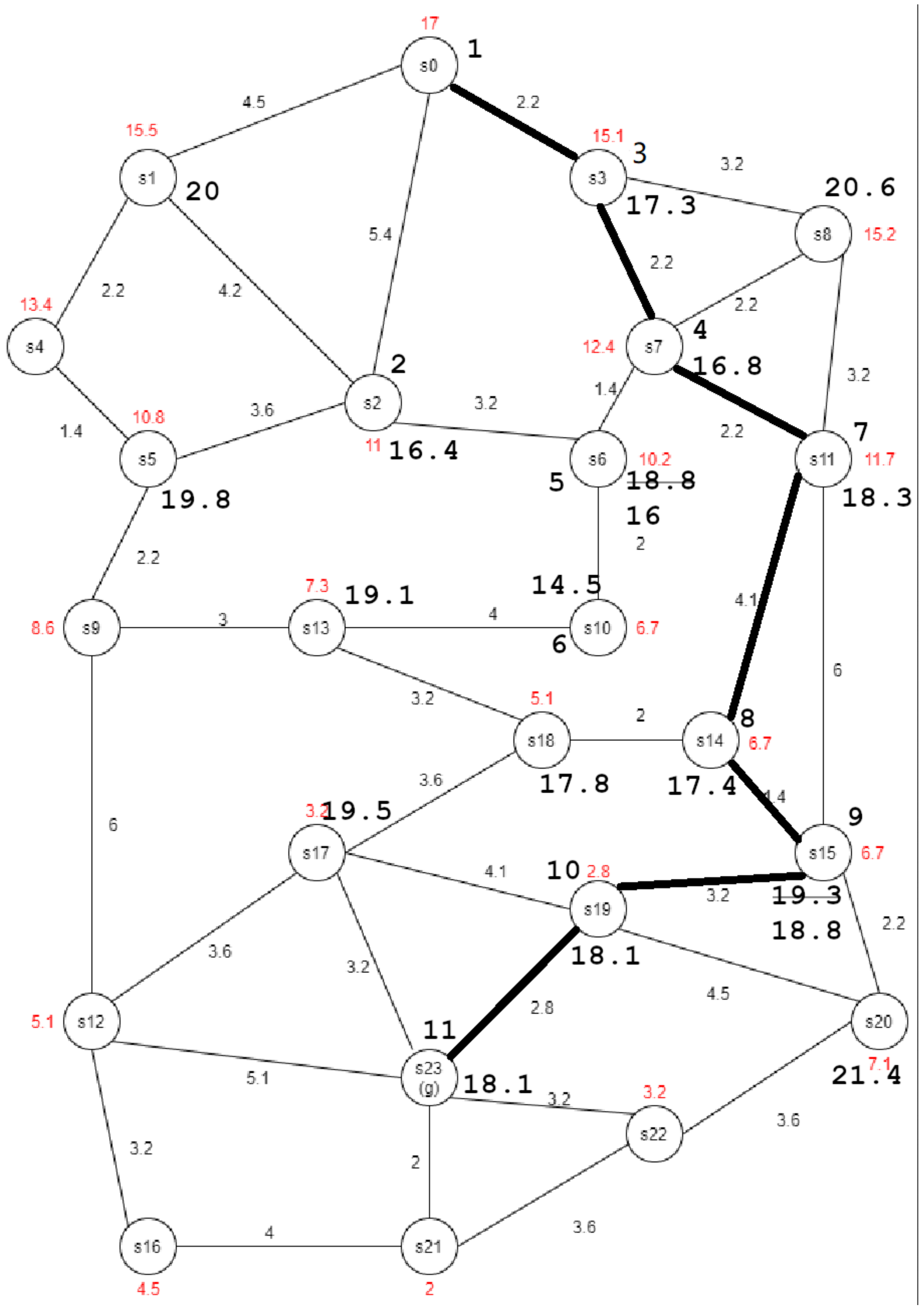


Figure 29: Drawing of DFS while expanding the highest node

We follow the same steps in the iteration process as for the Greedy search. The only difference is in the used heuristic function. For Greedy best-first search, we used only the distance from the goal. In this case, we also consider the cost of the route. The final value is therefore f(n) = h(n) + g(n), where h(n) is the distance to the goal (future measure), and g(n) is the cost of the route.
Therefore, the algorithm avoids using expensive routes that go closer to the goal. It tries to find a route around. For example, it takes the highway, which is a longer route in the distance but faster in total time, than going across

the villages.

The cost of current node can be computed as $f(parent) - h(parent) + g(parent- > current) + h(current)$. We select the node from the OPEN list which has the smallest evaluation function value (g+h). If node n is the goal node, we return success and stop. To find the final path, we backpropagate to the parents from the goal node in terminal nodes (the parent node is mentioned in the brackets). The updated nodes are bold in the iteration steps below.

Step 0

- Frontiers: s0(17)
- Terminated Nodes: -

Step 1

- Frontiers: s2(16.4) - s3(17.8) - s1(20)
- Terminated Nodes: s0(-)

Step 2

- Frontiers: s3(17.8) - s6(18.8) - s5(19.8) - s1(20)
- Terminated Nodes: s0(-) - s2(s0)

Step 3

- Frontiers: s7(16.8) - s6(18.8) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0)

Step 4

- Frontiers: **s6(16) -** s11(18.3) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3)

Step 5

- Frontiers: s10(14.5) - s11(18.3) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7)

Step 6

- Frontiers: s11(18.3) - s13(19.1) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6)

Step 7

- Frontiers: s14(17.4) - s15(19.3) - s13(19.1) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6) - s11(s7)

Step 8

- Frontiers: s18(17.8) - **s15(18.8)** - s13(19.1) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6) - s11(s7) - s14(s11)

Step 9

- Frontiers: s19(18.1) - s13(19.1) - s17(19.5) - s5(19.8) - s1(20) - s8(20.6)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6) - s11(s7) - s14(s11) - s15(s14)

Step 10

- Frontiers: s23(18.1) - s13(19.1) - s17(19.5) - s5(19.8) - s1(20) - s8(20.6) - s20(21.4)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6) - s11(s7) - s14(s11) - s15(s14) - s19(s15)

Step 11

- Final node reached (goal node reached and it is the smallest value at the same time)
- Terminated Nodes: s0(-) - s2(s0) - s3(s0) - s7(s3) - s6(s7) - s10(s6) - s11(s7) - s14(s11) - s15(s14) - s19(s15) - s23(s19)

Number of expanded nodes: 10 (final node is not expanded)
Number of frontiers: 11
Path length: 7 (18.1 using the cost of the edge)
Final path: $s0 \rightarrow s3 \rightarrow s7 \rightarrow s11 \rightarrow s14 \rightarrow s15 \rightarrow s19 \rightarrow s23$

The advantage of using A* instead of Dijkstra's algorithm is that A* is faster. It prioritizes paths that are leading us closer to the goal. In contrast, Dijkstra is just trying to expand the shortest route, which can lead to completely opposite direction from our goal node.

4.6 Comparison of Greedy and A* algorithm

4.6.1 Greedy best-first search

Advantages

- Expand the nodes that seem to be closest to the goal (takes expensive ways that look good in the future)
- Keeps only frontiers in memory

Disadvantages

- Does not look back, and once a wrong decision is made, it cannot be changed
- Only looks into future, does not use past knowledge (from that called greedy)
- Not complete and not optimal (meaning that we risk the path does not bring us to the goal = not complete, and also that the found path is not the shortest one = not optimal)

4.6.2 A* algorithm

Advantages

- Uses an admissible heuristic function (therefore * in the name)
- Is complete as IS optimal (if and only if using admissible heuristic)
- Combines past experiences as well as the heuristic (basically Greedy and Dijkstra combined)

Disadvantages

- Needs to keep all nodes in memory, not only the frontiers, as the Greedy best-first
- Impractical when the space search is huge

4.6.3 Which algorithm to use for aerial robots and why?

A* delivers the best results. It is optimal, complete, and able to solve complex tasks. It prioritizes paths that seem to be leading closer to a goal. The memory issue can be problematic if we want to discover a huge map, which is not a problem in ASTA. Therefore this algorithm would deliver the best results.

4.7 Upgrade the Greedy best-first search from 2D to 3D

After adapting the *greedy_2d* to account for the third dimension, the algorithm was run on the sample maze. The result is provided in the Figure 30 where the path is successfully found.

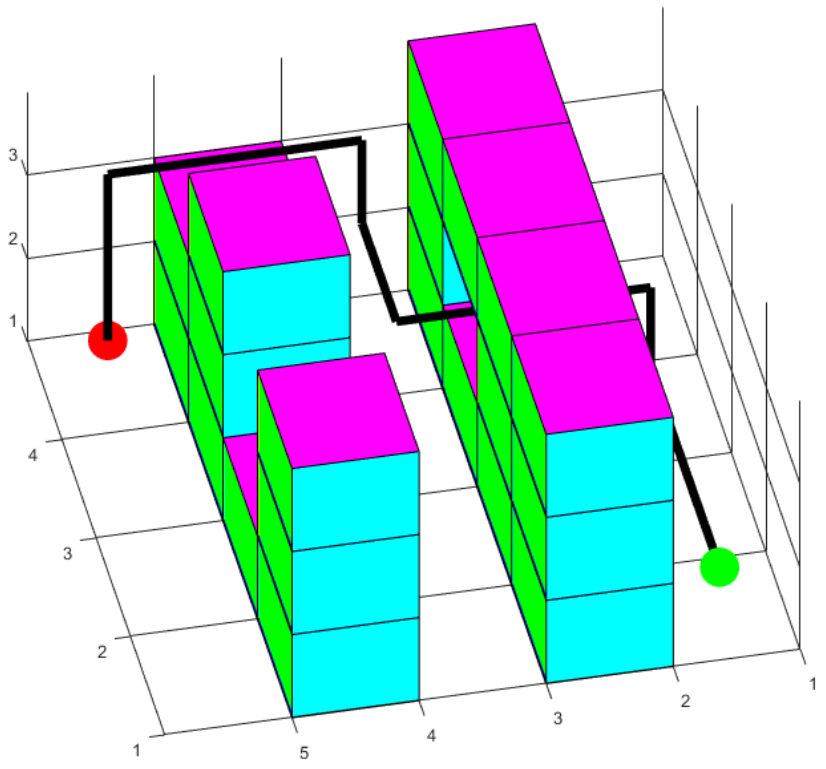


Figure 30: A greedy best-first search algorithm runs on the maze

.

4.8 Modify the implementation of the Greedy best-first search algorithm into an A* search algorithm

The cost function of the A* algorithm is different, i.e. incorporating more information than the distance to the end goal. Specifically, $g(n)$ - the cost between nodes. This could be heuristics to integrate how expensive it is to travel from one node to the other along the path. In the exercise toy problem, we do not have any heuristics that would play a major role when deciding on that. However, we can say that we penalize nodes that are far from the start position, meaning covering the shortest distance without wandering away too far. Final cost is $f(n) = h(n) + g(n)$ where $h(n)$ is the distance to the goal, and $g(n)$ is the cost between nodes. The Figure 31 shows the path generated by this version. It looks different from the previous one; however, the steps required to reach the goal are still the same compared to the greedy approach.

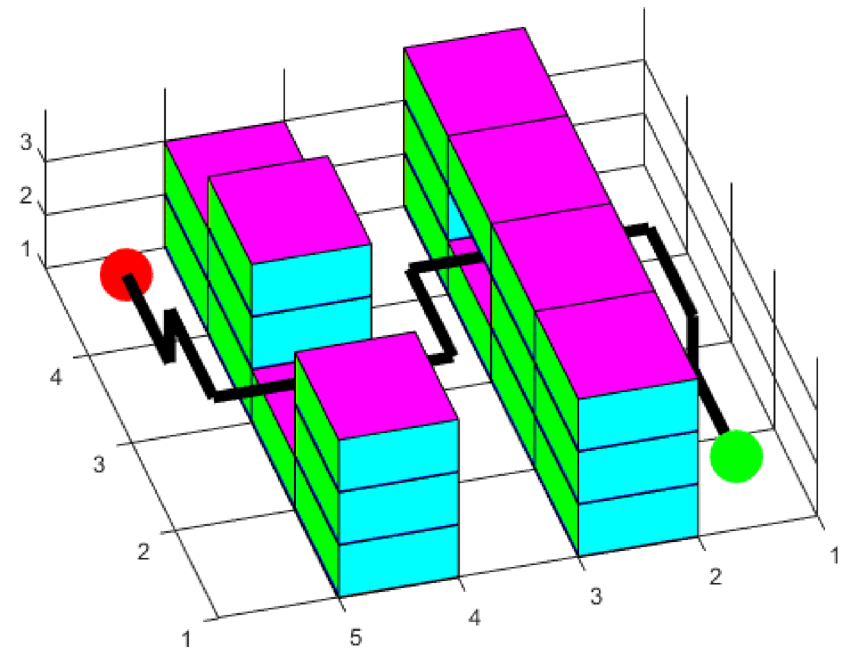


Figure 31: A* search algorithm runs on the maze

.

After implementing the suggested variant of A*, an experiment was run to investigate how new addition affects the algorithm's running time compared to BFS. The timing script is provided below. The results show that *astar* is almost twice as slow as *greedy* one. The comparison is not quite fair because the maze is so tiny, but we can generally reason that on each child node, we compute distance twice instead of once. A square root is an expensive operation, and the maze is small, so the difference is reasonable.

```
tg = [];
ta = [];
for i = 1:100
tic
route = greedy_3d(map, start, end_);
tg = [tg toc];
tic
route = astar_3d(map, start, end_);
ta = [ta toc];
end
mean(tg)
mean(ta)
```

If algorithm would be implemented with cost function $f(n) = g(n)$ where $g(n)$ - is the cost between nodes. This would be the standard Dijkstra algorithm.

5 Part V: Planning

5.1 Exercise 5.1: quintic splines

Having polynomial:

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

and six constraints with six unknown constants:

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

we can solve it by forming matrix C (like below) & vector b, and multiply b by inverse A^{-1}

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 2 & 6 & 12 & 20 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, a = A^{-1}b$$

and the constant in this case is:

$$a = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 10.0000 \\ -15.0000 \\ 6.0000 \end{bmatrix}$$

5.2 Exercise 5.2: B-splines

The notation in the slides and lecture is not well explained; thus we'll be following [2] definitions. Polynomial is defined as:

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

where basis is:

$$B_{k,1}(u) = \begin{cases} 1 & \text{if } u_k \leq u \leq u_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{k,\text{ d}}(u) = \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u), d > 1$$

The task defines $u_i = i$ which implies that knot vector is uniformly increasing one by one and has length of $m = d + n + 1 = 2 + 2 = 4$, where d - is order and $n + 1$ gives number of points which is 2 in this case. Thus knot vector is: $[0, 1, 2, 3]$. Figure 32 shows the recursion tree of Cox-deboor equations. The edges are marked with the coefficient computed, for example:

$$B_{1,2}(u) = \frac{u - u_1}{u_2 - u_1} B_{1,1}(u) + \frac{u_3 - u}{u_3 - u_2} B_{2,1}(u).$$

Knowing that $u_1 = 1; u_2 = 2; u_3 = 3$ we get coefficients:

$$B_{1,2}(u) = (u-1)B_{1,1} + (3-u)B_{2,1}$$

and this is done for all the other edges. In the end, we have a linear polynomial defined between the two points:

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

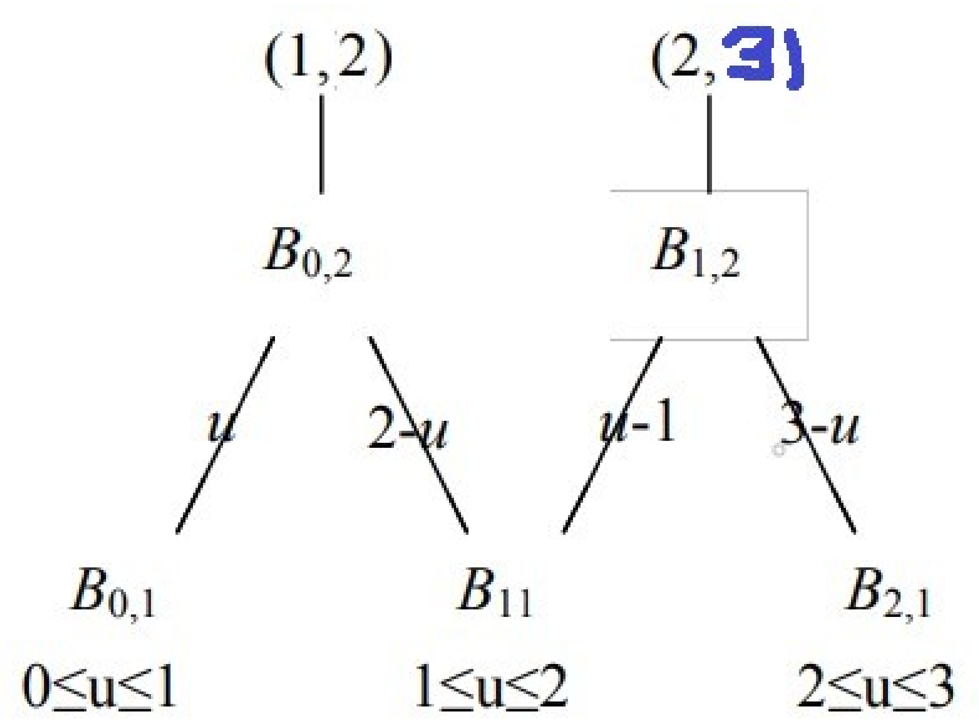


Figure 32: B-spline recursion tree. Modified picture from [2].

5.3 Exercise 5.3: computation of a DCM

According to [3] Equation 31 by Z-X-Y Euler angles convention:

$$R_z(\theta_z)\, R_x(\theta_x)\, R_y(\theta_y) = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{bmatrix}$$

we can factor out the angles by:

$$\theta_x = \operatorname{asin}(r_{21}), \quad \theta_z = \operatorname{atan2}(-r_{01}, r_{11}), \quad \theta_y = \operatorname{atan2}(-r_{20}, r_{22})$$

these correspond to rotation around x, z, y (ϕ, ψ, θ) or roll, yaw, pitch.
Authors of the paper derive R_B (in Z-X-Y Euler angles convention):

$$R_B = \begin{bmatrix} x_B & y_B & z_B \end{bmatrix}, y_B = \frac{z_B \times x_C}{\|z_B \times x_C\|}, x_C = \begin{bmatrix} \cos\psi \\ \sin\psi \\ 0 \end{bmatrix}, x_B = y_B \times z_B$$

$$R_B = \begin{bmatrix} x_{B_0} & y_{B_0} & z_{B_0} \\ x_{B_1} & y_{B_1} & z_{B_1} \\ x_{B_2} & y_{B_2} & z_{B_2} \end{bmatrix}$$

By employing the factorisation above, we can recover Euler angles.

$$\phi = \operatorname{asin}\left(r_{21}\right), \psi = \operatorname{atan2}\left(-r_{01}, r_{11}\right), \theta = \operatorname{atan2}\left(-r_{20}, r_{22}\right)$$

where $r_{21} = y_{B_2}$, $r_{01} = y_{B_0}$, $r_{20} = x_{B_2}$, etc.

and reconstructing R_B by the convention we are following in the class by simply putting these values in:

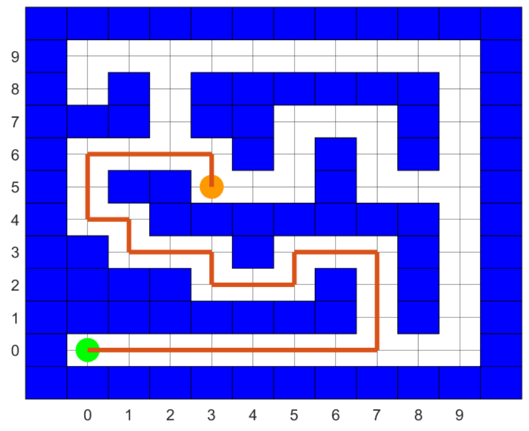
$$R_B = R_z(\psi)R_y(\theta)R_x(\phi)$$

which indeed follows our Z-Y-X representation.

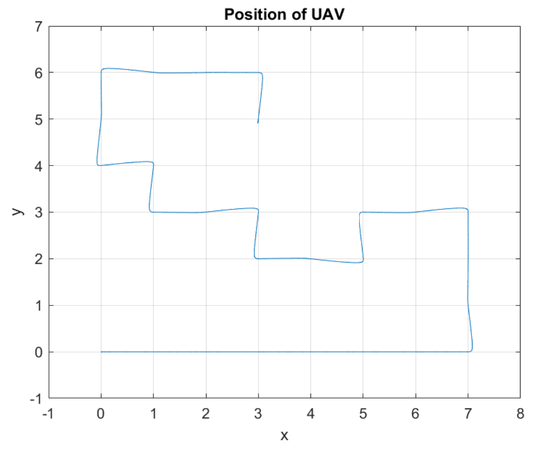
6 Part VI: Simulation

6.1 Navigating 2D Maze

We implemented the algorithms based on the greedy search from exercise 4.7. In the 2D maze, we navigated from the starting point (0,0,0) to (3,5,1) and in the 3D maze, we navigated from corner (0,0,0) to the opposite corner. After running the algorithm inside the simulator, we plotted the path of the drone. The code for this part can be found in 8.4.



(a) DFS for 2D Maze



(b) Simulated Drone Path

Figure 33: 2D Maze Search

6.2 Re-implementing the position controller

We replaced the position-velocity-acceleration controller with our previously developed PID controller, with one key difference; z is now also part of the input. For the previous controller we fed in 0 as z, using only x and y as the inputs, however, that was a result of z already being accounted for in the attitude controller. Now, we merge z into position and thus need to have a non-zero z reference as well. Without the saturation block, we get a system that is unstable for large reference inputs, see figure 35. Therefore we introduce a saturation block, which limits our response to a smaller value in order to ensure stability regardless of the initial error. Another key difference between our current system and the one outlined in part 3 is that we now need to transpose the rotation matrix. This is because while we get the measurements from the inertial frame, but apply the changes to the body frame - the rotation matrix given is the rotation from the body to the inertial. The resulting system can be found below:

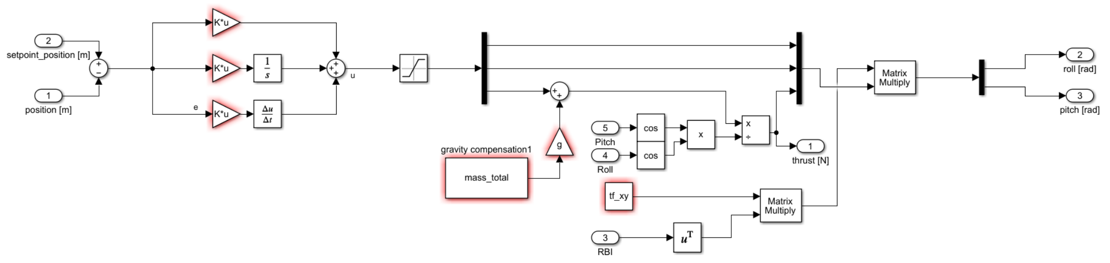
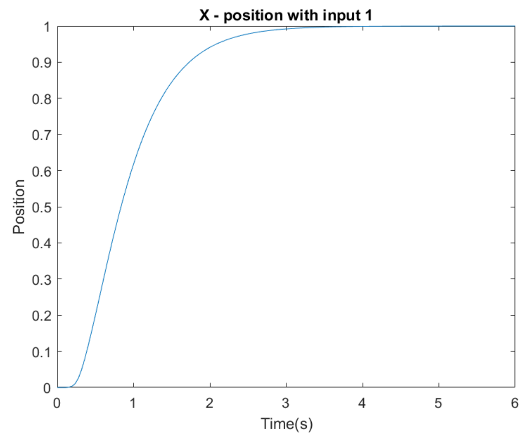


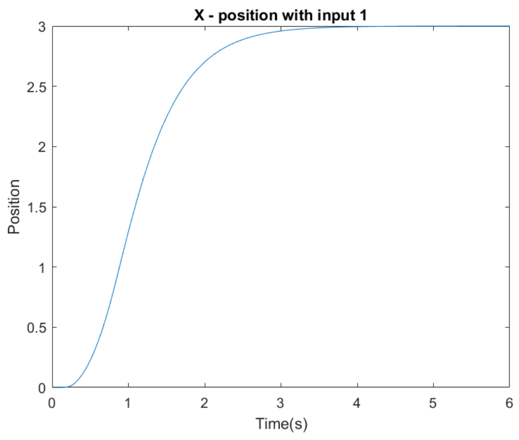
Figure 34: Adapted position controller

All values and matrices can be found in the appendix 4 - note however that the position gains are the final ones from 6.3, and not the initial ones. It is also worth noting that z and yaw have been tuned separately to x,y roll and pitch, as it was found that the system performs far better when it reacts less violently to changes in these parameters.

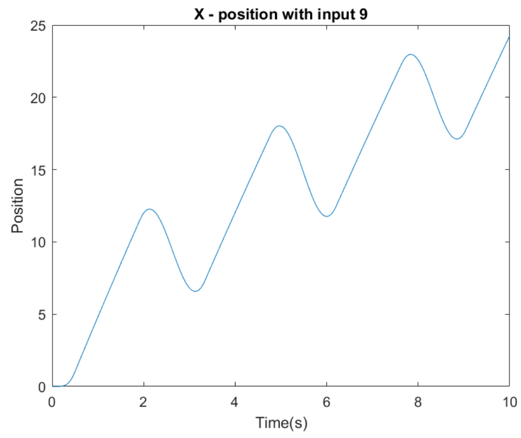
The tuning was done with a baseline of the values found in part 3, however, since the system is slightly different these were not the optimal values. Instead, by applying a step response of magnitude 5 - being roughly in the middle of the corridor, we tuned the system with trial and error until a satisfactory set of values were reached. We could have tried to employ Ziegler-Nichols, however, because of the walls of the maze the marginal stability would cause the drone to crash, thus rendering it impossible to properly tune. Thus, we instead start with a low K_d, increasing K_p until the response is fast enough, and then increasing K_d until we have almost no overshoot. Afterwards some fine-tuning was done through small increases and decreases in both values. The result is shown in the graphs below.



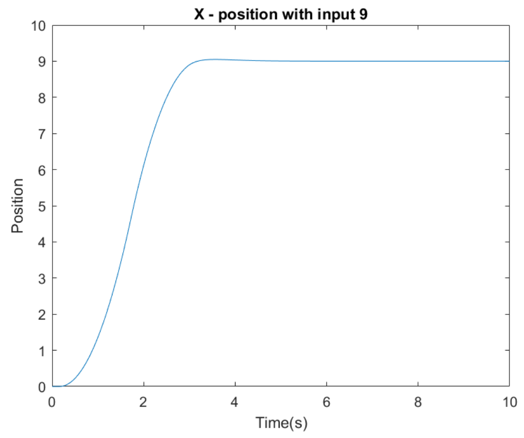
(a) Step response of 1.



(b) Step response of 3.



(c) Step response of 9.



(d) step response with output limited to 5.

Figure 35

The rise time is around 2 seconds, with a very low settling time due to the minimal overshoot accompanying a solution close to critical damping. It should however be noted that these were not the final values for the full system, for a reason that will be explained in the following subsection.

6.3 Re-implementing the attitude controller

Our implementation for the attitude controller is actually far more simple than it was before, as the manipulation regarding altitude has been moved over to the position controller. Thus, it merely requires an error, fed into the PID (or PD rather, as our K_I is zero for both controllers) and then multiplied with the respective inertia.

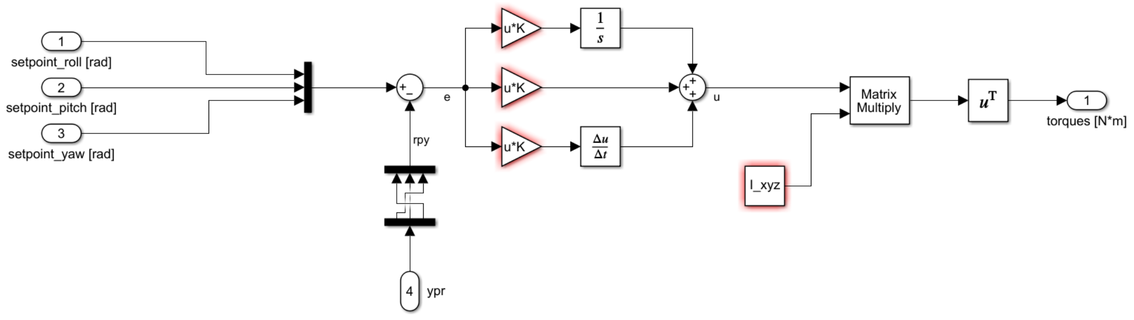
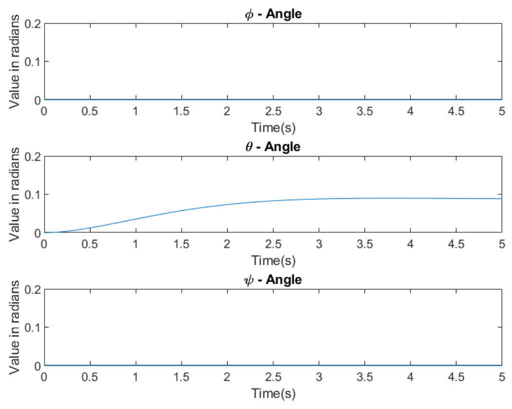
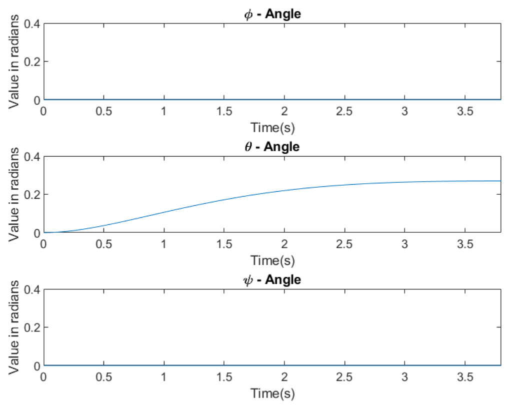


Figure 36: Attitude Controller

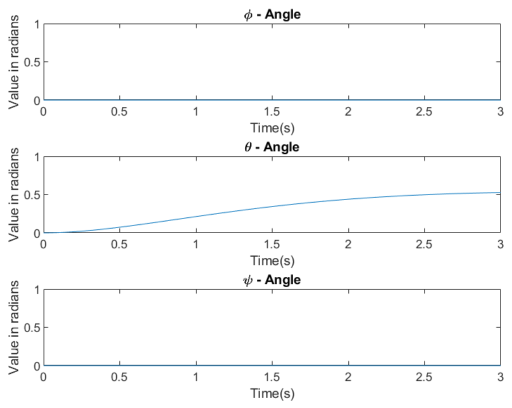
As these are cascaded controllers, the innermost controller should be tuning prior to the outermost one - as the outer controller sees the inner one as a part of its plant. Moreover, as it passes a reference value to the inner controller, tuning the outer controller first will have ramifications for the inner controller, meaning that it will be more difficult to find optimal values. In our case, that means that the attitude controller should be tuned before the position, as was the case in part 3, although omitted in the report. Here, however, this entails re-tuning the position controller afterwards for optimising performance. First, the attitude controller is tuned using the step response and a trial and error methodology. For this controller, this was even more forced upon us than with the position controller, as after only a few seconds the drone would crash into the wall. Thus, the task became to tune it such that it would have reached a stable value within the few seconds prior to the crash. The result of this tuning can be found below.



(a) Step response of 5 degrees

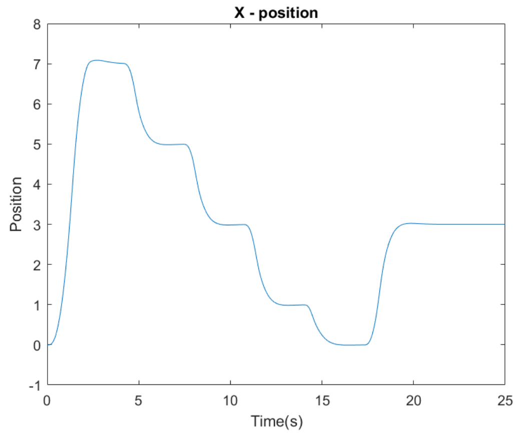


(b) Step response of 15 degrees

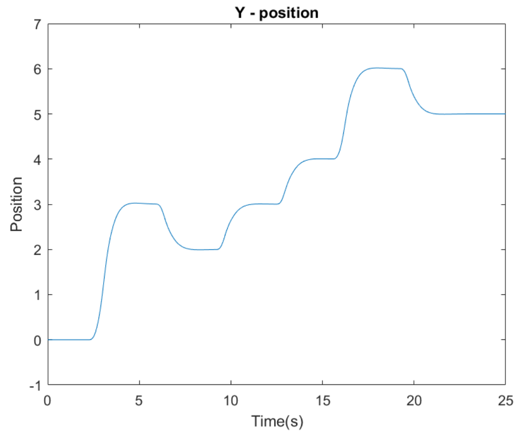


(c) Step response of 30 degrees

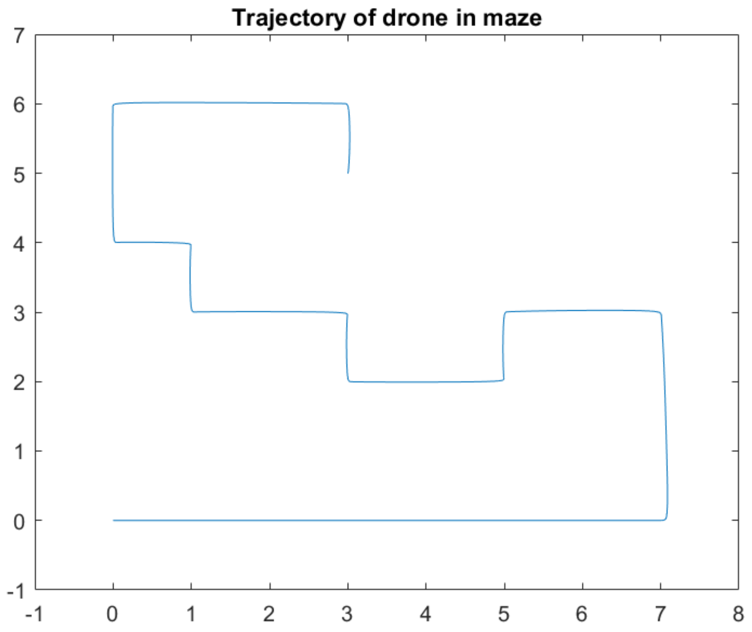
The resulting rise time is higher than for our position controller, and in fact, could be argued is slightly too high - meaning we should try to re-tune with a far larger K_p. However, as will be shown in a moment the result produces very stable movement, with a satisfactory speed for our purposes and therefore should not be a problem. In fact, larger values could push the non-linear system towards instability, so thus the smaller values are preferable. Afterwards, the two controllers have plugged into each other again and using a step response for x,y, and z values, the position controller was re-tuned to improve its performance when solving the maze. The resulting values can be found in the appendix 4. This tuning was then tested on the maze from6.1, which yielded the following result:



(a) X-position while going through the maze



(b) Y-position while going through the maze



(c) The trajectory of the drone through the maze

We see that although our implementation is slightly slower than the default one - roughly 2 seconds - it is highly accurate, with only noticeable overshoot for the larger input values - and even then it quickly recovers.

6.4 Aggressively navigating a 2D Maze

To navigate below 5s without colliding the walls, we needed to set the corridors where the drone must fly at what time (time is defined using *corridors.times*). Then we used the following package to compute the best trajectory for the drone traj-gen-matlab.

Firstly, we needed to correctly set up the Simulink environment to start the simulation.

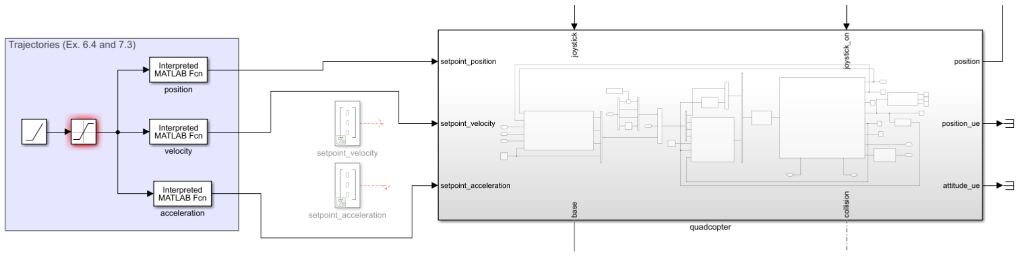


Figure 39: Connected blocks in the Simulink

After the simulation was ready to run, we tuned the parameters as follows.

```
order = 7;
corridors.times = [0 1.15 1.84 2.99];
corridors.x_lower = [-0.5 3.5 7.65 8.6];
corridors.x_upper = [0.5 4.5 7.85 9.25];
corridors.y_lower = [-0.5 -0.6 0.7 6.85];
corridors.y_upper = [0.5 -0.48 0.9  7.25];
corridors.z_lower = [0 0 0 0];
corridors.z_upper = [2 2 2 2];
```

This resulted in the following path of the drone, where the grey rectangles are the corridors that the drone needs to fly through.

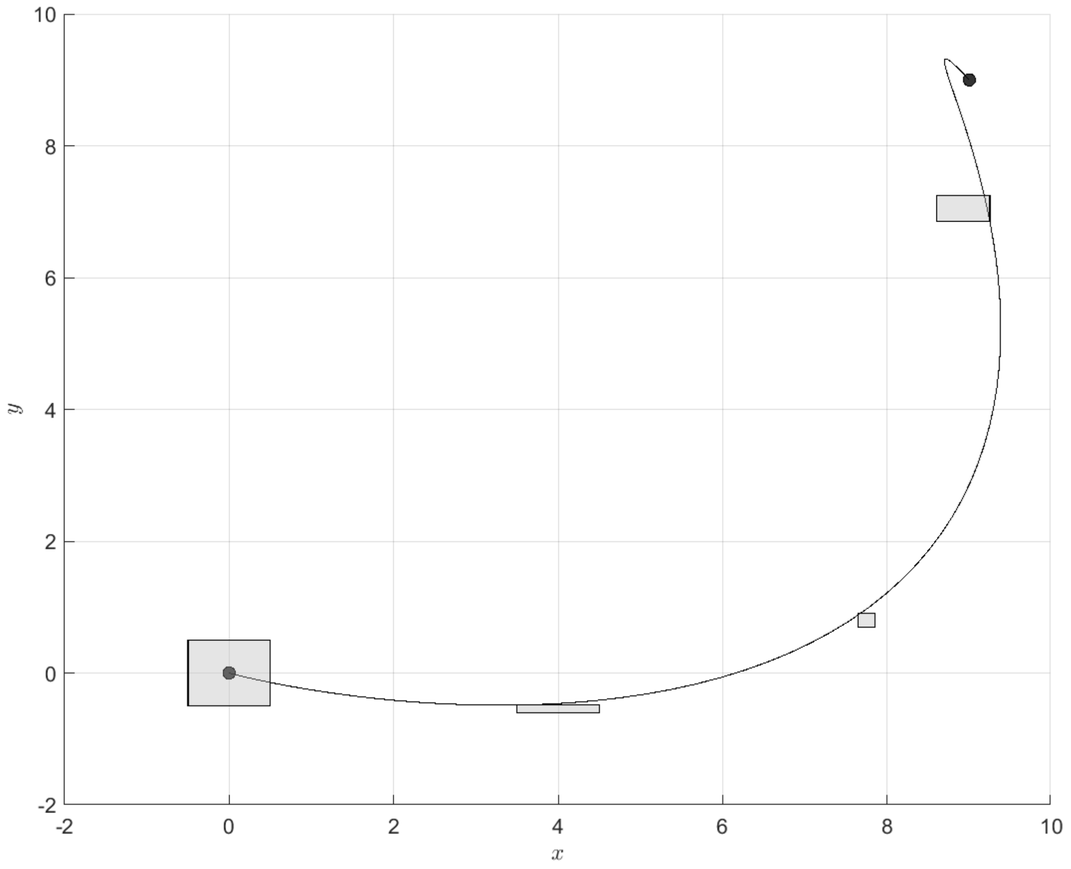


Figure 40: The drone's trajectory

For this path, we got also the graph of derivatives, which come along with the package.

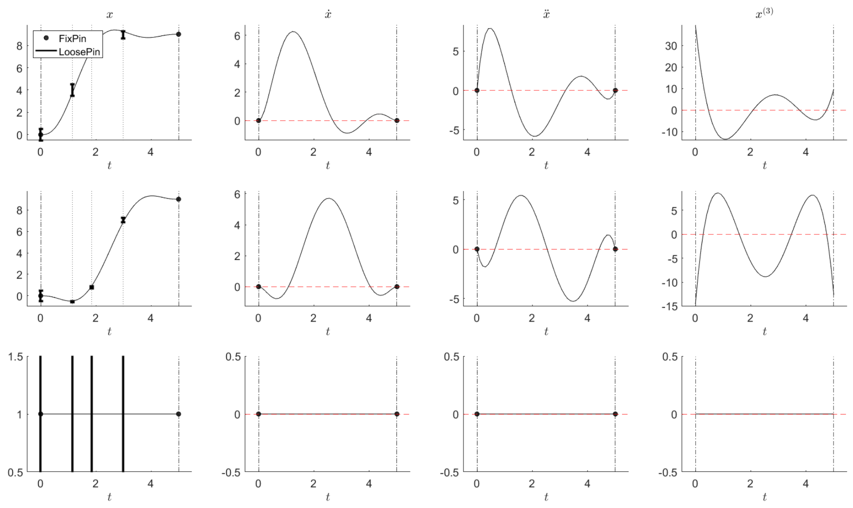


Figure 41: Position, velocity and acceleration of the drone path

The drone reached the position in 2.99s and took about 2s to reach the stable position. A Default PID controller was used for this task.
The video of the simulation could be seen on this YouTube link.

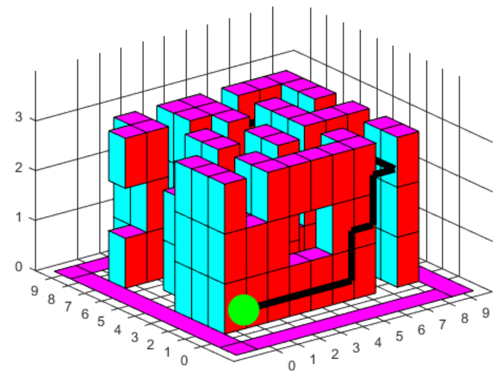
7 Part VII: Demonstration

7.1 Navigating a 3D maze

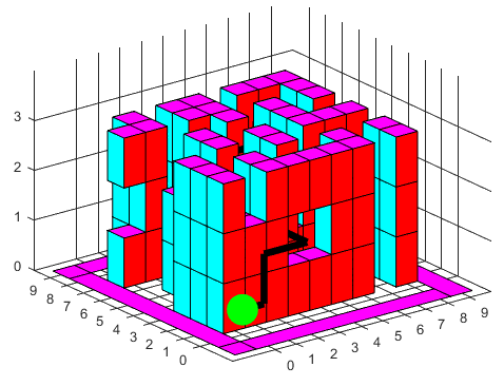
Placing the drone in the cage we see the world coordinate from the rostopic crazyflie/pose. Comparing this to the map coordinate gives us the offset, and flying the diagonal of the cage gives us both the x and y scale. We, however, have no markers for the z map coordinates so this was calibrated for offset as the ground was assumed to be 0, and for scale, we calibrated with a ruler.

x offset	0.16
y offset	0.42
z offset	0.08
x scale	0.65
y scale	0.55
z scale	0.75

Table 1: Offset and scaling for 3d maze.



(a) Greedy Search for 3D Maze



(b) A* for 3D Maze

Figure 42: 3D Maze Search

Using A* we get the route where we add 20cm everywhere in z to have a hovering height of that. We also added a final point identical to the last route point but with -1m in z to make the drone land. Using the onboard controller we see the quite unsteady performance, see figure 43.

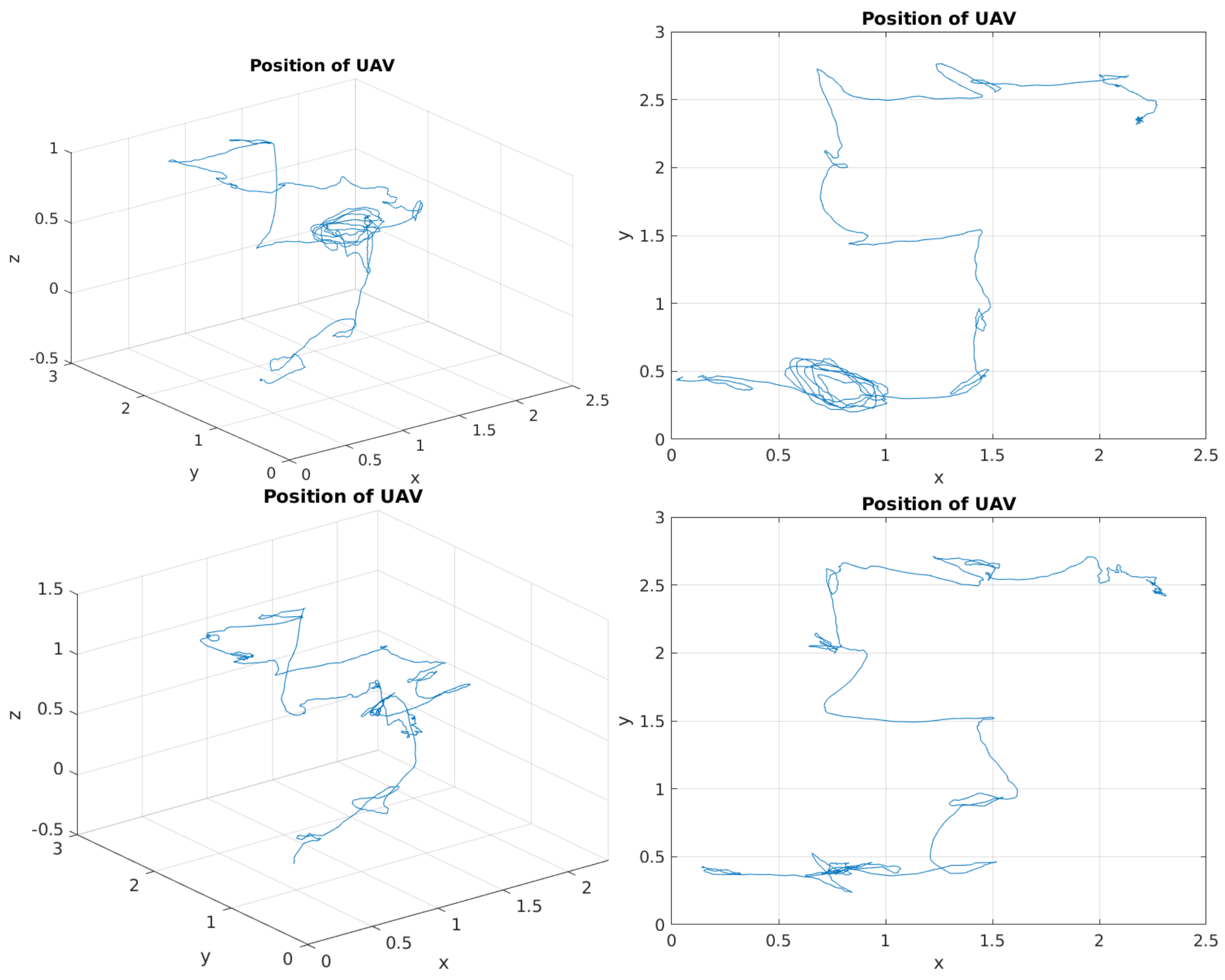


Figure 43: Rows: 2 attempts at flying the route for the 3d maze with publishing position references to the onboard position controller.

We can improve this performance by adding in an outer controller as with the rest of the system, thus we now implement the position controller as well before re-running the maze a couple of times. It is worth noting however that implementing the position controller requires the use of the PWM mapping found in section 7.2. See figure 44 for the results.

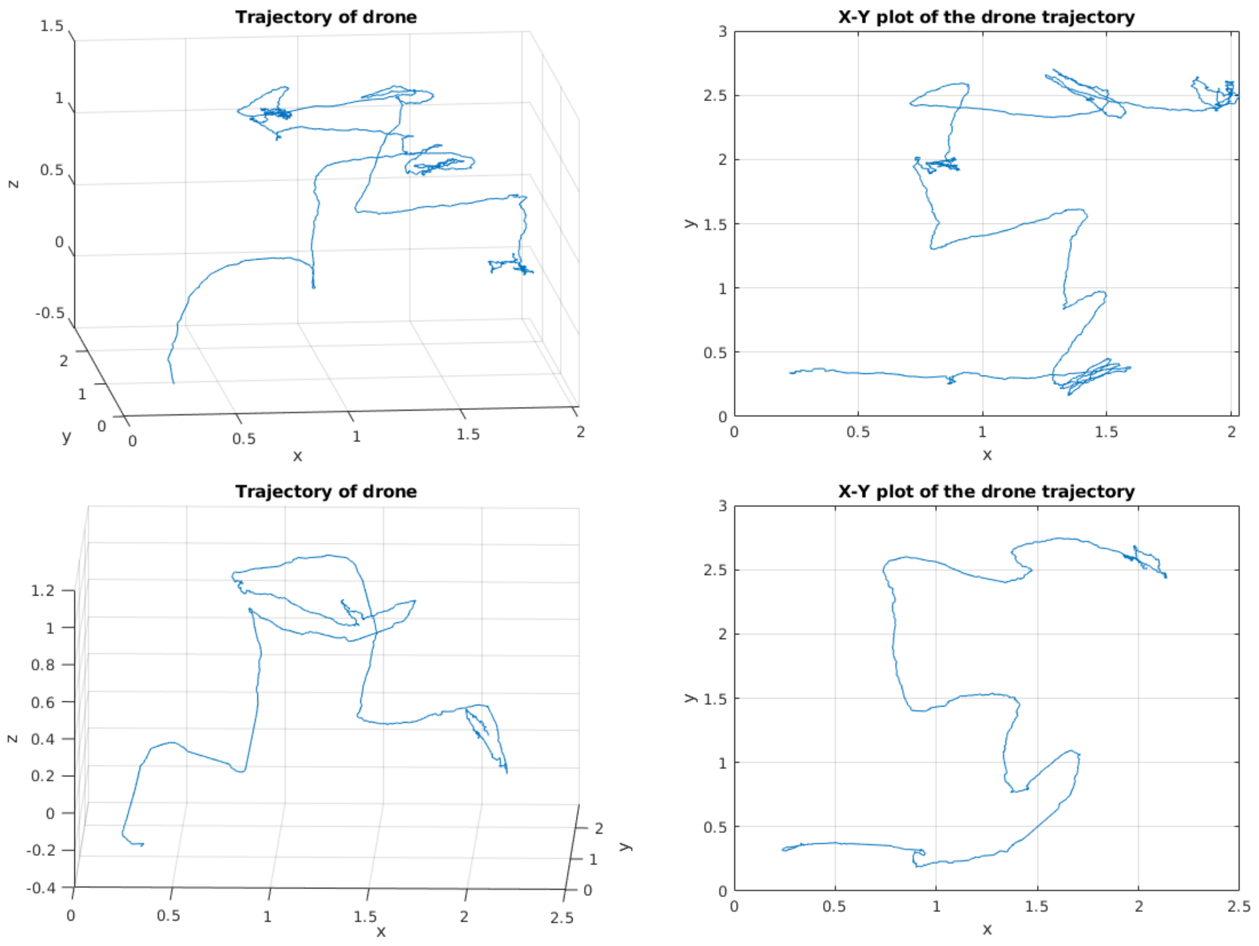


Figure 44: Rows: 2 attempts at flying the route for the 3d maze with sending attitude references to the onboard attitude controller.

With our own position controller, we see improved settling time, which reduces the swirling around set points.

7.2 Porting the position controller to the real system

As explained per the exercise description Crazyflie drone takes in roll and pitch commands as angles but the yaw must be passed as yaw rate instead of an angle and the thrust command is in PWM command i.e. integer values from 0 to 60 000. In order to achieve PWM mapping from the physical thrust parameter to this format, we can collect some data from a real drone and fit a line onto it. This was done in the following way:

- Control the drone with a joystick
- While flying try to generate thrust only along the Z-axis - no pitch or roll involved
- Record IMU acceleration data along Z-axis together with PWM commands
- Raw IMU data comes as Gs, thus we need to convert it to force by multiplying with mass and gravitational acceleration: $f_z = mgG$.
- Having data points relating to PWM and physical thrust do a linear regression to identify the relationship between them

Our mass was found for the system used in section7.1, where we used various data sheets to calculate the total mass of the drone. Although this mass is not 100% the same as for the optitrack tasks - given that we do not use the same board, and have the tracking balls to take into consideration - they are assumed to be comparable enough to be useful in both situations. Data and regression line is visualised in Figure 45 (x-axis is thrust in Newtons while y - PWM commands sent to drone). This was cross-validated with controller output in the simulation first, Figure 46 shows requested thrusts over time. The values are bounded around [6; 8] Newtons and this matches the data collected in the experiment (Figure 45 x axis). This will be further validated on the real drone. Parameters can be put in a function and integrated into the *Simulink* model to convert controller-generated thrust to the appropriate integer for sending it over to the drone:

```
function y = thrust_to_PWM(u)
    c0 = 20879;
    c1 = 31130;
    y = c0 + c1*u;
```

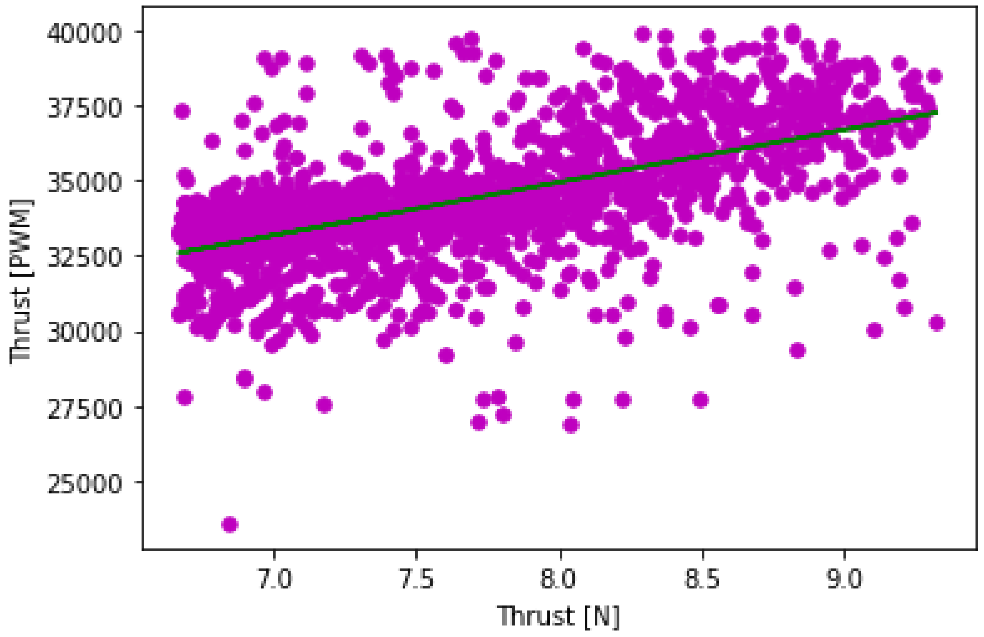


Figure 45: PWM linear regression.

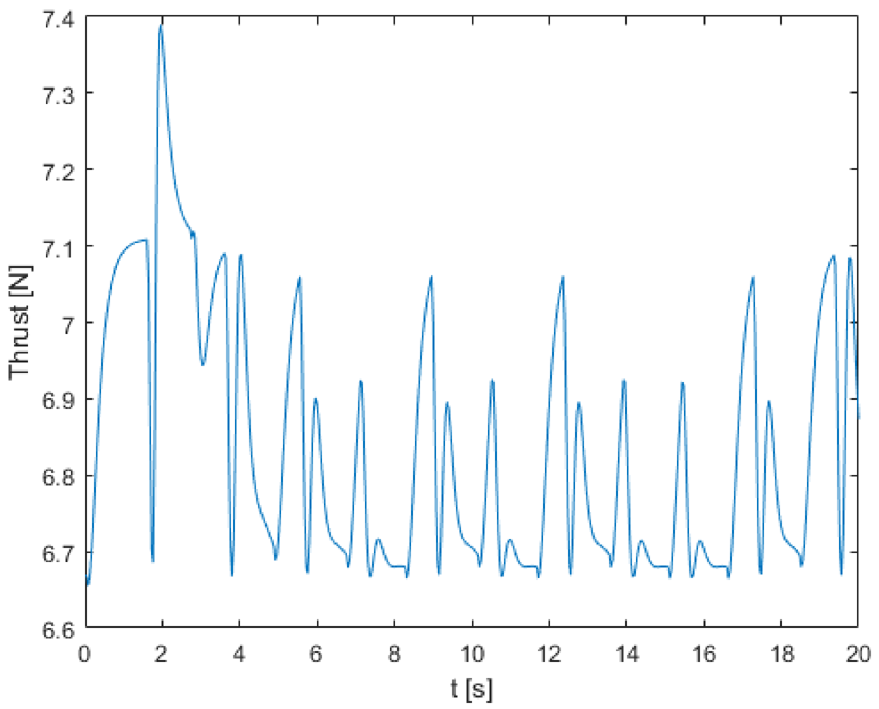


Figure 46: Thrust output from our position controller.

However, upon testing this solution we found that it did not provide the stable result we wanted. We did however find experimentally that the hovering PWM value was around 44000 - later slightly changed to be 44150 - thus we set this to be our constant and calculated our slope to be:

$$c1 = 44000 - mg * C \tag{10a}$$

$$y = 44000 + c1 * u \tag{10b}$$

Where u is the input and C is a constant value. We then tuned C such that the observed resulting mapping fit our expectations. Doing so led to a final mapping of

$$C = 37000 \tag{11a}$$

$$c0 = 44150 \tag{11b}$$

$$c1 = 30216 \tag{11c}$$

$$y = c0 + c1 * u \tag{11d}$$

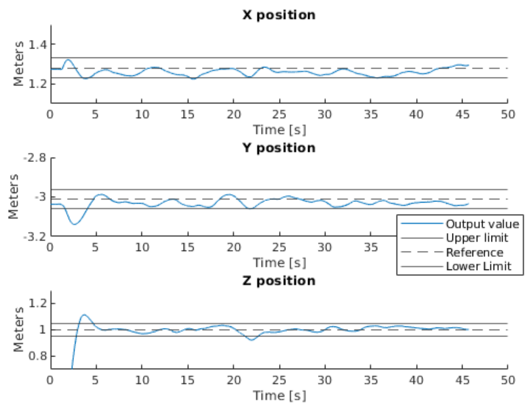
Notice that the only real change gained from this approach ended up being the shift in constant value to assure hovering. With these values, we also needed to re-tune our position controller. To do so we first started by changing the mass in the simulation block, and re-tuned it in the same manner explained in exercise 6. This gave an adequate starting point to improve upon, as it meant the drone would not be fully unstable immediately. We then used these values on the real drone and tuned its response to having less overshoot, faster response time and as little steady-state error as possible. The resulting values for the controller were:

$$K_p = \begin{bmatrix} 0.42 & 0 & 0 \\ 0 & 0.40 & 0 \\ 0 & 0 & 0.265 \end{bmatrix} \tag{12a}$$

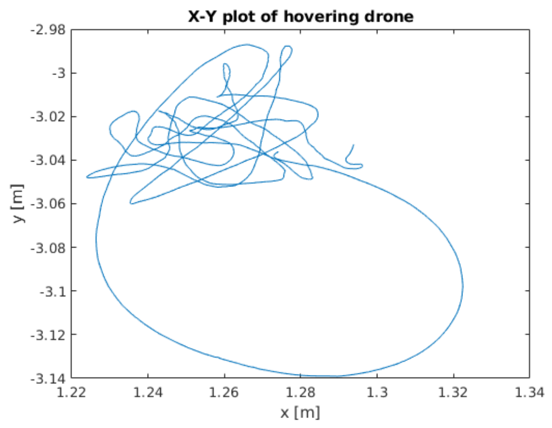
$$K_d = \begin{bmatrix} 0.28 & 0 & 0 \\ 0 & 0.29 & 0 \\ 0 & 0 & 0.18 \end{bmatrix} \tag{12b}$$

$$K_i = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.01 \end{bmatrix} \tag{12c}$$

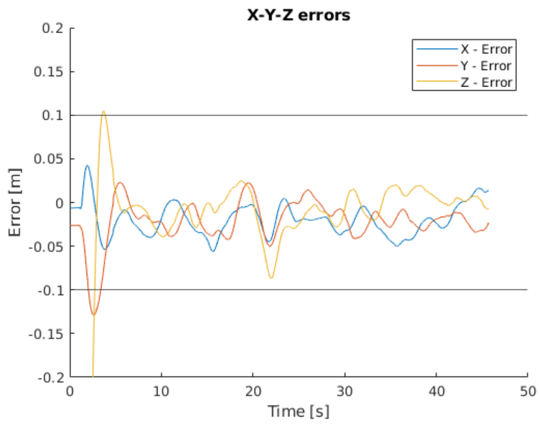
Note that we have a PID only for the altitude, whereas the position in the X-Y plane is a PD controller. This was done due to the fact that we had a large steady-state error in the z-direction, which the integrator is able to reduce quite significantly. However, this was not necessary for either x or y. So, our first test for this system is to make it hover 1m above starting position for at least 10 seconds.



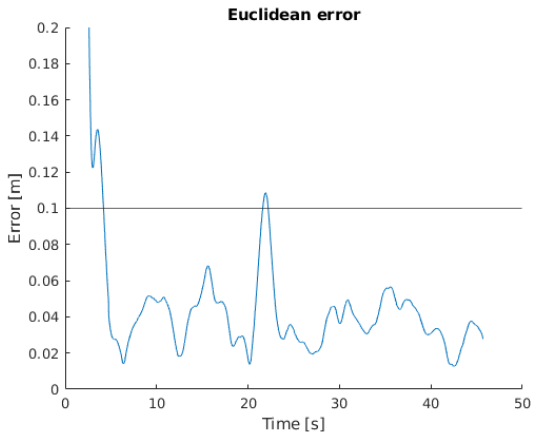
(a) x,y and z positions compared to references. The bounds signify upper and lower limits for the route to advance to the next point



(b) X-Y plot for the drone, notice the scale of the axis' are quite small



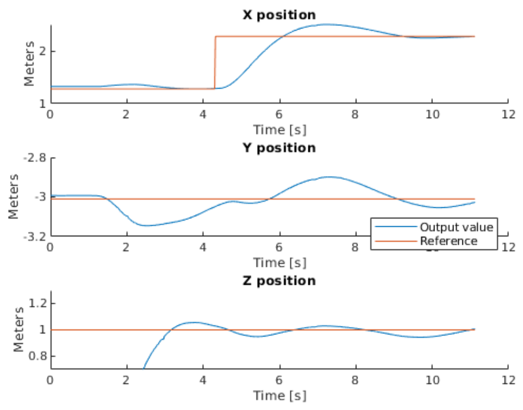
(c) Errors in the x,y and z position, as compared to the upper and lower boundaries of 10cm away from the reference position



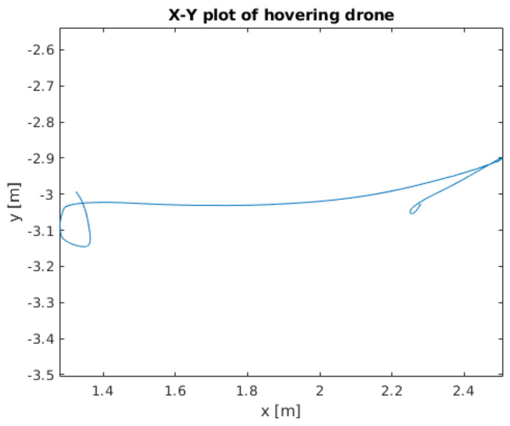
(d) The Euclidean error while hovering, also compared to the max distance of 10cm

We see that while the drone is not completely still, it is quite steady, and well within the bounds of 10cm on any given axis after an initial overshoot has been corrected for. In fact, even the euclidean error is only above 10cm for a tiny fraction of the time the drone is flying. Meaning overall the results are quite satisfactory.

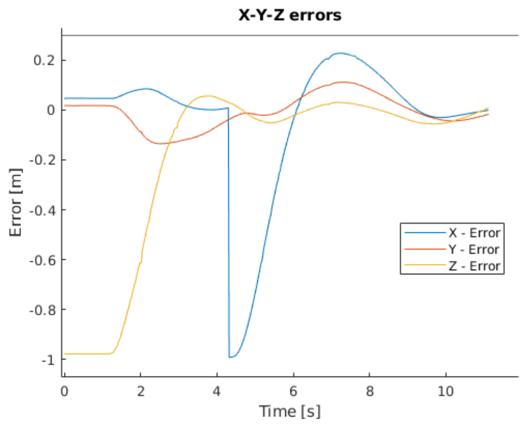
Thus, we move on to the second trial, moving one meter along the x-axis with less than 30cm overshoot.



(a) x,y and z positions compared to references.



(b) X-Y plot for the drone



(c) Errors in the x,y and z positions, compared to an overshoot of 30 cm, marked as 0.3

We see here that the drone does indeed overshoot, however, with the current controller values it only overshoots by around 20cm, which again is well within the target of 30cm. Moreover, errors for both z and y are approximately the same as seen for the hovering case.

7.3 Aggressively navigating through hoops

We used our experiences from the previous exercise to create the desired path. We set waypoints and created corridors in front and behind each hoop. Using this approach, we can be sure that the drone is flying straight through the hoop, which decreases the chances of hitting it. The position of the hoops are given as rostopics, therefore we can read them through internal Matlab functions. One potential issue we found with this approach however, is that since the drone does not know the orientation of the hoops, it could try to go through them during path generation. As such, we decided to add an extra corridor to align it properly with the first hoop. Moreover, to ensure accuracy we also added an upper and lower bound to the z position of our corridors. While this might not have been necessary considering the waypoints are exactly where we want to go, this could fix the edge-case where it flies to the hoop on ground level and then tries to rise up through the bottom part of the hoop. Experimentally it did also prove to be very important, as most failed attempts were due to hitting either the top or bottom part of a hoop. We also wanted to find a solution in which the drone spent slightly longer navigating, as initially there was too much overshoot partially due to the high speeds. The solution turned out to be a combination of a longer timeframe, as well as having offsets within the corridors, which were there to compensate for the (fairly constant) overshoot during critical parts of the navigation. These offsets can be seen as differences in the placements of the corridors - for example hoop two has a corridor with a floor and ceiling z-value higher up compared to the same corridors associated with other hoops.

The script that was used for the uas trajectory is as follows:

```
%% Initialisation
rosshutdown
rosinit
msg = rostopic("echo", '/vrpn_client_node/hoop1/pose');
hoop1 = [msg.Pose.Position.X msg.Pose.Position.Y, msg.Pose.Position.Z]';
msg = rostopic("echo", '/vrpn_client_node/hoop2/pose');
hoop2 = [msg.Pose.Position.X msg.Pose.Position.Y, msg.Pose.Position.Z]';
msg = rostopic("echo", '/vrpn_client_node/hoop3/pose');
hoop3 = [msg.Pose.Position.X msg.Pose.Position.Y, msg.Pose.Position.Z]';
msg = rostopic("echo", '/vrpn_client_node/hoop4/pose');
hoop4 = [msg.Pose.Position.X msg.Pose.Position.Y, msg.Pose.Position.Z]';
rosshutdown
clc
%%
clc;
close all;

knots = [0 6 11.5 16 22 28 29.5]; %time
waypoints = cell(1,7);
waypoints{1} = [1.60 ; -2.62 ; 0];
waypoints{2} = [1.60 ; -2.62 ; 1.30];
waypoints{h1} = hoop1;
waypoints{h2} = hoop2;
waypoints{h3} = hoop3;
waypoints{h4} = hoop4;
waypoints{7} = waypoints{6} - [0; 1; 0];
%waypoints{7} = [1.60 ; -2.62 ; 1];

order = 20;
corridors.times = [knots(1) knots(2)+2.5 ...
    knots(3)-1 knots(3)+1 ...
    knots(4)-1 knots(4)+1 ...
    knots(5)-1 knots(5)+1 ...
    knots(6)-1 knots(6)+1 ...
    knots(7)]; %time to hit the corridors
corridors.x_lower = [waypoints{1}(1)-0.25...
    waypoints{3}(1)+.8 ...
    waypoints{3}(1)+0.25...
    waypoints{3}(1)-0.5...
    waypoints{4}(1)-0.05...
    waypoints{4}(1)-0.05...
    waypoints{5}(1)-0.5...
    waypoints{5}(1)+0.25...
    waypoints{6}(1)-0.05...
    waypoints{6}(1)-0.05...
    waypoints{7}(1)-0.25];
corridors.x_upper = [waypoints{1}(1)+0.25...
    waypoints{3}(1)+1.3 ...
    waypoints{3}(1)+0.5...
    waypoints{3}(1)-0.25...
    waypoints{4}(1)+0.05...
    waypoints{4}(1)+0.05...
    waypoints{5}(1)-0.25...
    waypoints{5}(1)+0.5...
    waypoints{6}(1)+0.05...
    waypoints{6}(1)+0.05...
    waypoints{7}(1)+0.25];
corridors.y_lower = [waypoints{1}(2)-0.25...
    waypoints{3}(2)+0.7...
    waypoints{3}(2)-0.05...
    waypoints{3}(2)-0.1...
    waypoints{4}(2)-1 ...
    waypoints{4}(2)+0.25...
    waypoints{5}(2)-0.05...
    waypoints{5}(2)-0.05...
    waypoints{6}(2)+0.25...
    waypoints{6}(2)-0.5...
    waypoints{7}(2)-0.25];
corridors.y_upper = [waypoints{1}(2)+0.25...
    waypoints{3}(2)+1.2 ...
    waypoints{3}(2)+0.05...
    waypoints{3}(2)...
    waypoints{4}(2)-0.5...
    waypoints{4}(2)+0.5...
    waypoints{5}(2)+0.05...
    waypoints{5}(2)+0.05...
    waypoints{6}(2)+0.5...
    waypoints{6}(2)-0.25...
    waypoints{7}(2)+0.25];
corridors.z_lower = [waypoints{1}(3)-0.5...
    waypoints{3}(3)-0.1...
    waypoints{3}(3)-0.05 waypoints{3}(3)+0.1...
    waypoints{4}(3) + 0.02 waypoints{4}(3) + 0.02...
    waypoints{5}(3)-0.05 waypoints{5}(3)-0.1...
    waypoints{6}(3) waypoints{6}(3)...
    waypoints{7}(3)-0.5];
corridors.z_upper = [waypoints{1}(3)+0.5...
    waypoints{3}(3)+0.1...
    waypoints{3}(3)+0.1 waypoints{3}(3)+0.1...
    waypoints{4}(3)+0.07 waypoints{4}(3)+0.07...
    waypoints{5}(3)+0.1 waypoints{5}(3)+0.1...
    waypoints{6}(3)+0.1 waypoints{6}(3)+0.1...
    waypoints{7}(3)+0.5];

make_plots = true;
poly_traj = uas_minimum_snap(knots, order, waypoints, corridors, make_plots);

%% Superimpose plots
hold on
poly_traj.showPath(3)
plot(out.p.Data(:,1), out.p.Data(:,2))
xlim([-1 4])
```

Listing 1: Driver for the trajectory generation file

To run the script, *traj_gen − matlab* and *sub_utils* folders from Part VI need to be added to Matlab PATH. The simulink used was as follows:

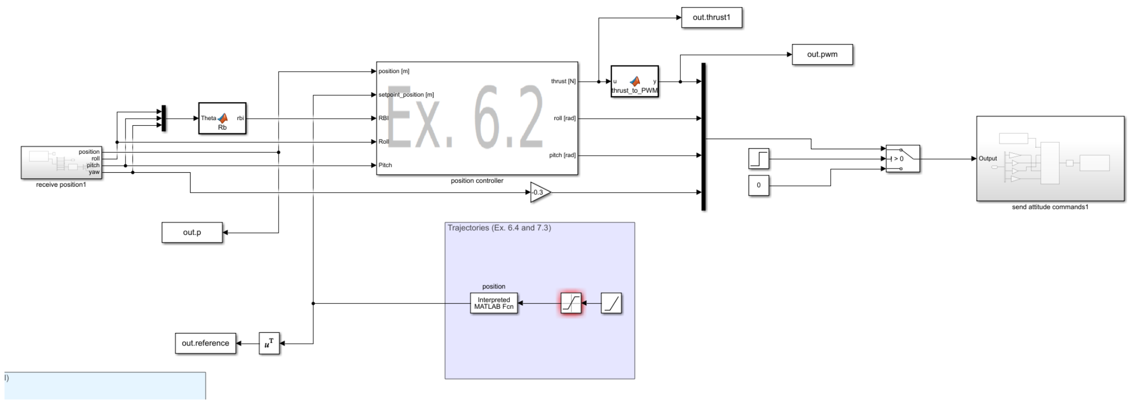


Figure 49: Simulink of the trajectory generation system

This system is largely the same as the one used for 7.2, however instead of the destination reached block we now instead get that from a variable in workspace. The ramp is a linearly increasing function, which then goes into a saturation running from 0 until the end point. That way we get all of the point needed for the trajectory.

Combined this gave us a generated path, which we after testing superimposed our own trajectory on top of.

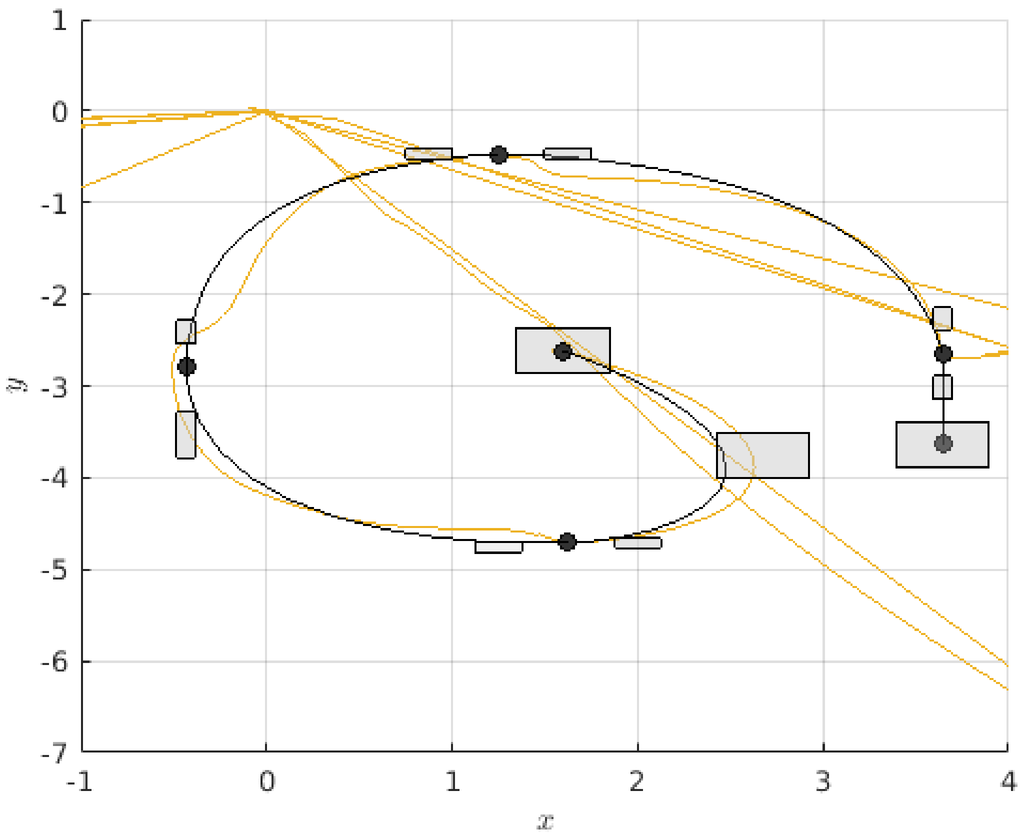


Figure 50: Generated path in the script, and our trajectory on top

Immediately we can see that something looks off. However, the chaotic lines are actually a result of a crash with the final hoop, which caused the drone to flip, hiding some tracker balls, and thus confusing the optitrack position. Sadly, we were unable to capture a run in which we navigated all four successfully, but a video of a 3 hoop run can be found with this link. Note that the trajectory is slightly different because although captured with the same parameters, it does not always hit the edge of the first hoop as it does in the video - in the plot shown it did not. As for the final hoop, it could have likely been fixed by further tuning the position of our corridor between hoop three and four - specifically trying to force the drone to be at a higher altitude since the crashes were consistently on the bottom part of the hoop. On the positive side however, the first three hoops were successfully navigated, and the drone followed the trajectory quite well all things considered.

8 Appendix

8.1 Simulink diagrams

8.1.1 Exercise 3

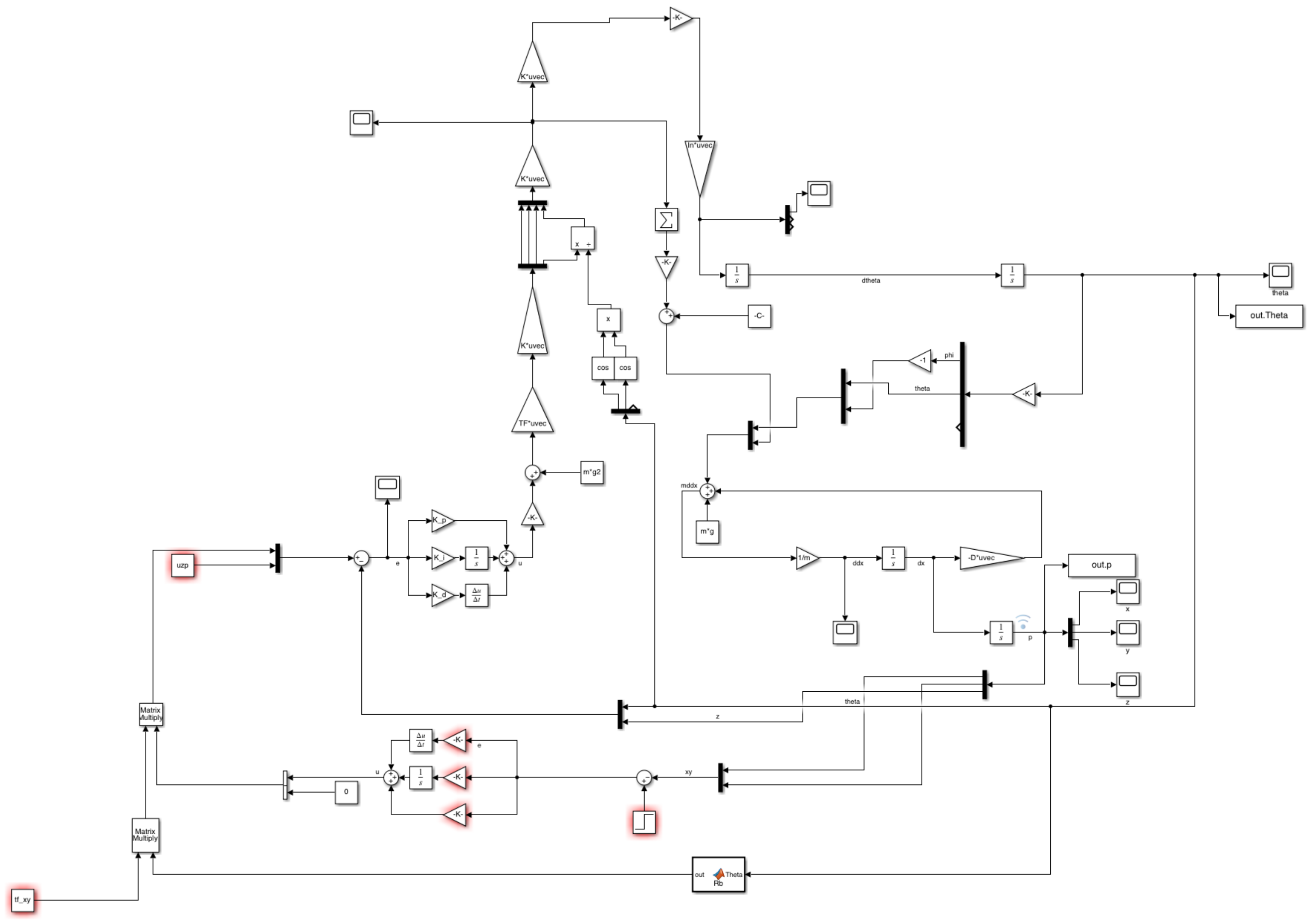


Figure 51: Simulink diagram of linearized system with attitude and position control loop closed.

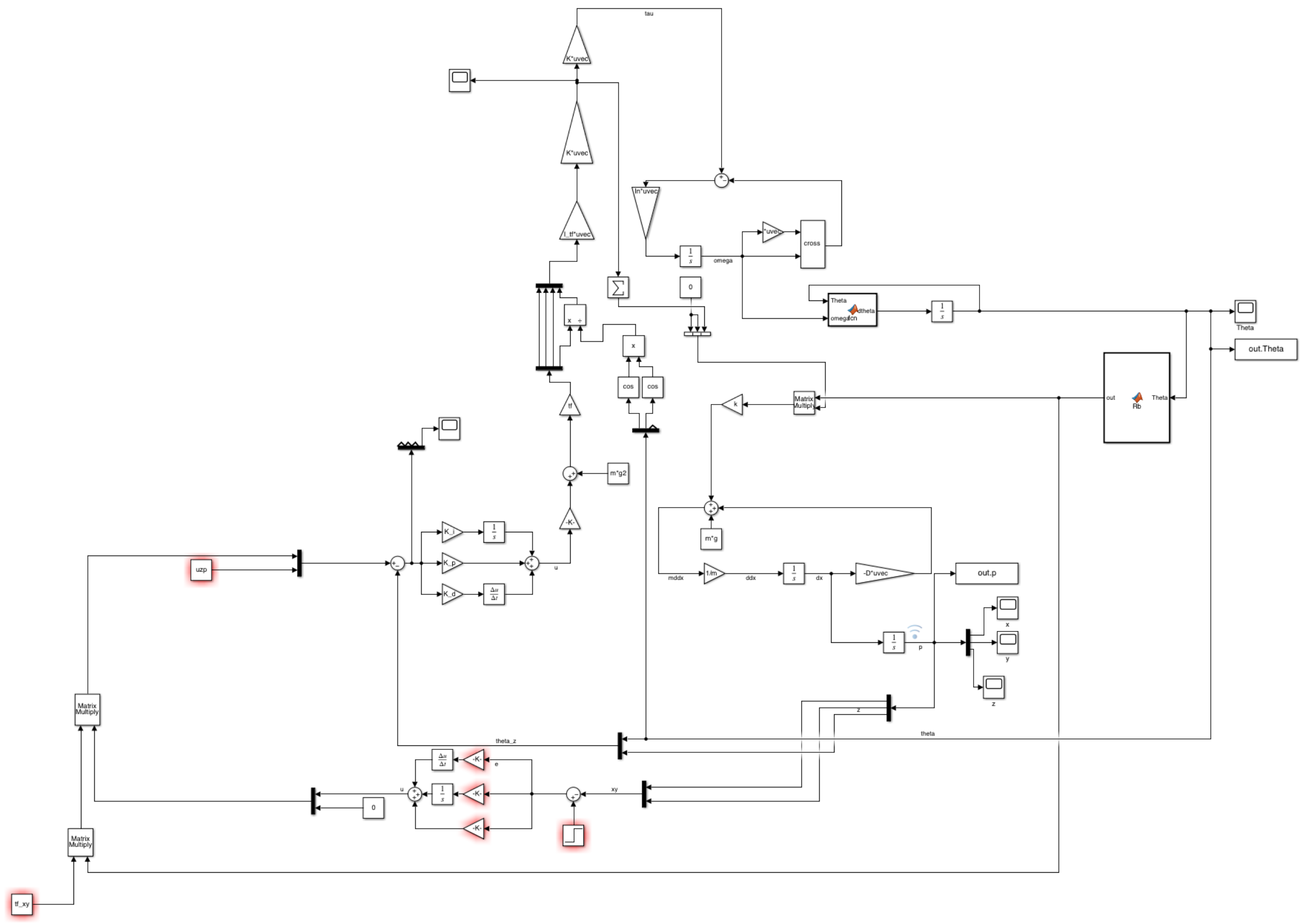


Figure 52: Simulink diagram of no linear system with attitude and position control loop closed.

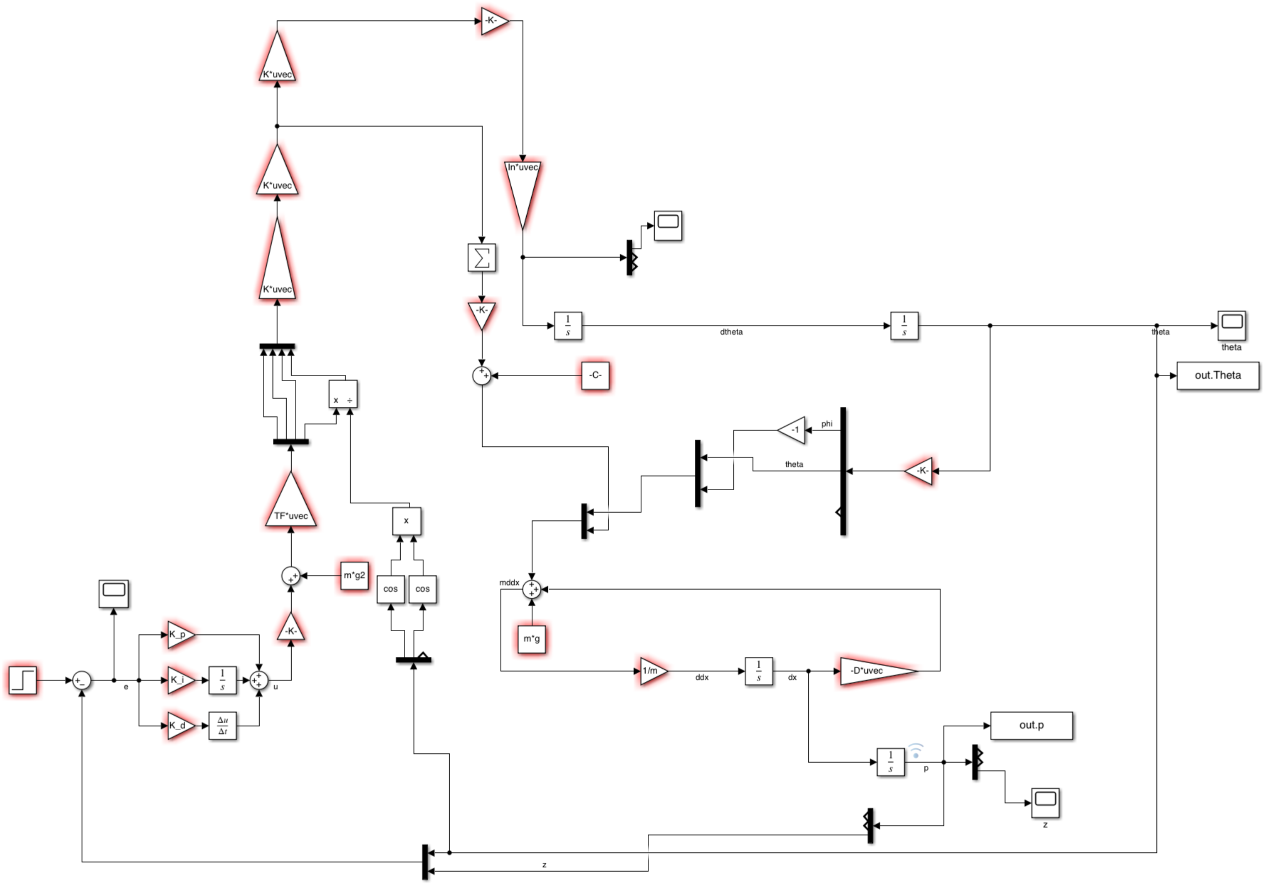


Figure 53: Simulink diagram of linearized system with attitude control loop closed.

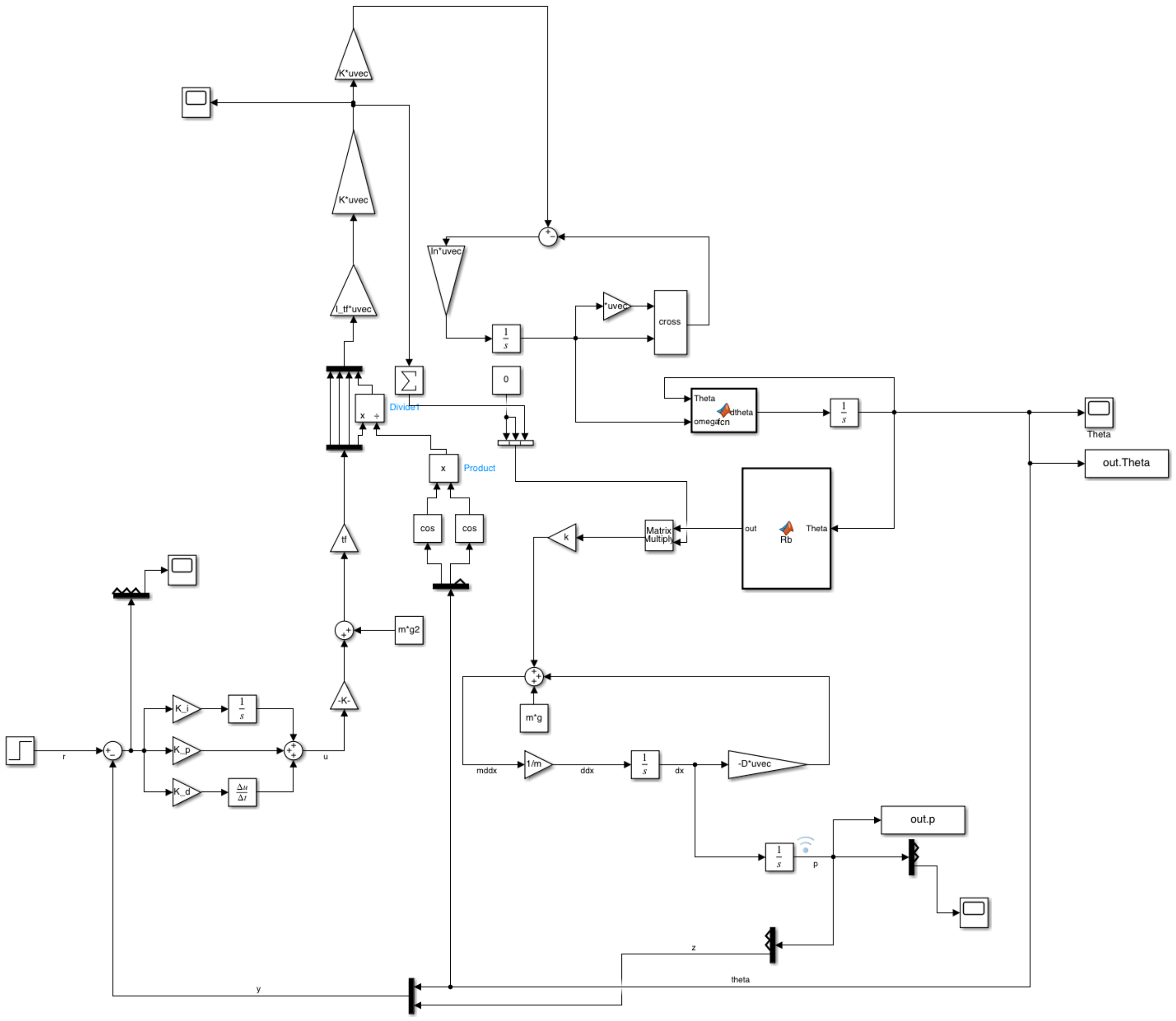


Figure 54: Simulink diagram of non linear system with attitude control loop closed.

8.2 Drivers

8.2.1 Exercise 3

```
% Constants
m = 0.5;
k = 0.01;
L = 0.225;
b = 0.001;
D = diag([0.01, 0.01, 0.01]);
I = diag([3e-6, 3e-6, 1e-5]);
I_tf = diag([3e-6, 3e-6, 1e-5, 1]);
In = inv(I);
w0 = sqrt(9.81*m/(4*k));
g = [0,0,-9.81];
g2 = [0,0,0,-9.81];

K_pxymax = 0.143;
```

```
T_cxy = 5.2;

% Position Controller
K_pxy = 0.2; % 0.6* Kpxymax = 0.0858
K_dxy = 1.25; % Tc/8 = 0.65
K_ixy = 0; % 2/Tc = 0.3846

% Controller angles
K_p = 1.5;
K_d = 0.8; %Tc / 8
K_i = 0.1; %1.4925....


ref = 10/180*pi;
% Reference
phi = 0;
theta = 0;
psi = 0;
z = 0;
x = 0;
y = 2;

uxy = [x
    y];

uzp = [psi
    z];

% Transfer func
tf = [1
    1
    1
    1/k];

mvec = [1
    1
    1
    m];

TF = diag(tf);

tf2 = [L*k, 0, -L*k, 0
    0, L*k, 0, -L*k
    b, -b, b, -b
    1, 1, 1, 1];

TF2 = inv(tf2);

tf_34 = [L*k, 0, -L*k, 0
    0, L*k, 0, -L*k
    b, -b, b, -b];

tf_xy = [0 -1 0
    1 0 0];

% Vector of angular speeds
Omega = [w0
    w0
    w0
    w0];

omg_vec = [0
    0
    sum(Omega.^2)];

tau = 2*w0*[L*k*(Omega(1) - Omega(3))
    L*k*(Omega(2) - Omega(4))
    b*(Omega(1) - Omega(2) + Omega(3) - Omega(4))];
```

Listing 2: Driver for Linear System

```
% Constants
m = 0.5;
k = 0.01;
L = 0.225;
b = 0.001;
D = diag([0.01, 0.01, 0.01]);
I = diag([3e-6, 3e-6, 1e-5]);
I_tf = diag([3e-6, 3e-6, 1e-5, 1]);
In = inv(I);
w0 = sqrt(9.81*m/(4*k));
g = [0,0,-9.81];
g2 = [0,0,0,-9.81];

K_pxymax = 0.143;
T_cxy = 5.2;

% Position Controller
K_pxy = 0.2; % 0.6* Kpxymax = 0.0858
K_dxy = 1.25; % Tc/8 = 0.65
K_ixy = 0; % 2/Tc = 0.3846

% Controller angles
K_p = 1;
K_d = 1.25; %Tc / 8
K_i = 0.1; %1.4925....

ref = 10/180*pi;
% Reference
phi = 0;
theta = 0;
psi = 0;
z = 0;
x = 0;
y = 2;

uxy = [x
    y];

uzp = [psi
    z];

% Transfer func
tf = [1
    1
    1
    1/k];
```

```

mvec = [1
    1
    1
    m];

TF = diag(tf);

tf2 = [L*k, 0, -L*k, 0
    0, L*k, 0, -L*k
    b, -b, b, -b
    1, 1, 1, 1];

TF2 = inv(tf2);

tf_34 = [L*k, 0, -L*k, 0
    0, L*k, 0, -L*k
    b, -b, b, -b];

tf_xy = [0 -1 0
    1 0 0];

TF_xy = tf_xy*eye(3);
% Vector of angular speeds
Omega = [w0
    w0
    w0
    w0];

omg_vec = [0
    0
    sum(Omega.^2)];

tau = 2*w0*[L*k*(Omega(1) - Omega(3))
    L*k*(Omega(2) - Omega(4))
    b*(Omega(1) - Omega(2) + Omega(3) - Omega(4))];
```

Listing 3: Driver for Nonlinear System

8.2.2 Exercise 6

```
%% INITIALIZATION

%clear
close all
clc

%% SIMULATION PARAMETERS
% Full test
route = [
    0 0 1
    7 0 1
    7 3 1
    5 3 1
    5 2 1
    3 2 1
    3 3 1
    1 3 1
    1 4 1
    0 4 1
    0 6 1
    3 6 1
    3 5 1
];
%% Non-full test
%route = load('route_ex61.mat').route_2d-1; %path from ex 6.1

%OPTION 1 - TEST ROUTE
%route = [0 0 1 ; 9 0 1 ; 9 9 1]; %test route

%OPTINOA 2 - EX6.1 ROUTE
%run desired section in maze-ex6.4/maze_plotting script and comment
%the route above to r
%route = [route ones(size(route,1),1)];

% 6.2 test routes
%route = [0 0 1; 1 0 1];
%route = [0 0 1; 5 0 1];
%route = [0 0 1; 9 0 1];


wall_color = [0.8 0.2 0.2];
sample_time = 4e-2;
publish_rate = 1 * sample_time;
x0 = 36;
y0 = 80;
z0 = 1;
g = 9.80665 ;
mass_drone = 0.68 ;
mass_rod = 0.0;
mass_tip = 0;
mass_total = mass_drone + mass_rod + mass_tip;
stiffness_rod = 100 ;
critical_damping_rod = 2 * sqrt(mass_total * stiffness_rod) ;
stiffness_wall = 100 ;
critical_damping_wall = 2 * sqrt(mass_total * stiffness_wall) ;
inertia_xx = 0.007 ;
inertia_yy = 0.007 ;
inertia_zz = 0.012 ;
arm_length = 0.17 ;
rotor_offset_top = 0.01 ;
motor_constant = 8.54858e-06 ;
moment_constant = 0.016 ;
max_rot_velocity = 838 ;
allocation_matrix = ...
    [1 1 1 1
     0 arm_length 0 -arm_length
     -arm_length 0 arm_length 0
     -moment_constant moment_constant -moment_constant moment_constant] ;
mix_matrix = inv(motor_constant * allocation_matrix) ;
air_density = 1.2041;
drag_coefficient = 0.47;
reference_area = pi * 75e-3^2;


% Position Controller values
```

```
tf_xy = [0 -1 0
    1 0 0];
K_pxy = diag([5 5 2]);
K_dxy = diag([3.5 3.5 1.8]);
K_ixy = diag([0 0 0]);

% Attitude Controller values
I_xyz = diag([inertia_xx inertia_yy inertia_zz]);
K_p = diag([5 5 1]);
K_d = diag([1.5 1.5 1.5]);
K_i = diag([0 0 0]);
```

Listing 4: Driver for 6.2 & 6.3

8.3 A* 3D

```
function [ route ] = astar_3d(map, start, end_, length_cost)
    % Check if length_cost was given
    if ¬exist('length_cost', 'var')
        length_cost = 1;
    end
    % Define the limits of the map
    max_x = length(map(:,1,1));
    max_y = length(map(1,:,1));
    max_z = length(map(1,1,:));

    % Children must be initalized to have nodes in it
    % The arrays keeping track of the nodes must initialized
    % containing a node. These flags tells the first node in the
    % closed and children array to be put in directly
    first_closed = 1;
    first_children = 1;
    closed = [];
    children = [];

    % Create the first node at the start position
    parent_node = node;
    parent_node.position = start;
    parent_node.h = parent_node.calc_dist_3d(end_);
    parent_node.g = parent_node.calc_dist_3d(start);
    parent_node.f = parent_node.h + parent_node.g;

    % Flag used to skip nodes which is already added
    continue_flag = 0;

    % Slow the calculation down,
    % so it can be followed in real time
    pause on;

    % Keep running until the end point is reached
    while ¬(parent_node.position(1) == end_(1) && ...
            parent_node.position(2) == end_(2) && ...
            parent_node.position(3) == end_(3))
        % Run through the surronding squares
        for x = -1:1
            for y = -1:1
                for z = -1:1
                % Skip the node itself
                % And also dont allow for diagonal movement
                % As that will create problems when navigating the
                % real maze
                if ¬((x == 0 && y == 0 && z == 0) ||...
                     (abs(x) + abs(y) + abs(z) > 1))
                    node_pos = [parent_node.position(1) + x, ...
                                parent_node.position(2) + y ...
                                parent_node.position(3) + z];
                    % Check if the children is within the map
                    if ¬(node_pos(1) < 1 || node_pos(1) > max_x || ...
                         node_pos(2) < 1 || node_pos(2) > max_y || ...
                         node_pos(3) < 1 || node_pos(3) > max_z)
                        % Check if the children is an obstacle
                        if ¬(map(node_pos(1), node_pos(2), node_pos(3)) == 1)
                            % Check if the node have been visited
                            for closed_i = 1:length(closed)
                                if node_pos == closed(closed_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end
                            % Check if the node is already a child
                            for child_i = 1:length(children)
                                if node_pos == children(child_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end

                            % Check if this node should be skipped
                            if continue_flag == 1
                                continue_flag = 0;
                                continue
                            end

                            % Define the child node
                            temp_node = node;
                            % Note its parent
                            temp_node.parent = parent_node;
                            % Note its position
                            temp_node.position = node_pos;
                            % Calculate the distance from the node
                            % to the end point
                            temp_node.h = temp_node.calc_dist_3d(end_);
                            % Calculate the distance from start
                            temp_node.g = temp_node.calc_dist_3d(start);
                            % Calculate the total cost of the node
                            temp_node.f = temp_node.h + temp_node.g;

                            % Add the node to the children array
                            % Check if it is the first child
                            % being added
                            if first_children == 1
                                first_children = 0;
                                children = [temp_node];
                            else
```

```
                                % Otherwise expand the children array
                                children(end+1) = temp_node;
                            end
                        end
                    end
                end
            end
        end
    end

    % Add the parent node to the list of closed nodes
    if first_closed == 1
        first_closed = 0;
        closed = [parent_node];
    else
        closed(end+1) = parent_node;
    end
        % Choose the child node with the lowest f value
        lowest_f = 999999;
        lowest_child_i = -1;
        for child_i = 1:length(children)
            if children(child_i).f < lowest_f
                lowest_f = children(child_i).f;
                lowest_child_i = child_i;
            end
        end

        % Check if there still is routes avaliable
        if length(children) == 0
           route = NaN;
           return
        end

        % Update the parent to the children
        % with the lowest f value
        parent_node = children(lowest_child_i);

        % Delete the new parent from the children
        children(lowest_child_i) = [];
    end

    % Find the route that the algorithm took
    % Init the route array
    route = [parent_node.position];
    % Keep going until the route is back at the start position
    while  ¬(parent_node.position(1) == start(1) && ...
             parent_node.position(2) == start(2) && ...
             parent_node.position(3) == start(3))
       % Update the route by going backwards through the parents
       parent_node = parent_node.parent;
       route = cat(1,route,parent_node.position);
    end
    route = flip(route);
end
```

8.4 Greedy Search 2D

```
function [ route ] = dfs( map, start, end_, length_cost )
    % Check if length_cost was given
    if ¬exist('length_cost', 'var')
        length_cost = 1;
    end
    % Define the limits of the map
    max_x = length(map(:,1,1));
    max_y = length(map(1,:,1));

    % Children must be initalized to have nodes in it
    % The arrays keeping track of the nodes must initialized
    % containing a node. These flags tells the first node in the
    % closed and children array to be put in directly
    first_closed = 1;
    first_children = 1;
    closed = [];
    children = [];

    % Create the first node at the start position
    parent_node = node;
    parent_node.position = start;
    parent_node.h = parent_node.calc_dist(end_);
    parent_node.f = parent_node.h;

    % Flag used to skip nodes which is already added
    continue_flag = 0;

    % Slow the calculation down,
    % so it can be followed in real time
    pause on;

    % Keep running until the end point is reached
    while ¬(parent_node.position(1) == end_(1) && ...
            parent_node.position(2) == end_(2))
        % Run through the surronding squares
        for x = -1:1
            for y = -1:1
                % Skip the node itself
                % And also dont allow for diagonal movement
                % As that will create problems when navigating the
                % real maze
                if ¬((x == 0 && y == 0) ||...
                     (abs(x) + abs(y) > 1))
                    node_pos = [parent_node.position(1) + x, ...
                                parent_node.position(2) + y];
                    % Check if the children is within the map
                    if ¬(node_pos(1) < 1 || node_pos(1) > max_x || ...
                         node_pos(2) < 1 || node_pos(2) > max_y)
                        % Check if the children is an obstacle
                        if ¬(map(node_pos(1), node_pos(2)) == 1)
                            % Check if the node have been visited
                            for closed_i = 1:length(closed)
                                if node_pos == closed(closed_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end
                            % Check if the node is already a child
                            for child_i = 1:length(children)
                                if node_pos == children(child_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end

                            % Check if this node should be skipped
                            if continue_flag == 1
                                continue_flag = 0;
                                continue
                            end

                            % Define the child node
                            temp_node = node;
                            % Note its parent
                            temp_node.parent = parent_node;
                            % Note its position
                            temp_node.position = node_pos;
                            % Calculate the distance from the node
                            % to the end point
                            temp_node.h = temp_node.calc_dist(end_);
                            % Calculate the total cost of the node
                            temp_node.f = temp_node.h;

                            % Add the node to the children array
                            % Check if it is the first child
                            % being added
                            if first_children == 1
                                first_children = 0;
                                children = [temp_node];
                            else
                                % Otherwise expand the children array
                                children(end+1) = temp_node;
                            end
                        end
                    end
                end
            end
        end

    % Add the parent node to the list of closed nodes
    if first_closed == 1
        first_closed = 0;
        closed = [parent_node];
    else
        closed(end+1) = parent_node;
    end
        % Choose the child node with the lowest f value
        lowest_f = 999999;
        lowest_child_i = -1;
        for child_i = 1:length(children)
            if children(child_i).f < lowest_f
                lowest_f = children(child_i).f;
                lowest_child_i = child_i;
            end
        end

        % Check if there still is routes avaliable
        if length(children) == 0
           route = NaN;
           return
        end

        % Update the parent to the children
        % with the lowest f value
        parent_node = children(lowest_child_i);

        % Delete the new parent from the children
        children(lowest_child_i) = [];
    end

    % Find the route that the algorithm took
    % Init the route array
    route = [parent_node.position];
    % Keep going until the route is back at the start position
    while  ¬(parent_node.position(1) == start(1) && ...
             parent_node.position(2) == start(2))
       % Update the route by going backwards through the parents
       parent_node = parent_node.parent;
       route = cat(1,route,parent_node.position);
    end
    route = flip(route);
end
```

8.5 Greedy Search 3D

```
function [ route ] = greedy_3d(map, start, end_, length_cost)
    % Check if length_cost was given
    if ¬exist('length_cost', 'var')
        length_cost = 1;
    end
    % Define the limits of the map
    max_x = length(map(:,1,1));
    max_y = length(map(1,:,1));
    max_z = length(map(1,1,:));

    % Children must be initalized to have nodes in it
    % The arrays keeping track of the nodes must initialized
    % containing a node. These flags tells the first node in the
    % closed and children array to be put in directly
    first_closed = 1;
    first_children = 1;
    closed = [];
    children = [];

    % Create the first node at the start position
    parent_node = node;
    parent_node.position = start;
    parent_node.h = parent_node.calc_dist(end_);
    parent_node.f = parent_node.h;

    % Flag used to skip nodes which is already added
    continue_flag = 0;

    % Slow the calculation down,
    % so it can be followed in real time
    pause on;

    % Keep running until the end point is reached
    while ¬(parent_node.position(1) == end_(1) && ...
            parent_node.position(2) == end_(2) && ...
            parent_node.position(3) == end_(3))
        % Run through the surronding squares
        for x = -1:1
            for y = -1:1
                for z = -1:1
                % Skip the node itself
                % And also dont allow for diagonal movement
                % As that will create problems when navigating the
                % real maze
                if ¬((x == 0 && y == 0 && z == 0) ||...
                     (abs(x) + abs(y) + abs(z) > 1))
                    node_pos = [parent_node.position(1) + x, ...
                                parent_node.position(2) + y ...
                                parent_node.position(3) + z];
                    % Check if the children is within the map
                    if ¬(node_pos(1) < 1 || node_pos(1) > max_x || ...
                         node_pos(2) < 1 || node_pos(2) > max_y || ...
                         node_pos(3) < 1 || node_pos(3) > max_z)
                        % Check if the children is an obstacle
                        if ¬(map(node_pos(1), node_pos(2), node_pos(3)) == 1)
                            % Check if the node have been visited
                            for closed_i = 1:length(closed)
                                if node_pos == closed(closed_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end
                            % Check if the node is already a child
                            for child_i = 1:length(children)
                                if node_pos == children(child_i).position
                                    % Note that this node is not
                                    % to be added to children
                                    continue_flag = 1;
                                end
                            end

                            % Check if this node should be skipped
                            if continue_flag == 1
                                continue_flag = 0;
                                continue
                            end

                            % Define the child node
                            temp_node = node;
                            % Note its parent
                            temp_node.parent = parent_node;
                            % Note its position
                            temp_node.position = node_pos;
                            % Calculate the distance from the node
                            % to the end point
                            temp_node.h = temp_node.calc_dist_3d(end_);
                            % Calculate the total cost of the node
                            temp_node.f = temp_node.h;

                            % Add the node to the children array
                            % Check if it is the first child
                            % being added
                            if first_children == 1
                                first_children = 0;
                                children = [temp_node];
                            else
                                % Otherwise expand the children array
                                children(end+1) = temp_node;
                            end
                        end
                    end
                end
            end
        end
    end

    % Add the parent node to the list of closed nodes
    if first_closed == 1
        first_closed = 0;
        closed = [parent_node];
    else
        closed(end+1) = parent_node;
    end
        % Choose the child node with the lowest f value
        lowest_f = 999999;
        lowest_child_i = -1;
        for child_i = 1:length(children)
            if children(child_i).f < lowest_f
                lowest_f = children(child_i).f;
                lowest_child_i = child_i;
            end
        end

        % Check if there still is routes avaliable
        if length(children) == 0
           route = NaN;
           return
        end

        % Update the parent to the children
        % with the lowest f value
        parent_node = children(lowest_child_i);

        % Delete the new parent from the children
        children(lowest_child_i) = [];
    end

    % Find the route that the algorithm took
    % Init the route array
    route = [parent_node.position];
    % Keep going until the route is back at the start position
    while  ¬(parent_node.position(1) == start(1) && ...
             parent_node.position(2) == start(2) && ...
             parent_node.position(3) == start(3))
       % Update the route by going backwards through the parents
```

```
        parent_node = parent_node.parent;
        route = cat(1,route,parent_node.position);
    end
    route = flip(route);
end
```

References

[1] B.-A. Moti, "A tutorial on euler angles and quaternions," 2014-2017.

[2] D. Starkey, "Cox-deboor equations for b-splines." https://www.cs.montana.edu/courses/spring2009/525/dslectures/CoxdeBoor.pdf.

[3] R. W. . David Eberly, Geometric Tools, "Euler angle formulas." https://www.geometrictools.com/Documentation/EulerAngles.pdf.