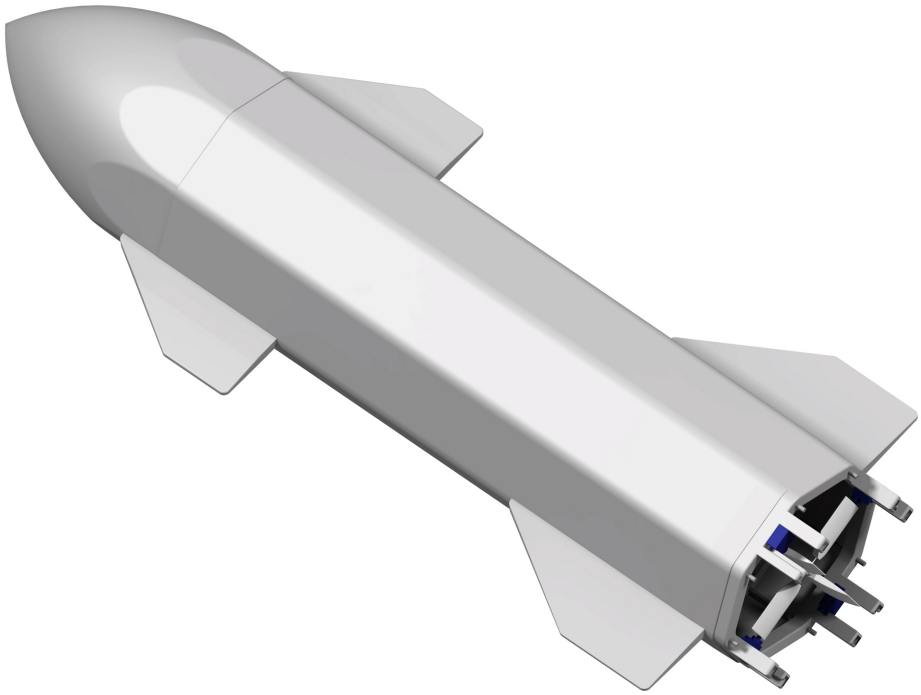
CHALMERS



Modellering och reglering av eldriven raket

För autonom uppskjutning, flygning och landning

Kandidatarbete vid Avdelningen för System- och Reglerteknik

INSTITUTIONEN FÖR ELEKTROTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2023
www.chalmers.se

Kandidatarbete 2023

Modellering och reglering av eldriven raket

För autonom uppskjutning, flygning och landning

Wilhelm Dahlin
Eric Fabricius
Gideon Geelnard
Anton Gleisner
David Klerebladh
Sofia Källhammer

CHALMERS

Institutionen för Elektroteknik
Chalmers Tekniska Högskola
Göteborg 2023

Modellering och reglering av eldriven raket
För autonom uppskjutning, flygning och landning

WILHELM DAHLIN
ERIC FABRICIUS
GIDEON GEELNARD
ANTON GLEISNER
DAVID KLEREBLADH
SOFIA KÄLLHAMMER

Handledare: Jonas Sjöberg, Chalmers Tekniska Högskola
Examinator: Jonas Fredriksson, Chalmers Tekniska Högskola

Examensarbete 2023
Institutionen för Elektroteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: CAD-modell av raketen som har konstruerats under projektets gång.

Skriven i LaTeX
Göteborg 2023

Abstract

The space industry has been growing rapidly in recent years, partly due to an increased demand for space data and related products and services. In order to lower production costs and make space travel more accessible to interested parties, several companies have started to develop reusable rocket parts. Since reaching low earth orbit (LEO) is the most energy demanding part of travel and also one of the most common routes, it is becoming more essential to re-use as many parts of the rocket as possible. The ultimate goal is to have a fully reusable rocket that only requires maintenance and additional propellant in order to be ready for launch again after returning to earth's surface. As of right now there are no fully reusable rockets with orbital capacity. The american spacecraft manufacturer SpaceX is a pioneer in this area and is currently developing a fully resuable rocket system called 'Starship'.

In order to fully understand and develop a similar solution, an electric rocket prototype was constructed with emphasis on it being able to lift off and land completely autonomously. Relevant components as well as software were examined, tested, tuned and implemented to be used in the final prototype in order to perform flight tests. Several steps in the process meant testing each sub-system individually to make sure everything was ready to combine for the final assembly. The developed reuseable rocket model is able to easily take off and fly while keeping itself stable in the air but in order to replicate the landing sequence of the Starship rocket, it requires further testing and development.

Sammanfattning

Rymdindustrin har vuxit snabbt de senaste åren, bland annat på grund av en ökad efterfrågan på rymddata och relaterade produkter och tjänster. För att sänka produktionskostnaderna och göra rymdresor mer tillgängliga för intresserade parter har en del företag börjat utveckla återanvändbara raketdelar. Eftersom resan i låg omloppsbana (LEO) är det mest energikrävande steget av rymdresor blir det allt viktigare att återanvända så många delar av raketen som möjligt under detta steg. Det ultimata målet är att ha en fullt återanvändbar raket som endast behöver regelbundet underhåll och ytterligare drivmedel för att vara redo för uppskjutning igen efter att ha återvänt till jordens yta. Idag finns det ännu inte fullt återanvändbara raketer med orbital kapacitet. Den amerikanska rymdfarkosttillverkaren SpaceX är en pionjär inom detta område och utvecklar för närvarande ett fullt återanvändbart raketsystem vid namn 'Starship'.

För att till fullo förstå och utveckla en liknande lösning konstruerades en elektrisk raketmodell med tonvikt på att kunna lyfta och landa helt autonomt. Relevanta komponenter undersöktes, testades, anpassades och implementerades inför användning i den slutgiltiga raketmodellen och de utförda flygtesterna. Raketmodellen som utvecklades kan flyga och håller sig stabil i luften, men för att replikera landningssekvensen för 'Starship' krävs det ytterligare tester och vidareutveckling.

Förord

Denna rapport är en del av kandidatarbetet EENX16-23-05 på Avdelningen för System- och Reglerteknik på Institutionen för Elektroteknik på Chalmers Tekniska Högskola. Kandidatarbetet pågick under våren 2023 och omfattade 15 högskolepoäng.

Vi vill rikta ett stort tack till vår handledare Jonas Sjöberg, för kontinuerlig och värdefull vägledning under projektets gång. Tack till Chalmers CASE-Lab för tillgång till arbetslokaler, mätinstrument och modelleringsverktyg. Tack även till vår examinator, Jonas Fredriksson, för medverkande.

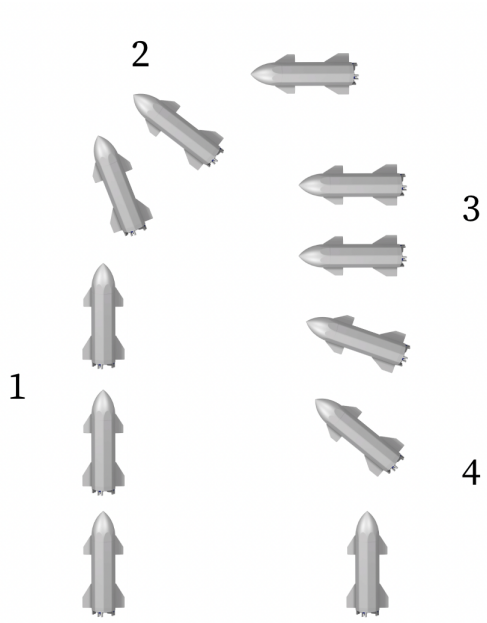
Beteckningar

τ	Tidskonstant
λ	Länkat flöde [Wb]
ϕ	Vinkel relativt tyngdkraften [°]
ρ_{luft}	Luftens densitet vid marknivå [kg/m^3]
A	Area [m^2]
c_d	Luftmotståndskoefficient [-]
d	Raketens diameter [m]
F	Kraft [N]
g	Gravitationskonstant [m/s^2]
H	Raketkroppens höjd [m]
h	Höjd från botten på raketen till masscentrum [m]
i_{motor}	Motorströmmen [A]
J	Massträghetsmoment [kg m^2]
L_{motor}	Motorinduktansen [H]
M	Moment [Nm]
m	Massa [kg]
n	Varvtal [rpm]
P	Propellerstigning (engelska pitch) [m]
r	Raketens radie [m]
R_{motor}	Motorresistansen [Ω]
s	Sträcka [m]
U_{motor}	Motorspänningen [V]
v	Hastighet [m/s]

Innehåll

1 Inledning

Raketer har under en lång tid varit en engångsprodukt som bara har kunnat användas för en enda färd. Innan uppskjutning består raketen av flera så kallade raketsteg, det vill säga självständiga delar av raketen som innehåller egna motorer[1]. De största och dyraste stegen kopplas bort vid en förbestämd altitud för att sedan brinna upp i atmosfären eller förstöras när de fallit tillbaka till jordytan. Än idag är det fortfarande vanligt med denna typ av raketer, men på senare år har det skett många tekniska framsteg som har gjort att återanvändbara delar av raketen ökat signifikant. Företaget SpaceX har legat i framkant inom detta område då de år 2014 utförde en flygning med världens första delvis återanvändbara raket som kunde nå omloppsbana [2]. De arbetar just nu med sitt nya program för att utveckla den helt återanvändbara raketen 'Starship' [3]. Syftet med detta system är delvis att sänka de totala uppskjutningskostnaderna vilket i sin tur lett till ett antal omkonstruktioner, en stor förändring är flygsekvensen vid landning som blivit väldigt annorlunda. För att spara så mycket bränsle som möjligt roteras raketen när den återvänder till jordens atmosfär så att dess huvudaxel är orienterad vinkelrätt i förhållande till flygriktningen, för att erhålla maximalt luftmotstånd. Detta sänker i sin tur raketens fallhastighet avsevärt men innebär också att raketen måste utföra en vändningsmanöver nära slutskedet för att dess orientering ska vara vertikal när den väl landar. I figur 1 finns dessa sekvenser visualiserade. Genom dessa unika sekvenser och sin enorma storlekt satsar SpaceX på att göra resor till rymden mer tillgängliga och kostnadseffektiva med Starship.



Figur 1: Visualisering av de olika stegen för flygsekvensen - uppskjutning (1), vändning till horisontalt läge (2), fritt fall (3) och vändning till vertikalt läge samt landning (4)

1.1 Projektets bakgrund och syfte

Denna rapport beskriver modellering och konstruktion av en elektrisk och autonom raketmodell som utför kontrollerade lyft och landningar, samt utvecklingen av ett reglersystem till denna. Raketmodellen är återanvändbar och kan således påvisa hur raketer kan utvecklas till att bli mer hållbara, både ekonomiskt och miljömässigt.

Rapporten utgår från ett projektförslag vars initiala syfte var att konstruera en autonom och eldriven raketmodell som kan utföra kontrollerade lyft, fall och landningar likt SpaceX Starship. Projekt visade sig i ett tidigt stadie vara mer omfattande än planerat och därför förenklades målet till att få raketmodellen att lyfta och flyga stabilt i vertikal position.

1.2 Avgränsningar

För att minska den mängd lyftkraft som motorerna behöver producera strävades det efter att få en så låg totalvikt på modellen som möjligt. Höjden på modellen skulle vara runt 1 m, vilket begränsade hur mycket modellens vikt kunde påverkas. Utöver detta hade projektet en budget om 5000 kr och tiden var begränsad till en termins halvtidsstudier.

1.3 Frågeställningar

- Vilka sensorer och aktuatorer krävs för att realisera raketmodellen?
- Vad krävs av reglersystem och modellering för att raketmodellen autonomt ska vara stabil runt x-, y- och z-axlarna?
- Vad krävs för att modellen ska kunna uppnå önskad höjd och sedan återvända för att landa?

2 Matematisk modellering för uppskjutning och landning

Här förklaras den nödvändiga teorin som krävs för att konstruera ett reglersystem som styr raketmodellen. De matematiska sambanden i kombination med hur reglersystemen ser ut används för att implementera samt programmera de nödvändiga komponenterna.

2.1 Dimensionering av lyftkraft

För att beräkna en kropps teoretiska, maximala hastighet vid fritt fall används ekvation (1). Vidare kan ekvation (2) användas för att beräkna den nödvändiga lyftkraften som måste genereras vid retardationen under en landningssekvens[4].

$$v_0 = \sqrt{\frac{2mg}{c_d A \rho_{luft}}} \tag{1}$$

$$F_{thrust} = m\Big(g + \frac{v_0^2}{2s}\Big) \tag{2}$$

I ovanstående ekvationer krävs kroppens luftmoståndskoeficient, c_d i detta fallet vid fritt fall med sidan av kroppen riktad mot marken, kroppens massa m, kroppens area vinkelrätt mot marken A, samt luftens densitet vid marknivå ρ_{luft}. I ekvation (2) är s höjden över marken som raketen börjar bromsa upp fallet från.

2.2 Lyftkraft genererad av propeller i kombination med elmotor

Motorspänningen U_{motor} hos en elmotor beräknas enligt ekvation (3).

$$U_{motor} = R_{motor} i_{motor} + L_{motor} \frac{\partial i_{motor}}{\partial t} + \lambda \frac{\pi n}{30}, \tag{3}$$

Ovan betecknar i_{motor} motorstömmen, R_{motor} motorresistansen, L_{motor} motorinduktansen, n motorns varvtal och λ flödeskopplingen i motorn. Det som tydligt framgår här är att motorns varvtal beror på motorspänningen samt motorströmmen. Varvtalet kan lösas ut enligt ekvation (4).

$$n = \frac{30(U_{motor} - R_{motor} i_{motor} - L_{motor} \frac{\partial i_{motor}}{\partial t})}{\pi \lambda}. \tag{4}$$

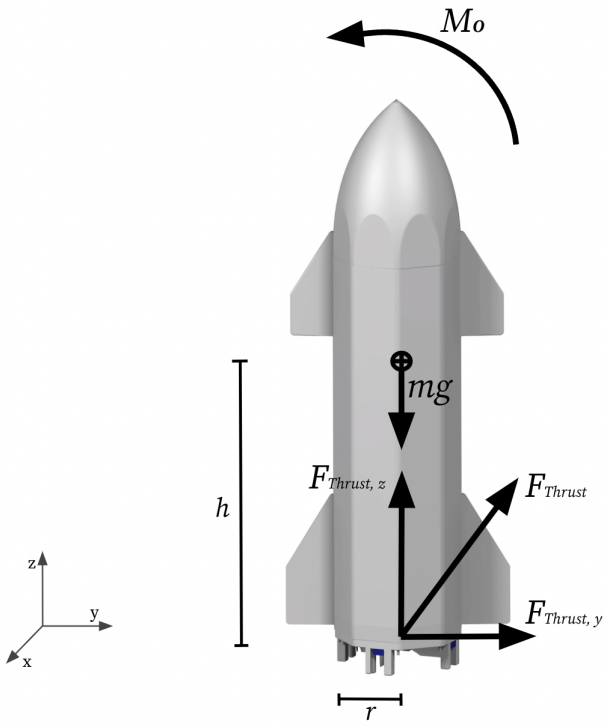
Eftersom att lyftkraften beror på motorns varvtal innebär det således att den också påverkas av motorspänningen och motorströmmen[5]. Lyftkraften som genereras av en propeller beskrivs enligt ekvation (5).

$$F_{Thrust} = 1.225 \frac{\pi d^2}{4} \left(\Big(\frac{n}{60} P\Big)^2 - \frac{n}{60} P \cdot V_0 \right) \left(\frac{d}{3.3 \cdot P} \right)^{1.5} \tag{5}$$

Ekvation (5) beskriver ett förenklat och approximativt samband för propellerns lyftkraft i Newton, där d är propellerns diameter i meter, P propellerstigningen (pitch) i meter och V_0 är lufthastigheten in i propellern i meter per sekund.

2.3 Styrning av raketmodell

För att upprätthålla en stabil flygning krävs reglering kring och längs samtliga axlar. Rotation kring x- och y-axlarna, som illustreras i figur 2, måste motverkas för att hålla kroppen i vertikalt läge under uppskjutning, flygning och landning. Notera att endast friläggning gjorts för y-axeln, men denna ser likadan ut för x-axeln på grund av den rådande symmetrin.



Figur 2: Krafter och moment vid uppskjutning (2D)

För denna reglering utnyttjas de matematiska sambanden i ekvation (6), (7) och (8).

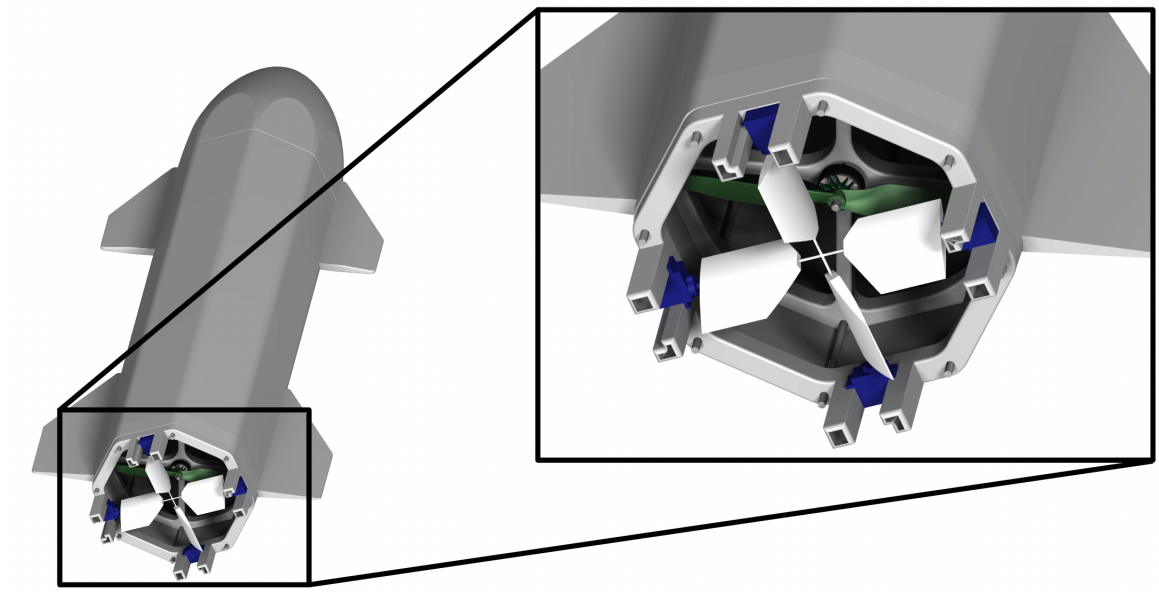
$$\uparrow: \; F_{Thrust,z} - mg = m\ddot{z} \tag{6}$$

$$\rightarrow: \; F_{Thrust,y} = m\ddot{y} \tag{7}$$

$$\overset{\curvearrowleft}{M}: \; F_{Thrust,y} \cdot h - J\ddot{\phi} = M_0, \tag{8}$$

Ovan betecknar J raketmodellens massströghetsmoment kring aktuell axel, ϕ är vinkeln mellan raketmodellens huvudaxel och tyngdkraftens riktning och h är avståndet mellan raketmodellens masscentrum och roder.

Denna styrning åstadkoms med hjälp av tidigare nämnda roder och illustreras i figur 3. Rodren omdirigerar luftflödet i botten av raketmodellen och genererar således ett moment kring dess masscentrum [6].



Figur 3: Roder för reglering av modellens flygriktning.

För att erhålla lyftkraft kommer motorer med roterande propellrar användas. Dessa utvecklar ett vridmoment som verkar kring raketmodellens huvudaxel och leder därför till att den roterar kring huvudaxeln[7]. Vridmomentet för en elmotor beräknas enligt ekvation (9).

$$T = K_t I \tag{9}$$

där T är vridmoment, K_t är förhållandet mellan vridmoment och strömmen I. K_t beräknas enligt ekvation (10).

$$K_t = 1/K_{vSI}. \tag{10}$$

K_v beskriver förhållande mellan motorns varvtal och spänningen. Förutsatt att K_v finns tillgängligt från tillverkaren av elmotor kan K_t beräknas enligt ekvation (11).

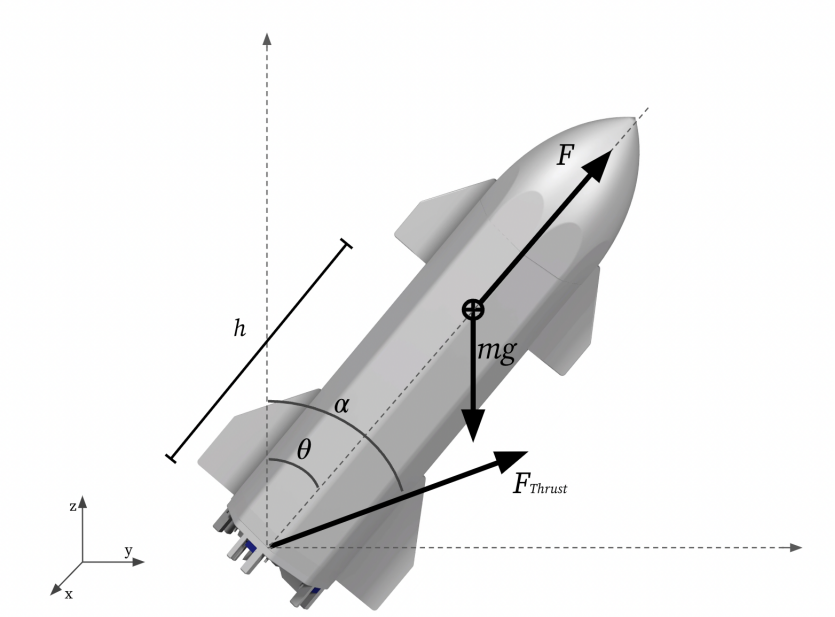
$$K_{vSI} = \frac{K_v \cdot 2\pi}{60} \tag{11}$$

Detta innebär att ekvation (9) kan skrivas om till ekvation (12).

$$T = \frac{1}{K_{vSI}} I \tag{12}$$

Detta medför återigen att kroppen som elmotorn är monterad på kommer att rotera kring motorns huvudaxel, vilket innebär att detta fenomen måste tas i beaktning och motverkas för att raketmodellen skall kunna flyga stabilt.

2.4 Tillståndsmodell



Figur 4: Friläggning av raketmodell för framtagning av tillståndsmodell

Systemets tillståndsmodell kan beskrivas enligt följande om luftmotståndet försummas. Raketens masscentrums position (y_r och z_r) samt raketens riktning kan då beskrivas i två dimensioner med följande ekvationer:

$$y_r = y + \sin(\theta)h \tag{13}$$
$$z_r = z + \cos(\theta)h \tag{14}$$
$$\omega = \dot{\theta} \tag{15}$$

Ur detta kan den horisontella och vertikala accelerationen för systemet tas fram:

$$\ddot{y_r} = \ddot{y} + h\frac{\partial}{\partial t}\omega\cos(\theta) = \ddot{y} + h\dot{\omega}\cos(\theta) - h\omega^2\sin(\theta) \tag{16}$$
$$\ddot{z_r} = \ddot{z} - h\frac{\partial}{\partial t}\omega\sin(\theta) = \ddot{z} - h\dot{\omega}\sin(\theta) - h\omega^2\cos(\theta) \tag{17}$$

Vidare gäller Newtons andra lag:

$$m\ddot{y_r} = F\sin(\theta) = F_{Thrust}\cos(\alpha - \theta)\sin(\theta) \tag{18}$$
$$m\ddot{z_r} = F\cos(\theta) - mg = F_{Thrust}\cos(\alpha - \theta)\cos(\theta) - mg \tag{19}$$

Genom att eliminera kraften F ur ekvationerna (18) och (19) fås:

$$m\ddot{y}_r \cos(\theta) - m\ddot{z}_r \sin(\theta) = mg\sin(\theta) \tag{20}$$

Detta ger den dynamiska modellen för raketens rörelse:

$$h\dot{\omega} + \ddot{y}\cos(\theta) = g\sin(\theta) \Rightarrow \tag{21}$$

$$\Rightarrow \dot{\omega} = \frac{g\sin(\theta) - \ddot{y}\cos(\theta)}{h}, \tag{22}$$

$$J\dot{\omega} = F_{Thrust} h \sin(\alpha - \theta) \Rightarrow \tag{23}$$

$$\Rightarrow F_{Thrust} = \frac{J\dot{\omega}}{h\sin(\alpha-\theta)} = \frac{J(g\sin(\theta) - \ddot{y}\cos\theta)}{h^2\sin(\alpha-\theta)} \tag{24}$$

Tillståndsvektor $\begin{bmatrix}\theta & \omega & y & v\end{bmatrix}^T$ införs där $v = \dot{y}$ och den generaliserade tillståndsmodellen fås enligt:

$$\dot{\theta} - \omega = 0 \tag{25}$$

$$h\dot{\omega} + \dot{v}\cos(\theta) - g\sin(\theta) = 0 \tag{26}$$

$$\dot{y} - v = 0 \tag{27}$$

$$\frac{J(g\sin(\theta) - \ddot{y}\cos\theta)}{h^2\sin(\alpha-\theta)} - F_t = 0 \tag{28}$$

Linjärisering av modellen med $\begin{bmatrix}\theta & \omega & y & v\end{bmatrix}^T = \begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}^T$ som arbetspunkt ger:

$$\dot{\theta} - \omega = 0 \tag{29}$$

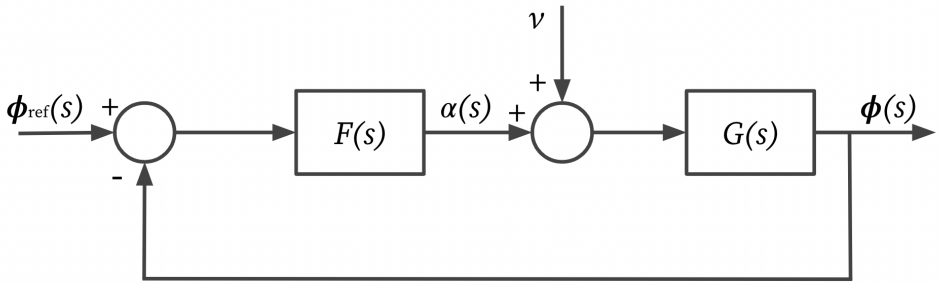
$$h\dot{\omega} + \dot{v} - g\theta = 0 \tag{30}$$

$$\dot{y} - v = 0 \tag{31}$$

$$\frac{J(g\theta - \dot{v})}{h^2(\alpha-\theta)} - F_t = 0 \tag{32}$$

2.5 Implementering av regulatorer

Det återkopplade systemet för raketmodellens stabilisering kring x- och y-axeln illustreras i figur 5.



Figur 5: Återkopplat system med processtörning v som styr vinkeln

$F(s)$ representerar funktionen av PID-regulatorn som är felet mellan vinkeln av raketmodellens huvudaxel och tyngdkraftens riktning $\phi(s)$ och den önskade vinkeln $\phi_{ref}(s)$. Vidare representerar v de laststörningar som introduceras under flygningen. Utifrån detta beräknas $\alpha(s)$ vilket är den vinkel mellan rodren och raketmodellens huvudaxel som krävs för att kontrollera dess riktning kring x- och y-axeln. Funktionen $F(s)$ för en PID-regulator kan skrivas i Laplacedomänen enligt ekvation (33).

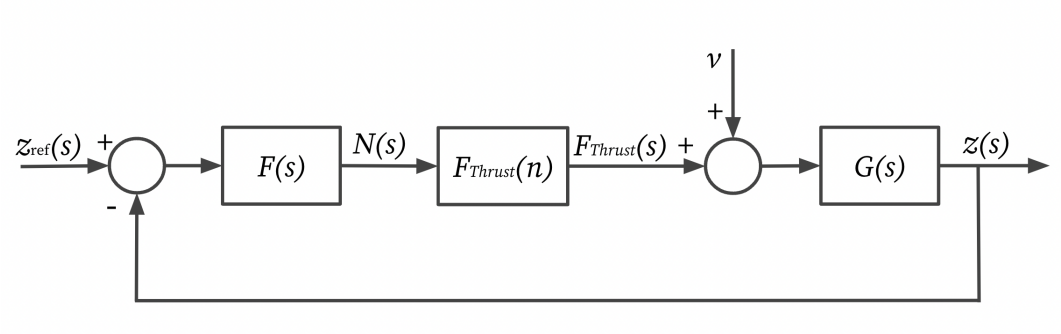
$$F(s) = K_p + \frac{K_i}{s} + K_d s \tag{33}$$

Tillsammans med ekvationerna (6)-(8), blir överföringsfunktionen för systemet enligt ekvation (34) [8].

$$G(s) = \frac{\phi(s)}{\alpha(s)} = \frac{hF_{thrust}}{Js^2} \tag{34}$$

För att bestämma J och h genomfördes ett antal tester på den fysiska modellen, då dess massfördelning är komplex och inte kan approximeras tillräckligt noggrannt med hjälp av enklare geometrier. Dessa tester förklaras i avsnitt 5.1.2.

Det återkopplade systemet för höjdregulatorn som styr motorns rotationshastighet syns i figur 6.



Figur 6: Återkopplat system med processtörning, v, som styr höjd

Här är $z_{ref}(s)$ den önskade höjden och $z(s)$ är den faktiska höjden. $F_{Thrust}(n)$ beskriver relationen mellan varvtal och lyftkraft, se ekvation (5). Även höjden styrs med en PID-regulator där $F(s)$ ser ut på samma sätt som i ekvation (33). $G(s)$ härleds ur ekvation (6).

$$G(s) = \frac{Z(s)}{F_{thrust}(s)} = \frac{1}{ms^2} \tag{35}$$

2.6 Ziegler-Nichols metoden för beräkning av PID-konstanter

För att kunna ta fram värden på PID-regulatorer experimentellt kan Ziegler-Nichols metoden vara användbar. Fördelen med att arbeta med denna metod är att empiriska studier eller tunga beräkningar inte krävs för att få ut PID-värdena. Dock är metoden av generell karaktär vilket innebär att de värdena som beräknas inte alltid stämmer utan kan komma att behöva justeras ansenligt. Oavsett kan metoden användas som utgångspunkt för preliminära PID-konstanter för att sedan gå vidare med justering av en eller flera av dessa [9]. Metoden går ut på att stänga av I- och D-komponenterna i PID-regulatorn för att sedan ändra P-regulatorns värde till 0. Därefter ökas P-värdet successivt fram till självsvängning uppstår. Utifrån periodtiden på självsvängningen kan initiala värden på PID-konstanterna beräknas enligt nedan ekvationer [10].

$$K_p = \frac{3}{5}K_u \tag{36}$$

$$K_i = \frac{T_u}{2} \tag{37}$$

$$K_d = \frac{8}{T_u}, \tag{38}$$

där K_p, K_i och K_d är PID-konstanterna, K_u är värdet på K_p då självsvängningen uppstår, och T_u är periodtiden för självsvängningen.

3 Aktuatorer, sensorer och filter

I detta avsnitt beskrivs de aktuatorer som användes vid konstruktion av raketmodellen. Vidare tas det upp vilka sensorer som används för att styra aktuatorerna samt vilka filter som implementerats för sensorerna.

3.1 BLDC-Motor för framåtdrivning

Raketmodellen drivs av BLDC-motorer (förkortning för 'Brushless Direct Current'), med tillhörande propellrar. En BLDC-motor består av två komponenter, en stator och en rotor. Rotorn, som är lokaliserad i mitten av motorn, är den del som roterar och på den sitter en permanentmagnet. Beroende på motorns utförande är det två eller fler poler i motorn. Dessa poler har ett "N" och ett "S". För att få rotorn att rotera tillförs ström till statorn som skapar magnetfält som i sin tur attraherar motstående pol på rotorn. Om detta utförs med en sådan hastighet att strömmen som tillförs alltid går till den stator som ligger framför rotorn så kommer BLDC motorn att kunna erhålla ett varvtal. Fördelen med denna motor är möjligheten att kunna variera varvtalet samt vridmomentet som kan uppnås. Nackdelarna med denna motorn är kostnaden att tillverka vilket leder till ett högre inköpspris samt kravet att använda ett ESC, vilket leder till en ännu högre kostnad [11].

3.2 ESC för fartreglering

Fartreglering och styrning av BLDC motorn, sker med hjälp av ett ESC/fartreglage (förkortning för 'Electronic Speed Controller'). Detta sker genom en in-signal från t.ex. en arduino som talar om vilket varvtal som önskas [11]. För att bestämma varvtalet så anpassar ESC den spänning som går till motorn för att generera det magnetfält som är nödvändigt för att rotera rotorn till valt varvtal. Beroende på hur fort fargregalget kan hantera insignalen eller vilken frekvens som kan uppnås så avgör det i sin tur vilket max varvtal motorn får, detta begränsas också av motorns maximala kapacitet.

3.3 Servomotorer för positionering

För att ändra rodernas position genom rotation, används en servomotor. Denna aktuator består av en elmotor, positions-potentiometer, mikrokontroller, samt ofta kugghjul för att anpassa utväxlingen beroende på användningsområdet. En insignal från till exempel arduinon talar om vilket värde på potentiometern som önskas, sedan roterar elmotorn för att justera armen på utsidan. Denna arm sitter ofta ihop med en kontrollyta t.ex. ett roder som gör att att rodret kommer ändra sin position till önskat värde.

3.4 IMU för mätning av vinkelhastighet och kraft

För mätning av vinkelhastighet och kraft, används en IMU, som är en förkortning för 'Inertial Measurement Unit'. IMU:n består av ett gyroskop för mätning av vinkelhastighet, samt en accelerometer för kraftmätning. Ibland kan en IMU även innehålla magnetometrar som mäter magnetfältet runtom enheten.

De sensorer som används i projektet är gyroskopet och accelerometern. Accelerometern på den inköpta IMU:n är en MEMS accelerometer. En MEMS är i princip en fjäder med en vikt fäst i mitten på fjädern. När accelerometern utsätts för en linjär acceleration kommer det resultera i att vikten på fjädern skiftar antingen uppåt eller neråt beroende på i vilken riktning accelerometern accelererar.

Ett gyroskop är en sensor som mäter vinkelhastighet med respekt till en given referens. Gyroskopet på IMU:n som används i denna rapport är ett MEMS-gyroskop. MEMS-gyroskop mäter vinkelhastigheten genom att använda sig av teorin om Corioliseffekten som bygger på den tröghetskraft som verkar på föremålet i rörelse i förhållande till en roterande ram. Även här är det en vikt fäst till fjädrar i x-led, y-led och z-led. Massan kommer ha en drivande kraft i x-led, vilket leder till att den oscillerar snabbt i x-led. När den är i rörelse kommer en vinkelhastighet att appliceras kring z-axeln, vilket resulterar i att det blir en kraft i y-led, som ett resultat av Corioliseffekten [12]. Den vinkeln som IMU:n har roterat kring de tre axlarna ges av ekvation (39).

$$\phi(t) = \int_0^t \omega(t)\, dt \tag{39}$$

3.4.1 Exponential smoothing-filter

För att undvika väldigt hackiga och ryckiga servomotorer, kan ett exponential smoothing-filter användas. Detta filter tar både nya och gamla mätvärden och sätter en vikt på dessa, vilket leder till en interpolering av de två värdena. Detta leder till att signalen får mindre extremt brus och på så sätt blir servomotorerna mindre ryckiga på gund av det brus som IMU:n i kombination med Madgwick-filtret orsakar [13]. Det som sker är att högfrekventa variationer i tidsserien dämpas, medans lågfrekventa variationer bevaras i en större utsträckning. Enligt Rob J Hyndman and George Athanasopoulos är det önskvärt att fästa större vikt vid nyare data än äldre, det vill säga använda ett högre värde på α i ekvation (40).

$$y_{T+1} = \alpha \cdot y_T + (1 - \alpha) \cdot y_{T-1} \tag{40}$$

3.4.2 Madgwick-filter

För att finna IMU:ns orientering och därmed raketens orientering, används ett Madgwick-filter. Madgwick-filtret är en algoritm som används inom sensorfusion för orienteringsberäkning av ett föremål med hjälp av sensorinput från en IMU som innehåller gyrometer, accelerometer och eventuellt magnetometer. Genom att kombinera datan som givs av dessa tidigare nämnda sensorer kan algoritmen estimera sensorns orientering i rymden, vilket då tagits fram genom att beräkna kvaternioner. En kvaternion kan beskrivas som ett matematiskt objekt som återger raketens rotation kring en stationär referensram, vilken oftast är den initiella orienteringen av enheten. För att uppskatta felet i beräkningarna av kvaternionerna och utifrån det förbättra nogrannheten i beräkningarna används en gradientbaserad optimeringsalgoritm, vilket resulterar i en snabb och stabil algoritm. För matematiska detaljer om filtrets implementering se [14][15].

3.5 Barometrisk trycksensor för höjdmätning

I projektet görs höjdmätning med hjälp av en barometrisk trycksensor, som mäter förändringar i atmosfärstrycket. Denna trycksensor har ett tunt cirkulärt membran och en töjningsmätare. Den märker av förändringar genom att membranet deformeras av antingen högre eller lägre tryck, då omvandlar töjningsmätaren detta till en elektrisk signal. Sensorn mäter förändringen relativt den referens som sensorn kalibreras till, p_0 [16]. Ekvation (41) tar hänsyn till både tryck, p, och temperatur, T, för att sedan omvandla detta till en höjd relativt marknivån [17].

$$h = \frac{\left(\left(\frac{p_0}{p}\right)^{\frac{1}{5.257}} - 1\right) \cdot (T + 273.15)}{0.0065} \tag{41}$$

3.5.1 Lågpassfilter

Ett lågpassfilter släpper endast igenom signaler som ligger under gränsfrekvensen medan det dämpar alternativt eliminerar signaler med en högre frekvens [18]. Ett första ordningens lågpassfilter skrivet som en överföringsfunktion beskrivs enligt ekvation (42), där τ är beteckningen för tidskonstanten.

$$G(s) = \frac{1}{\tau s + 1}, \tag{42}$$

4 Realisering av modell

I detta avsnitt beskrivs hur raketmodellen realiserades. Först förklaras de förberedande testerna vilka låg som grund för de krav som sammanställdes för modellens konstruktions- och elektronikkomponenter. Dessutom förklaras hur samtliga filter implementerats för att reducera brus och stokastiska mätvärden för de aktuella sensorerna.

4.1 Förberedande tester inför konstruktion

Initialt utfördes ett antal tester för att undersöka hur stor lyftkraft som kunde uppnås med olika konfigurationer av komponenter. Dessa tester utfördes med en redan befintlig och svagare testmotor, ESC och batteri, som redan fanns till förfogande. Detta för att erhålla preliminära resultat som användes som underlag inför de slutgiltiga inköpen. I testerna var de varierande faktorerna propellerdiameter, antal propellerblad, samt när- och frånvaron av ett ytterhölje kring propellern. Resultatet från dessa tester användes för att fastställa en kombination av komponenter som genererar tillräckligt hög lyftkraft för den slutgiltiga raketmodellen.

4.1.1 Verifiering av motorns prestanda

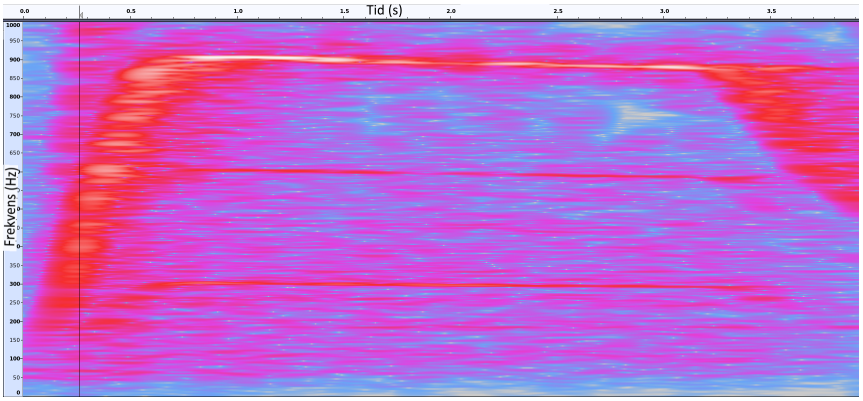
I figur 7 visas en anordning där testmotorn är monterad på en vertikal arm, och en lastcell som är placerad under en horisontell arm. Dessa två armar har samma längd och är vinkelräta mot varandra. När motorn och därmed propellern roterar genereras en kraft som verkar i samma riktning som motorns rotationsaxel. Detta frambringar ett moment kring armarnas fästpunkt och ger ett kraftutslag på lastcellen som motsvarar den av motorn och propellern genererade lyftkraften. På så vis var det möjligt att avgöra huruvida motoruppsättningen lyckas uppnå en tillräckligt hög lyftkraft.

(a) Prototyp av testanordning **(b)** CAD-modell av testanordning

Figur 7: Testanordning för preliminär mätning av lyftkraft

Dessa tester spelades in och ljudfilen användes sedan för att bestämma propellerns rotationshastighet.

Ljudet analyserades med hjälp av programvaran Audacity, och det framgår enligt figur 8 att motorns rotationsfrekvens når 300 Hz, vilket motsvarar 18000 rpm.



Figur 8: Frekvensanalys för trebladig propeller (8 tum) i Audacity. Intensitet i stigande ordning visas i skalan blå-röd-vit.

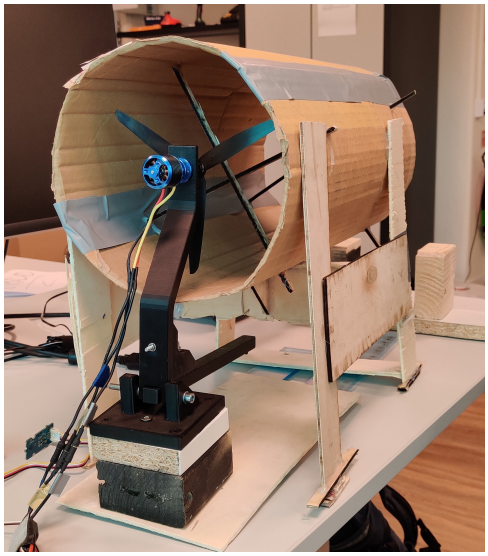
Testanordningen ovan användes även för att mäta lyftkraft och varvtal hos en av motorerna som användes i den slutgiltiga modellen. Resultatet redovisas i tabell 1 nedan.

Tabell 1: Maximal lyftkraft från motor som används i slutgiltig raketmodell

Antal propellerblad	Propellerdiameter (tum)	Maximalt varvtal (rpm)	Maximal lyftkraft (N)
3	8	18000	21,5

4.1.2 Lyftkraft vid maximalt varvtal med och utan ytterhölje

Testanordningen som visas i figur 9 användes för att approximera hur stor lyftkraft som genererades vid förändring av raketkroppens utformning. Testet utfördes med ett cylindriskt ytterhölje som omsluter propellern. Resultatet gav en uppfattning av hur mycket raketkroppens skal påverkar lyftkraften på grund av den begränsade lufttillförseln.



Figur 9: Testanordning med cylindriskt skal framför propeller

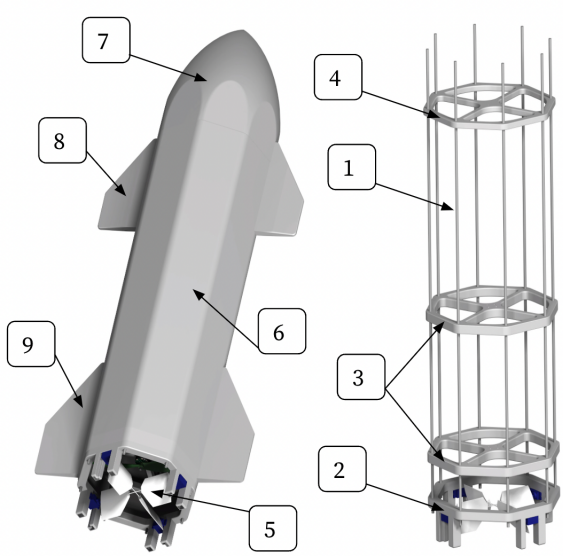
Från de uppmätta lyftkrafterna enligt tabell 2 är det tydligt att ytterhöljet har en stor påverkan på lyftkraftens magnitud - närvaron av ett ytterhöljde ledde till en minskning av den maximala lyftkraften på 31 %.

Tabell 2: Andel av maximal lyftkraft beroende på närvaron av ytterhölje

Cylindriskt ytterhölje	Uppmätt lyftkraft (% av maximal lyftkraft)
Utan ytterhölje	100
Med ytterhöljde	69

4.2 Materialval för konstruktionskomponenter

De komponenter som utgör raketmodellen är tillverkade av ett flertal olika material. Val av material har gjorts baserat på de krav som ställts på främst hållfasthet, temperatur, och vikt. Notera att det inte är möjligt att göra materialval på vissa komponenter, såsom motor och ESC, då dessa beställts och de ingående materialen är således förutbestämda. I dessa fall lades fokus på att maximera komponentens prestanda och samtidigt minimera kostnaden. Raketens olika konstruktionskomponenter med tillhörande materialval visas i figur 10 och tabell 3.



Del	Beskrivning	Antal (st)	Material
1	Stav	8	Kolfiber
2	Roderskiva	1	CPE
3	Motorskiva	2	CPE
4	IMU-skiva	1	CPE
5	Roder	4	CPE
6	Ytterhölje	1	Polystyren
7	Nos	1	CPE
8	Fena, nos	2	CPE
9	Fena, kropp	2	CPE

Figur 10 & Tabell 3: Överblick av raketmodellens huvudsakliga konstruktionskomponenter

Raketmodellens massa är en starkt dimensionerande faktor då denna är direkt proportionell till den lyftkraft som krävs [19]. Detta innebär att material med höga hållfasthetsegenskaper och låg vikt är att föredra. Ett material som fyller dessa krav är kolfiber, som består av en tunn polymerväv som läggs i flera lager för att åstadkomma önskad geometri. Kolfiber och stål har ungefär lika hög styvhet, men densiteten för kolfiber är nästan fem gånger lägre än stål [20]. Detta gör kolfiber till ett mycket passande material för raketmodellens skelettstruktur.

En del av komponenterna är tillverkade med hjälp av 3D-skrivare, detta på grund av deras specifika geometrier och dimensioner. Samtliga skivor och roder är tillverkade av CPE-plast (förkortning för 'Chlorinated Polyethylene Elastomer') då dess höga elasticitetsmodul och duktilitet är fördelaktiga egenskaper vid eventuella kollisioner [21]. CPE tillåter även utskrift av små och tunna komponenter, vilket gör det mycket lämpligt fr utskrift av rodren. Dessutom har CPE en högre temperaturtålighet, vilket innebär att risken för smälta och deformationer på grund av värmen som genereras av elektroniken minimeras [22]. Utskriftstiden ökar som följd av den höga temperaturtåligheten, men med tanke på att raketmodellen är relativt liten är detta inte något som påverkar tidsaspekten avsevärt. Skivorna har en ifyllnadsgrad på 50 procent, detta för att minimera vikten men också göra den kollisionstålig [23]. De komponenter som tillverkades med 3D-skrivare modellerades i CATIA V5. Slicer-programvaran som användes var Ultimaker Cura (v5).

Notera att raketmodellen som realiserats inte har ytterhölje, nos eller fenor. Beslutet att exkludera dessa komponenter togs på grund av den begränsade tidsramen samt de förberedande testerna under avsnitt 4.1.2 som visade på att raketens lyftkraft minskar signifikant med ett ytterhölje. Ett högre luftflöde och lyftkraft ger mer auktoritet att snabbt styra raketens riktning enligt ekvation (8). Implementering av dessa komponenter och eventuella anpassningar för dessa diskuteras i kapitel 8.

4.3 Raketmodellens utformning och dimensioner

Raketen som använts för inspiration under arbetets gång, SpaceX 'Starhip', är 50 m hög och 9 m i diameter [3]. Detta ger ett höjd/diameter-förhållande på 5.5 och kommer användas som utgångspunkt

vid konstruktion av raketmodellen. Utöver detta finns det ett antal begränsningar som påverkat raketmodellens dimensioner och form. De 3D-skrivarna som användes under projektets gång tillät endast utskrift av komponenter med en diameter på 0.2 m. Propellerdiametern som användes var dock 0.203 m. Av denna anledning utformades skivorna med en åttakantig geometri där kolfiberstavarna placeras i skivornas hörn. Detta ökade avståndet mellan stavarna från 0.2 m till 0.23 m och propellern får därmed plats mellan stavarna. Dessutom innebär den oktogonala geometrin att montering av komponenter underlättas och tillåter enkel och symmetrisk uppställning av dessa. Roderskivan har exempelvis anpassade fästen för de servon som används, och motorskivan har genomgående hål för enkel montering av motor. För fullständiga specifikationer och ritningar för dessa komponenter, se bilaga A.2.

En diameter på 0.23 m för raketmodellen innebär att höjden ska vara ungefär 1.1 m för att erhålla ett höjd/diameter-förhållande på 5.5. Kolfiberstavarna har en maximal längd på 1 m, vilket gör raketmodellen något kortare. Detta är dock inte något som är direkt avgörande för raketmodellens prestanda, utan används återigen som utgångspunkt för rimliga dimensionsförhållanden.

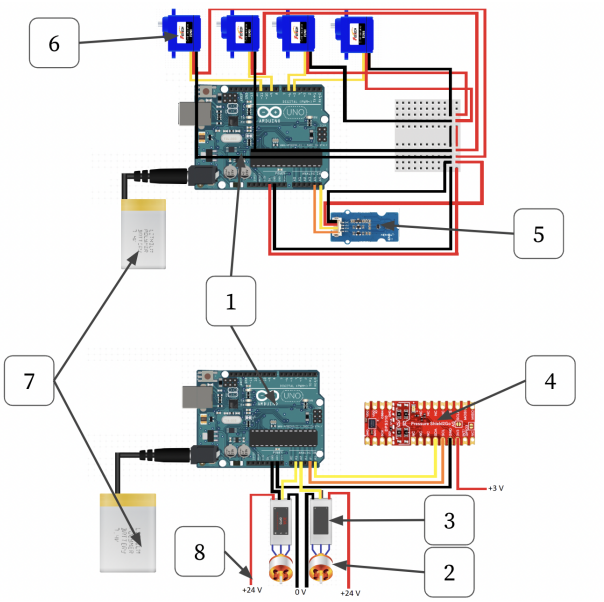
Den sammanställda massan för samtliga konstruktionskomponenter som användes blev 1.4 kg. Det omfattar stavar, roderskiva, motorskivor, IMU-skiva och roder. Detta låg som underlag för de specifikationer som krävs av motorer, ESC och batterier. Valen av dessa komponenter var en iterativ process där en lämplig kombination av komponenterna användes för att beräkna dess maximala lyftkraft. Utvärdering av och jämförelse mellan den genererade lyftkraften och raketmodellens vikt utfördes för att fastställa huruvida tillräcklig lyftkraft erhålls. Processen utfördes iterativt eftersom varje kombination av motor, ESC och batteri har varierande totalvikt och lyftkraft, vilket således ger olika vikt/lyftkraftsförhållande. Notera alltså att denna vikt inte ligger som grund för vidare beräkningar av exempelvis massströghetsmoment och masscentrum. Dessa fastställs när elektroniken är monterad då det är i denna utformning som raketmodellen ska användas.

4.4 Elektriska komponenter

De elektriska komponenter som används samt deras tillhörande kopplingsscheman presenteras i nedan figur och tabell. För mer information om komponenterna, se bilaga A.3.

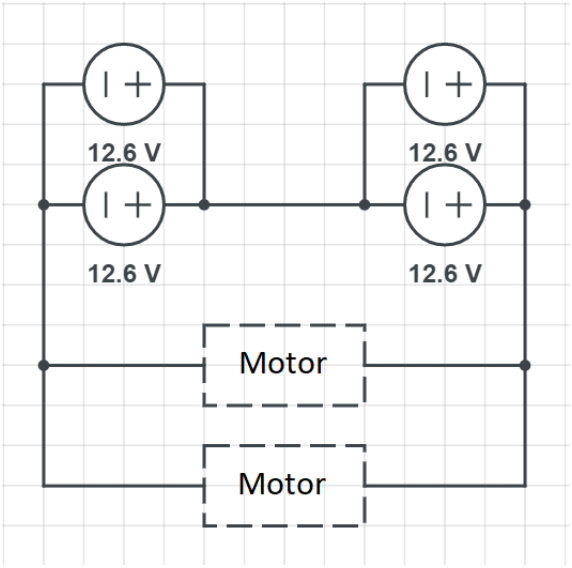
Raketmodellen behöver en lyftkraft som är större än dess egenvikt för att modellen ska kunna lyfta. För att ha ytterligare kontroll över modellen togs beslutet att lyftkraften föredragsvis är 1.5 gånger större än egenvikten. Enligt specifikationerna på den valda motorn kan den generera 31,4 N i lyftkraft när den förses med en spänning på 24 V och en ström på 70 A. Två av dessa motorer skulle då ge en lyftkraft på 62,8 N, vilket är mer än tillräckligt för att lyfta raketmodellen utifrån den uppskattade massan från avsnitt 4.3.

Kravet på batterierna är att de måste leverera minst 24 V och 70 A kontinuerligt till båda motorerna. För att uppnå detta samt för att ha tillräcklig kapacitet för att modellen ska kunna genomföra minst en färd valdes 4 batterier som är kopplade enligt figur 12.



Del	Beskrivning	Antal (st)
1	Arduino Uno	2
2	BLDC motor	2
3	ESC	2
4	Trycksensor	1
5	IMU	1
6	Servo	4
7	Batteri till arduino	2
8	Batteri till motor	2

Figur 11 & Tabell 4: Överblick av raketmodellens huvudsakliga elektronikkomponenter.

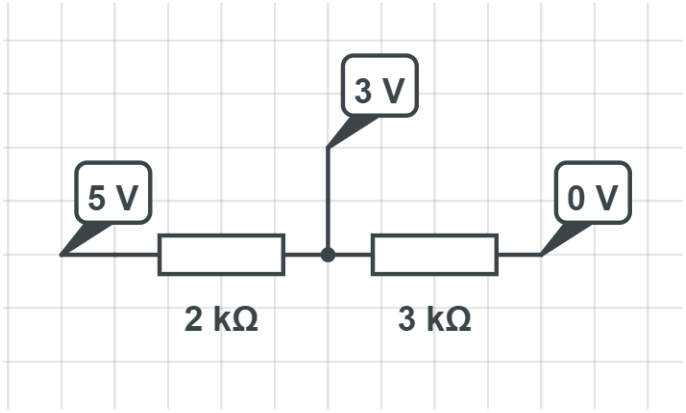


Figur 12: Kopplingsschema över motorer och batterier

För styrning av motorerna används två stycken ESC, en till vardera motor. Dessa behöver både kunna hantera den spänning som batterierna levererar och den maximala strömmen som flödar i kretsen. Denna komponent har överdimensionerats något för att säkerställa att den inte havererar när maximal effekt förses från batterierna.

Som mikrokontroller används två stycken Arduino Uno. Eftersom Arduino Uno endast har två kontakter tillgängliga för sensor-inputs behövs två stycken för att använda både tryckmätning från trycksensorn, samt lutning och acceleration från IMU:n.

Trycksensorn klarar av en maximal spänning på 4 V. Eftersom det inte kan garanteras att spänningsmatningen från Aruduino Unon:s 3,3 V ligger konstant under 4 V krävs en separatspänningsmatning till denna. Spänningsnivån till denna valdes till 3 V och åstadkoms enligt figur 13 , där spänningsmatningen på 5 V kommer från Arduinon:s 5 V port.



Figur 13: Resistorkoppling för 3 V spänningsmatning

5 Tester och validering av raketmodellens komponenter

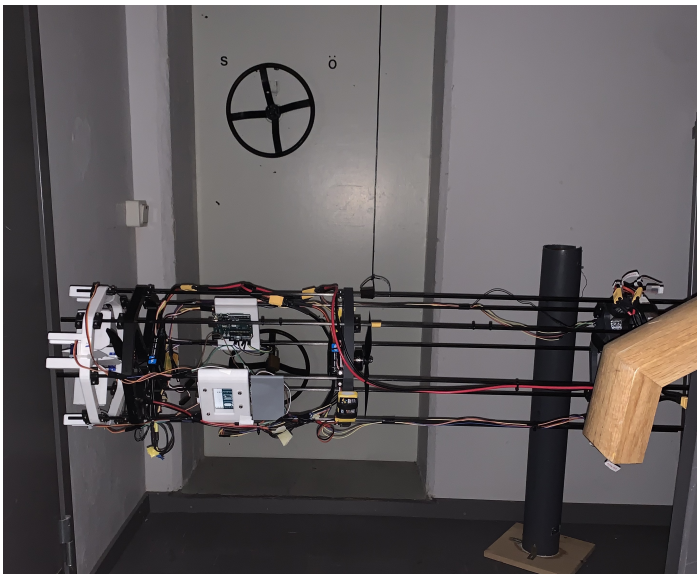
Följande avsnitt berör de tester som utfördes på den slutgiltiga raketmodellen, för att kontrollera att den teori som togs fram innan modelleringen stämmer överens med den fysiska modellen. Här beskrivs även en simulering av hur en flygsekvens kan se ut.

5.1 Framtagande av fysikaliska storheter för matematisk modellering

Här beskrivs de matematiska konstanter som krävs för att modellera raketen och för att senare hitta de optimala värdena för PID-regulatorerna.

5.1.1 Raketens vikt och masscentrum

Raketmodellen har en totalvikt på 2.355 kg, detta innefattar alla komponenter, såväl konstruktion som elektronik. Fastställning av modellens masscentrum utfördes genom att hänga raketmodellen i en lina och sedan balansera den tills den hängde vågrätt. Detta verifierades sedan med ett digitalt vattenpass. Avståndet från raketens bas till dess masscentrum mättes till 0.45 m.



Figur 14: Mätning av masscentrum hos raketen

5.1.2 Experimentell framtagning av massträghetsmoment

För att beräkna massträghetsmomentet kring dess masscentrum hängdes raketmodellen upp i två linor enligt nedan figur, där avståndet mellan respektive lina noterades. Modellen sattes sedan i svängning, och därefter kan massträghetsmomentet beräknas experimentellt.

Figur 15: Uppsättning för beräkning av massströghetsmoment hos raketen

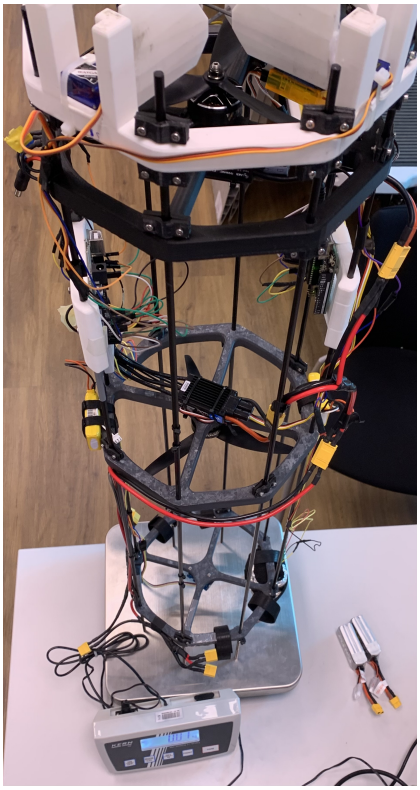
Sambandet som användes vid beräkning av raketmodellens massströghetsmoment vid rotation runt x- och y- axeln redovisas i ekvation (43) [24].

$$J = \frac{mgD^2T^2}{16\pi^2 L}, \tag{43}$$

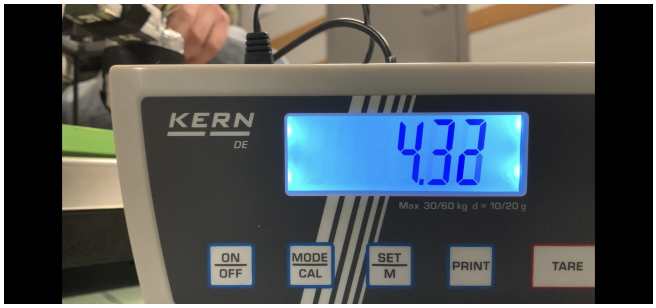
Ovan betecknar T periodtiden för modellens oscillering, L är linornas längd och D är avståndet mellan linorna. Insättning av parametervärdena $D = 0.33$ m, $L = 1.53$ m, $T = 5.4$ s i ekvation (43) ger att $J = 0.304$ kgm^2.

5.1.3 Maximal producerad lyftkraft

För att mäta den lyftkraft som raketmodellen kan producera vändes raketen upp och ner och placerades på en våg. Vågen nollställdes sedan och motorerna aktiverades vilket riktar kraften direkt mot vågen. På så vis kunde raketmodellens lyftkraft mätas. Under detta test visade vågen en maximal vikt på 4.32 kg, vilket motsvarar ungefär 42.4 N enligt Newtons andra lag.



(a) Uppsättning vid mätning av lyftkraft

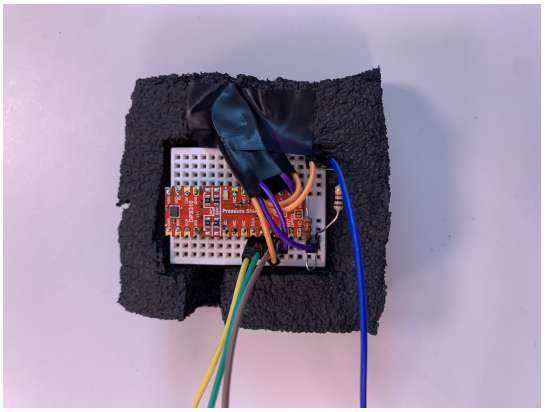


(b) Uppmätt lyftkraft vid maximal effekt

Figur 16: Mätning av lyftkraft

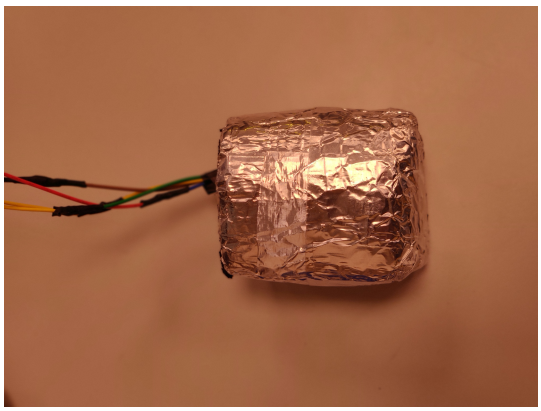
5.2 Test av barometrisk trycksensor

Trycket som mäts upp med trycksensorn konverteras till altitud med ekvation (41) [25]. Vid höjdmätning är trycksensorn mycket känslig för små tryckförändringar. Svaga vindpustar och tryckförändringar till följd av att en dörr öppnas ett par meter bort leder till ett mätfel i storleksordningen 0.5 m. För att undvika dessa störningar är trycksensorn inkapslad i ett skal av polyuretan enligt figur 17. Skalet innehåller små porer, som starkt begränsar svaga luftflödens påverkan, vilket ger ett mer stabilt tryck inuti skalet. I figuren syns endast hälften av skalet. Ett likadant skal är placerat även på ovansidan av sensorn för att isolera trycksensorn i alla riktningar.



Figur 17: Trycksensor monterad i polyuretan-skal

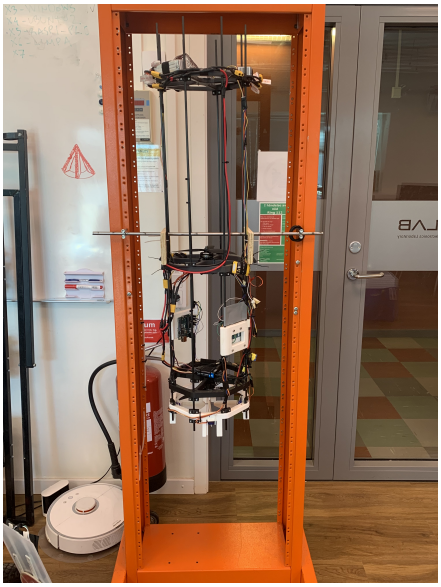
Utöver detta omslöts även polyuretan-skalet med ett tunt lager av aluminiumtejp, se figur 18. Detta gjordes för att reflektera så mycket solljus som möjligt, då ljuset bidrar till att temperaturen höjs inuti skalet.



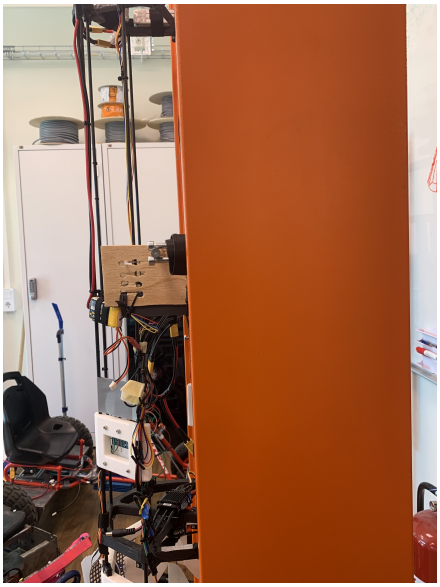
Figur 18: Trycksensor omsluten i aluminiumtejp

5.3 Framtagande av PID-konstanter för stabilitetsreglering

I figur 19 syns testuppställningen som användes för att experimentellt bestämma PID-konstanterna K_p, K_i och K_d, för stabilisering av raketmodellen. Testet utfördes genom att montera raketmodellen så att masscentrum hamnar ovanför staven som modellen är monterad på. Detta gör att modellen hänger med nosen nedåt om motorerna är avstängda. Därefter utfördes tester för att låta raketmodellen styra tillbaka till upprät position. På så sätt kunde olika värden för PID-regulatorn testas och gemföras för att finna gynnsamma värden.



(a) Testrigg för stabiliseringsreglering framifrån



(b) Testrigg för stabiliseringsreglering från sidan

Figur 19: Testrigg för stabilitetskontroll

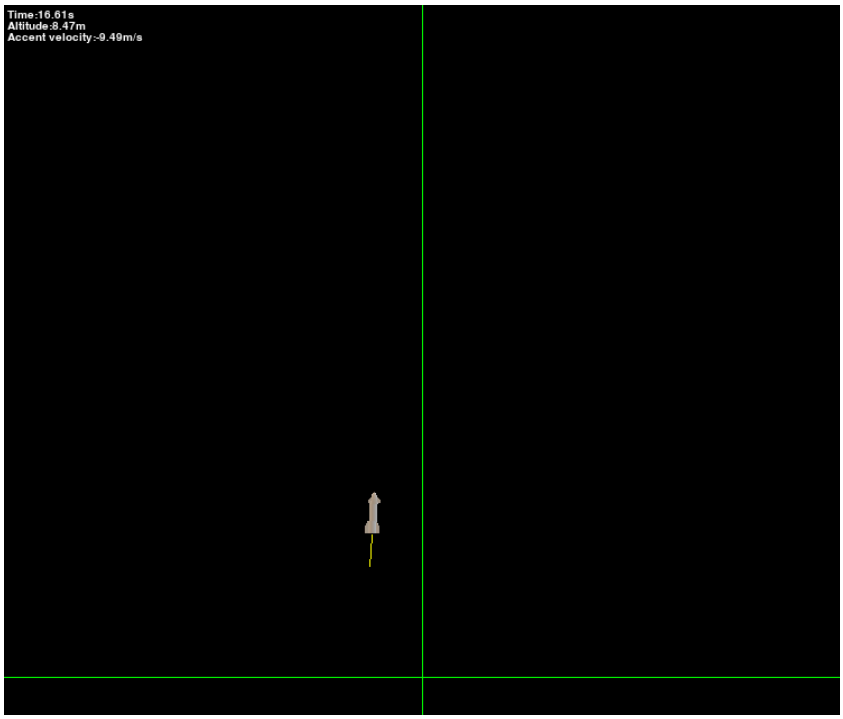
5.4 Framtagande av PID-konstanter för höjdreglering

Konstanterna K_p, K_i och K_d bestämdes även för höjdregleringen experimentellt med hjälp av Ziegler-Nicholsmetoden enligt testet i figur 20.

Figur 20: Uppställning för experimentell bestämning av höjdregulator

5.5 Simulering av flygsekvens med reglering

Syftet med simuleringen var att se hur olika variabler påverkade regleringen av raketen i de olika stadierna av flygsekvensen. Koden för simuleringen presenteras i bilaga A.4, simuleringen såg då ut på följande sätt (se bilaga A.1):



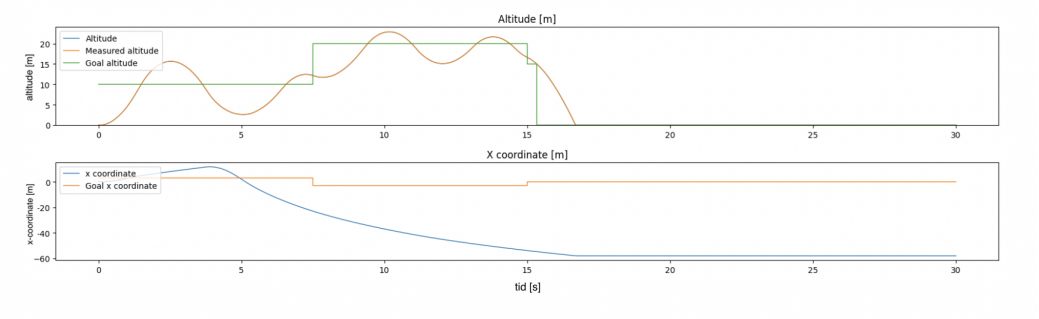
Figur 21: Simulering av raket

I simuleringen användes ekvationerna (4) - (8), som presenterades i kapitel 2. För att utföra simuleringen användes Newtons stegmetod för att simulera systemet i tidsdiskreta steg. Simuleringen sker i två dimensioner för att förenkla implementeringen, men principerna för regulatorerna stämmer även i tre dimensioner. Nedan presenteras resultat från olika regulatortyper. Dessa resultat kan användas för att motivera typ av regulator som kommer användas i den fysiska implementationen.

Testsekvensen bestod av att raketen först skulle nå en serie av koordinater: (3,10), (-3,20), (0,15), (0,0). Efter att raketen försökt nå en höjd på 20 m byter regulatorerna läge till att 'fall', det vill säga att servostyrning och motorstyrning helt stängs av. Först när raketen nått en höjd av 15 m sätts styrningen på igen och får i uppgift att landa raketen på koordinaten (0,0). Under hela sekvensen försöker regulatorerna styra raketens riktning (illustrerat med 'Direction [rad]' i graferna) till att vara 0, vilket är ekvivalent med att raketen pekar rakt upp. Till och börja med testas en P-regulator, det vill säga en regulator vars överföringsfunktion kan skrivas enligt nedan (44).

$$F(s) = K_p \tag{44}$$

Denna regulatorn tar endast hänsyn till det nuvarande felet och agerar proportionerligt mot detta. Med en P-regulator blev resultatet följande:

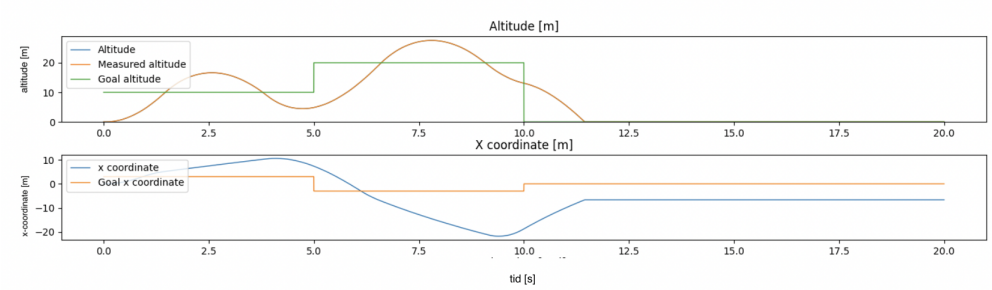


Figur 22: Simulering av raket, P-regulator

Det är tydligt att denna regulatorn inte är tillräcklig för att styra systemet då raketen har svårt att följa referenssignalen samt blir ytterst instabil på grund av avsaknaden av integrerande och deriverande komponenter i regulatorn. Detta är då en P-regulator saknar förmåga att kompensera för kvarstående fel som bland annat kan förklara varför x-koordinaten börjar avvika.

I nästa försök används en PI-regulator. En sådan regulator har följande överföringsfunktion (45).

$$F(s) = K_p + \frac{K_i}{s} \tag{45}$$

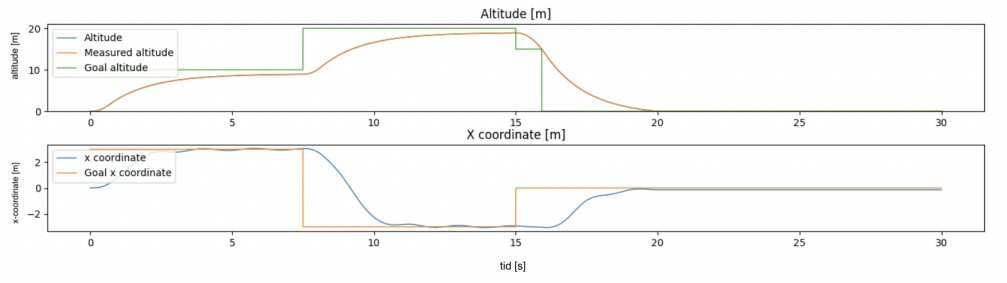


Figur 23: Simulering av raket, PI-regulator

Systemet är fortfarande instabilt, men det har nu en ökad förmåga att följa referenssignalerna utan att avvika till en början. Dock överskjuter systemet referenssignalerna vilket resulterar i ocillationer som inte är önskvärda. Detta kan förklaras med att regulatorn inte har någon del som tar hänsyn till felets derivata som motverkar dessa ocillationer.

Även en PD-regulator undersöktes, det vill säga en regulator med följande överföringsfunktion (46).

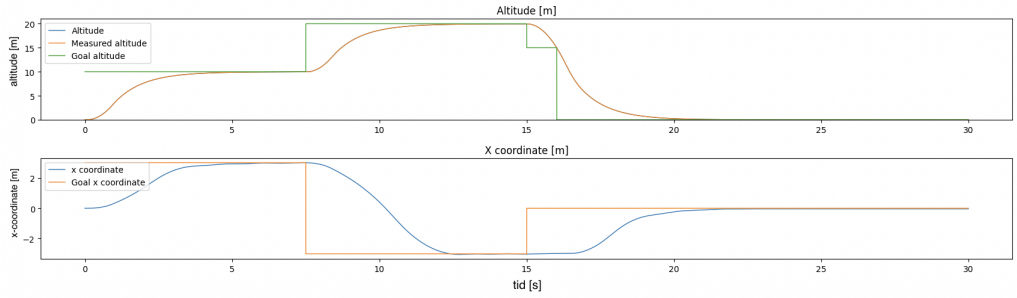
$$F(s) = K_p + K_d s \tag{46}$$



Figur 24: Simulering av raket, PD-regulator

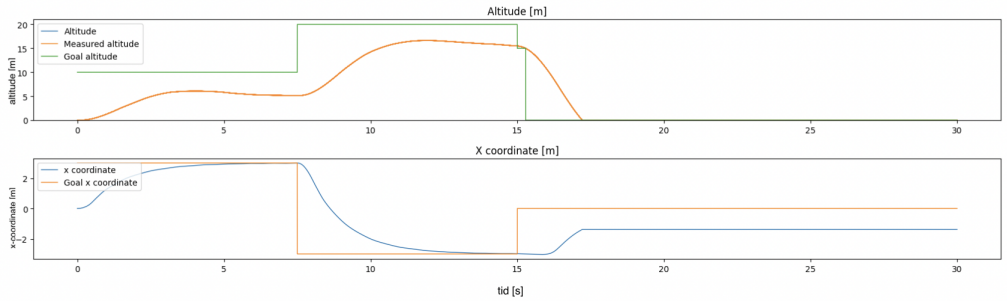
PD-regulatorn var mycket effektiv för att motverka oscillationer och stabilisera systemet. Dock så fanns det alltid ett kvarstående fel vid användingen av denna regulatortyp. Detta kan motverkas med en integrerade konstant i överföringsfunktionen, det vill säga en PID-regulator på följande form (47).

$$F(s) = K_p + \frac{K_i}{s} + K_d s \tag{47}$$



Figur 25: Simulering av raket, PID-regulator

Med en PID-regulator blir det kvarstående felet nästan obefintligt. Av detta skäl väljs en regulator av denna typ i för implementationen av den fysiska modellen. Ett ytterligare test utfördes där ett signifikant mätbrus infördes, till skillnad från tidigare test då det endast användes låga nivåer av brus. Detta mätbrus resulterade i mycket lägre prestanda som kan ses i graferna nedan.



Figur 26: Simulering av raket, med stort mätbrus

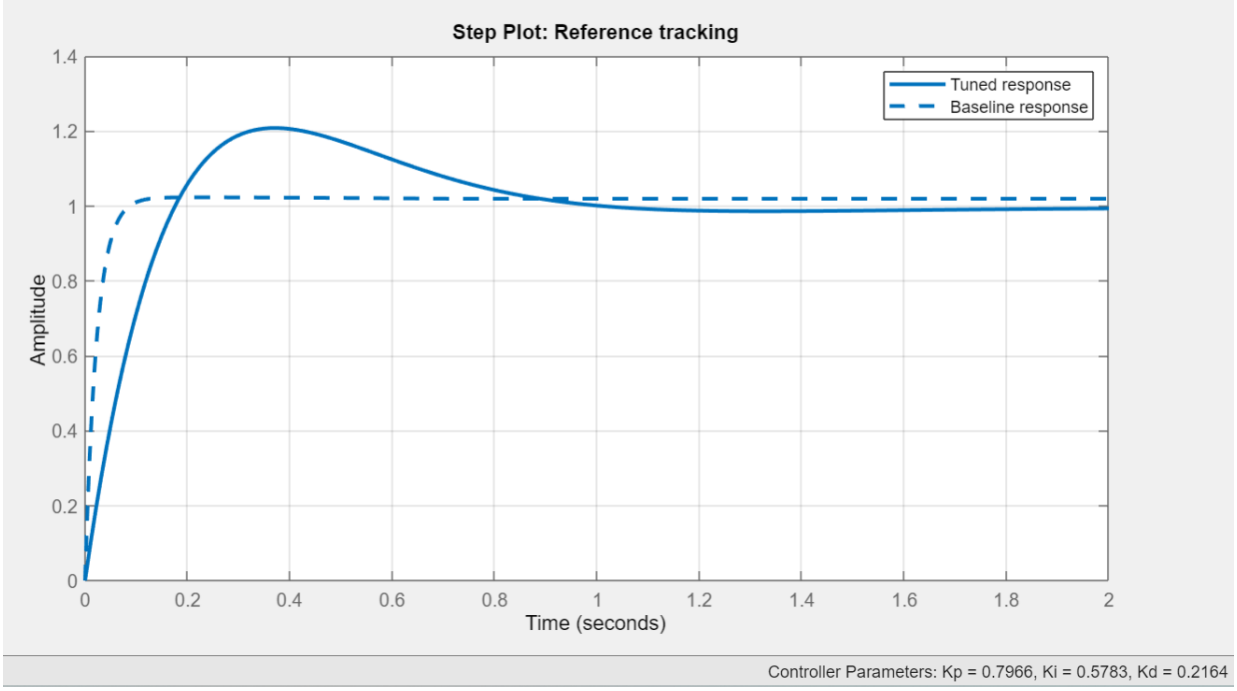
Detta innebär att mängden mätbrus måste reduceras, i detta fall genom att isolera trycksensorn från vind och undvika att montera IMU:n där det uppstår vibrationer. Utöver detta används ett Madgwick-filter, se avsnitt 3.4.2.

6 Resultat

Här beskrivs hur väl den slutgiltiga raketmodellen kan styras, med avseende på reglersystemen och tillhörande aktuatorer, sensorer och filter.

6.1 PID-konstanter framtagna genom matematisk modellering

För att få fram värden till PID-regulatorn används ekvationen för överföringsfunktionen (34). De experimentellt framtagna värdena för tröghetsmoment från ekvation (35), höjd för masscentrum från ekvation (8) och en lyftkraft på 30 N användes ett MATLAB-program, vilket finns i avsnitt A.5. De värden som programmet gav syns i figur 27. Dessa värden testades men funkade inte bra i verkligheten.



Figur 27: Framtagna värden för PID-regulator med hjälp av MATLAB

K_p	0,80
K_i	0,58
K_d	0,22

Tabell 5: PID-konstanter till stabilitetsregulator framtagna genom matematisk modellering

6.2 Reglering av raketmodellens stabilitet

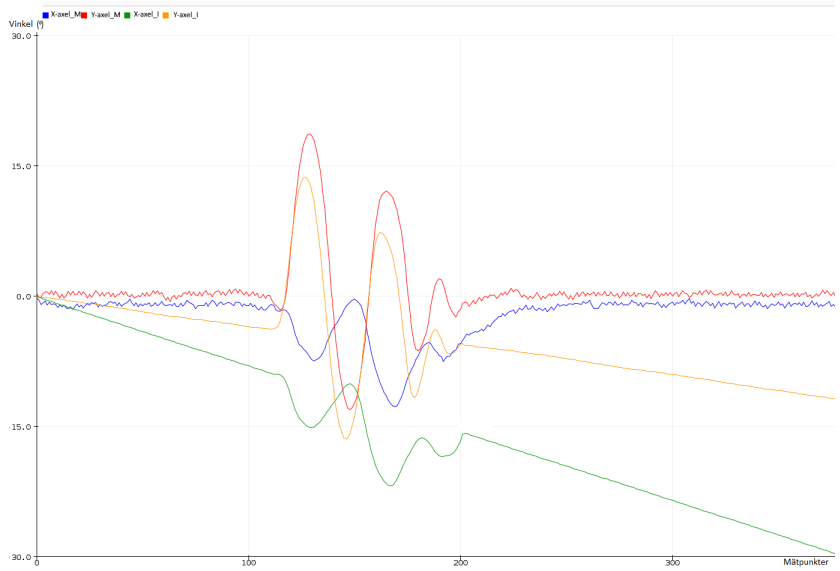
För att upprätthålla stabilitet hos raketmodellen behövde först och främst dess rotation motverkas. Detta åstadkoms delvis genom att använda två propellrar som roterar i motsatt riktning för att motverka varandras vridmoment. Det observerades under tester att detta inte var tillräckligt för att hålla

raketmodellen stabil. Därför behövde roderstyrning implementeras. Denna styrning åstadkoms med en P-regulator där P-konstanten K_p bestämdes experimentellt genom observationer och redovisas i tabell 7. För att inte överskrida vinklar som börjar skapa mycket turbulens och sluta styra luftflödet så sätts begränsningar på hur stora vinklar regulatorn kan ge, dessa presenteras också i tabell 6.

Tabell 6: PID-konstanter till rotationssregulator samt min- och maxvinklar

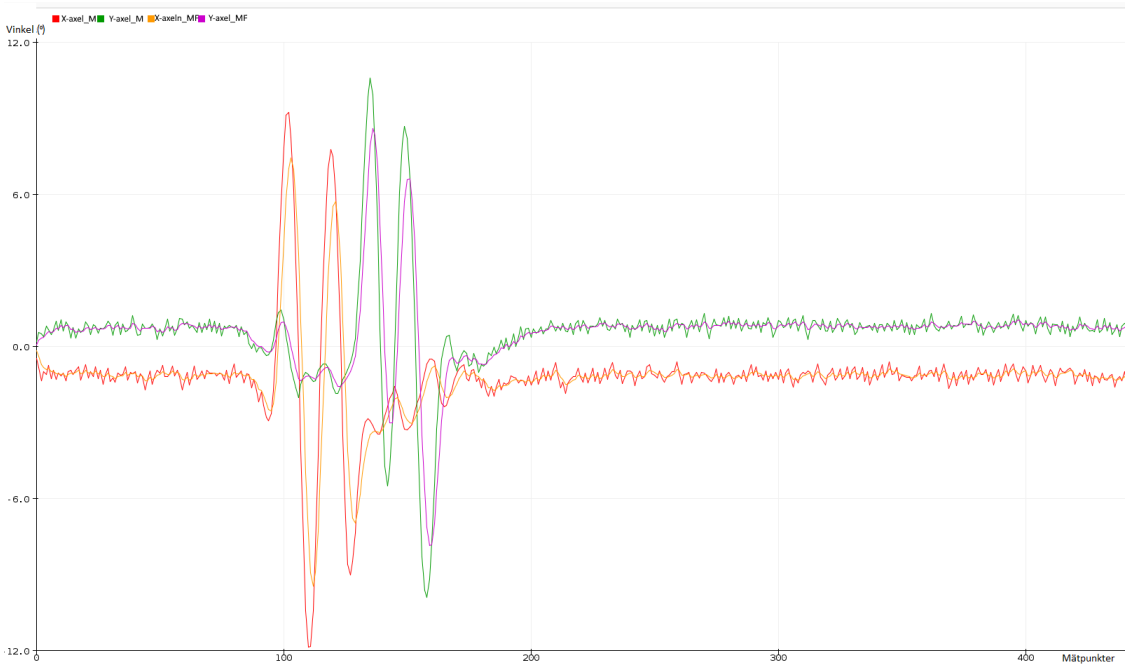
K_p	0,3
Maxvinkel	10°
Minvinkel	-10°

När signalen innehållande vinkelhastighet från IMU:n integrerades med ekvation (39) till den vinkel modellen roterats runt de tre axlarna upptäcktes att ett mätfel introducerats. Detta mätfel växte obegränsat med tiden och redovisas i figur 28 som den gröna grafen för rotation kring x-axeln och den gula grafen för rotation kring y-axeln. När ett Madgwickfilter applicerades på de uppmätta värdena från IMU:n eliminerades det växande mätfelet och resultatet redovisas i figur 28 som den blåa grafen för rotation kring x-axeln och den röda grafen för rotation kring y-axeln.



Figur 28: Uppmätt rotation kring x- och y-axel med Madgwickfilter (_M) och med hjälp av att integrera vinkelhastigheten (_I)

Ur figur 28 framgår att Madgwickfiltret har en stor mängd mätbrus på utsignalerna. Detta är inte önskvärt då detta har negativ inverkan på utsignalen från regulatorn då den deriverade delen av PID-regulatorn är känslig för mätbrus. Detta löstes med ett lågpassfilter och redovisas i figur 29, där signalerna som slutar på _M de som endast har ett Madgwickfilter applicerat på sig, medan signalerna markerade med _MF även har ett lågpassfilter applicerat på sig.

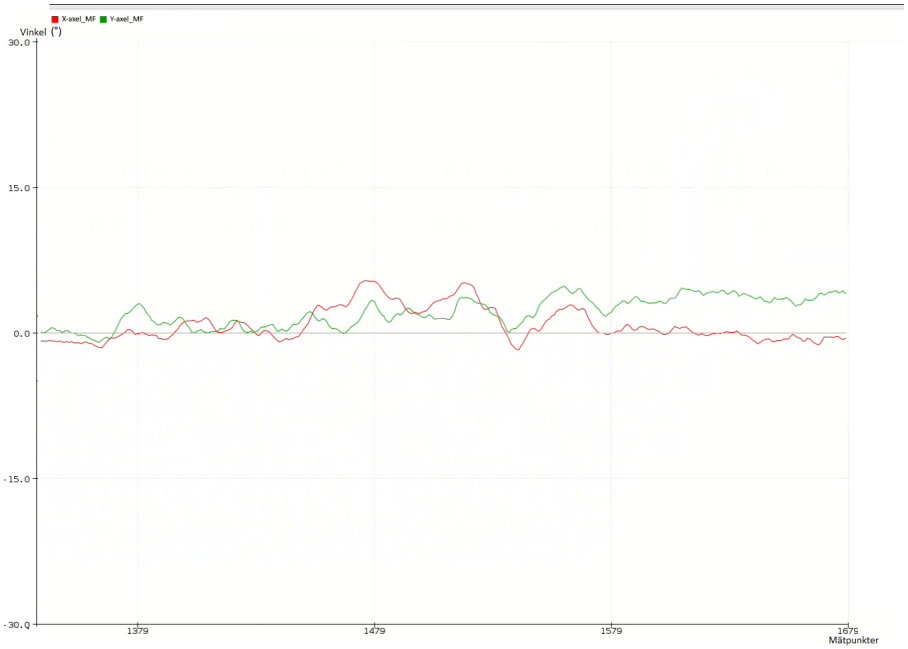


Figur 29: Mätbrusfiltrering med och utan lågpassfilter

Tester som utfördes enligt principer som presenteras i Ziegler-Nicholsmetoden, vilken beskrivs i avsnitt 2.6, resulterade i att PID-konstanterna enligt tabell 7 hittades. Resultatet av att de konstanterna implementerades och testades i testställningen i figur 19 redovisas i figur 30. Raketen är relativt bra på att hålla en upprätt position även om den är monterad i en instabil position. På samma sätt som för rotationsregulatorn så har även har min- och maxvinklar implementerats här för att inte stoppa luftflödet.

Tabell 7: PID-konstanter till stabilitetsregulator samt min- och maxvinklar

K_p	3,00
K_i	0,08
K_d	55,00
Maxvinkel	40°
Minvinkel	-40°



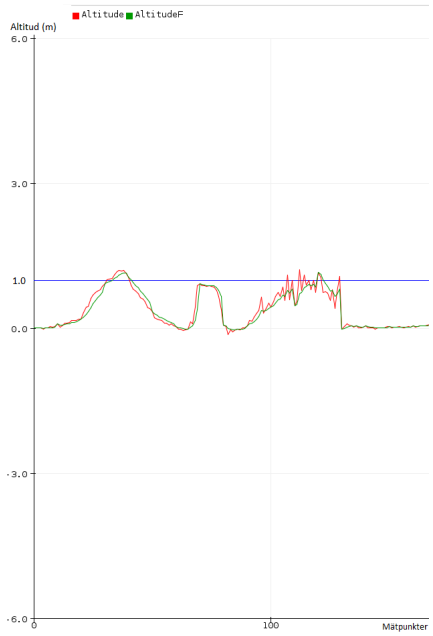
Figur 30: Resultat av raketmodellens stabilitetsregulator

6.3 Reglering av raketmodellens höjd

Som tidigare nämnts används en tryck- och temperatursensor för att estimera altituden av raketmodellen med hjälp av ekvation (41). I kombination med denna används även ett lågpassfilter för att filtrera ut de högfrekventa mätbrusen och få en mer stabil signal att reglera höjden efter. Det värde som valdes på τ i ekvation (42) var 0.5, vilket resulterade i nedan ekvation:

$$G(s) = \frac{1}{0.5s + 1}, \tag{48}$$

Resultatet av att implementera ekvation (48) framgår i figur 31. Under detta test lyftes sensorn upp till cirka 1 m, först i en relativt långsam takt, sedan snabbare och slutligen med mycket vibrationer på cirka 0.1 m i amplitud, vilket förväntas under en flygtur. Den blåa linjen i figuren visar en konstant altitud på 1 m, den röda linjen är den ofiltrerade signalen och den gröna linjen är signalen där ekvation (48) har implementerats och filtrerat bort mätbrus. Det framgår tydligt att mätbruset i princip helt eliminerat i den högra delen av figur 31, där sensorn lyftes upp med mycket vibrationer.



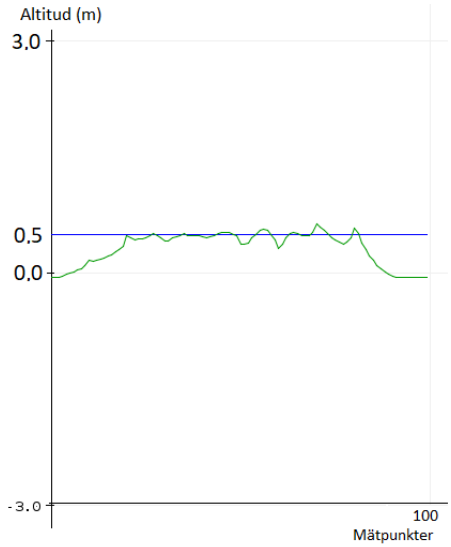
Figur 31: Test att höja trycksensor till 1 m (blå) med filtrering (grön) och utan filtrering (röd)

En signal till ESC:n på 1100 motsvar att motorn ska hålla hastigheten 0 rpm samt att 1940 motsvarar maximal rotationshastighet, därav agerar dessa värden som gränsvärden för PID-regulatorn. Övriga värden som bestämdes med Ziegler-Nicholasmetoden enligt testet i figur 20 redovisas i tabell 8.

Tabell 8: PID-konstanter till höjdregulator

K_p	450
K_i	100
K_d	200
Min-signal	1100
Max-signal	1940

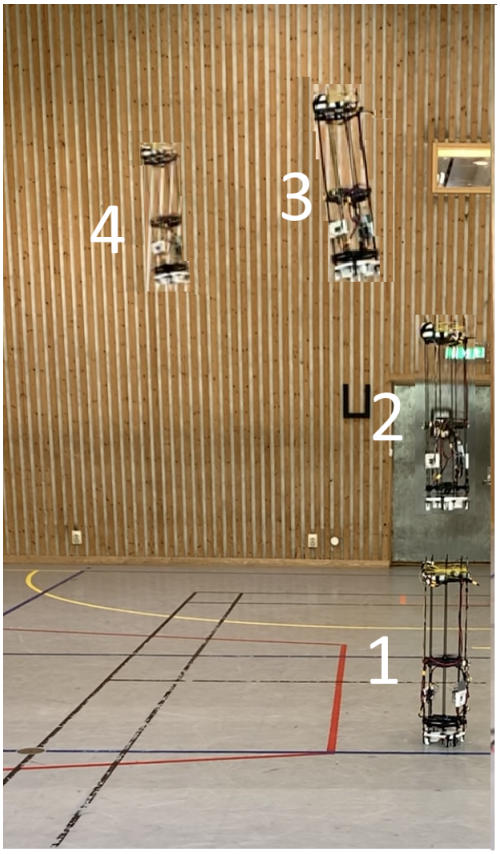
I figur 32 syns den loggade datan där raketmodellen når 0,5 m höjd, för att sedan återvända till marken. Det framgår tydligt ur figur 32 att höjdreglering lyckats väl.



Figur 32: Test av höjdregulatorn för att nå upp till 0,5 m höjd

6.4 Verifiering av reglersystem genom testflygning

I figur 33 syns vad som skedde när raketmodellen utförde en testflygning där regleringen från avsnitt 6.2 och 6.3 var implementerad. Här var målet att raketmodellen skulle nå en höjd av 3 m, för att sedan återvända till marken. Det framgår att modellen stiger kontrollerat och det observeras under testet att modellens rotation motverkas väl. Det är tydligt att det inte finns någon reglering i sidled implementerad, vilket resulterar i att raketmodellen driver iväg till vänster i figuren. Vidare observeras att höjdregleringen fungerar tillräckligt bra för att raketmodellen ska nå önskad höjd, för att sedan närma sig marken. Under detta test hann raketmodellen aldrig nå marken, då den kolliderade med en vägg innan dess. Det observerades under testet att raketmodellen fortsatte tappa höjd ända fram till kollisionen.



Figur 33: Testflygning av raketmodellen i bildsekvens

Hela testflygningen kan ses på bilaga A.1.

7 Diskussion

I detta avsnitt diskuteras hur den färdiga modellen blev samt de komponenter som monterades. Det gås även igenom hur väl regleringen fungerade och vad det resulterade i samt utmaningar som uppstod. Vidare ges förslag på vilka förbättringar som kan göras för att lösa befintliga problem, för att den ska kunna utföra mer avancerade manövrar.

7.1 Färdig modell

Den färdiga modellen innehåller de komponenter som är nödvändiga för att utföra de flygtester som arbetet satsade på att utföra. Raketmodellen uppnår en lyftkraft som innebär att den kan lyfta men också landa från ett fritt fall. Då den primära planen var att få raketen att kunna landa från fritt fall som bromsas upp av raketkroppens luftmotstånd så designades modellen på så sätt att den skulle ha lagom mycket lyftkraft för att klara även detta. Då det var ett problem i början av projektet att stor del av lyftflödet gick till att motverka rotationen av raketen gjordes valet att montera en till likadan motor längre upp i raketmodellen. Denna motor roterade motsatt riktning vilket nästan löste det initiala rotationsproblemet. Detta ökade dock raketmodellens massa men eftersom BLDC motorerna som valdes var av en kraftfull modell innebar det att luftflödet som tillkom var avsevärt mer är vikten som raketmodellen ökade med.

Utifrån ekvation (1) och (2), kan raketens sluthastighet tas fram vid fritt fall, v_0. Denna fås från raketens luftmotståndskoeficient, c_d vid fall med framsidan av raketen först, raketens massa m, raketens area parallellt mot marken A, samt luftens densitet vid marknivå, $\rho_{luft} \approx 1.3$. När v_0 tagits fram kan även lyftkraften som krävs för att bromsa raketen till stillastående från sträckan s från marken beräknas. Om uppskattningar på dessa värden görs utifrån det som är känt om modellen från preliminära beräkningar så fås $c_d = 0.8$ då modellens ytterhölje approximeras till en cylinder, $m = 2.355$ kg utifrån de komponenter som planeras att användas, $A = 0.2$ m^2 samt att raketen vänts med luftflödet genererat av motorn riktat mot marken vid $s = 14$ m fås:

$$v_0 \approx 15m/s$$
$$F_{Thrust} \approx 42N$$

Utifrån detta är det tydligt att en motoruppsättning som kan producera minst 42 N lyftkraft är önskvärt. Då nuvarande modell väger 2.36 kg och dess producerade lyftkraft är 4.32 kg, vilket motsvarar 42.4 N, ligger raketen strax över denna nivå. Detta betyder att modellen har potential att klara av denna manövern i framtida projekt.

7.2 Reglering av modell

De värdena som togs fram med hjälp av beräkningar och MATLAB som redovisades i avsnitt 6.1 funkade väldigt dåligt därför togs de PID värdena som användes fram experimentellt istället. Anledningen till detta kan vara att testerna för att ta fram raketens massströghetsmoment inte var exakta nog, eller att ekvation (34) inte var representativ nog av systemet som utvecklades i detta arbete. Till exempel antar denna ekvationen att masscentrum ligger på raketens centerlinje, vilket inte nödvändigtvis är sant för raketmodellen även om detta försökte uppnås under konstruktionen. Dessutom är denna ekvationen en förenkling av en mer komplex överföringsfunktion. I ekvationen antas det även

att lyftkraften kan omdirigeras exakt till den önskade vinkeln vilket stämmer för en raketmotor, men inte för ett luftflöde försöker omdirigeras med hjälp av roder. Till exempel kommer inte allt luftflöde som genereras av motorena flöda över rodrena i raketmodellen, utan bara fortsätta vinkelrätt mot propellerns rotationsplan. Mycket av flödet över rodren kommer förmodligen inte omdirigeras till rodrens precisa vinkel utan kommer istället bilda turbulens. Detta är förmodligen varför PID-konstanterna som gavs av beräkningarna i MATLAB var mycket lägre än dem som användes i verkligheten, då rodrena behöver göra större utslag för att uppnå önskad stabilisering.

Sidregleringen har inte implementerats under detta projekt. Anledning till detta är att en testrigg som tillåter lätt testning och finjustering av sidregleringen inte har utvecklats. Detta är anledningen till att raketen inte lyckades landa under testflygningarna vilket går att se i avsnitt A.1. Om en välfungerande sidreglering hade varit implementerad hade driften in i väggen gått att undvika. Utveckling av denna reglering har påbörjats men har ej testats eller implementerats på den verkliga modellen. Den går dock att se i simuleringen då det går att se hur raketen kan förflytta sig i sidled och parera för att förhindra förflyttelse i sidled. Dock fungerade höjdregleringen, stabilitetsregleringen samt rotationsregleringen bra nog att utföra en kontrollerad stigning och påbörja en landningsekvens.

7.3 Testflygningar av färding modell

Flygningarna som utfördes när alla delsystem var testade gjordes i olika miljöer dels med säkerhetslinor men också helt fritt. För att minimera risken för skador påbörjades flygtester med säkerheterslinor fästa i toppen av raketmodellen. Detta gav möjligheter att ser hur väl alla system fungerade tillsammans för att sedan kunna justeras ytterligare om nödvändigt. Vad som tidigt blev uppenbart var att rotationsreglering var nödvändigt då det inte var tillräckligt att bara låta de två BLDC motorerna snurra i motsatt riktning. Det som var svårt att avgöra när raketmodellen hade säkerhetslinor var hur väl den behöll sin position utan att driva åt något håll. Detta var oundvikligt eftersom linan skulle ha som funktion att förhindra eventuella krascher genom att begränsa raketens rörelser.

När testflygningarna genomfördes utan testlinor var risken att raketmodellen skadades avsevärt större vilket innebar beslutet tog att genomföra korta flygningarna till en början för att skadorna inte skulle bli allt för omfattande om något skulle gå snett. Detta gav dock problem, raketenmodellen påverkades av det egna luftflödet som träffade marken för att sedan studsa tillbaka upp mot raketmodellen. För att undkomma detta togs beslutet att flyga raketen till en högre höjd. Vid testet på 3 m höjd (se bilaga A.1) så uppkom problem med att sidregleringen ej var implementerad vilket resulterade i att raketmodellen drev in i väggen innan den kunde landa. Flygprofilen följde det programmet som kodats in vilket anses vara framgångsrikt.

7.4 För- och nackdelar med eldriven framdrift

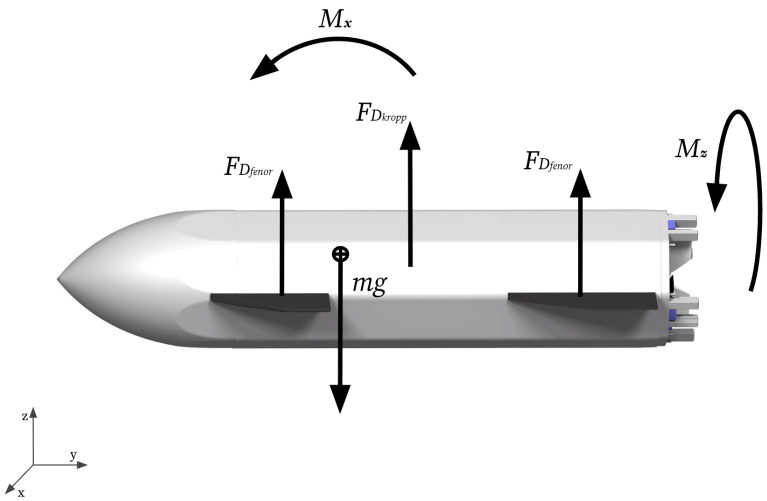
Raketmodellen var begränsad till att ha en elektriskt framdrivning, istället för en raketmotor. Detta gav i sin tur fördelar och nackdelar. Den största fördelen är säkerheten samt enkelheten kring att arbeta med elmotorer och lithiumjonbatterier jämfört med raketmotorer och raketbränsle. Även fast det finns vissa risker med lithiumjonbatterier är dessa försumbara jämfört med handhavande av raketmotorer. Dessutom är det avservärt enklare att kontrollera lyftkraften på en elmotor i form av att ändra varvtalet jämfört med en raketmotor, då raketmotorer för modellraketer saknar styrning av lyftkraften, vilket skulle innebära maximal lyftkraft levereras vid alla tillfällen som raketmotorn är igång.

Nackdelen med elmotorn är att lyftkraften som kan genereras blir avsevärt mycket lägre jämfört med en raketmotor i närliggande vikt och fysiska dimensioner. Om målet med en raketmodell är att ta sig

långt upp i atmosfären eller till och med upp till rymden där vakuum existerar krävs det en motor som självskjuter ett medie som den kan reagera mot, precis som en typisk raketmotor gör. En elmotor blir alltså fysiskt omöjlig att använda i dessa miljöer.

8 Vidare arbete

Projektet har stegvis skalats ner från vad som initialt planerades. Gällande modelleringen valdes det att utesluta fenor på raketens sidor, samt en konformad nos på raketens topp. Det gjordes dels av tidsskäl, samt att gruppen bedömde att dessa delar inte var de mest centrala delarna för att fullfölja projektet. Simuleringar och beräkningar gjordes däremot med både fenor och nos. I simuleringen simulerades till exempel fallet på samma sätt som i figur 34.



Figur 34: Krafter och moment vid fritt fall (2D)

Under fallet behöver den vertikala positionen av raketmodellen regleras, då den ska falla med framsidan ner, se Figur 34. Då kommer fenorna på sidorna kunna vinklas relativt raketkroppen för att ändra den aerodynamiska karaktäristiken av modellen. När fenorna är vinkelräta mot luftflödet kommer de att uppleva högre luftmotstånd och när de är mer indragna mot kroppen, det vill säga vinkeln mellan kroppen och fenorna är mindre, kommer luftmotståndet att minska på fenorna. Var och en av fenorna kommer därmed uppleva en kraft vinkelrätt mot luftflödet som kommer att generera moment kring raketens centerlinje (M_2) och raketens masscentrum (M_1). Detta ger upphov till följande ekvationer:

$$\uparrow: \; F_{d\_fena1,z} + F_{d\_fena2,z} + F_{d\_fena3,z} + F_{d\_fena4,z} + F_{d\_kropp,z} - mg = m\ddot{z} \tag{49}$$

$$\rightarrow: \; F_{d\_fena1,y} + F_{d\_fena2,y} + F_{d\_fena3,y} + F_{d\_fena4,y} + F_{d\_kropp,y} = m\ddot{y} \tag{50}$$

$$\hat{M_1}: (F_{d\_fena3} + F_{d\_fena4}) \cdot d_2 + F_{d\_kropp} \cdot d_3 - (F_{d\_fena1} + F_{d\_fena2}) \cdot d_1 - J_x\ddot{\phi}_x = M_1 \tag{51}$$

$$\hat{M_2}: \; (F_{d\_fena1} + F_{d\_fena3}) \cdot r - (F_{d\_fena2} + F_{d\_fena4}) \cdot r - J_y\ddot{\phi}_y = M_2 \tag{52}$$

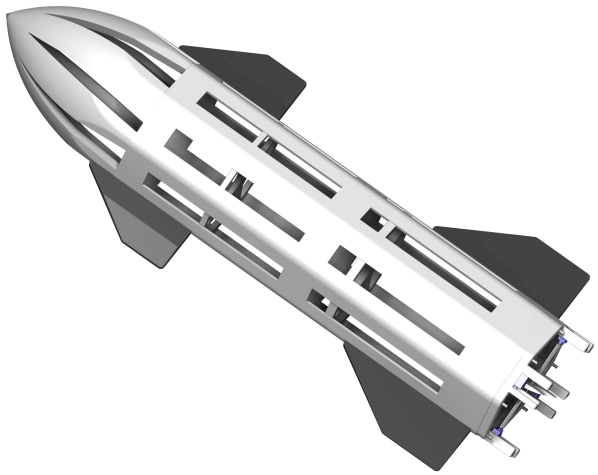
Där indexeringen på fenorna är sådan att index 1 och 2 är fenorna vid toppen av raketen samt index 3 och 4 är fenorna vid botten av raketen. J_x respektive J_y är massträghetsmomenten kring axel x och y samt $\ddot{\phi}_x$ och $\ddot{\phi}_y$ är vinkelaccelerationen kring x- och y-axeln som illustreras i Figur 34. Krafterna delas sedan in i sina komponenter i horisontal och vertikalled (med index y respektive z) för att beskriva modellens rörelse. Kraften F_d som genereras kan förenklat beskrivas av följande formel:

$$F_d(v) = c_d A \frac{\rho_{luft} v^2}{2} \tag{53}$$

Där F_d är kraften från luftmotståndet för varje fena samt raketens kropp, c_d är luftmotståndskoefficienten för fenan/kroppen, A är arean vinkelrätt mot luftflödet, ρ_{luft} är luftens densitet vid marknivå och v är hastigheten som luften rör sig med relativt modellen, i det här fallet fallhastigheten. De två sätt för att kontrollera fallet som är aktuella för prototypen är antingen att varje fena kan styras oberoende av varandra, vilket i sin tur skulle kunna ge upphov till ett moment kring y-axeln, illustrerat med M_x i Figur 34. Även ett moment M_x skulle kunna genereras. Detta skulle kunna vara fördelaktigt om modellen visar sig vara instabil kring y-axeln då den faller. Det andra alternativet skulle vara att reglera fenorna parvis, det vill säga att de övre respektive undre fenorna kontrolleras separat men att paren av fenorna speglar varandras rörelser. Detta skulle kunna användas för att generera momentet M_x men inget moment kring y-axeln, vilket skulle vara fördelaktigt då färre servos skulle behövas för konstruktionen, men då förutsätts det att raketen är naturligt stabil kring y-axeln.

En regulator för att styra fallet påbörjades men nådde aldrig full funktion i simuleringen, vilket skulle kunna vidareutvecklas i framtida projekt för att skapa förståelse för hur detta skulle kunna implementeras på raketmodellen, se bilaga A.4.

Eventuella efterföljande arbeten inom samma område skulle kunna ta fram och utveckla nos, ytterhölje och fenor, enligt de CAD-modeller som presenterats i detta arbete. Frånvaron av ytterhölje och nos bidrar till ett högre luftflöde, som diskuterats i avsnitt 4.1.2. Detta innebär att addering av dessa komponenter kan kräva eventuella justeringar, såsom de har gjort i figur 35, för att maximera luftflödet och därmed lyftkraften. I figuren har luftspalter implementerats i nos och ytterhölje för att optimera luftflödet.



Figur 35: Förslag på vidareutvecklad raketmodell, med luftspalter i ytterhölje och nos

Med fenor kan regleringen förbättras och gör raketmodellen mer stabil även vid statisk montering, vilket gör det till en betydande förbättringsaspekt. Dessutom har fenorna en central roll om modellen ska utvecklas så den faller med sidan nedåt, se avsnitt 2.3 om styrning. För att kunna utföra ett fall med sidan nedåt är det fördelaktigt att implementera ett ytterhölje på modellen. Detta skulle leda till

en lägre fallhastighet som följd av det ökade luftmotståndet, vilket i sin tur minskar den lyftkraft som krävs för landning.

Vidare skulle regulatorn för kontroll i sidled behöva utvecklas och PID-konstanter behöver bestämmas. Denna regulator finns implementerad och kan hittas i bilaga A.6, men den hann aldrig testas och optimeras. Utveckling av denna skulle förhindra att raketmodellen driver iväg som visat i avsnitt 6.4.

Fortsättningsvis skulle det även vara bra om en metod utvecklades för att logga signaler från sensorerna och aktuatorerna för lättare felsökning. Detta skulle göra det lättare att jämföra de olika resultaten med varandra och det skulle bli lättare att förstå varför modellen beter sig som den gör. Detta är ett väsentligt steg för att kunna finjustera all reglering som krävs för att erhålla en fullt autonom raketmodell.

9 Slutsats

Utifrån de frågeställningar som fastställdes har det påvisats att det som krävs för att utföra en kontrollerad stigning till en önskad höjd och sedan återvända för att landa på marken är ett antal komponenter i kombination med mjukvara för styrsystemen. För att nå den önskade höjden krävs sensorer som mäter höjd och en regulator som agerar på den uppmätta altituden och beräknar vilken lyftkraft som ska ges. I detta projekt åstadkoms detta med en trycksensor i kombination med en Arduino som styrdator och en PID-regulator för regleringen. Det var även väsentligt att ha någon form av styrning för att hålla raketmodellen i en upprät position, då den som ett instabilt system direkt förlorar kontrollen över sin färdriktning utan detta. Det som krävdes för att implementera denna styrning var en IMU som i kunde återge raketens vinkelhastighet runt sina axlar och acceleration längs axlarna i kombination med en styrdator (även i detta fall en Arduino) samt en PID-regulator. Det visade sig även vara mycket svårare att få noggranna brusfria mätvärden från sensorerna än väntat, vilket gjorde mätbrusfiltrering med lågpassfilter och sensororienteringalgoritmer som Madgwickfiltret nödvändigt vid utvecklingen av styrsystemen. Lyftkraften som kunde produceras av raketens drivsystem i kombination med raketens egna vikt var även detta väldigt viktigt då detta hade potentialen att ge mer kontrollauktoritet ju högre lyftkraften var i proportion till tyngdkraften, vilket gjorde val av elektriska komponenter och materialval essentiella för projektets framgång.

Slutligen kan det konstateras att möjligheten att konstruera en raketmodell med ett reglersystem kapabelt att klara av lyftningar och landningar är fullt genomförbar. Även om en helt lyckad flygsekvens precis som den riktiga raket skall utföra inte hanns med i detta projekt så var testresultaten lovande. Med den informationen som presenterats i denna rapporten så finns det möjlighet för framtida projekt att slutföra implementeringen av raketmodellens styrsystem med reglering för raketmodellens position i sidled och förbättrad filtrering av sensorvärden.

Referenser

[1] Rymdstyrelsen. (2019-08-30). *Hur fungerar en raketuppskjutning?*. https://www.rymdstyrelsen.se/upptack-rymden/bloggen/2019/08/hur-fungerar-en-raketuppskjutning/

[2] SpaceX. (Hämtad 2023-05-26). *Falcon 9 Rocket* https://www.spacex.com/vehicles/falcon-9/

[3] SpaceX. (Hämtad 2023-05-26). *Starship*. https://www.spacex.com/vehicles/starship/

[4] NASA.(Hämtad 2023-05-26).*Terminal velocity Interactive*. https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/termvel/

[5] Flite test. (2013-09-18). *Propeller Static & Dynamic Thrust Calculation*. https://www.flitetest.com/articles/propeller-static-dynamic-thrust-calculation

[6] FAA. (2016). *Pilot's Handbook of Aeronautical Knowledge.* https://www.faa.gov/regulations_policies/handbooks_manuals/aviation/phak/media/08_phak_ch6.pdf

[7] TYTO Robotics. (2022-06-21). *How to Calculate Electric Motor Torque 2023*. https://www.tytorobotics.com/blogs/articles/how-to-calculate-electric-motor-torque

[8] Stanford. (Hämtad 2023-05-26). *Thrust Vector Control of a Hypersonic Re-entry Vehicle* https://charm.stanford.edu/ENGR1052016/CharlesCoxZacClausing

[9] Microstar Laboratories (Hämtad 2023-05-26). *Ziegler-Nichols Tuning Rules for PID* https://www.mstarlabs.com/control/znrule.html

[10] Ziegler, J.G Nichols, N. B. (1942). *Optimum settings for automatic controllers.* Transactions of the ASME. 64: 759–768. https://web.archive.org/web/20170918055307/http://staff.guilan.ac.ir/staff/users/chaibakhsh/fckeditor_repo/file/documents/Optimum%20Settings%20for%20Automatic%20Controllers%20(Ziegler%20and%20Nichols,%201942).pdf

[11] How to mechatronics. (2019-03-14). *How Brushless DC Motor Works? BLDC and ESC Explained* https://howtomechatronics.com/how-it-works/how-brushless-motor-and-esc-work/

[12] Vectornav. (Hämtad 2023-05-26) *What Is An Inertial Measurement Unit?*. https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu

[13] Hyndman, R.J., & Athanasopoulos, G. (2018). *Forecasting: principles and practice, 2nd edition* https://otexts.com/fpp2/index.html

[14] x-io Technologies. (2010-04-30). *An efficient orientation filter for inertial and inertial/magnetic sensor arrays* https://x-io.co.uk/downloads/madgwick_internal_report.pdf

[15] MadgwickAHRS. (2022-10-25). *Madgwick Library* https://github.com/arduino-libraries/MadgwickAHRS

[16] Applied Measurements. (Hämtad 2023-05-26). *Barometric Pressure Sensor | Pa600B*https://appmeas.co.uk/products/pressure-sensors/barometric-pressure-sensor-pa600b/

[17] Keisan Casio. (Hämtad 2023-05-26). *Altitude from atmospheric pressure Calculator* https://keisan.casio.com/has10/SpecExec.cgi?id=system/2006/1224585971

[18] Analog Devices (Hämtad 2023-05-26) *Low-Pass Filter, What is a Low-Pass Filter?* https://www.analog.com/en/design-center/glossary/low-pass-filter.html

[19] NASA. (2012-01-05). *The Tyranny of the Rocket Equation* https://www.nasa.gov/mission_pages/station/expeditions/expedition30/tryanny.html

[20] Liu, Y & Zwingmann, B & Schlaich, M. (2015-10-21). *Carbon Fiber Reinforced Polymer for CableStructures — A Review* https://www.mdpi.com/2073-4360/7/10/1501

[21] McGraw-Hill. (1997) . *Materials Handbook* . Brady, George S., Henry R. Clauser, and John A. Vaccari

[22] Makenica. (2020-09-24). *CPE Filament* https://makenica.com/cpe-filament/

[23] All3DP. (2022-12-03). *3D Printing Infill: The Basics for Perfect Results* https://all3dp.com/2/infill-3d-printing-what-it-means-and-how-to-use-it/

[24] Journal of natural sciences and mathematical research. (2019-12-01). *Measurement of Moment of Inertia Through a Bifilar Pendulum Swing Based on a Microcontroller* https://journal.walisongo.ac.id/index.php/JNSMR/article/view/11028/3930

[25] Engineering ToolBox. (2003). *Atmospheric Pressure vs. Elevation above Sea Level* https://www.engineeringtoolbox.com/air-altitude-pressure-d_462.html

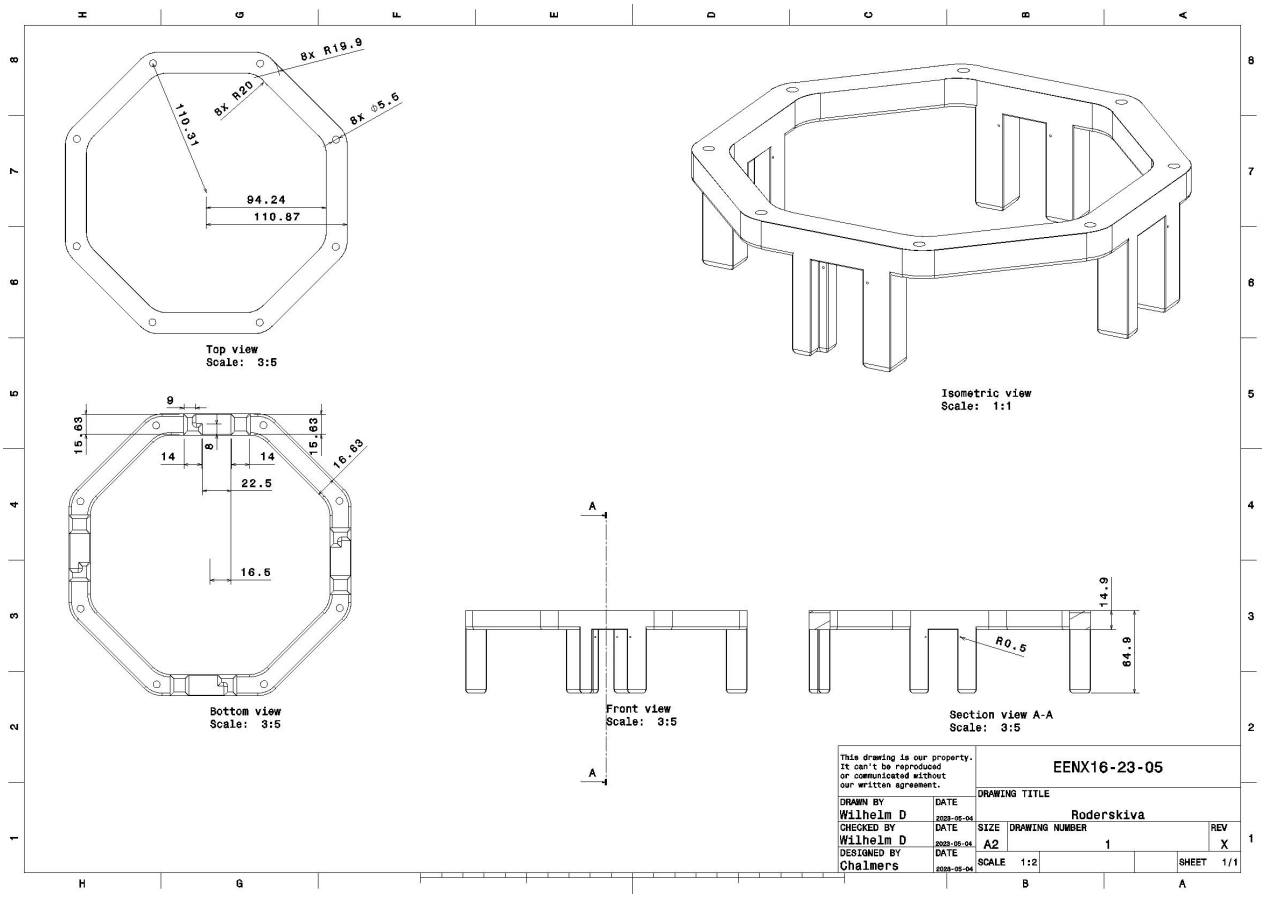
A Bilagor

A.1 Videomaterial från testflygningar

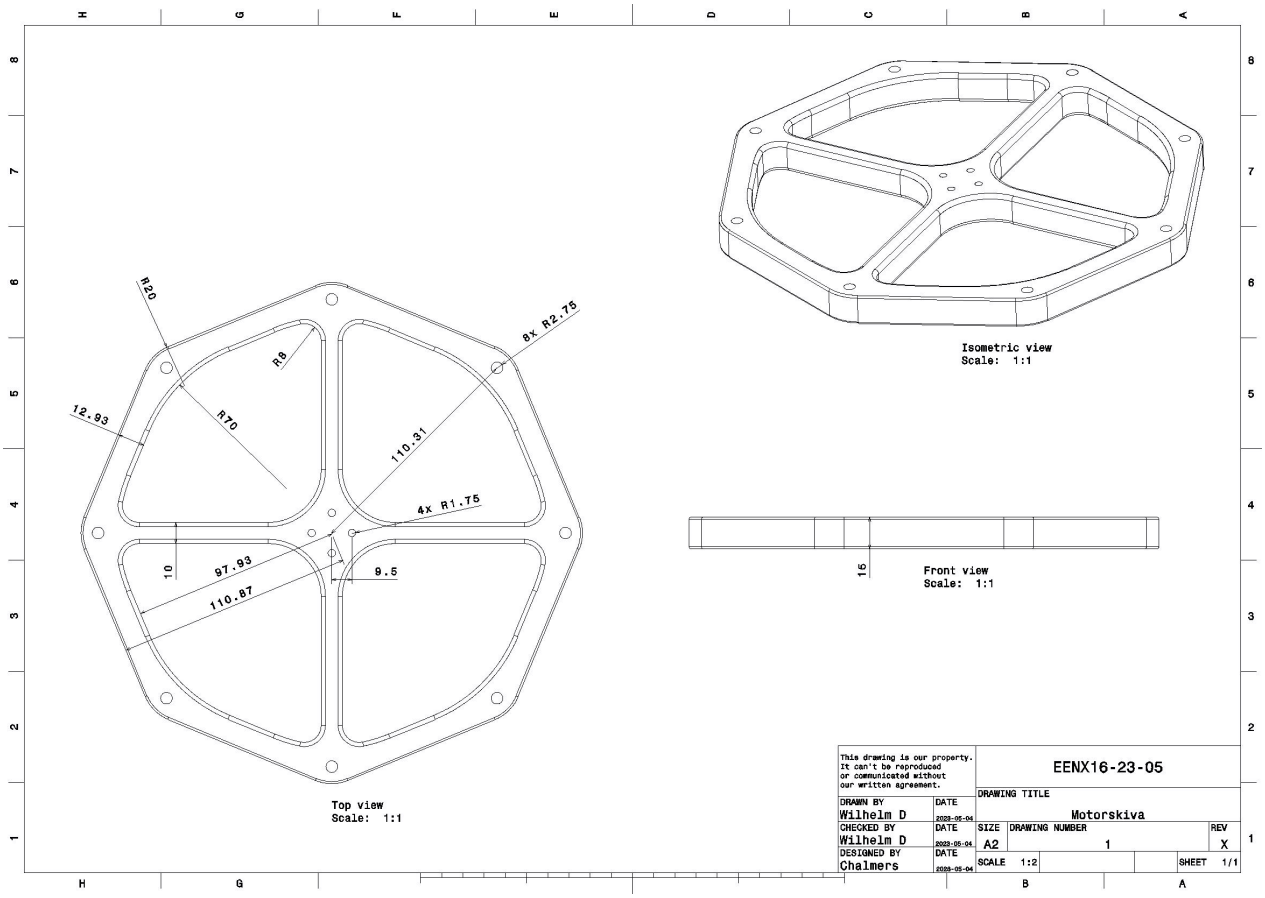
- Video av testflygning:
  https://youtube.com/shorts/BkpVR5XaDJw?feature=share

- Video av flygsimulering:
  https://youtu.be/iFh1bFqX8OE

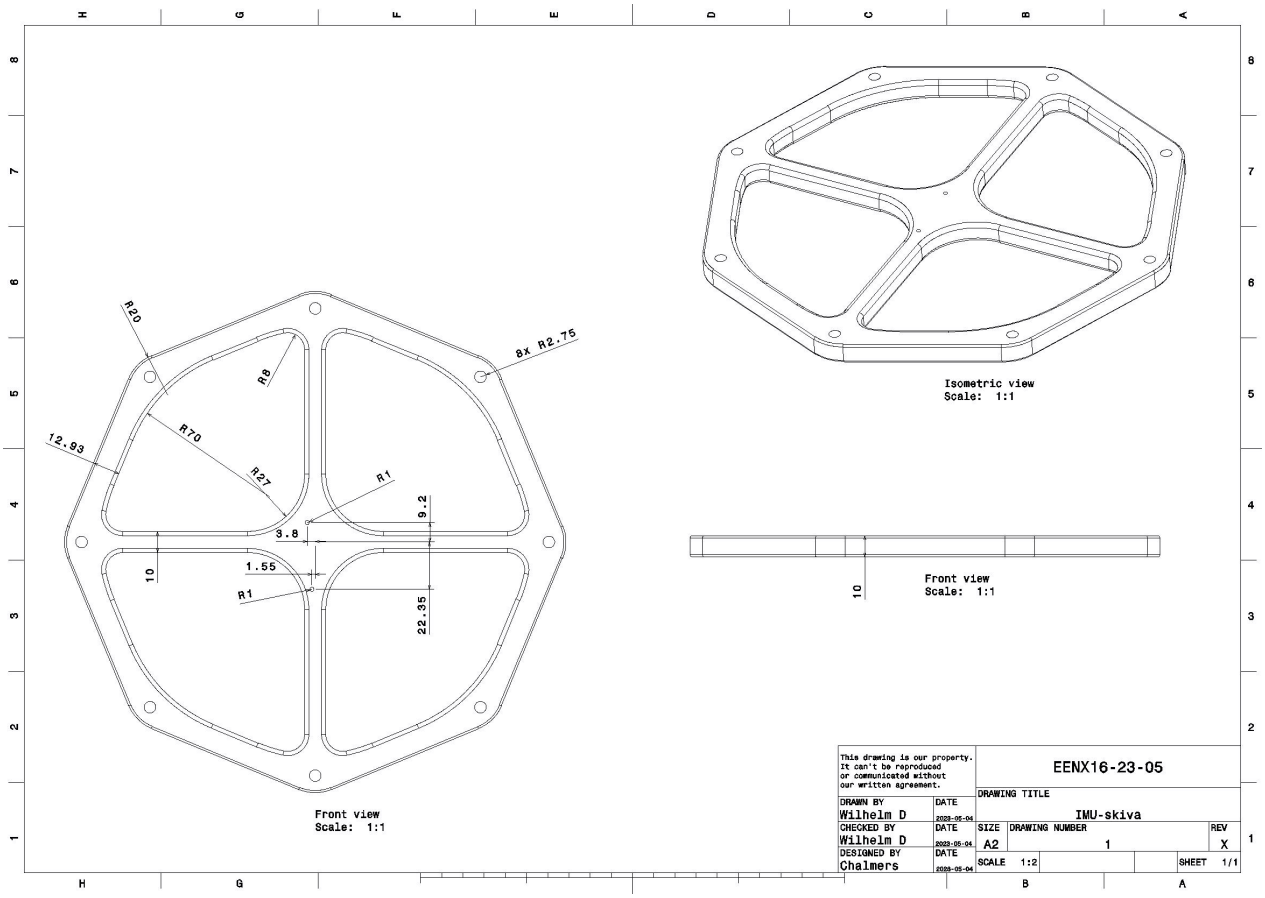
A.2 Ritningar för konstruktionskomponenter



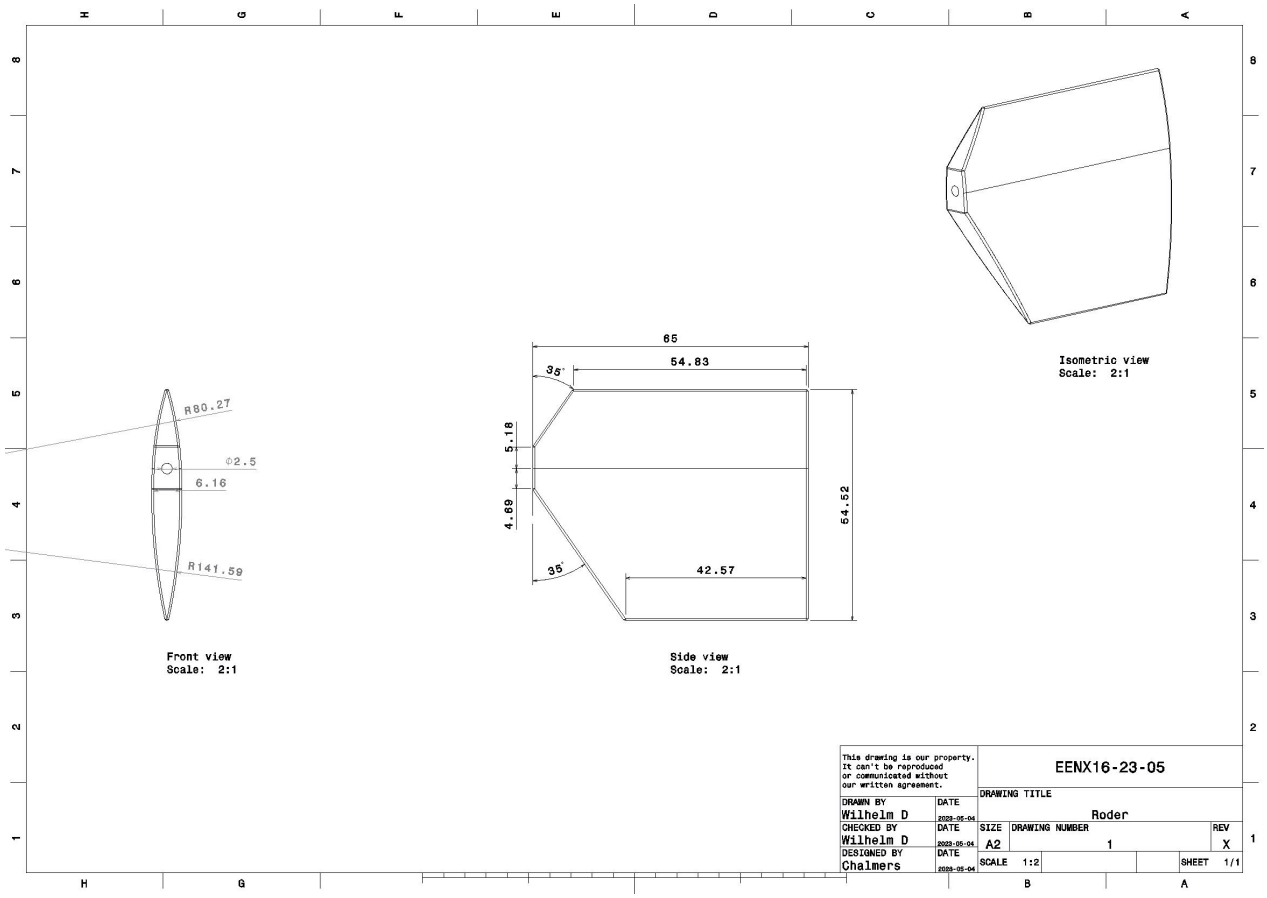
Bilaga 1: Ritning för roderskiva.



Bilaga 2: Ritning för motorskiva.



Bilaga 3: Ritning för IMU-skiva.



Bilaga 4: Ritning för roder.

A.3 Länkar till komponenter

- Motor (T-motor V3008 1350KV):

  https://www.drone-fpv-racer.com/en/v3008-1350kv-motor-by-t-motor-9965.html

- Propeller (Gemfan 8040-3 Glass Fiber Nylon):

  https://www.drone-fpv-racer.com/en/gemfan-8040-3-glass-fiber-nylon-for-cinelifter-macro-quad-9981.html

- Motorbatterier (Tattu FunFly Lipo Battery 3S 1550mAh 100C):

  https://www.drone-fpv-racer.com/en/tattu-funfly-lipo-battery-3s-1550mah-100c-5540.html

- ESC (Hobbywing FlyFun V5 120A 3-8s):

  https://www.elefun.se/p/prod.aspx?v=35885

- Mikrokontroller (Arduino, Uno Rev 3 SMD):

  https://se.rs-online.com/web/p/arduino/7697409

- Batterier till Arduino Uno (Zippy compact 25C 850 mah LiPo 2s):

  https://hobbyking.com/en_us/zippy-compact-850mah-2s-25c-lipo-pack.html?____store=en_us

- Servomotor (Luxorparts SG90):

  https://shorturl.at/mpATZ

- IMU (Grove IMU 9 DOF):

  https://www.elfa.se/sv/grove-imu-dof-seeed-studio-101020585/p/30127499?trackQuery=imu&pos=2&origPos=2&origPageSize=50&track=true

- Trycksensor (Infineon Shield2Go Equipped with XENSIV™ Barometric Pressure Sensor):

  https://shorturl.at/ntEFW

A.4 Simuleringskod i Python

```
class PID:
    def __init__(self, Kp, Ki, Kd, ref):
        self.kp = Kp                            #PID constants to be set when called
        self.ki = Ki
        self.kd = Kd
        self.ref = ref                          #reference signal
        self.error_p = 0
        self.error_i = 0
        self.lower_limit = None                 #Set limits (for example for current
    controller to not overshoot engine/controller/servo specs)
        self.upper_limit = None                 #to be set when controller is called

    def __call__(self, signal, dt):             #dt = timestep
        error_p = self.ref - signal             #proportional error is the difference
    between measured signal and reference signal
        error_d = (error_p - self.error_p)/dt   #take the new measured error and
    subtract the old one divided by the timestep to get the approximate derivative
        self.error_p= error_p                   #update proportional error

        out_signal = self.kp * self.error_p + self.ki * self.error_i + self.kd *
    error_d   #output signal from the controller, generated by PID constants and errors

        self.error_i = self.error_i + error_d * dt  #update integration error by
    eulers method

        if self.lower_limit != None:            #to prevent controller to overshoot
    given limits
            if out_signal > self.upper_limit:
                out_signal = self.upper_limit
            if out_signal < self.lower_limit:
               out_signal = self.lower_limit

        return out_signal

    def change_ref(self, new_ref):
        self.ref = new_ref

    def set_limits(self, lower_limit, upper_limit):
        self.lower_limit = lower_limit
        self.upper_limit = upper_limit
```

Listing 1: PID regulator för simulation

```
from initialPID import PID
import pygame
import math
import numpy as np
import random
import matplotlib.pyplot as plt
from visualization import Sim

#Global parameters-----------------------------------------------------------

#Time parameters

dt = 0.00001 #[s], time step
t = 30 #[s], simulation duration
N = round(t/dt) #number of steps in the simulation
time = np.arange(0, N) * dt

#Rocket
m = 1.8 #[kg]
```

```
nosecone_height = 0.20 #[m]
tube_section_height = 0.64 #[m]
rocket_height = nosecone_height+tube_section_height #[m]
rocket_radius = 0.10 #[m]
center_of_mass = rocket_height/2 #[m]
thrust_height = 0 #[m], rocket thrust generated (allmost) at its bottom
moment_of_inertia_rocket = 1/2*m*rocket_radius**2+1/12*m*rocket_height**2     #
    approximation with thin tubular shell
rocket_body_drag_c = 0.80       #to be found in wind tunnel tests
rocket_belly_area = 2*rocket_radius*rocket_height #[m2]
rocket_body_center_of_preassure = 2*rocket_height/5 #[m] to be found in wind tunnel
    tests

rocket_flap_drag_c = 0.90
bottom_flap_area = 0.025 #[m2]
bottom_flap_center_of_preassure = 0.25/2 #[m]
top_flap_area = 0.020   #[m2]
top_flap_center_of_preassure = tube_section_height-0.20/2 #[m]

rocket_direction = np.zeros(N)   #rad
rocket_angular_vel = np.zeros(N)
rocket_angular_acc = np.zeros(N)
thrust_direction = np.zeros(N)   #rad
accent_velocity = np.zeros(N)
side_velocity = np.zeros(N)
F_thrust = np.zeros(N)
F_thrust_x = np.zeros(N)
F_thrust_tangential = np.zeros(N)
F_thrust_y = np.zeros(N)
rotation_torque = np.zeros(N)
acceleration_x = np.zeros(N)
acceleration_y = np.zeros(N)
rocket_altitude = np.zeros(N)
rocket_x_coordinate = np.zeros(N)
measured_altitude = np.zeros(N)
measured_dir = np.zeros(N)
rotation_torque_from_air = np.zeros(N)
air_resistance_body_x = np.zeros(N)
air_resistance_body_y = np.zeros(N)
air_resistance_body = np.zeros(N)
air_resistance_top_flaps_x = np.zeros(N)
air_resistance_top_flaps_y = np.zeros(N)
air_resistance_top_flaps = np.zeros(N)
angle_top_flaps = np.zeros(N)
air_resistance_bottom_flaps_x = np.zeros(N)
air_resistance_bottom_flaps_y = np.zeros(N)
air_resistance_bottom_flaps = np.zeros(N)
angle_bottom_flaps = np.zeros(N)

#Motor

motor_resistance = 0.5 #[ohm]
motor_inductance = 1.5*10**(-3) #[H]
U_rated = 15 #[V]
I_rated = 15 #[A]
Ke = 0.0020 #[Vs]
Kt = Ke #[Vs]
efficiency = 0.85
motor_Kv = 2100

armature_current = np.zeros(N)
armature_voltage = np.zeros(N)
T_dev = np.zeros(N)
energy_consumption = np.zeros(N)
```

```
#Propeller

propeller_mass = 0.017 #[kg]
prop_diam = 0.18 #[m]
prop_pitch = 0.17 #[m]

rpm = np.zeros(N)

#Servos

servo_turn_speed = 10*math.pi/3 #[rad/s]

#Sensors

height_sensor_speed = 1000 #[Hz]
height_sensor_measurement_error = 0.1 #[mm]
height_sensor_constant_error = 10 #[mm]

gyro_sensor_speed = 1000 #[Hz]
gyro_sensor_measurement_error = 0.01 #[deg]
gyro_sensor_constant_error = 0.1 #[deg]

#Other parameters

g = 9.81 #[m/s2]
air_density = 1.225 #[kg/m3]
air_prop_damping_constant = 0.0005
ground_hit_speed = []
ratio = 2                                                                    #higher
    ratio gives more emphasis on gyro stabilization in the cascade controll

control_signal_U = np.zeros(N)
control_signal_A = np.zeros(N)
control_signal_V = np.zeros(N)
control_signal_top_flaps = np.zeros(N)
control_signal_bottom_flaps = np.zeros(N)
goal_altitudes = np.zeros(N)
goal_dirs = np.zeros(N)
goal_x_coords = np.zeros(N)
has_crashed = np.zeros(N)
modes = []


#Simulation-----------------------------------------------------------

def height_sensor(real_height, noise):
    if noise:
        measured_height = real_height + random.uniform(-
    height_sensor_measurement_error/1000, height_sensor_measurement_error/1000)
        measured_height += height_sensor_constant_error/1000  #constant error
    else:
        measured_height = real_height

    return measured_height

def gyro(angle, noise):
    if noise:
        measured_angle = angle + random.uniform(-gyro_sensor_measurement_error*math.pi
    /180, gyro_sensor_measurement_error*math.pi/180)
        measured_angle += gyro_sensor_constant_error*math.pi/180
    else:
        measured_angle = angle

    return measured_angle
```

```
def thrust_components(rocket_direction, thrust_direction, thrust):
    relative_thrust_dir = - rocket_direction + thrust_direction
    thrust_tangential = thrust*math.sin(relative_thrust_dir)

    thrust_x = thrust*math.sin(thrust_direction)
    thrust_y = thrust*math.cos(thrust_direction)

    return thrust_x, thrust_y, thrust_tangential

def turn_rudders(current_dir, ref_angle):
    if (ref_angle == current_dir) or (ref_angle < (current_dir - servo_turn_speed*dt))
     or (ref_angle > (current_dir + servo_turn_speed*dt)):
        return ref_angle
    elif ref_angle < current_dir:
       return current_dir - servo_turn_speed*dt
    else:
        return current_dir + servo_turn_speed*dt


def update(i, reference_V, reference_angle, noise, crash, land, top_flaps_angle_ref,
    bottom_flaps_angle_ref):
    armature_voltage[i] = reference_V

    if i > 0:
        armature_current[i]  = ((armature_voltage[i] - Ke*rpm[i]*2*math.pi/60)*dt +
    motor_inductance*armature_current[i-1])/(motor_resistance*dt + motor_inductance)
      #2

    T_dev[i] = Ke*armature_current[i]

    air_resistance_body_y[i] = -np.sign(accent_velocity[i])*rocket_body_drag_c*
    rocket_belly_area*abs(math.sin(rocket_direction[i]))*(air_density*accent_velocity[i
    ]**2)/2
    air_resistance_body_x[i] = -np.sign(side_velocity[i])*rocket_body_drag_c*
    rocket_belly_area*abs(math.cos(rocket_direction[i]))*(air_density*side_velocity[i
    ]**2)/2
    air_resistance_body[i] = math.sqrt(air_resistance_body_x[i]**2 +
    air_resistance_body_y[i]**2)

    air_resistance_top_flaps_y[i] = -np.sign(accent_velocity[i])*2*rocket_flap_drag_c*
    top_flap_area*abs(math.sin(angle_top_flaps[i]))*abs(math.sin(rocket_direction[i]))
    *(air_density*accent_velocity[i]**2)/2
    air_resistance_top_flaps_x[i] = -np.sign(side_velocity[i])*2*rocket_flap_drag_c*
    top_flap_area*abs(math.sin(angle_top_flaps[i]))*abs(math.cos(rocket_direction[i]))
    *(air_density*side_velocity[i]**2)/2
    air_resistance_top_flaps[i] = math.sqrt(air_resistance_top_flaps_x[i]**2 +
    air_resistance_top_flaps_y[i]**2)

    air_resistance_bottom_flaps_y[i] = -np.sign(accent_velocity[i])*2*
    rocket_flap_drag_c*bottom_flap_area*abs(math.sin(angle_bottom_flaps[i]))*abs(math.
    sin(rocket_direction[i]))*(air_density*accent_velocity[i]**2)/2
    air_resistance_bottom_flaps_x[i] = -np.sign(side_velocity[i])*2*rocket_flap_drag_c
    *bottom_flap_area*abs(math.sin(angle_bottom_flaps[i]))*abs(math.cos(
    rocket_direction[i]))*(air_density*side_velocity[i]**2)/2
    air_resistance_bottom_flaps[i] = math.sqrt(air_resistance_bottom_flaps_x[i]**2 +
    air_resistance_bottom_flaps_y[i]**2)

    rotation_torque_from_air[i] = np.sign(rocket_direction[i])*(air_resistance_body[i
    ]*(center_of_mass- rocket_body_center_of_preassure) + air_resistance_bottom_flaps[i
    ]*(center_of_mass-bottom_flap_center_of_preassure) - air_resistance_top_flaps[i]*(
    top_flap_center_of_preassure-center_of_mass))

    #formula from #https://www.flitetest.com/articles/propeller-static-dynamic-thrust-
    calculation:
    F_thrust[i] = 4.392399*10**(-8)*rpm[i]*(39.3700787*prop_diam)**3.5/math.sqrt
```

```
    (39.3700787*prop_pitch)*(4.23333*10**(-4)*rpm[i]*prop_pitch*39.3700787-
    accent_velocity[i]*math.cos(rocket_direction[i])- side_velocity[i]*math.sin(
    rocket_direction[i]))
    F_thrust_x[i], F_thrust_y[i], F_thrust_tangential[i] = thrust_components(
    rocket_direction[i], thrust_direction[i], F_thrust[i])
    acceleration_y[i] = (F_thrust_y[i]+air_resistance_body_y[i]-m*g)/m      #newtons 2
    nd law
    acceleration_x[i] = (F_thrust_x[i]+air_resistance_body_x[i])/m
    rotation_torque[i] = F_thrust_tangential[i]*center_of_mass -
    rotation_torque_from_air[i]
    rocket_angular_acc[i] = rotation_torque[i]/moment_of_inertia_rocket


    if i<N-1:
        if rocket_altitude[i]<0 and accent_velocity[i]>-1 and not crash and land:
            #Rocket landed
            rocket_x_coordinate[i+1] = rocket_x_coordinate[i]

        elif rocket_altitude[i]<0 and accent_velocity[i]<-1 or crash:
            #Rocket crashed
            crash = True
            has_crashed[i] = 1
            rocket_x_coordinate[i+1] = rocket_x_coordinate[i]
            rocket_direction[i+1] = rocket_direction[i]

        else:
            #Rocket flying
            thrust_direction[i+1] = turn_rudders(current_dir = thrust_direction[i],
    ref_angle = reference_angle)
            accent_velocity[i+1] = accent_velocity[i] + dt*acceleration_y[i]      #
    newtons step method
            rocket_altitude[i+1] = rocket_altitude[i] + dt*accent_velocity[i]
            side_velocity[i+1] = side_velocity[i] + dt*acceleration_x[i]
            rocket_x_coordinate[i+1] = rocket_x_coordinate[i] + dt*side_velocity[i]
            rocket_angular_vel[i+1] = rocket_angular_vel[i] + rocket_angular_acc[i]*dt
            rocket_direction[i+1] = rocket_direction[i] + rocket_angular_vel[i]*dt
            rpm[i+1] = air_prop_damping_constant*armature_voltage[i]*motor_Kv + (1-
    air_prop_damping_constant)*rpm[i]      #adding som inertia to the system, fluid
    mechanics of propeller too advanced so this is just to get similar behavior of real
     system
            energy_consumption[i+1] = energy_consumption[i] + dt*(armature_current[i]*
    armature_voltage[i])/efficiency

            angle_top_flaps[i+1] = turn_rudders(current_dir = angle_top_flaps[i],
    ref_angle = top_flaps_angle_ref)
            angle_bottom_flaps[i+1] = turn_rudders(current_dir = angle_bottom_flaps[i
    ], ref_angle = bottom_flaps_angle_ref)


    measured_altitude[i] = height_sensor(rocket_altitude[i], noise)
    measured_dir[i] = gyro(rocket_direction[i], noise)

    return measured_altitude[i], rocket_x_coordinate[i], measured_dir[i], crash


def run_sim(altitude_path, x_path, dir_path, altitude_PID_vals, x_PID_vals_thrust,
    yaw_PID_vals_thrust, yaw_PID_vals_fall, mode, noise):
    #Initialize controllers
    speed_controller = PID(Kp = altitude_PID_vals['Kp'], Ki= altitude_PID_vals['Ki'],
    Kd = altitude_PID_vals['Kd'], ref = altitude_path[0])
    speed_controller.set_limits(0,U_rated)       #the PID should not overshoot the
    rated voltage and should only be positive

    thrust_vector_controller_x = PID(Kp = x_PID_vals_thrust['Kp'], Ki=
    x_PID_vals_thrust['Ki'], Kd = x_PID_vals_thrust['Kd'], ref = x_path[0])
```

```
thrust_vector_controller_x.set_limits(-math.pi/4, math.pi/4)

thrust_vector_controller_a = PID(Kp = yaw_PID_vals_thrust['Kp'], Ki=
yaw_PID_vals_thrust['Ki'], Kd = yaw_PID_vals_thrust['Kd'], ref = dir_path[0])
thrust_vector_controller_a.set_limits(-math.pi/4, math.pi/4)

top_flap_controller = PID(Kp = yaw_PID_vals_fall['Kp'], Ki= yaw_PID_vals_fall['Ki'
], Kd = yaw_PID_vals_fall['Kd'], ref = dir_path[0])
top_flap_controller.set_limits(0, math.pi/2)

bottom_flap_controller = PID(Kp = yaw_PID_vals_fall['Kp'], Ki= yaw_PID_vals_fall['
Ki'], Kd = yaw_PID_vals_fall['Kd'], ref = dir_path[0])
bottom_flap_controller.set_limits(0, math.pi/2)

reference_voltage = 0
reference_angle = 0             #yaw x coordinate thrust
reference_angle_2 = 0           #yaw stability thrust
reference_angle_top_flaps = 0
reference_angle_bottom_flaps = 0
altitude = 0                    #starts from the ground
x_coord = 0
direction = 0

#Run simulation

crash_occured = False
land = False

for i in range(N):
    #Print progress
    if (i)%(N//10) == 0 or i==(N-1):
        print('Loading...', round(100*i/N), '%')
    #Jump over to next index of reference signals in the given flight path
    if i%(int(N/len(mode))) == 0:
        if round(len(mode)*i/N) < len(mode):
            index = round(len(mode)*i/N)

    current_mode = mode[index]
    modes.append(current_mode)

    if index == len(mode)-1:
        land = True

    if current_mode == 'thrust':
        reference_angle_top_flaps = math.pi/2
        reference_angle_bottom_flaps = math.pi/2

        current_goal_altitude = altitude_path[index]
        current_goal_x = x_path[index]
        current_goal_dir = dir_path[index]
        goal_altitudes[i] = current_goal_altitude
        goal_x_coords[i] = current_goal_x
        goal_dirs[i] = current_goal_dir

        speed_controller.change_ref(current_goal_altitude)
        thrust_vector_controller_x.change_ref(current_goal_x)
        thrust_vector_controller_a.change_ref(current_goal_dir)

        #Only update at new measurement
        if i%(int(1/(dt*height_sensor_speed))) == 0:
            reference_voltage = speed_controller(altitude, dt)

        control_signal_U[i] = reference_voltage

        if i%(int(1/(dt*gyro_sensor_speed))) == 0:
```

```
                reference_angle = thrust_vector_controller_x(x_coord, dt)
                reference_angle_2 = thrust_vector_controller_a(direction, dt)

            control_signal_V[i] = reference_angle
            control_signal_A[i] = reference_angle_2

            #cascade control
            if i%ratio == 0 :
                altitude, x_coord, direction, crash_occured  = update(i, reference_V =
     reference_voltage, reference_angle = reference_angle, noise = noise, crash=
    crash_occured, land= land, bottom_flaps_angle_ref= reference_angle_bottom_flaps,
    top_flaps_angle_ref= reference_angle_top_flaps)              #x-coordinate
            else:
                altitude, x_coord, direction, crash_occured  = update(i, reference_V =
     reference_voltage, reference_angle = reference_angle_2, noise = noise, crash=
    crash_occured, land= land, bottom_flaps_angle_ref= reference_angle_bottom_flaps,
    top_flaps_angle_ref= reference_angle_top_flaps)             #stability

        elif current_mode == 'fall':
            reference_voltage = 0
            reference_angle = 0

            current_goal_altitude = altitude_path[index]
            current_goal_x = x_path[index]
            current_goal_dir = dir_path[index]
            goal_altitudes[i] = current_goal_altitude
            goal_x_coords[i] = current_goal_x
            goal_dirs[i] = current_goal_dir

            top_flap_controller.change_ref(current_goal_dir)
            bottom_flap_controller.change_ref(current_goal_dir)

            if i%(int(1/(dt*gyro_sensor_speed))) == 0:
                reference_angle_top_flaps = top_flap_controller(direction, dt)
                reference_angle_bottom_flaps = top_flap_controller(direction, dt)

            control_signal_top_flaps[i] = reference_angle_top_flaps
            control_signal_bottom_flaps[i] = reference_angle_bottom_flaps

            altitude, x_coord, direction, crash_occured  = update(i, reference_V =
    reference_voltage, reference_angle = reference_angle, noise = noise, crash=
    crash_occured, land= land, bottom_flaps_angle_ref= reference_angle_bottom_flaps,
    top_flaps_angle_ref= reference_angle_top_flaps)

            if altitude <= current_goal_altitude:
                index += 1

        if i > 0 and rocket_altitude[i] <= 0 and rocket_altitude[i-1] > 0:
            ground_hit_speed.append(accent_velocity[i])


def display_sim(simulation_speed, rocket_scale):
    scale_factor = 20/max(max(goal_altitudes), 2*max(abs(goal_x_coords)))  #to make
    the rocket move higher/lower in the simulation display
    sim = Sim(scale_fac = scale_factor, rocket_c_of_preassure=
    rocket_body_center_of_preassure, top_flaps_c_of_preassure=
    top_flap_center_of_preassure, bottom_flaps_c_of_preassure=
    bottom_flap_center_of_preassure ,rocket_w= rocket_radius*2, rocket_h= rocket_height
    , rocket_scale = rocket_scale)
    pygame.init()
    pygame.display.set_caption("Flight simulation")
    speed_factor = simulation_speed  #changes playback-speed of simulation

    for i in range(0,N,speed_factor):
        sim.screen.fill('white')
```

```
        sim.run(x = sim.gridSize/2 + rocket_x_coordinate[i:min(i+simulation_speed,N)]*
    scale_factor, y = sim.start_height_factor*sim.gridSize - rocket_altitude[i:min(i+
    simulation_speed,N)]*scale_factor, goal_x= goal_x_coords[i]*scale_factor,
                goal_y = sim.start_height_factor*sim.gridSize - goal_altitudes[i]*
    scale_factor, thrust_x = F_thrust_x[i:min(i+simulation_speed,N)], thrust_y=
    F_thrust_y[i:min(i+simulation_speed,N)], direction= rocket_direction[i:min(i+
    simulation_speed,N)],
                mode= modes[i], air_res_body_x= air_resistance_body_x[i:min(i+
    simulation_speed,N)], air_res_body_y= air_resistance_body_y[i:min(i+
    simulation_speed,N)],air_res_top_flaps_x= air_resistance_top_flaps_x[i:min(i+
    simulation_speed,N)], air_res_top_flaps_y= air_resistance_top_flaps_y[i:min(i+
    simulation_speed,N)],
                air_res_bottom_flaps_x= air_resistance_bottom_flaps_x[i:min(i+
    simulation_speed,N)],air_res_bottom_flaps_y= air_resistance_bottom_flaps_y[i:min(i+
    simulation_speed,N)] , time= time[i:min(i+simulation_speed,N)], has_crashed=
    has_crashed[i], acc_vel = accent_velocity[i], altitude = rocket_altitude[i])
        pygame.display.update()

        if not sim.running:
            pygame.display.update()
            pygame.time.delay(500)
            break

    if ground_hit_speed:
        print('Rocket hit ground at speed ', str(abs(round(ground_hit_speed[0], 4))),
    '[m/s]')

def plot_results():
    fig,axs = plt.subplots(4, figsize=(15,15))
    plt.rcParams['lines.linewidth'] = 1
    fig.tight_layout(pad = 5)
    axs[0].plot(time, rocket_altitude, label = 'Altitude')
    axs[0].plot(time, measured_altitude, label = 'Measured altitude')
    axs[0].plot(time, goal_altitudes, label = 'Goal altitude')
    axs[0].set_ylim(0)
    axs[0].legend(loc='upper left')
    axs[0].set_title('Altitude [m]')
    axs[1].plot(time, rocket_x_coordinate, label = 'x coordinate')
    axs[1].plot(time, goal_x_coords, label = 'Goal x coordinate')
    axs[1].set_title('X coordinate [m]')
    axs[1].legend(loc='upper left')
    axs[2].plot(time, rocket_direction, label = 'Direction')
    axs[2].plot(time, measured_dir, label = 'Measured direction')
    axs[2].set_title('Direction [rad]')
    axs[2].legend(loc='upper left')
    axs[3].plot(time, accent_velocity)
    axs[3].set_title('Accent velocity [m/s]')

    plt.show()
```

Listing 2: Simuleringskod

```
import pygame
import math
from statistics import mean


class Sim:
    def __init__(self, rocket_h, rocket_w, rocket_c_of_preassure,
    top_flaps_c_of_preassure, bottom_flaps_c_of_preassure ,scale_fac, rocket_scale):
        self.scale_fac = scale_fac
        self.screenSize = (self.w, self.h) =  (750, 750)
        self.screen = pygame.display.set_mode(self.screenSize)
```

```
        self.gridSize = 50
        self.running = True
        self.rocket_c = (0, 0)
        self.rocket_h = rocket_scale*15*rocket_h*self.scale_fac
        self.rocket_w = rocket_scale*24*rocket_w*self.scale_fac
        self.body_c_of_preassure = rocket_scale*15*rocket_c_of_preassure*self.
    scale_fac
        self.top_flaps_c_of_preassure = rocket_scale*30*top_flaps_c_of_preassure*self.
    scale_fac
        self.bottom_flaps_c_of_preassure = rocket_scale*15*bottom_flaps_c_of_preassure
    *self.scale_fac
        self.start_height_factor = 4/5
        self.rocket_image = pygame.transform.scale(pygame.image.load('simulering/
    starship_model_svart_small.png'), (self.rocket_w,self.rocket_h))
        self.explosion_image = pygame.transform.scale(pygame.image.load('simulering/
    explosion.png'), (self.rocket_w,self.rocket_h))
        self.crash = False
        self.crash_location = (0,0)

    def run(self, x, y, goal_x, goal_y, thrust_x, thrust_y, direction,mode,
    air_res_body_x, air_res_body_y, air_res_top_flaps_x, air_res_top_flaps_y,
    air_res_bottom_flaps_x, air_res_bottom_flaps_y,time, has_crashed, acc_vel, altitude
    ):
        direction = mean(direction)
        x = mean(x)
        y = mean(y)
        thrust_x = mean(thrust_x)
        thrust_y = mean(thrust_y)
        air_res_body_x = mean(air_res_body_x)
        air_res_body_y = mean(air_res_body_y)
        air_res_top_flaps_x = mean(air_res_top_flaps_x)
        air_res_top_flaps_y = mean(air_res_top_flaps_y)
        air_res_bottom_flaps_x = mean(air_res_bottom_flaps_x)
        air_res_bottom_flaps_y = mean(air_res_bottom_flaps_y)
        time = mean(time)

        rocket_y_offset = self.rocket_h/(self.h/self.gridSize)
        rocket_x_offset = self.rocket_w/(self.w/self.gridSize)
        self.rocket_c = [x-rocket_x_offset/2,y-rocket_y_offset]
        if self.running:
            for event in pygame.event.get():
                self.get_key(event)        #returns false if quit is pressed,
    otherwise check for keypresses
            self.screen.fill((0,0,0))
            self.draw_rocket(direction, has_crashed)
            self.draw_thrust(thrust_x = thrust_x, thrust_y = thrust_y, direction=
    direction)
            if mode == 'fall':
                self.draw_air_res(res_x= air_res_body_x, res_y= air_res_body_y,
    direction= direction, c_of_preassure= self.body_c_of_preassure)
            pygame.draw.line(self.screen, (255,255,255), (0, self.start_height_factor*
    self.h), (self.w, self.start_height_factor*self.h))  #ground
            pygame.draw.line(self.screen, (0,255,0), ((25+goal_x)*self.w/self.gridSize
    , 0), ((25+goal_x)*self.w/self.gridSize, self.h))
            pygame.draw.line(self.screen, (0,255,0), (0, goal_y*self.h/self.gridSize),
     (self.w, goal_y*self.h/self.gridSize))
            self.write_stats(time,altitude, acc_vel)
            self.collision(x,y)

            pygame.display.update()
            pygame.time.delay(1)

    def write_stats(self, time, y, acc_vel):
        font = pygame.font.SysFont('didot.ttc', 15)
        time_text = font.render('Time:'+str(round(time,2))+'s', True, (255, 255, 255))
```

```
        altitude_text = font.render('Altitude:'+str(round(max(y,0),2))+'m', True,
    (255, 255, 255))
        acc_vel_text = font.render('Accent velocity:'+str(round(acc_vel,2))+'m/s',
    True, (255, 255, 255))
        self.screen.blit(time_text, (5, 5))
        self.screen.blit(altitude_text, (5, 15))
        self.screen.blit(acc_vel_text, (5, 25))

    def draw_rocket(self,direction, crash):
        rocket_center_size = self.w / self.gridSize
        coordinates = self.rocket_c
        (x,y) = (coordinates[0], coordinates[1])
        width = rocket_center_size
        if crash:
            rocket_image = self.explosion_image
            self.screen.blit(rocket_image, (x *width, y * width, width, width))
        else:
            rocket_image = pygame.transform.rotate(self.rocket_image, -180*direction/
    math.pi)
            self.screen.blit(rocket_image, (x * width, y * width, width, width))

    def draw_thrust(self, thrust_x, thrust_y, direction):
        thrust_len_factor = 0.05*self.scale_fac
        rocket_thrust_offset_y = self.rocket_h/30 + self.rocket_h/30*math.cos(
    direction)        #because the rocket images center is in the upper left corner
        if direction < 0 : rocket_thrust_offset_x = self.rocket_w/30 - (self.rocket_h
    - self.rocket_w)/15*math.sin(direction)
        else: rocket_thrust_offset_x = self.rocket_w/30 - self.rocket_w/30*math.sin(
    direction)
        (x, y) = self.rocket_c
        pygame.draw.line(self.screen, (255,255,0), ((x+rocket_thrust_offset_x)*self.w/
    self.gridSize, (y+rocket_thrust_offset_y)*self.h/self.gridSize),
                    ((x+rocket_thrust_offset_x-thrust_len_factor*thrust_x)*self.h/self
    .gridSize, (y+rocket_thrust_offset_y+thrust_len_factor*thrust_y)*self.h/self.
    gridSize))

    def draw_air_res(self, res_x, res_y, direction, c_of_preassure):
        res_len_factor = 1.00*self.scale_fac
        rocket_offset_y = self.rocket_h/30 + self.rocket_h/30*math.cos(direction)
       #because the rocket images center is in the upper left corner
        if direction < 0 : rocket_offset_x = self.rocket_w/30 - (self.rocket_h - self.
    rocket_w)/15*math.sin(direction)
        else: rocket_offset_x = self.rocket_w/30 - self.rocket_w/30*math.sin(direction
    )

        (x, y) = self.rocket_c

        pygame.draw.line(self.screen, (0,0,255), ((x+rocket_offset_x)*self.w/self.
    gridSize, (y+rocket_offset_y)*(self.h-c_of_preassure)/self.gridSize),
                            ((x+rocket_offset_x+res_len_factor*res_x)*self.h/self.
    gridSize, (y+rocket_offset_y-res_len_factor*res_y)*(self.h-c_of_preassure)/self.
    gridSize))

    def collision(self,x, y):
        center = self.rocket_c
        if (center[0] > self.gridSize - 1 or center[0] < 0) or (center[1] < 0 or
    center[1] > self.start_height_factor*self.gridSize):
            self.running = False
        if y < 0:
            self.crash = True
            self.crash_location = (x,y)

    def get_key(self, event):
        if event.type == pygame.QUIT:
            self.running = False
```

Listing 3: Visualisering av simulering

```
from flightsim_test import run_sim, display_sim, plot_results
import math

def main():
    mode = ['thrust', 'thrust', 'fall', 'thrust']
    altitude_path = [10,20,15,0]      #[m] altitudes
    x_path = [3,-3,0,0]               #[m] parallel to ground
    direction_path = [0, 0,-math.pi/2,0]        #[rad]

    altitude_controller = {'Kp': 10, 'Ki': 500, 'Kd': 5}
    x_axis_controller_thrust = {'Kp': 2.5, 'Ki': 10, 'Kd': 0.16}
    yaw_controller_thrust = {'Kp': 0.7, 'Ki': 5, 'Kd': 0.05}
    yaw_controller_fall = {'Kp': 0.7, 'Ki': 5, 'Kd': 0.05}


    rocket_scale = 3                                                    #
    displaying rocket in a size rocket_scale:1 to make visibility better at high
    altitude flights
    simulation_speed = 300                                              #
    changes display-speed of simulation, 300 real time
    noise = False                                                       #adds
    noise to measured signals

    run_sim(altitude_path= altitude_path, x_path= x_path, dir_path= direction_path,
            altitude_PID_vals= altitude_controller, x_PID_vals_thrust=
    x_axis_controller_thrust, yaw_PID_vals_thrust= yaw_controller_thrust,
            yaw_PID_vals_fall= yaw_controller_fall, mode = mode, noise = noise)
    display_sim(simulation_speed, rocket_scale)
    plot_results()


if __name__ == "__main__":
    main()
```

Listing 4: Kod för att köra simuleringen

A.5 MATLAB-kod för att finna PID-konstanter

```
Kp = 1;
Ki = 1;
Kd = 1;

s = tf('s');
C = Kp + Ki/s + Kd*s
P=0.45*30/(0.304*s^2)
pidTuner(P,C)
```

Listing 5: Kod för att ta fram PID-värden

A.6 Kod i C++ för raketflygning

```
#ifndef PID_CONTROLLERS_H
#define PID_CONTROLLERS_H
#include <Arduino.h>

double thrust_vector_pid_pitch(double dir, double reference_dir);
double thrust_vector_pid_roll(double dir, double reference_dir);
double thrust_vector_pid_x(double acc, double reference_acc);
double thrust_vector_pid_y(double acc, double reference_acc);
double rotation_pid(double dir, double reference_dir);

#endif
```

Listing 6: PID regulatorer för servostyrning (header-fil)

```
#include "pid_controllers.h"

//PID error variables

double tv_cumulative_error_pitch;
double tv_previous_error_pitch;

double tv_cumulative_error_roll;
double tv_previous_error_roll;

double tv_cumulative_error_x;
double tv_previous_error_x;

double tv_cumulative_error_y;
double tv_previous_error_y;

double rot_cumulative_error;
double rot_previous_error;

//PID limits

int max_servo_angle_rot = 10;
int min_servo_angle_rot = -10;

int max_servo_angle_tv_stab = 40;
int min_servo_angle_tv_stab = -40;

int max_servo_angle_tv_pos = 10;
int min_servo_angle_tv_pos = -10;

//PID constants
```

```
double tv_stability_Kp = 3.000;
double tv_stability_Ki = 0.080;
double tv_stability_Kd = 55.00;

double tv_position_Kp = 3.000;
double tv_position_Ki = 0.040;
double tv_position_Kd = 20.00;

double rot_Kp = 0.3;
double rot_Ki = 0.000;
double rot_Kd = 0.000;


//PID controllers

double thrust_vector_pid_pitch(double dir, double reference_dir){
  double tv_dir_error = reference_dir - dir;
  double p = tv_stability_Kp * tv_dir_error;
  double i = tv_stability_Ki * tv_cumulative_error_pitch;
  double d = tv_stability_Kd * (tv_dir_error-tv_previous_error_pitch);

  float pid_val = p+i+d;

  if (!((pid_val<=min_servo_angle_tv_stab && tv_dir_error<=0)|| (pid_val>=
    max_servo_angle_tv_stab && tv_dir_error>=0))){
    tv_cumulative_error_pitch += tv_dir_error;
  }

  tv_previous_error_pitch = tv_dir_error;

  if(pid_val>=max_servo_angle_tv_stab){
    return max_servo_angle_tv_stab;
  }
  else if(pid_val<=min_servo_angle_tv_stab){
    return min_servo_angle_tv_stab;
  }
  else{
    return pid_val;
  }
}

double thrust_vector_pid_roll(double dir, double reference_dir){
  double tv_dir_error = reference_dir - dir;
  double p = tv_stability_Kp * tv_dir_error;
  double i = tv_stability_Ki * tv_cumulative_error_roll;
  double d = tv_stability_Kd * (tv_dir_error-tv_previous_error_roll);

  float pid_val = p+i+d;

  if (!((pid_val<=min_servo_angle_tv_stab && tv_dir_error<=0)|| (pid_val>=
    max_servo_angle_tv_stab && tv_dir_error>=0))){
    tv_cumulative_error_roll += tv_dir_error;
  }
  tv_previous_error_roll = tv_dir_error;

  if(pid_val>=max_servo_angle_tv_stab){
    return max_servo_angle_tv_stab;
  }
  else if(pid_val<=min_servo_angle_tv_stab){
    return min_servo_angle_tv_stab;
  }
  else{
    return pid_val;
  }
}
```

```
double thrust_vector_pid_x(double acc, double reference_acc){
  double tv_acc_error = reference_acc - acc;
  double p = tv_position_Kp * tv_acc_error;
  double i = tv_position_Ki * tv_cumulative_error_x;
  double d = tv_position_Kd * (tv_acc_error-tv_previous_error_x);

  tv_cumulative_error_x += tv_acc_error;
  tv_previous_error_x = tv_acc_error;

  float pid_val = p+i+d;

  if(pid_val>=max_servo_angle_tv_pos){
    return max_servo_angle_tv_pos;
  }
  else if(pid_val<=min_servo_angle_tv_pos){
    return min_servo_angle_tv_pos;
  }
  else{
    return pid_val;
  }
}

double thrust_vector_pid_y(double acc, double reference_acc){
  double tv_acc_error = reference_acc - acc;
  double p = tv_position_Kp * tv_acc_error;
  double i = tv_position_Ki * tv_cumulative_error_y;
  double d = tv_position_Kd * (tv_acc_error-tv_previous_error_y);

  tv_cumulative_error_y += tv_acc_error;
  tv_previous_error_y = tv_acc_error;

  float pid_val = p+i+d;

  if(pid_val>=max_servo_angle_tv_pos){
    return max_servo_angle_tv_pos;
  }
  else if(pid_val<=min_servo_angle_tv_pos){
    return min_servo_angle_tv_pos;
  }
  else{
    return pid_val;
  }
}

double rotation_pid(double dir, double reference_dir){
  double rot_error = reference_dir - dir;
  double p = rot_Kp * rot_error;
  double i = rot_Ki * rot_cumulative_error;
  double d = rot_Kd * (rot_error-rot_previous_error);

  rot_cumulative_error += rot_error;
  rot_previous_error = rot_error;

  float pid_val = p+i+d;

  if(pid_val>=max_servo_angle_rot){
    return max_servo_angle_rot;
  }
  else if(pid_val<=min_servo_angle_rot){
    return min_servo_angle_rot;
  }
  else{
    return pid_val;
  }
```

```
}
```

Listing 7: PID regulatorer för servostyrning

```
#include "pid_controllers.h"
#include <Servo.h>
#include "ICM20600.h"
#include <Wire.h>
#include "AK09918.h"
#include <Dps310.h>
#include "MadgwickAHRS.h"
#include <SimpleFOC.h>

//Initialize components

Servo servo_tvZ1;
Servo servo_tvZ2;
Servo servo_tvY1;
Servo servo_tvY2;
ICM20600 icm20600(true);
AK09918_err_type_t err;
AK09918 ak09918;
Dps310 Dps310PressureSensor = Dps310();
Madgwick filter;
LowPassFilter filter_LP_roll = LowPassFilter(0.25);
LowPassFilter filter_LP_pitch = LowPassFilter(0.25);

//Global variables
int32_t magn_x, magn_y, magn_z;
int32_t offset_x, offset_y, offset_z;
double declination_shenzhen = 4.78;      //From http://www.magnetic-declination.com/

int servo_tvZ1_pin = 6;
int servo_tvZ2_pin = 11;
int servo_tvY1_pin = 9;
int servo_tvY2_pin = 5;

const float sensorRate =  25.00; //[Hz]
const float ALPHA = 0.35;
float roll_filtered, pitch_filtered, yaw_filtered;
float roll_filtered_, pitch_filtered_, yaw_filtered_;

int flight_time = 30;
unsigned long start_time, current_time;

//Calibrate/set up sensors and components

void setup() {
  start_time = millis();
  Wire.begin();
  filter.begin(sensorRate);

  Serial.begin(9600);

  err = ak09918.initialize();
  ak09918.switchMode(AK09918_POWER_DOWN);
  ak09918.switchMode(AK09918_CONTINUOUS_100HZ);
  icm20600.initialize();

  err = ak09918.isDataReady();                //code from library examples
  while (err != AK09918_ERR_OK) {
      Serial.println("Waiting Sensor");
      delay(100);
      err = ak09918.isDataReady();
```

```
  }

  calibrate_magn(10000, &offset_x, &offset_y, &offset_z);         //calibrate magnetic
     sensor, takes 10 seconds

  servo_tvZ1.attach(servo_tvZ1_pin);
  servo_tvZ2.attach(servo_tvZ2_pin);
  servo_tvY1.attach(servo_tvY1_pin);
  servo_tvY2.attach(servo_tvY2_pin);

  turn_servo(0, servo_tvZ1);                        //Put rudders in neutral position
  turn_servo(0, servo_tvZ2);
  turn_servo(0, servo_tvY1);
  turn_servo(0, servo_tvY2);

  current_time = (millis()-start_time)/1000;
  while (current_time<30){
     current_time = (millis()-start_time)/1000;         //wait until 30 seconds have
    passed from startup to sync with other arduino
  }

  start_time = millis();
}

//Run control program

void loop() {
  current_time = (millis()-start_time)/1000;

  float xAcc, yAcc, zAcc;
  float xGyro, yGyro, zGyro;

  float roll, pitch, yaw;

  xAcc = icm20600.getAccelerationX();
  yAcc = icm20600.getAccelerationY();
  zAcc = icm20600.getAccelerationZ();

  xGyro = icm20600.getGyroscopeX();
  yGyro = icm20600.getGyroscopeY();
  zGyro = icm20600.getGyroscopeZ();

  ak09918.getData(&magn_x, &magn_y, &magn_z);
  magn_x = magn_x - offset_x;
  magn_y = magn_y - offset_y;
  magn_z = magn_z - offset_z;


  filter.updateIMU(xGyro, yGyro, zGyro, xAcc, yAcc, zAcc);
  //filter.update(xGyro, yGyro, zGyro, xAcc, yAcc, zAcc, magn_x, magn_y, magn_z);

  roll = filter.getRoll();            //x-axis on IMU
  pitch = filter.getPitch();          //y-axis on IMU
  yaw = filter.getYaw()-180.0;        //z-axis on IMU

  //LP_filter(roll,pitch,yaw);
  roll_filtered = filter_LP_roll(roll);
  pitch_filtered = filter_LP_pitch(pitch);

  float ang_rot = rotation_pid(-icm20600.getGyroscopeZ(), 0);
  float ang_y = thrust_vector_pid_pitch(pitch_filtered, 0);
  float ang_x = thrust_vector_pid_roll(roll_filtered, 0);
  float ang_y_2 = thrust_vector_pid_x(xAcc, 0);
  float ang_x_2 = thrust_vector_pid_y(yAcc, 0);
```

```
  int mode = 5; // all:1, rotation:2, position:3, stability:4, stability and rotation
    :5
  control(ang_x, ang_y, ang_x_2, ang_y_2, ang_rot, mode);


  //Serial.print(",");
  //Serial.print("servoR:");
  //Serial.print(ang_x);
  //Serial.print(",");
  //Serial.print("servoP:");
  //Serial.print(ang_y);
  //Serial.print(",");
  //Serial.print("Yaw_MF:");
  //Serial.print(yaw_filtered);
  Serial.print(",");
  Serial.print("Roll_MLF:");
  Serial.print(roll_filtered);
  //Serial.print(",");
  //Serial.print("Roll_MLF_2:");
  //Serial.print(roll_filtered_);
  //Serial.print(",");
  //Serial.print("Roll_MF:");
  //Serial.print(roll);
  Serial.print(",");
  Serial.print("Pitch_MF:");
  Serial.print(pitch_filtered);
  Serial.print(",");
  Serial.print("goal:");
  Serial.println(0);

  delay(1);

  if (current_time>6+5){
    exit(0);
  }
}

//Functions

void turn_servo(int angle, Servo servo){
  servo.write(90+angle);
}

void control(float ang_x, float ang_y, float ang_x_2, float ang_y_2, float ang_rot,
    int mode){
  switch(mode){
    case 1:
      turn_servo(-ang_y-ang_y_2-ang_rot, servo_tvY1);
      turn_servo(ang_y+ang_y_2-ang_rot, servo_tvY2);
      turn_servo(-ang_x-ang_x_2-ang_rot, servo_tvZ1);
      turn_servo(ang_x+ang_x_2-ang_rot, servo_tvZ2);
      break;
    case 2:
      turn_servo(-ang_rot, servo_tvY1);
      turn_servo(-ang_rot, servo_tvY2);
      turn_servo(-ang_rot, servo_tvZ1);
      turn_servo(-ang_rot, servo_tvZ2);
      break;
    case 3:
      turn_servo(-ang_y_2, servo_tvY1);
      turn_servo(ang_y_2, servo_tvY2);
      turn_servo(-ang_x_2, servo_tvZ1);
      turn_servo(ang_x_2, servo_tvZ2);
      break;
    case 4:
```

```
      turn_servo(-ang_y, servo_tvY1);
      turn_servo(ang_y, servo_tvY2);
      turn_servo(-ang_x, servo_tvZ1);
      turn_servo(ang_x, servo_tvZ2);
      break;
    case 5:
      turn_servo(-ang_y-ang_rot, servo_tvY1);
      turn_servo(ang_y-ang_rot, servo_tvY2);
      turn_servo(-ang_x-ang_rot, servo_tvZ1);
      turn_servo(ang_x-ang_rot, servo_tvZ2);
      break;
    default:
      break;
  }
}

void calibrate_magn(uint32_t timeout, int32_t* offsetx, int32_t* offsety, int32_t*
    offsetz) {      //function from IMU library
    int32_t value_x_min = 0;
    int32_t value_x_max = 0;
    int32_t value_y_min = 0;
    int32_t value_y_max = 0;
    int32_t value_z_min = 0;
    int32_t value_z_max = 0;
    uint32_t timeStart = 0;

    ak09918.getData(&magn_x, &magn_y, &magn_z);

    value_x_min = magn_x;
    value_x_max = magn_x;
    value_y_min = magn_y;
    value_y_max = magn_y;
    value_z_min = magn_z;
    value_z_max = magn_z;
    delay(100);

    timeStart = millis();

    while ((millis() - timeStart) < timeout) {
        ak09918.getData(&magn_x, &magn_y, &magn_z);

        if (value_x_min > magn_x){
            value_x_min = magn_x;
        }
        else if (value_x_max < magn_x){
            value_x_max = magn_x;
        }
        if (value_y_min > magn_y){
            value_y_min = magn_y;
        }
        else if (value_y_max < magn_y){
            value_y_max = magn_y;
        }
        if (value_z_min > magn_z){
            value_z_min = magn_z;
        }
        else if (value_z_max < magn_z){
            value_z_max = magn_z;
        }

        Serial.print(".");
        delay(100);
    }
    *offsetx = value_x_min + (value_x_max - value_x_min) / 2;
    *offsety = value_y_min + (value_y_max - value_y_min) / 2;
```

```
    *offsetz = value_z_min + (value_z_max - value_z_min) / 2;
}

void LP_filter(float roll, float pitch, float yaw){
  roll_filtered_ = ALPHA*roll+(1-ALPHA)*roll_filtered;
  pitch_filtered_ = ALPHA*pitch+(1-ALPHA)*pitch_filtered;
  yaw_filtered_ = ALPHA*yaw+(1-ALPHA)*yaw_filtered;
}

float get_goal_angle(float current_time){
  return 0;
}
```

Listing 8: Styrkod servos

```
#ifndef PID_CONTROLLERS_H
#define PID_CONTROLLERS_H
#include <Arduino.h>

double altitude_pid(double altitude, double reference_altitude);

#endif
```

Listing 9: PID regulator för motorstyrning (header-fil)

```
#include "pid_controllers.h"

//PID error variables

double alt_cumulative_error;
double alt_previous_error;

//PID limits

int max_motor_speed = 1940;
int min_motor_speed = 1100;

//PID constants

double alt_Kp = 450;
double alt_Ki = 100.00;
double alt_Kd = 200.00;

//PID controllers

double altitude_pid(double altitude, double reference_altitude){
  double alt_error = reference_altitude - altitude;
  double p = alt_Kp * alt_error;
  double i = alt_Ki * alt_cumulative_error;
  double d = alt_Kd * (alt_error-alt_previous_error);

  alt_cumulative_error += alt_error;
  alt_previous_error = alt_error;

  double pid_val = min_motor_speed + p+i+d;

  if(pid_val>=max_motor_speed){
    return max_motor_speed;
  }
  else if(pid_val<=min_motor_speed){
    return min_motor_speed;
  }
```

```
  else{
    return pid_val;
  }
}
```

Listing 10: PID regulator för motorstyrning

```
#include "pid_controllers.h"
#include <Wire.h>
#include <Servo.h>
#include <Dps310.h>
#include <SimpleFOC.h>

//12 cm fr n toppen IMU
//57 cm fr n toppen Motor

//Initialize components

Servo dc_motor_1;
Servo dc_motor_2;
Dps310 Dps310PressureSensor = Dps310();
LowPassFilter filter = LowPassFilter(0.5);

//Global variables

int dc_motor_1_pin = A2;
int dc_motor_2_pin = A1;

float prev_pressure;
float init_pressure;
float prev_temperature;
int n_samples=0;

const float ALPHA = 0.50;
float altitude_filtered = 0;

float flight_goal_altitude = 1.00;
int flight_time = 300*1000;
unsigned long start_time, current_time;

//Calibrate/set up sensors and components

void setup() {
  start_time = millis();
  Wire.begin();
  Serial.begin(9600);

  dc_motor_1.attach(dc_motor_1_pin);
  dc_motor_2.attach(dc_motor_2_pin);

  dc_motor_1.write(1940);                                    //Initialize main ESC
    calibration sequence
  dc_motor_2.write(1940);
  delay(5000);
  dc_motor_1.write(1100);
  dc_motor_2.write(1100);
  delay(3000);

  Dps310PressureSensor.begin(Wire);

  //Serial.println("Calibrating barometric sensor.");
  init_pressure = calibrate_barometric_sensor();  //Find pressure at ground level,
    takes 17 seconds
```

```
  current_time = (millis()-start_time)/1000;
  while (current_time<30){
     current_time = (millis()-start_time)/1000;         //wait until 30 seconds have
    passed from startup to sync with other arduino
  }

  start_time = millis();
}

//Run control program

void loop() {
  current_time = (millis()-start_time);

  float altitude = measure_altitude();
  float altitude_filtered = filter(altitude);
  //LP_filter(altitude);

  float goal_altitude = 1;//get_goal_altitude(current_time);
  int speed = altitude_pid(altitude, goal_altitude);

  Serial.print("Goal:");
  Serial.print(goal_altitude);
  Serial.print(",");
  Serial.print("Altitude:");
  Serial.print(altitude);
  Serial.print(",");
  Serial.print("AltitudeF:");
  Serial.println(altitude_filtered);
  thrust(speed);
  //thrust(1650);

  if (current_time>flight_time+3*1000){
    exit(0);
  }
}

//Functions

float calibrate_barometric_sensor(){
  float pressure;
  uint8_t oversampling = 1;
  float pressure_mean = 0;
  int num_samples = 0;
  Serial.println("Calibrating...");
  for (int j = 0; j <= 1000; j++){
    int16_t r = Dps310PressureSensor.measurePressureOnce(pressure, oversampling);
    if (r == 0){
      pressure_mean += pressure;    //if nothing went wrong use the sample
      num_samples++;
    }
  }
  Serial.println(".");
  return pressure_mean/num_samples;
}

float measure_altitude(){
  float pressure;
  float temperature;
  uint8_t oversampling = 7;

  int16_t r1 = Dps310PressureSensor.measurePressureOnce(pressure, oversampling);
  if (r1 != 0){
    pressure = prev_pressure;     //something went wrong, return old sample
  }
```

```
  if(n_samples%10 == 0){            //only sample temperature every 10th iteration
    int16_t r2 = Dps310PressureSensor.measureTempOnce(temperature, oversampling);
    if (r2 != 0){
      temperature = prev_temperature;     //something went wrong, return old sample
    }
  }
  else{
    temperature = prev_temperature;
  }

  //Serial.print(",");
  //Serial.print("Temp:");
  //Serial.println(temperature);

  n_samples++;
  prev_pressure = pressure;
  prev_temperature = temperature;
  float height = ((pow((init_pressure/pressure),(1/5.25588))-1)*(temperature+273.15))
    /0.0065;  //formula from https://keisan.casio.com/has10/SpecExec.cgi?id=system
    /2006/1224585971
  return height;
}

void thrust(int speed_main){
  dc_motor_1.write(speed_main);
  dc_motor_2.write(speed_main);
}

void LP_filter(float altitude){
  altitude_filtered = ALPHA*altitude+(1-ALPHA)*altitude_filtered;
}

float get_goal_altitude(float current_time){
  if (current_time <= flight_time/2){
    return 2*flight_goal_altitude*current_time/flight_time;
  }
  else if (current_time > flight_time/2 && current_time <= flight_time){
    return flight_goal_altitude-2*flight_goal_altitude*(current_time-flight_time/2)/
    flight_time;
  }
  else{
    return 0.0;
  }

}
```

Listing 11: Styrkod motor