Danmarks
Tekniske
Universitet

Unmanned Autonomous Systems
- Group 14 -

Authors

Frederik - s173916
Gowrisankar - s212919
Hiten - s210208
Marcus - s213424

June 14, 2023

Contents

1 Introduction **1**

1.0.1 Contributions 1

2 Rotation **1**

2.1 Ex.1.1 - Euler Angles 2

2.2 Ex.1.2 - Euler Angles 2

2.3 Ex.1.3 - Roll, Pitch, Yaw 2

2.4 Ex.1.4 - Minimal Rotation 2

2.5 Ex.1.5 - Qaurtenions 3

2.6 Ex.1.6 - Compositions of Rotation Matrices and Qaurtenions 4

3 Modelling **6**

3.0.1 Define the rotation matrix representing the orientation of the body-fixed frame w.r.t. the inertial frame 6

3.0.2 Define the relation between the angular velocity and the rotational velocity of the body-fixed frame 7

3.0.3 Write the linear and angular dynamic equation of the drone in compact form 7

3.0.4 Make a MATLABSimulink model of the drone, given initial conditions $p(0) = [0, 0, 0]^T$ 7

3.0.5 Exercise 3.3 - 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): 14

4 Path Planning **16**

4.1 Depth First Search 16

4.2 Breadth First Search 20

4.2.1 Advantages of BFS vs DFS 21

4.2.2 What algorithm is preferred for robots? 22

4.3 Dijkstra's Algorithm 22

4.4 Greedy Best First Search 24

4.5 A* Search 25

4.6 Advantages and Disadvantages of Greedy best-first search and A* search algorithms 27

4.7 Solving 3D Maze with Greedy best-first Search 28

4.8 Solving 3D Maze with A* Search 28

4.8.1 Additional Remarks 29

5 Trajectory Generation **29**

5.1 Quintic Splines 29

5.2 B-splines 30

5.3 Computation of a DCM 31

6 Simulation **33**
6.1 Ex.6.1 - Navigate a 2D maze 33
6.2 Ex.6.2 - Re-implementing the Position Controller 33
6.3 Ex.6.3 - Re-implementing the Attitude Controller 35
6.4 Ex.6.4 - Aggressively Navigating a 2D Maze 37

7 PART 7: Demonstration **38**
7.1 Ex.7.1 - Navigating a 3D maze 38
7.2 Ex.7.2 - Porting the position controller to the real system 39
7.3 Ex.7.3 - Aggressively navigating through hoops 42
7.4 Error Optimization - Hopefully helpful for future groups 44

List of Figures **I**

List of Tables **III**

Nomenclature **IV**

References **IV**

8 Appendix A **V**

1 Introduction

In the course 31390 we had the task of getting a drone to fly around autonomously on a generated path. Throughout the course, we have made theoretical assignments, which have aimed to prepare us for the necessary theories and form an understanding of the end product. The last tasks have thus been to make the drone fly, by designing controllers and using known methods to generate the most optimal path.

The report places most emphasis on tasks 6 and 7, as they are the ones that are important in describing the work done. These tasks hereby describe the choices we in the group have made as well as our design process. furthermore, the results of each of the tasks are described against the theoretical/desired results. During the demo day, the group also showed that the generated solutions achieved the desired results, where all the tests were performed and performed live. Therefore, not much emphasis is placed on tasks 1 to 5 in the task, as these have primarily been to provide an understanding of the system to be designed/processed.

1.0.1 Contributions

We have chosen to work together during the execution of all the tasks/exercises. In this way, all team members have achieved the greatest benefit from the project. We have therefore each contributed approx. 25% in each part/assignment.

2 Rotation

Euler Angle Representations

When talking about transformations and orientations, the rotation matrix is important. The rotation matrix for rotation about the x-axis, y-axis and z-axis, respectively, is given below.

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

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

$$R_z = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

2.1 Ex.1.1 - Euler Angles

The rotation matrix corresponding to the Euler angles ZXZ is given by:

$$R_{ZXZ} = R_z.R_x.R_z \quad (4)$$

$$\begin{bmatrix} \cos(\psi)^2 - \cos(\phi)\,\sin(\psi)^2 & -\sigma_1 & \sin(\phi)\,\sin(\psi) \\ \sigma_1 & \cos(\psi)^2 + \cos(\phi)\,\cos(\psi)^2 - 1 & -\cos(\psi)\,\sin(\phi) \\ \sin(\phi)\,\sin(\psi) & \cos(\psi)\,\sin(\phi) & \cos(\phi) \end{bmatrix} \quad (5)$$

where

$\sigma_1 = \frac{\sin(2\,\psi)\,(\cos(\phi)+1)}{2}$

2.2 Ex.1.2 - Euler Angles

The rotation matrix corresponding to the Euler angles ZYZ with $c_\theta = 0$ is given by:

$$R_{zyz} = \begin{bmatrix} cos(2\psi) & -sin(2\psi) & 0 \\ sin(2\psi) & cos(2\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6)$$

2.3 Ex.1.3 - Roll, Pitch, Yaw

Roll Pitch Yaw

The rotation matrix corresponding to the Roll Pitch Yaw representation with $c_\theta = 0$ is given by:

$$R_{zyx} = \begin{bmatrix} cos(\phi) & sin(\phi)sin(\psi) & cos(\phi)sin(\psi) \\ sin(\psi) & -cos(\psi)sin(\phi) & -cos(\phi)cos(\psi) \\ 0 & cos(\phi) & -sin(\phi) \end{bmatrix} \quad (7)$$

2.4 Ex.1.4 - Minimal Rotation

Given a pair of unit vectors u and w (v = from and w = to), the minimal rotation that brings u in w is described by the axis of rotation and the angle. The axis of rotation is obtained as the cross product of the two vectors and the angle as the arccosine of the dot product between the vectors.

```
axis = cross(u,w)
theta = acos(dot(v,w)/norm(v)*norm(w))
```

Quartenions

A quartenion is given by , q1 = (a1 b1 c1 d1): The elements b, c, and d are the "vector part" of the quaternion, and can be thought of as a vector about which rotation should be performed. The element a is the "scalar part" that specifies the amount of rotation that should be performed about the vector part.

$$q = \begin{bmatrix} cos(0.5\theta) \\ v_x sin(0.5\theta) \\ v_y sin(0.5\theta) \\ v_z sin(0.5\theta) \end{bmatrix} \tag{8}$$

2.5 Ex.1.5 - Qaurtenions

The quartenion for a rotation of 180 degree about x-axis is obtained from 8 with

$$v = [v_x v_y v_z] = [100]$$

as:

$$q_1 = q_{180,x} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \tag{9}$$

The quartenion for a rotation of 180 degree about z-axis is obtained from 8 with

$$v = [v_x v_y v_z] = [001]$$

as:

$$q_2 = q_{180,z} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

The rotation represented by the composite quartenion,

$$q = q_1 q_2$$

is obtained by applying the formula for quartenion multiplication. For two quartenions defined as ,

$$q = q_0 + q_1 i + q_2 j + q_3 k$$

$$p = p_0 + p_1 i + p_2 j + p_3 k$$

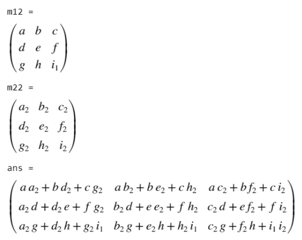
$$
\begin{aligned}
qp &= (q_0 + q_1 i + q_2 j + q_3 k)(p_0 + p_1 i + p_2 j + p_3 k) && (11)\\
&= (q_0p_0 + q_1p_1 i^2 + q_2p_2 j^2 + q_3p_3 k^2) + && (12)\\
&(q_0p_1 i + q_1p_0 i + q_2p_3 jk + q_3p_2 kj) + && (13)\\
&(q_0p_2 j + q_2p_0 j + q_1p_3 ik + q_3p_1 ki) + && (14)\\
&(q_0p_3 k + q_3p_0 k + q_1p_2 ij + q_2p_1 ji) && (15)\\
&= (q_0p_0 - q_1 - q_2p_2 - q_3p_3) + && (16)\\
&(q_0p_1 + q_1p_0 + q_2p_3 - q_3p_2) i + && (17)\\
&(q_0p_2 + q_2p_0 - q_1p_3 + q_3p_1) j + && (18)\\
&(q_0p_3 + q_3p_0 + q_1p_2 - q_2p_1) k && (19)
\end{aligned}
$$

Thus,

$$
\begin{aligned}
q &= q_1 q_2 && (20)\\
&= \begin{bmatrix} 0 & 0 & -1 & 0 \end{bmatrix} && (21)
\end{aligned}
$$

2.6 Ex.1.6 - Compositions of Rotation Matrices and Qaurtenions

- **compose two rotation matrices:** When composing two rotation matrices it corresponds to multiply two matrices of the size 3x3 as the two matrices and the result which are defined in 1. This gives an result which contains 18 additions and 27 multiplications.



```
m12 =
```

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i_1 \end{pmatrix}$$

```
m22 =
```

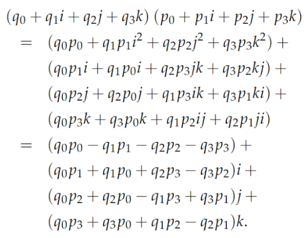
$$\begin{pmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{pmatrix}$$

```
ans =
```

$$\begin{pmatrix} a\,a_2 + b\,d_2 + c\,g_2 & a\,b_2 + b\,e_2 + c\,h_2 & a\,c_2 + b f_2 + c\,i_2 \\ a_2 d + d_2 e + f\,g_2 & b_2 d + e\,e_2 + f\,h_2 & c_2 d + e f_2 + f\,i_2 \\ a_2 g + d_2 h + g_2 i_1 & b_2 g + e_2 h + h_2 i_1 & c_2 g + f_2 h + i_1 i_2 \end{pmatrix}$$

Figure 1: calculations for exercise 1.6.1

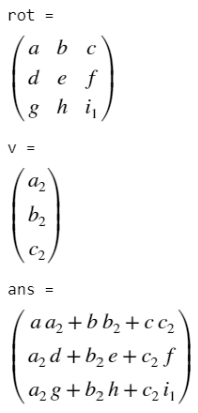
- **compose two quaternions:** For composing two quaternions the method described in Figure 2 is used. This returns 16 multiplications and 15 additions



$$
\begin{aligned}
&(q_0 + q_1 i + q_2 j + q_3 k)\,(p_0 + p_1 i + p_2 j + p_3 k)\\
= \;&(q_0p_0 + q_1p_1 i^2 + q_2p_2 j^2 + q_3p_3 k^2) +\\
&(q_0p_1 i + q_1p_0 i + q_2p_3 jk + q_3p_2 kj) +\\
&(q_0p_2 j + q_2p_0 j + q_1p_3 ik + q_3p_1 ki) +\\
&(q_0p_3 k + q_3p_0 k + q_1p_2 ij + q_2p_1 ji)\\
= \;&(q_0p_0 - q_1p_1 - q_2p_2 - q_3p_3) +\\
&(q_0p_1 + q_1p_0 + q_2p_3 - q_3p_2) i +\\
&(q_0p_2 + q_2p_0 - q_1p_3 + q_3p_1) j +\\
&(q_0p_3 + q_3p_0 + q_1p_2 - q_2p_1) k.
\end{aligned}
$$

Figure 2: calculations for exercise 1.6.2

- **Apply a rotation matrix to a vector:** In Figure 3 we are multiplying a matrix with the size corresponding to a rotation matrix with a vector. From this result we get 9 multiplications and 6 additions



```
rot =
```
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i_1 \end{pmatrix}$$
```
v =
```
$$\begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}$$
```
ans =
```
$$\begin{pmatrix} a\,a_2 + b\,b_2 + c\,c_2 \\ a_2\,d + b_2\,e + c_2\,f \\ a_2\,g + b_2\,h + c_2\,i_1 \end{pmatrix}$$

Figure 3: calculations for exercise 1.6.3

- **Apply a quaternion to a vector:** for multiplying a quaternion with a vector it is necessary to insert an extra element i the beginning of the vector(insert 0 in the top of the vector). Then by using the rule fro multiplaying a vector with a quaternion. Figure 4 illustrates the vector, the quaternion and the result of those two multiplied. We achieve 12 multiplications 8 additions. We are not taking the first negative sign into account due to the fact that it is seen as the same operation for a computer as the multiplication, which is taken into account.

```
v =
```

$$\begin{pmatrix} 0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

```
q =
```

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

```
qv =
```

$$\begin{pmatrix} -b\,v_1 - c\,v_2 - d\,v_3 \\ a\,v_1 + c\,v_3 - d\,v_2 \\ a\,v_2 - b\,v_3 + d\,v_1 \\ a\,v_3 + b\,v_2 - c\,v_1 \end{pmatrix}$$

Figure 4: calculations for exercise 1.6.4

3 Modelling

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

The rotation matrix is defined as:

```
Rot_xyz =
```

$$\begin{pmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\phi)\sin(\theta) - \cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi) + \cos(\phi)\cos(\psi)\sin(\theta) \\ \cos(\theta)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\phi)\sin(\psi)\sin(\theta) & \cos(\phi)\sin(\psi)\sin(\theta) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\phi)\cos(\theta) \end{pmatrix}$$

Figure 5: Rotation matrix representing the orientation of the body-fixed frame

3.0.2 Define the relation between the angular velocity and the rotational velocity of the body-fixed frame

```
omega =
```

$$\begin{pmatrix} \dot{\phi} - \dot{\psi}\sin(\theta) \\ \dot{\theta}\cos(\phi) + \dot{\psi}\cos(\theta)\sin(\phi) \\ \dot{\psi}\cos(\phi)\cos(\theta) - \dot{\theta}\sin(\phi) \end{pmatrix}$$

Figure 6: Relation between $\dot{\theta}$ and ω of the body-fixed frame

3.0.3 Write the linear and angular dynamic equation of the drone in compact form

Figure 8 shows the dynamic equation for the drone.

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

Figure 7: Dynamics given in the lecture

$$\begin{pmatrix} -\frac{-K\,L\,\omega_1{}^2 + \omega_3{}^2 - I_{xx}\,\sigma_1\,\sigma_2 + I_{yy}\,\sigma_1\,\sigma_2}{I_{xx}} \\ -\frac{\omega_4{}^2 - K\,L\,\omega_2{}^2}{I_{yy}} \\ \frac{b\,\left(\omega_1{}^2 - \omega_2{}^2 + \omega_3{}^2 - \omega_4{}^2\right) + I_{xx}\,\left(\dot{\phi} - \dot{\psi}\sin(\theta)\right)\sigma_1 - I_{yy}\,\left(\dot{\phi} - \dot{\psi}\sin(\theta)\right)\sigma_1}{I_{xx}} \end{pmatrix}$$

where

$$\sigma_1 = \dot{\theta}\cos(\phi) + \dot{\psi}\cos(\theta)\sin(\phi)$$

$$\sigma_2 = \dot{\theta}\sin(\phi) - \dot{\psi}\cos(\phi)\cos(\theta)$$

Figure 8: Final dynamic equation describing the drone

3.0.4 Make a MATLABSimulink model of the drone, given initial conditions $p(0) = [0, 0, 0]^T$

To set up a Simulink model of the system, the equations described above are used to model the model. There are several ways to approach the task. Either you can set up some function blocks in Simulink, where it is possible to insert the described equations, or as we in the group have chosen to describe the whole circuit as a function block diagram.

The setup model contains three main functions/blocks - namely the block which describes the dynamic equation, the rewriting of the omega and the linear dynamics of the system itself, which is given by the double-derived differential equation. the composition/setup of the three blocks is shown in Figure 9.

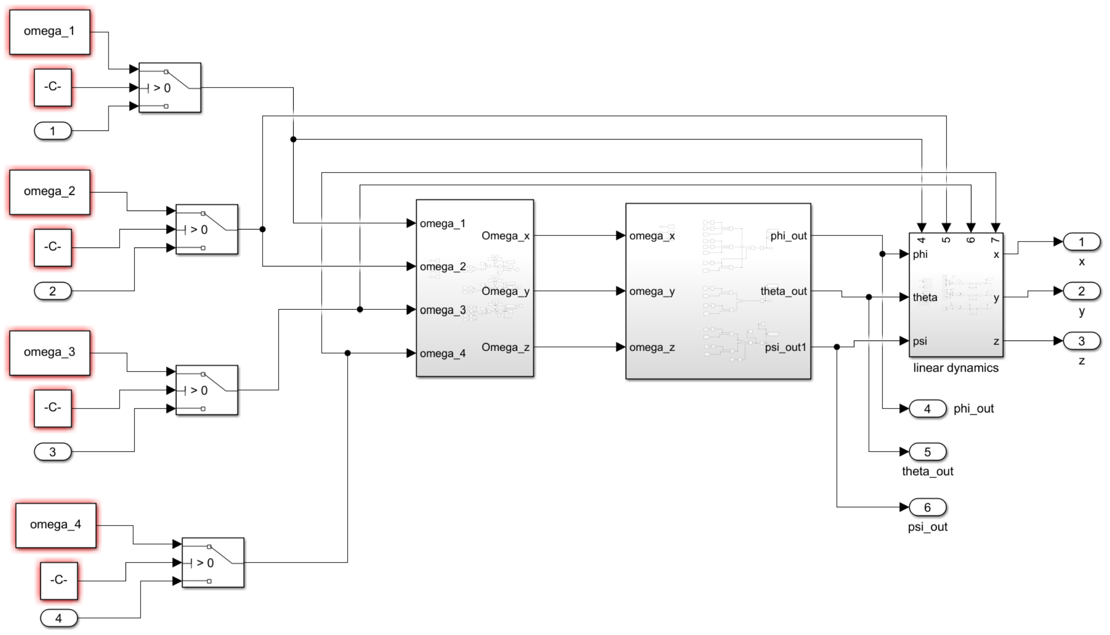


Figure 9: The overall model for the drone system

First, the dynamic equation is described as in Figure 10

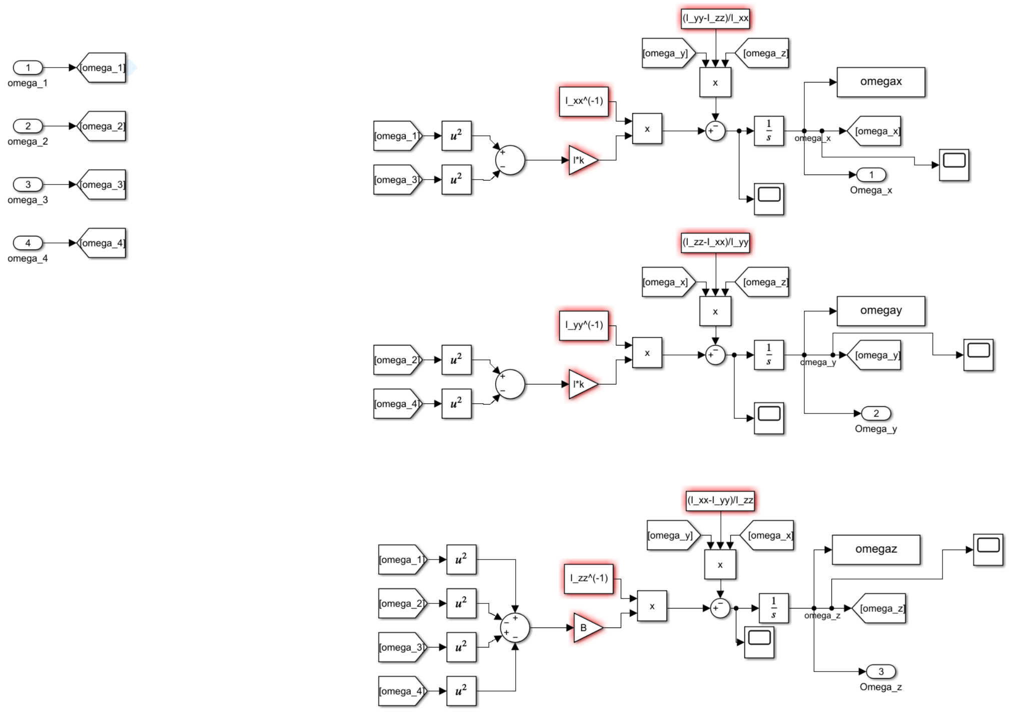


Figure 10: The dynamic equation for the drone

Secondly, the inverse function of the omega arrow is described as in figure 11.

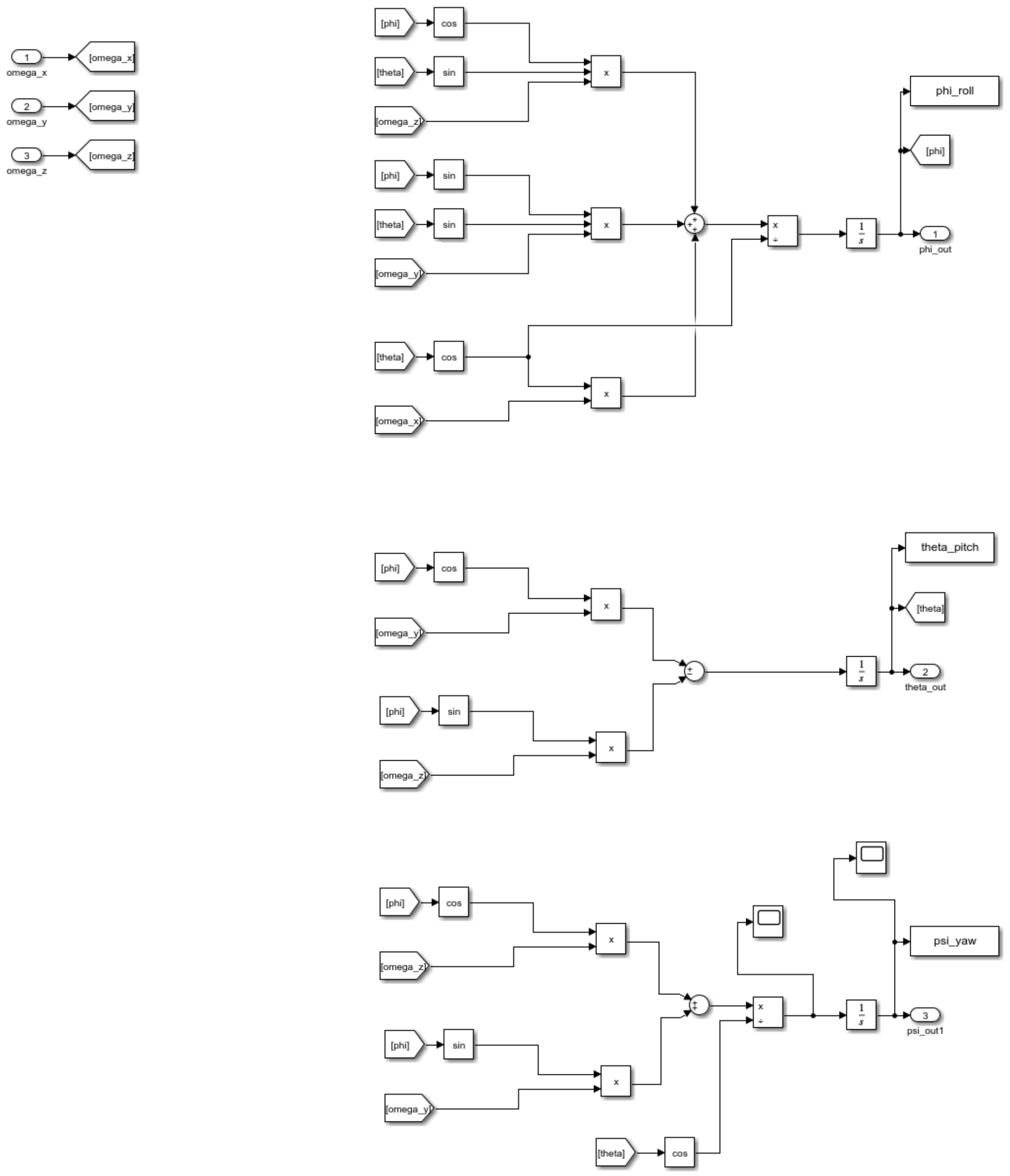


Figure 11: conversion of omega arrow (inverse of omega arrow) - for transforming omega_x,y,z to (phi, theta, psi)

Finally, the linear dynamic is described in figure 12.

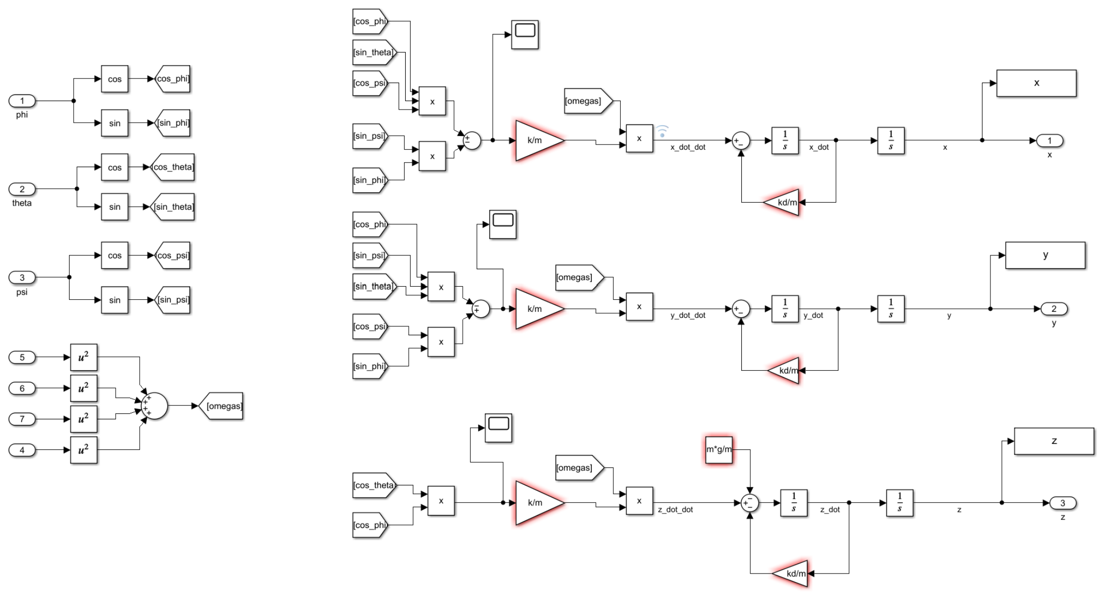


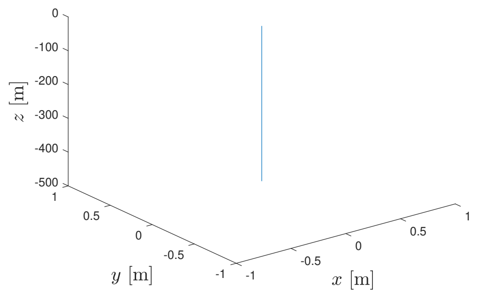
Figure 12: linear dynamic for the drone model

The model described above constitutes the model for the nonlinear system.

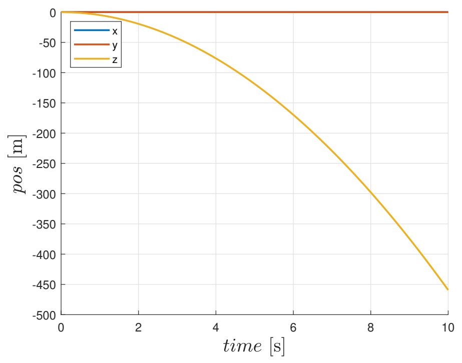
If we now look at testing the system for different omega inputs, we get the following results:

- Make a plot of P and Θ, given $\Omega = [0, 0, 0, 0]^T$ and explain the result

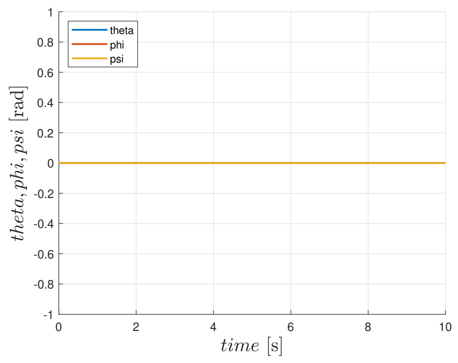
Figure 13a and Figure 13b describe the change in the P (x, y, z) directions, respectively, where Figure 13c illustrates the change for $\Theta(\phi, \theta, \psi)$



(a) The simulation of the system for x, y and z when $\Omega = [0,0,0,0]^T$



(b) The simulation of the system for x, y and z against time when $\Omega = [0,0,0,0]^T$



(c) The simulation of the system for x, y and z when $\Omega = [0,0,0,0]^T$

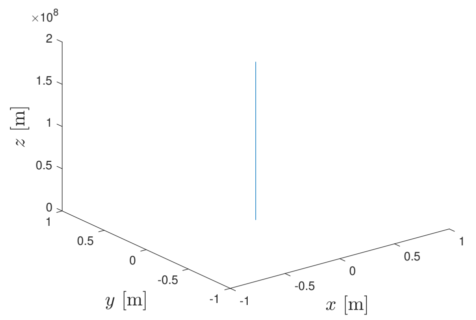
Figure 13: Plots of the behavior of P and Θ, when $\Omega = [0,0,0]^T$

It is clear that for the input [0,0,0,0], the drone falls straight down the z axis as the engines are switched off and the drone is affected by the gravity.

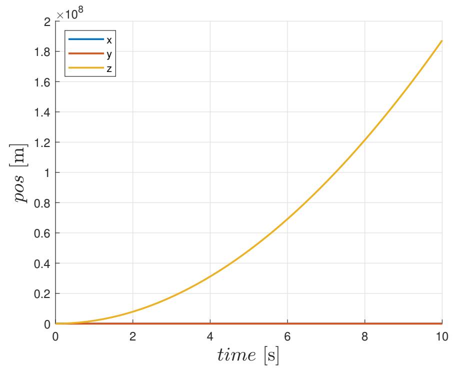
- Make a plot of P and Θ, given $\Omega = [10000, 0, 10000, 0]^T$ and explain the result.

Since the drone is built so that motor 1 and 3 rotates in the same direction and motor 2 and 4 rotate in the same direction, but opposite of motor 1 and 2, it is expected that the drone will rotate around itself while flying. This is because only motor 1 and 3 gets an input.

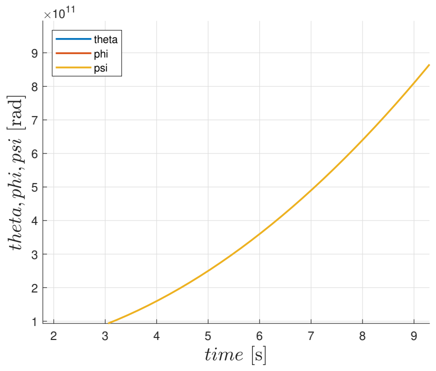
Figure 14a and Figure 14b describe the change in the P (x, y, z) directions, respectively, where Figure 14c illustrates the change for $\Theta(\phi, \theta, \psi)$



(a) The simulation of the system for x, y and z when $\Omega = [10000, 0, 10000, 0]^T$



(b) The simulation of the system for x, y and z against time when $\Omega = [10000, 0, 10000, 0]^T$



(c) The simulation of the system for x, y and z when $\Omega = [0, 10000, 0, 10000, 0]^T$

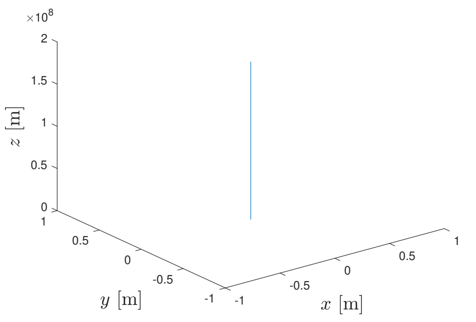
Figure 14: Plots of the behavior of P and Θ, when $\Omega = [10000, 0, 10000, 0]^T$

As expected, it can be seen from the above figures that the two motors form a moment which causes the drone to spin about the z axis due to the direction of rotation of the rotors. It is clear from figure 14c that the drone is now actuated with a positive rotation around ψ, which means that the drone rotates in a positive direction of rotation - ie counterclockwise due to the torque.

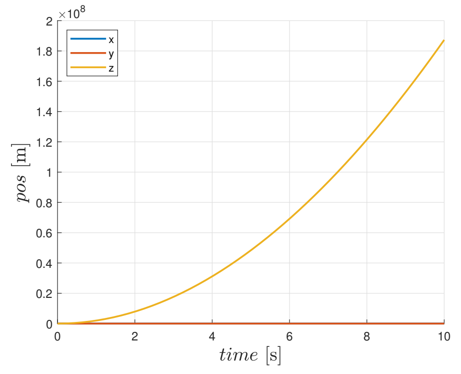
- Make a plot of P and Θ, given $\Omega = [0, 10000, 0, 10000]^T$ and explain the result

It is now expected that the drone will rotate in the opposite direction relative to the previous test, but rise by the same distance up the z axis. This is because it is now motor 2 and 4 that get the same input as motors 1 and 3 had before. Therefore, a moment will be formed which causes the drone to rotate in the negative direction of rotation - clockwise.

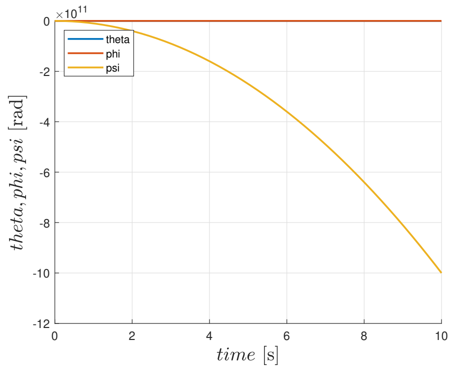
Figure 15a and Figure 15b describe the change in the P (x, y, z) directions, respectively, where Figure 15c illustrates the change for $\Theta(\phi, \theta, \psi)$



(a) The simulation of the system for x, y and z when $\Omega = [0, 0, 10000, 0, 10000]^T$



(b) The simulation of the system for x, y and z against time when $\Omega = [0, 10000, 0, 10000]^T$



(c) The simulation of the system for x, y and z when $\Omega = [0, 10000, 0, 10000]^T$

Figure 15: Plots of the behavior of P and Θ, when $\Omega = [0, 10000, 0, 10000]^T$

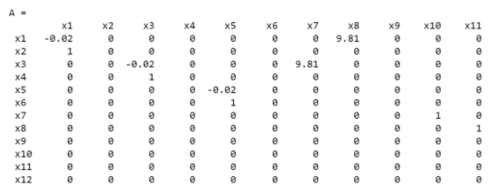
The above plots describe the expected assumption, i.e., it is clearly seen that the drone flies up the z axis as it rotates around itself in the clockwise direction.

3.0.5 Exercise 3.3 - 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):

To linearize, first of all check that the operating points of the system are the right ones, this is done using the matlab function trim. This function takes the nonlinear model in as input, as well as a guess as to what the operation points are.

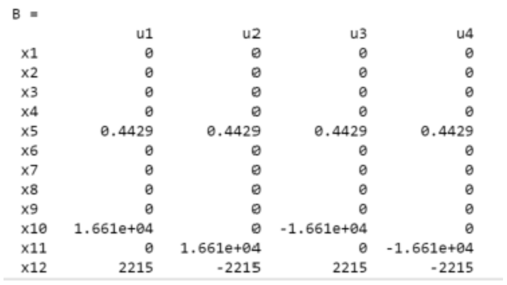
For all states and outputs, therefore, vectors are defined, all of which contain only 0. where for the input we calculate the force required to make the drone stay in position 0 on the z axis of the four motors. The input values can thus be solved from the given equations in the problem sets and thus obtain $z = \sqrt{\frac{m*g}{4*k}} = 11.0736$. The result from the trim function gives the exact same values as our guess, which is why we have the right operating point for the drone.

Now the set model can then be linearized using the matlab function linmod. This function takes the nonlinear model in as input, and the operation points. This now gives the linear model given in state space form. figure 16a, 16b and 16c show the A, B and C matrices of the linear system, respectively. The D matrix consists exclusively of 0s.



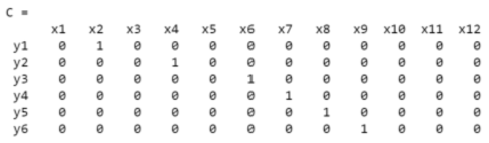
```
A =
        x1     x2     x3     x4     x5     x6     x7     x8     x9    x10    x11
 x1  -0.02      0      0      0      0      0      0   9.81      0      0      0
 x2      1      0      0      0      0      0      0      0      0      0      0
 x3      0      0  -0.02      0      0      0   9.81      0      0      0      0
 x4      0      0      1      0      0      0      0      0      0      0      0
 x5      0      0      0      0  -0.02      0      0      0      0      0      0
 x6      0      0      0      0      1      0      0      0      0      0      0
 x7      0      0      0      0      0      0      0      0      0      1      0
 x8      0      0      0      0      0      0      0      0      0      0      1
 x9      0      0      0      0      0      0      0      0      0      0      0
 x10     0      0      0      0      0      0      0      0      0      0      0
 x11     0      0      0      0      0      0      0      0      0      0      0
 x12     0      0      0      0      0      0      0      0      0      0      0
```

(a) The A matrix for the linear state space model



```
B =
              u1          u2          u3          u4
 x1            0           0           0           0
 x2            0           0           0           0
 x3            0           0           0           0
 x4            0           0           0           0
 x5       0.4429      0.4429      0.4429      0.4429
 x6            0           0           0           0
 x7            0           0           0           0
 x8            0           0           0           0
 x9            0           0           0           0
 x10   1.661e+04           0  -1.661e+04           0
 x11           0   1.661e+04           0  -1.661e+04
 x12        2215       -2215        2215       -2215
```

(b) The B matrix for the linear state space model



```
C =
      x1   x2   x3   x4   x5   x6   x7   x8   x9  x10  x11  x12
 y1    0    1    0    0    0    0    0    0    0    0    0    0
 y2    0    0    0    1    0    0    0    0    0    0    0    0
 y3    0    0    0    0    0    1    0    0    0    0    0    0
 y4    0    0    0    0    0    0    1    0    0    0    0    0
 y5    0    0    0    0    0    0    0    1    0    0    0    0
 y6    0    0    0    0    0    0    0    0    1    0    0    0
```

(c) The C matrix for the linear state space model

Figure 16: The matrices for the state space model

If we now set up a model in Simulink, which implements the linear model as in figure 17, we can now simulate the linear output.

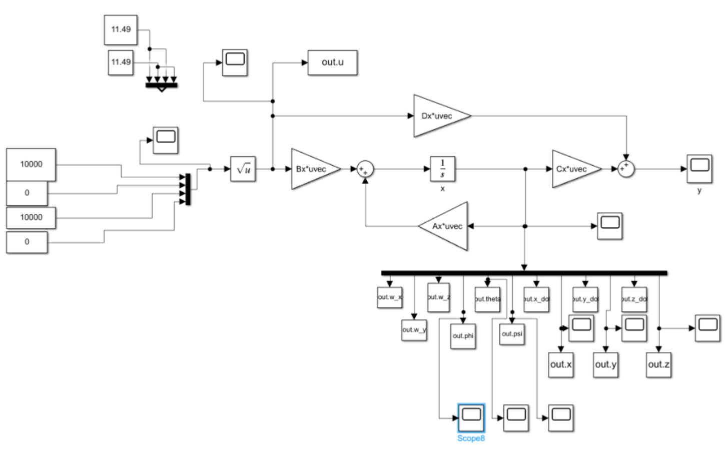


Figure 17: The Simulink model for the linear state space system.

If we simulate the three scenarios where the input to the motors is respectively $\Omega =$

$[0,0,0,0],[10000,0,10000,0] and [0.10000,0,10000]$, we get almost the same results as for the nonlinear model, but where the outputs have been linearized. An example is given by the z change for omega = [10000, 0, 10000, 0], where z changes completely linear instead of exponentially. This is shown in figure 18. Furthermore, the ψ is still changing to illustrate that the drone is rotating around itself. This applies for all the tests, that they show the same as for the non-linear system, but where the outputs have been linearized.

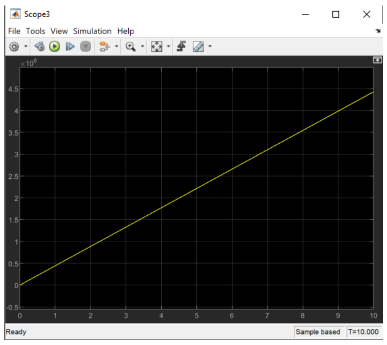


Figure 18: The output for z when the input is $\Omega = [10000, 0, 10000, 0]$

4 Path Planning

The maps shown in the figures in this section represents cities connected by roads. Each node is a city and the edge is a connecting road. The objective is to find the path from city $s0$ to city $s23$.

4.1 Depth First Search

In depth first search algorithm, we start at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

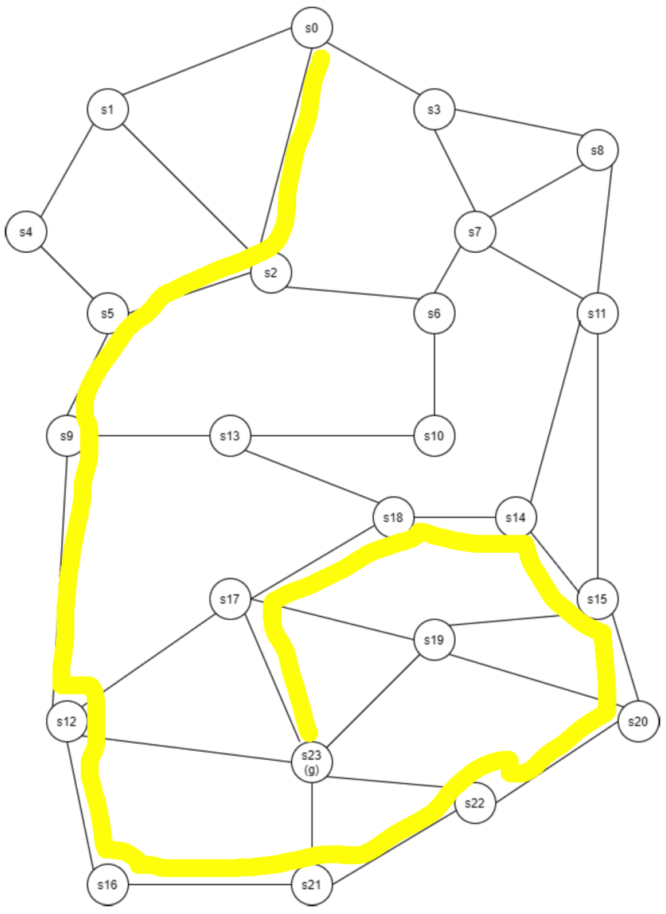


Figure 19: DFS path with stacking lower numbered nodes first

Step	Node Visited	Frontier
1	s_0	s_1, s_2, s_3
2	s_1	s_2, s_4
3	s_2	s_5, s_6
4	s_5	s_9
5	s_9	s_{12}, s_{13}
6	s_{12}	s_{16}, s_{17}
7	s_{12}	$s_{16}, s_{17}, s_{23}(g)$
8	s_{16}	s_{21}
9	s_{21}	$s_{22}, s_{23}(g)$
10	s_{22}	$s_{20}, s_{23}(g)$
11	s_{20}	s_{15}, s_{19}
12	s_{15}	s_{11}, s_{14}, s_{19}
13	s_{11}	s_7, s_8, s_{14}
14	s_7	s_6
15	s_6	s_{10}
16	s_{10}	s_{13}
17	s_{13}	s_{18}
18	s_{10}	s_{13}
19	s_{13}	s_{18}
20	s_{18}	s_{17}
21	s_{17}	$s_{19}, s_{23}(g)$
22	s_{19}	$s_{23}(g)$

Table 1: DFS with stacking lower numbered nodes

Total No. of Expanded Nodes: 22
Total No. of Frontiers Generated: 38
Length of Path: 22

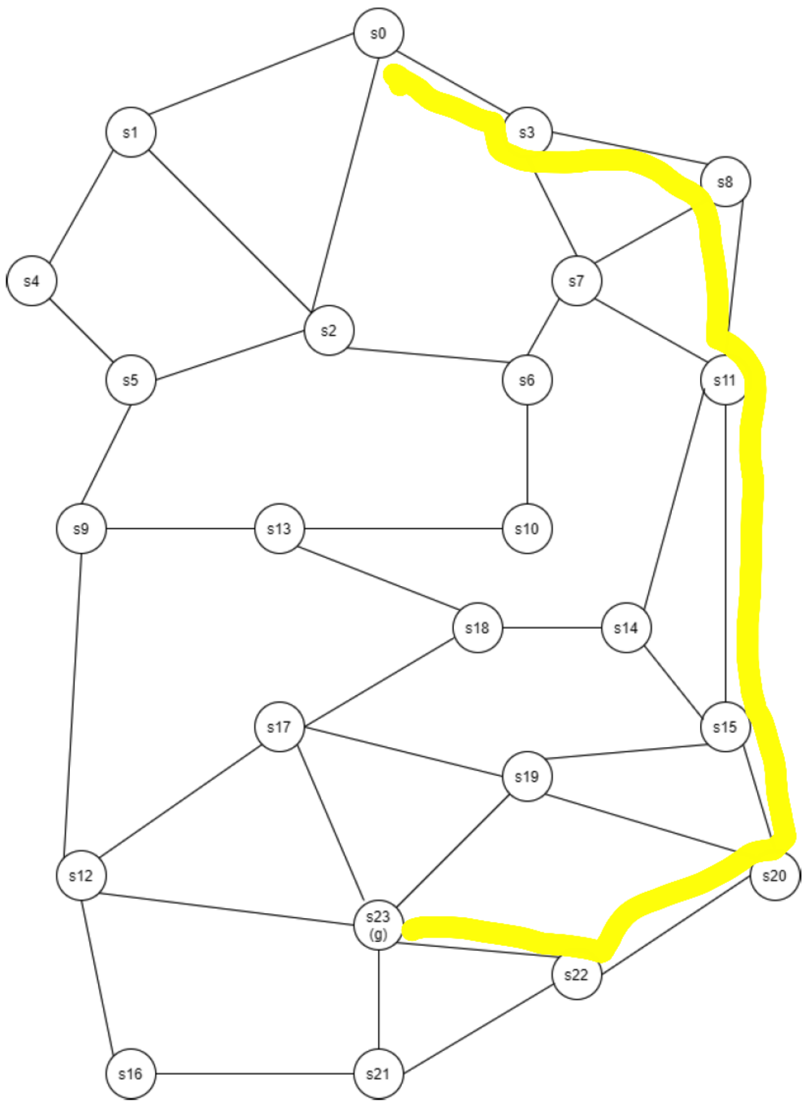


Figure 20: DFS path with stacking higher numbered nodes first

Step	Node Visited	Frontier
1	s_0	s_1, s_2, s_3
2	s_3	s_7, s_8
3	s_8	s_{11}
4	s_{11}	s_{14}, s_{15}
5	s_{15}	s_{19}, s_{20}
6	s_{20}	s_{22}
7	s_{22}	$s_{21}, s_{23}(g)$

Table 2: DFS with stacking higher numbered nodes

Total No. of Expanded Nodes: 7
Total No. of Frontiers Generated: 13
Length of Path: 7

The route generated with stacking the lower number nodes is longer than the one with stacking with the higher number nodes because the DFS explores as far as possible when expanding a node until it backtracks. The goal in this case was easier to reach when the stacking is done with the higher number nodes.

4.2 Breadth First Search

BFS algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

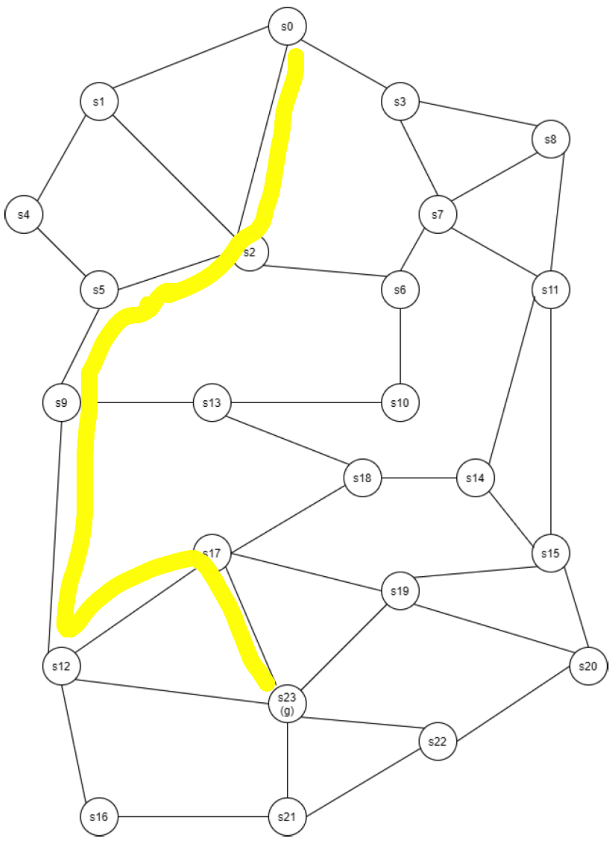


Figure 21: BFS Path with stacking lower numbered nodes first

Step	Node Visited	Frontier
1	s_0	s_1, s_2, s_3
2	s_1	s_2, s_3, s_4
3	s_2	s_3, s_4, s_5, s_6
4	s_3	s_4, s_5, s_6, s_7, s_8
5	s_4	s_5, s_6, s_7, s_8
6	s_5	s_6, s_7, s_8, s_9
7	s_6	s_7, s_8, s_9, s_{10}
8	s_7	s_8, s_9, s_{10}, s_{11}
9	s_8	s_9, s_{10}, s_{11}
10	s_9	$s_{10}, s_{11}, s_{12}, s_{13}$
11	s_{10}	s_{11}, s_{12}, s_{13}
12	s_{11}	$s_{12}, s_{13}, s_{14}, s_{15}$
13	s_{12}	$s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{23}(g)$
14	s_{13}	$s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{23}(g)$
15	s_{14}	$s_{15}, s_{16}, s_{17}, s_{18}, s_{23}(g)$
16	s_{15}	$s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{23}(g)$
17	s_{16}	$s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{23}(g)$
18	s_{17}	$s_{18}, s_{19}, s_{20}, s_{21}, s_{23}(g)$
19	s_{18}	$s_{19}, s_{20}, s_{21}, s_{23}(g)$
20	s_{19}	$s_{20}, s_{21}, s_{23}(g)$
21	s_{20}	$s_{21}, s_{22}, s_{23}(g)$
22	s_{21}	$s_{22}, s_{23}(g)$
22	s_{22}	$s_{23}(g)$

Table 3: BFS with stacking lower numbered nodes

Total No. of Expanded Nodes: 22
Total No. of Frontiers Generated: 92
Length of Path: 7

4.2.1 Advantages of BFS vs DFS

	BFS	DFS
Advantages	The solution is guaranteed if it exists	It has less time and space complexity as compared to BFS
Disadvantages	It might have memory constraints as it stores all the nodes already visited	It does not guarantee that it will give a solution.

4.2.2 What algorithm is preferred for robots?

BFS search algorithms are preferred for robots as it promises a solution if it exists for the particular robot application.

4.3 Dijkstra's Algorithm

Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller.

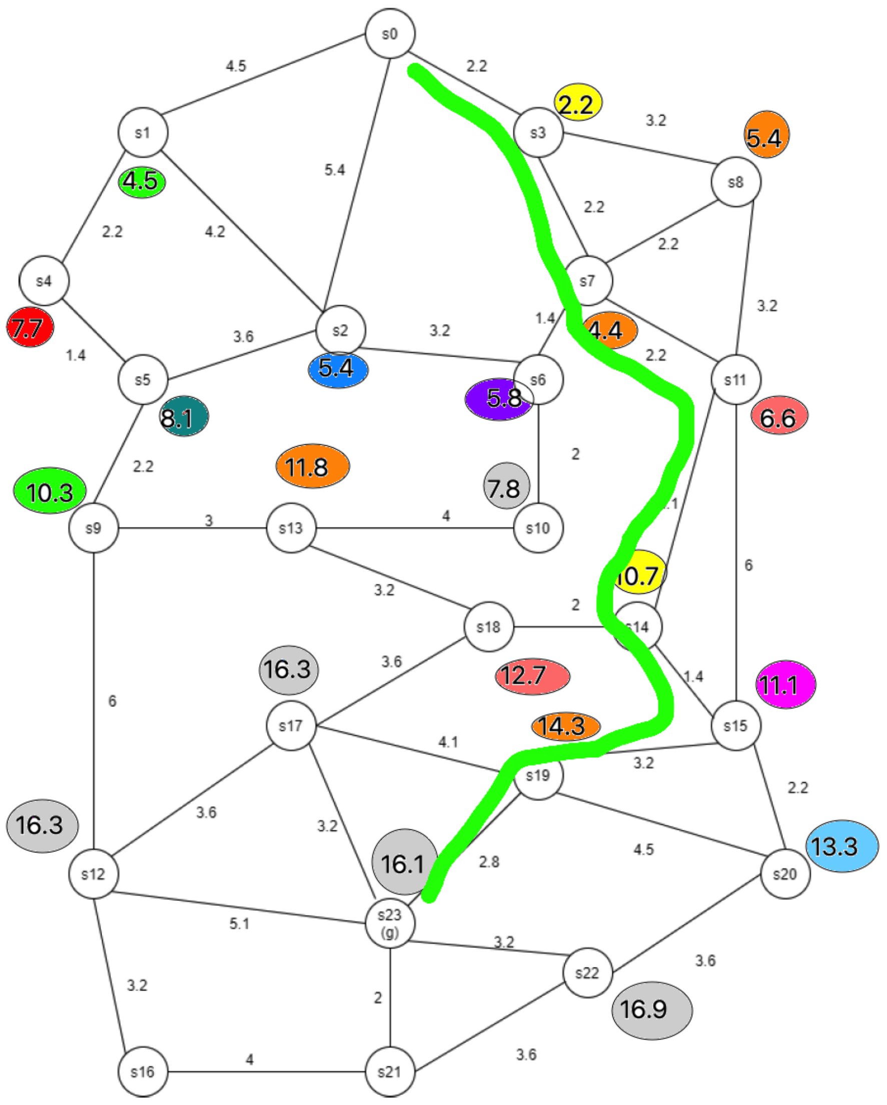


Figure 22: Path with Dijkstra's Algorithm

Step	Node Visited	Distance	Frontier
1	s_0	0	s_1, s_2, s_3
2	s_3	2.2	s_1, s_2, s_7, s_8
3	s_7	4.4	$s_1, s_2, s_6, s_8, s_{11}$
4	s_1	4.5	$s_2, s_4, s_6, s_8, s_{11}$
5	s_2	5.4	$s_4, s_5, s_6, s_8, s_{11}$
6	s_8	5.4	s_4, s_5, s_6, s_{11}
7	s_6	5.8	s_4, s_5, s_{10}, s_{11}
8	s_{11}	6.6	$s_4, s_5, s_{10}, s_{14}, s_{15}$
9	s_4	7.7	$s_5, s_{10}, s_{14}, s_{15}$
10	s_{10}	7.8	$s_5, s_{13}, s_{14}, s_{15}$
11	s_5	8.1	$s_9, s_{13}, s_{14}, s_{15}$
12	s_9	10.3	$s_{12}, s_{13}, s_{14}, s_{15}$
13	s_{14}	10.7	$s_{12}, s_{13}, s_{15}, s_{18}$
14	s_{15}	11.1	$s_{12}, s_{13}, s_{18}, s_{19}, s_{20}$
15	s_{13}	11.8	$s_{12}, s_{18}, s_{19}, s_{20}$
16	s_{18}	12.7	$s_{12}, s_{17}, s_{19}, s_{20}$
17	s_{20}	13.3	$s_{12}, s_{17}, s_{19}, s_{22}$
18	s_{19}	14.3	$s_{12}, s_{17}, s_{22}, s_{23}(g)$
19	s_{19}	16.1	$s_{23}(g)$

Table 4: Dijkstra's Algorithm

Total No. of Expanded Nodes: 19
Total No. of Frontiers Generated: 81
Length of Path: 16.1

4.4 Greedy Best First Search

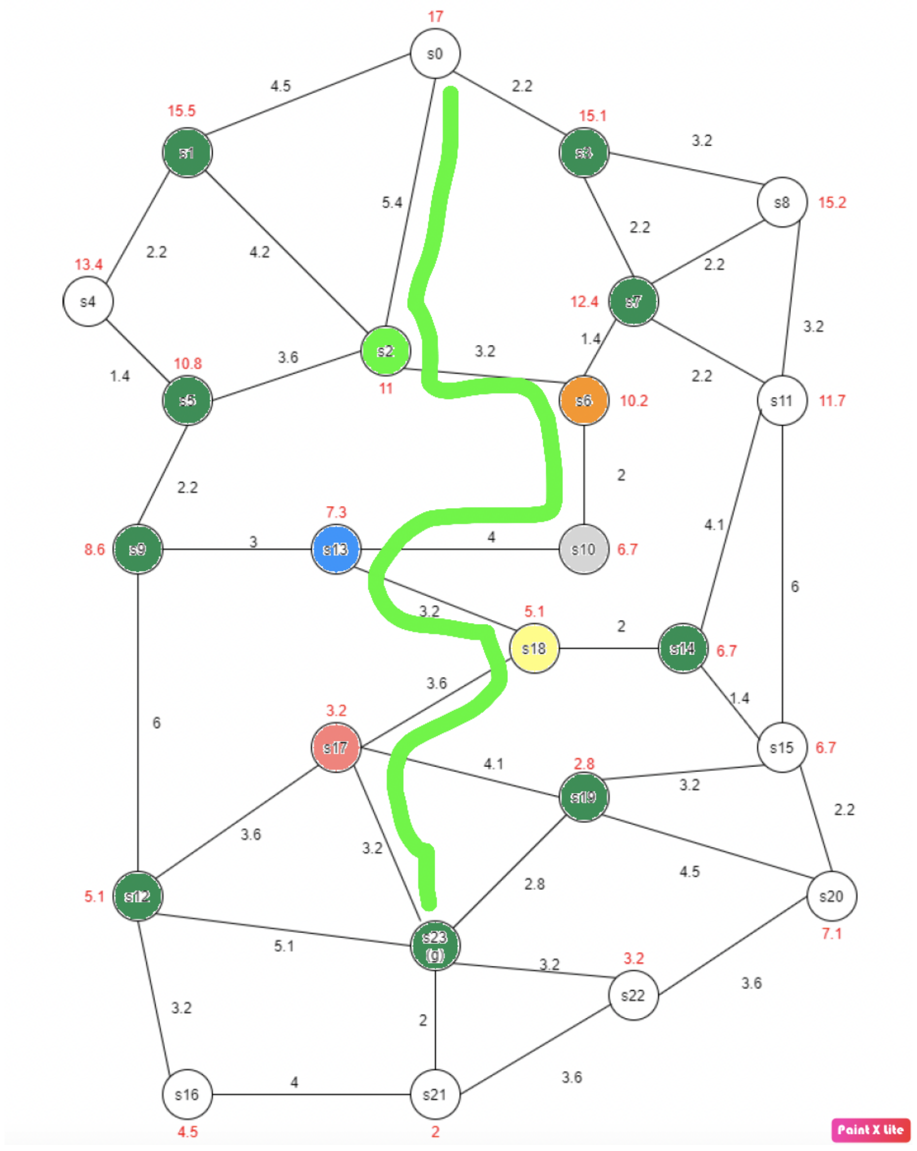


Figure 23: Path Planning with Greedy Search

Step	Node Visited	Heuristic Distance	Frontier
1	s_0	17	s_1, s_2, s_3
2	s_2	11	s_1, s_3, s_5, s_6
3	s_6	10.2	$s_1, s_3, s_5, s_7, s_{10}$
4	s_{10}	6.7	$s_1, s_3, s_5, s_7, s_{13}$
5	s_{13}	7.3	$s_1, s_3, s_5, s_7, s_9, s_{18}$
6	s_{18}	5.1	$s_1, s_3, s_5, s_7, s_9, s_{14}, s_{17}$
7	s_{17}	3.2	$s_1, s_3, s_5, s_7, s_{14}, s_{19}, s_{23}(g)$
8	$s_{23}(g)$	0	

Table 5: Greedy Best First Search

Total No. of Expanded Nodes: 8
Total No. of Frontiers Generated: 36
Length of Path: 24.6

4.5 A* Search

The A* algorithm only finds the shortest path from a specified source to a specified goal, and not the shortest-path tree from a specified source to all possible goals.

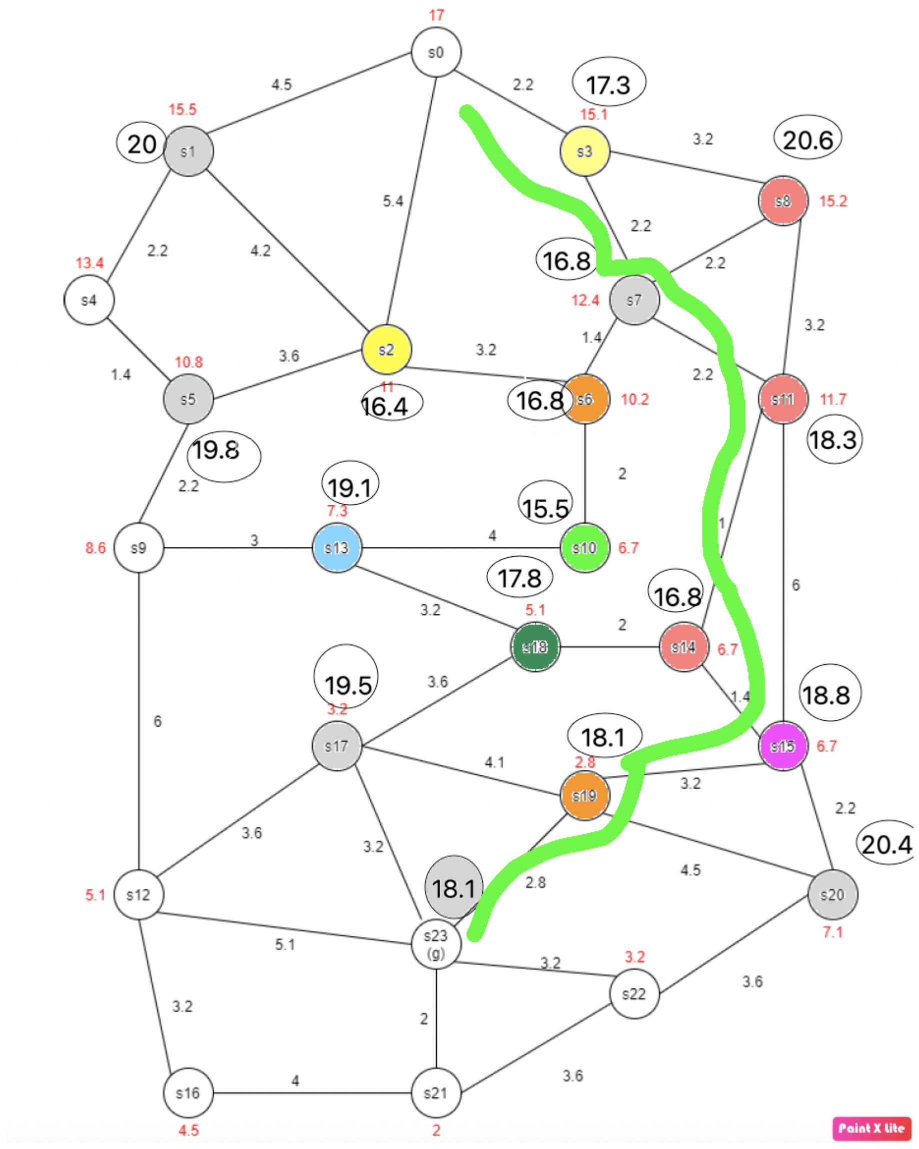


Figure 24: Path with A* Search

Step	Node Visited	Heuristic Distance	Frontier
1	s_0	17	s_1, s_2, s_3
2	s_2	16.4	s_1, s_5, s_6, s_3
3	s_3	17.3	s_1, s_5, s_6, s_7, s_8
4	s_7	16.8	$s_1, s_5, s_6, s_8, s_{11}$
5	s_{11}	18.3	$s_1, s_5, s_6, s_8, s_{14}, s_{15}$
6	s_{14}	16.8	$s_1, s_5, s_6, s_8, s_{15}, s_{18}$
7	s_{18}	17.8	$s_1, s_5, s_6, s_8, s_{15}, s_{17}$
8	s_{15}	18.3	$s_1, s_5, s_6, s_8, s_{15}, s_{19}$
9	s_{19}	17.9	$s_1, s_5, s_6, s_8, s_{15}, s_{23}(g)$
10	$s_{23}(g)$	0	

Table 6: A* Search

Total No. of Expanded Nodes: 10
Total No. of Frontiers Generated: 47
Length of Path: 16.1

Advantages of Using A* over Dijkstra

A* tries to look for a better path by using a heuristic function which gives priority to nodes that are supposed to be better than others while Dijkstra's just explore all possible paths. Thus A* is faster and memory efficient compared to Dijkstra.

4.6 Advantages and Disadvantages of Greedy best-first search and A* search algorithms

1. What are the advantages and disadvantages of Greedy best first search algorithm?
Advantages:

- Greedy BFS can reach the goal faster if it finds the correct path, i.e it is similar to the DFS.
- In the presence of a heuristic function it has an advantage over DFS.

Disadvantages:

- The algorithm is not complete neither it is optimal.
- It may not terminate at all and can get stuck in loops.

2. What are the advantages and disadvantages of A* search algorithm?

Advantages:

- A* algorithm can find the shortest path between the start and the goal state with the help of heuristics.

- It is optimal in terms of heuristics.

Disadvantages:

- The performance of the algorithm is highly dependant on the heuristic chosen.
- The algorithm is complete only if it has a fixed cost.

3. Which algorithm is best for aerial robots and why?

A* algorithm should be the best suited for aerial robotics as we can alter the heuristic function according to the use case and be optimal for that particular application.

4.7 Solving 3D Maze with Greedy best-first Search

After upgrading the *map_script_3d.m* file and the *greedy_2d* function to be able to work for a 3D environment, the following 3D map and path is generated.

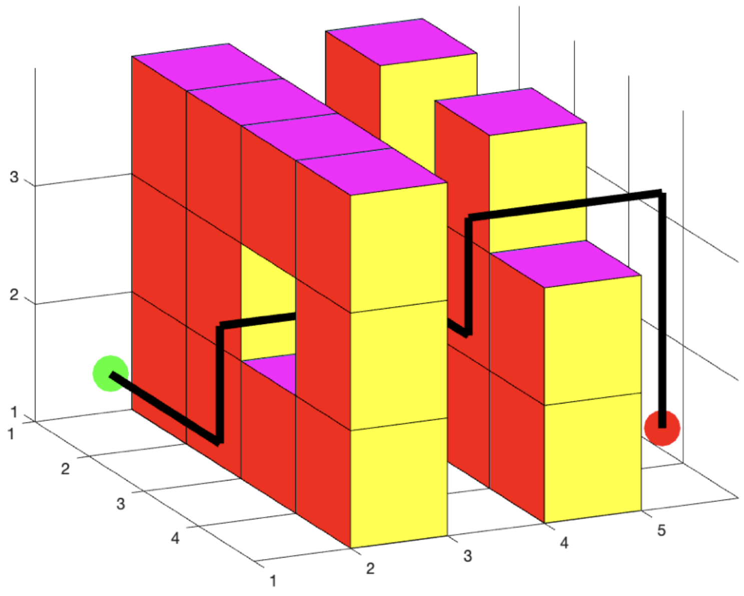


Figure 25: Maze Navigation with Greedy Best First

4.8 Solving 3D Maze with A* Search

In order to run the A* algorithm, the *greedy_3d.m* file is modified to change the heuristic from $f(n) = h(n)$as $f(n) = h(n) + g(n)$, where $h(n)$ is the distance to goal and $g(n)$ is the distance travelled.

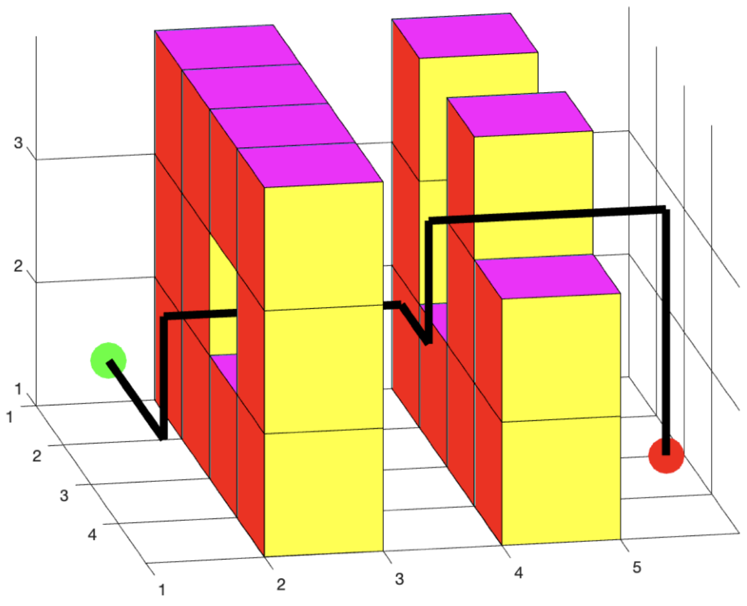


Figure 26: Maze Navigation with A* Search

4.8.1 Additional Remarks

The following running times were observed for the two algorithms on the same maze:
A* Search: Elapsed time is 0.065284 seconds.
Greedy Search: Elapsed time is 0.093074 seconds.

What algorithm would be implemented if the formula was f(n) = g(n) instead?
Djikstra

5 Trajectory Generation

5.1 Quintic Splines

Consider the following 1-D quintic spline:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a3t3 + a4t4 + a5t5. \tag{22}$$

Find the constants ai, i = 0, 1, 2, 3, 4, 5, to respect the constraints $x(0) = 0, \dot{x}(0) = 0, \ddot{x}(0) = 0, x(1) = 1, \ddot{x}(1) = 0, \ddot{x}(1) = 0$.

```

syms a0 a1 a2 a3 a4 a5 t;
x = @(t) a0 + a1*t +  a2*t^2 + a3*t^3 + a4*t^4 + a5*t^5
```

```
xdot = diff(x(t))
xddot = diff(xdot)

x_0 = x(0)
xdot_0 = subs(xdot,0)
xddot_0 = subs(xddot,0)

x_1 = x(1)
xdot_1 = subs(xdot,1)
xddot_1 = subs(xddot,1)

A0 = solve(x_0==0)
A1 = solve(xdot_0==0)
A2 = solve(xddot_0==0)
sol = solve([x_1==1,xdot_1==0,xddot_1==0],[a3,a4,a5]);
a3 = subs(sol.a3,{a0,a1,a2},{A0,A1,A2})
a4 = subs(sol.a4,{a0,a1,a2},{A0,A1,A2})
a5 = subs(sol.a5,{a0,a1,a2},{A0,A1,A2})
```

The above MATLAB script differentiates the equation twice to get $\dot{x}$ and $\ddot{x}$ and substitute the values as given, getting//

$$a_0 = 0, a_1 = 0, a_2 = 0, a_3 = 10, a_4 = -15, a_5 = 6 \tag{23}$$

5.2 B-splines

The *Cox-de Boor recursion formula* is given by:

$$N_{i,0}(t) = \begin{cases} 1, & \text{if } t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$
$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t) \tag{24}$$

As $t_i = i$, so

$$t_0 = 0, t_1 = 1, t_2 = 2 \tag{25}$$

Evaluating $N_{0,1}, N_{1,1}$ and $N_{0,2}$,

$$\begin{aligned} N_{0,1}(t) &= t.N_{0,0}(t) + (2 - t)N_{1,0}(t) \\ N_{1,1}(t) &= (t - 1)N_{1,0}(t) + (3 - t)N_{2,0}(t) \\ N_{0,2}(t) &= \frac{t}{2} N_{0,1}(t) + \frac{3 - t}{2} N_{1,1}(t) \end{aligned} \tag{26}$$

If $1 \leq t < 2$,

$$\begin{aligned} N_{0,2}(t) &= \frac{t}{2}(2-t) + \frac{(3-t)(3-t)}{2} \\ N_{0,2}(t) &= \frac{1}{2}(-3+6t-2t^2) \end{aligned} \tag{27}$$

If $2 \leq t < 3$,

$$\begin{aligned} N_{0,2}(t) &= \frac{1}{2}(3-t)(3-t) \\ N_{0,2}(t) &= \frac{1}{2}(3-t)^2 \end{aligned} \tag{28}$$

From equation 27 and 29,

$$\begin{aligned} \frac{1}{2}(-3+6t-2t^2) &= \frac{1}{2}(3-t)^2 \\ (t-2)^2 &= 0 \\ t &= 2 \end{aligned} \tag{29}$$

5.3 Computation of a DCM

A Direction Cosine Matrix (DCM) is found knowing the upward direction of the drone z_B and the yaw angle ψ.

For the Z-Y-X form, $R_B = R_z(\psi)R_y(\theta)R_x(\phi)$ with the roll, pitch and yaw angles ϕ, θ and ψ respectively.

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} R_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} R_z(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{30}$$

$$z_B = R_z(\psi).R_y(\theta).R_x(\phi). \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \& = \begin{bmatrix} sin(\phi).sin(\psi) + cos(\phi).cos(\psi).sin(\theta) \\ cos(\phi).sin(psi).sin(\theta) - cos(\psi).sin(\phi) \\ cos(\phi).cos(\theta)] \end{bmatrix} \tag{31}$$

$$x_C = \begin{bmatrix} cos(\psi) \\ sin(\psi) \\ 0 \end{bmatrix} \tag{32}$$

So, y_B can be calculated as follows:

$$y_B = z_B \times x_C \tag{33}$$

Also,

$$x_B = y_B \times z_B \tag{34}$$

Finally,

$$R_B = \begin{bmatrix} x_B & y_B & z_B \end{bmatrix} \tag{35}$$

```

syms phi theta psi
Rx_phi = [1 0 0;
          0 cos(phi) -sin(phi)
          0 sin(phi) cos(phi)];
Ry_theta = [cos(theta) 0 sin(theta)
            0 1 0
           -sin(theta) 0 cos(theta)];
Rz_psi = [cos(psi) -sin(psi) 0
          sin(psi) cos(psi) 0
          0 0 1];

zB = Rz_psi*Ry_theta*Rx_phi*[0;0;1]
xC = [cos(psi); sin(psi); 0]
yB = cross(zB, xC)
xB = cross(yB,zB)

R_B = [xB yB zB]
```

6 Simulation

6.1 Ex.6.1 - Navigate a 2D maze

To generate a route for the drone to follow, the script from Exercise 4 was used. We chose to use A* although we could have used Dijkstra as well, since both are guaranteed to find to optimal route (Given that A* heuristic is admissible).

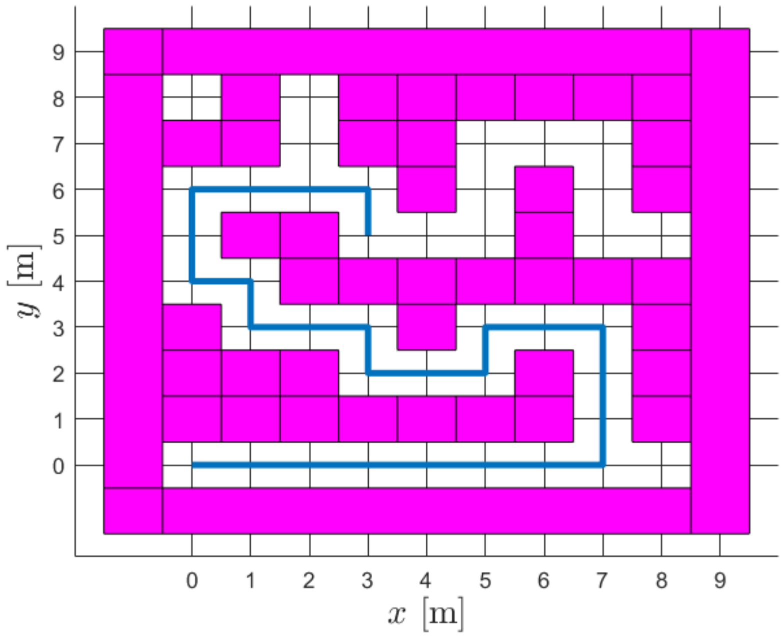


Figure 27: Plotted route generated

Figure 27 shows the route generated by the path finding algorithm.

6.2 Ex.6.2 - Re-implementing the Position Controller

The Position Controller we want to implement is based on the one described in the article "Control of Complex Maneuvers for a Quadrotor UAV using Geometric Methods" [1]. In the predefined system we are given setpoint for velocity, position and acceleration. Furthermore, we are also given the drones position velocity and yaw as input for our control block. The controller is structured as shown in Figure 28

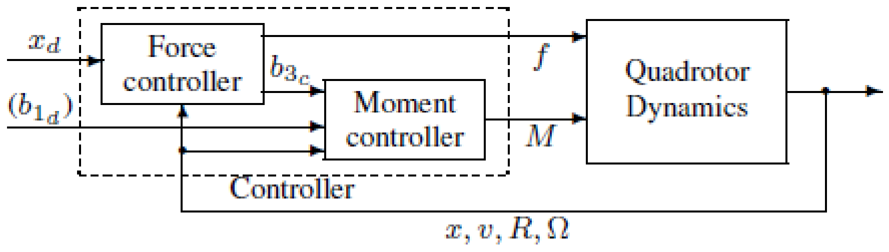


Figure 28: Position Controller based on the article [1]

The equation for finding f is given in the article as:

$$f = (k_x(x - x_d) + k_v(v - \dot{x}_d) + mge_3 - m\ddot{x}_d) \cdot Re_3 \tag{36}$$

where k_x and k_v are constants, $x - x_d$ are current position minus the wanted position as x,y,z vectors and $v - \dot{x}_d$ are the velocity of the drone minus the set velocity. mge_3 is mass of the drone times gravity times the error of the 3rd element of the vector (Aka the Z-axis which is also the thrust).

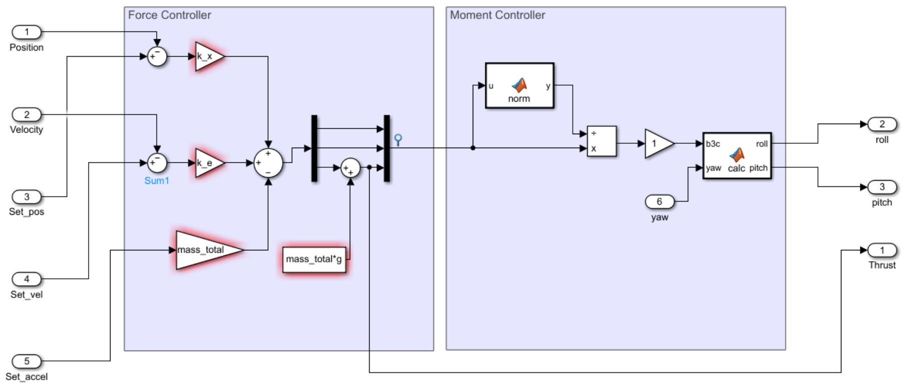
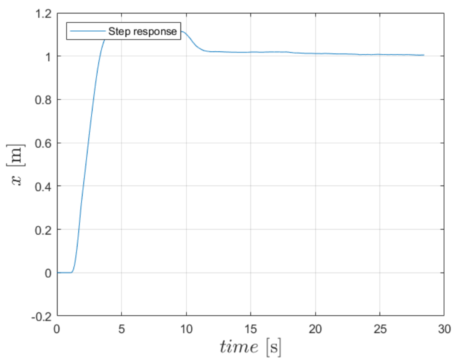
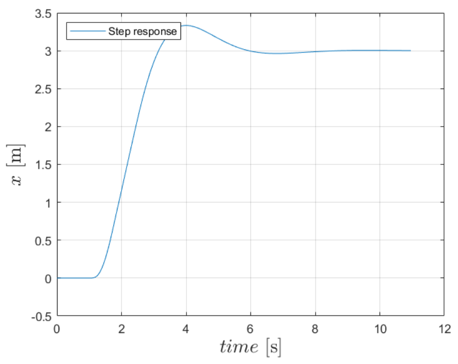


Figure 29: Position Controller

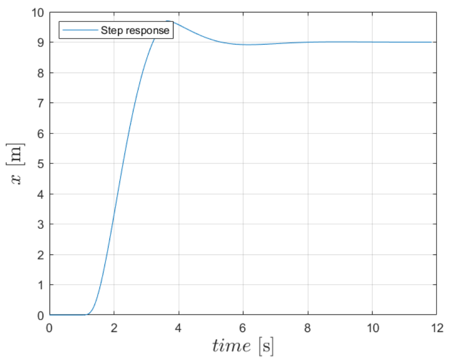
Next we can compute b_{3_c} by dividing the result of f with the norm of f. We can then use this to calculate pitch and roll.



(a) Step response for 1 m



(b) Step response for 3 m



(c) Step response for 9 m

Figure 30: Step response for the position controller

Figure 30 show the step response of the controller. Note that the two constants were kept at 1 and not tuned. As seen, the controller overshoots a bit but then goes back and stabilizes around the desired value.

6.3 Ex.6.3 - Re-implementing the Attitude Controller

The Attitude Controller we want to implement is based on the one described in the article "Control of Complex Maneuvers for a Quadrotor UAV using Geometric Methods" [1]. In the predefined system we are given set-point for roll, pitch and yaw.
The design of the attitude controller begins with the definition of the errors associated with the attitude dynamics of the quadrotor UAV. The attitude tracking error is defined as:

$$e_R = \frac{1}{2}(R_d^T R - R^T R_d)^V \tag{37}$$

and the tracking error for angular velocity is defined as:

$$e_\Omega = \Omega - R^T R_d \Omega_d \tag{38}$$

Here, an arbitrary smooth attitude tracking command $R_d(t) \; \epsilon \; \mathrm{SO}(3)$ is given as a function of time and represents the set-point yaw, pitch and roll, while R represents the Euler representation of actual roll, pitch and yaw derived from the quartenion output. The non-linear controller for the attitude controlled flight mode is described by an expression for the moment vector as:

$$M = -k_R e_R - k_\Omega e_\Omega \tag{39}$$

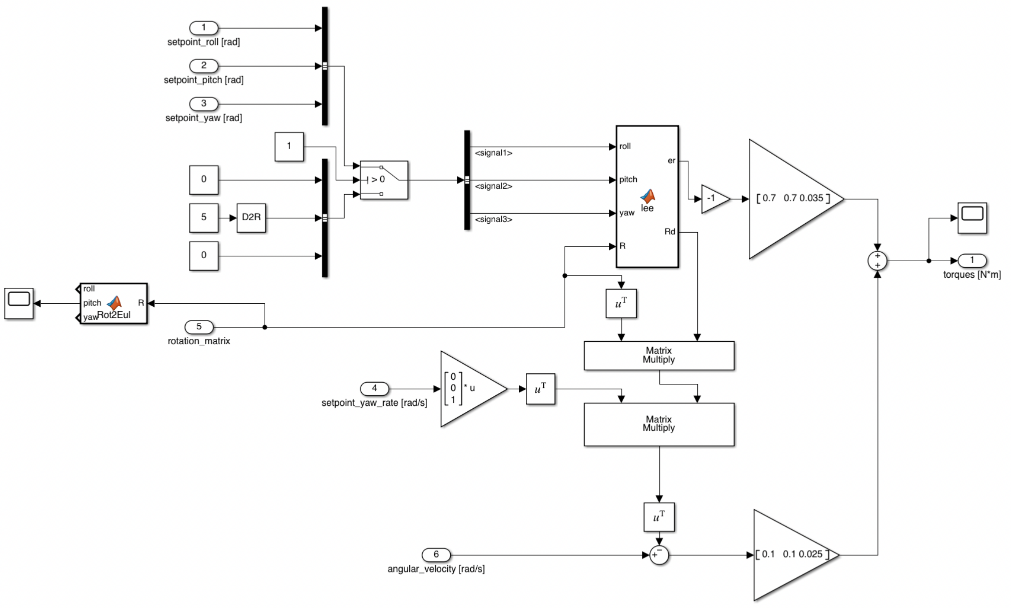
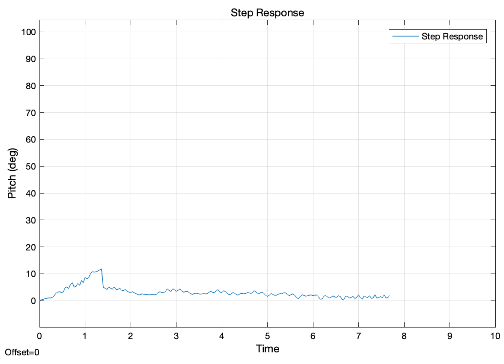
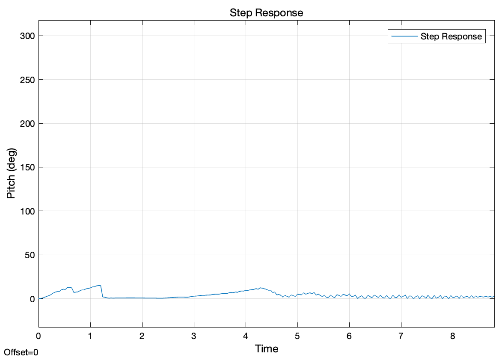


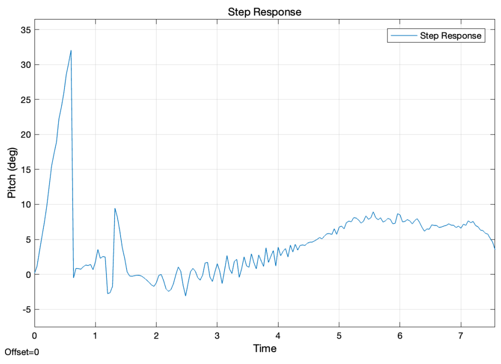
Figure 31: Attitude Controller based on the article [1]



(a) Step response for 5 deg fwd pitch



(b) Step response for 10 deg fwd pitch



(c) Step response for 15 deg fwd pitch

Figure 32: Step response for the attitude controller

Figure 32 shows the step response of the controller. Note that the not much time was spent on tuning the constants for the error gains. As seen, the controller overshoots and goes back to the setpoint value of 5 degree in the first case, although it isn't quite stable. The responses obtained with 10 degree and 15 degree input for forward pitch were also not satisfactory. Further investigation is required and perhaps a more efficient design using a PID controller would be required for a stable response. However, we had focused on the implementation of position controller and relied on the onboard attitude controller in the CrazyFlie for the actual experiments due to the time constraints.

6.4 Ex.6.4 - Aggressively Navigating a 2D Maze

The objective of this exercise was to navigate a maze and go from (0,0,1) to (9,9,1) in under 5 seconds. To aggressively navigate the maze we ended up using the path finding from Section 6.1 to generate the shortest route. Then we took each point and basically turned it into a corridor (Technically a cube). For finding out at what time the drone needed to be at each corridor, we simply just divided 4 seconds by the number of waypoints. This then gave us the times for each corridor. We then just fed all these corridors to the *uas_minimum_snap* function to generate the final trajectory.

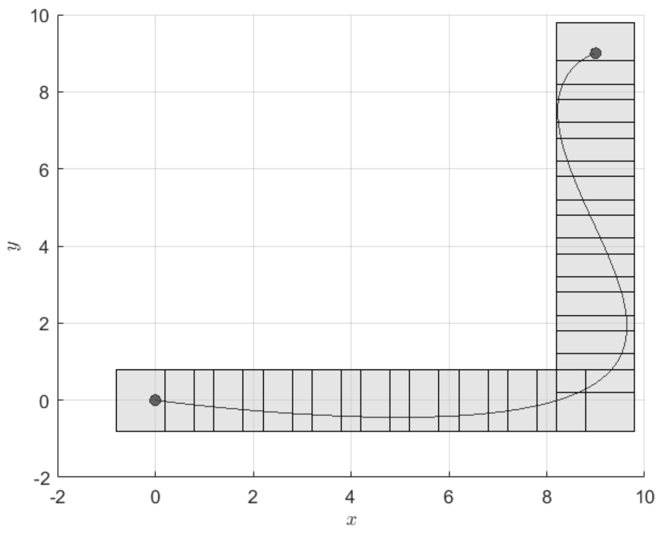


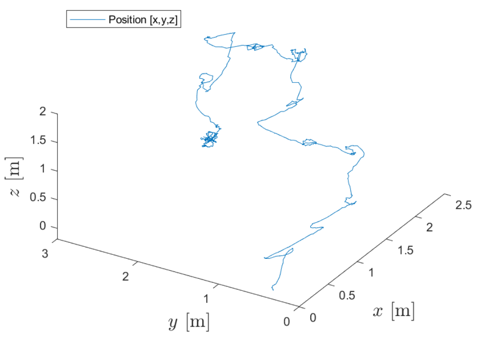
Figure 33: Generated trajectory

This gives us the trajectory shown in Figure 33. The drone is able to finish this route in around 4 seconds.

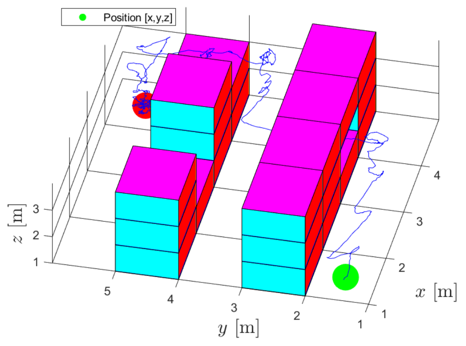
7 PART 7: Demonstration

7.1 Ex.7.1 - Navigating a 3D maze

Likewise as in Section 6.1, this exercise uses A* to calculate the shortest route. To make the drone fly to the generated waypoints, the built in position controller was used.



(a) The position of the drone when flying the path



(b) Step response for 3 m

Figure 34: The flown route, which is scaled to fit in the maze-plot

As seen in Figure 34 the drone is quite unstable. A way to fix this could be to try and tune the controller better. However, in this exercises we did not have access to the controller.

Instead we tried to interpolate points between the points generated by A*, thus doubling the amount of points. This made the drone a bit more stable, but not by much so we ended up not using it anyway.

7.2 Ex.7.2 - Porting the position controller to the real system

First of all this exercise handle the question to connect the position controller.This implies that we need to determine the relation between the thrust and the acceleration in the vertical direction (z). By setting up and thereby using a joystick to control the drone for keeping it flying for a certain amount of time. This way it was possible to store the thrust values and the acceleration data from the IMU (inertial measurement unit). In Figure 35 the collected data is plotted as the acceleration in the z -direction and the thrust against time respectively.

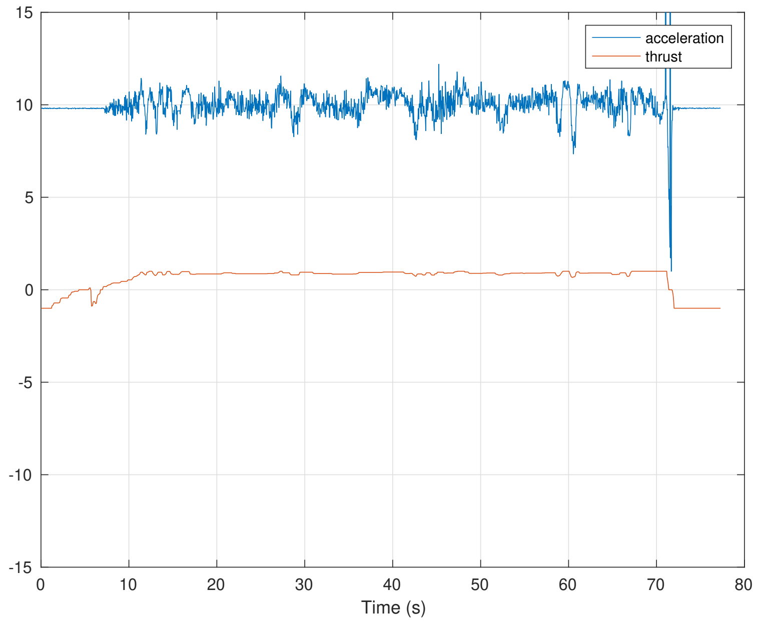


Figure 35: The diagram of the pwm position controller

If we instead are plotting the data for trust and acceleration against each other, we can set up a linear model, which can later be used to describe the pwm signal from the thrust value. Figure 36 describes the relationship between the acceleration on the y axis and the thrust on the x axis as well as the fitted graph of the linear model. the linear model is achieved by using the Matlab function "fit" and given by the constants $p1 = 1.18 * 10^{(-4)}$ and $p2 = 0.737$.

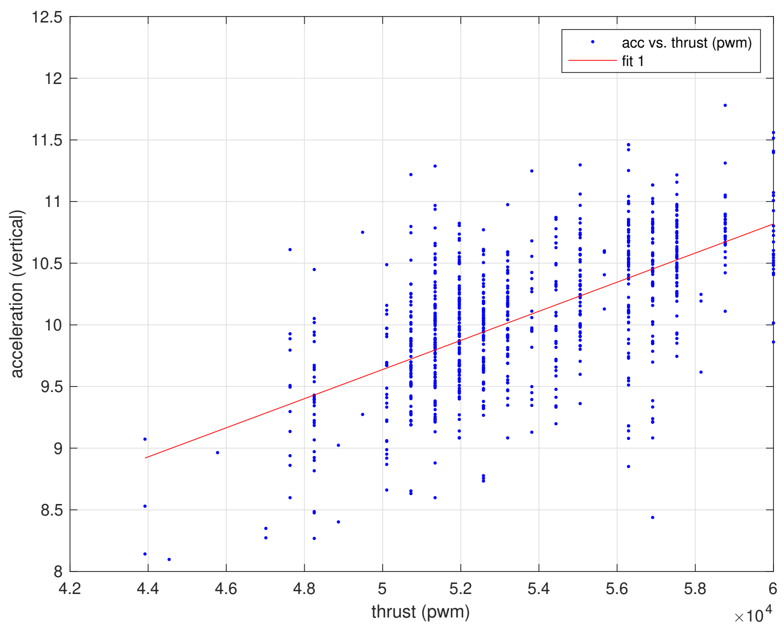


Figure 36: The diagram of the pwm position controller controller

Now the task is to use the controller from task 6.2 or the default one as well as adding our linear model for conversion to pwm. These circuits are shown in Figure 37 and 38.

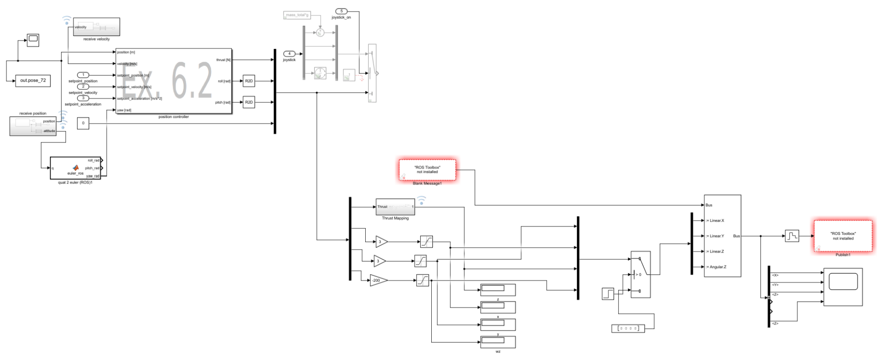


Figure 37: The diagram of the pwm position controller controller

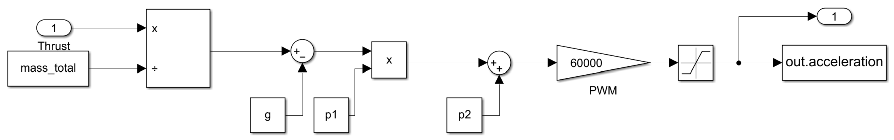


Figure 38: The linear model for converting the thrust into a pwm signal.

During the first few tests, the drone did not fly as desired, first and foremost, this was due to the drone controls its orientation too aggressively, which is why we fine tuned the gain for x and y to be lower. Furthermore, the drone did not reach the desired height, so we tuned the p1 and p2 values for the linear model.

From the first tests we could see that we wanted to achieve a slightly larger slope to achieve the right converting between different heights, therefore p1 was modified to $p1 = 0.1$, in addition we wanted to achieve a bit more trust in general to reach the exact set position, and thereby changed p2 to be $p2 = 0.85$.

Due to the Optitrack system being off when we did our testing we instead of point [1; 1; 1], which was very close to the pool in Asta, that we used another point, which we wanted the drone to be tested in. Point [1.55; -2.65; 1] was used instead of accommodating the offset and thereby preventing the drone from ending up in the pool.

Figure 39 illustrates the flown route, where it is clear to see that the drone is going up to the witched position and keeps that position in more than 15 seconds. Afterward the drone goes 1m along the x-axis as the original exercise ask for.

If, we on the other hand,are looking at the error, it is clear that the drone keeps its position within the desired margin which is 10 cm for more than 10 seconds. Furthermore, it must be said that the error does not grow to one for the x axis when the drone is moved one meter on the x axis. This is because we keep the same set-point to make the plot clear when the drone flies one meter off the x-axis. It is therefore seen that the error is still less than the requirement and that the drone stays within the desired range. furthermore, it is seen that the drone obtains an overshoot in the x direction of less than 20 cm, which in turn is within the requirement of 30cm.

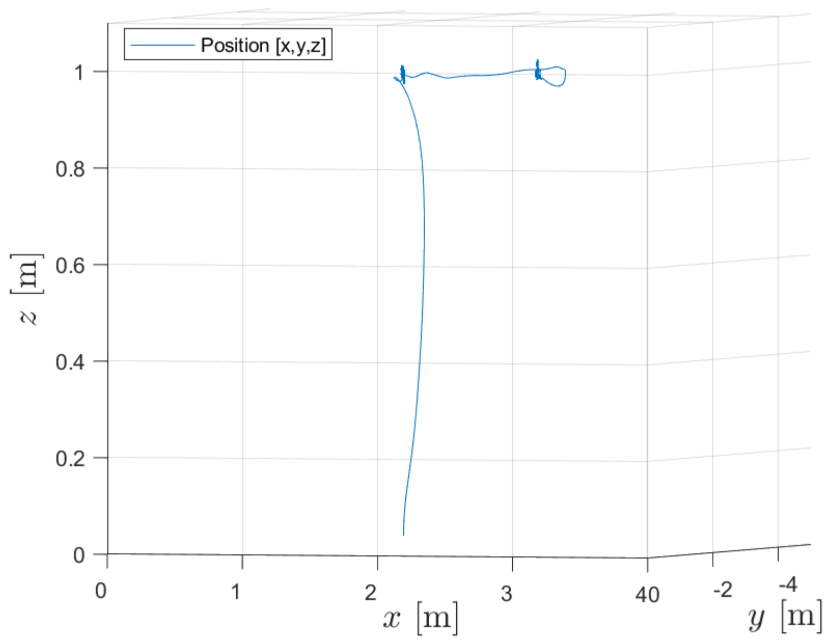


Figure 39: The flown route, where the drone is standing at the same position with the z coordinate of 1m and then move the drone 1m at x- direction too.

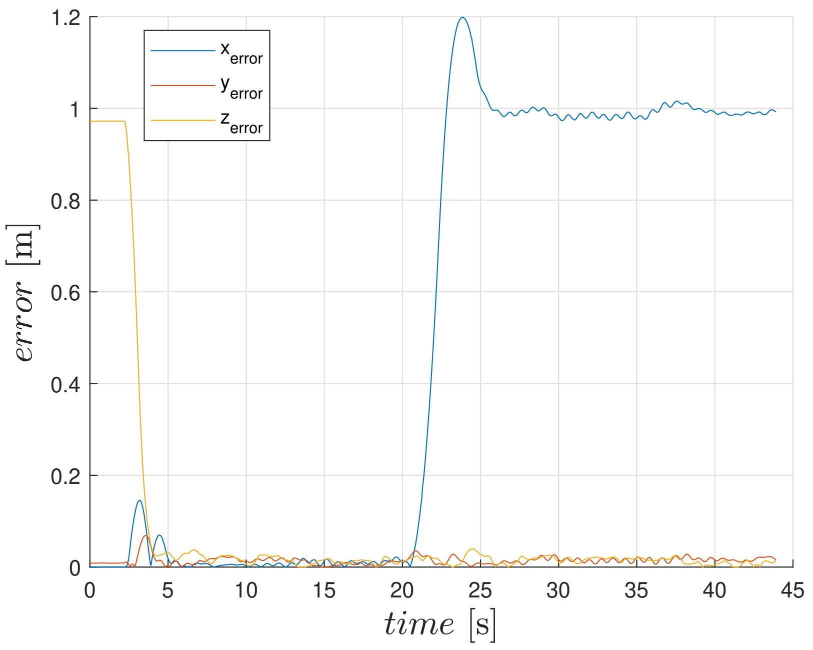


Figure 40: The error for the 3 directions (x,y,z)

7.3 Ex.7.3 - Aggressively navigating through hoops

For exercise 7.3 we use the same method as for exercise 6.4. The only difference is that we also get the position of the hoops from the Optitrack system. The way that we create the 8 corridors shown in Figure 41 is that for the one in the hoops be basically just take the hoop position and create a tiny cube in the middle. For the bigger cubes in the corners we take the two closest hoops and use the x coordinate for the first hoop and the y coordinate from the second hoop. That way we can create a corner between all the hoops. The reason why we create corners is that when we were using the Optitrack, somebody had broken the orientation unit vector that each hoop was supposed to have. We therefor made the assumption that hoop 1 and 3 would line up with the x-axis and hoop 2 and 4 would line up with the y-axis.
Had the orientation unit vector been working, it would be quite simple to take that into account. The procedure is basically that you create a cube in the middle of the hoop as usual, then you take that cube and shift it by adding the orientation unit vector given from optitrack. You then do this again but this time subtract it instead. This creates a cube in front of the hoop, in the hoop and behind the hoop, thus ensuring that the drone flies through the hoop correctly.

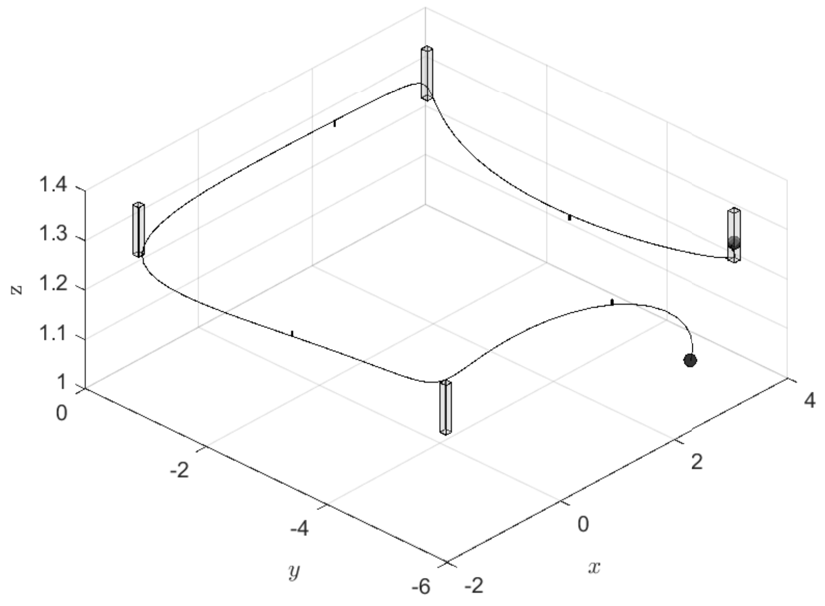


Figure 41: The generated corridors and the final trajectory.

As shown in Figure 42 we had a run where we were able to successfully go through all the hoops. Although, it wasn't very robust. It seemed like that we sometimes lost tracking of the drone, which meant that it started to behave weird.

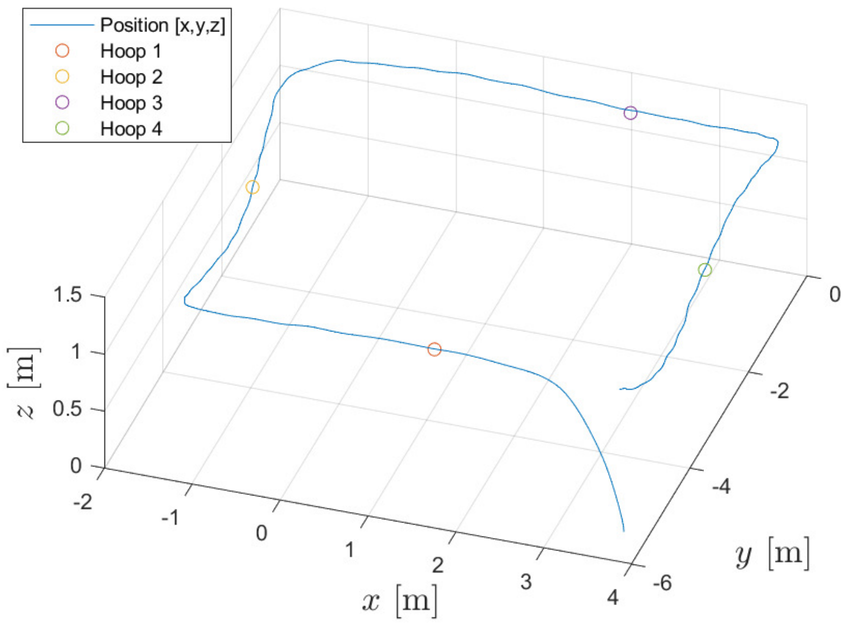


Figure 42: The flown route through the hoops.

7.4 Error Optimization - Hopefully helpful for future groups

For all pratical exercises we kept running into the error *crazyflie runtime error could not find in log toc.* It turned out that the fix for this was to simply delete the log files and try again. The easiest way to do this is to simply clear the logs each time by using the following command to start the drone: *$ rm -rf <logdir>; cfstart <drone_id>*
Furthermore, we found out that it seems to be a connection issue, so moving the drone close to the transmitter seemed to reduce the probability of the error occuring.

List of Figures

1 calculations for exercise 1.6.1 4
2 calculations for exercise 1.6.2 4
3 calculations for exercise 1.6.3 5
4 calculations for exercise 1.6.4 6
5 Rotation matrix representing the orientation of the body-fixed frame 6
6 Relation between $\dot{\theta}$ and ω of the body-fixed frame 7
7 Dynamics given in the lecture 7
8 Final dynamic equation describing the drone 7
9 The overall model for the drone system 8
10 The dynamic equation for the drone 9
11 conversion of omega arrow (inverse of omega arrow) - for transforming omega_x,y,z to (phi, theta, psi) 10
12 linear dynamic for the drone model 11
13 Plots of the behavior of P and Θ, when $\Omega = [0, 0, 0]^T$ 12
14 Plots of the behavior of P and Θ, when $\Omega = [10000, 0, 10000, 0]^T$ 13
15 Plots of the behavior of P and Θ, when $\Omega = [0, 10000, 0, 10000]^T$ 14
16 The matrices for the state space model 15
17 The Simulink model for the linear state space system. 15
18 The output for z when the input is $\Omega = [10000, 0, 10000, 0]$ 16
19 DFS path with stacking lower numbered nodes first 17
20 DFS path with stacking higher numbered nodes first 19
21 BFS Path with stacking lower numbered nodes first 20
22 Path with Dijkstra's Algorithm 22
23 Path Planning with Greedy Search 24
24 Path with A* Search 26
25 Maze Navigation with Greedy Best First 28
26 Maze Navigation with A* Search 29
27 Plotted route generated 33
28 Position Controller based on the article [1] 34
29 Position Controller 34
30 Step response for the position controller 35
31 Attitude Controller based on the article [1] 36
32 Step response for the attitude controller 37
33 Generated trajectory 38
34 The flown route, which is scaled to fit in the maze-plot 38
35 The diagram of the pwm position controller 39
36 The diagram of the pwm position controller controller 40
37 The diagram of the pwm position controller controller 40
38 The linear model for converting the thrust into a pwm signal. 40
39 The flown route, where the drone is standing at the same position with the z coordinate of 1m and then move the drone 1m at x- direction too. 41
40 The error for the 3 directions (x,y,z) 42

41 The generated corridors and the final trajectory. 43
42 The flown route through the hoops. 43

List of Tables

1 DFS with stacking lower numbered nodes 18
2 DFS with stacking higher numbered nodes 19
3 BFS with stacking lower numbered nodes 21
4 Dijkstra's Algorithm 23
5 Greedy Best First Search 25
6 A* Search 27

References

[1] T. Lee, M. Leok, and N. H. McClamroch, "Control of complex maneuvers for a quadrotor uav using geometric methods on se (3)," *arXiv preprint arXiv:1003.2005*, 2010.

8 Appendix A