Nr. _____ av _____

_____

Sjøkrigsskolen

Bacheloroppgave

Autonom Undervannsdrone

av

Naversen, Bo Jenseg og Gran, Magnus Vold

Levert som en del av kravet til graden:

BACHELOR I MILITÆRE STUDIER MED FORDYPNING I ELEKTRONIKK OG DATA

Innlevert: desember 2019

Godkjent for offentlig publisering

I. Publiseringsavtale

En avtale om elektronisk publisering av bachelor/prosjektoppgave

Kadettene har opphavsrett til oppgaven, inkludert rettighetene til å publisere den. Alle oppgaver som oppfyller kravene til publisering, vil bli registrert og publisert i Bibsys Brage når kadettene har godkjent publisering.

Oppgaver som er graderte eller begrenset av en inngått avtale vil ikke bli publisert.

Vi gir herved Sjøkrigsskolen rett til å gjøre denne oppgaven tilgjengelig elektronisk, gratis og uten kostnader	[X] Ja	[ ] Nei
Finnes det en avtale om forsinket eller kun intern publisering? (Utfyllende opplysninger må fylles ut) Hvis ja: kan oppgaven publiseres elektronisk når embargoperioden utløper?	[ ] Ja [X] Ja	[X] Nei [ ] Nei

II. Plagiaterklæring

Vi erklærer herved at oppgaven er vårt eget arbeid og med bruk av riktig kildehenvisning. Vi har ikke nyttet annen hjelp enn det som er beskrevet i oppgaven. Vi er klar over at brudd på dette vil føre til avvisning av oppgaven.

Dato: 03.12.2019

Bo Jenseg Naversen

Kadett, navn

Signatur

Magnus Vold Gran

Kadett, navn

Signatur

III Forord

Denne Bacheloroppgaven er Skrevet av Bo Naversen og Magnus Vold Gran i perioden fra Mai til Desember 2019 som en del av kravet om bachelor i militære studier med fordypning i elektronikk og data. Oppgaven tar for seg mange aspekter ved navigasjon under vann for autonome systemer og løsningene som vi har laget for å overkomme disse.

Oppgaven er interessant fordi den belyser løsninger på tidligere forholdsmessig dyre sensorer som de siste årene har blitt billigere. Inspirasjonen for oppgaven kommer fra tidligere tjeneste hvor vi fant ut at alle nye systemer er kostbare og dermed begrenser antallet og nytten av innkjøpet. Vi bestemte oss derfor at vi skulle lage en undervannsdrone som kunne gjøre mange av de samme oppgavene som de dyrere undervannsdronene som REMUS.

Mange av systemene som vi har montert i dronen er hentet fra tidligere prosjekter som masteroppgaver og vil bli kreditert, men sammensetningen som den fremkommer i dronen vi har laget er unik. Vi vil derfor gjennom oppgaven kreditere kildeholdere fortløpende. En takk skal også rettes til førsteamuensis Alexander Sauter og avdelingsingeniør Frode Wikne for god hjelp underveis med både software og 3D- printing av deler. Vi vil også takke blueye Robotics for hjelp og svar på spørsmål i startfasen av prosjektet.

IV Oppgaveformulering

«En norsk forsvarsindustriell kapasitet innenfor viktige teknologiske kompetanseområder er vesentlig for å sikre forsvarssektoren riktig materiell og kompetanse til rett tid. Dette øker vår evne til å ivareta nasjonal sikkerhet på områder der særnorske forhold krever spesiell kompetanse.» - Meld. St. 9 (2015–2016).

Stortingsmeldingen hevder at økt kapasitet i forsvarsteknologi er påkrevd i årene fremover. Oppgaven svarer på denne satsningen fra Stortinget med å se nærmere på flere viktige temaer for fremtiden, droner og autonomi. Problemstillingen blir derfor: *Hvordan kan man konstruere en autonom undervannsdrone med begrensede midler tilgjengelig?*

V Sammendrag

Oppgaven har med sitt omfang tatt for seg problemstillingen: *Hvordan kan man konstruere en autonom undervannsdrone med begrensede midler tilgjengelig?* Oppgaven tar for seg denne problemstillingen gjennom 6 krav som er ansett som essensielle for en funksjonell undervannsdrone. Disse 6 kravene blir deretter drøftet og utforsket hver for seg til hele systemet settes sammen med den kravspesifikasjonen som er satt.

Oppgaven er interessant sett i sammenheng med at forsvarssystemer og autonomi alltid sikter seg mot et høyt prissjikt og lave kvanta. Dette prosjektet ønsker å gå motsatt vei med å se nærmere på hvilke rimelige muligheter som er mulig å utvikle og på sikt kanskje å produsere. Oppgavens omfang gjenspeiler også omfanget av selve oppgaven og mengde arbeid som har gått med på å utvikle og teste et slikt komplisert system.

Resultatene viser at det er fullt mulig å lage autonome droner som bruker rimelige løsninger for både navigasjon, posisjonering og kontroll på egenrotasjon. Utfordringen ligger derimot i nøyaktigheten disse skal operere under og ikke tilgjengeligheten av billige komponenter på markedet. Oppgaven tar også for seg delkomponenter for blant annet navigasjon som er ansett som den største utfordringen i forkant. Det ble underveis fremvist at man kan lage en nøyaktig posisjonsmåler ved hjelp av kamera under isolerte forhold, samt at det er mulig å videreutvikle til mer avansert bruk med bedre komponenter.

Innholdsfortegnelse

Innhold

Oversikter

Figurer

Formler

Nomenklatur

Pi	Raspberry Pi
	Ettkortsdatamaskin som fungerer som hjernen i systemet
Jetson	Nivida Jetson Nano
	Ettkortsdatamaskin somstyrer avstandsmålingen
Arduino	Arduino Mega 2560
	Programerbar mikrokontroller
Pitch	rotasjon om y- aksen
	opp og ned i fartsretningen
Yaw	Rotasjon om z-aksen
	forklares som sving
Roll	Rotasjon om x- aksen
	forklares som krengning
Input	Data som mottas
	Engelsk for inngang
Servo	Servomotor
	Elektrisk roterende motor med kontrolerbar posisjon

Turnlinje	Er linjen dronen følger når den holder seg på kursen den har fått angitt
Turnpunkt	Er punktet der dronen skal turne videre til neste turnlinje
Thrustere	Motor og propell samlebetegnelse for motor og propell
Drift	Feil eller endring Feil på et parameter over tid betegnes som drift
OpenCV	Open Source Computer Vision Bildebehandlingsprogram som nyttes til både enkle og komplisere bildeoperasjoner
Strølys	Ikke ønskelig lys fra lyskilde Lysstøy fra en lyskilde
Kamera-feed	Nåtids visning av kameraets bilder/video
Tracke	Å følge etter Når datamaskinen klarer å følge etter et spesifikt gitt objekt
Hamming Distance	Et mål på forskjellen mellom to binære datastrenger

REMUS	Autonom undervannsdrone fra Kongsberg
Pixelendring	Endringen et punkt/område har i et bilde, målt i pixler
Pixelavstand	Avstand mellom to eller punkter/områder i et bilde, målt i antall pixler
Feauture Matching	Metode for å finne to lignende punkter/objekter i mellom to bilder
Seriellkommunikasjon	Metode for å kommunisere mellom maskinene på
GPS-fix	Når GPS-sensor har fått rett posisjon

Forkortelser

INS	Inertial Navigation System
	akselerometerbasert navigasjonssystem
HMI	Human Machine Interface
	Grensesnittet hvor oprøteren leverer input til maskinen
MEMS	MikroElektroMekaniske systemer
	Sensorer som benyttes til måling av f.eks pitch/roll/yaw
VVS	Varme-Ventilasjon-Sanitærteknikk
	Fellesbegrep for deler og komponenter som benyttes til slike forhold
PID	Proporsjonal-Integral-Derivasjon
	Algoritme som benyttes i reguleringsteknikk for å motvirke ytre påvirkninger
I2C	Inter-Integraded Circuit
	Er en seriell databus for kommunikasjon med mange MEMS

1. INTRODUKSJON

1.1 Bakgrunn

Vi lever i en tid der satsningen mot undervannsdroner øker da det har store gevinster militært og kommersielt. Fra et militært standpunkt blir ubåter mer og mer avanserte, og samtidig enda mer stillegående enn før, noe som spesielt i et norsk perspektiv er av særegen interesse der norske ubåter er sett på som en strategisk ressurs. Men mens de blir mer avanserte blir de også kostbare og det vil være nyttig å vurdere billigere løsninger for å møte utfordringene man står ovenfor. Kommersielt er det store muligheter spesielt mot inspeksjon og vedlikehold av maskineri og rør som befinner seg på dype hav, som i dag kan kreve store ressurser å holde i drift. Videre er den sivile skipsfarten nødt til å vedlikeholde og inspisere

sine fartøy, og kostnadene med å legge skip i tørrdokk eller utføre inspeksjoner er såpass store at det åpner muligheter for undervannsdroner til å utvikles for å ta over jobben.

Mulighetene er mange, og i en tid der dette forskes mye på åpner også mulighetene seg enda mer. Hvis man ser på vanlige kommersielle og militære undervannsdroner i dag, baserer konseptet seg ofte på en kablet løsning. Det vil si at dronen har en kommunikasjonskabel som følger dronen fra overflaten og til en operatørkonsoll. Med andre ord er dronen ikke bare avhengig av menneskelig input og styring for å kunne bevege seg sikkert under vann, men også at data sendes fysisk over kabel. Dette fører til kostnader som man ikke nødvendigvis ønsker, men enda viktigere fører det til *begrensninger* for hva undervannsdroner kan evne å få til. Man kan ikke bruke kablede undervannsdroner til operasjoner langt unna en menneskelig operatør, ei heller operere dronen i farvann som går på risiko for å ødelegge kabelen.

Årsaken er som kjent at signaler ikke går spesielt langt under vann, som gjør posisjonering og rekkevidde til et problem kontra overflatefartøy og flygedroner over vann som kan bruke trådløse signaler som GPS og radiobølger. Dette har ført til at utviklingen av undervannsdroner har sakket akterut til sammenligning til dronene som kan operere med tilgang på signaler. Dette tomrommet i utviklingen er noe som i slike tider er meget interessant å se videre på, da mulighetene hvis man har suksess er store. Ikke bare vil man kunne gi utviklingen av området et solid hopp, men også bidra til videre utvikling på andre områder som opererer under vann. Ser man på militære ubåter brukes det kostbare sensorer og god menneskelig navigering for å ha kontroll på sin egen posisjon under vann. Selv med alt dette på plass vil man ha unøyaktigheter som må kompenseres for med å dykke opp til overflaten.

Det finnes veldig få undervannsdroner som evner å drive på autonomt grunnet disse utfordringene. Mange av utfordringene baserer seg på at slike kostbare sensorer som brukes av ubåter ikke er tilgjengelig på markedet. Slik har mangelen på en GPS-erstatning gitt muligheten for å kunne utvikle noe helt nytt, som skiller seg ut fra tidligere oppgaver som er levert på sjøkrigsskolen. Det er denne muligheten vi har lyst å prøve ut, selv om det vil kreve mange flere timer og mye større omfang enn hva en vanlig oppgave på Sjøkrigsskolen var tiltenkt å ha.

Autonomi er en av de elementene som får mye oppmerksomhet i disse dager. Å konstruere en autonom drone fra bunnen av vil kreve at man lærer seg enormt mye på ulike fagområder på relativt kort tid. Det vil også kreve at vi nytter oss av elektronikk og teknologi som finnes

på det sivile markedet til en rimelig pris. Oppgaven har til hensikt å løse problemstillingen: *Hvordan kan man konstruere en autonom undervannsdrone med begrensede midler tilgjengelig?*

1.2 Mål

For at en drone skal kunne være effektiv og sikker uten å være styrt manuelt/kablet, krever det en sensorpakke god nok til å måle posisjon og fart, samt et robust kontrollsystem som kan oppdage hindringer. Systemet bør dermed evne å vite sin egen relative posisjon til enhver tid og oppdage objekter under vann. Dernest må denne informasjonen kunne raskt behandles av kontrollsystemet som så skal agere hensiktsmessig slik en menneskelig operatør ville gjort.

Målet med oppgaven er å finne ut om med billige og enkle midler det er mulig å lage en fullt funksjonell undervannsdrone. Med "funksjonell" mener vi at oppgaven skal tilstrebe å utvikle en drone som har de samme funksjonene og evnene som en vanlig kommersiell kablet undervannsdrone. Videre legges det til grunn at det kun er navigeringen under vann som vil bli utviklet, men at oppgaven samtidig skal ha som mål å utvikle dronen til å kunne brukes i videreutvikling. Slik vil oppgaven lage en base for en autonom undervannsdrone som så kan med enkle utskiftninger og oppgraderinger bli brukt til å løse mer spesifikke oppgaver som for eksempel inspeksjon av skrog, søk og redning og mer militære formål. Dette innebærer at dronen må kunne måle sin egen posisjon, ha kontroll på egen rotasjon og agere når den støter på hindringer, ha kontroll på når og hvordan den skal bevege seg og oppdage og unngå objekter den støter på. Ut ifra dette formulerer vi følgende hovedmål:

«*Dronen skal kunne manøvrere under vann med alle forutsetninger som følger med, på lik linje med en vanlig kommersiell kablet drone. Denne skal så kunne brukes som base for videreutvikling til mer oppdragsspesifikke oppgaver*»

For å teste konseptet vil dronen bli satt ut i et miljø som har klar sikt og rolige vannforhold, i den hensikt å fjerne flest mulig feilkilder og sørge for en god basis for utviklingen. Den vil så bli gitt en gitt en rute den skal navigere som skal ende opp tilbake i utgangsposisjon, der en eller flere objekter vil bli lagt i veien for ruten slik at den må agere. Dronen vil da bli utsatt for et ukjent miljø med hindringer den må bruke kontrollsystemet sitt til å trygt og nøyaktig

navigere ruten den er blitt gitt. Sensorpakken som dronen benytter vil være hentet fra kommersielle aktører til lav kostnad og bestå av kameraer, lasere og mikroelekromekaniske sensorer(MEMS). Samlet og integrert skal disse være nøyaktige nok til at dronen skal kunne tilfredsstille målet.

1.3 Begrensninger

Vi har kort snakket om utfordringene knyttet til at signaler og lys ikke fungerer under vann. I all hovedsak skyldes dette at vann som væske bryter lys og dermed endrer retningen. Vår største begrensning er altså at så lenge vi befinner oss under havoverflaten vil vi ikke ha tilgang til det som er det foretrukne systemet for samtlige fartøy i verden når det kommer til posisjonering. GPS er forhåpentligvis kjent for leseren, men i korte detaljer baserer systemet seg på å motta posisjon fra minimum tre satellitter som triangulerer posisjonen din. Dette gir nøyaktig posisjon til ethvert system som bruker det. Vi må derfor i utviklingen gjøre det klart at vi må finne andre systemer som kan gjøre samme jobb, og satse på at det evner å ha tilnærmet samme nøyaktighet.

Der GPS er tilgjengelig til lav kostnad på det sivile markedet, er systemene som nyttes som erstatning i militære undervannsfartøy veldig dyre. Til sammenligning med en militær ubåt for posisjonsmåling brukes det data fra såkalte Inertial Navigation Systems (INS) i samhandling med sensorer som måler hastigheten i vann. Der ubåtens INS-systemer koster ekstremt mye vil de også ha meget god nøyaktighet til å måle posisjon og rotasjon om aksene i det tre-dimensjonale rom. Dette er systemer som er for dyre for oss og heller ikke tilgjengelig på det sivile markedet, og er ikke en løsning vi kan benytte oss av. Når man har gode data på områdets strømninger vil man kunne bruke hastighetsmåling til å bedømme posisjon enda mer nøyaktig. Hastighetsmåler som brukes på ubåter, såkalte pitometer er dyre og store i størrelse som gjør det uaktuelt for oss å bruke dem. Vi har altså en begrensning i hvilke systemer vi har tilgjengelig for å kartlegge posisjon under vann.

Vi er to personer som skal utvikle konseptet med begrensning i tid på litt over 6 måneder med påfølgende ferier og andre fag. Ser vi i sammenligning med hvordan slike konsepter ville vært utviklet militært og kommersielt har vi liten tidsfrist og begrenset med arbeidskapasitet. Videre vil vi definere leveringssystemet til Sjøkrigsskolen som en begrensning i utviklingen av konseptet, da leveringer tar alt fra 4 til 10 uker. Dette har vist seg å hemme utviklingen ved at oppgaven har blitt satt på vent ved flere anledninger i mangel på deler. Firmaer som driver utvikling har muligheten til å påvirke leveringstiden med å betale mer for å få delene levert tidligere, noe vi ikke har mulighet til å gjøre selv om pengene er tilgjengelig. Leveringen må gå gjennom et ledd hos Forsvarets Høgskole, før det

i det hele tatt kan bestilles til oss. Denne begrensningen ble en enda større påvirkning da vi i begynnelsen ikke var klar over at den eksisterte i et slikt omfang.

For firmaer og forskningsgrupper med større budsjett enn oss, vil det være mulig å designe og konstruere skroget til undervannsdronen etter hva som passer best i forhold til kravene som er satt. Vi har ikke tilgang til å drive med egenkonstruksjon samt at gode ferdigløsninger for et skrog er for kostbare i forhold til budsjettet vi har tilgjengelig. Med ferdigløsninger mener vi løsninger som er spesifikt designet for å bli brukt som skrog for undervannsdroner. Som nevnt er slike ferdigløsninger dyre og ikke innenfor vårt budsjett, som gir oss visse begrensninger. Vi må velge noe som ikke er designet i utgangspunktet for å blir brukt som undervannsdrone, men som likevel er vanntett og hydrodynamisk. Denne løsningen må så få implementert tilleggsutstyret vi trenger for å bruke den til vårt formål, noe som krever tid og utvikling av kompetanse. Videre vil en slik ferdigløsning ha et endelig gitt rom tilgjengelig til montering av nødvendig elektronikk, der rommet også ikke er designet for dette vil det også medføre ekstra montering av plassholdere og andre løsninger for å holde elektronikken stabil. Til slutt vil en løsning med thrustere ikke være hensiktsmessig både grunnet kostnaden, men også at skroget ikke er designet for å ha slike løsninger som dermed fører til at man ikke kan få til en vanntett montering. Dette vil igjen gjøre dronen mindre fleksibel for forflytning om alle aksene, som igjen er en begrensning i hvordan vi utvikler styringssystemet. Design og konstruksjon av skrog er dermed en begrensning som vil ha stor påvirkning på hvordan vi utvikler systemet.

1.4 Metode

Med hensyn til hovedmålet vi har satt og begrensningene vi har oppsummert over, vil arbeidet med oppgaven kreve at vi i starten må lese oss opp på ulike konsepter og systemer. Først ble det tidlig klart at begge to ikke kunne drive med hele utviklingen av oppgaven sammen. Dette var en vurdering i lys av at oppgaven i våre øyne er ganske stor og omfattende til en bachelor-oppgave å være, noe som innebærer at det vil kreve enormt mye for å komme i mål. Derfor var det naturlig å dele opp oppgaven i to separate linjer der vi hver for oss har ansvaret for vår linje, med påfølgende enighet om hvordan linjene skal utformes. Inndelingen var i hardware og software, hvor hardware også innbefatter skrogkonstruksjon og design.

På software-siden ble kompetanse på området startet med allerede i januar 2019 for å bygge tidlig forståelse av de ulike sensorpakkene og systemene vi trengte. Rent praktisk ble det bestemt at styringen av all fremdrift og behandling av de fleste sensorkomponentene skulle skje via en Arduino Mega mikrokontroller. Vurderingen ble gjort med bakgrunn i at det er kanskje den mest kjente mikrokontrolleren på markedet for hobbyvirksomhet innenfor

elektronikk, som er sammenfattende med å dermed ha mest informasjon og hjelp tilgjengelig på nettet. For utvikling av avstandsmåler skal det nyttes kamera og lasere til å beregne avstand til objekter, som skal skje via bruk av Python som program. Python er i likhet med Arduino veldokumentert på nettet og innehar ekstremt mange tilleggsfunksjoner man kan ta i bruk. Språket har også flere likheter med programmeringsspråket C++ vi har lært i andre fag på skolen, og er enkelt å lese seg opp på. Til maskin brukes her en Jetson Nano(Heretter kalt Jetson). For posisjonsmåling blir det under utviklingen testet bruk av akselerometer samt kamera for å gi ut en nøyaktig nok posisjon, og sjekke om dette vil være godt nok for å tilfredsstille hovedmålet. For bruk av kamera vil vi også nytte oss av Python. Til maskin brukes det her en Raspberry Pi(Heretter kalt Pi).

For hardware ble det fokusert på å bruke egen-erfaring med konstruksjon i sammenheng med et besøk hos Blueeye Robotics for å høre deres tanker rundt konseptet. Som skrogfasong brukes et PVC-rør som normalt brukes for varme-ventilasjon-sanitærteknikk(VVS), da dette røret har som krav å være tett. Ut ifra denne fasongen ble ekstra utstyr montert på som to motorer, flapper til å rotere dronen og gjennomsiktig pleksiglass for at kamera skal kunne se ut.

For testing og validering av dronens egenskaper vil vi benytte oss av svømmehallen på Haakonsvern som tilfredsstiller målet om å teste i et miljø som har klar sikt og rolige vannforhold. Her vil vi teste flere av komponentene separat som å måle nøyaktigheten i avstandsmålingen under vann, posisjonsmålingen og hvordan dronen reagerer på påvirkninger som for eksempel uønsket rotasjon. Målingene vil plottes og sammenlignes med reell data for å se hvor nøyaktige komponentene er. Før testen foregår i svømmehallen vil vi gjennomføre testene tørt i klasserom for å forsikre oss om at ting fungerer som forventet på land før de gjøres under vann. Den siste og avgjørende funksjonstesten er hele systemet testet sammen, der vi gir dronen en rute den skal navigere med objekter i veien, slik at vi får testet og validert hele konseptet. Under testen vil posisjonsdataen logges for sammenligning mens avstandsmåleren vil bli filmet og validert fysisk av oss. Slik vil vi sikre oss at vi har data og egenerfart validering av konseptet til å kunne besvare problemstillingen hensiktsmessig.

1.5 Struktur

Oppgaven er delt inn i ulike deler, der målet er at den videre lesingen skal ha en rød tråd i seg. For å kunne forstå helheten er man avhengig av å først se på hvordan de ulike del-systemene er utviklet, testet og drøftet. Når de er gjennomgått separat vil man sette det i system slik at man kan oppnå en forståelse av hel-systemet.

Først vil det bli formulert ulike systemkrav eller under-krav basert på hovedmålet spesifisert i 1.4, som er de direkte styrende oppgavene til del-systemet. Ut ifra dette vil man utvikle de nevnte del-systemene med hensikt å tilfredsstille kravene. For å oppnå en god struktur vil det være to separate kapitler som beskriver del-systemene henholdsvis grovt og mer detaljert. Hensikten er å kunne gi forståelse både oversiktlig og mer spesifikt avhengig av hva leseren selv ønsker. Man vil til slutt gå gjennom tester/resultater og drøfte hvordan hvert enkelt system svarer til sitt spesifikke systemkrav, for igjen å kunne svare på hvordan hel-systemet svarer til hovedmålet. Konklusjonen vil oppsummere systemkravene, som igjen oppsummerer hovedmålet i den hensikt å til slutt kunne besvare problemstillingen.

Det er derfor viktig at leseren er klar over at hver enkelt del har en egen struktur som minner om en egen oppgave. Dette begrunnes med at oppgaven i sin helhet er såpass omfattende at det krever ekstra støttestruktur for å opprettholde den røde tråden i oppgaven. Teksten er skrevet med forventning om at leseren har noe kunnskap og erfaring innenfor emner som blir beskrevet her, da spesielt de fag vi har hatt i løpet av vår tid på Sjøkrigsskolen. Dette vil spesielt gjelde for dataprogrammering og elektronikk.

Vi har valgt å ikke ha et egent kapittel som heter «teori» da denne kommer frem i andre deler av oppgaven. Teoridelen i denne oppgaven ville på mange måter blitt overflødig da det er mer naturlig å forklare de viktigste delene av teorien i sammenheng med konseptet. En annen grunn til at teorikapittelet ikke er tatt med er fordi teorien ligger i denne sammenheng ute som åpne kilder. Disse kildene vil i denne sammenheng ofte være forum eller andre plattformer som er lett tilgjengelige.

2.Utvikling av konsept

Denne delen av oppgaven skal ta for seg hvordan vi har valgt å løse kravene som er satt ved å gjennomgå hver del-komponent og sammensetningen. Utviklingen deles som nevnt i 1.5 i to kapitler der første del går mer grovt igjennom utviklingen, mens det mer detaljerte og kodemessige som er blitt gjort kommer i kapittel 3. For de som ikke har behov for å vite spesifikt hvordan del-systemene er utviklet, er det ikke nødvendig å lese kapittel 3 for å forståelse.

Det presiseres også at dette delkapittelet er strukturert med egne innledninger, teori- og drøftingsdeler som minner om separate oppgaver, men er helt nødvendig strukturelt for å oppnå omfanget som denne oppgaven gir.

2.1 Systemkrav

Del-kapitlet vil handle om å få en oversikt over kravene til del-systemene, heretter kalt systemkrav. Systemkrav defineres som vår vurdering av hva som må kreves av dronen basert på hovedmålet med oppgaven. Hovedmålet fra 1.2 var:

«*Dronen skal kunne manøvrere under vann med alle forutsetninger som følger med, på lik linje med en kommersiell operatørstyrt drone. Denne skal så kunne brukes som base for videreutvikling til mer oppdragsspesifikke oppgaver*»

Oversikten vil ta for seg ulike begrunnelser og vurderinger som ligger til grunn for kravene fastsatt for dronen. Oppgaven vil systematisk gå igjennom kravene med referanse til teori/erfaring, vurderinger og fastsetting av kravet. Mye av argumentasjonen og krav-settingen er sammenlignet med operatør-styrte droner basert på hovedmålet, og for en enklere formulering.

2.1.1 Krav til Avstandsmåler/objektdeteksjon

For kommersielle droner er det operatøren selv som oppdager objekter og styrer unna dem for å hindre en direkte kollisjon. Operatøren vet ikke avstanden til objektene får som regel følelsen av hvor nært eller fjernt objektet er basert på størrelsen til objektet.

I lys av dette vil de fleste kommersielle undervannsdroner ikke ha noe krav om at dronen skal automatisk unngå kollisjon med objekter eller grunn. Dette er derimot ikke linjen en denne dronen kan legge seg på når den skal være designet for å være autonom. Det må derfor ettes som krav at dronen skal unngå kollisjon, herav alt av faste objekter som finnes under vann. Det fastsettes derfor:

Krav 1: «*Dronen skal på egenhånd evne å detektere objekter innenfor en rimelig avstand, samt å beregne avstanden til objektene*»

2.1.2 Krav til kontroll på egen-rotasjon om de tre aksene

Når man opererer under vann må en forholde seg til samtlige av de tre dimensjonene x, y og z. Der en operatør-styrt drone vil kunne ta i bruk operatørens forståelse av omgivelsene til å bedømme egen-rotasjon, vil vår drone ikke ha denne muligheten. Selv operatører vil møte på situasjoner der man ikke vet hvilken retning man styrer da siktforhold og feil persepsjon kan forstyrre. Vår drone må dermed klare å holde styr på sin egen-rotasjon i forhold til alle tre aksene, slik at den til enhver tid er klar over hvordan den befinner seg i et tre-dimensjonalt rom. Det fastsettes derfor:

Krav 2: «*Dronen skal evne å holde kontroll på sin egen rotasjon i det tre-dimensjonale rom*»

2.1.3 Krav til kontroll på egen posisjon

For dronene man finner på markedet har det allerede blitt snakket om kravet om en operatør til stede for å unngå objekter. Den neste mangelen man ofte finner på slike kommersielle droner er at de ikke holder posisjonen sin. Elektromagnetiske bølger brer seg forskjellig i overgangen mellom medier, slik som luft og vann. Dette gjør at GPS- signaler er ubrukelige under vann. Utfordringen ligger derfor i at GPS er det foretrukne posisjonsverktøyet på overflaten, men ubrukelig under vann.

Da dronene er koblet via kabel opp til overflaten vil det alltid være mulig for operatøren å «fiske opp» dronen når man skal ha den tilbake, eller å styre den selv. For en autonom drone vil man ikke ha en kabel den bare kan følge opp igjen, den er derimot avhengig av å vite hvor den har dratt slik at den kan følge samme vei tilbake til der den startet. Samtidig finnes det flere metoder man kan velge som ikke krever at en holder mål på posisjonen. Ser man på en autonom gressklipper er den utstyrt med en sensor foran som snur unna objekter. Men den har heller ikke kontroll på hvor den er til enhver tid som selvsagt gjør klippingen til dels ineffektiv. Samtidig vil denne være avhengig av en fast områdeavgrensning lagt ned ved bruk av kabel da hageplenen selvsagt ikke endrer form og farge i løpet av dagen.

En slik løsning er ikke tilfredsstillende for en drone tiltenkt å operere i ulike havområder og havner, slik at det er en løsning som gjør det mulig å kjøre innenfor begrensningene som gis av operatøren, uten at man på forhånd må legge ut fysiske gjenstander som skal gjøre jobben. Her kunne selvsagt en posisjonsoppdatering lignende den brukt på en REMUS nyttes. REMUS har en sonar som legges i vannet som sender ut lydbølger med informasjon om posisjon til moderfartøyet som sender ut informasjonen. REMUS finner da avstanden fra moderfartøyet til seg selv ved triangulering. Men dette er ekstremt kostbart og vil gå utenfor budsjettrammene våre. Det trengs dermed et fastsatt posisjoneringssystem ombord på dronen og setter:

Krav 3: «*Dronen skal ha et eget posisjoneringssystem som holder kontroll på posisjonen under vann*»

2.1.4 Krav til et reaktivt styringssystem

Det ble fastsatt krav til at dronen skal ha sensordata som gir den kontroll på egenrotasjon, posisjon og mulige objekter. Dronen er også avhengig av at informasjonen bearbeides og at det tas korrekte vurderinger basert på denne bearbeidingen. For de fleste kommersielle dronene er det operatøren som innhenter all denne informasjonen, som ofte gis til operatøren på en skjerm eller annen form for HMI. Slik vil operatøren kunne agere på oppdukkende objekter, rotere dronen etter eget ønske og styre den i den retningen han måtte ønske.

For vår del innebærer dette at dronen har en form for «hjerne» som mottar all relevant informasjon for bearbeiding, for så å styre dronen i en satt retning. Man kunne her ha kjørt videre på gressklipper-analogien om å la deler i dronen arbeide separat, der posisjonen i ettertid kunne loggføres for å se hvilken rute som man endte opp med. Men en har allerede nevnt i kravet til posisjonering at man ønsker å nytte dronen i ulike områder, og at det skal gjøres mest mulig effektivt. Det settes derfor:

Krav 4: «*Dronen skal benytte sensordata for å være i stand til å styre hensiktsmessig, samt å kunne reagere raskt og effektivt på forstyrrelser og hinder*»

2.1.5 Krav til operasjonstid

Det er uklart hva som bør være en god nok operasjonstid for dronen vår. Som referanse brukes den operatørstyrte dronen til BlueEye. På nettsiden deres oppgir de en brukstid på to timer. Sett i forhold til prisantydningen for en slik drone ligger den på fire ganger over denne oppgavens budsjett, noe som vurderes dit at den burde holde ut lenger enn oppgavens drone. Med hovedmålet lagt til grunne må det også legges vekt på at oppgaven har et begrenset budsjett til å kjøpe inn god batterikapasitet.

Krav 5: *«Dronen skal kunne ha en god nok operasjonstid til å demonstrere at den oppnår hovedmålet, anslagsvis et sted mellom 30 min og én time»*.

2.1.6 Krav til fremdrift og forflytning

Kommersielle droner har som regel elektromotorer ofte referert til som thrustere til å forflytte seg og rotere om aksene. Det var en mulighet å gå på nettet og kjøpe ferdige skall til å bruke, men disse er kostbare og ville spist opp mye eller hele budsjettet for oppgaven. Ønskelig skulle kravet om at dronen kan forholde seg i ro blitt satt, men dette innebærer innkjøp at ekstra thrustere og et skrogkonsept som ikke er mulig å kjøpe for en billig penge. Det ble derfor naturlig å se andre løsninger da hovedmålet ikke krever stillestående drone. Dronen skal også evne å holde seg selv stabil i både pitch og roll til fordel til utstyr som kan implementeres. Det settes:

Krav 6: «*Dronen skal evne å ha fremdrift og rotere om aksene, selv i mye motstrøm»*

Det har nå blitt satt 6 krav til dronens egenskaper. I kapitlet om utvikling vil det bli gått igjennom hvordan disse kravene er blitt løst, og hvilke vurderinger som ligger til grunn for valgene som har blitt gjort på bakgrunn av kravene.

2.2 Avstandsmåler/Objektdeteksjon

Basert på 2.1.1 krav til Avstandsmåler/Objektdeteksjon må dronen unngå objekter selvstendig. Delkapittelet 2.2 vil omhandle hele prosjektets avstandsmåling og

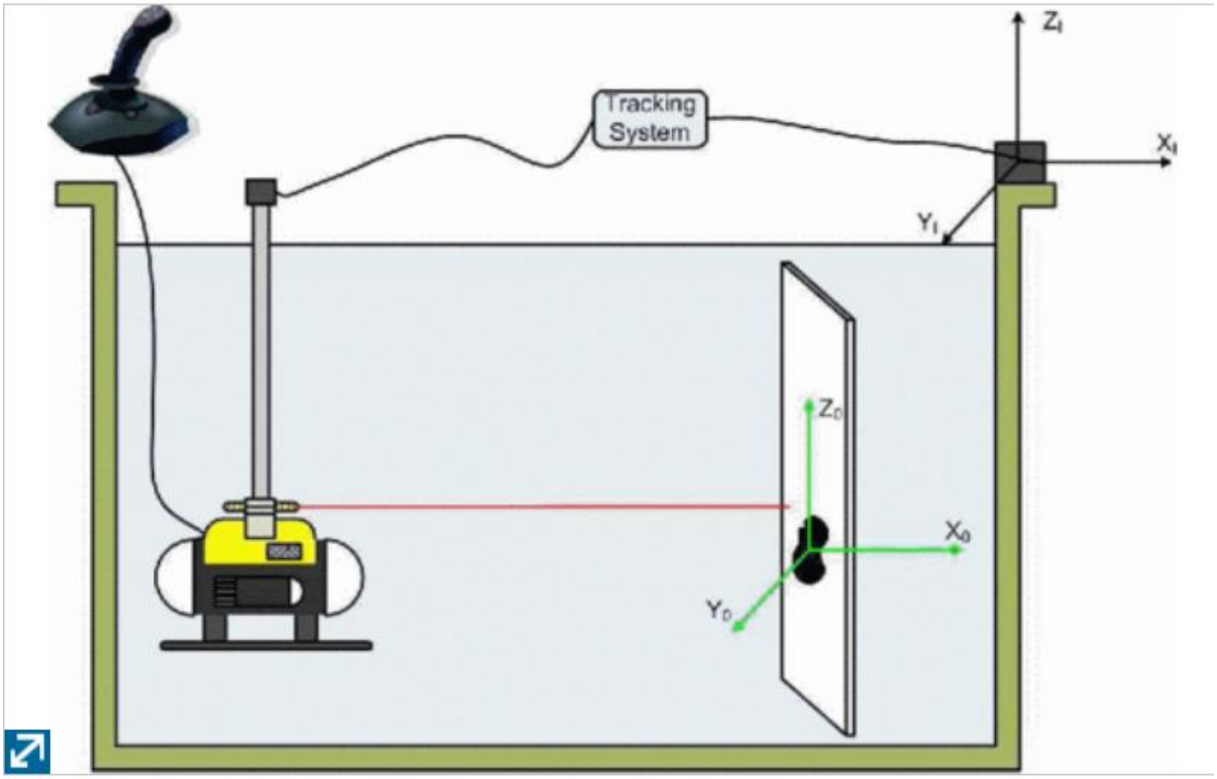
objektdeteksjon med en egen selvstendig oppbygning, igjen på bakgrunn av oppgavens omfang. Det vil i tillegg foregå en drøfting av valgene som er gjort også i denne delen av oppgaven da det faller naturlig på bakgrunn av oppgavens karakter og omfang.

Da dronen er utstyrt med et kamera forut i skroget, vil kapittelets innhold ta for seg å oversette forholdet mellom pixelavstand i sin egen kamera-feed i forhold til avstanden til et objekt. Dette innebærer at man trenger å ha en forståelse av hvordan en maskin kan tolke bilder på.

En kan se på dronen på lik linje med en menneskelig operatør kunne bedømme den totale pixelstørrelsen på et objekt i kamera-feeden og så bedømt ca avstand. Problemet er derimot at dronen ikke vet hva som er normal størrelse på objektet. Skulle for eksempel dronen oppdage en skrogplate på en god avstand, kunne den tro at det var en mindre plate som var veldig nær og dermed viket unna selv om man befant seg på trygg avstand. En vanlig operatør vil kunne tolke objektets faktiske størrelse mye bedre, og dermed avgjør nærheten til dronen. For dronen trenger vi å få til et referansepunkt, slik at den kan tolke hva den ser rett.

Idéen kom gjennom en fysikk-time vi hadde der man lærte om hvordan man ved å sende lys gjennom et spekter kunne bedømme avstanden til en vegg basert på spredningen mellom lyspunktene. En maskin vil kunne skille ut spesifikke fargeområder og vite hvor i bildet de befinner seg, noe som gir et referansepunkt. Samtidig vil slike stråler ha for liten effekt til å gi en god nok rekkevidde på deteksjonen, da man vil være avhengig av at kamera ser differansen imellom lyspunktene. Men, om en nå bytter ut lyspunktene i spekteret med egne konsentrerte stråler vil man også kunne justere avstanden slik at vi maskinen kan detektere en differanse på større avstander.

Nøkkelen baserte seg på artikkelen til George Karras med fler[1]. Artikkelen utforsket å montere lasere på siden av en drone, som ville prosjektere seg over på et objekt. På lik linje med hvordan mennesker ser ting som er langt borte som mindre, vil to laserpunkter virke nærmere hverandre jo lenger borte man er. Maskinen vil tolke dette på lik måte. Vi har dermed et referansepunkt som kan tolkes rett, slik at vi kan bedømme avstanden til objekter.

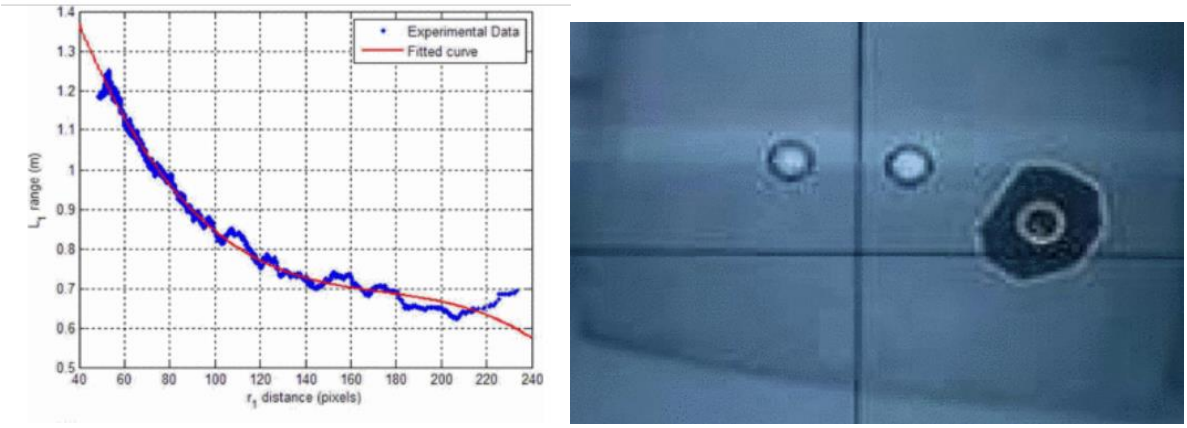


Figur 1: En drone med monterte lasere som ser mot et flatt objekt for å bedømme avstanden(Karras et al(2006))

I artikkelen bruker man et program som heter Snakes for å detektere de to laserpunktene. Når man har posisjonen til dette kan man så predikere avstanden. Måten dette gjøres på er å bruke statistikk med å plotte avstanden i pixler mellom laserpunktene i forhold til den faktiske avstanden, og så lage en trendlinje. Ved bruk av trendlinjen vil vi ha en funksjon som tar inn pixelavstand og gir ut avstanden til objektet. Med pixelavstand mener vi avstand i pixler mellom to objekter/områder i bildet.

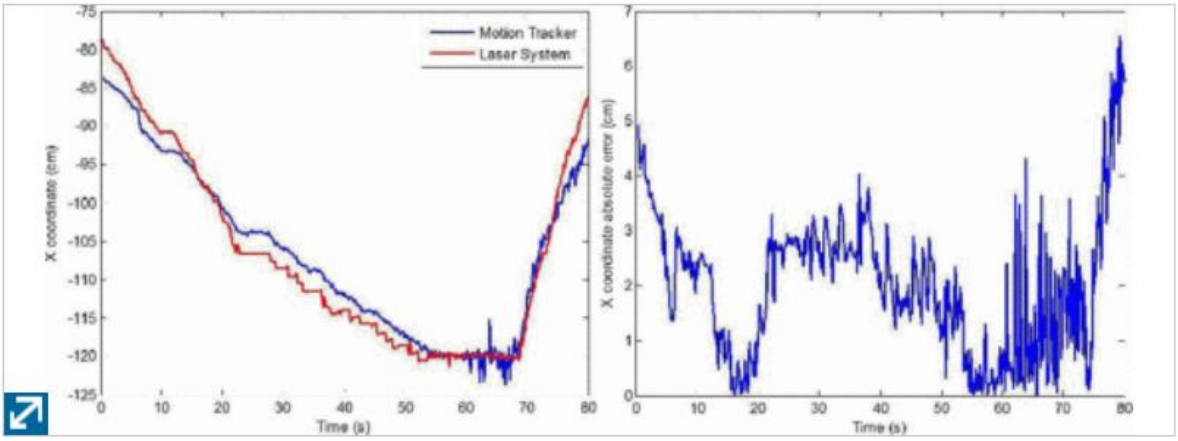
$$f(x, y) = avstand\ til\ objekt$$

Hvor x og y er avstand i pixler mellom laserpunktene i x- og y-retning.



Figur 2: Plottet data sammen med trendlinje, samt utsnitt av video-feed fra kamera. Begge bildene er hentet fra Karras et al(2006)

Etter å ha implementert trendlinjen i programmet til dronen, har man festet et faktisk posisjoneringssystem til dronen for å sammenligne med algoritmen. Man kan se her at resultatene er ganske gode.

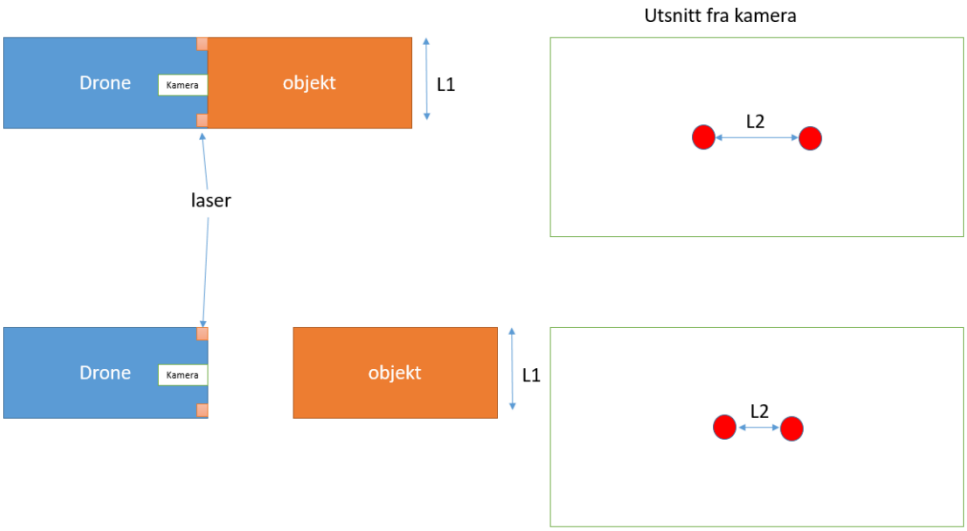


Figur 3: Resultatene til Karras. Bildet til venstre viser faktisk posisjon sammen med predikert posisjon. Til høyre ser vi feilen til algoritmen i forhold til faktisk posisjon.(Karras et al(2006))

Det tas utgangspunkt i fremgangsmåten til Karras[1] for hvordan man skal løse kravet om å kunne unngå objekter.

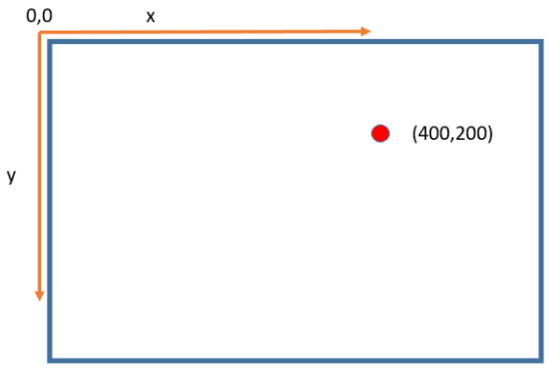
2.2.1 Vår løsning

Vi har nå en fremgangsmåte for å bedømme avstand til objekter, og skal implementere en løsning basert på dette til vår drone. Dukker det opp et objekt foran dronen vil lasernes projiserte laserpunkter treffe objektet. Ref artikkelen til Karras[1] vil det for kameraet virke som laserpunktene er nærmere hverandre jo lenger borte fra dronen objektet er og motsatt for når det er nært.



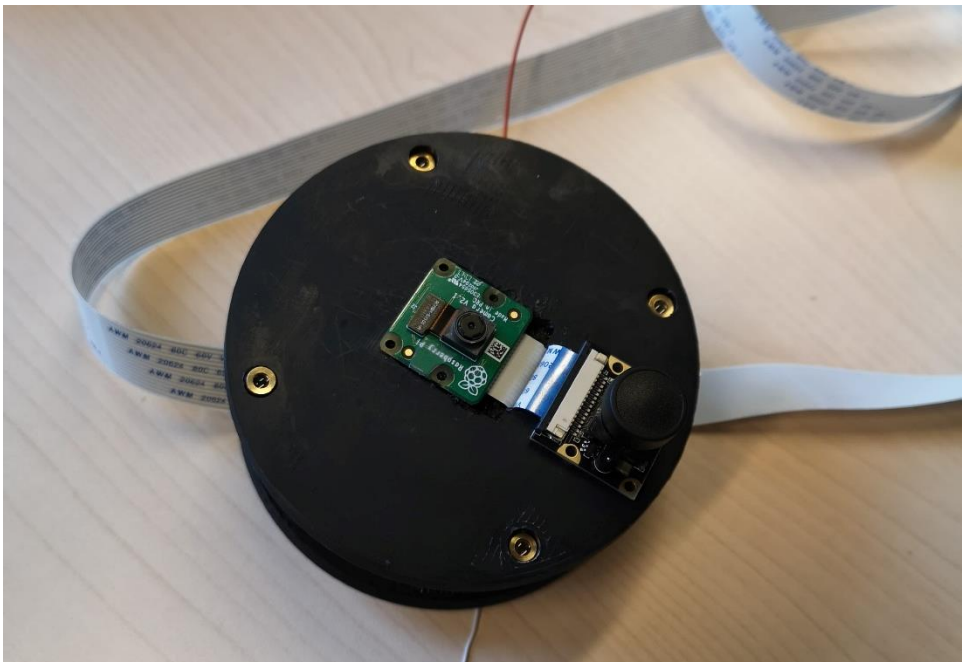
Figur 4:Illustrasjon av hvordan laserne forholder seg til hverandre i forhold til den faktiske avstanden til objektet

Hvis vi så vet forholdet mellom endring i pixelavstand mellom laserpunktene og den faktiske avstanden, kan vi bedømme avstand mellom drone og objekt. Vi trenger derfor å vite posisjonen i bildet til laserpunktene, og så utføre vanlig matematisk utregning for avstand mellom to punkter. Vi definerer så ordet pixelkoordinat som betyr posisjonen i bildet gitt ved pixler i x og y.



Figur 5: Illustrasjon av pixel-koordinater i et bilde

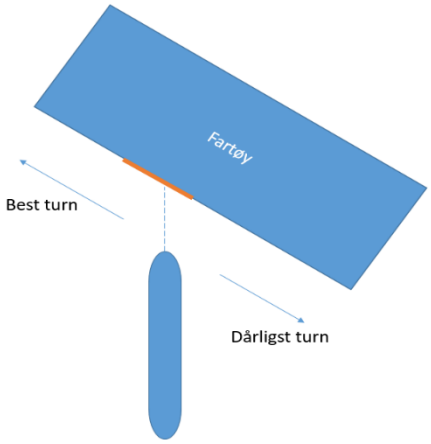
Merk at vi starter koordinatsystemet oppe i venstre hjørne da det er slik programmet vårt definerer dem, og for å unngå unødvendig konvertering definerer vi det slik også. Det neste for oss er å 3D- printe en holder for å holde lasere samt med et kamera for billedtaking. Ser man på figur 2 ser man at Karras[1] har benyttet seg av to lasere, som er nok til å bedømme avstanden. Men vi har valgt å benytte oss av 4 lasere plassert i hjørnene til et tiltenkt kvadrat med kameraet i midten.



Figur 6: Kamera-rack med 4 lasere sammen med et Pi-kamera, det svarte kameraet hører ikke til her

Grunnen til at vi velger å ha fire er at vi ønsket å hente ut vinkelen til objektet også. Dette ble vurdert med bakgrunn i masteroppgaven til Andreas Henriksen[2], som vi kommer tilbake til. På dette tidspunktet i planleggingen av dronens oppsett og styringssystem var det enda ikke avklart nøyaktig hvordan vi ønsket å styre rundt objekter på. Styringssystemet vil diskuteres senere i kapitlet, men det viktige å ta med her er at en av løsningene som ble vurdert innebar å vite vinkelen objektet hadde i forhold til dronen. Det var samtidig ønskelig at hvis objektet forholdt seg veldig skjevt i forhold til dronen, vil det kunne innebære at deler av objektet var nærmere dronen enn det avstandsberegningen ga oss. En annen løsning som vi ønsket å holde åpen på dette stadiet var at dronen skulle kunne bestemme hvilken vei det skulle svinge avhengig av hvilken vei objektet sto i forhold til dronen.

Et tenkt scenario her var at dronen kom kjørende med en vinkel mot en skuteside, der den eneste logiske retningen å svinge unna ville være den veien objektet ikke vinklet seg mot.



Figur 7: Illustrasjonsbilde av hvordan vinkel til objekt kunne bidratt til å turne rett vei

Da vi enda ikke hadde løst hvordan vi skulle ha styringssystemet, og at vi anslå viktigheten av å vite vinklene som høy bestemte vi oss for at vi ønsket å implementere det. Grunnen til at vi trenger fire lasere er best forklart med å se på bildene under:

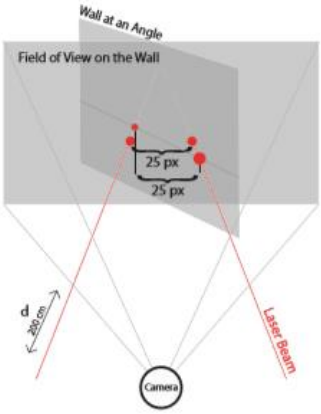


Figure 6.3: Laser geometry for Range Finder with two parallel lasers.

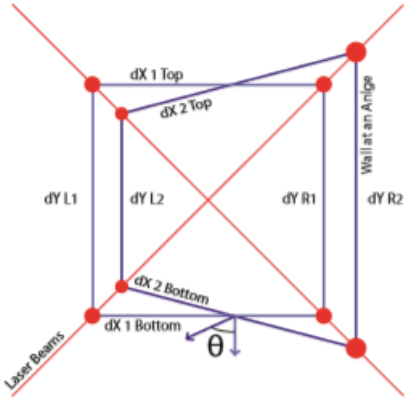
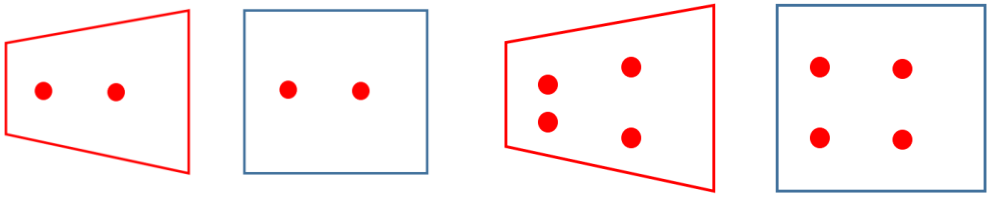


Figure 6.4: Proposed setup with four parallel lasers.

Figur 8: Utsnitt som illustrerer hvorfor vi trenger fire lasere for vinkler(Henriksen, 2016), Kilde:https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2501083



Figur 9: De røde prikkene er laserpunkter mens firkanten er et tenkt objekt

De røde prikkene illustrerer lasere og firkantene illustrerer objekter. Når man bare har to lasere og ser på et objekt med en vinkel vil man ha problemer med å bruke laseravstanden de to imellom til å bedømme at det er en vinkel. Mennesker ser selvsagt objektet og evner da å se at det treffer sideveis, men for en maskin har ikke referansesystemet evnen til å skjønne dette. Ser man på bilde nummer to fra venstre ses det til sammenligning hvordan laserne antatt hadde stått hvis det samme objektet(på samme avstand) var vinkelrett på dronen. Her ser man at det ikke er noe forskjell for maskinen sin del, og to lasere holder ikke til å bedømme vinkel.

Ser man derimot på de to bildene til høyre er det en tydelig forskjell i hvordan laserne oppfattes på et bilde avhengig av om objektene står vinkelrett på dronen eller ikke. Dette kan man implementere i løsningen slik at maskinen også kan tolke når objektet har vinkel, samt hva vinkelen er. Det har allerede blitt nevnt at man bruker pixelavstanden til å bedømme avstand til objektet. Det er samme type formel som brukes for å bedømme vinkler, formelen for avstand var:

1

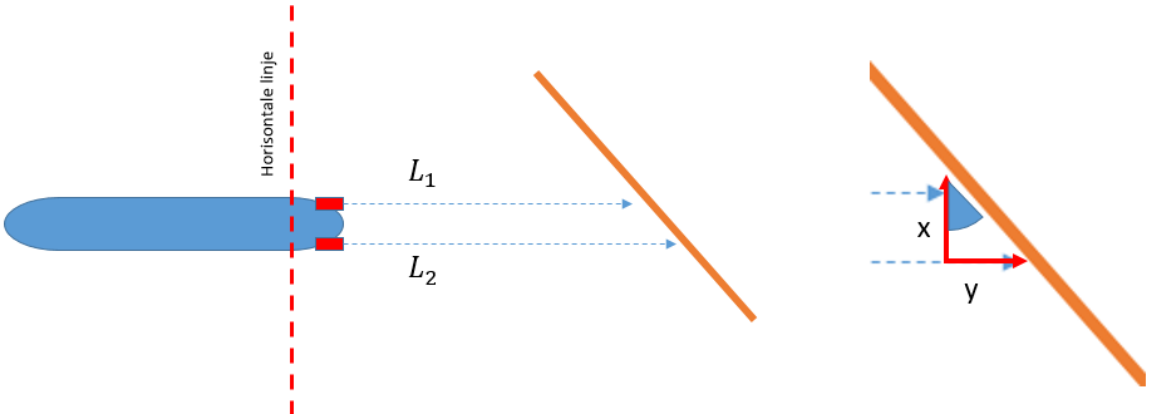
$$\Delta x = |x_1 - x_2| \qquad (1)$$

$$\Delta y = |y_1 - y_2| \qquad (2)$$

Hvor Δx er forskjell i pixelavstand mellom to laserpunkter i x-retning og Δy er lignende for y-retning. Ut av disse to kan man få avstanden til objektet som beskrevet tidligere.

$$f(\Delta x, \Delta y) = avstand\ til\ objekt \qquad (3)$$

Så man har nå en mulighet til å bedømme avstanden til noe som ligger imellom to laserpunkter, som ved 4 laserpunkter gir 4 avstander på et objekt. Ser man for seg et objekt som har en vinkel i forhold til dronens front:



Figur 10: Illustrasjon av dronen som ser på et objekt med en vinkel i forhold til den horisontale linjen

Ved å bruke pixelavstanden mellom de to babord laserne finner man lengden L_1og likt bruke de to styrbord for å finne L_2. På bildet til høyre er en illustrasjon av hvordan man kan finne vinkelen til objektet. Først må det defineres at vinkelen til et objekt er endring i vinkel i forhold til den horisontale linjen sett fra fronten til dronen. Tilbake til illustrasjonen til høyre ser man at der er mulig å finne vinkelen til objektet ved å bruke x og y:

$$Vinkel\ til\ objekt = \ tan^{-1}\left(\frac{y}{x}\right) \qquad (4)$$

Hvor

$$y = \ L_2 - L_1 \qquad (5)$$

Og x er den faktiske avstanden mellom laserne fra styrbord til babord montert på dronen. Man har dermed en ligning med bare kjente verdier, og kan slik finne vinkelen til objektet. Legg merke til at det ikke er absoluttverditegn for utregning av y. Dette er fordi der er interessant å vite hvilken retning objektet har vinkel på, slik at en teoretisk negativ y vil gi en negativ vinkel. En negativ vinkel vil igjen indikere at objektet er vinklet motsatt retning i forhold til illustrasjonen. Man kan implementere samme tankesett for vinkler i forhold til den vertikale linjen, med noen små justeringer.

2.2.1.1 Beregne faktisk avstand

Det er to viktige oppgaver som må løses: Først må man evne å få datamaskinen som brukes til å oppdage og skille ut laserpunktene, for så hente ut pixelkoordinatene. Det neste er å sette opp et objekt på ulike distanser fra kamera, og så plotte pixelavstanden slik at man kan konstruere en trendlinje. Man trenger dermed et bildebehandlingsprogram og et program som utfører utregningene på maskinen. Det benyttes OpenCV til bildebehandling og Python

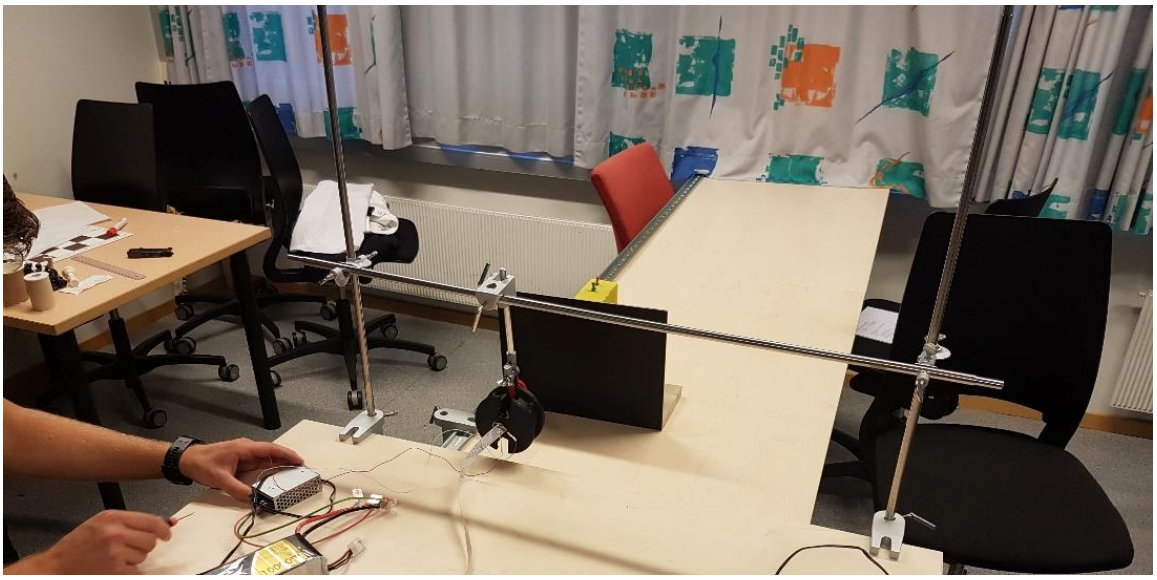
som hovedprogram. Python har som beskrevet i 1.4 en fordel i å være ekstremt godt dokumentert på nettet, samt ha flere relevante biblioteker som enkelt kan importeres til bruk. Et av disse bibliotekene var nettopp OpenCV.

Videre skal det brukes ettkortsmaskiner for å behandle dataen. Problemet med ettkortsmaskiner er mangel på prosessorkraft, noe som begrenser hastigheten. Dette bød på en del utfordringer og vurderinger som måtte tas for å lande på en god løsning. Diskusjonen og forklaringen er lagt under beskrivelse av systemet senere, da det passet bedre å ta det i samhandling med en mer detaljert beskrivelse av software.

Det er nevnt tidligere at det brukes statistikk for å bedømme avstand. Metoden er ganske simpel, da man måler pixelavstand mellom laserne fra 0cm til 170cm med 10cm intervaller. Disse målingene brukes til å lage en trendlinje (fra regresjon i matematikken), slik at når programmet har en pixelavstand Δx skal den kunne skrives inn i en funksjon og gi ut en avstand.

$$f(\Delta x, \Delta y) = avstand\ til\ objekt \qquad (6)$$

Fysisk ble dette gjort med å montere en metallskinne langs to bord, med en svart plate som fungerer som objekt. Alle tilnærmingene baserer seg på avgrensningen som er gjort med at objekter er tilnærmet flate. Racket med kamera og laser monteres så vinkelrett på metallskinnen slik at man best mulig sikrer gode målinger. Videre benyttes linjal sammen med en standard kommersiell avstandsmåler til å sette avstandene mellom rack og objekt. Sammensatt har man nå en måler på faktisk avstand sammen med pixelavstand oppgitt av datamaskinen.



Figur 11: Festet Rack vinkelrett på en svart plate. Platen flytted med fastsatte avstander

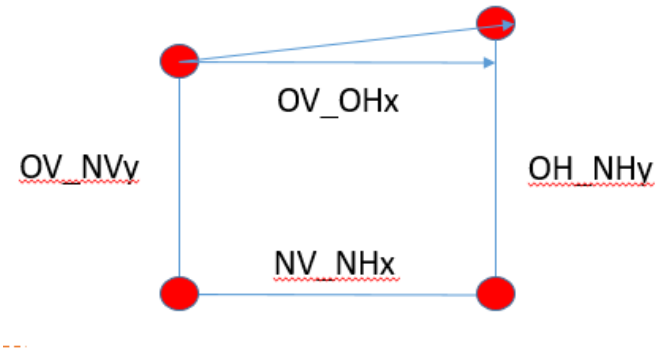
Disse verdiene plottes inn i Excel. Når platen står på for eksempel 20cm avstand til laserene, måler man pixelavstanden Python gir og plotter den inn. Siden man har 4 ulike lasere må samtlige av dem måles. Her måtte det vurderes om det var mest hensiktsmessig å bruke den kombinerte lengden av x og y, eller bare én av dem om gangen. Lengden mellom to punkter er:

$$lengde = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \qquad (7)$$

Hvor x og y er hhv pixelkoordinatene til to laserpunkter. Mens endringen i pixelavstand for x og y separat er:

$$\Delta x = |x_1 - x_2| \ og \ \Delta y = |y_1 - y_2| \qquad (8)$$

Ideelt sett skulle laserne være helt vinkelrette på hverandre, noe som igjen gjør at avstanden mellom dem til enhver tid er nøyaktig. Men praktisk sett viser erfaringen at det er utfordringer både med å få nøyaktigheten god nok på laserne, samtidig som at Python baserer pixelkoordinatene på midtpunktet til laserpunktet. Dette punktet kan av mange varierende forhold som sikt, lys osv ha en variasjon i nøyaktighet som gjør at pixelkoordinatene maskinen leser ut ikke alltid er vinkelrette. Velger man å bruke lengden mellom to pixelkoordinater vil feilen kunne bli mye tydeligere og betydningsfull enn hvis vi kun benytter oss av Δx eller Δy.



Figur 12: Illustrasjon av hvordan avstanden kan måles enten av lengde eller kun av endring i x

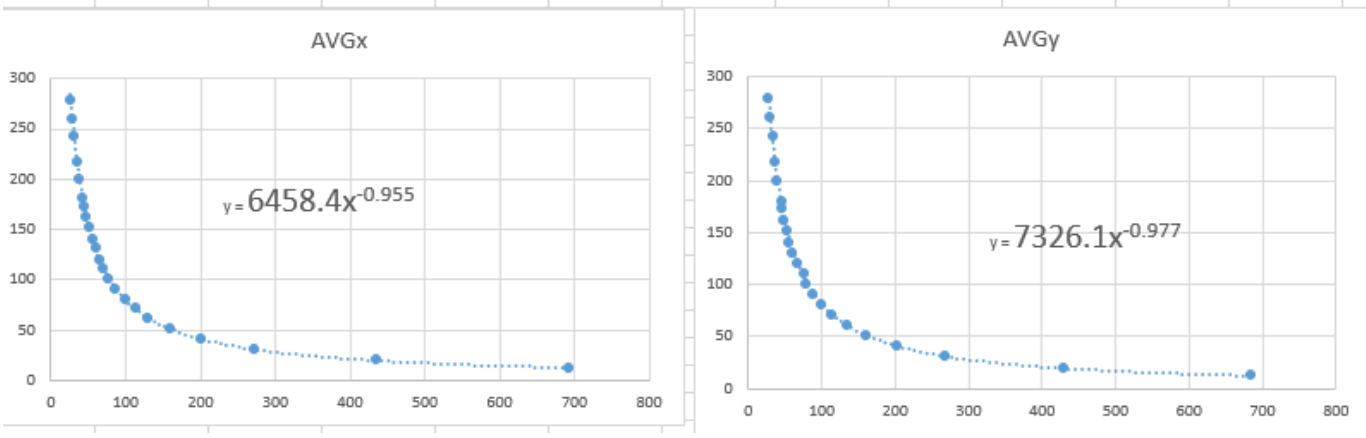
Men ser man på avstanden mellom de to laserne på høyre side vil man fortsatt ha problemet med feil lengde uansett om det kun benyttes Δx . For avstandsberegning løses dette ved at den midterste størrelsen av pixelavstandene benyttes, eller medianen. Men løsningen vil aldri kunne gi en helgardering mot unøyaktighet. Derfor er det viktig at vi tar for oss hvilket krav vi har stilt til dronens egenskaper i forhold til objektdeteksjon. For det første er kravet at dronen skal *unngå* objekter, noe som betyr at et frafall i nøyaktighet på et par cm ikke vil ha innvirkende effekt. For det andre vil slike unøyaktigheter gi størst effekt nært objekter.

Altså, det er ved små avstander at eventuelle små avvik i nøyaktighet på monteringen vil vise seg å gi utslag. Med andre ord vil eventuelle feil i avstandsberegningen ikke bare være på veldig få cm, men også skje på slike avstander at det ikke forstyrrer dronens vurdering av objektet. Skulle objektet være på 100cm istedenfor antatte 90cm vil det bare innebære at dronen styrer rundt objektet litt tidligere enn forventet.

De 4 avstandene defineres som:

- Δx OppeVenstre til OppeHøyre: OV_OHx
- Δy OppeVenstre til NedreVenstre: OV_NVy
- Δy OppeHøyre til NedreHøyre: OV_NHy
- Δx NedreVenstre til NedreHøyre: NV_NHx

OV_OHx	OV_NVy	OH_NHy	NV_NHx		AVGx	AVGy	avstand(cm)
694	690	682	694		694	686	12.5
445	430	429	426		435.5	429.5	20
271	267	272	274		272.5	269.5	30.2
202	205	201	200		201	203	40.3
161	162	160.5	160		160.5	161.25	50.9
124	135	134	134		129	134.5	60.5
113	114.5	112.7	112.6		112.8	113.6	71.3
100	101	99.5	99.5		99.75	100.25	80.7
87	89.5	88	87		87	88.75	90.9
78	80.5	78.7	78.2		78.1	79.6	100.7
71	72.5	82	70.5		70.75	77.25	110.3
65.2	67.7	66	65.5		65.35	66.85	120.2
60	61.8	60.8	60		60	61.3	130.6
56	57.4	55.5	56		56	56.45	140.2
51	53.5	52	51		51	52.75	151.4
48	50	49.2	47.2		47.6	49.6	162
45.9	45.5	46	44		44.95	45.75	172.2
43.2	45.4	44.7	42.8		43	45.05	180.2
37.5	40.2	38.4	37.6		37.55	39.3	199.9
36.2	36.1	36.1	35.4		35.8	36.1	217.3
32	33.4	34	30.5		31.25	33.7	241.6
29	30.3	31	28.4		28.7	30.65	259.9
26.5	28.5	28.6	26.3		26.4	28.55	278.3



Figur 13: Tabellen med de ulike plottene for ulike avstander, med trendlinje for begge tilfellene

Her er de fire pixelavstandene samt snittet av de to Δx og Δy. Videre ser man at unøyaktighetene imellom de to parene er større jo nærmere kameraet er objektet, og et gjennomsnitt vil dermed bidra til å finne en mellomverdi mot hvilken pixelavstand man faktisk bør forvente. Punktene er heller ikke kvadratiske når man ser på gjennomsnittene. Dette antas å skylde en kombinasjon av vanskelig montering og at selve monteret er 3D-printet. Materialet er plastikk, og når denne blir varm beveger den seg i vilkårlig retning. Dette vil på

sikt skape unøyaktighet i retningen på laserene, som vil gi feil måleverdi siden faktisk avstand mellom laserprikkene på objektet ikke passer med trendlinjen som ligger i programmet.

Ser man på grafen til tabellen, følger den en negativ eksponentiell kurve som går mot 0. Altså er det snakk om eksponentiell regresjon. Ved å plotte dette i Excel kan man bruke en funksjon som lager en trendlinje. Den beste løsningen var en potens-funksjon som fulgte kurven veldig bra og hadde høy R^2. Funksjonene vi fikk ut ser du i grafen, og avhenger av hvilken pixelavstand man bruker:

$$f(\Delta x) = 6458.4 * x^{-0.955} \qquad (9)$$

$$f(\Delta y) = 7326.1 * x^{-0.977} \qquad (10)$$

2.2.1.2 Utfordringer med å implementere vinkel på objekt

Det er nevnt at man hadde problemer med å få nøyaktig posisjon og retning på laserne. Som en konsekvens av dette vil disse unøyaktighetene påvirke vinkelutregningen. Eksempelvis befinner dronen seg 240cm fra et objekt, som er en distanse hvor man kan tenke oss at dronen skal begynne å vike unna. Man kunne nå brukt faktiske data til å sammenligne, men bruker målingene da de allerede viser avvik imellom de sammenhengende pixelavstandene. Følger man tabellen i figur 13 har Δx en mulig feildifferanse på 1-2 pixler, mens Δy har en på 0.5. I seg selv er dette ikke veldig mye forskjell når man snakker om et bilde i størrelsesorden 1280x720 pixler.

Hvis man derimot regner ut beregnet avstand basert på målingene fra tidligere, som også igjen kan variere:

$$f_x(32) = 235.88 \quad f_x(30.5) = 246.955$$

$$f_y(33.4) = 237.778 \quad f_y(34) = 233.678$$

Koden baserer seg på å velge ut laveste verdi og dermed bedømme hvilken retning vinkelen går i sammen med høyeste verdi, her 233.678 som kommer fra OH_NHy og 237.778 som høyeste fra OV_NVy. Maskinen beregner da en rotasjon om den vertikale aksen, og bruker formelen for å finne vinkel:

$$f(233.678, 237.778) = tan^{-1}\left(\frac{237.778 - 233.778}{6.5}\right) = 32.24°$$

Dette hadde ikke vært en utfordring hvis problemet ikke var at objektet står vinkelrett på kamera, og har en forventet vinkel på 0! Ved å kjøre samme test med objektet på 20cm avstand:

$$f_x(445) = 19.0959 \quad f_x(426) = 19.9085$$

$$f_y(430) = 19.5873 \quad f_y(429) = 19.6319$$

På lik linje med over velges den laveste verdi til å bedømme hvilken vinkelretning man har, som er OV_OHx. Hvis den settes sammen med den høyeste verdien, som blir 19.9085 fra NV_NHx:

$$f(19.0959{,}19.9085) = tan^{-1}\left(\frac{19.9085 - 19.0959}{6.5}\right) = \ 7.12°$$

Feilen i vinkel er betydelig lavere enn ved avstand på 240cm, men er fortsatt 7 grader av! Problemet oppstår altså for alle avstander, men vil bli enda mer tydelig ved større. Det er nevnt at styringssystemet skal vike unna objekter, som skal skje på god avstand til å unngå kollisjon. I motsetning til avstandsberegning der man godtar en feilmargin på noen centimeter, vil feil i vinkler ha en direkte påvirkning på funksjonaliteten til styringssystemet hvis man skal ta det i bruk.

Det har allerede blitt diskutert om at unøyaktighet i montering kanskje er den største synderen, men hva er egentlig unøyaktighet i denne skalaen? Laserholderen er en 3D-printet holder som initialt ble antatt at ville ha ekstremt god presisjon, noe som også til dels stemmer da den er opplyst å være 0,3 mm. Dette vil i de fleste tilfeller være en meget god nøyaktighet, men for presisjon i montering av lasere bød det på problemer. Hvis man har en laser montert med 0,3 mm feil vil feilen være 0,3 mm per mm vi lyser, noe som gir en feilmargin på 0,03 grader.

Dette tilsvarer et maksimalt avvik på 3 cm per meter! Siden systemet er tenkt til avstander opp til 3 meter er ikke dette nøyaktig nok. Etter mye testing av systemet med kompensasjoner gjort i software havnet tolererbar feilmargin på omtrent 1mm på 1,7m avstand. For å oppnå dette trengs det stor presisjon i selve innfestningen av laseren. Løsningen på denne presisjonen var å utvide hullene til laseren i holderen og lime de fast med hurtiglim. Dette er nok ikke den beste måten for innfesting med tanke på service og bytte av deler til senere, men var helt avgjørende i forhold til nøyaktighet på laseren. Det hensiktsmessige hadde vært å støpt laserfestet i hardplast for at delen skal bli så riktig som mulig. Dette var ikke gjennomførbart da støpning er kostbart og tidkrevende.

Ved selve installasjonen kommer store deler av unøyaktigheten. Denne unøyaktigheten påvirker kontinuiteten i målingene og dermed også troverdigheten til sensoren. Over tid har

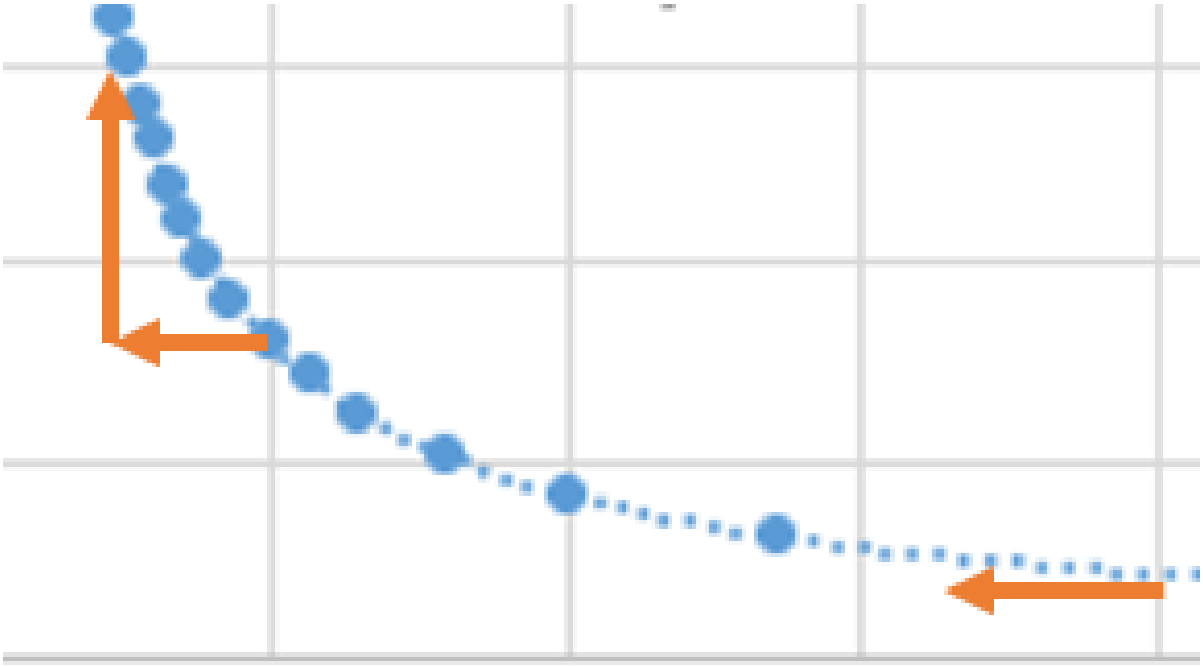
man også oppdaget at varmen fra laserne deformerer plasten som holder dem på plass. En annen feilkilde er også plasten sin evne til å absorbere fuktighet fra omgivelsene, som igjen skaper unøyaktighet i målingene.

En tredje kilde til unøyaktighet ligger i algoritmen for beregning av avstand. Det er brukt de to potensfunksjonene:

$$f(\Delta x) = 6458.4 * x^{-0.955} \quad (9)$$

$$f(\Delta y) = 7326.1 * x^{-0.977} \quad (10)$$

Når pixelavstanden x blir mindre har man en kraftigere økning i beregnet avstand, noe kan ses igjen på grafen. Det skal en kraftigere endring i pixelavstand til for å endre beregnet avstand når man er nært objektet enn når man er lenger borte. Dermed vil en feil på bare 0,3 mm i nøyaktighet kunne skape feil i beregnet avstand som er større enn det man kunne forvente med så nøyaktig montering av lasere. Selv om feilen i beregnet avstand bare ender opp på +- 5 cm er dette nok til å skape store feil i vinkelberegningen.



Figur 14: Man ser at den deriverte, eller endringen imellom pixelavstand og beregnet avstand er ekstremt ulik

2.2.2 Sammendrag

Dronen har nå et system som klarer å oppdage relativt flate objekter og beregne avstanden til dem med god nøyaktighet. Dette tilfredsstiller kravet om objektdeteksjon.

Vi vet nå at beregning av vinkel til objektet gir store feil ved lengre avstander. Selv om det ved mindre avstander har vist seg at feilmarginen vil være mye mindre, eksisterer det en feil her også som ikke forblir helt ubetydelig. Ser man tilbake på hvilke måter som ble forespeilet

å implementere vinkeldeteksjonen på, vil det være nødvendig å beregne korrekte vinkler på større avstander. Da dette ikke ville være mulig uten å forbedre nøyaktigheten fysisk, skrotes vinkelberegningen helt.

Men til tross for at man ikke kom i mål, vil systemet fungere hvis man hadde brukt mer penger på å montere laserne enda mer nøyaktig. Dette ville blitt kostbart og tatt mye av tiden, og ble derfor ikke prioritert. Men alt i alt ville vinkelberegningen fungert, noe som ville blitt en viktig ressurs i styringssystemet. Hadde man videreutviklet dronen med mer penger hadde beregningen av vinkler blitt tatt i bruk.

Det som er viktig for videre design og konstruksjon er at man nå ikke har tilgang til solide vinkeldata på objektet, noe som krever en endring i tilnærmingen til styringssystemet.

2.3 Posisjonsmåler

Basert på systemkrav må dronen ha et selvstendig system for å ha kontroll over posisjonen sin. Dette er et hardt krav å stille med bakgrunn i hvilke midler som er tilgjengelig. Av andre typer fartøy og kjøretøy som oppholder seg over vann er sensoren som nyttes for posisjonering, GPS. Utfordringen er at GPS sendes via signal, og lys bøyes og reflekteres i vannoverflaten slik at signalet er umulig å nytte seg av etter bare et par meter ned i vann for bruk ved posisjonsmåling. Man må dermed finne andre løsninger for hvordan vi kan hente ut posisjonen.

2.3.1 Løsning ved bruk av akselerometer

I sensorpakken er det et akselerometer som henter ut treghetens akselerasjon i alle tre akser. Første vurdering var å prøve å integrere akselerasjonen to ganger, som ville resultert i at man hadde fått ut posisjonen. Man har:

$$v(t) = \int a(t)dt \quad og\ s(t) = \int v(t)dt \qquad (11)$$

Hvor a er akselerasjon, v er fart og s er posisjon. Dette gir:

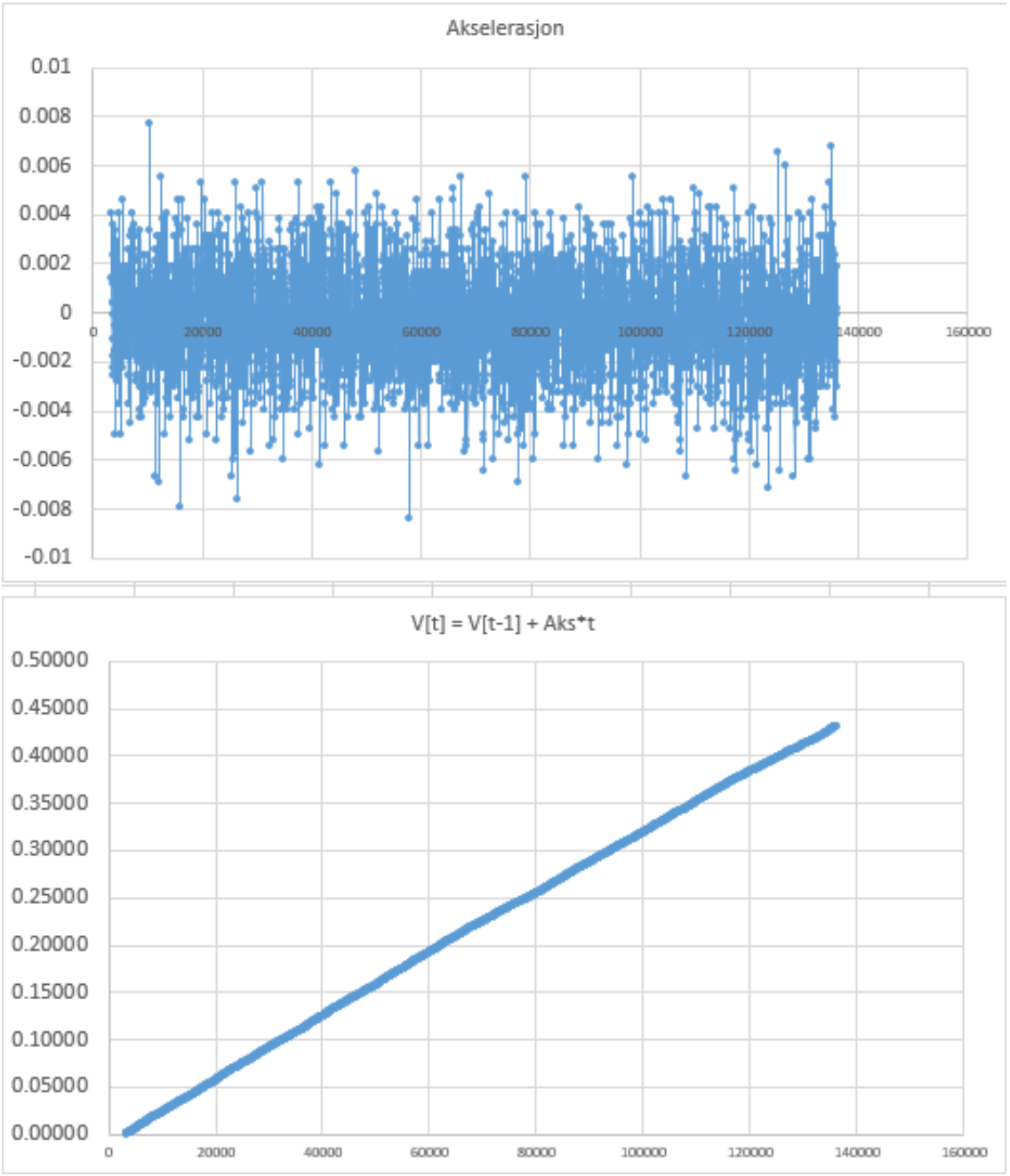
$$s(t) = \iint a(t) \qquad (12)$$

Eller i diskret form:

$$v(t) = v(t-1) + a * t \qquad (13)$$

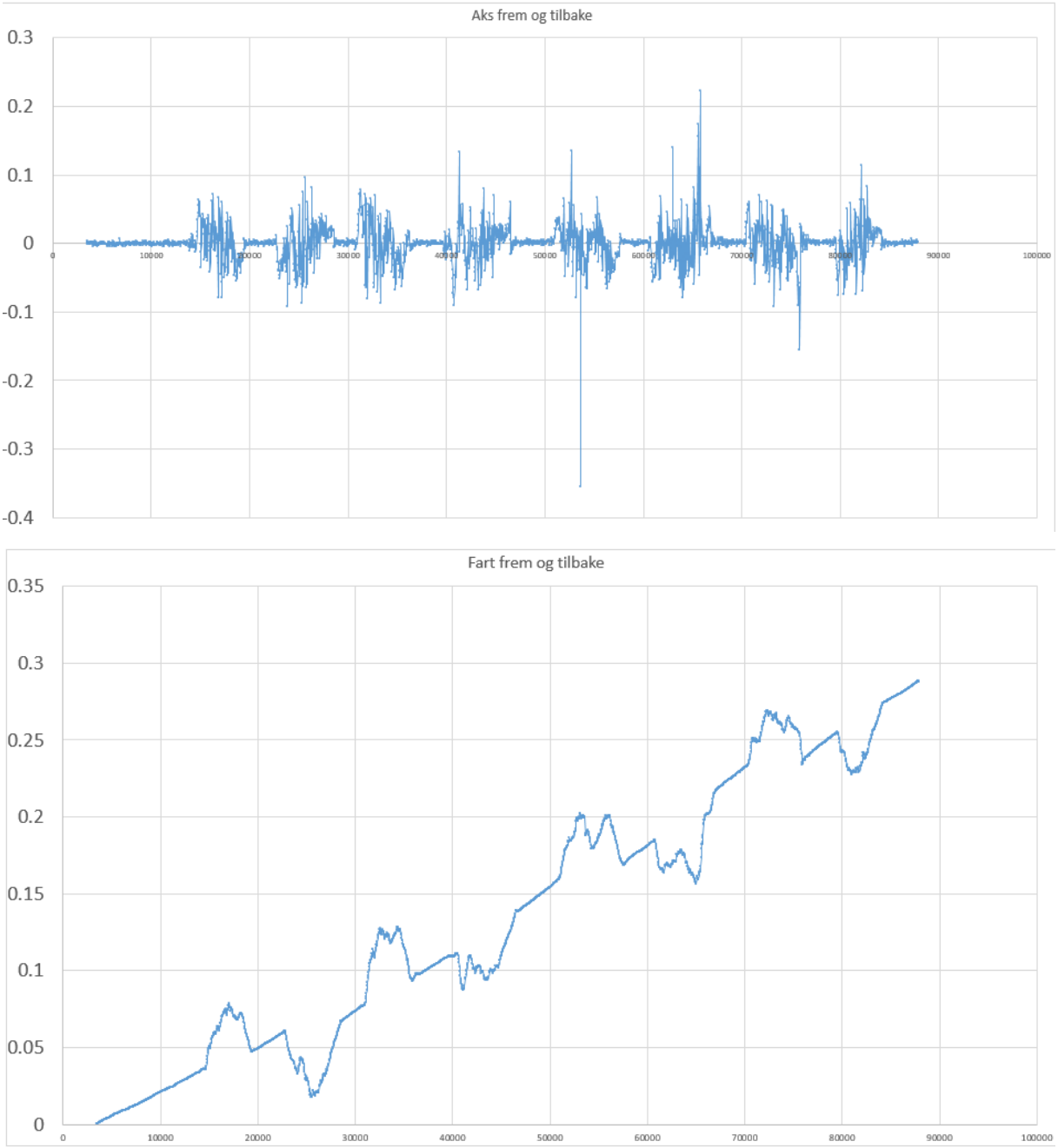
$$s(t) = s(t-1) + v * t \qquad (14)$$

Som i teorien betyr at man kan integrere akselerasjonsdataen og få ut posisjonen, men så enkelt er det ikke. For det første vil akselerometeret oppleve en konstant akselerasjon i z-aksen som følge av gravitasjonen. Hvis man roterer på akselerometeret vil også gravitasjonen fordele seg over aksene, det er akkurat slik man henter ut rotasjonen til dronen senere. Men her må man altså ha en løsning for å fjerne gravitasjonen fra dataen. Hvis det kompenseres basert på hvor mye rotasjon man har om x- og y-aksen kan man ha en dynamisk offset som fjerner gravitasjonen. Men man vil likevel ha en feilkilde i at eventuelle unøyaktigheter i målingen av rotasjonen pitch og roll vil gi feil i kompenseringen. Dette er første feilkilde i beregning av posisjon. For det andre vil billige akselerometre være utsatt for støy, som igjen vil akkumulere til feil i datastrømmen. Støyen er tilfeldig og blir dermed ekstremt vanskelig å filtrere skikkelig bort. Til sammenligning har man samlet data fra akselerometeret der det ligger helt i ro over tid og beregnet farten:



Figur 15: Data fra akselerometer som ligger i ro, under ser vi farten vi har regnet ved å integrere akselerasjonen

Vi har her også integrert akselerasjonen for å beregne farten. Vi ser fra figuren at akselerometeret gir ut ganske mye tilfeldig støy, og at når denne støyen blir integrert over tid vil den gi utslag som såkalt drift. Maskinen vil altså tro at den beveger seg og får dermed en økende fart, mens den faktisk er i ro. Ser vi på lignende data bare når vi beveger akselerometeret på en linje frem og tilbake, ser vi at akselerometeret gir utslag i forhold til støyen.



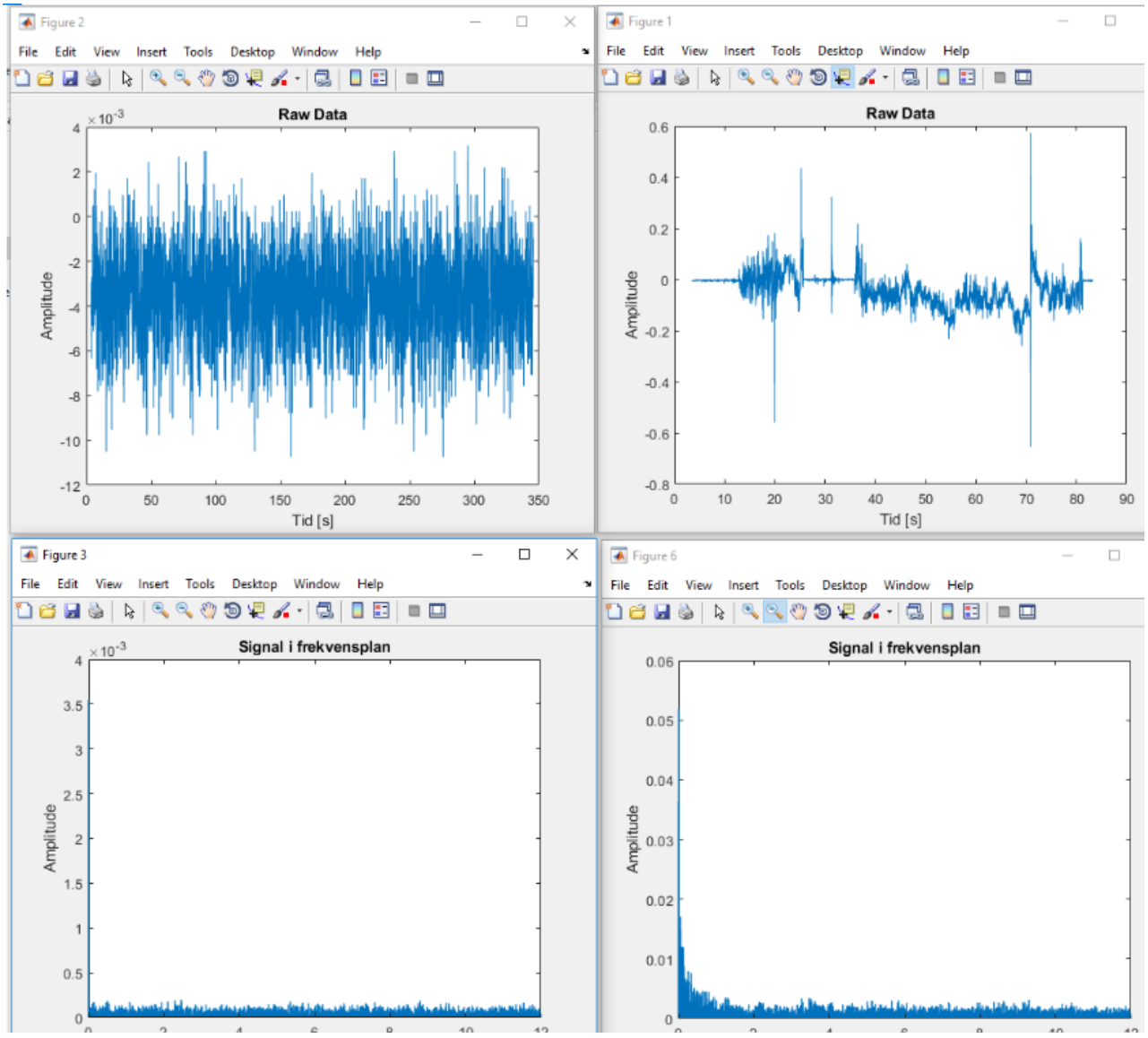
Figur 16: Plotting av akselerasjon og fart når akselerometeret beveger seg frem og tilbake

Det er tydelig i fartsgrafen at det registreres at man beveger seg frem og tilbake, og at den er i et tydelig skille til støyen. Med andre ord ligger den korrekte dataen tilgjengelig for oss til å få rett posisjon, problemet ligger i at man ikke har noen gode løsningen på å skille den ut.

Det ble vurdert om man skulle legge på et tak som akselerometer-dataen måtte overstige for at man skulle telle den som gyldig. Dette ville fungert hvis den hadde stått i ro, men når man har en faktisk bevegelse på akselerometeret vet man ikke hvor mye av dataen som er støy eller hva som er faktisk bevegelse. Dette skyldes at støyen er tilfeldig og ikke følger et

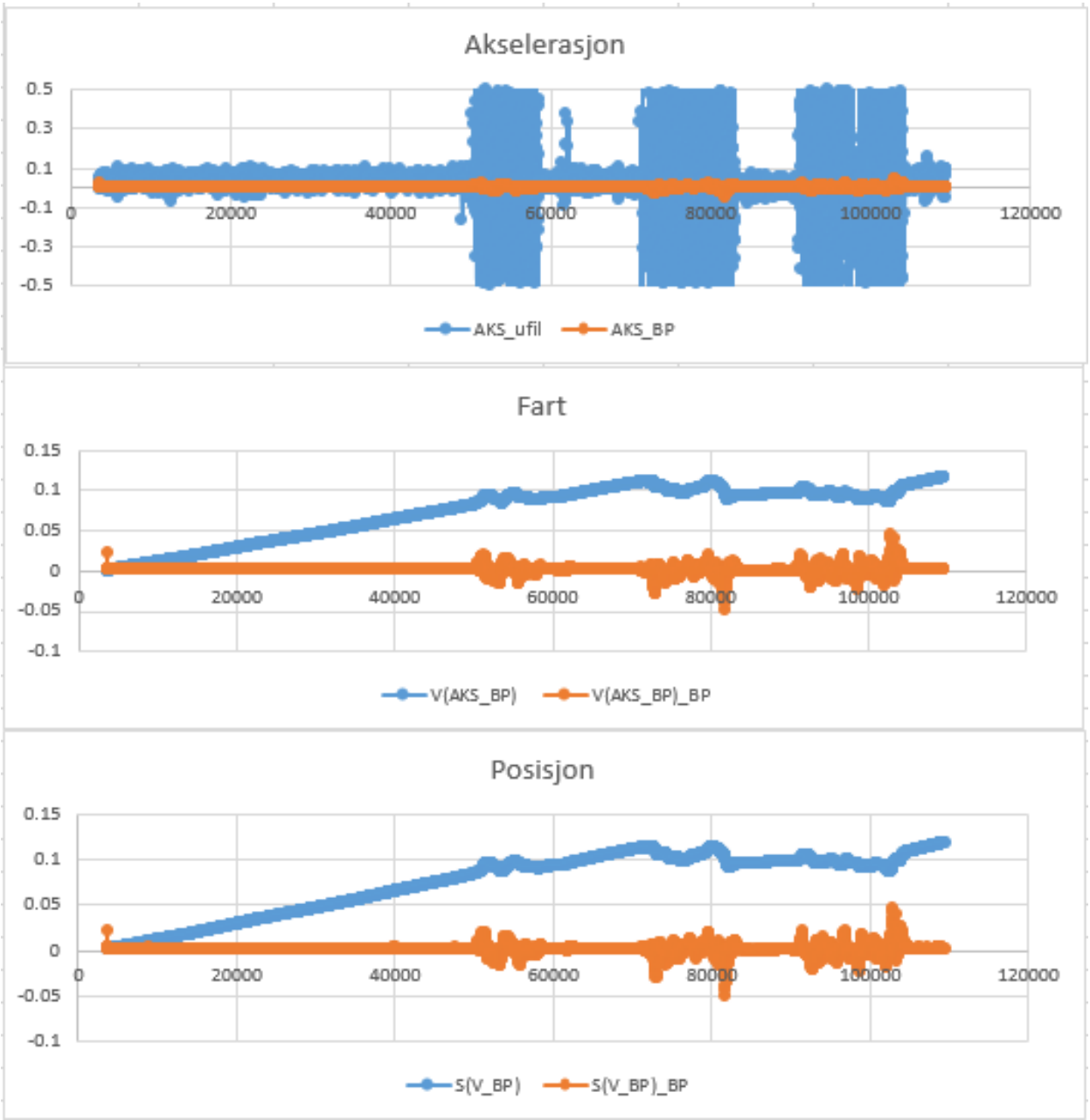
mønster, som gjør at det blir vanskelig. XIO Technologies har brukt en lik tilnærming til å måle en fots bevegelse når man går. Her er feilen såpass liten at man kan godta dataen, men baserer seg igjen på at man oppnår en ro-tilstand på sensoren ved flere anledninger. For en drone som beveger seg i vann vil man aldri oppnå en ro-tilstand der man kan resette akselerometeret og farten.

Man kan derimot prøve å løse utfordringen med å logge støyen både i bevegelse og ved å ligge i ro, for så å bedømme et passende filte. Tanken er at man vil miste kun små deler av eventuell korrekt data eller kun få liten drift ved å benytte filter. Hvis unøyaktigheten er liten nok, vil det tilfredsstille kravet. Man kan se nærmere på de to plottene våre i Matlab ved hjelp av vår veileder.



Figur 17: Utsnitt av plottet data satt inn i Matlab av Alexander Sauter, her ingen bevegelse til venstre og frem og tilbake til høyre. Legg merke til at lavere frekvenser blir større når man har bevegelse

Støyen er veldig høyfrekvent, samtidig har man noen frekvenser nært null som er støy, noe som gjør det ideelt å teste ut et båndpassfilter på akselerasjonen. Samtidig vil det være noe støy som ikke filtreres bort og som blir med når man integrerer akselerasjonen for å få farten. Det blir derfor nødvendig å prøve å filtrere farten og posisjonen også. Under er plotting av akselerasjon, fart og posisjon med og uten båndpassfilter:



Figur 18: Plotting av data med bruk av filter. Man ser tydelig at filteret får vekk mesteparten av driften for fart og posisjon

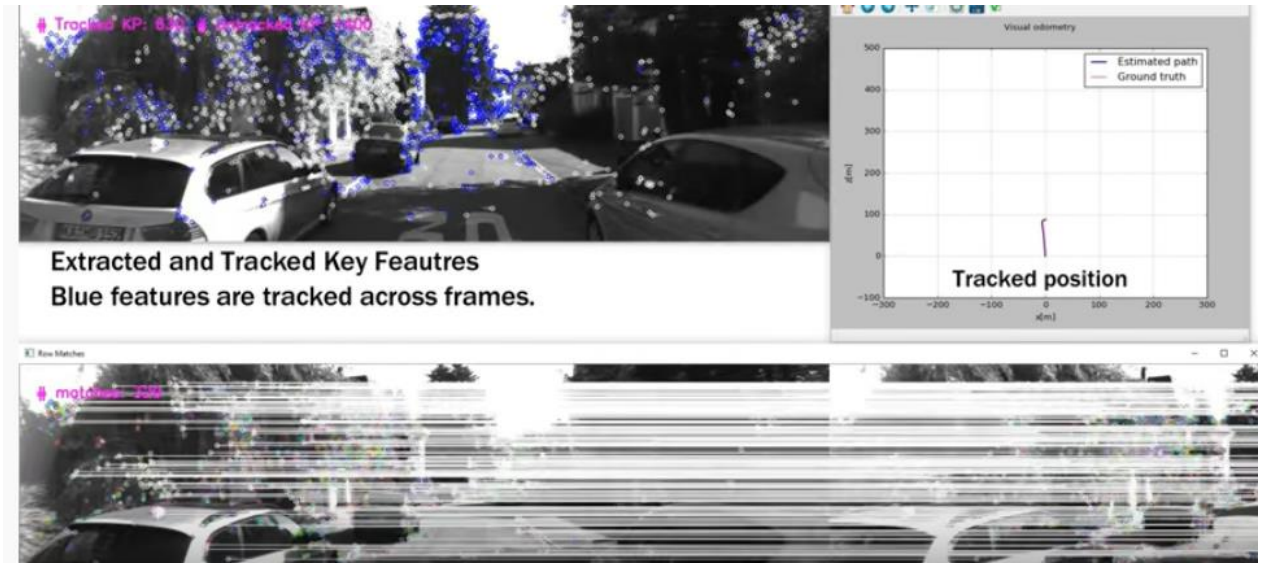
Resultatet kan se veldig lovende ut ved første øyekast, og dette var kanskje noe man kunne benyttet oss av. Man har tydelige utslag på grafen når akselerometeret beveges frem og

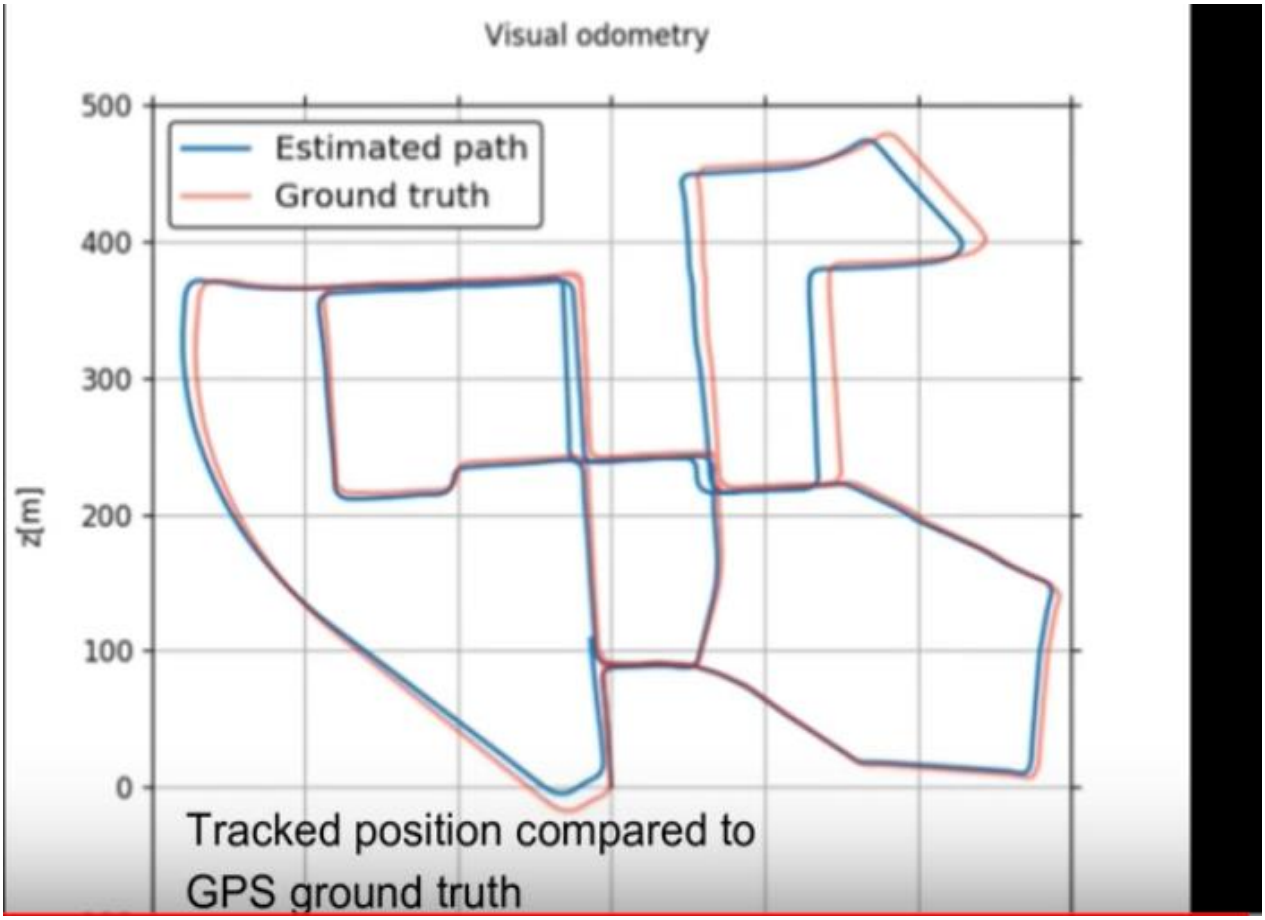
tilbake. Utfordringen ligger i at man her er nødt til å filtrere posisjonen også, noe som gjør at den alltid går tilbake til 0 i verdi. Dette stemmer sjeldent da man forventer en kontinuerlig fremdrift på posisjonen, og en tilbakegang til 0 indikerer at filtreringen mister mye data. Videre ser man og at det er et totalutslag som er mindre enn 0.05 meter eller 5 cm. Dette stemmer dårlig da man har beveget akselerometeret over 40cm! man ser også at posisjonen på lik linje med fart og akselerasjon veksler mellom positive og negative verdier hele perioden den beveger seg. Dette stemmer ikke da man skulle hatt positiv posisjonsdata som så skulle gått ned til 0 igjen først når man beveger seg bakover.

Summert opp vil selv en god filtrering miste for mye data til at man kan bruke dette som en troverdig posisjonsmåler, og tilfredsstiller ikke kravet. Løsningen blir skrotet, og man må se videre mot andre løsninger.

2.3.2 Løsning ved bruk av bildebehandling

Det har allerede blitt brukt kamera og bildebehandling til å gjenkjenne spesifikke ting med bilder, der man blant annet filtrerer ut visse deler av bildet som ikke behøves. Hvis bildebehandlingsprogrammet klarer gjenkjenne spesifikke deler av bilder, vil det da og klare å merke om et objekt har flyttet seg fra et bilde til et annet? Her nyttes Feature Matching(FM) som er en metode for å gjenkjenne spesifikke områder imellom to bilder. Denne hadde blitt brukt i et eksperiment der et kamera var montert foran på en bil som kjører, der dataen hadde blitt sammenlignet med data fra GPS:



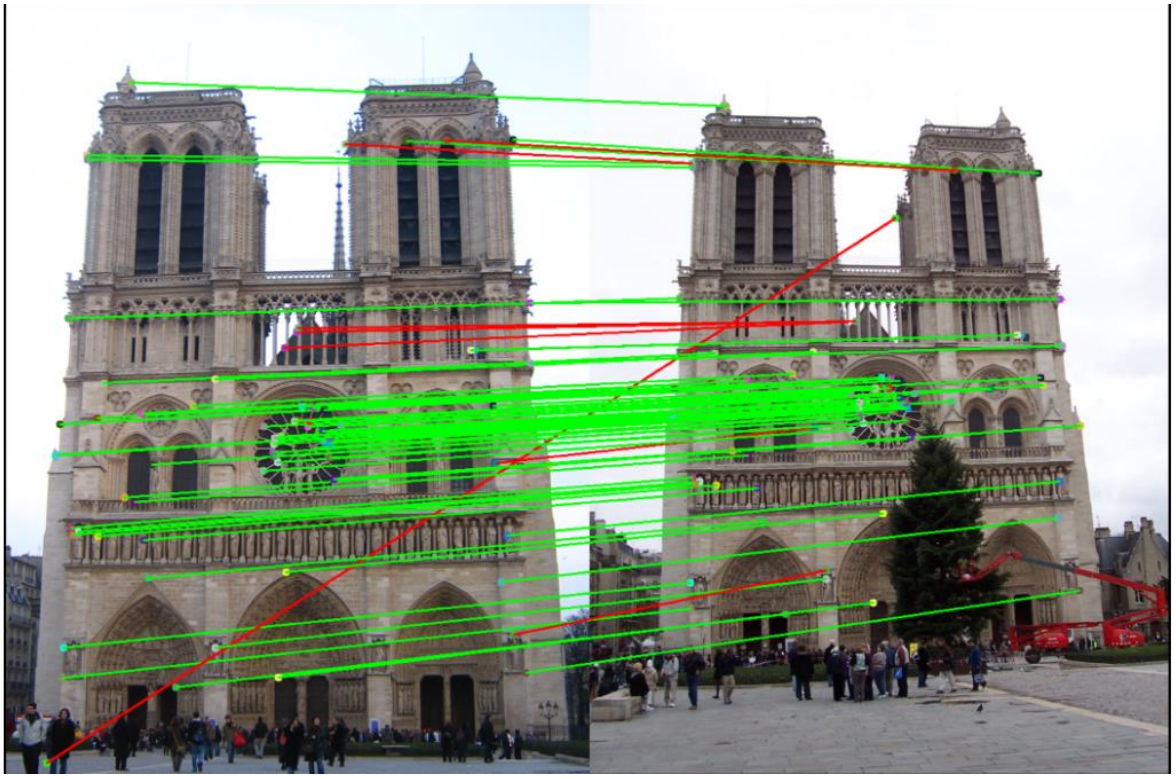


Figur 19: Utsnitt fra video der man måler posisjonen basert på Feature matching, for så å sammenligne med GPS
Kilde: https://www.youtube.com/watch?v=RJf809CVfWY&t=36s

Hvis denne metoden kunne brukes til å kartlegge endring i pixelavstand kan man lage en lignende algoritme som for Avstandsmåleren i 2.2 til å kartlegge posisjonsendring.

Først må man altså forstå konseptet med FM. Det fungerer ved at man leser ut informasjon om spesifikke punkter i et bilde som man lagrer, såkalte features eller trekk på norsk. Denne

informasjonen kan variere ut ifra typen FM man bruker, men lagrer for eksempel informasjon om konturene og fargen av området rundt punktene til mer kompleks informasjon. Så tar man et nytt bilde og henter ut lignende informasjon fra dette, for så å sammenligne informasjonen. De punktene som har likest informasjon, eller minst ulikhet og som også er nærmest hverandre lagres videre. Man har nå effektivt sammenlignet punkter i to bilder, og så sagt hvilke som er likest hverandre. I samme informasjonspakke ligger også posisjonen til de to punktene i pixelkoordinater lagret, noe som betyr at man kan definere avstandsendringen mellom de to punktene. Vi definerer ordet pixelendring som endringen i pixler målt i x og y.

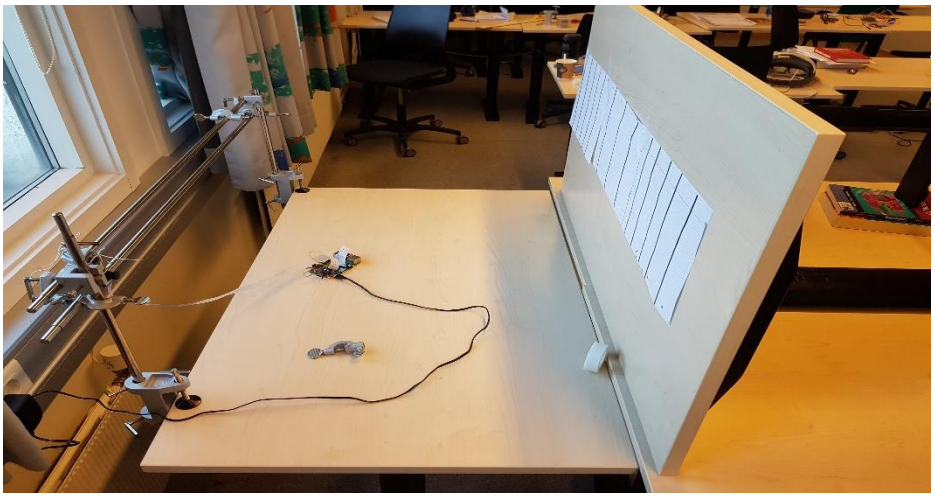


Figur 20: Eksempel på matching av like punkter mellom to bilder. Bilde hentet fra: https://medium.com/@deepanshut041/introduction-to-feature-detection-and-matching-65e27179885d

Nå vil ikke alle punktene man har plukket opp være like, og det vil være viktig å sette en grense på hvor mange av punkt-matchene som brukes. Det som er veldig positivt er at i sammenligningen av punktene lagres det også informasjon om hvor like de er, slik at man kan velge å kun se nærmere på de som er mest like!

Det som er viktig å tenke på nå er at man snakker om pixelavstand, og at man vet at den endrer seg i størrelse avhengig av avstanden til det som ses på bildet. Det er også viktig å huske på at måten man skal benytte seg av dette rent praktisk er å montere et kamera på undersiden av dronen, som så skal måle posisjonsendring i forhold til bunnen. Det er altså ikke et objekt som beveger seg, men dronen som beveger seg i forhold til objektet.

Med bakgrunn i dette var det viktig å teste hvor bra FM vil fungere, og om det er godt nok for at vi oppnår kravet vårt. Testingen besto i å montere et kamera på et monter der man kunne bevege kamera langs én akse i forhold til et fast objekt, for så å bevege kameraet frem og tilbake imellom en fast avstand. Kameraets bevegelseslengde er målt til 62.7 cm, og avstanden til objektet varierer i testen fra 63.5cm til 4.3m. Kriteriet for testen var at når kameraet returnerte til sin utgangsposisjon etter å ha gått frem og tilbake, så skulle den totale pixelendringen være ideelt sett 0. I henhold til kravet ville man godta at endringen var noe unøyaktig, men det var også viktig at vi målte denne unøyaktigheten slik at man kunne anslå en total unøyaktighet for posisjonsmålingen.



Figur 21: Test av FM. Til venstre ser man stenger som har et kamera festet til seg som kun beveger seg frem og tilbake langs én akse. Papiret på «veggen» er til hjelp for kamera

Det velges tre måter å måle pixelendring:

- Minste posisjonsendring
- Beste matchen
- Snittet av de tre

Dette for å spille på flere muligheter når man skulle teste om det var nøyaktige målinger til å tilfredsstille kravet. På lik linje med 2.2 måltes det for flere avstander til det faste objektet, ved å flytte objektet lenger bak. Det var også viktig for testen at man kuttet ned kamerabildet til å kun treffe selve objektet og ikke veggen som var lenger bak, da dette er en solid feilkilde. Etter å ha prøvd et par ganger ble bordet i figur 21 byttet ut og man tok i bruk en hel vegg som objekt/bunn for målingene.

Det man er interessert i å få ut av testen er to målinger. Den første er å se på hvor mye pixelendringen varierer for samme type målinger/avstander. Hvis variasjonen er liten, vitner det om at FM evner å ha en ganske god nøyaktighet i målingene sine når kamera beveger

seg. Er variasjonen derimot stor betyr det at nøyaktigheten er liten, og skaper stor usikkerhet og potensielle store feil i posisjonsmålingen. Det andre man vil vite er om målingene *fra* utgangsposisjon varierer mye i forhold til målingene *tilbake til* utgangsposisjon. Hvis variasjonen er liten betyr det at FM klarer å beregne endringer i forhold til et referansepunkt veldig bra, selv om forskyvningen og endringen kan variere i størrelse. Hvis variasjonen er stor betyr det at FM ikke klarer å beregne endring i forhold til et referansepunkt, og hvis det skulle bli implementert som posisjonsmåler ville man ikke hatt evne til å finne tilbake til startpunkt med stor sannsynlighet.

For at FM skal tilfredsstille kravet til løsning for posisjonsmåling, må begge målingene ha lav variasjon og dermed gi god nøyaktighet. Matematikkens velkjente standardavvik på 5% brukes som et tall som kan godtas. Samtidig er dette målinger i liten skala som er utført i et miljø der det finnes mange feilkilder. For eksempel vibrasjon under bevegelse av kamera og forskyvning av linse. Målingene er så plottet i Excel:

			Fra utgangsposisjon						
	avstand(m)	distanse(m	1	2	3	4	5	Snitt	Avvik i %
Minste	0.625	0.628	300.22	302.15	305.5	295.51	309.16	302.62	4.51
Snitt			511	482	518.51	643.66	485.63	505.05	32.01
Beste match			339.17	326.429	335.47	334.9	334.09	334.82	3.81
Minste	0.93	0.628	229.2	213.84	209.7	207.573	209.5	211.01	10.25
Snitt			137.23	155.3	203.77	331.8	439.6	230.29	131.30
Beste match			236.64	225.32	227.54	230.244	236.4	231.39	4.89
Minste	1.465	0.628	146.32	140.68	144.19	148.63	148.03	146.18	5.44
Snitt			256.77	256.37	222.33	292.63	258.2	257.11	27.34
Beste match			152.33	151.32	151	156.64	152.8	152.15	3.71
Minste	2.087	0.628	102.23	103.95	104.76	103.84	100.744	103.34	3.89
Snitt			203.69	158.62	183.94	137.15	231.44	182.08	51.78
Beste match			107.08	107.39	110.03	109.66	111.3	109.03	3.87
Minste	2.848	0.628	76.463	82.06	74.04	72.6	75.87	75.46	12.54
Snitt			137.85	142.62	131.7	165.35	133.94	138.14	24.36
Beste match			82.24	81.91	78.83	78.74	80.58	80.44	4.35
Minste	3.35	0.628	65.45	61.935	62.264	64.259	61.61	62.82	6.11
Snitt			114.82	109.06	108.047	112.679	112.1	111.28	6.09
Beste match			68.632	65.68	64.595	67.35	69.67	67.22	1.49
Minste	4.321	0.628	49.27	52.487	48.07	49.07	50.343	48.80	9.05122951
Snitt			90.8	90.32	91.73	89.799	92.96	86.40	3.65856481
Beste match			55.4	54.896	55	55	54.66	54.97	0.61851919

			Til utgangsposisjon						
	avstand(m)	distanse(m	1	2	3	4	5	Trim snitt	Avvik i %
Minste	0.625	0.628	315.62	306.58	299.251	321.86	313.88	312.026667	7.24585506
Snitt			484	595.06	511.37	609.61	566.34	557.59	22.527305
Beste match			334.31	330.24	330.93	333.88	333.87	332.893333	1.22261385
Minste	0.93	0.628	232	231.399	227.38	204.93	207.33	222.036333	12.1916984
Snitt			179.946	157.66	130.18	334.233	383.311	223.946333	113.031991
Beste match			229.76	235.67	232.11	233.644	227.033	231.838	3.72544622
Minste	1.465	0.628	142.06	144.48	147.68	145.71	146.631	145.607	3.85970455
Snitt			291.84	293.81	259.922	258.86	251.4	270.207333	15.6953549
Beste match			151.55	151.72	153.64	154.12	152.2	152.52	1.68502491
Minste	2.087	0.628	100.575	101.84	101.383	105.23	97.34	101.266	7.79136137
Snitt			131.4	157.9	159.41	138.78	232.59	152.03	66.5592317
Beste match			109.25	110.7	111.51	111.65	112.2	111.286667	2.65081172
Minste	2.848	0.628	79.03	75.83	74.59	76.07	72.58	75.4966667	8.54342355
Snitt			136.39	134.12	94.76	133.55	132.03	133.233333	31.2459345
Beste match			81.14	80.34	77.72	80.55	80.43	80.44	4.25161611
Minste	3.35	0.628	63.295	59.2	63.427	62.36	64.211	63.0273333	7.95051882
Snitt			80.11	107.79	112.045	113.0944	117.28	110.976467	33.4935875
Beste match			64.315	64.43	65.44	66.47	65.34	65.07	3.31181804
Minste	4.321	0.628	51.24	52.96	51.99	47.65	47.56	50.2933333	10.7370095
Snitt			102.58	56.37	103.013	78.53	78.61	86.5733333	53.8768674
Beste match			53.71	54	53.47	53.77	53.264	53.65	1.37185461

Figur 22: Målinger av pixelendring for ulike avstander til objekt/bunn

I tabellen over er målingene delt i *fra* og *tilbake* til utgangsposisjon. Det er interessant å se nærmere på hvor mye pixelendringen varierer for samme type måling, og ikke minst hva forskjellen i målingen er for frem og tilbake. Ser man på figuren er «beste match» markert ut som den som gir minst variasjon i samme type måling konsistent. Ser man på tallene for «minste posisjonsendring» er det også ganske liten variasjon i målingene, da det er en spesifikk måling som forstyrrer det totale snittet.

Beste match gir variasjon på mellom 0.6 til 4.9% i pixelendring for 5 målinger. Dette er lavt nok til at man kan si at det gir gode resultater for videre utvikling av konseptet. Basert på dette gir Beste match godt resultat i variasjon mellom like type målinger. Andre måling skulle ta for seg variasjonen mellom målingene man har fra utgangsposisjon og tilbake til utgangsposisjon, for å se om man klarer å ha god nøyaktighet på pixelendring ut ifra et referansepunkt:

	avstand(m)	distanse(m)	Snitt avvik	Snitt avvik i %
Minste	0.625	0.628	11.4298	3.719124705
Snitt			52.392	9.860755166
Beste match			2.8902	0.86570085
Minste	0.93	0.628	8.5704	3.958160303
Snitt			35.4776	15.62076716
Beste match			6.9134	2.984849946
Minste	1.465	0.628	3.1738	2.175422483
Snitt			30.1344	11.42925051
Beste match			1.388	0.911149769
Minste	2.087	0.628	2.3872	2.333460407
Snitt			20.064	12.01029591
Beste match			1.97	1.78836203
Minste	2.848	0.628	3.2214	4.268045745
Snitt			16.122	11.88193242
Beste match			1.148	1.427150671
Minste	3.35	0.628	2.1106	3.35424061
Snitt			9.11468	8.201960381
Beste match			2.3244	3.514080106
Minste	4.321	0.628	2.1132	4.265643924
Snitt			16.5264	19.10157831
Beste match			1.3484	2.482784018

Figur 23: Gjennomsnitt av avviket mellom fra og til utgangsposisjon, med avviket også i prosent

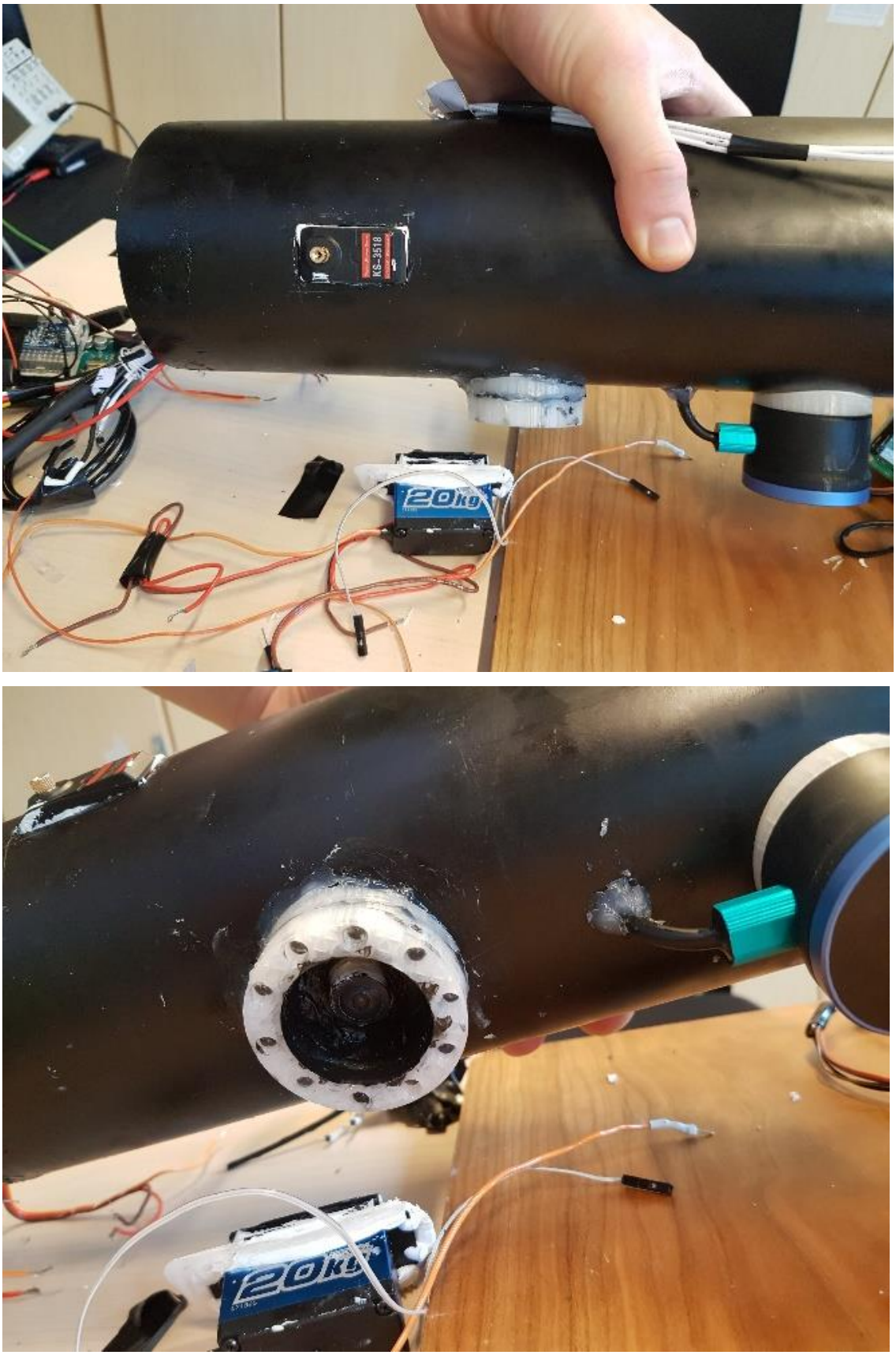
Både minste pixelendring og beste match gir i gjennomsnitt variasjoner som er under 5%. Det kan dermed vurderes slik at begge disse to vil evne å måle endring ut ifra et referansepunkt og holde kontroll over når man er tilbake til utgangsposisjon. Samtidig har beste match en mye lavere variasjon og dermed en bedre nøyaktighet enn minste pixelendring, noe som gjør den til et bedre valg.

Samlet sett for de to målingene kunne man tatt i bruk både minste pixelendring og beste match. Minste pixelendring har samtidig noen enkeltmålinger som varierer kraftig, og som kan skape store feil i posisjonsmålingen. Beste match har derimot ingen store variasjoner hverken i samme type måling eller i forhold til motsatt måling. Det innebærer at ut ifra målingene vil beste match ha et nøyaktig forhold mellom pixelendring og faktisk posisjonsendring, noe som gir muligheten til å utarbeide en algoritme for posisjonsmåling. Men dette er også kun 5 målinger per avstand/retning og ikke 1000 som er det man burde hatt for å forsikre seg om at systemet fungerer.

Det er flere kilder til mulige feil man må være observant på når en går videre. For det første er testen gjennomført på land og ikke i undervannsforhold. Selve testen av dronen vil også foregå i et basseng med klar sikt og få eller ingen partikler i vannet som kan skape forstyrrelser i målingen. For det andre er det testet kun for 4.3 meter, noe som innebærer at det ikke er sjekket og testet FM opp mot større avstander der sikten til bunn også forverres og krever bedre lysforhold. For det tredje er det også gjennomført målinger over et strekk på litt over en halv meter, noe som gjør at man ikke har god nok data på om unøyaktigheten blir større jo lengre avstandene er. Dette er noe man selvsagt vil få testet når en har dronen i bassenget, men det er likevel en feilkilde å være klar over. Man kan dermed ikke utelukke at FM vil få problemer med nøyaktigheten når det legges inn uklar sikt og lengre avstander til bunn. Samtidig må en huske på at oppgaven er en form for prototype med et svært begrenset budsjett. Et skarpere og mer nøyaktig kamera som er bedre, pluss et kraftigere lys vil kunne bedre systemets evne en god del.

2.3.3 Endelig løsning

Den løsningen man gikk for var å hente posisjon gjennom bildebehandling, da samtlige målinger som er gjort viser til en god nok nøyaktighet som tilfredsstiller kravene. Man har følgende oppgaver å løse: En må ha et fastmontert kamera som til enhver tid har retning vinkelrett i forhold til bunnen, og en må ha en algoritme som regner ut posisjonen basert på avstanden til bunnen. Enhver endring på skroget vil endre hydrodynamikken til dronen, og skape uante endringer i fremdriften. Allikevel er dette den eneste løsningen basert på hvordan skroget vårt er bygget opp, og det monteres derfor et kamera på undersiden av dronen sammen med en sonar.



Figur 24: Montert kamera av typen wide-angle Pi-kamera med lysdioder rundt. Merk sonaren til høyre

Det er noen utfordringer med bruk av kamera til å holde posisjonen. En av dem er at en er avhengig av sikt til bunn for å i det hele tatt måle endringer i bildene, ellers vil dronen anta at den står i ro. For havområder som er mørke og dybder der lyset ikke når ned vil en være avhengig av å ha en form for lys til å nå bunnen. Rent praktisk har man her brukt lysdioder montert i en ring rundt kameraet for å hjelpe kameraet i å se bunnen.

Figur 25: test av lysstyrken til lysdiodene, dette er godt nok lys til å teste kameraet nært bunnen

For at lyset ikke skulle forstyrre kamerabildene ble det nødvendig å farge området rundt kameraet svart slik at lyset ikke skulle treffe direkte på linsen. Men det er ikke alltid lysforholdene kan utbedres godt nok til å få sikt til bunn. En slik løsning for posisjonsmåler begrenser dronens evne til å brukes på dypere havområder med mindre den designert skal brukes nært bunnen. En annen utfordring er at bunntopografien kan variere mye og ikke alltid oppføre seg flatt, ulik avstand til bunn på samme kameraområde eller helning/skråning vil kunne forstyrre nøyaktigheten til posisjonsmåleren. Videre er en annen utfordring om hvordan posisjonsmåleren reagerer på objekter som flyter forbi imellom dronen og bunnen. I teorien vil objektet lagres som et flyttbart punkt i posisjonsmåleren, og kan dermed forstyrre nøyaktigheten. Men man har allerede nevnt at vi begrenser oss til de aller beste matchene imellom punkter. Det baserer seg på at slike objekter vil være blandt de dårligste matchene og ha høy forandring i posisjon og dermed ikke forstyrre posisjonsmåleren, men dette er rene antagelser. En kan ikke utelukke at dette er en feilkilde før det blir skikkelig testet over tid, noe som ikke blir gjort i denne oppgaven.

Det at kameraet er fastmontert skaper også en utfordring med hva som skjer hvis dronen begynner å rotere. Dette blir løst ved at man setter kriteria om at dronen aldri skal rotere seg om x- og y-aksen, slik at kamera alltid fra dronens perspektiv vil stå vinkelrett nedover. Kriteriet blir gjengitt i 2.8.

Videre var det nødvendig å vite avstanden til bunnen for at posisjonsmålingen skal bli nøyaktig nok. Ser man på figur 24, er det montert en sonar til høyre for kamera(akterut i referanse til dronen) kalt Ping Sonar. Denne brukes til å bestemme avstanden til bunn som

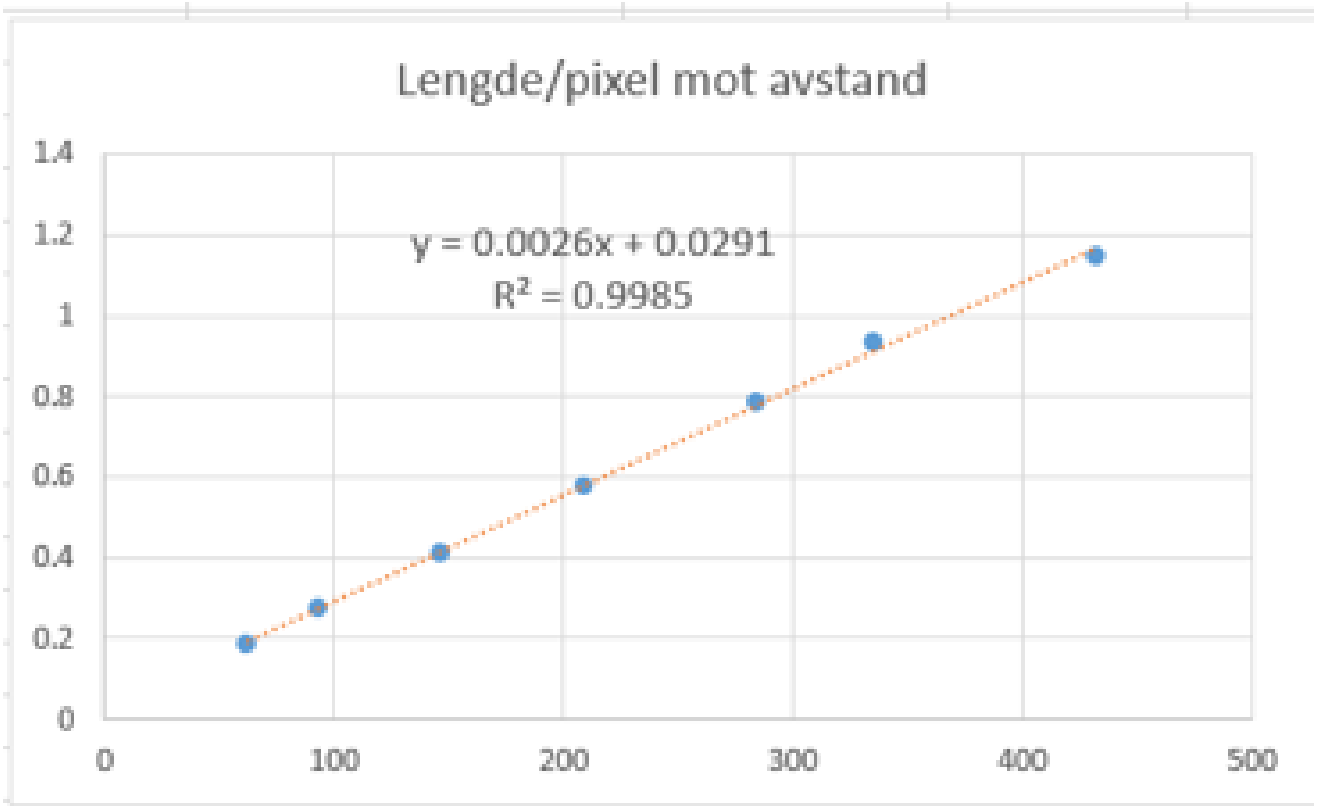
så brukes av dronen til å bedømme rett posisjonsmåling. Sonaren har en 30-graders strålebredde til hver side, og kalkulerer ut fra hele sonarbildet hva som oppfattes som nærmeste faste objekt. Dette gjør den ideell for bruk som måler av avstand til bunn.

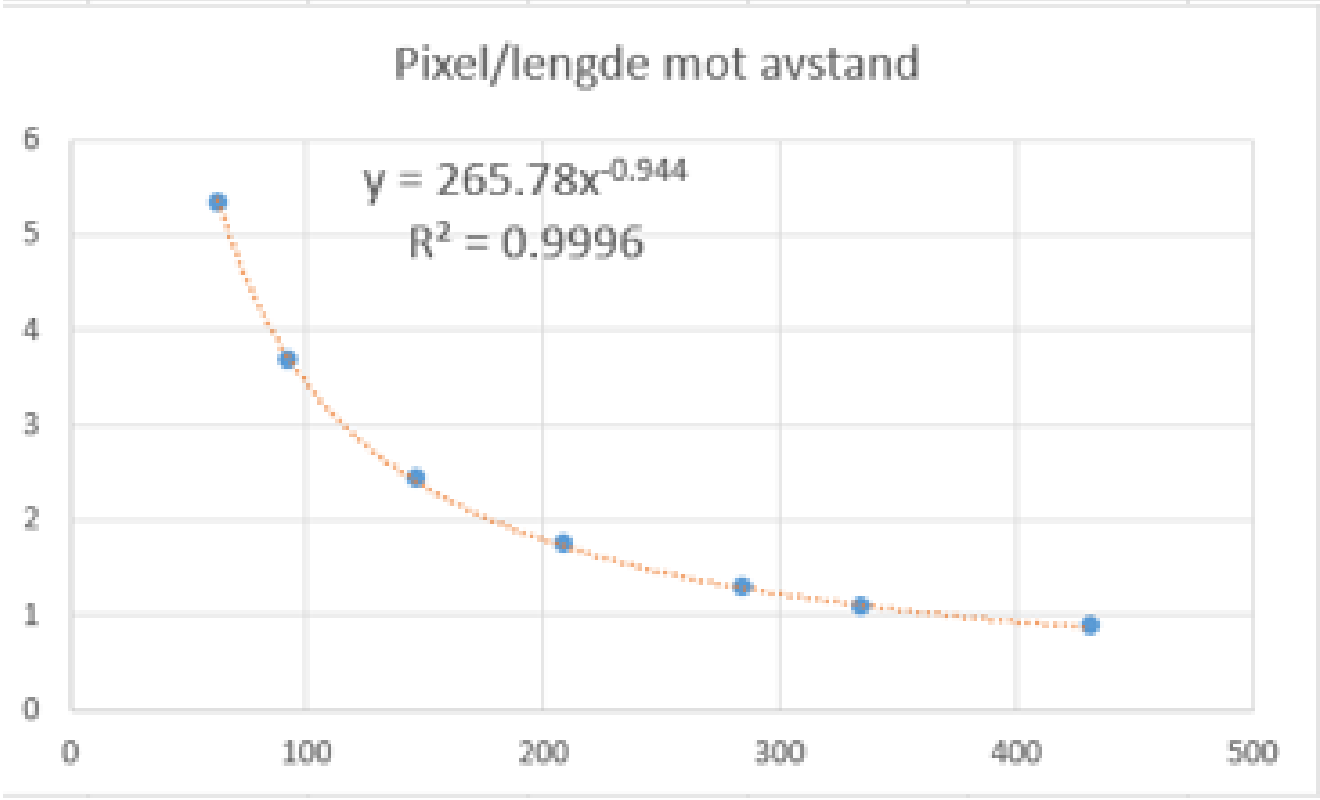
For beregning av posisjon tas det utgangspunkt i målingene fra tabellen. Man kan nå velge imellom å bruke dataen *fra* utgangsposisjon eller tilbake *til* utgangsposisjon. Basert på at dronen skal bevege seg fremover mesteparten av tiden, vil det være hensiktsmessig å velge den første alternativet, samt at variasjonene var så lave at nøyaktigheten fortsatt skal være god. Snittet av målingene beregner forholdet til lengden man har beveget kameraet, som gir to muligheter:

$$Metode\ 1 = \frac{Pixelendring}{Lengde(cm)}$$

$$Metode\ 2 = \frac{Lengde(cm)}{Pixelendring}$$

Hvor lengde er avstanden en har beveget kameraet som i målingen var 62.8cm. Dette gir to forholdstall som sammen med pixelendringen beregner faktisk forflytning i x- og y-retning. Begge vil gjøre jobben, men har ulike egenskaper som en ser når grafene plottes:





Figur 26: Plotting og beregning av trendlinje for de to forholdstallene

Fra 2.2 så vil eksponentielle linjer være utsatt for feil skulle målingene være unøyaktige. Men ser en direkte på hvor god trendlinjen er, er det den eksponentielle som er mest nøyaktig. Basert på høyest nøyaktighet velges metode 1 til videre bruk. Ut ifra denne fås trendlinjen og algoritme:

$$f(z) = 265.78 * z^{-0.944} = Pixel/Lengde(cm) \quad (15)$$

$$f(\Delta p_x) = \frac{\Delta p_x}{f(z)} = \frac{\Delta p_x}{265.78 * z^{-0.944}} = \Delta x \ \ og \ f(\Delta p_y) = \frac{\Delta p_y}{f(z)} = \frac{\Delta p_y}{265.78 * z^{-0.944}} = \Delta y \quad (16)$$

2.3.4 Sammendrag

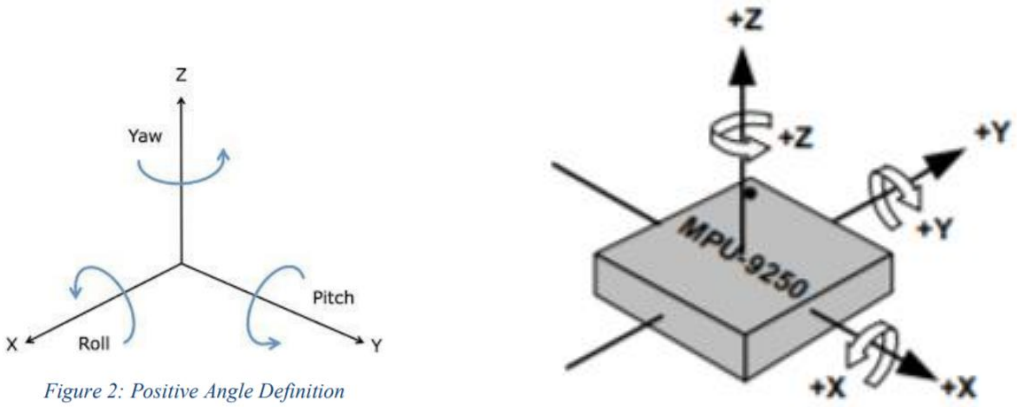
En har nå et system for å beregne posisjonen som er nøyaktig nok til å tilfredsstille kravet. Samtidig er dette enda ikke testet i vann, og man må få kjørt gode tester her før en kan si at systemet fungerer godt nok. Videre vil videre utvikling av dronen kreve en kraftig utvikling for posisjonsmåleren. Den må forbedre evnen til å fungere i varierende og ikke alltid flate bunnforhold, samt ha bedre og skarpere kameraer for å takle dårlig sikt og varierende lys. En del av testingen som derfor må gjøres er å sjekke dronens evne til å beregne posisjonen nøyaktig nok for flere dybder og i varierende lysforhold.

2.4 Kontroll på egenrotasjon

Kommersielle droner har som regel et display som viser hvilken vei dronen er rotert i det tre-dimensjonal rom. Dette er en egenskap som kreves så og si for alle undervannsdroner, og det finnes nok av oppskrifter på nettet for hvordan man skal gjøre det. For å tilfredsstille våre krav trengs derfor ulike sensorkomponenter som kan måle egenrotasjon. Først må rotasjonen om de tre aksene X,Y og Z defineres: .

- Yaw: Rotasjon om z-aksen
- Pitch: Rotasjon om y-aksen
- Roll: Rotasjon om x-aksen

I figuren under ser du rotasjonsaksene med definerte retninger i henhold til høyre-håndsregelen.



Figur 27: Definisjon av de tre rotasjonsaksene, og lignende rotasjon for sensorbrikken en skal bruke. Kilde: http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1422&context=eesp https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

På markedet finnes det komponenter innenfor vår budsjettramme som kan tas i bruk. En sensor vi baserer oss på er et gyroskop, som kan måle vinkelhastighet og dermed måle rotasjon om aksene. Gyroskopet evner å gjøre dette for samtlige tre akser, men har en utfordring med at det er utsatt for drift. Tidligere ble det nevnt at gyroskopet måler vinkelhastighet, noe som innebærer at for å få ut vinkelposisjon må det gjøres noen utregninger:

$$Vinkel_t = \int w(t)\,dt = Vinkel_{t-1} + \ w * \Delta t \qquad (17)$$

Hvor «w» er vinkelhastigheten målt i grader per sekunder og t er tiden. Problemet oppstår ved støy i vinkelhastigheten, altså noen verdier er falske. Da dette er en summasjon vil det være drift av vinkel over tid. Med andre ord, gyroskopet vil drifte og komme bli feil over tid. Dette betyr at det trengs flere sensorer for å fusjonere sammen med gyroen

2.4.1 Rotasjon om x- og y-aksen (roll og pitch)

Et akselerometer har andre egenskaper i forhold til støy enn et gyroskop som har utfordringer med høyfrekvent støy. Der gyroskopet derimot har utfordringer med lavfrekvent støy som blir integrert slik at det drifter over tid, har akselerometeret utfordringer med høyfrekvent støy. En kombinasjon av disse to vil dermed i teorien kunne gi en mer nøyaktig rotasjon. Der gyroskopet drifter og vil ha problemer med å komme seg tilbake til utgangsposisjon, beregne akselerometeret rotasjonen i forhold til gravitasjonen som er bedre på å beregne statiske perioder. For akselerometeret har vi:

$$pitch = \frac{Ax}{\sqrt{Ax^2+Ay^2+Az^2}} \qquad roll = \frac{Ay}{\sqrt{Ax^2+Ay^2+Az^2}} \tag{18}$$

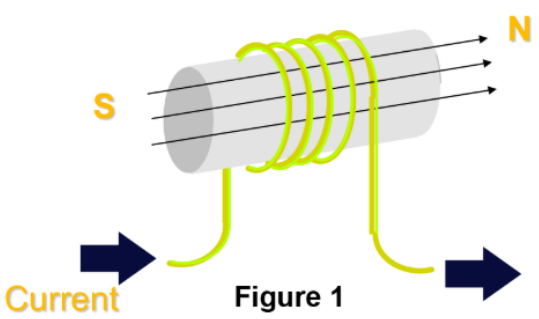
Hvor Ax,Ay og Az representerer en tredimensjonal vektor av gravitasjonen, som ved ro-tilstand skal være [0,0,-1].

2.4.2 Rotasjon om z-aksen (yaw)

I motsetning til gyroskopet som klarer å måle rotasjon om alle aksene, kan ikke akselerometeret måle yaw. Akselerometerets beregninger av rotasjon baserer seg på gravitasjon, som er en vektor med kraft konstant nedover mot jorden. Dermed vil den ikke oppføre seg annerledes om man roterer objektet om z-aksen, akkurat som mennesker ikke opplever forskjell i gravitasjon avhengig av hvilken vei du står på jorda. Det trengs derfor en tredje komponent til å fusjonere gyroens måling på yaw. Til dette brukes et magnetometer, som igjen vil ha en del utfordringer med magnetisk støy og påvirkning av rotasjon om de andre aksene. Heldigvis er utfordringene til magnetometeret mulig å løse uten at det påvirker de to andre rotasjonsmålingene nevneverdig.

Magnetometeret må både kompenseres for magnetiske støykilder i operasjonsområdet, samt å kompensere for målinger med samtidig rotasjon om x- eller y-aksen. Det første kan være ganske krevende om de magnetiske støykildene ikke er faste, altså at forstyrrelsene endrer seg over tid. Slike støykilder vil det ikke være mulig å kompensere for og er derfor avhengig av at de er konstante. I det tilfellet kan det enkelt kalibreres ut slik at de ikke forstyrrer sensoren. Problemet vil derimot oppstå hvis det er skiftende støykilder som ikke er konstante og dermed ikke kan kalibreres skikkelig. En utfordring som oppsto under testing var at klasserommet som ble brukt hadde ekstremt mange varierende støykilder som gjorde det tilnærmet umulig å lage en god kalibrering for magnetometeret.

Hvordan dette påvirker magnetometeret når det plasseres inne i dronen vil avhenge av hvordan de to potensielle støykildene skaper forstyrrelser. Motorene antas å ikke skape noen problemer da de både er montert på utsiden av dronen, og at magnetometeret vil finne seg i fronten og langt unna motorene. Derimot kan servoene skape utfordringer da de befinner seg rett ved siden av magnetometeret. Grunnen til at dette er en utfordring er at måten servoene roterer på skjer via magnetisk fluks, som vil skape et skiftende magnetfelt avhengig av hvilken vei man roterer servoen. Tidligere ble det nevnt at konstant magnetfelt kan kalibreres bort, men da servoen vil skifte stilling ofte, vil også det magnetiske feltet skifte styrke og retning. Dette er ikke mulig å kalibrere bort, og er noe vi må være obs på når vi monterer magnetometeret.



This magnetic field created by current passing through the conductor will have a north pole and a south pole. With magnetic poles located on the stator (when energized) and on the permanent magnets of the rotor, how do you create a state of opposite poles attracting and like poles repelling?

The key is to reverse the current going through the electromagnet. When current flows through a conducting coil in one direction, north and south poles are created.

When the direction of the current is changed, the poles are flipped so what was a north pole is now a south pole and vice versa. Figure 1 provides a basic illustration of how this works. In figure 2, the image on the left shows a condition

Figur 28: Servoens skiftende magnetfelt, Kilde: http://blog.parker.com/a-quick-guide%3A-how-servo-motors-work

Måten vi kalibrerer dataen på er å bevege sensoren rundt om alle de tre aksene, akkurat som om vi malte innsiden av en ball. I en eksempelkalibrering er dataen plottet i henholdsvis xy, xz og yz. Så kan man gjennomføre to kalibreringstyper som kalles hard-iron og soft-iron hentet fra Kris Winers[3] innlegg på Github. Når man «maler innsiden» og plotter resultatet skal et ideelt magnetometer ha tre perfekte sirkler i samme størrelse som har sentrum i origo. Et hvert avvik fra dette skyldes da støy eller feil i magnetometeret som må kalibreres. Hard-iron sentrerer dataen rundt origo, mens soft-iron utjevner skjevheter i den sirkulere formen på dataen. For Hard iron regner man ut gjennomsnittet av alle tre aksene hver for seg.

$$xyz_{avg} = \frac{xyz_{max}+xyz_{min}}{2} \qquad (19)$$

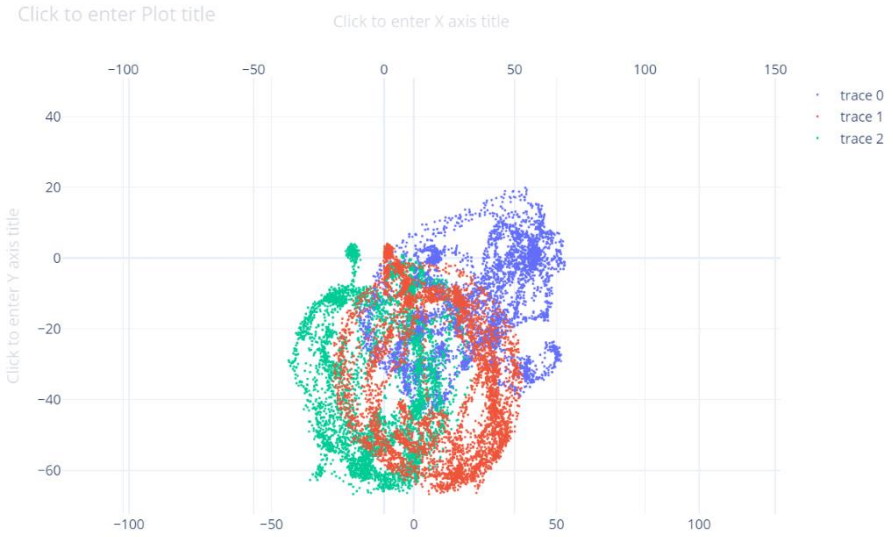
Hvor xyz representerer alle tre aksene separat. I et ideelt miljø skal x,y og z være lik 0, altså like mye bevegelse til begge sider av origo, som gir en perfekt sirkel. For soft-iron er det kompensering i form av skalering. Her er man ikke interessert i gjennomsnittet men heller *spennet* til de tre aksene. Ved å så vekte dem i forhold til totalspennet kan en skalere dem slik at de får like spenn. I et plot vil dataen da være sirkuler siden man har skalert dem like.

$$xyz_{spenn} = \frac{xyz_{max}-xyz_{min}}{2} \qquad (20)$$

$$spenn_{snitt} = \frac{(x_{spenn}+y_{spenn}+z_{spenn})}{3} \qquad (21)$$

$$xyz_{skalering} = \frac{spenn_{snitt}}{xyz_{spenn}} \qquad (22)$$

Hvor xyz representerer alle tre aksene separat. I et ideelt miljø skal spennet til x,y og z være like slik at skaleringen til alle tre aksene er lik 1. Ellers vil det ved forstyrrelser i det magnetiske feltet være noen av de tre aksene som må skaleres ned og noen som må skaleres opp slik at de har likt spenn i forhold til hverandre.
Her er det plottet inn verdiene for de tre aksene i forhold til hverandre med xy(blå), xz(rød) og yz(grønn):



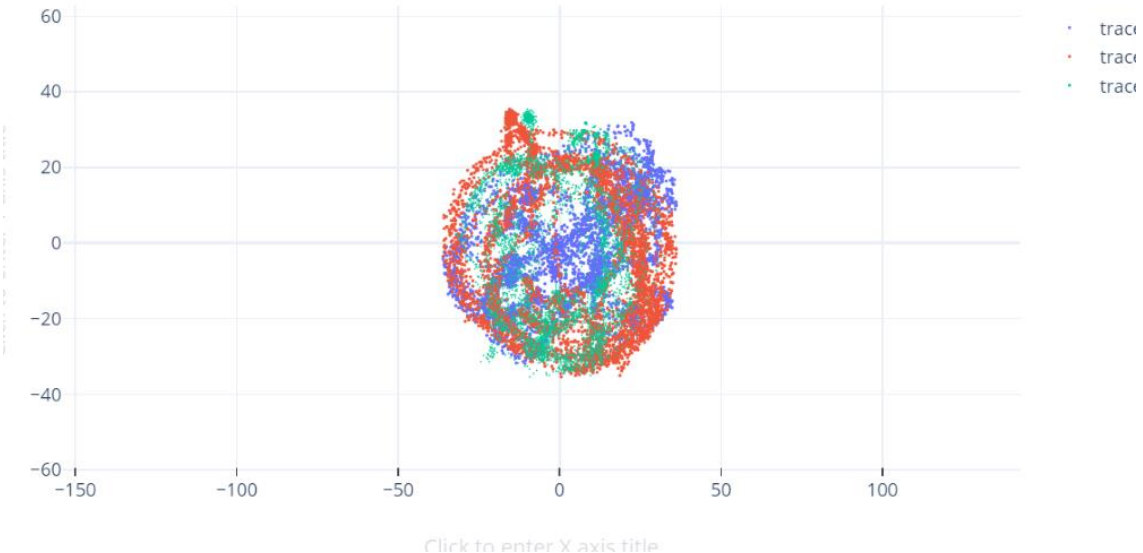
Figur 29: Plotting av en måling av magnetometerdata der man har rotert sensoren akkurat som man maler innsiden av en ball

Det er ganske tydelig at samtlige av de tre aksene må forskyves oppover i grafen for at de skal være sentrerte om origo. Plottet kan vitne om en form for elipse for xz og yz, mens xy ser ut til å ha en litt mer sirkulær tilnærming. Det kunne vært nødvendig å hatt flere målinger for lavere verdier slik at skillet ble enda mer tydelig, men man kan allerede bedømme at samtlige av aksene må forskyves positivt og at en må skalere om noen av dem. Ved å utføre kalkulasjonene kan man kalibrere slik:

$$XYZ_{kalibrert} = (xyz_{rådata} - xyz_{avg}) * xyz_{skalering} \qquad (23)$$

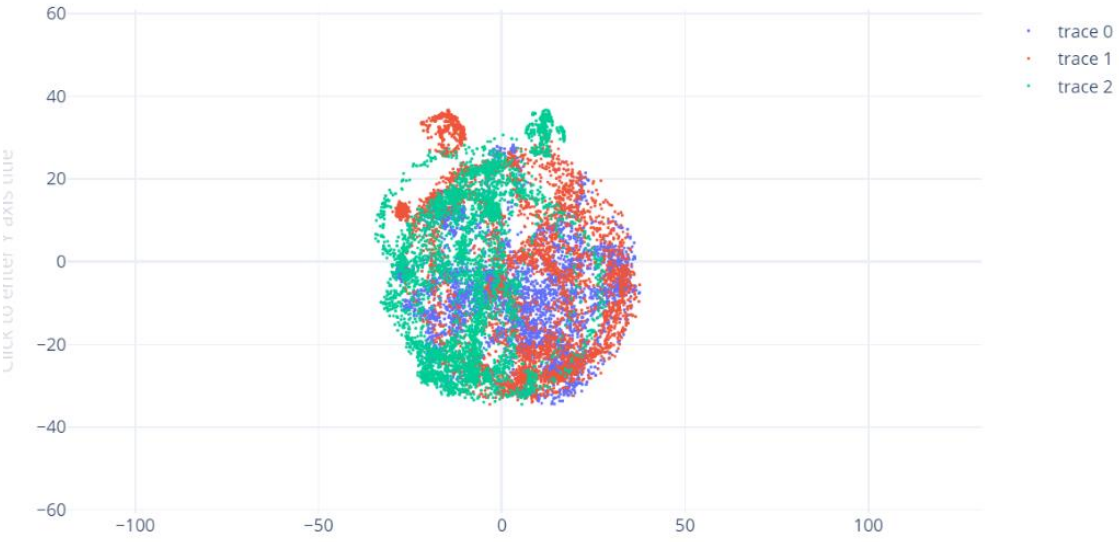
Hvor xyz er de tre separate aksene. Hvis en har en negativ forskyvning der minste verdi er lavere i forhold til største verdi, vil også ligningen over gi ut positiv forskyvning mot høyre for å komme til origo. Ut ifra oppsettet for ligningen til skaleringen vil den være større enn 1 hvis man har et lavere spenn enn gjennomsnittet av de tre aksene, og dermed forstørre spennet sitt. Motsatt vil en positiv forskjell gi en negativ forskyvning mot origo, og hvis man har et

større spenn enn gjennomsnittet vil ligningen for spenn gi ut en skalering mindre enn 1, som dermed nedskalerer dataen. Ved å re-kalibrere samme datasett får man:



Figur 30: Samme data rekallibrert med forskyvning og skalering

En ser tydelige forskjeller på dataen nå og tidligere. Dataen er nå sentrert rundt origo og har en mye tydeligere sirkulær form, noe som gjør at dataen er kalibrert riktig. Dette var som sagt samme datasett som tidligere, så det som gjenstår er å implementere den nye kalibreringen og måle ny data:



Figur 6: Ny data etter implementering av ny kalibrering, den er mye bedre enn første måling

Man ser tydelige forskjeller også på nye målinger i forhold til de første vi hadde, dog har man noe ujevnhet oppe i venstre del av figuren som tyder på at man kan få til en enda bedre måling med mer data. Da magnetiske forstyrrelser er ulike avhengig av hvilket miljø en

befinner seg i, er det viktig at man får tatt en ny kalibrering når dronen befinner seg i operasjonsområdet. Dette innebærer at en må gjøre kalibreringen automatisk i motsetning til hva man har nå, men dette er ikke noe vi kommer til å gå mer inn på i detalj.

Når man roterer magnetometeret om x- eller y-aksen vil den beregnede yaw variere, og altså gi ut feil data. Derfor må en tilt-kompensere magnetometeret slik at det takler å beregne yaw også når det blir utsatt for rotasjon om x- og y-aksen. Formelen for tilt-kompensering finner man i ulike publikasjoner og instruksjoner på nettet der:

$$Y_{komp} = Y_{filtrert} * \cos(roll) + Z_{filtrert} * \sin(roll) \quad (24)$$

$$X_{komp} = X_{filtrert} * \cos(pitch) + Y_{filtrert} * \sin(roll)\sin(pitch) - Z_{filtrert} * \cos(roll)\sin(pitch) \quad (25)$$

Hvor filtrert er rådata etter at den har gått igjennom kalibreringen beskrevet over, og pitch og roll hentes ut fra fusjoneringen av gyroskop og akselerometer.

2.4.3 Sammendrag

Man har nå et system som gir dronen mulighet til å ha kontroll på egenrotasjon i alle akser, ved bruk av en fusjonering mellom tre sensorkomponenter. Samtidig er det nevnt at akselerometeret er utsatt for vibrasjoner og at magnetometeret har problemer når forstyrrelsene ikke lenger er konstante. Dette er utfordringer som kan dukke opp når en opererer dronen, og krever derfor at man er observant når en tester. Blir avvikene for store kan dronen ende opp med å ikke lenger tilfredsstille kravet om egenrotasjon og dermed ikke løse hovedmålet.

2.5 Konstruksjon av drone

Dette kapittelet vil ta for seg viktige momenter for konstruksjonen. Denne delen vil også oppgi hvilke deler som er brukt og metoder for installasjon.

2.5.1 Skrog

Valget av skrog var et resultat av en lengre diskusjon med Blueye Robotics. Valget kom på bakgrunn i at vi ønsket en plattform som først og fremst var enkel å få tett, der nest være strømlinjeformet uten for store endringer på det allerede vanntette skroget måtte gjøres. Valget falt derfor på en rørdimensjon som hadde alle tilkoblinger som hyllevare. Som et resultat av dette ble det valgt et PVC-rør med ytre diameter på 110mm. Denne type rør er

mye brukt i VVS med integrerte vanntette pakninger. Disse pakningene gav mulighet til å sette sammen og demontere dronen enkelt. Det dette systemet er det også mulighet til å enkelt sette på ekstra seksjoner som for eksempel sonar. Diameteren på dronen var lenge en diskusjon, og en utfordring ved montasje. I alle systemer må det gjøres kompromisser, et av disse kompromissene var å velge et rørstykke som var liten nok til at det fortrengte volumet ikke var for stort, slik at volumet måte kompenseres med ekstra tyngde. Denne tyngden ville skape større treghet og ville derfor kreve mer energi for fremdriften.

Formen på skroget og styringsmekanismen er en viktig del i forhold til hvordan type oppdrag som dronen skulle utføre. Det tre løsningene som finnes på markedet i dag er enten fastmonterte motorer som er plassert i ulike retninger slik at dronen har mulighet til å bevege seg i alle 3 dimensjonene. Den andre er fastmonterte motorer med finner som styrer. Den siste er en blanding av de to, motorer som kan roteres. Til denne oppgaven ble fastmonterte motorer valgt da oppdragene som ble fokusert på ikke krevde finmanøvrerbarhet i lav fart, men heller en lengre rekkevidde.

For å gjøre navigasjonsprogrammet mindre komplisert og åpne for muligheten til å bruke billigere sensorer ble aksene låst. Det som menes med å låse aksene er at programmet ikke tillater endringer i pitch og roll. På bakgrunn av dette var det nødvendig med 8 ror som kunne fungere både selvstendig og parvis. Hvis for eksempel dronen skal dykke dypere settes de fire rorene som er monter på sidene av dronen til å peke nedover. Dette vil da presse dronen dypere uten at den skal endre retning i pitch. En utfordring med denne løsningen er at den ikke kan dykke like fort som droner som ikke har låste akser. Denne problemstillingen er en avveining som har gitt større handlingsrom enn den har snevret inn i programmeringsdelen av denne oppgaven.

2.5.2 Batteri

Dronen bruker to litium polymer (LiPo) batterier som hver har en kapasitet på 10 000mAh og 14,8 volt. Dette gir en effekt på 148Wh per batteri. Batteriene er ikke koblet sammen i serie eller parallell da dette gir utfordringer med kortslutning hvis en av de fire battericellene inne i batteriet blir ødelagt eller skadet.

Hver av batteriene vil drive hver sin motor. Utover dette vil det ene batteriet drive alle servomotorene gjennom en DC/DC konverter som transformerer spenningen fra 14,8Volt til 5 Volt som de 8 servomotorene vil bli drevet av. Det andre batteriet vil også ha en DC/DC ovformer som senker spenningen fra 14,8Volt til 5Volt. Dette batteriet vil drive Raspberry pi, arduino mega og en Jetson nano.

2.5.3 Motor

Dronen har 2 motorer av typen T200 som er satt på hver side av kroppen av dronen. Motorene ble valgt siden de var beregnet for bruk under vann og siden de hadde med sikkerhet nok effekt til å drive dronen i de flest relevante hastigheter. Maksimal effekt er oppgitt til 350 Watt med maksimal spenning på 20 Volt. Selv om 20 Volt er mer enn det batteriene kan levere ble motoren valgt på bakgrunn av mulig effekt vi kunne bruke hvis vi fant andre løsninger på batteriene.

Effekten motoren leverer kontrolleres gjennom en motorkontroller av typen Basic ESC levert av Blue Robotics. Denne motorkontrolleren skal kunne ta imot 7-26V og opp til 30 Amper med strøm.

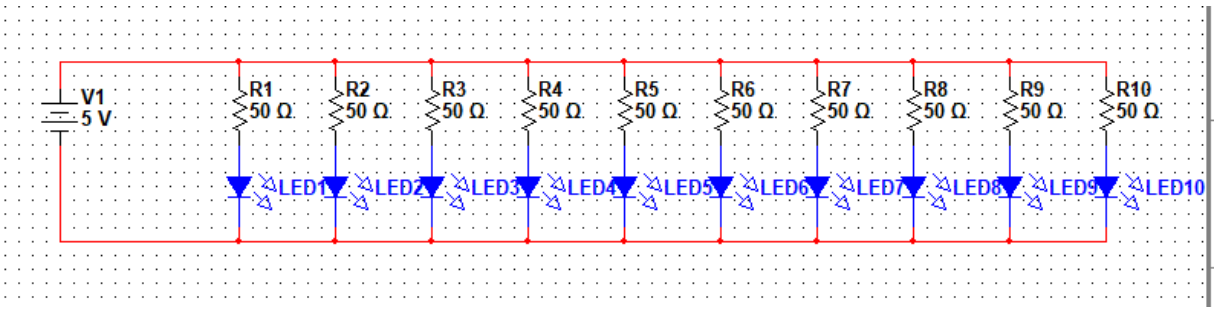
2.5.4 Servomotorer

Modell: KS-3518 Digital Servo med utslag på 90 grader hver vei. Produsenten oppgir at den har en maksimal kraft på 20kg/cm med 6,6V. Omformeren vil levere 5V siden flere systemer skal kobles på samme spenningsnett og ikke nødvendigvis kan tåle så mye spenning. Produsenten oppgir også at servomotorene vil gi 18kg/cm med kraft ved 4,8v. I henhold til spesifikasjonene fra produsenten skal servomotorene være vanntette. Måten dette er gjort for seg er å legge pakninger mellom de forskjellige delene som servomotoren der laget av og en pakning mellom kulelageret og topplokket ved utgangen til kronen på toppen av servomotoren.

2.5.5 Lyskilde

Lyskilden er laget ut fra 10 parallellkablene lysdioder med hvitt lys satt i en sirkel. Systemet skal levere 5V over hver komponent i parallellkoblingen. Ut ifra diodekalkulator skal det settes en 50ohms motstand i serie med dioden for å nå riktig spenning over diodene som skal være 3.3V. Strømforbruk per diode blir da 40mA. Dette gir da 0,04A per diode i strømforbruk og et totalt forbruk på 2W for alle 10 diodene. Diodene er satt inn i en ring som er 3D-printet for hver enkelt diode med kabelføring imellom. Innsiden av ringen som peker

mot kameraet er lakkert med sort gummilakk for å begrense strølys fra diodene som i mange tilfeller gir et gråskjær på linsen.



Figur 31: Forsøk av kretsen kjørt i Multisim med 50 ohm motstander i serie med hver diode for å nå en spenning på 3V over hver av diodene.

Figur 32: Lyskilde for kamera under dronen.

2.5.6 Omformer

Dronen har tre omformere som har total kapasitet på 10 amper. En av typen SD-50A-5 - DC/DC-omformer og 2 av typen: SD-15B-5 - DC/DC-omformer. Den første typen omformer kan levere 10 amper alene, mens de to andre kan kun levere 3 amper hver.

2.5.7 Metode for montasje

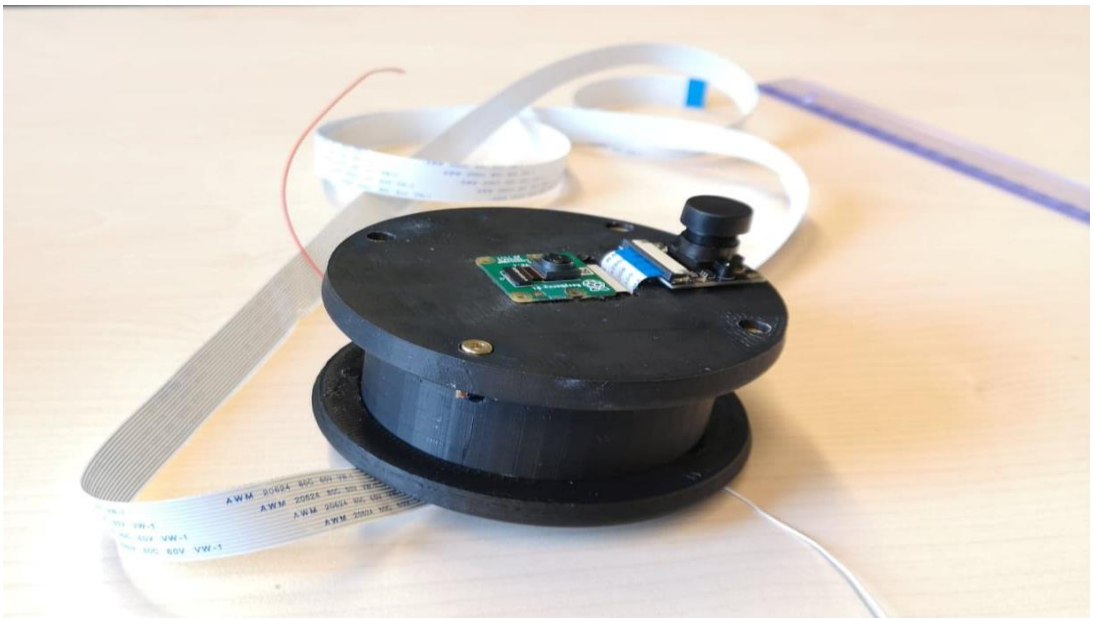
På bakgrunn av at vi ønsket en oppgave som hadde fokus mot elektronikk og data ble konstruksjonen av dronen så enkel som mulig. Før vi startet konstruksjonen besøkte vi i mars 2019 Blueye Robotics i Trondheim for å forhøre oss om hvilke utfordringer de kunne

hjelpe oss med. Det ble luftet ideen rundt bruk av 110mm pvc rør med muffe i hver ende. Dette var Blueye meget positive til og oppfordret oss til å velge noe som allerede var tett.

Røret er 3mm tykt og en diameter på 110mm i pwc plast. Endene er tettet med muffer som holder innsiden tett. Lengden på selve røret er 100cm, men totallengden er 110cm når begge muffer er lagt til i hver ende av røret. I muffene er det satt inn propper som er den siste forseilingen i enden. Oppdriften for dronen er beregnet til 102,55N som gir en optimal vekt på 10,45kg på land.

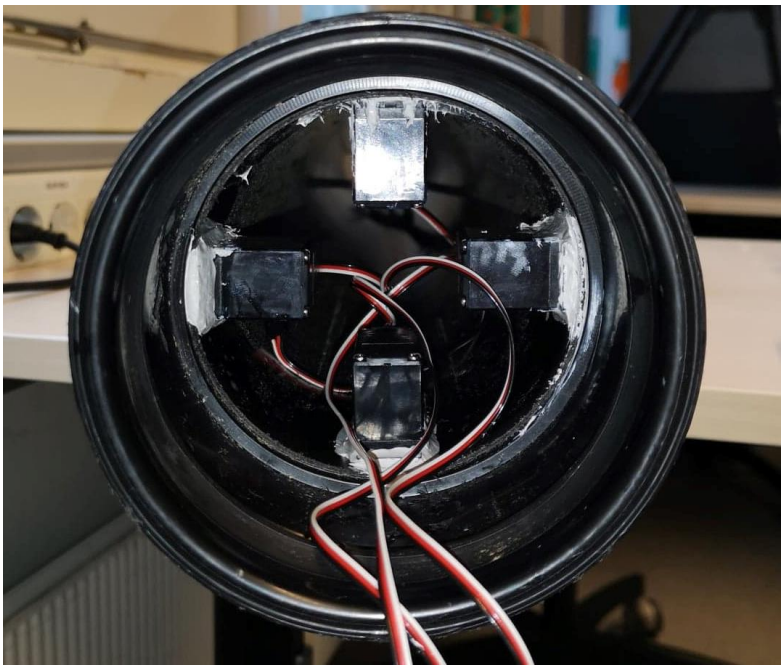
I kroppen er det festet to motorer disse motorene er plassert litt lenger bak enn midten grunnet plass på innsiden av dronen. Plassen som skulle frigjøres ble brukt til batteriene som ble plassert mot midten av dronen for økt manøvrerbaritet i pitch-reting. Dette skyldes hovedsakelig at batteriene er det tyngste komponenten som blir satt inn i dronen utenom ballast.

Frontmuffen med flatt pleksiglass har et kamera i senter med fire lasere plassert i et perfekt kvadrat rundt. Denne holderen for laserne og kameraet ble 3D-printet for minimal feilmargin. Holderen ble printet i to deler og limt sammen til slutt. De to sirkelformede platene som holderen er laget av er av samme form og radius som gjør at laserne vil peke samme retning som dronen seiler. Dette gjorde monteringen av hele fronten på dronen enklere i neste steg av monteringen. Et problem er at 3D-printing ikke er spesielt nøyaktig og laserne måtte stilles inn for hånd. Utfordringen gikk ut på at hullene som blir printet ikke er nøyaktige nok og gjør at laseren ikke peker helt rett frem. Løsningen på dette problemet ble å bore ut større hull og lime laserne i posisjon med hurtiglim. Posisjonen laserne skulle ble funnet ved å lyse mot et rutepapir med oppmålte punkter som laserne skulle treffe.



Figur 33: Kamera i midten og de fire laserne plassert i et kvadrat rundt kameraet. Fish eye kamera med stor linse ble i starten brukt som backup- kamera.

I frontmuffen er det også satt inn to servomotorer. Disse servomotorene er plassert lenger frem enn det servoparet som styrer pitch for å kunne sette inn og bytte komponenter inne i dronen. Servomotorene er limt fast med tec7 og deretter fuget tett med våtroms silikon. Pleksiglasset som dekker for fronten er 6mm klarplast. Denne tykkelsen ble valgt for at ikke glasset skulle deformeres og dermed lage diffraksjoner av laseren under dykking. 7 cm bak det første servoparet sitter servopar nummer to som styrer pitch og roll. Disse er plassert slik at frontmuffen kan presses helt inn sitt originale spor uten å hindre utslaget til roret som monteres på servoparet. Helt akterut på dronen er de fire andre servoene plassert i pluss formasjon symmetrisk om dronen som vist i Figur 34.

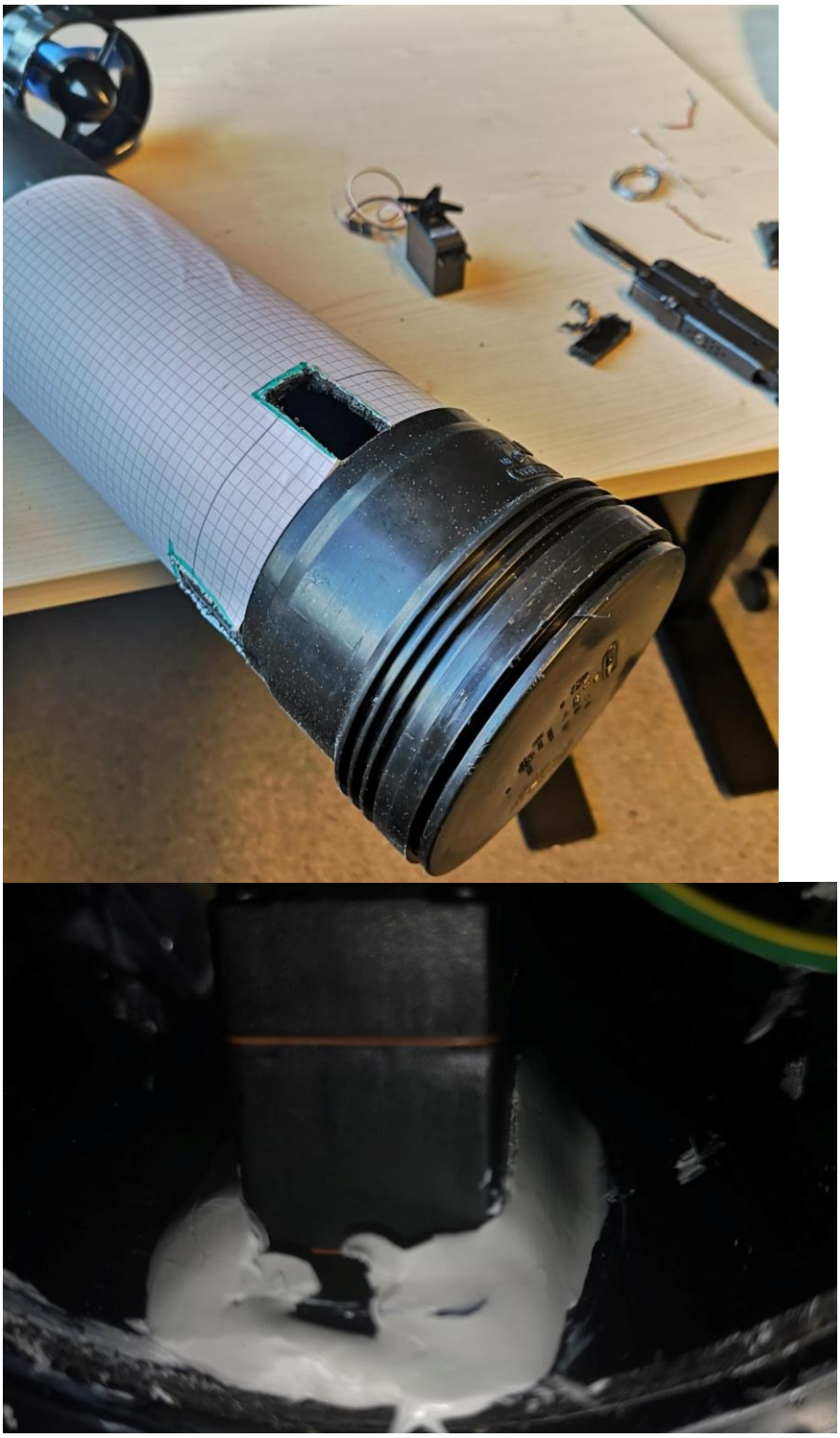


Figur 34: Plassering av aktre servomotorer

2.5.8 Montering av servoer

Monteringen av servomotorene utnytter trykket i vannet for tetningen. På innsiden av skroget er servoen limt med ESS Tack PRO1 som har en festekraft oppgitt til 27 tonn/m2. Dette har blitt lagt i en fuge med så stort anleggsområde som mulig. I tillegg er limet blitt presset inn mellom skrogsiden og selve servomotoren for å Ytterligere øke anleggsområdet. For å være sikker på at innfestningen blir tett ble det laget en fuge av et mykere materiale som blir presset sammen i kanten mellom skroget og servomotoren. Dette mykere materialet er silikon fra Essve som er beregnet for tening av basseng og andre våtrom. Disse to materialene sammen vil skape en vanntett forsegling over lengre tid.

Figur 35: skisse av tetningen rundt servomotorene. Skroget er tegnet i sort, rød er lim av typen Tec-7 og det grønne er silikon.

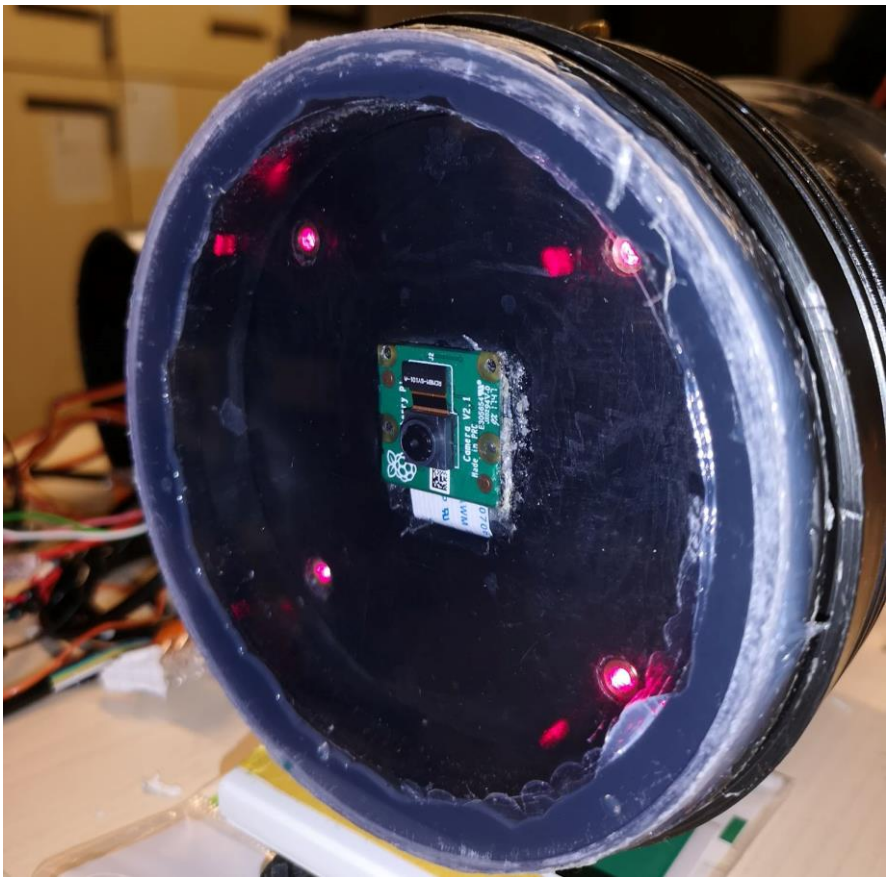


Figur 36: bilde viser hvordan fugen på innsiden av dronen har stor anleggsflate og samtidig har et jevn lag rundt hele servomotorer som sikrer et vanntett resultat

2.5.9 Montasje av front

I fronten av dronen er det satt fire lasere og et kamera som alle peker vinkelrett frem. Disse er beskyttet av et 6mm pleksiglass som er montert flatt på en 110mm muffe. For å lage en tett forsegling ble konseptet med å bruke trykket fra vannet igjen brukt ved at glasset ble

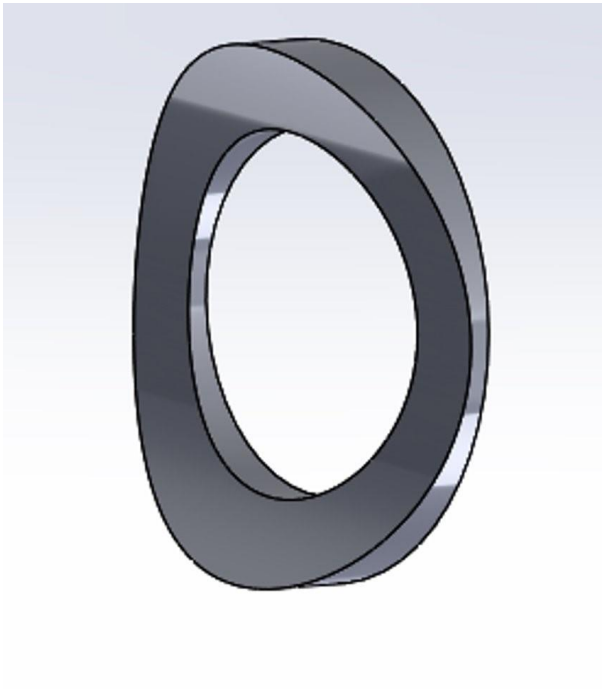
montert flatt på muffen slik at trykket på vannet presset sammen fugen bak glasset. Faktum ble derfor at det trengtes et mykt middel som lar seg komprimere for en tettere pasning. Igjen ble det brukt silikon fra Essve som beste løsning.



Figur 37: Bildet viser den tykke fugen mellom glasset og muffen.

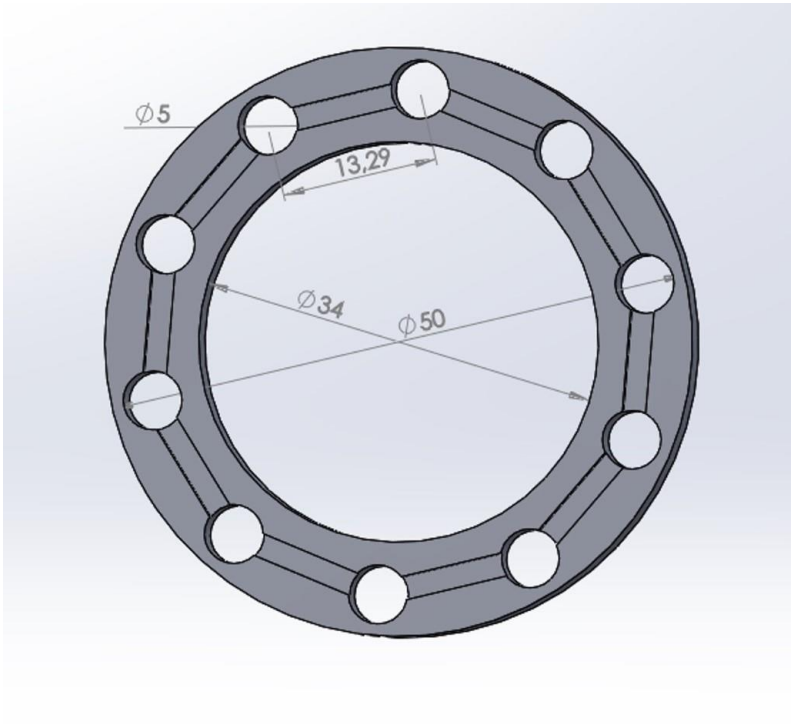
2.6 3D-print:

Som en del av prosjektet har gruppen tilegnet seg kunnskaper innen 3D-print. Dette var helt nødvendig for å kunne gjennomføre prosjektet siden mange av produktene som ble brukt ikke nødvendigvis passer sammen uten tilpasninger. Videre i oppgaven vil vi beskrive de forskjellige delene som ble printet og oppgaven til delene.



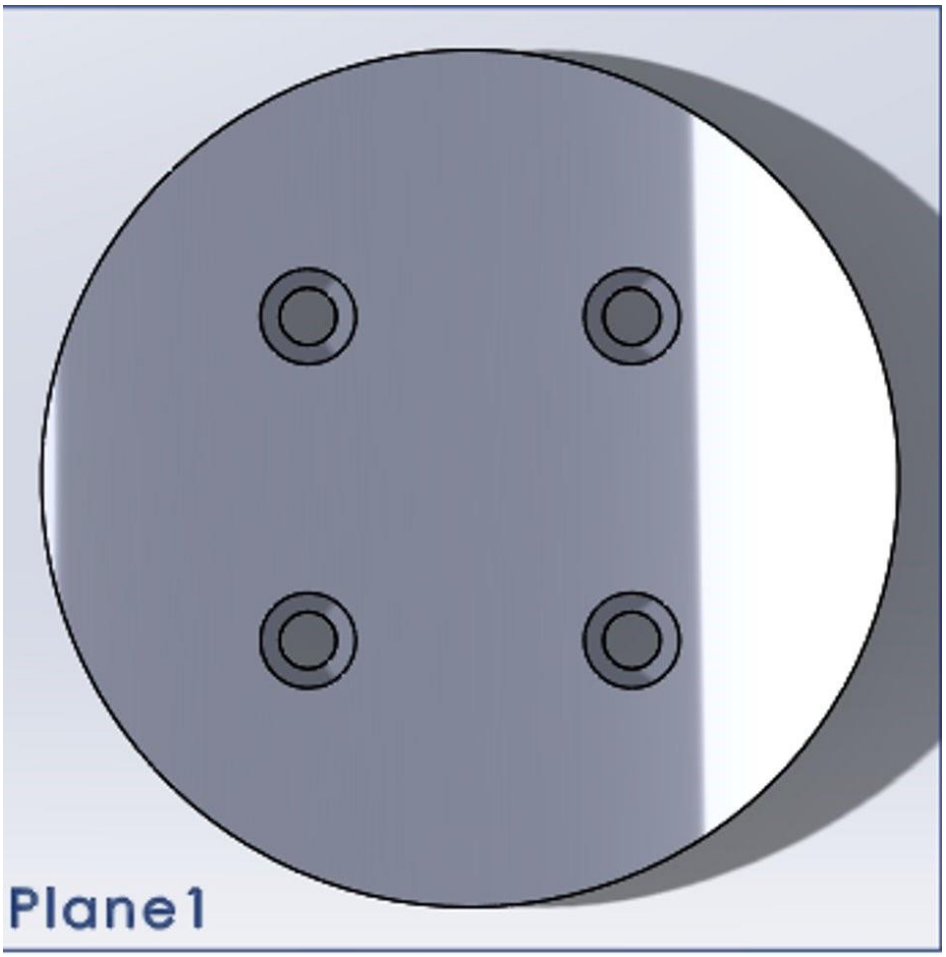
Figur 38: brakett til lysfeste

Indre diameter på 34mm og en ytre diameter på 50mm. Braketten har en krumning med radius på 55mm.

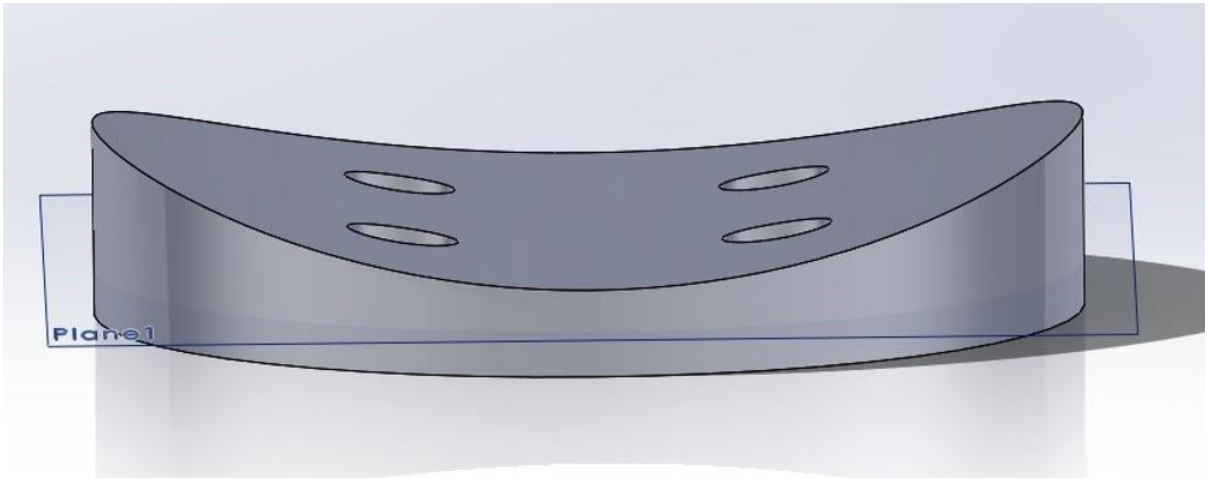


Figur 39: Feste til 10 lysdioder i et symmetrisk oppsett som skaper jevnt lys for kameraet.

Holderen for diodene er sett bakfra og har kanaler til kabler mellom hullene som måler 2mm i diameter. Diameteren på de 10 hullene er 5mm og avstanden mellom hullene er 13,29mm. Indre diameter for ringen er 34mm og ytre er 50mm. Høyden på ringen er 10mm.

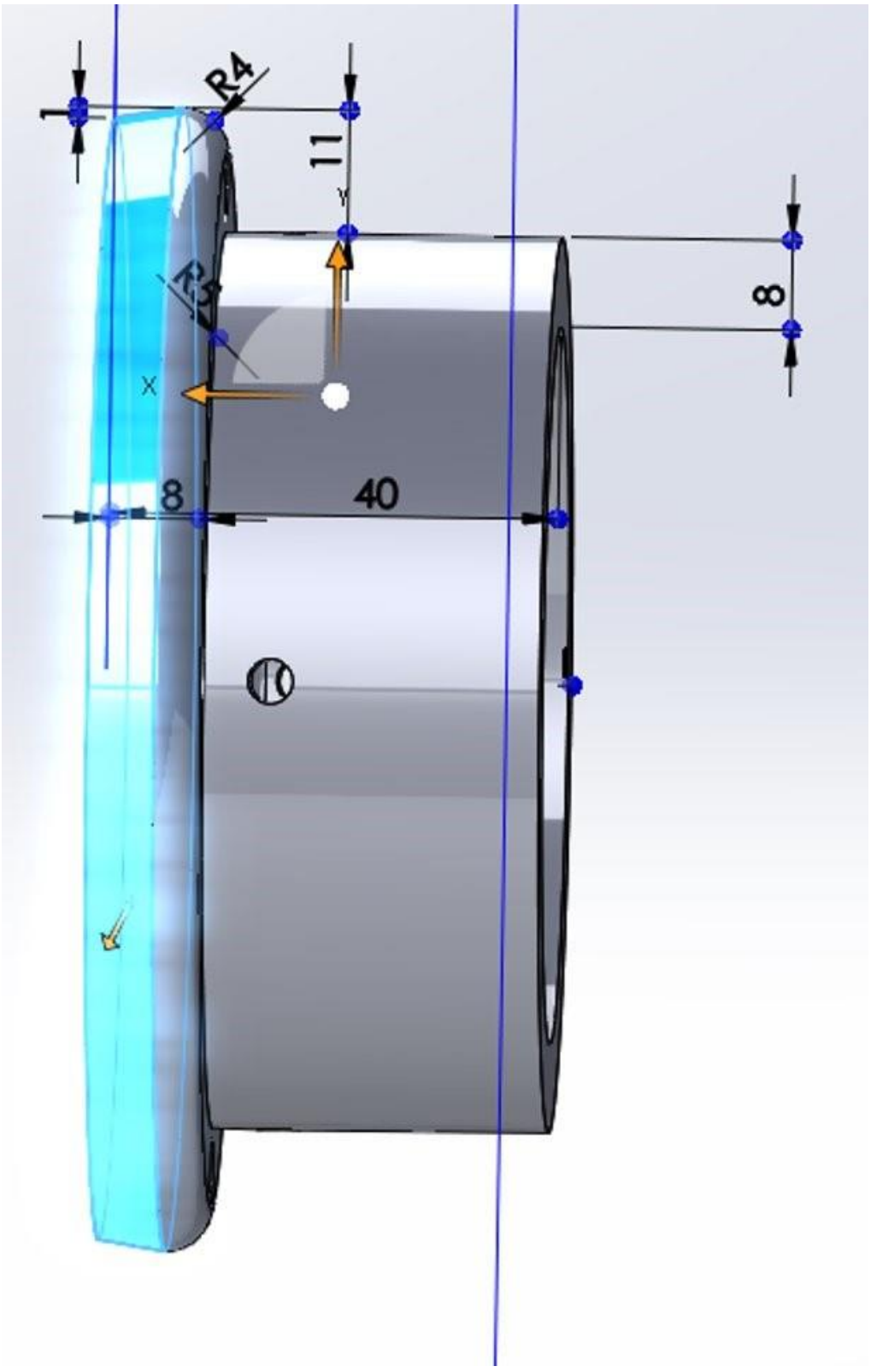


Figur 40: feste til sonar sett ovenfra



Figur 41: feste til sonar sett fra siden

Siden røret er rundt med en ytre diameter på 110mm ble en brakett som kobler sylinderen til den flate runde sonaren. Diameter på braketten er 45mm. Minste tykkelse er satt til 4mm for å få plass til nedsenkbare skruer som fester sonaren til braketten. Hullene har en avstand på 17mm senter til senter i et kvadrat om senter av braketten. Hullene er 3mm i bunnen og 5mm øvre del. Tykkelsen av nedre del er 2mm.



Figur 42: feste til laser og kamera sett fra siden med faktiske mål

Figur 43: holderen sett bakfra uten bakplaten

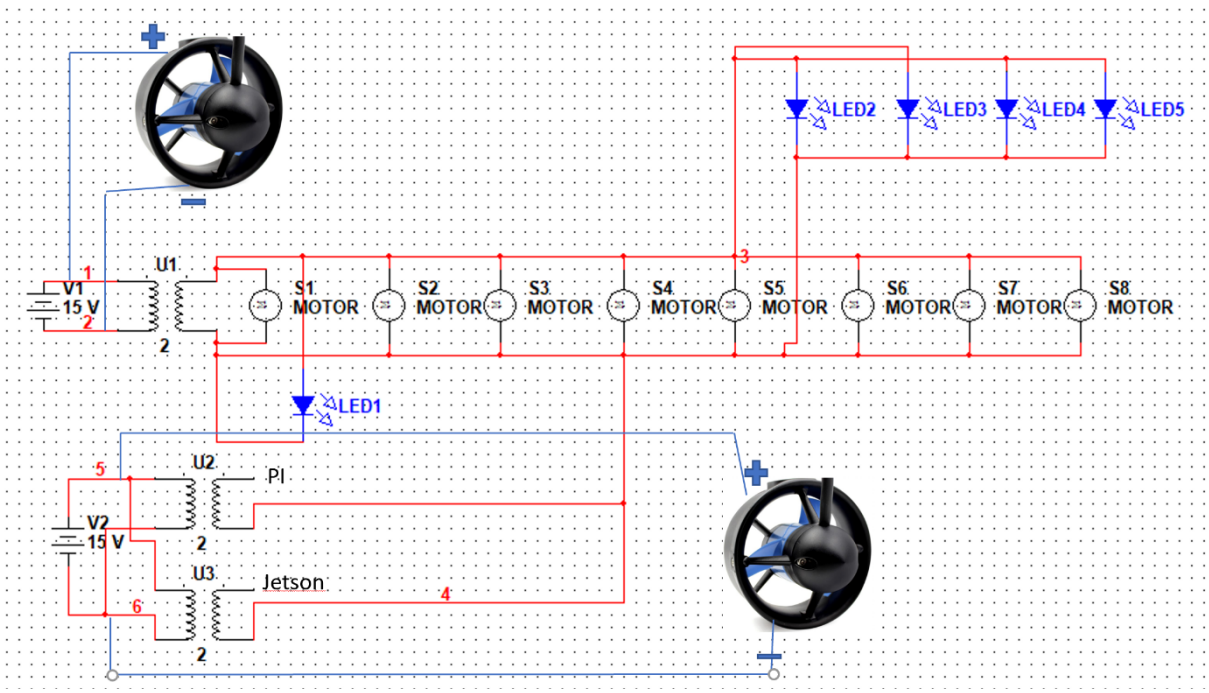
Ytre diameter på 106mm med avrunde hjørner for bedre pres-fit. De fire hullene til laserne er 5,8mm. Disse ble boret ut til 6mm for hånd i etterkant. På fremsiden er kameraet sentrert i midten av alle fire laserene og kabelen til kameraet trekkes ut på baksiden og inn i kroppen på dronen hvor den blir videre koblet til Jetson nano. I sidene på rørstykket som kobler sammen bakre og fremre del av festet for laserene finnes 5mm hull som er tiltenkt kabler til laserene. Disse blir trukket sammen med kabelen for kameraet og festet direkte til strømkilde. Fremre del og bakre del er av lik diameter og av samme avrunde form.

2.7 Elektronikk

Elektronikken om bord baserer seg på strøm fra to batterier som driver all elektronikk. I utgangspunktet er systemet delt i to hoveddeler: 5V og 15V. Der 15V er direkte knytter sammen batteri og fremdriftsmotorene. Resterende systemer går på 5V som blir levert av 3 DC/DC omformere. For sikker strømforsyning til Raspberry Pi og Jetson Nano har disse en egen omformer til hver av enhetene da Jetson Nano har et maksimalt strømforbruk på 2,5A. Dette kunne derfor føre til at Raspberry Pi ikke fikk nok effekt til å drive seg selv om Arduinoen som er koblet videre fra Raspberry Pi. Den siste omformeren brukes til å drive resterende enheter, det innebærer 8 servomotorer, fire røde lasere og 10 lysdioder. Denne omformeren kan levere 10 amper, som skal være tilstrekkelig for å drifte enhetene som er koblet til.

Et annet konsept som er viktig i et strømfordelingssystem er felles jord. Felles jord er avgjørende for å kunne ha et felles referansepunkt som alle enhetene tar utgangspunkt i. Bildet under viser at jord på omformer U2 og U3 er koblet sammen med jorden på U1. Reelt er alle jordingene koblet utenfor arduinoen sin dedikerte jord da dette gir en merkonstant referanse uten å gå gjennom andre enheter.

,



Figur 44: figuren viser strømfordelingen som skjer fra batteriene og ut til de forskjellige komponentene. LED1 er 10 dioder i parallell. LED2-5 representerer laserne i front. S1-8 representerer servomotorene i parallell og U1-3 er de forskjellige DC/DC transformatorene. Motorene er koblet rett på batteriene gjennom en motorkontroller som er en del av motorene. Utgangene merket PI og Jetson er 5V utgang rett til Jetson Nano og Raspberry Pi.

2.7.1 Effektberegninger

På bakgrunn av kravene som ble satt i starten av prosjektet måtte det gjøres en effektberegning for alle delkomponenter slik at vi er sikre på at vi kan nå målet som er:

Krav 5: *«Dronen skal kunne ha en god nok operasjonstid til å demonstrere at den oppnår hovedmålet, anslagsvis et sted mellom 30 min og én time»*.

komponent	antall	spenning (V)	strøm (A)	effekt (W)	
T200 elektromotor	2	15	0,1	3	
Stor omformer	1	15		7,0818	effekttap på 29%
liten omformer	2			7,25	effekttap på 29%
lysdioder	10	3,3	0,04	1,32	
laser rød	4	5	0,13	2,6	
sonar	1	5	0,1	0,5	
Raspberry Pi	1	5	0,5	2,5	
Jetson Nano	1	5	2	10	
servomotorer	8	5	0,5	20	
arduino Mega	1	5	2,5	12,5	
total				63,7518	W
total effekt batteri(80%)				236	Wh
dykketid				3,70185626	timer

Figur 45: figuren viser effektberegninger på bakgrunn av målte verdier og verdier fra datablad.

Effektberegningen som er vist i figur 46 er satt på bakgrunn av målte verdier og dokumenterte verdier. Da det kommer til Servomotorene kan disse trekke mellom 0,1A og 2,5A ut ifra kraften de må produsere. Sett ut i fra den kraften servomotorene må produsere for å rotere dronen vil de i snitt kun bruke 0,5A per motor. Effekttapet som er forventet av omformerene er på 29% som beregnes ut i fra hvilke komponenter som er koblet opp til utgangen. Resultatet tilsier en dykketid på 3,7 timer ved bruk av 80% effekt, noe som er nødvendig å regne med når det kommer til LiPo batterier siden cellene ikke klarer å lade uten høy nok basestrøm.

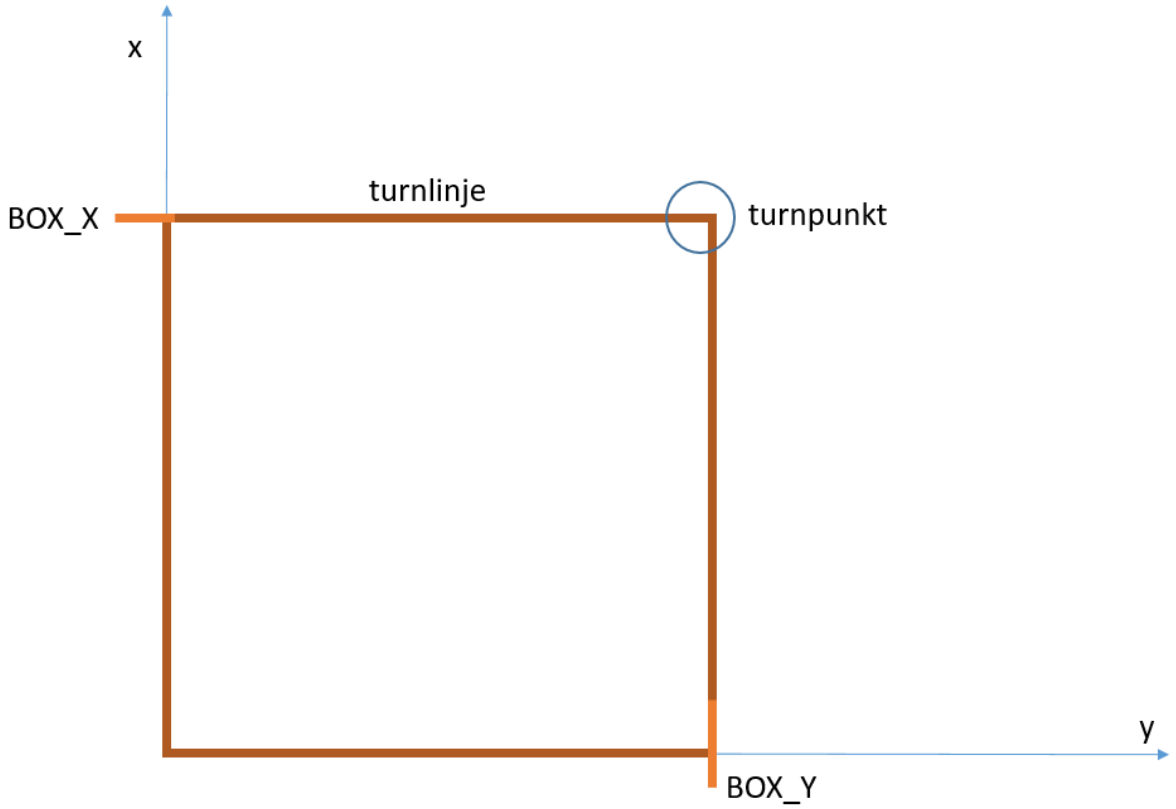
I forhold til kravet leverer dronen langt over forventet. Dette skyldes bruken av langt større batterier i forhold til hva som var nødvendig for dette prosjektet. Grunnlaget for denne overdimensjoneringen var en tanke om at en videreutvikling med ekstra sensorer trenger en vesentlig større mengde energi, for eksempel sonarer. Det var også tiltenkt at dronen skulle dimensjoneres til høyere hastigheter en nåværende hastighet som er satt til kun 10% av maks effekt.

2.8 Styringssystemet

For å tilfredsstille krav 4 om et reaktivt styringssystem, må et system kunne bearbeide informasjonen for å kunne gi ut rett styring for dronen. For å gjøre det så simpelt som mulig,

er det utviklet en funksjon vi kaller for BOX. For å understreke vil denne funksjonen være veldig avgrenset i hva den vil klare å gjøre.

Funksjonen tar inn to parametere x og y fra brukeren, og utvikler en firkant som er kjøremønsteret til dronen. Vi definerer turnpunkt som punktet der dronen skal turne og turnlinje som linjen vi følger når vi kjører.



Figur 46: Utsnitt av kjøremønster til dronen, med begrensning i x- og y-retning

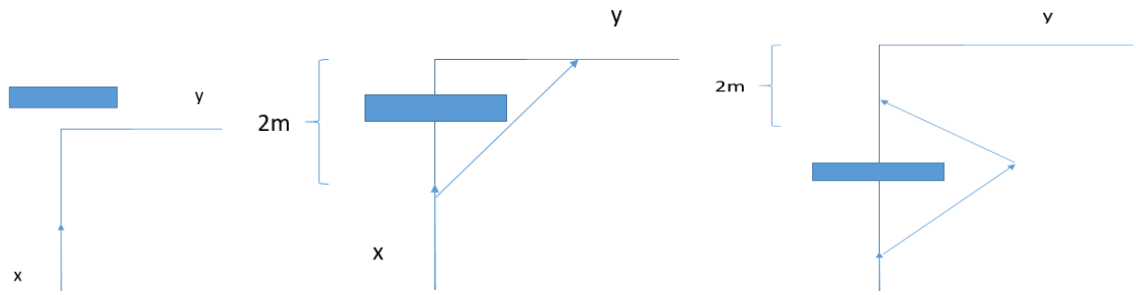
Dronen vil da følge kjøremønsteret og komme tilbake til startpunkt. En av de nevnte avgrensningene er at man ikke søker gjennom området i midten av boksen, men kun får sett og undersøkt langs turnlinjene. Det gjør ikke dronen veldig effektiv til å bidra til for eksempel søk. Men, vi har definert kravet slik vi har gjort med bakgrunn i hovedmålet i den hensikt å simplifisere så mye som mulig. Det er beskrevet at oppgaven er stor og omfattende, der målet i stor grad handler om å vise funksjonalitet i sin enkleste form for så å muliggjøre videre utvikling. Derfor vil også styringssystemet bli veldig simpelt og avgrenset i den hensikt å fokusere på å vise evne og muligheter.

2.8.1 Reagere på objekter

Skulle dronen oppdage et objekt i veien for kjøremønsteret vil den agere på ulike måter avhengig av avstanden til neste turn:

- Hvis objektet ligger mer enn 50cm *etter* turnpunkt: ignorerer det

- Hvis objektet ligger *mellom* 50 cm etter turnpunkt og 2 meter nærmere: dronen tar en skråturn inn til neste turnlinje
- Objektet ligger *mer* enn 2 meter fra turnpunkt: dronen styrer rundt objektet og inn til samme turnlinje



Figur 47: De tre alternativene, og hvordan dronen er programmert til å agere, hhv mulighet 1, 2 og 3

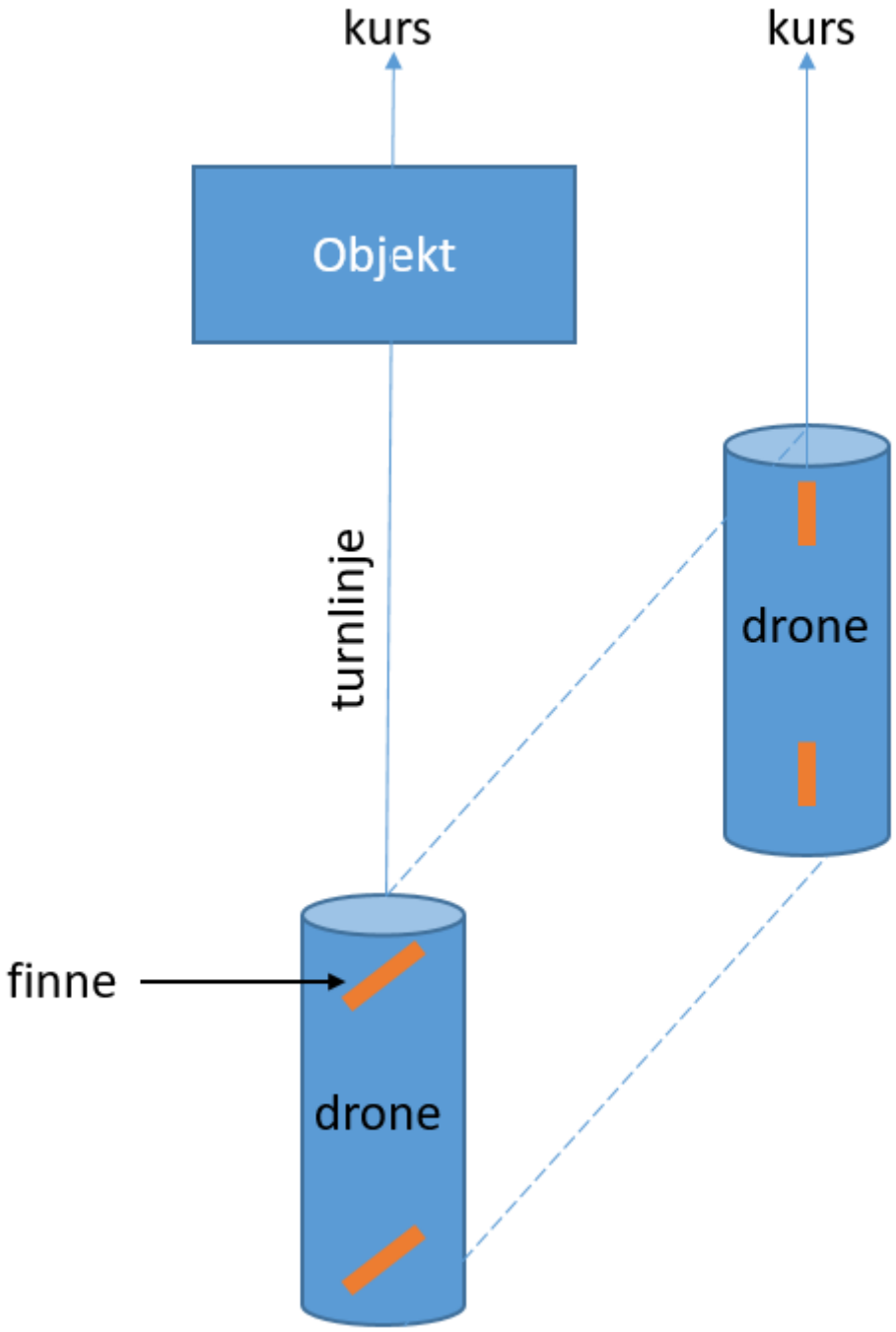
For metode 1 vil ikke hjernen gjøre noe som helst, den vil rett og slett overse objektet og kjøre videre i sin vanlige bane. metode 2 og 3 er ganske like, men skiller seg ut med hva man gjør etter at man befinner seg på siden av objektet. For at programstrukturen skal kunne skille på dem, vil de to metodene gå under to ulike navn. Vi kaller metode 2 for Turn og metode 3 for Rund.

Fra 2.2 var planen å bruke vinkelen mellom dronen og objektet til hjelp. Målet var at dronen skulle lagre vinkelen til objektet i det øyeblikket man ikke lenger så objektet, for så å bruke informasjonen til å bedømme hvilken kurs vi måtte ta for å unngå objektet. Som en konsekvens av at man ikke fikk til en nøyaktig nok beregning av vinklene ble denne ideen skrotet. Men, originaltanken om å benytte informasjon ut ifra når vi mister objektet ble tatt med videre i designet av styringssystemet.

Slik designet på systemet er vil det være hensiktsmessig å definere at all turning vil gå mot styrbord. Vurderingen er at man da sikrer at dronen i minst mulig grad beveger seg utenfor BOX-begrensningen som er satt av bruker. Å bevege seg utenfor begrensningen medfører at en har mindre kontroll på dronen. Samtidig vil en slik avgrensning i mulighetsrommet til dronen bidra til å svekke robustheten. Skulle en for eksempel møte på et objekt som er avlangt langs styrbord side vil det på ingen måte være hensiktsmessig å turne samme vei. Videre vil eventuelle møter med havbunn, der eneste mulighet for turn er oppover gjøre dronen lite kapabel.

Basert på avgrensningen om å kun gå styrbord, kan en nå definere funksjoner for Turn og Rund. Vi starter med å designe likhetene for de to. Vi ønsker at systemet skal fungere slik at i det den oppdager et objekt nærmere enn 360cm, så skal den lagre avstanden til objektet og starte med å forskyve seg styrbord over. Når dronen så ikke lenger ser objektet skal den

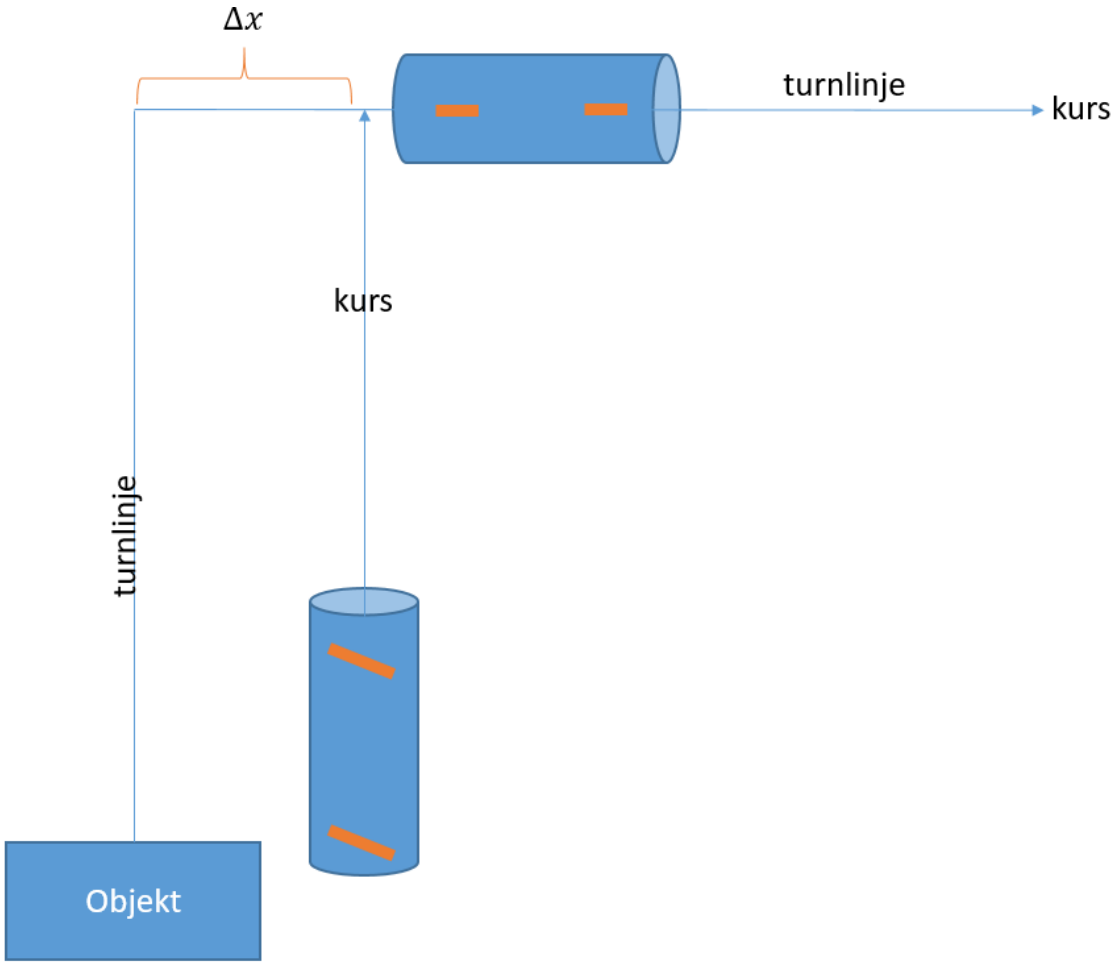
slutte med forskyvningen og gå tilbake til normal styring, helt til den er på linje med objektet. Siden man har lagret avstanden til objektet i det vi oppdaget det, så vet dronen når den er på linje. I utgangspunktet var tanken at den bare skulle turne mot styrbord, og ikke forskyve seg bortover. Men med bakgrunn i hvordan posisjonsmåleren fungerer, fungerer den optimalt ved å forskyve dronen bortover da det gir bedre kontroll for posisjon. En er derfor ikke avhengig av vinkelen slik man i utgangspunktet hadde tenkt.



Figur 48: Illustrasjon av fellesdelen for Turn og Rund for hvordan dronen skal unngå objekter

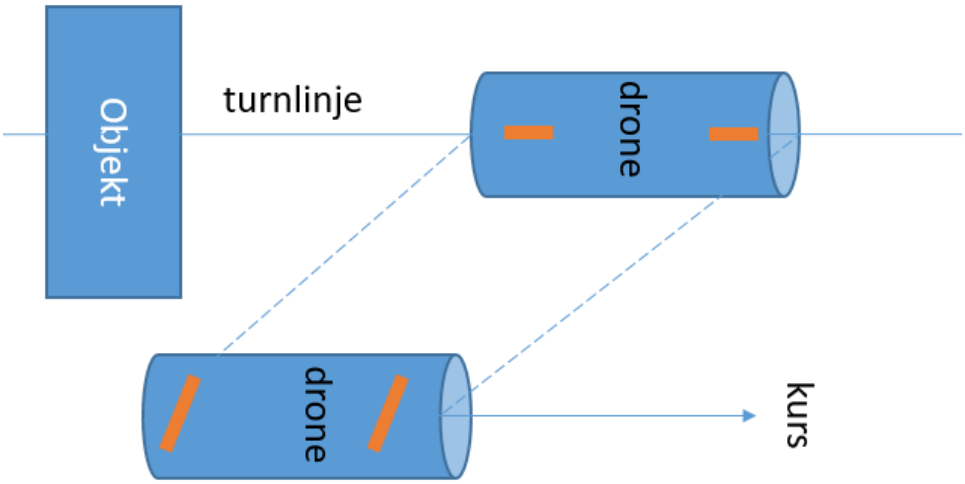
Det som skiller dem fra hverandre er avstanden til neste turnlinje og hva man skal gjøre når man er på linje med objektet.

For Turn er det ansett å ikke være god nok plass til å komme seg tilbake på samme turnlinje, slik at dronen fortsetter på nåværende kurs helt til en når neste turnlinje. Det er her viktig at man er klar over at dronen når turnlinjen på en annen plass enn i turnpunktet, slik at differansen må beregnes med i posisjonen. Dronen vil derfor fortsette på samme kurs når den er på linje med objektet, og avvente nye endringer til den treffer neste turnlinje:



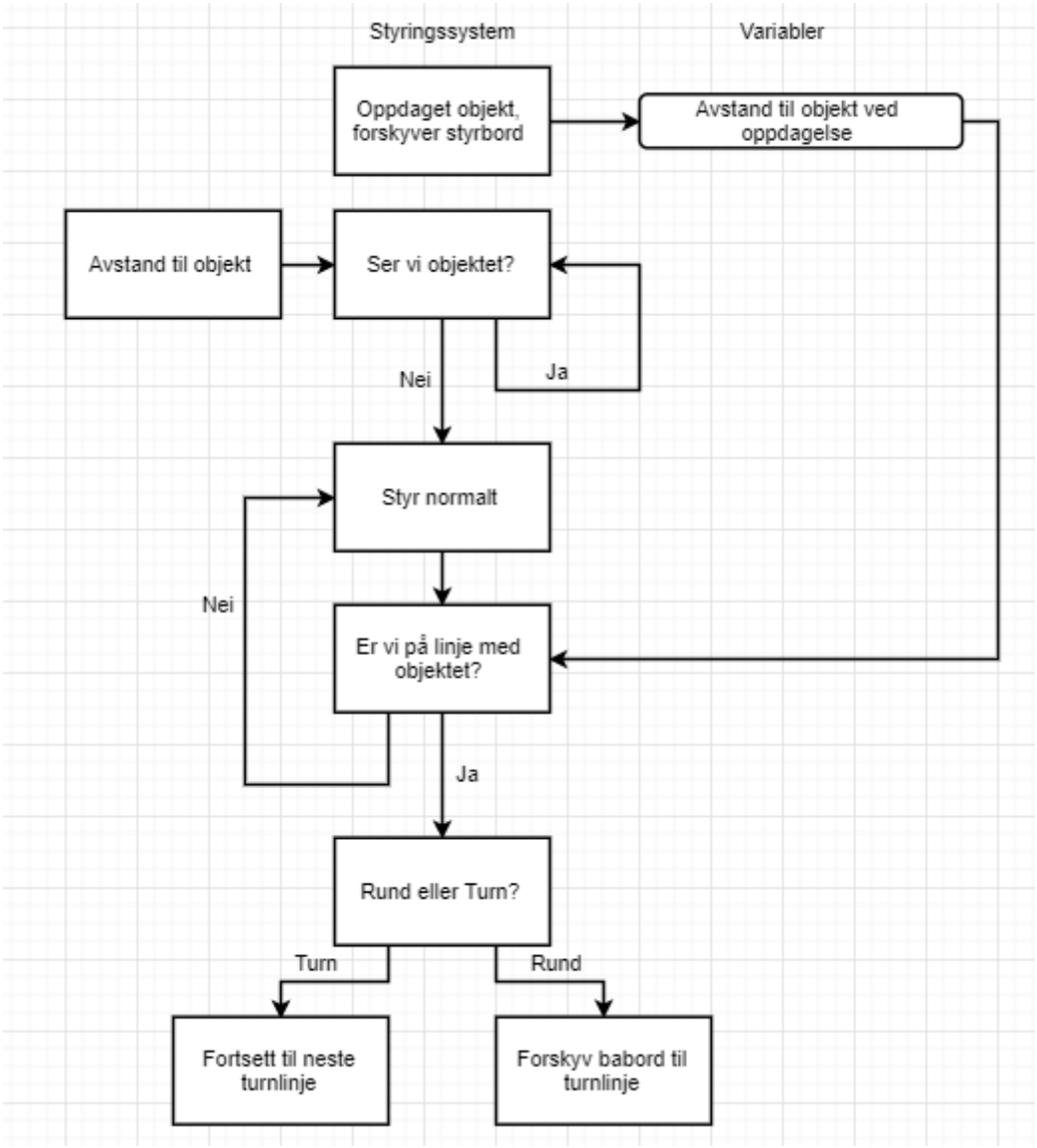
Figur 49: Eksempel på hva Turn-funksjonen gjør etter at den har passert objektet, merk differansen i posisjon dronen må beregne med

For Rund forskyver dronen babord til den når den samme turnlinjen igjen. Når man når den skal dronen fortsette på kursen man hadde i utgangspunktet før dronen så objektet. Her benytter man posisjonen i y for å bedømme når en har nådd turnlinjen igjen.



Figur 50: Eksempel på hva Rund-funksjonen gjør etter at den har passert objektet. Når den når turnlinjen fortsetter den på samme kurs

Hvis man fører alt inn i et blokkskjema ser det slik ut:



Figur 51: Prosedyren til styringssystemet når et nytt objekt oppdages. Laget hos: https://www.draw.io/

2.8.2 Utfordringer

Det er en utfordring med hvilke typer objekter man kan unngå med dronen. Her finnes det muligheter man kunne ha videreutviklet men som vi valgte å ikke gjøre grunnet tid. Når man prøver å unngå objektet og merker at det ikke gir utfallet en ønsker med at objektet enda er der, er det en god indikasjon på at det å turne styrbord ikke er den beste løsningen. Skulle

en slik turn vise seg å ikke være nok til å komme rundt objektet kan dronen dykke under eller stige over objektet. Vurderingen på hva den skal gjøre kan avhenge av data fra sonaren. Hvis man har liten avstand til bunnen kan det være smartere å stige over objektet kontra hvis en har god avstand til bunn kan det antas at det er et flytende objekt og ikke en del av bunnen. Etter å ha nådd en god nok dybde til å unngå objektet kan den fortsette som normalt, bare at den må dykke oppover når den skal utføre Rund eller Turn.

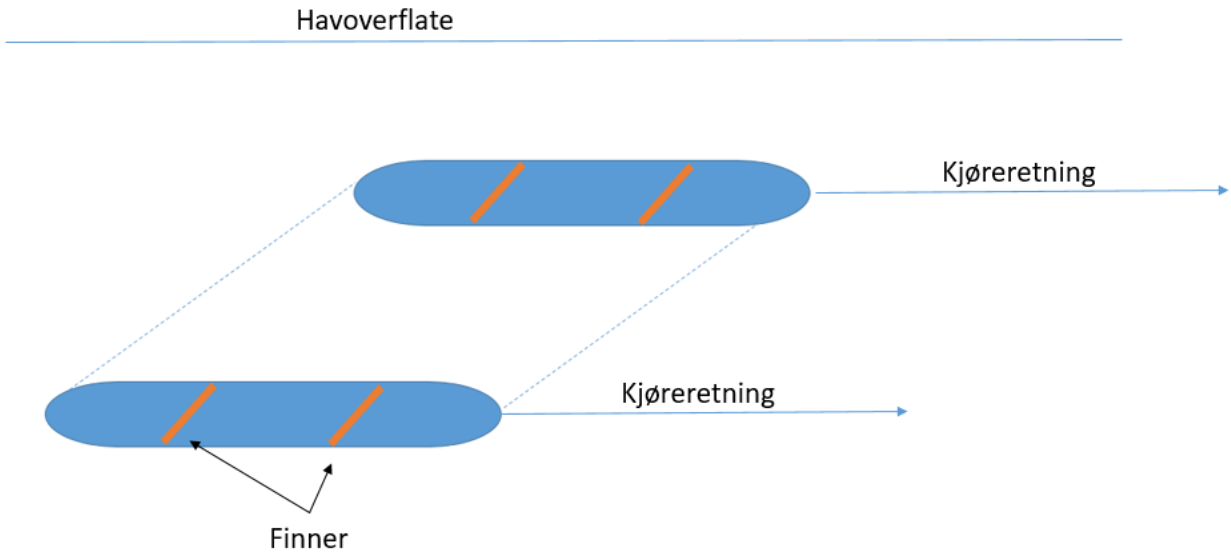
Det finnes flere eventualiteter man ikke tar høyde for, som i at det finnes objekter *bak* det første objektet en ser. Dette vil sørge for kollisjon mest sannsynlig, og gjør igjen dronen lite robust. Hadde dronen hatt for eksempel thrustere kunne den turnet i ro-stilling tilbake mot det første objektet, og avsluttet turn i det den oppdaget et objekt. Slik ville man unngått kollisjon og kun hatt bruk for små endringer.

2.8.3 Fysisk styring

Her brukes ordet *Servo* for å beskrive klaffene/finnene som gir rotasjon om aksene for dronen.

Det er 8 servoer knyttet til dronen. 4 av dem styrer den horisontale kursen(yaw), mens de 4 andre styrer pitch og roll samt dybde. Det er komplekst å drive rotasjon om alle aksene. En av utfordringene ser man ved yaw der sensoren kan gå fra å vise 359 til 0 grader, noe som vil gjøre ting vanskelig for systemet å tolke. Skal man så ha dette for alle tre aksene vil det by på vanskeligheter. Det settes et kriteria om at dronen skal ha 0 grader i pitch og roll til enhver tid. Dette er også et nødvendig kriteria for posisjonsmåleren til Pi da den beregner seg på at kamera står vinkelrett til bunnen, og avvik fra dette vil føre til en feilkilde i posisjon. En utfordring er oppstigning og neddykk når man ikke kan basere seg på å rotere dronen med nesen oppover slik vanlige ubåter dykker opp. Dermed må hele dronen stige likt, altså at bakenden og forenden dykker opp med samme stigning.

Men selv om man beveger seg i z-retning kan dronen få en påvirkning som endrer pitch og roll som den må kunne agere på. Løsningen baserer seg på at de ulike servoene må brukes til å løse flere oppgaver på en gang. Dette krever litt mer tenkning, men det gjør også at dronen er fleksibel for påvirkning som gir utslag i mer enn bare én akse. Dette samstemmer mer med virkeligheten også da det sjeldent er en ren endring i en akse når man får påvirkninger fra for eksempel strømninger.



Figur 52: Illustrasjon av hvordan dronen skal stige/dykke, sett fra siden av dronen

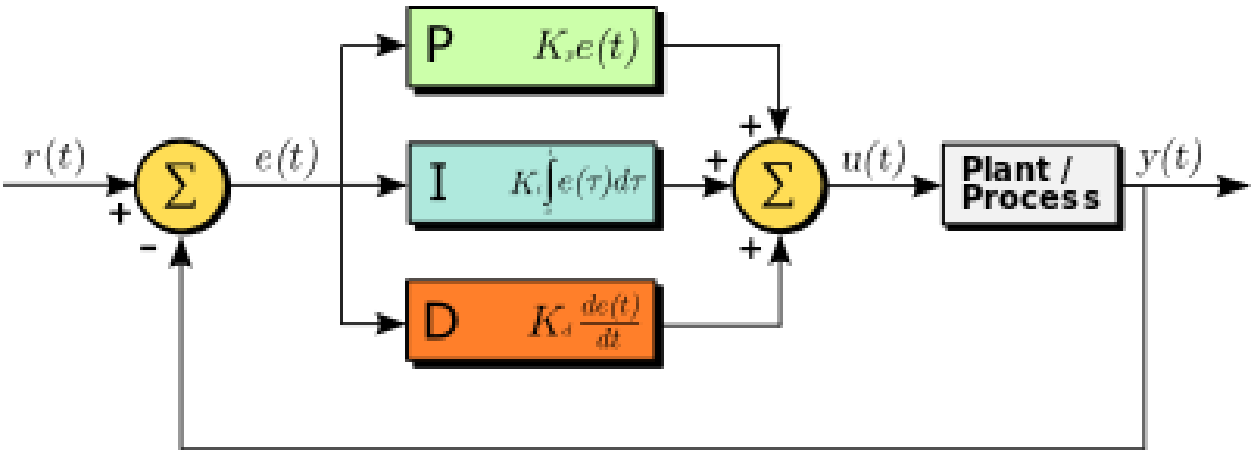
For servoene som styrer yaw skal alle kun styre dronen i forhold til rotasjon om z-aksen, og baserer seg på input fra data om yaw. Men for servoene på siden av dronen vil man måtte håndtere endringer i både pitch og roll, samt dybden. For roll vil en påvirkning kunne ageres på ved at to finner/servoer på samme side gjør en endring for å kompensere. Man kan også ha at de to sidene styrbord og babord agerer i motsatt retning av hverandre, noe som vil gi en raskere kompensasjon. Men for at ting skal holdes mest mulig simpelt velges kun en av sidene til å kompensere for roll. For pitch vil det være enten de to fremme eller de to bak som kan gjøre en endring for å kompensere. For dybde har man kriteriet om lik stigning foran og bak på dronen. Dermed trenger man at samtlige 4 servoer har likt pådrag for å dykke.

Pitch styres av de fremste servoene, roll styres av de babord og alle 4 styrer dybde. Eksempelvis vil PR_BBF ha 3 jobber mens PR_SBB vil bare ha én jobb. En utfordring med montering er at servoene som står på hver sin side av hverandre vil ha utslag motsatt vei av den andre. Servoene må derfor inverteres i forhold til hverandre.

PID-kontroller

Man trenger en PID-kontroller for pitch,roll og yaw. PID står for Proporsjonal-Integral-Derivasjon og beskriver en metode for å kompensere for forstyrrelser. PID-kontrollerne sin jobb er å etterjustere for feil i styringen som følge av ytre påvirkninger. Det betyr at sensorene måler eventuelle feil i pitch, roll og yaw i forhold til hva man har satt dem til å være. PID vil så kalkulere ut et pådrag for å kompensere for påvirkningen. Samtlige PID'er skal være Reversvirkende, som betyr at ved en uønsket økning i input vil føre til negativ output. PID-kontrolleren kommer fra et ferdig bibliotek på lik linje med servostyringen. Det

man trenger er å stille inn konstantene rett slik at en evner å reagere raskt på endringene men samtidig ikke agerer på bittesmå eller støyete målinger.



Figur 53: Skjematikken for PID-kontroller. Kilde: https://en.wikipedia.org/wiki/PID_controller

2.8.3 Sammendrag

Da styringssystemet har blitt avgrenset ganske kraftig vil det ikke fullføre noen form for direkte relevant oppgave. I henhold til kravet om å være reaktivt vil det bare bli delvis tilfredsstilt da dronen evner å reagere på objekter og unngå dem, men har igjen flere hendelsesforløp og eventualiteter den ikke er forberedt på. Samtidig er det få muligheter som fører til kollisjon slik systemet er designet, men forblir likevel en utfordring som må bygges videre på.

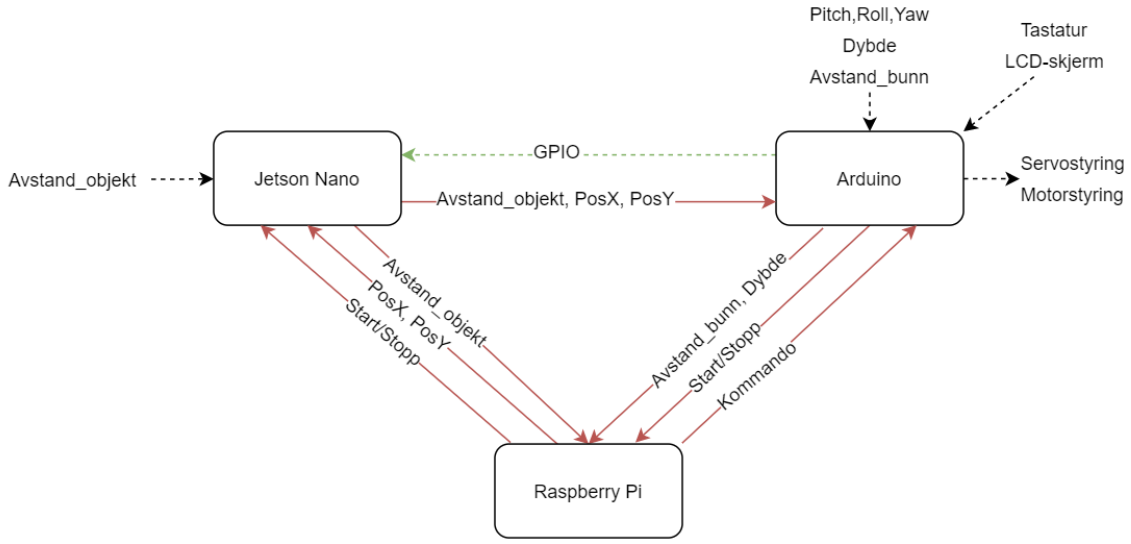
Oppsettet for dronen vil bestå av 8 servoer som gjør ulike oppgaver avhengig av sensordataen vi mottar. Disse vil bli bearbeidet av en PID-kontroller før det gis videre føringer til servoene. Selv om man trenger dybde og kurs fra Pi, vil all kontroll og bearbeiding av forstyrrelser skje lokalt og ikke være avhengig av styringssystemet.

3.Utviklingen mer detaljert

I denne delen om utvikling vil det gå mer detaljert inn på utviklingen program- og kodemessig. Først vil en ta for oss det samlede Operativsystemet, som innebefatter alle elektroniske komponenter som bidrar til helsystemet. Her vil også felles utfordringer og vurderinger tas med som er viktige. Etter hele systemet kommer hver maskin med en mer detaljert forklaring til hvilke sensorer de har og hva maskinen separat og i samhandling gjør.

3.1 Operativsystemet

3.1.1 Oversikt



Figur 54: Oversikt over operativsystemet. Laget hos: https://www.draw.io/

Stiplet linje indikerer uthenting/sending av data fra påkoblede sensorer/enheter. Røde, heltrukne linjer indikerer serielloverføring av data/kommandoer. Grønn, stiplet linje indikerer en sending av digital høy/lav.

Tegningen over viser oversikten over hvordan operativsystemet til dronen er satt opp grovt. Det vil i løpet av dette kapitlet bli gjennomgått hver og en av delene på tegningen for å skape oversikt. Etter dette vil kommunikasjon mellom enhetene forklares, før hvert enkelt system blir mer forklart i detalj.

3.1.2 Kort om de tre maskinene

Raspberry Pi (Pi) er en ettkortsmaskin som fungerer på mange måter som en meget enkel datamaskin i liten størrelse. Som en konsekvens er prosessorkraften lav og lagringsplassen minimal, men den vil allikevel ha nok kraft til å gjøre oppgavene vi trenger. Dens oppgave er å beregne posisjon i x og y, samt å fungere som hjernen i operativsystemet.

Jetson Nano(Jetson) er en ettkortsmaskin som på mange måter baserer seg på designet til Raspberry Pi, bare at den har en mye kraftigere prosessor. Som en følge er den noe større og trekker mer strøm. Dens oppgave er å beregne avstand til objekter som vi lagrer i variabelen Avstand_objekt.

Arduino er en mikrokontroller som kan laste opp en kode som går i en kontinuerlig sløyfe. Dens oppgave er å styre alt av fremdrift, samt behandle data fra sensorene som gir pitch/roll/yaw, dybde og avstand til bunn som lagres i variabelen Avstand_bunn. Den har også tilkoblet en enkel HMI som består av en LCD-skjerm og et tastatur for brukerinputs.

3.1.3 Utvikling

Måten operativsystemet henger sammen er ved at alt kontrolleres fra Pi. Det er den som angir styreretning og ønskede aksjoner basert på hvilken data den mottar fra de andre. Det betyr at all form for relevant sensordata skal inn til Pi for bearbeiding. Her må man vurdere konsekvensene med at Pi er den som henter ut posisjonsdata. I korte trekk brukes det to kameraer, og både Pi og Jetson har kun én inngang hver, slik at man har en fysisk begrensning som gjør at posisjonsdataen ikke kan flyttes til Jetson.

Arduino mottar posisjonsdata slik vi har designet styringssystemet. Det er en fordel å unngå at Pi skal sende to ulike typer data, herav både kommandoer og sensordata til Arduino, da det kan enkelt skape forstyrrelser. Da det i oppsettet til styringssystemet er kritisk at Arduino mottar korrekte kommandoer fra Pi, vil man ha en verifiseringsprosess som innebærer at Arduino sender mottatt kommando i retur. En slik prosess fungerer optimalt når man ikke samtidig sender sensordata fra Pi. Videre vil det kreve mye koding å legge inn en prioriteringsgang på meldingene for å få gjennom kommandoene når det trengs. Løsningen er at posisjonsdata sendes via Jetson til Arduino. Nedsiden er naturlig nok hastighet da dataen må via et ekstra ledd, og at dataen kan stoppe opp i Jetson. Men fordelen med å frigi meldingsutveksling mellom Pi og Arduino ses på som en større fordel enn ulempen med å kunne miste litt hastighet.

En annen ulempe som dukket opp var at Arduino ikke klarte å sende data til Jetson, selv om andre veien gikk helt fint. Dette ga problemer sett opp mot valget vi tok om å unngå mye trafikk imellom Arduino og Pi, da man ikke kunne sende relevant sensordata fra Arduino via Jetson til Pi. Løsningen blir å opprette et kø-system i koden til Arduino. Siden Pi kun skal sende kommando til Arduino og heller sender sensordata via Jetson, trenger en fortsatt ikke noe etablert køsystem hos Pi.

Køsystemet hos Arduino baserer seg på at verifisering av kommando-data har høyeste prioritering over sending av sensordata. Dette kan være utfordrende med at man får «hull» i sendingen av sensordata til Pi. Basert på styringssystemet vil dybden endre seg veldig lite over tid, slik at en ikke forventer store endringer i de periodene man ikke sender dybdedata. Derimot er det større risiko knyttet til sonaren som skal måle avstand til bunn, der en kan ha drastiske endringer i bunnforholdene. Slik styringssystemet er bygget opp betyr dette at manglende oppdatering på bunnforholdene vil kunne skape feil i posisjonsmålingen, som

igjen gir en følgefeil i styringen. Risikoen for dette ligger i hvor godt Arduino evner å tolke og sende i retur kommandodata. Evner Arduino å gjøre dette veldig raskt, vil også tiden mellom oppdatering på bunnforholdene bli liten, som igjen minsker sannsynligheten for feil i posisjonsdata. Dette gjør systemet til dels avhengig av at kommunikasjonen imellom Arduino og Pi går raskt nok, som igjen betyr at hindringen i effektiviteten til systemet avhenger av dette.

3.1.4 Beskrivelse av oversikten

Vi begynner øverst til venstre med Avstand_objekt. Denne informasjonen skal både sendes til Pi som all data skal, men man sender den og direkte til Arduino for at den skal kunne agere på egenhånd skulle en komme alt for nært et objekt. I samme datapakke som Avstand_objekt sender Jetson også posisjonsdata som den får fra Pi. Når Arduino er klar for ny melding, sender den logisk høy til Jetson slik at den kan sende ny data.

Arduino styrer all fremdrift og holder flere sensorer med viktig data. Det relevante for Pi er Avstand_bunn til bruk for posisjonsberegning og dybde for plotting av posisjon i z-retning.. Dette er i motsetning til pitch, roll og yaw som kun brukes til styringssystemet på Arduino, og ikke behøves av Pi. Arduino får også data fra vår HMI. HMI'en består av en LCD-skjerm som viser en meny og et tastatur der man inngir kommandoer fra menyvalget. De fleste kommandovalg er innstillinger som løses av Arduino helt på egenhånd og man trenger ikke dele det med de andre enhetene. Derimot er det noen av kommandoene som trenger å sendes til Pi, herav start og stopp av dronen og begrensninger brukeren ønsker å sette i styringssystemet.

Fra Pi må man sende ut kommandoen til Arduino slik at styringssystemet vet hvilken retning det skal styre og hvilken dybde å holde. Ellers vil Pi hente ut posisjonsdata lagret i variblene PosX og PosY som sendes til Arduino via Jetson. Grunnet problem med sending fra Arduino til Jetson må Pi videresende start/stopp til Jetson.

3.1.5 Kommunikasjon mellom enhetene

Protokoll for sending og mottaking av data

Serielloverføring innebærer at man sender én og én byte til motparten. Det er nødvendig med en form for protokoll som begge parter forstår og skjønner, slik at vi kan sende og motta data sømløst. Løsningen er at hver melding som sendes skal ha en karakter som indikerer start og en annen som indikerer slutt. Slik vil mottaker vite når den starter og når den slutter, og unngår korrupte meldinger og feil data. Slik kan man også robustgjøre sendingene våre enda mer for å bryte sendinger eller overføringer som ikke stemmer.

Protokollen som gjelder for samtlige er at startkarakteren er ‘<’ og at sluttkarakteren er ‘>’. Dette er karakterer som aldri kommer til sendes i form av data, kommandoer og lignende. Derfor passer disse utmerket, som innebærer at all data som skal sendes må «pakkes» inn i mellom «<>». Et eksempel er hvis man skal sende dataen 340.5 vil en sende den som «<340.5>» til motparten.

Kodemessig for mottaker vil en vente på ‘<’ før man begynner å lagre dataen som kommer via seriell. En vil så lese denne helt til man mottar ‘>’, som betyr at en har mottatt hele meldingen. Programmene må ha kriterier liggende inne for hva som skjer hvis det går for lang tid før den mottar sluttkarakteren. Å sette disse grensene kan vise seg å være vanskelige. Hvis man skal sette en størrelsesbegrensning på antall karakterer lest inn, må en også vite maksgrensen for hvor mange karakterer avsender kommer til å sende. Samtidig er det viktig å tenke på at en slik avgrensning også kan begrense dronens operasjonsevne. Skulle en for eksempel sende posisjonsdata med en mottaker som begrenser seg til 6 karakterer, har man effektivt begrenset kjørelengden til dronen.

Hastighetsavgrensning

Man har tre ulike datamaskiner som alle jobber i ulikt tempo. Hastigheten hver og en av dem jobber avhenger både av fysiske begrensninger men og om kodestørrelse. Jetson Nano er raskere enn Pi for eksempel, og vil prosessere ting mye kjappere. Den har samtidig enklere kode enn Pi, da Pi driver med threading men alt dette nevnes i de separate beskrivelsene.

Utfordringen ligger derimot i at man ikke kan sende data vilkårlig uten noen form for trafikk-kontroll imellom maskinene. Hadde samtlige tre brukt samme tid på å gjennomføre en sløyfe ville det kanskje fungert, men man er også avhengige av at dataoverføring midlertidig pauses skulle det oppstå feil eller komme ukorrekte meldinger.

For å konkludere angående hastighetsbegrensning for de tre maskinene. Det er hastigheten på serielloverføringen, både ved fysisk hastighet og maskinenes hurtighet til å tolke dataen som vil avgjøre hvor hurtig dronens samlede operativsystem klarer å jobbe. Jo mer effektiv man klarer å gjøre serielloverføringen, jo raskere vil hele systemet jobbe.

Eksempelvis vil dronens evne til å unngå objekter avhenge av hvor raskt Arduino klarer å motta data om objektet fra Jetson. Dette da Arduino styrer motorer og klaffer/servoer, mens Jetson oppdager objekter.

3.2 Software Arduino

Arduino skal ta seg av følgende oppgaver:

- Sensordata:
    - Pitch, Roll og Yaw

    - Dybde
    - Avstand til bunn
    - GPS
    - Temperatur
- HMI
    - LCD-skjerm som gir output til bruker
    - Tastatur som gir kommandoer til hele systemet
- Styring
    - Servostyring for bevegelse i x,y,z-plan
    - Motorstyring
    - Objektunngåelse
- Kommunikasjon
    - Mottaking av annen sensordata
    - Mottaking av kommandoer
    - Overføring av sensordata

Strukturen tar for seg objektunngåelse til slutt, da den er avhengig av de andre funksjonene og oppgavene for å fungere.

3.2.1 Sensordata

Vi starter med å se på sensordataen. Vi har ulike sensorer som gir oss ulike typer data til å tolke, og forklaringen her blir dermed delt opp i de ulike sensorbrikkene. Man har følgende fysiske sensorer:

- MPU9250+BMP280: 10 Degrees-of-Freedom(DoF) Akselerometer, Gyroskop, Magnetometer og Temperatur
- LSM-303 6 Degrees-of-Freedom(DoF) Akselerometer og Magnetometer
- Ping Sonar Echosounder: Sonar for måling av avstand til bunn
- Bar30 High-resolution Depth Sensor: Måling av dybde under vann
- NEO-6 Series GPS

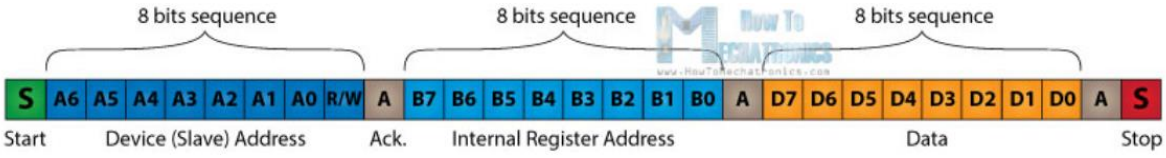
Man kommuniserer med sensorene enten ved bruk av I2C eller Tx/Rx. I2C baserer seg på kommunikasjon mellom en master(Arduino) og en eller flere slaver(sensorer). Den bruker to innganger Serial Data(SDA) og Serial Clock(SCL), der SDA er linjen som sender og mottar data mens SCL er klokken til systemet.

Hver sensor som kobles til vil få en unik adresse som Arduino da kan sende melding til og få svar tilbake. Meldingsgangen er delt opp i segmenter bestående av 8 bits(1 byte). Først i meldingen fra Arduino kommer en 7-bit adresse som spesifiserer hvilken sensor som skal

motta meldingen. Den siste biten brukes til å bestemme om man skal skrive noe til slaven(write) eller om en skal lese ut data fra den(read).

Etter dette vet slaven at meldingen er til den. De neste 8 bitsene spesifiserer *hva* i sensoren man ønsker å skrive til/lese av data. Disse bitsene spesifiserer dermed interne adresser i sensoren, som f.eks i en sensor som har både et gyroskop og et akselerometer vil man spesifisere at det er akselerometerdataen en ønsker å lese av.

Det er nå åpnet for trafikk mellom Arduino og slaver langs SDA-linjen. Veien trafikken går er avhengig av om man har valgt å skrive til eller lese av data. Hvis en for eksempel har bedt om data fra akselerometeret vil sensoren sende kontinuerlige segmenter av 8-bits data til Arduino helt til den er ferdig. Når den er ferdig sender den noe til Arduino slik at den vet at man er ferdig(stopper kommunikasjonen) og kan stenge trafikken.



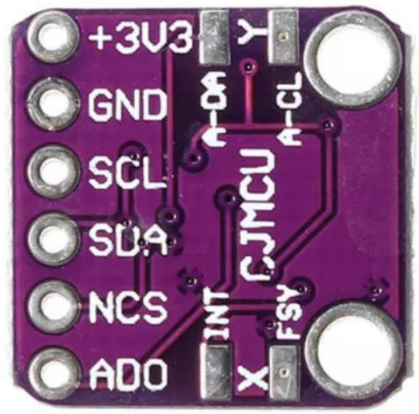
Figur 55: Utsnitt av meldingsgangen mellom master og slave for I2C. Kilde: https://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/

For Tx/Rx er det kun en master og en slave som fungerer på lik måte fra Arduino sin side som seriellkommunikasjon med eksempelvis Pi. Man trenger dermed ikke ha samme form for meldingsgang imellom dem. Siden Tx(sende data) og Rx(motta data) er separate fysiske ledninger kan man ha en mer kontinuerlig overføring av data mellom Arduino og sensor. Det viktige her er at Arduino kan tolke dataen som kommer inn korrekt, men i alle tilfeller en bruker dette er det brukt fabrikklagede biblioteker med de rette funksjonene som tolker dataen for oss.

Pitch, Roll, Yaw og temperatur

MPU9250 baserer seg på I2C, og har 6 innganger herav 5 benyttes: 3v3, GND, SCL,SDA og ADO. 3v3 og GND er strømforsyning til sensoren, som man kobler opp via Arduino. Før det gås igjennom I2C er det viktig å forklare grunnen til at man bruker ADO-inngangen. Hver sensor har som sagt en egen adresse for å kommunisere med den, og i MPU9250 og BMP280 sitt tilfelle er den laget til å være en av to fastsatte adresser.

En utfordring er at dybdesensoren hadde lik adresse som MPU9250, som gjorde at sensoren ikke fungerte. Siden man ikke kan endre adressen på den, må man endre adressen på MPU9250, noe som enkelt gjøres ved å sende logisk høy til ADO.



Figur 56: Utsnitt av MPU9250 + BMP280 brikken. Kilde: https://www.banggood.com/

Over til kommunikasjonen med bruk av SDA og SCL. Grunnet logisk høy på ADO har man adresse 0x69 for MPU9250 og 0x77 for BMP280. Biblioteket Wire.h brukes til å kommunisere ved bruk av I2C. I dette biblioteket har man flere funksjoner en bruker for å sende/motta data fra sensorene, som kan finnes på Arduino's hjemmeside. Det neste som trengs er alle de interne adressene til sensoren, som man finner på nettet i såkalte register. Slik kan en sette de rette innstillingene for sensoren og hente ut dataen.

Rådataen som skal hentes ut er på 16 bits. Siden man kun leser ut 8 bits om gangen må man slå sammen de to segmentene på en måte. Dette gjennomføres ved å først flytte det ene segmentet 8 plasser til venstre og så bruke en logisk eller.

```
          10110011  Første segment          11001100 andre segment
10110011 00000000 Forskyves 8 plasser
10110011 00000000   Logisk eller  11001100
10110011 11001100
```

Figur 57: Eksempel på sammenslåing av to segment

Etterpå må dataen konverteres i forhold til hvilken sensitivitet man har satt. Dette er en egen tabell som finnes i registeret. Ved å benytte et mindre spekter vil man ha bedre sensitivitet, eller flere bits per 9.81 Newton. Dataen man får ut i bits deles på et tall avhengig av hvilken sensitivitet, eksempelvis 16384 som man har for akselerometeret i henhold til tabellen under.

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

Figur 58: Oversikt over bits per g(9,81N) man har for akselerometer. Kilde: https://www.invensense.com/wp-content/uploads/2015/02/RM-MPU-9250A-00-v1.6.pdf

For å summere må man først skrive til sensoren ved å bruke rett adresse, så må en vite hvilken data fra sensoren man ønsker å lese av ved å bruke rett interne adresse. Til slutt må man slå sammen de to segmentene og dele på rett sensitivitetskonstant for å få ut data.

```
void HentData_AKS() {
    Wire.beginTransmission(MPU);
    Wire.write(0x3B);
    Wire.endTransmission(false);
    Wire.requestFrom(MPU, 6, true);
    AccX = (Wire.read() << 8 | Wire.read()) / 16384.0; // For 1000deg/s tilsvarer å dele på 32.8,ref liste
    AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;
    AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;
```

Figur 59: Hvordan uthenting av rådata gjøres rent kodemessig, her vist med uthenting av akselerometer-data

Det brukes akselerometer, gyroskop og magnetometer som sammen skal gi rotasjon i de tre aksene (pitch,roll,yaw). Men først må det gjøres en filtrering av dataen. For akselerometeret må man for det første lage en forskyvning ut ifra det som skal være vårt nullpunkt. Ved å ta 200 målinger og finne gjennomsnittet, kan man så forskyve dataen i forhold til det. Videre har man et lavpassfilter som skal ta bort eventuell støy som akselerometeret kan ha, før en omgjør dataen. Fusjoneringen for pitch og roll består av å bruke 90% av gyrodata og 10 % av akselerometerdata. Dette hindrer drift av gyroskopet samt at man unngår kraftig påvirkning av vibrasjon.

For yaw må det tas i bruk magnetometeret (kompass) til å hjelpe gyroskopet slik at det ikke drifter. Det nyttes to sensorer som begge henter ut den samme informasjonen. Fordelen med å bruke LSM-303 er at den har et ferdig bibliotek som inkluderer tilt-kompensering for magnetometeret. Denne gjør alt av kompensering og kalkulering på egenhånd, og har samtidig vist seg å være mer nøyaktig. Til sammenligning ville man ved bruk av MPU9250 gjort om på tilt-formelen fra 2.4 da magnetometeret har ulik rotasjon i forhold til gyroskopet:

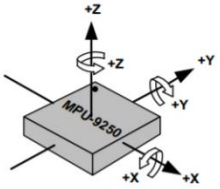


Figure 4. Orientation of Axes of Sensitivity and Polarity of Rotation for Accelerometer and Gyroscope

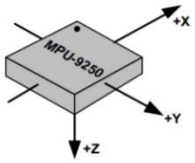


Figure 5. Orientation of Axes of Sensitivity for Compass

Figur 60: Utsnitt fra registeret til MPU. Vi ser at orienteringen til Gyroskop/akselerometer er ulik enn magnetometeret. Kilde:https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

Som innebærer at x og y bytter plass, og at z blir negativ.

Etter at man har en god nok kalibrering kan den tilt-kompenserte funksjonen i biblioteket til LSM-303 brukes. Uten å gå i detaljer baserer den seg på informasjon om pitch og roll til å kompensere med yaw. Denne fungerer meget bra opp til +- 10 grader pitch/roll, som er godt nok basert på at dronen skal holde 0 grader pitch og roll. 10 grader ujevnhet for dronen i disse aksene indikerer at PID-kontrolleren må justeres enn at man må fikse på magnetometeret. Det brukes samme vekting som mellom gyro og akselerometer med hhv 90% gyro og 10% magnetometer data.

Omgjøring til bibliotek for bedre oversikt i koden

En siste ting som ble gjort var å skrive om all prosessen til et bibliotek. Ved bruk av samlede funksjoner for oppsett, omgjøring og filtrering, samt fusjonering blir ting mer oversiktlig i koden.

Dybde og Avstand bunn

For å måle dybde og avstand til bunnen brukes Bluerobotics' dybdesensor og sonar. Sonaren baserer seg på Tx/Rx for kommunikasjon med egne biblioteker for omgjøring av data. Dybdesensoren derimot baserer seg på I2C, men har og biblioteker for uthenting. Det er nevnt at det er en utfordring når i2c-adressene var like for dybdesensor og MPU9250, men det ordnes ved bruk av ADO slik at dybdesensoren forholder seg til biblioteket som følger med.

Figur 61: Bilde av dybdesensor og Ping sonar fra Bluerobotics. Kilde: https://bluerobotics.com/

Det man trenger å tenke på er bruksområdet for sensorene og hvordan en skal tolke dataen fra sonaren. Dybdesensoren vil stå for posisjonsmåling i z-aksen, som betyr at man må få sendt dataen kontinuerlig til Pi. For sonardataen skal den primært understøtte posisjonsmåleren til Pi for x og y-aksen. Som vist i 2.3 trenger man avstanden til bunn for å måle rett posisjon. Derfor er det viktig at en tolker dataen fra sonaren rett.

Ifølge dokumentasjonen skal sonaren ignorere bevegelige objekter som dukker opp i bildet. Sonaren har +-30 graders sensorbredde til å bedømme kun én avstand(ikke flere objekter), som gir den godt grunnlag for å bedømme rett avstand til bunn. Likevel vil man legge inn en form for sikkerhet ved hjelp av dybdesensoren. Den tar utgangspunkt i at bunnforholdene vil ha en glidende overgang og ikke plutselige høydeendringer. Man bruker så endringen i dybdesensoren sammenlignet med eventuell endring i avstand til bunn for å bedømme om en skal godta den nye dataen eller forkaste den.

```
void Hent_SonarData(){
  ping.update();

float delta_avst_bunn = Gammel_Avst_bunn - ping.distance();
float delta_dybde = (Gammel_Dybde - SensorData.Dybde)*100;
if(delta_avst_bunn > 0.95*delta_dybde && delta_avst_bunn < 1.05*delta_dybde){
  Avst_bunn = ping.distance();
}
```

Figur 62: Funksjonen som bedømmer om man skal godta avstandsendring eller ikke

GPS

GPS er med som sensor fordi man ønsker å legge til rette for at dronen skal kunne gå opp til overflaten underveis i kjøringen for å rekalibrere sin egen posisjon. Sensoren har både I2C og Tx/Rx-mulighet, men siden det finnes biblioteker for omgjøring og tolkning av Tx/Rx velges det å bruke disse.

Figur 63: Bilde av GPS

Man må konvertere koordinater om til et referansepunkt i meter for å kunne brukes til å kalibrere posisjonen. Formelen som brukes er hentet fra nettet:

$$x = R * \cos(latitude) * \cos(longtitude) \quad (26)$$

$$y = R * \cos(latitude) * \sin(longtitude) \quad (27)$$

Hvor R er jordens omtrentlige radius fra ekvator på 6371 * 10^3 m. Hvis man setter et GPS-nullpunkt ved oppstart, kan man nå bruke GPS-data når dronen er nært overflaten til å rekalibrere posisjonen.

Utfordringen med dette er selvsagt unøyaktigheter i GPS'en sin nøyaktige posisjon samt at det kan ta tid før man får data fra GPS og at de første datasettene ofte kan være feil. Det er også en utfordring rent praktisk med å få testet en slik sammenkobling da dronen skal testes inne i et basseng, hvor signalet er dårlig.

Rent kodemessig tas det i bruk seriell-port 3. Når GPS får det såkalte «GPS-fix», som innebærer at den har fått triangulert posisjonen sin, vil den sende data til Arduino. Når det er data tilgjengelig omgjøres dataen til et referansesystem. Dette vil være i forhold til det satte GPS-nullpunktet, så dermed vil all data som kommer inn trekkes fra nullpunktet.

```
TinyGPS gps; // Til omforming av Gps-data
```

```
void HentGPSdata(){

while(Serial3.available()){ // check for gps data
  if(gps.encode(Serial3.read()))// encode gps data
  {

  gps.f_get_position(&lat,&lon); // get latitude and longitude
  // display position

  if(SensorData.Dybde < 0.5){
    float x = R*cos(lat*PI/180)*cos(lon*PI/180);
    float y = R*cos(lat*PI/180)*sin(lon*PI/180);

    float x_old = R*cos(GPS_pos[0]*PI/180)*cos(GPS_pos[1]*PI/180);
    float y_old = R*cos(GPS_pos[0]*PI/180)*sin(GPS_pos[1]*PI/180);

    pos[0] = pos[0] + (x - x_old);
    pos[1] = pos[1] + (y - y_old);
    GPS_pos[0] = lat;
    GPS_pos[1] = lon;

  }}
```

Figur 64: Utsnitt av koden for uthenting av GPS data og omgjøring til et referansesystem

Temperatur

Temperaturen finner man og på sensorbrikken som har pitch,roll og yaw. Denne har som sagt også en BMP280 som kan lese ut trykk og temperatur. Her finnes det et eget bibliotek som kan lese ut dataen, men man må huske at AD0 står til logisk høy som innebærer et bytte av I2C-adresse for å kunne hente ut temperaturen.

Temperaturen har ingen direkte betydning per nå for styring av dronen, men fungerer som en del av loggen. Den vil dermed måtte sendes fra Arduino til Pi, men utenom det vil det ikke være noen funksjoner som utfører handlinger skulle det bli varmt/kaldt.

Kodemessig brukes oppskriften fra en av eksempelfilene til biblioteket for uthenting av data. Selv om man henter ut både trykk og temperatur, brukes ikke BMP280 til dybdesensor da den er av mye dårligere kvalitet enn primærsensoren. Fortsatt henter man ut denne dataen som kan brukes om vår primære dybdesensor skulle streike, men det er ikke lagt inn noe kode for en eventuell redundans.

```
Adafruit_BMP280 bmp; //i2c
}
void HentData_TEMP() {
    // Trykk og temp BMP280//
    temp = bmp.readTemperature();
    pres = bmp.readPressure();
}
```

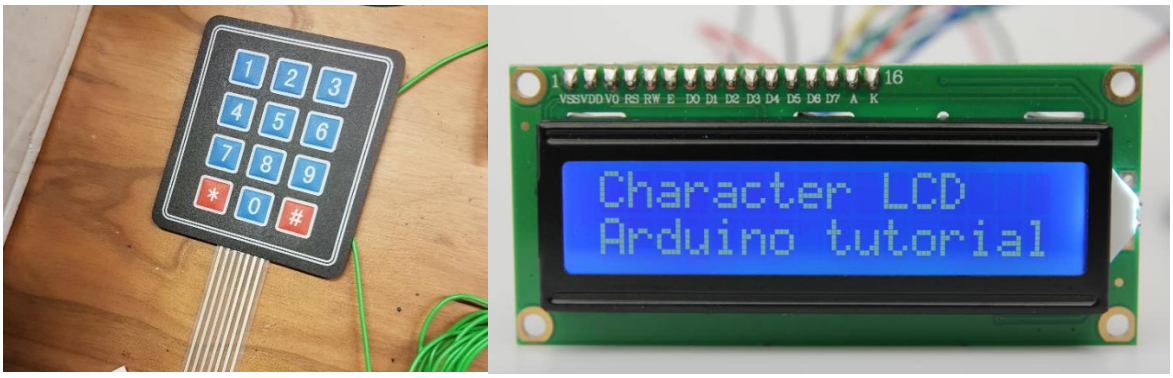
Figur 65: Utsnitt av koden for uthenting av data, all annen omgjøring gjøres i bakgrunnen av funksjonene

3.2.2 HMI

Siden man ikke har noen fysisk styring av dronen fra operatørsiden, må en ha mulighet til å sette noen pre-innstillinger til dronen før man lar den kjøre. Følgende muligheter er valgt:

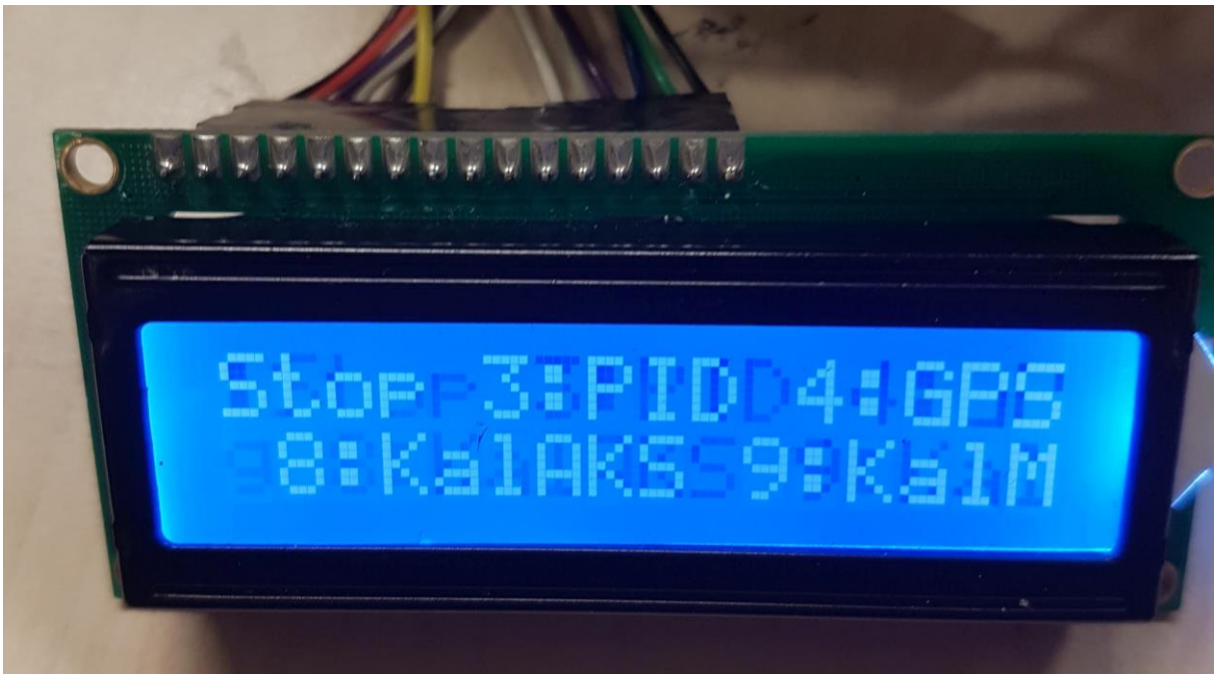
- Lage en ny kalibrering av de ulike sensorene
- Sette box-grensene(styring av drone)
- Sette nullheading
- Sette null-GPS
- Endre på konstantene til PID-regulatoren til servoene
- Kunne angi at man skal logge data
- Starte og Stoppe drone
- Vise hvilken pitch,roll og yaw dronen har for øyeblikket

HMI består av en enkel kompatibel lcd-skjerm med ferdige biblioteker som brukes til å sette innstillinger. Man bruker også et enkelt tastatur med 0-9 til å skrive inn kommandoer/innstillinger. Dette har også et ferdig bibliotek man kan bruke, slik at det å lage en HMI tar lite tid og koster nesten ingenting.



Figur 66: Tastatur og LCD-skjerm brukt som HMI. Kilde for bilde til høyre: https://www.makerguides.com/character-lcd-arduino-tutorial/

Oppsettet blir slik at man lager kommandoer fra 0-9 der «*» brukes til å aktivere kommandoene mens «#» brukes for å angre/slette det man har gjort. En har dermed mulighet til å ha 10 innstillinger/kommandoer. Måten man lager de ulike kommandoene er ved å kjøre en while-sløyfe som kun avsluttes av at bruker enten trykker stjerne eller firkant. Imens sløyfen kjøres vil man hente ut siste trykkede knapp og så gjøre ulike operasjoner. Inne i denne sløyfen må man også skrive informasjon til LCD-skjermen slik at bruker kan se hva som blir gjort.



Figur 67: Utsnitt av oppstartsmenyen til LCD-skjermen

Det er nødvendig å forskyve akselerometeret til et nullpunkt, samt å ha en god kalibrering på magnetometeret i det område den skal brukes. Videre ønskes separate knapper for start og stopp av dronen, for å hindre mulige feil. De andre trenger bare én knapp satt av slik at oppsettet blir som følger:

0: Box-grense

Er til bruk for styringssystemet. Hensikten er at man skal sette yttergrensene for kjøringen av dronen, som så sendes videre til Pi.

1: Start drone

Aktiverer en boolsk variabel kalt *oppstart_drone* som settes til sann. I hoved-loopen til Arduino er det to ruter som kan kjøres avhengig av verdien til denne variabelen. Samtidig må man med oppstart sende melding til Pi at man starter, samt nullstille eventuelle tidsvariabler som brukes.

Til slutt er det planlagt at HMI'en befinner seg inne i dronen, som betyr at man må legge på en oppstarts-forsinkelse slik at man får tid til å lukke igjen åpne deler i dronen. Denne settes standard til 100 sekunder hvor man har en nedtelling på LCD-skjermen.

2: Stopp drone

Setter variabelen *oppstart_drone* til usann. Man vil samtidig tømme eventuell buffer som finnes imellom seriellkoblingen slik at den ikke er der ved ny oppstart.

3: Sett PID-konstanter

Gir mulighet til å endre konstantene *Kp, Ki* og *Kd* som hører til PID-regulatorene for kontroll av pitch, roll og yaw.

4: Sett null-GPS

Gir to muligheter. Her kan man både se siste GPS-data om posisjonen, men også sette nullpunkt i referansesystemet. Den fungerer slik at LCD-skjermen viser siste GPS-data, og når man er fornøyd med den(at den er på rett plass) aktiveres den som nullpunkt.

5: Vis pitch, roll og yaw

LCD-skjermen viser dronens pitch, roll og yaw i sanntid. Er primært tiltenkt å bruke til å sette nullheading da man må vite hvilken heading man har akkurat nå.

6: Sett nullheading

Gir mulighet til å sette hva som er nullheading eller relativ 0.

7: Aktiver logging

Setter en variabel *Logmode* til sann eller usann, som igjen avgjør om man skal sende loggdata til Pi.

8: Kalibrering for Akselerometer

Kjører en kalibrering av akselerometeret med å ta snitt av 200 målinger og forskyve deretter.

9: Kalibrering for Magnetometer

I motsetning til for akselerometeret trenger denne hjelp fra bruker. Når man aktiverer denne kalibreringen må dronen beveges som om man maler innsiden av en ball, ref 3.2. Kalibreringen lagrer ytterverdiene og holder på så lenge man har en bevegelse. Med andre ord så lenge «ballen males» vil den fortsette, men når dronen står i ro, vil kalibreringen avsluttes og sette inn nye kalibreringsverdier.

3.2.3 Styring

Servostyring

Vi setter følgende navn på servoene:

- Yaw_OB: Oppe Bak
- Yaw_UB: Under Bak
- Yaw_OF: Oppe Fremme
- Yaw_UF: Under Fremme
- PR_SBB: StyrBord Bak
- PR_SBF: StyrBord Frem
- PR_BBB: BaBord Bak
- PR_BBF: BaBord Frem

Rent kodemessig bruker servoene et ferdig bibliotek. Bibliotekets funksjoner og servoene har et spenn på -90 til +90 grader, men skrives inn i koden i spennet 0 til 180.

For PID-styring må det velges *når* man ønsker at den skal agere. Sensordataen har en unøyaktighet på +- 0.5 grader, som man ikke ønsker at PID skal reagere på. Fordelen er at ønsket verdi for pitch og roll alltid skal være 0 grader. Man kan dermed sette inn en begrensning i PID-kontrolleren at den skal kun begynne å kalkulere nytt pådrag når dronen passerer en endring over unøyaktigheten.

Oppsett

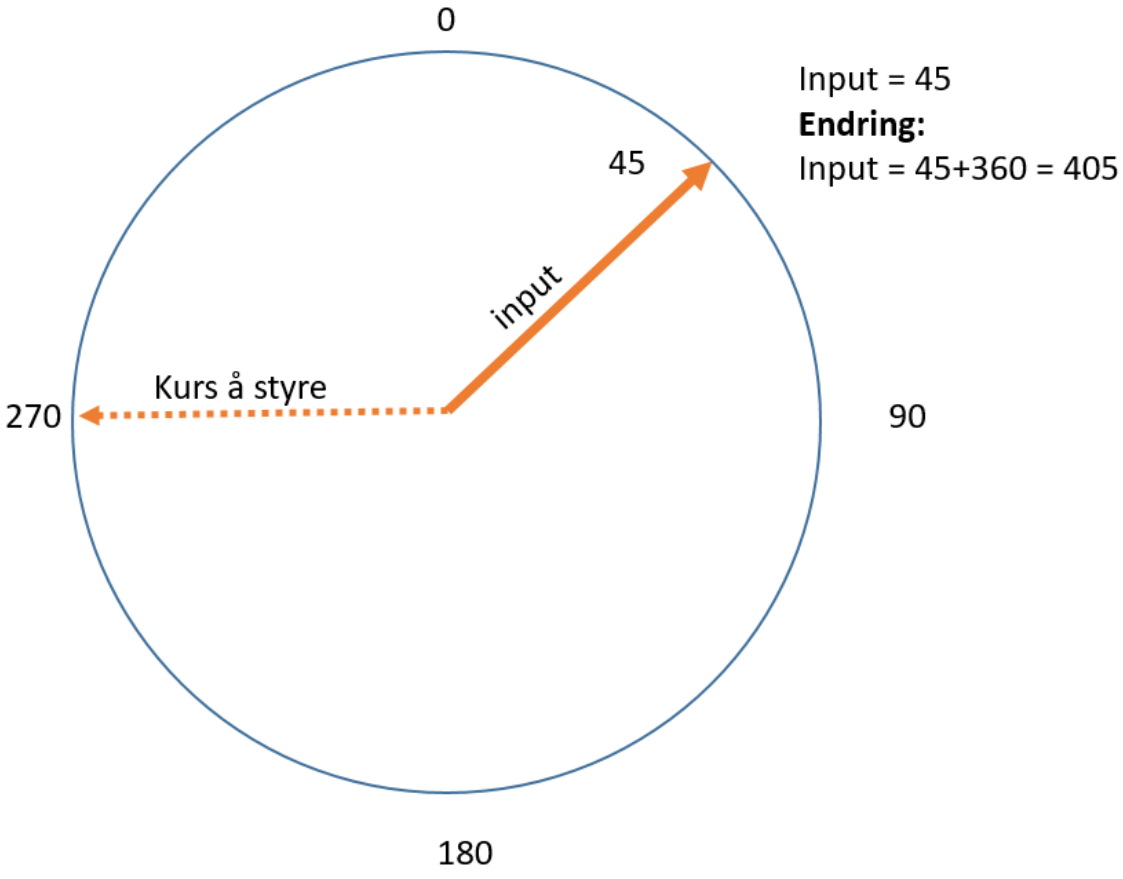
Servostyringen fungerer ulikt avhengig av hvilke moduser vi opererer i. For pitch og roll skal disse alltid være 0 og vil dermed fungere likt uansett, men man kan underveis ha en endring i ønsket dybde.

Styringen av yaw avhenger av om dronen har oppdaget et objekt eller ikke. Ved vanlig modus skal den fungere likt som for pitch og roll med et gitt settpunkt i henhold til kommandoen som er gitt. Hvis man har aktivert Rund eller Turn skal derimot yaw settes til maks utslag for å styre dronen styrbord for å unngå objektet. Enda viktigere for styring av yaw er at inputen vil ha et spenn på 0 til 359 grader som nevnt i 2.8. Løsningen er å konvertere input til PID avhengig av kursen dronen styrer. Inputen skal være sentrert rundt kursen i et spenn på [-180,180].

```
if (Kurs_kommando < 180){
  if (input_yaw > Kurs_kommando + 180){
    input_yaw -= (360);
  }
}
else if (Kurs_kommando > 180){
  if (input_yaw < Kurs_kommando-180){
    input_yaw += (360)  ;
  }
}
```

Figur 68: Omgjøring av input til et spenn på [-180,180] sentrert rundt kursen

Som en konsekvens vil input noen ganger gå over grensen for vanlig 360-graders system, men for PID vil dette ikke bety noe da den ikke er basert på et gradsystem. Her har man et eksempel:

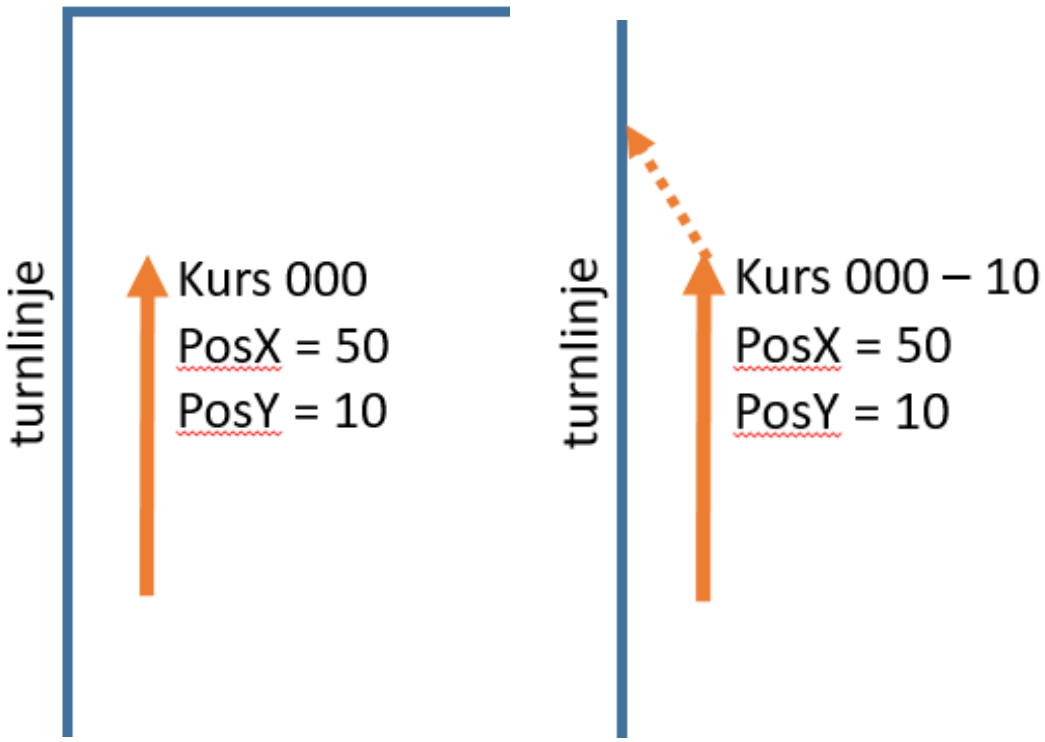


Figur 69: Tenkt situasjon der man ønsker å gå 270 men går 45 isteden. Med ny regning vil dronen gå babord og ikke styrbord, som er raskest.

Den hele linjen indikerer faktisk retning, mens den stiplede indikerer ønsket retning. Uten omregningen ville dronen ha beveget seg styrbord da input er lavere enn settpunkt. Med

omregning vil input til PID være 405 som er høyere enn 270. Resultatet er at dronen beveger seg babord for å komme på rett kurs istedenfor styrbord.

Videre for yaw var det to ulike måter den ville bli beregnet og styrt på. Vanlig mode er når man følger kursen fra Pi og ikke har noen objekter i veien. I tillegg til at settpunkt bruker kursen ifra kommando, vil den også legge til en korrigeringskurs. Korrigeringskursen er beregnet ut ifra posisjonsmålingen, slik at hvis man ikke er nøyaktig på turnlinjen, vil korrigeringskursen legge på ekstra pådrag på servoene for å styre riktig.



Figur 70: Eksempel på hvordan korrigeringskursen hjelper å få dronen inn på riktig linje

Slik eksempelet viser vil dronen tro at den kjører korrekt da den er på rett kurs, mens posisjonsmålingen avdekker at den ligger litt av og vil etterjustere kursen.

```
//VANLIG MODE YAW
if((RUND == false && TURN == false) || Vanlig_mode == true){
Korrigeringskurs();
Setpoint_yaw = (double)Kurs_kommando +(double)Korr_kurs;
//BEREGNING
YawPID.Compute();

Yaw_OB.write(Output_yaw+90);
Yaw_OF.write(Output_yaw+90);
Yaw_UB.write(90+Output_yaw);
Yaw_UF.write(90+Output_yaw);

}
```

Figur 71: Koden for normal kjøring med yaw, med beregning av korrigeringskurs

Den andre moden for yaw er hvis man skal unngå et objekt. Da skal dronen sette alle servoene til yaw i samme retning for å forskyve seg styrbord og vekk fra objektet. Man må derfor ta kontroll over hva som sendes til servoene. Dette gjøres ved å hindre at PID-kalkulatoren for yaw brukes når dronen er i en slik mode:

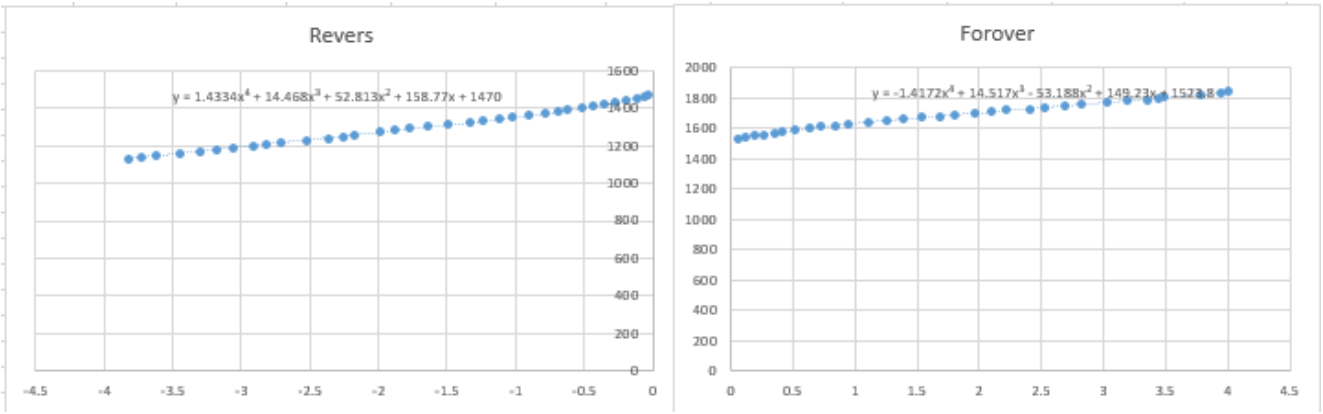
```
//Objektunngåing
else if(RUND == true || TURN == true){
  Rund_objekt();
  if (Vanlig_mode == false){
  Yaw_OB.write(Output_yaw+90);
  Yaw_OF.write(Output_yaw+90);
  Yaw_UB.write(90-Output_yaw);
  Yaw_UF.write(90-Output_yaw);
  }
}
}
```

Figur 72: Koden for kjøring med yaw hvis man skal unngå objekter

Motorstyring

Dronen har to motorer plassert på hver sin side ca midt på dronen. Disse styres i koden på lik linje med servoene ved bruk av det samme biblioteket. En utfordring ligger i at spennet på 0 til 180 er veldig lite for en motor som skal kunne gå frem og i revers. Det ble og oppdaget at nullstillingen på 90 grader ga litt fremdrift på motoren. Biblioteket tilbyr heldigvis også en annen funksjon som lar deg skrive i et spenn på 1000 til 2000, som gir muligheten til å sette en null-stilling som gjør at motorene står i ro. Videre måtte også motorene monteres motsatt vei av hverandre, noe som må beregnes med i koden. I følge kraftmålinger på motor fra leverandørens hjemmeside gir motortypen ulik kraft ved forover og revers. Det måtte derfor

gjøres noen beregninger basert på disse målingene for å fordele kraften jevnt utover slik at de produserer lik kraft.



Figur 73: Utregnet trendlinje for målte data på motorkraft, Fra vedlegg G

For enkelhetens skyld ønskes det å angi fremdrift i prosent fra -100% for full revers til +100% for full fremdrift. Det brukes en funksjon som konverterer mellom de to spennene, slik at man kan benytte oss av +-100% når det skal legges inn hastighet. Sammen med trendlinjene kan man implementere det i koden:

```
void til_motor(float prosent, Servo ESC){
  float til_ESC = 0;
  if(prosent > 0){
  prosent = (4.00522539/100)*prosent;
  til_ESC = -1.4172*pow(prosent,4) +14.517*pow(prosent,3) - 53.188*pow(prosent,2) + 149.23*prosent + 1523;
  }
  else if(prosent < 0){
    prosent = (3.82832416/100)*prosent;
    til_ESC = 1.4334*pow(prosent,4) + 14.468*pow(prosent,3) + 52.813*pow(prosent,2) + 158.77*prosent + 1470;
  }
  else{
    til_ESC = 0;
  }
  ESC.writeMicroseconds(int(til_ESC));

}
```

Figur 74: Konvertering fra prosent til reell motorkraft via funksjonen writeMicroseconds()

Siden dronen roteres ved bruk av servoer, vil man bruke en fastsatt hastighet for motorene. Men man kan også komme i situasjoner der strøm i vannet og andre ytre påvirkninger vil kunne føre til at man trenger ekstra motorkraft for å bevege dronen. Det gjør at den bør ha en form for etterjustering av motorkraften som er satt, men da trenger man posisjonsdata eller fart.

I 2.3 nevnes det at det å beregne posisjon og fart fra akselerometer ikke lar seg gjøre. Men man vil få posisjonsdata for x,y og z-aksen fra Pi via dens posisjonsmåler. Ved å lagre forrige posisjonsverdi samt tiden siden sist mottak, kan man beregne en tilnærmet fart eller i hvertfall en form for posisjonsendring. Negativ fart eller endring indikerer at man ikke har nok

kraft til å bevege dronen fremover, noe som er en enkel innstilling å legge ved i koden. Derimot er det vanskeligere hvordan man gjør hvis en ikke har nok kraft til å rotere om aksene.

Kommunikasjon

Arduino skal kommunisere med en Jetson Nano og en Raspberry Pi. Med Pi er de koblet sammen med en USB A til USB B, som gir oss tilgang til å bruke den serielle overgangen(Tx/Rx) dem imellom. Koblingen mellom Arduino og Jetson Nano skjer med ledninger imellom en av Tx/Rx-inngangene til Arduino og via en Tx/Rx-til-USB brikke som kobles i en av USB-inngangene til Jetson.

I denne seksjonen det snakkes om alt som gjøres fra Arduino sin side, imens det som skjer på motpartens vil skrives om deres delkapitler.

Mottak av data

Arduino skal motta data fra to ulike datamaskiner. Datapakkene som mottas fra Jetson vil alltid være i samme format, mens fra Pi vil man motta to ulike typer med kommandoer. Selv om Pi har posisjonssensoren vil den sendes via Jetson, slik at det kun er kommandoer som kommer fra Pi.

Man har en funksjon som baserer seg på protokollen som ble innført i 3.1. Siden det som sendes imellom Pi og Arduino er kommandoer, er det viktigere at man kontrollerer at dataen er korrekt i motsetning til sensordata fra Jetson. I det Arduino har mottatt en fullverdig melding, sender den meldingen i retur til Pi. Pi sjekker så at kommandoen er korrekt og sender en bekreftelsesmelding i retur. Når Arduino mottar bekreftelsesmeldingen fra Pi, vet den at dataen er korrekt og lagrer den som ny kommando. Kommandoen må kunne sendes flere ganger skulle den feile første gang, men det fikses fra Pi.

Koden er basert på en løsning publisert på Arduino’s åpne forum:

```
boolean newData = false;
while (Serial.available() > 0 && newData == false) {
    rc = Serial.read();

    if (recvInProgress == true) {
        if (rc != endMarker) {
            receivedChars[ndx] = rc;
            ndx++;
            if (ndx >= numChars) {
                ndx = numChars - 1;
            }
        }
        }
        else {

            receivedChars[ndx] = '\0'; // terminate the string
            recvInProgress = false;
            ndx = 0;
            newData = true;

        }
    }

    else if (rc == startMarker) {
        recvInProgress = true;
    }

}
if (newData == true ) {

    String Kommando_data = String(receivedChars);
    Serial.println("<" + Kommando_data + ">");  //Sender kommando i retur til Pihjerne
```

Figur 75: Mottaking av data fra Pi, legg merke til startmarker og endmarker

I korte trekk, så lenge man har data kjører funksjonen i en loop som leser én og én byte. Når den får «<» aktiveres *recvInProgress* som starter å lagre dataen i en tabell, helt til den mottar «>». Da vil funksjonen deaktivere *recvInProgress* og samle dataen. Nederst i bildet ser man at Arduino sender den mottatte dataen i retur for sjekk.

Med Jetson Nano opererer Arduino annerledes, selv om protokollen for mottak av data er lik. For det første var det utfordringer med å sende data fra Arduino til Jetson, noe som betydde at man ikke kunne verifisere meldingene på lik linje som mellom Arduino og Pi. For det andre er det ikke en meldingsgang som skjer sjeldent, men derimot en kontinuerlig sending av sensordata.

Fordelen med å sende hyppige meldinger til Arduino er at det ikke er så nøye at den ikke mottar korrekte meldinger. Med dette menes at Arduino selvsagt ikke skal tolke feilmeldingene som rett data, men at ved inspeksjon skal kunne forkaste dem og heller vente på ny data. Samtidig må man ha kontroll på trafikken slik at en ikke overbelaster Arduino med data, men heller får en kjapp flyt. Her blir utfordringen med å sende data fra Arduino til Jetson fort et problem, da man ikke kan bruke den til å kvittere på at den har mottatt en melding.

Løsningen ble å bruke General-Purpose-Input-Output(GPIO)-brettet til Jetson, der man kobler en fysisk ledning imellom en av de digitale inngangene på Jetson og Arduino. Løsningen baserer seg på at det er Arduino som sender logisk høy (digital verdi 1) til Jetson når den:

- Nettopp har lest ferdig en melding og godkjent den som korrekt data
- Anser datamengden som ukorrekt eller for lang og tømmer buffer

Utgangen til Arduino skal være logisk høy så lenge den ikke har data i bufferen. Når den mottar et sett med data fra Jetson vil Arduino sette logisk lav(digital verdi 0) på utgangen, som gjør at Jetson avventer med å sende ny data. Så har man de to måtene å få logisk høy igjen. Den første måten er ganske enkel, ved at man etter å ha godkjent dataen som korrekt setter logisk høy igjen. For den andre metoden innebærer det at Arduino skal vurdere meldingsgangen som kommer som ukorrekt. I 3.1 nevnes det at protokollen må ha en maksbegrensning på meldingen som sendes som begge parter er klar over. Det er sagt at Jetson skal sende tre ting: avstand til objekt, posisjon i x-retning og posisjon i y-retning. Hos Jetson Nano er det satt en maks avstand til objekter på 360cm, altså at alt bak dette behandles som usant. Denne delen av meldingen vil dermed ta opp 3 plasser. For posisjon i x- og y-retning vil det avhenge av hva man setter dronen til. Ref 3.1 er en av problemstillingene med å sette maksbegrensning på meldingen at man også begrenser distansen dronen kan dra. Det settes en begrensning i posisjon på 7 siffer, med andre ord en rekkeviddebegrensning på rett under 100 kilometer.

Det er også nødvendig med et tilleggskriteria basert på muligheten for overlappende meldinger. Skulle det komme en ny «<» etter at man allerede har mottatt én, vil programmet slette buffer og be om ny melding. Slik kan man hindre at meldinger overlapper hverandre og gir ukorrekt data.

```
else if(rc == startMarker || ndx > 20){
  while(Serial2.available()){
    char t = Serial2.read();
    Print_LCD("Y",1,1);
  }
  digitalWrite(12,HIGH);  //Send ny data
  digitalWrite(13,HIGH);
  return;
}
```

Figur 76: Kriteriene som avbryter mottakelse av data, rydder opp buffer og ber om å få ny melding, de gule delene er ikke viktige her

På bildet man at enten en overskridelse av meldingsstørrelse på 20 bytes eller overlappende meldinger vil tømme buffer og be om ny data. Linjene markert med gult er for feilsøking og kan ignoreres. Utenom denne linjen er mottakelsen av melding helt lik som mellom Arduino og Pi.

Sending av data

Det var utfordringer med å få til å sende data fra Arduino til Jetson, derfor er det kun til Pi Arduino sender data. Arduino skal sende mottatt kommando i retur for bekreftelse av at den er korrekt. Når meldingen er verifisert vil den lagres som en ny kommando. Ref 3.1 skal Arduino innkapsle meldingen i «<>».

Arduino skal videreformidle oppstart og stopp til Pi samt å sende oppdatert sensordata på avstand bunn og dybde. HMI med LCD-skjerm og tastatur er koblet til Arduino, noe som betyr at når man trykker oppstart skal den varsle Pi om dette, hvor det blir Pi sin jobb å varsle videre til Jetson. Det må etableres et køsystem hos Arduino for sending av både sensordata og kommando. Dette løses med at når Arduino har mottatt en ny kommando fra Pi, så blokkeres sendingen av sensordata helt til man får bekreftet kommandoen av Pi. Dette kunne skapt en flaskehals i koden hvis man ikke hadde fått bekreftelsen, altså at kommandoen var feil. Men dette løses med at Pi fortsetter å sende kommando helt til den får bekreftet at den er mottatt korrekt.

Med tanke på at man har funnet en god løsning med enkel informasjonsutveksling ved bruk av GPIO, kunne det tenkes at en slik løsning kunne blitt brukt for å markere oppstart og stopp av dronen. Men samtidig vil dette forekomme som regel ganske tidlig etter at man har startet opp de tre maskinene, noe som innebærer at det vurderes som nyttig fra vår side å få testet at seriellkoblingen fungerer mellom alle tre. Oppstart-seansen benyttes derfor til å verifisere at kommunikasjon mellom Arduino/Pi og Pi/Jetson fungerer.

```
if (newData == true ) {
    String Kommando_data = String(receivedChars);
    Serial.println("<" + Kommando_data + ">");  //Sender kommando i retur til Pihjerne
    if (Kommando_data == "999"){
      rett_kommando_bekreftelse = true;
      avvent_sending = false;
    }
```

Figur 77: Vi sender en bekreftelsesmelding til Arduino og blokkerer annen transmisjon ved Avvent_sending, når den så får verifikasjon at den er rett (her "999") lagrer Arduino kommandoen

Hvis Arduino mottar «999» etter å ha sendt ny kommando i retur, vet den at kommandoen er rett og kan lagre den. Da sendes også «999» i retur slik at Pi vet at ting er bekreftet. Her benyttes variabelen *avvent_sending* som er den som stopper all sending av sensordata til Pi

så lenge den står til sann. Når man mottar «999» kan denne settes til usann og sensordata kan sendes igjen.

3.2.4 Objektunngåelse

Det er Pi som hjerne for hele systemet som avgjør hvordan en skal reagere når det oppdages nye objekter. Men selve styringen av motorer og rotasjon på dronen er det Arduino som gjør. Styringssystemet har de tre metodene:

- Overse
- Turn
- Rund

Ved overse vil Arduino aldri motta noen kommando fra Pi, og dermed heller ikke gjøre noen tiltak. For Turn og Rund er de like helt til det som skjer når dronen er på linje med objektet. Beskrivelsen tar utgangspunkt i 2.8, der fellestrekkene gjennomgås først:

- Oppdagelse
- Unnvikelse
- Kjøring til man er på linje med objektet

For hvert steg har Arduino en egen boolsk variabel som blir satt til sann når steget er ferdig slik at programmet beveger seg videre til neste steg. Steg 1 lagrer avstanden til objektet i variabelen *Avstand_oppdagelse_objekt* for bruk senere, samtidig som man setter målpunktet i y-akse for senere steg.

```
if(Rund0 == false && Rund1 == false && Rund2 == false && Rund3 == false){
    Rund0 = true;
    PosY_maal = 0;
    Avstand_oppdagelse_objekt = Avst_obj;
}
```

Figur 78: Steg 1 som lagrer avstand til objekt og setter målpunkt for y-aksen

Steg 2 starter å forskyve styrbord ved at alle fire servoer har retning samme vei. Den forskyves med samme teknikk som ved dykking. Dette krever at man ignorerer PID-kontrolleren og setter output til servoen på egen hånd. Det benyttes en variabel kalt *Vanlig_mode* som er styrende for hvilken mode som skal nyttes i Servostyringen. Usann verdi tilsvarer ingen PID. Kriteriet for at steget er ferdig er når man ikke lenger ser objektet. Da vil Arduino benytte *Avstand_oppdagelse_objekt* til å beregne målpunkt i x-retning. Dronen vil ha beveget seg litt i x-retning under forskyvningen som gjør at målpunktet vil være etter man er på linje med objektet. Dette er et bra slingringsmonn skulle objektet ha en tykkelse eller at avstandsmåleren feilberegner litt, slik at man ikke kolliderer.

```
//Turn SB til du ikke ser objektet lenger
else if(Rund0 == true && Rund1 == false && Rund2 == false && Rund3 == false){

  Vanlig_mode = false;
  Output_yaw = 60;  //Ror i bordet, turner bortover

  if(naert_objekt == false){
    Rund1 = true;
    PosX_maal = PosX + abs(Avstand_oppdagelse_objekt);

    Vanlig_mode = true;

  }
}
```

Figur 79: Steg 2 med forskyvning styrbord helt til man unngår objektet. Merk en allerede her setter Vanlig_mode til usann

Steg 3 skjer når dronen er forbi objektet slik at den styrer vanlig kurs igjen, ved å sette *Vanlig_mode* til usann(dette gjøres i det øyeblikket steg 2 er avsluttet). Dermed tar PID-kontrolleren igjen over og holder dronen på kurs, helt til steg 3 er ferdig. Kriteriet for at steg 3 avsluttes er når man har passert målpunktet i x-retning satt tidligere.

```
//Kjør til du er på linje med objekt
else if(Rund0 == true && Rund1 == true && Rund2 == false && Rund3 == false){

  if(PosX > PosX_maal){  //Avventer til vi har passert målpunktet
    Rund2 = true;
  }

}
```

Figur 80: Steg 3 som ikke gjør noe annet enn å avvente sluttkriteriet

Når man er forbi/på linje med objektet vil Turn og Rund være forskjellige.

Turn

Måten Turn fungerer på er beskrevet tidligere, men det som funksjonen skal gjøre er å fortsette videre på samme kurs helt til den når ny turnlinje. Avhengig av hvilken turnlinje man befinner oss på, så vil turnpunktet variere. Arduino har lagret de fire turnpunktene i en tabell, slik at man har tilgang til dem alle. Tabellen heter *BOX_X* der indeksen endrer seg fra 0 til 3 avhengig av hvilken turnlinje man er på. Dette skaper et generelt uttrykk en kan bruke istedenfor å måtte ha fire if-setninger hver gang man er nødt til å ta i bruk disse punktene.

```
}
//for TURN: fortsett i samme retning til vi treffer ny linje
else if(Rund0 == true && Rund1 == true && Rund2 == true && Rund3 == false && TURN == true){
   float deltax = BOX_X[Box_nummer] - PosX;

   if(abs(deltax) < 10 || PosX > BOX_X[Box_nummer]){
    Rund3 = true;
   }
}
```

Figur 81: Steg 4 for Turn. Man fortsetter på samme kurs helt tileni når ny turnlinje

Rund

Rund er også beskrevet tidligere, men skal returnere til original turnlinje i det dronen har passert objektet. *Vanlig_mode* settes til sann igjen slik at servoene kan roteres likt, bare i motsatt retning fra tidligere. Avslutningskriteriet for steg 4 er når dronen har kommet seg tilbake til turnlinjen, som er definert som 0 langs y-aksen.

```
//for RUND:Vend tilbake til turnlinje
else if(Rund0 == true && Rund1 == true && Rund2 == true && Rund3 == false && RUND == true){
  Vanlig_mode = false;
  Output_yaw = -60;
  float deltay = PosY_maal - PosY;

   if((abs(deltay) < 10 || PosY < PosY_maal)){
    Rund3 = true;
    Vanlig_mode = true;
   }


}
```

Figur 82: Steg 4 for Rund. Dronen forskyver seg babord tilbake til original turnlinje

Avslutte objektunngåelse

Arduino skal fortelle Pi at den er ferdig med objektunngåelse og avventer å resette steg-variablene til den vet at Pi har mottatt. Steg 4 er likt for begge funksjonene og sender «DONE» til Pi helt til den får samme melding i retur. Arduino stopper også sending av sensordata i denne perioden for å sikre seg at meldingene kommer frem.

```
//Vi er ferdige og sender det inn
else if(Rund0 == true && Rund1 == true && Rund2 == true && Rund3 == true){
    Serial.println("<DONE>");

    avvent_sending = true;

}
```

Figur 83: Siste steg som sender bekreftelsesmelding om at vi er ferdig til Pi

Sikkerhetssjekk

Da stegene blir gjennomført ved boolske variabler kan man komme i situasjoner der noen av disse er aktivert og dronen er på vei ut av grensene man har definert i 2.8. Man må ha en

form for «failsafe». Det legges inn et kriteria på at funksjonene avbrytes hvis man går for langt utenfor grensene. Når kriteriet er sant settes alle steg-variablene til usann, slik at man avbryter unngåelsen:

```
if(PosX > BOX_X[Box_nummer] + 10){
  Serial.println("<DONE>");

  avvent_sending = true;
  RUND = false;
  TURN = false;
  Rund0 = false;
  Rund1 = false;
  Rund2 = false;
  Rund3 = false;
```

Figur 84: Avbrytning av objektunngåelsen hvis dronen passerer de satte grensene for BOX

Videre må man ha en løsning skulle dronen ikke klare å forskyve seg unna objektet. Når dronen er 30cm unna objektet vil man ikke lenger klare å manøvrere styrbord uten å kollidere, og setter da inn at dronen skal gå i full revers. Da man har prøvd å forskyve styrbord må man resette finnen når en bakker bakover, slik at man beholder dronen langs turnlinjen. Dronen skal rygge bakover til den er minst 200cm unna objektet, før man prøver igjen med Rund/Turn.

```
else if(Avst_obj < 30 || For_naert_objekt == true){
  Rund0 = false;
  Rund1 = false;
  Rund2 = false;
  Rund3 = false;
  For_naert_objekt = true;
  til_motor(-10,ESC_BB); //Reverserer motor
  til_motor(+10,ESC_SB);
  Vanlig_mode = true;
  Output_yaw = 0;
  if (Avst_obj >= 200){
    For_naert_objekt = false;
    til_motor(Fart_fra_Bruker,ESC_BB);
    til_motor(-Fart_fra_Bruker,ESC_SB);
  }
}
```

Figur 85: Failsafe for å unngå kollisjon hvis dronen ikke klarer å forflytte seg unna objektet

3.3 Software Jetson

Jetson Nano skal ta på seg følgende oppgaver:

- Sensordata:
    - Avstand til objekt

    - Vinkel til objekt
- Logging
    - Bildetaging av objekt
- Kommunikasjon
    - Mottaking av kommando
    - Overføring av sensordata
    - Videresending av sensordata

I motsetning til beskrivelsen av Arduino og Raspberry Pi, vil det bli diskutert deler som ellers ville hørt hjemme under kapitlet om drøfting. Vurderingen som ligger bak hvorfor man tar det her er at mye av diskusjonen vil referere til detaljer i så stor grad at det passer bedre å få det med i beskrivelsen av dette systemet. Diskusjonen vil ta for seg beslutninger man foretok seg rundt å gå til innkjøp av Jetson for å erstatte en Pi som originalt ble brukt til avstandsmåling.

Grunnet byttet av maskin vil forklaringen ta for seg oppsettet ut ifra hvordan man originalt gjorde det på Pi. Til slutt vil en forklare hvilke endringer man gjorde i forbindelse med overgangen til Jetson.

3.3.1 Sensordata

Basert på oppsettet beskrevet tidligere i utviklingsdelen må programmet hente ut bildet fra kamera, oppdage laserpunktene, bedømme avstanden mellom dem og sette avstanden inn i algoritmen som beregner faktisk avstand. Strukturen på forklaringen er delt opp i henhold til den rekkefølgen.

Få tak i kamera-feed

Til å begynne med må man altså få tak i kamera-feeden. Måten det gjøres dette på er at man benytter seg av en OpenCV-funksjon kalt VideoCapture, som «snapper» et bilde av feeden og lagrer den i en variabel.

```
cap = cv2.VideoCapture(0)

while True:


    frame = cap.read()
```

Figur 86: "Snapping" av kamera-feeden

Man «snapper» bildet inne i en while-loop, som betyr at man kontinuerlig henter ut bilder fra kamera. Siden programmeringshastigheten er høy vil man hente bilder såpass raskt at det blir akkurat som man har kamera-feeden.

Pi vs Jetson Nano på bildeoppdateringsrate

Desto raskere man klarer å gjennomføre while-loopen jo bedre oppdateringsrate får man på kamera-feeden. Når en startet med utviklingen benyttet man en Raspberry Pi. Utfordringen er her at OpenCV sine funksjoner bruker enormt mye prosessorkraft som igjen resulterer i at de bruker enormt mye tid(relativt sett fra et dataprograms perspektiv) på å kjøre gjennom loopen. Som en konsekvens vil bildeoppdateringsraten være ekstremt dårlig hvis man ikke har en god nok grafisk prosessor tilgjengelig. For at dronen skal kunne være effektiv nok må den kunne ha en rask nok oppdatering på kameraet foran slik at en alltid har god kontroll på hva som er foran oss. Hastigheten man fikk ut av Pi ble ansett som ikke god nok til dette, da en ofte kunne holde hånden foran kamera og måtte vente flere sekunder før man så den på skjermen vår!

Detektere laserprikkene

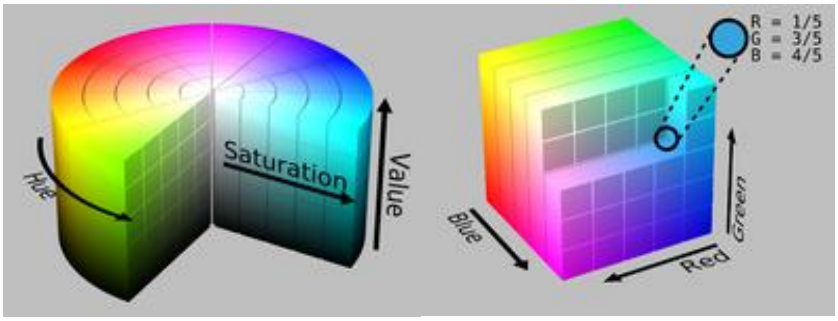
Man har lagret kamerabildet i en variabel kalt *frame*. Det er nå denne variabelen man skal bruke til å hente laserpunktene, som må skilles ut fra resten av bildet. Variabelen er en tre-dimensjonal tabell siden en driver med farger. Hver pixel vil være koblet til tre 8-bits verdier mellom 0 og 255 som indikerer intensiteten til pixelen i rød, gul og blå-farge. Denne kombinasjonen for å lage bilder i variabler kalles RGB.

Fordelen er at laserpunktene både har en sterk og kraftig rødfarge samt en kraftig lysstyrke. Å bruke RGB for å detektere denne kombinasjonen er derimot vanskelig. Årsaken er at det krever at vi finner nøyaktig RGB-kombinasjon til laserpunktene i bildet og så gjenskaper denne i filtreringen. Men fargen i bildet og lysstyrken vil kunne endre seg avhengig av laserstyrken, bildeforhold og bakgrunnen til objektet som også vil endre RGB-kombinasjonen til laserpunktene. Det er dermed enklere å konvertere over til et HSV-bilde, som står for Hue(farge), Saturation(mengde) og Value(lysstyrke).

OpenCV har funksjonen *cvtColor* som kan konvertere bilder til andre typer. Det er viktig å merke at OpenCV henter ut bilder som BGR kontra RGB, men dette utgjør ingen forskjell utenom hvilket format man konverterer fra.

```
hsv = cv2.cvtColor(FRAME, cv2.COLOR_BGR2HSV)
```

HSV har fargespekteret fordelt kun på Hue, og så lysstyrken i fargen fordelt på Value. Ved å konvertere bildet over til HSV kan man enklere skille ut laserpunktene da de vil ha rød farge og ekstremt kraftig lysstyrke.



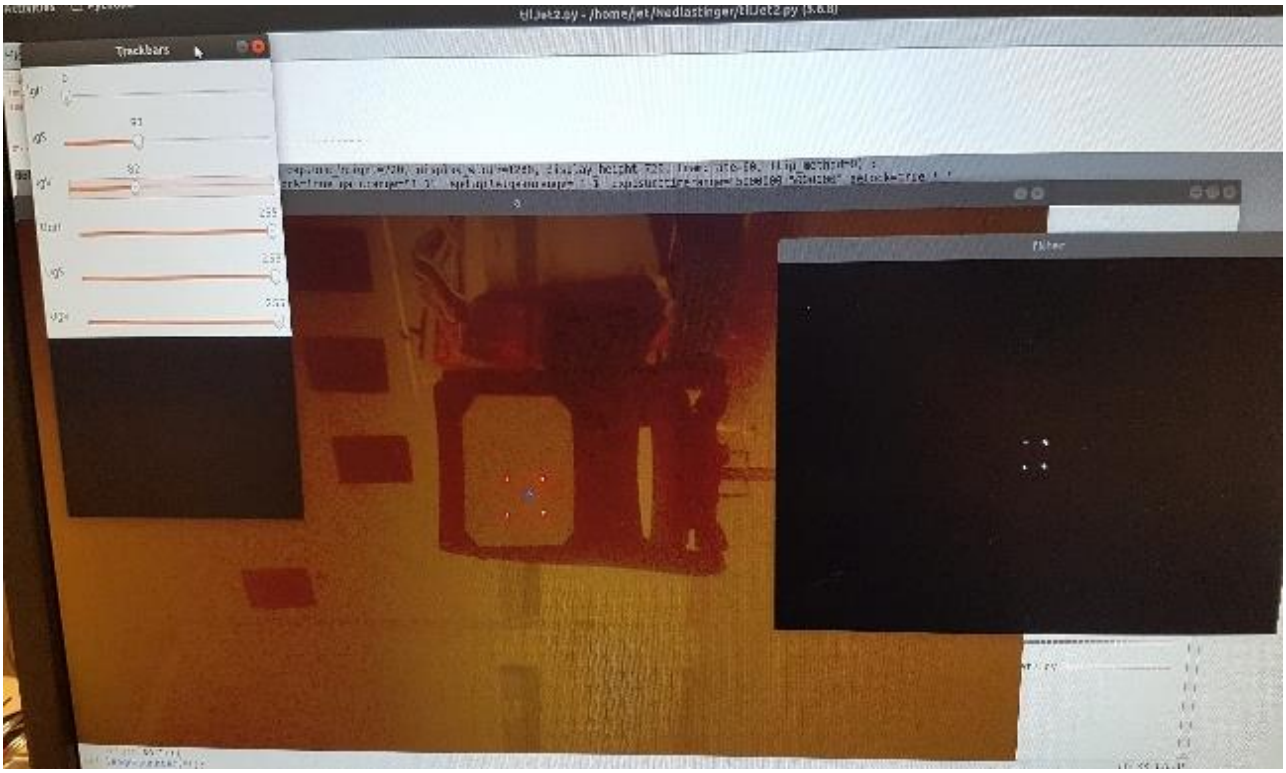
Figur 87: HSV-fordeling til venstre og en RGB-fordeling til høyre. Kilde: https://en.wikipedia.org/wiki/HSL_and_HSV

Måten man så skal skille ut laserpunktene fra resten er å ta i bruk en funksjon kalt *inRange*. Den tar inn minimum og maksbegrensning i verdier for Hue, Saturation og Value og kjører så igjennom alle pixlene i bildet. Den returnerer følgende verdi for hver pixel:

$$f(p) = \begin{cases} 255 \; hvis \; hsv_{min} \leq p_{hsv} \leq hsv_{max} \\ 0 \; ellers \end{cases}$$

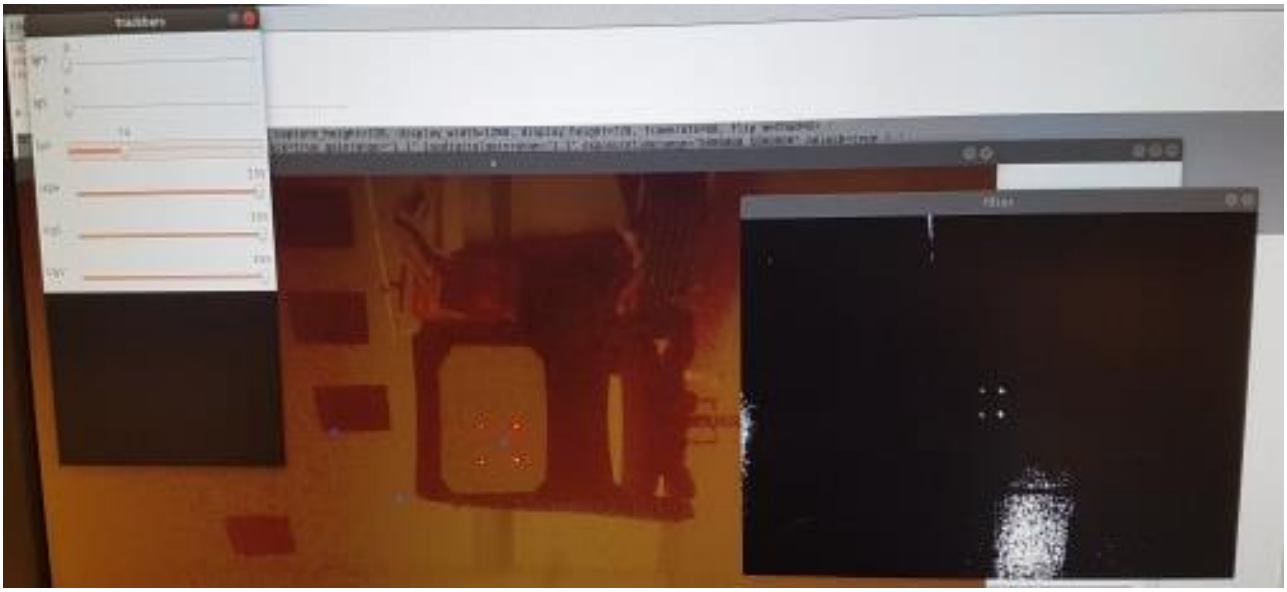
Man må spesifisere hvilke grenseverdier man ønsker for bildet. Vi har en anelse om hvor rødfargen vil ligge i henhold til HSV-spekteret, men den beste metoden er å prøve seg frem. Her vil også lysforholdene være avgjørende for hvor godt kameraet oppfatter laserpunktene. Hvis man har laserne i et lyst område vil kameraet ikke oppfatte laserpunktene som like høy intensitet i value enn hvis man er i et mørkt område. Det må testes både i lyse og mørke forhold slik at en har to ulike rekkevidder avhengig av lysforholdene. Det trengs derfor en gjenkjenner av lysforhold i bildet for å kunne nytte rett filter.

inRange returnerer noe som kalles for en maske. En maske har i motsetning til et vanlig kamerabilde kun lysintensitet som farge der verdien enten er helt lys(hvit, 255) eller helt mørk(svart, 0). Funksjonen skal returnere en maske som har hvite pixler der laserpunktene er, og mørkt ellers. Hvis man har et bra filter skal funksjonen returnere en maske som ser slik ut:



Figur 88: Kamera-feed av vanlig bilde til venstre og masken vår til høyre. Vi ser at det kun er laserpunktene som er hvite(verdi 255) i masken

Hvis man har et filter med større min/maks-begrensninger vil man kunne få med støy som i bildet under:



Figur 89: Kamera-feed når vi har et filter som ikke klarer å skille ut alt uønsket

Selv om man kan legge inn funksjoner som gjør at filteret i det andre bildet også kan brukes, så vil flere høye verdier i masken føre til at programmet går tregere, som igjen gir dårligere oppløsning.

Fordelen nå er at masken, eller variabelen *mask* har verdier 255 i de områdene man har laserpunkter og 0 ellers. Dette gjør at man vet pixelkoordinatene til laserpunktene. For å hente ut punktene brukes en funksjon som heter *findContours*. Som navnet tilsier henter den ut konturene i et bilde. Å hente ut alt vil kreve ekstra datakraft, så en kan nøye seg med et par avgrensninger. Ved å gi argumentet *cv2.CHAIN_APPROX_SIMPLE* bes den kun om å lage ytterpunktene til hver kontur istedenfor å lagre hele konturen. Det vil kreve mindre plass og ta kortere tid.

```
contours,_ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

Figur 90: funksjonen findContours

Funksjonen returnerer strengt tatt to ting, men utenom konturene returnerer den et hierarki som inneholder topologien til bildet. Dette er ikke nødvendig og markeres ut med en «_» i koden. I variabelen *contours* har man nå en god del punkter til hvert sitt laserpunkt. Ut av disse må en finne senter som blir regnet som pixelkoordinater. Man tar i bruk noe som kalles moments for å finne senter hentet fra nettsiden til Satya Mallick[6]:

The centroid is given by the formula:-

$$C_x = \frac{M_{10}}{M_{00}}$$

$$C_y = \frac{M_{01}}{M_{00}}$$

C_x is the x coordinate and C_y is the y coordinate of the centroid and M denotes the Moment.

Figur 91: Formel for å finne senterpunktene. Kilde: https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/

```python
# calculate moments of binary image
M = cv2.moments(thresh)

# calculate x,y coordinate of center
cX = int(M["m10"] / M["m00"])
cY = int(M["m01"] / M["m00"])
```

Figur 92: Kode for å hente ut senterpunktene. Kilde:https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/

Det kan være tilfeller der man får flere punkter utenom de fire laserpunktene som gjør det trengs en form for filtrering. Hvis man deler bildet i fire kvadranter rundt senter, skal det ideelt sett være ett punkt per kvadrant. Filteret designes til å kun beholde det punktet som er nærmest senter for hver kvadrant. Ulempen er selvsagt hvis det finnes støy som gir ut punkter nærmere senter enn laserpunktet, så vil det gi feil i beregning av avstand. Videre vil også en forskyvning i kameraets senter, altså at kamera endrer posisjon uten at laserne gjør det, også kunne gjøre at filteret fjerner faktiske laserpunkter. Dette var en utfordring da Pi-kamera viste seg å ikke alltid være sentrert.

```python
for i in range(len(punkter)):
    x = punkter[i][0]
    y = punkter[i][1]
    avstand = math.sqrt((x-centerX)*(x-centerX) + (y-centerY)*(y-centerY))
    avs_OV  = math.sqrt((OVx-centerX)*(OVx-centerX) + (OVy-centerY)*(OVy-centerY))
    avs_OH  = math.sqrt((OHx-centerX)*(OHx-centerX) + (OHy-centerY)*(OHy-centerY))
    avs_NV  = math.sqrt((NVx-centerX)*(NVx-centerX) + (NVy-centerY)*(NVy-centerY))
    avs_NH  = math.sqrt((NHx-centerX)*(NHx-centerX) + (NHy-centerY)*(NHy-centerY))

    if avstand < avs_OV and x <= centerX and y <= centerY: #OV
        OVx,OVy = x,y
    if avstand < avs_NV and x <= centerX and y >= centerY: #NV
        NVx,NVy = x,y
    if avstand < avs_OH and x >= centerX and y <= centerY: #OH
        OHx,OHy = x,y
    if avstand < avs_NH and x >= centerX and y >= centerY: #NH
        NHx,NHy = x,y
```

Figur 93: Filter som kun tar med de punktene som er nærmest sentrum av bildet.

Hvor *centerX* og *centerY* er pixelkoordinater til bildets sentrum. I programmet er det andre filtre som skal hjelpe på, skulle for eksempel tilfellet skje at det kommer støy som gir ut et punkt nærmere sentrum enn laserpunktet. Disse filtrene vil ikke bli gjennomgått nå men er dokumentert i koden som blir lagt ved oppgaven.

Pi vs Jetson Nano på detektering av laserpunkter

Styringssystemet trenger tid til å agere på eventuelle objekter som kommer i veien for dronen. Dette krever at man kan oppdage dem på god nok avstand, der over 3 meter bør være minimum. Begrensningene man har nevnt tidligere for Pi gjør at en også må begrense størrelsen på bildet til 640x320. Resultatet er at det blir vanskeligere å beregne pixelavstanden når man er lengre unna da de på kamera vil bli veldig nært. Jo mindre pixler man har i bildet jo mindre sensitiv blir den på endringer når laserpunktene er langt unna. Grafikkortet til Jetson er mye bedre enn Pi, slik at programmet har høyere oppløsning i bildene. Det gir den muligheten til å måle enda mer nøyaktig på lengre avstander. Grunnet høyere hastighet kan man også benytte 1280x720 som for Jetson gir deteksjon på større avstander. Under er målinger på gjennomsnittlig pixelavstand for Jetson og Pi på hhv 160cm og 170cm avstand til objektet:

Jetson	Pi	avstand(cm)
49.6	21.5	160
45.75	20.5	170

Figur 94: Snitt på pixelavstand for Jetson og Pi

Ikke bare har man større avstand hos Jetson som følge av bedre oppløsning, men 170cm er også den største avstanden programmet klarte å detektere laserpunktene stabilt over tid hos Pi. For Jetson derimot fikk man gode målinger opp til 360cm som er nesten to meter lenger!

Beregne pixelavstand

Hvis filtreringen har vært suksessfull står man igjen med fire pixelkoordinater som tilsvarer koordinatene til laserpunktene. OpenCV definerer nullpunktet (0,0) oppe i venstre hjørne av bildet. Funksjonen *findContours* velger punkter ved å starte i (0,0) og gå mot høyre og så ned. Dette avgjør rekkefølgen hvert punkt hentes ut i, noe som kan endre seg skulle man ha unøyaktigheter i laserpunktene. I filteret er de fire punktene lagret i en spesifikk rekkefølge ut ifra hvilken kvadrant de er i, slik at man hindrer at det nevnte problemet oppstår.

```
OV_OHx = abs(LP[0][0] - LP[1][0]) #Oppe,venstre til Oppe,hoyre x-verdi
OV_OHy = abs(L[P0][1] - LP[1][1]) #Oppe,venstre til Oppe,hoyre y-verdi

OV_NVx = abs(LP[0][0] - LP[3][0]) #Oppe,venstre til Nedre,venstre x-verdi
OV_NVy = abs(LP[0][1] - LP[3][1]) #Oppe,venstre til Nedre,venstre y-verdi

OH_NHx = abs(LP[1][0] - LP[2][0]) #Oppe,hoyre til Nedre,hoyre x-verdi
OH_NHy = abs(LP[1][1] - LP[2][1]) #Oppe,hoyre til Nedre,hoyre y-verdi

NV_NHx = abs(LP[3][0] - LP[2][0]) #Nedre,venstre til Nedre,hoyre x-verdi
NV_NHy = abs(LP[3][1] - LP[2][1]) #Nedre,venstre til Nedre,hoyre y-verdi
```

Figur 95: Beregning av pixelavstand mellom punktene

Programmet har nå fire avstander som det kan bruke i algoritmen fra 2.2 for å hente ut faktisk avstand.

$$f(\Delta x) = 6458.4 * x^{-0.955} \; og \; f(\Delta y) = 7326.1 * x^{-0.977}$$

Programmet må holde styr på om man benytter en avstandsendring i x eller y, og dette gjøres ved å benytte rekkefølgen en har spesifisert tidligere:

```
def pixel_til_avstand(pixelavstand, index):
    if index in [0,3]:
        avstand = 6548.4 * pixelavstand**(-0.955)
    elif index in [1,2]:
        avstand = 7326.1 * pixelavstand**(-0.977)
return avstand
```

Figur 96: Beregning av faktisk avstand i forhold til pixelavstand

3.3.2 Overgang fra Pi til Jetson

Med bakgrunn i resultatene for Pi og Jetson var det naturlig å bytte. Det var derimot ikke slik at det ikke innebar noen andre utfordringer. Den første var hensyn til strømforsyning. Raspberry Pi 3B+ bruker alt fra 350mA(idle) til 950mA(400% CPU-kraft). I motsetning til Jetson som i normal bruk trenger 2A, som er en betydelig økning som må tas i betraktning. Den andre og desidert største utfordringen var at programmet for Pi var så og si ferdig i det man valgte å bytte over til Jetson. Selv om begge maskinene kjører Linux bruker Pi Debian imens Jetson nytter Ubuntu. Der Raspberry Pi har et veldokumentert fora på nettet der man finner svar på de fleste spørsmål, er Jetson ny og lite brukt i forhold. Derfor er det vanskeligere å finne svar på spørsmål og utfordringer man har med å flytte programmet over til Jetson. Det vil her bli gjennomgått de spesifikke forskjellene i overgangen.

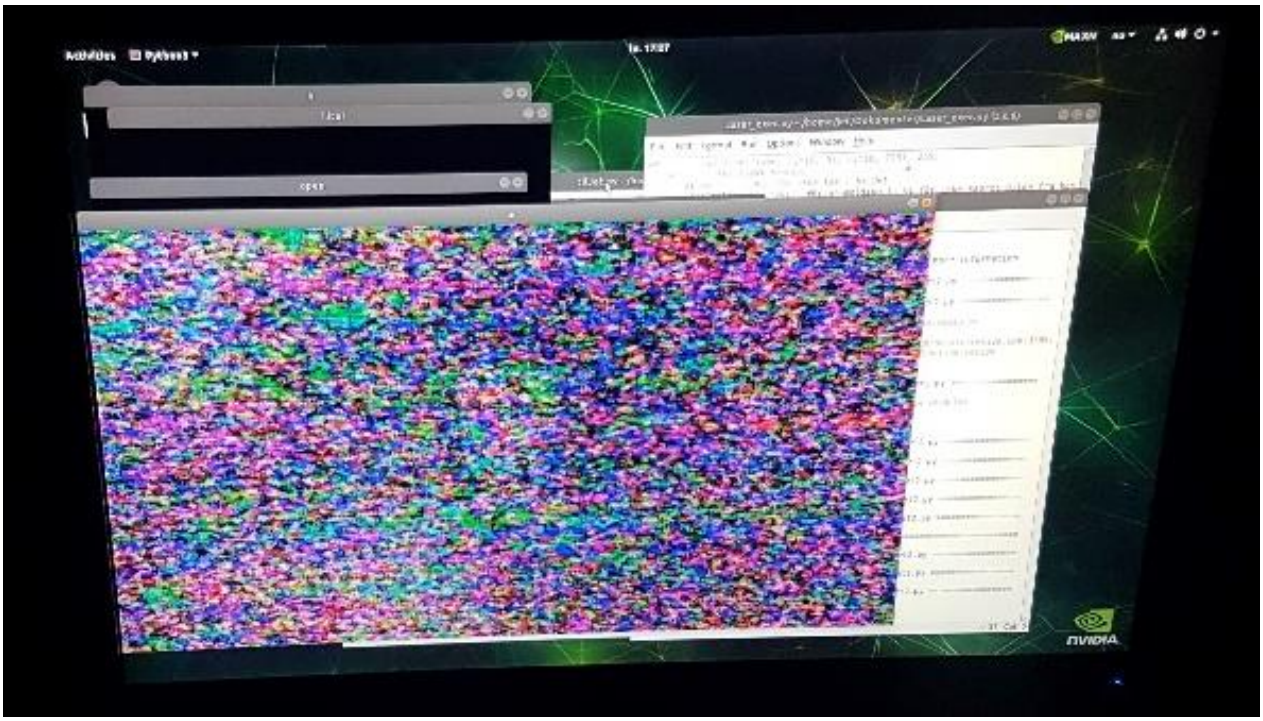
Den første utfordringen var at Ubuntu baserer seg på et pipeline-system for å hente ut kamera-feeden. Løsningen finnes på Jetsons eget forum, men denne skapte igjen problemer på andre måter.

```
def gstreamer_pipeline (capture_width=1280, capture_height=720, display_width=1280, display_height=720, framerate=60
    return ('nvarguscamerasrc ! '
    'video/x-raw(memory:NVMM), '
    'width=(int)%d, height=(int)%d, '
    'format=(string)NV12, framerate=(fraction)%d/1 ! '
    'nvvidconv flip-method=%d ! '
    'video/x-raw, width=(int)%d, height=(int)%d, format=(string)BGRx ! '
    'videoconvert ! '
    'video/x-raw, format=(string)BGR ! appsink'  % (capture_width,capture_height,framerate,flip_method,display_width

cap = cv2.VideoCapture(gstreamer_pipeline(flip_method=0), cv2.CAP_GSTREAMER)
```

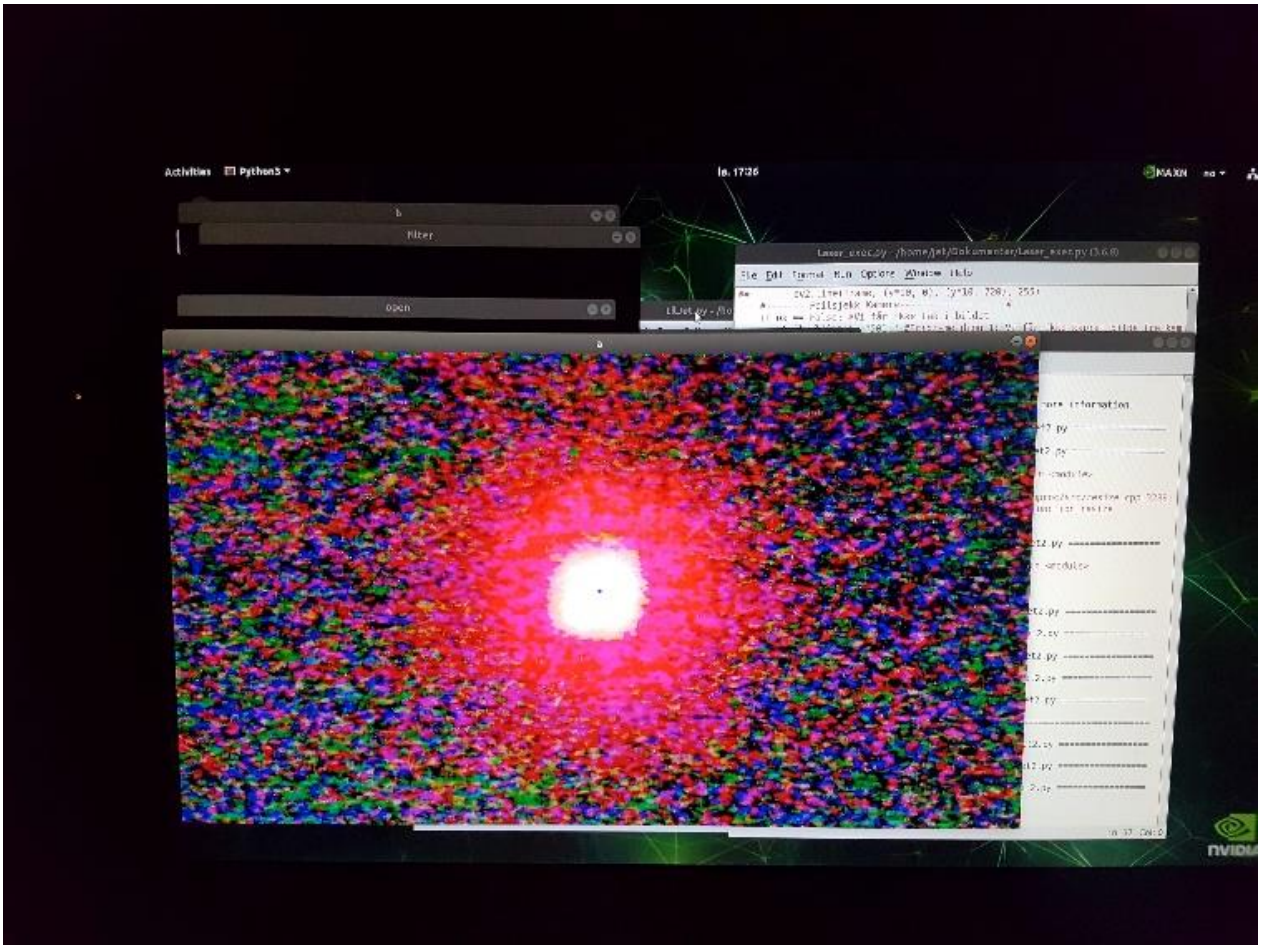
Figur 97: Løsningen for å "snappe" bilde fra kamera-feeden i Ubuntu

Slik Ubuntu er satt opp med OpenCV vil en ikke klare å benytte seg av diverse funksjoner som kan endre innstillingene på selve kamera. Eksempelvis hvis man ønsker å endre kontrasten i bildet vil dette ikke fungere med bruk av OpenCV sine funksjoner. Problemet var at man ikke hadde kunnskap på hvordan dette skulle gjøres, og det var enda mindre nyttig hjelp å finne på nettet. Vi ble derfor nødt til å stille et spørsmål selv i Jetsons forum i håp om at noen kunne hjelpe. Dette tok selvsagt tid, og var på dette tidspunktet ikke noe en forventet å måtte fikse.

Årsaken til at man må endre kamerainnstillingene baserer seg på hvordan kamerafeeden hos Jetson er ulik Pi når det er mørkt. Pipelinen til Ubuntu har høy forsterkning på signalet kamera bearbeider. Som en følge av dette vil den lage falske positive. Når programmet hentet ut feeden første gang fikk man følgende resultat:



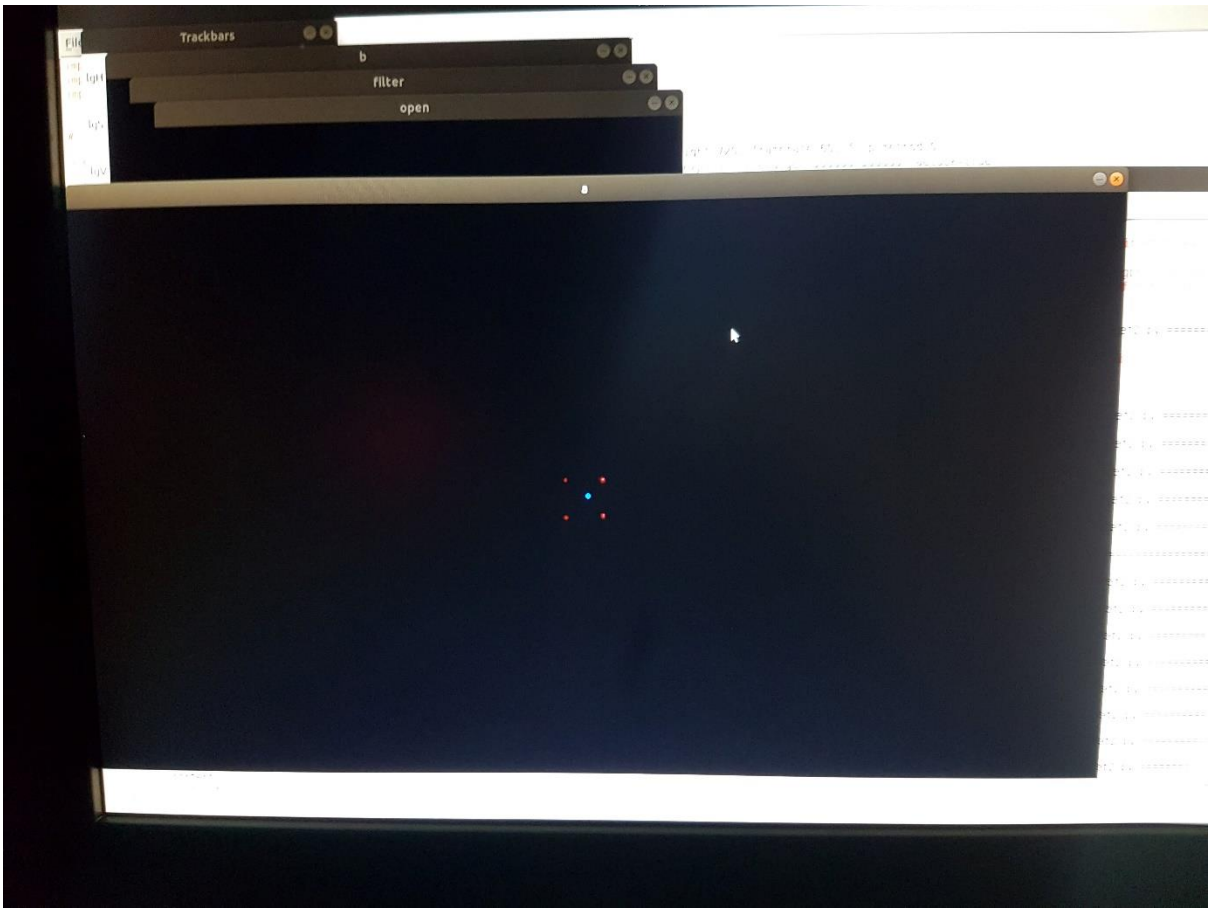
Figur 98: Utklipp av kamera-feeden når vi har kamera i et helt mørkt rom



Figur 99: Utklipp av kamera i samme mørke rom når vi slår på de fire laserpunktene. Ingen filter vil klare å skille ut de fire punktene

Hvor bildet øverst egentlig skal være helt mørkt, og bildet til høyre er med laserne på der man egentlig skal se fire laserpunkter. Når det er mørkt og lite lys vil altså den høye forsterkningen gi feil, og selv ikke det beste filteret man har for *inRange* vil klare å skille ut laserne da de er en samlet klump på bildet. Spørsmålet vi stilte i forumet var hvordan en kunne endre eksponeringstiden og senke forsterkningen[1].

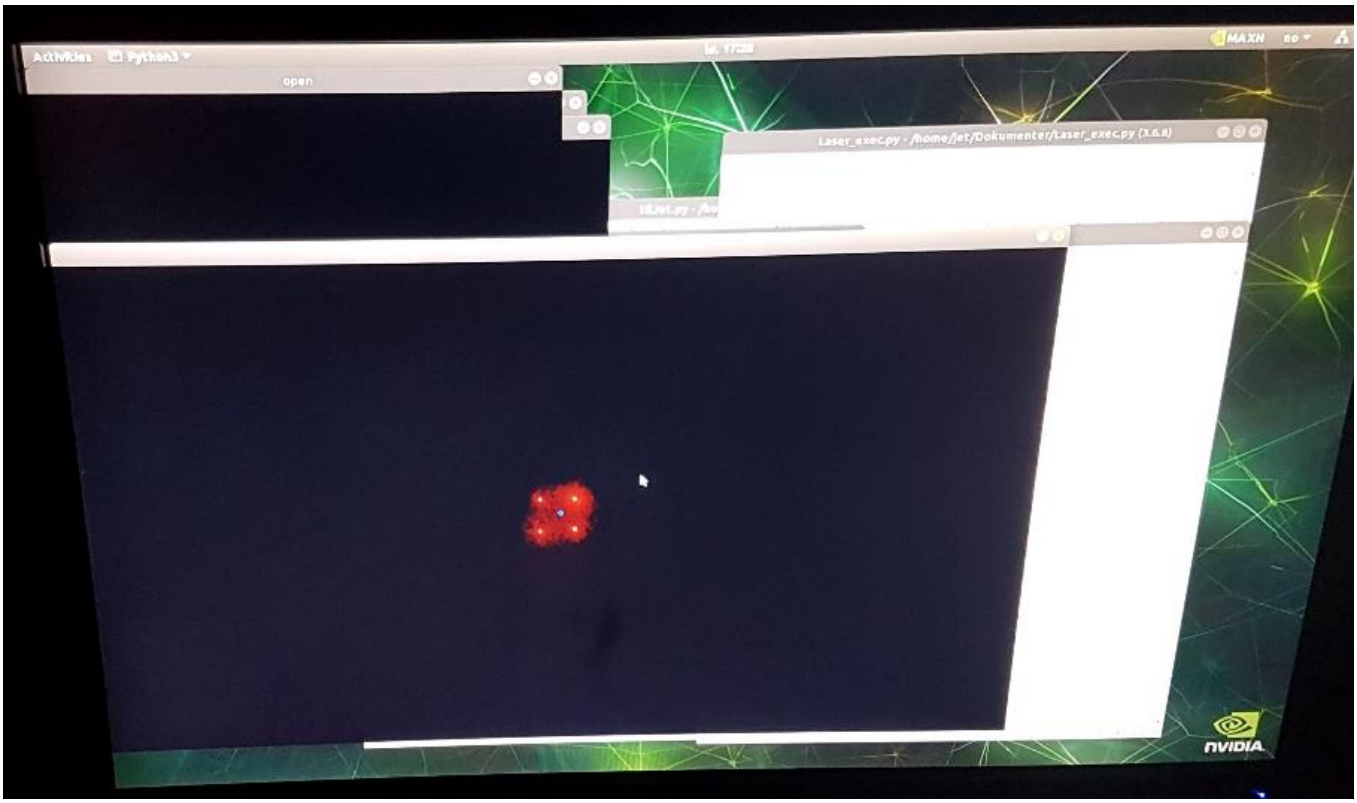
Etter å ha fått hjelp fikk man endret eksponeringstiden og senket forsterkningen lavt nok, med følgende resultat:



Figur 100: Kamera-feed etter at mani har senket forsterkningen og endret på eksponeringstiden til kamera

Som er et meget bra resultat da det eneste i bildet som har HSV-verdier ulik 0 er de fire laserpunktene, som gjør at det ikke er mulig for oss å få støy så lenge det ikke er annet lys i bildet. Man ser at forsterkningen har fjernet alt av de falske positive en hadde i bildet. Endring av eksponeringstiden gjør at man ikke får støy fra laserpunktene. Her er et eksempel når man har 10 ganger så høy eksponeringstid som over:

[1] https://devtalk.nvidia.com/default/topic/1064347/opencv-python-is-it-possible-to-change-camera-properties-from-gstreamer-so-that-the-camera-doesn-t-get-false-positives-/?offset=1#5389610



Figur 101: Eksempel med 10 ganger så høy eksponeringstid. Det er tydelig at laserpunktene lager støy i kamera-feeden

3.3.3 Logging

Det er ønskelig å ta bilde av eventuelle objekter programmet oppdager. De skal lagres i rekkefølge, slik at man kan se i etterkant hvilke bilder som hører til hvilket objekt.

```
if avstand < 300 and Nytt_objekt == False:
    #TA BILDE og kall det etter nummerering

    cv2.imwrite(str(Bildenr) + ".jpg", frame)
    Nytt_objekt = True
    Bildenr +=1
#Nytt_objekt settes til false når vi mister objektet lenger oppe
```

Figur 102: Når man har nytt objekt og liten nok avstand til å få et godt bilde, lagres bildet i mappen

Figur 103: Eksempel på bilde lagret som 1.jpg

3.3.4 Kommunikasjon

Fra Jetson skal man sende avstand til objektet både til Pi og Arduino, men til Arduino skal det også videresendes data den mottar fra Pi. Det brukes Tx/Rx-inngangene på begge brettene, samt en felles jording for å kommunisere. Jetson har to slike innganger som i utgangspunktet gjorde det mulig å kommunisere med både Arduino og Pi, men etter kort tid sluttet den ene inngangen å sende rett data og måtte skrotes. Løsningen var å ta i bruk en DAC23 USB til RS232-konverter:

Figur 104: DAC23 USB til RS232 konverter. Kilde:https://www.matiot.com/usb-to-rs232-converter-dac13

Her kan man koble på Tx/Rx og jord og sette USB-utgangen direkte inn i Jetson. Rent kodemessig vil man lage et seriell-objekt ved hjelp av biblioteket pyserial. Det er viktig at man setter samme baud-rate som de andre maskinene har satt(her 115200), slik at dataen sendes og mottas i rett klokketempo. Filene som det skrives og leses til er her ttyS1, men det er ikke alltid man har kontakt med Arduino/Pi og må derfor robustgjøre programmet. Objektet settes i en try/except der man først prøver ttyS1, for så å prøve ttyS2 hvis det ikke fungerer. Siden dronen er avhengig av å ha seriellkommunikasjon for å fungere vil man avslutte programmet skulle en ikke få kontakt med noen av dem. Årsaken til at man avslutter programmet kommer frem senere.

```
try:
    ser = serial.Serial('/dev/ttyS1',115200)  #Main
except serial.serialutil.SerialException:
    ser = serial.Serial('/dev/ttyS2', 115200) #Backup
except:
    #Avslutter programmet, Ubuntu restarter da Python-script
    time.sleep(5)
    sys.exit()
```

Figur 105: Oppkobling av seriell-kommunikasjon mellom enhetene

Hvis denne feilen oppstår skjer det ved oppstart av maskinene, altså før man har montert dronen helt sammen. Dette gjør det mulig å feilsjekke og avvente oppstart til alt fungerer.

En annen utfordring som dukket opp var at man mistet tilgangen til å benytte software-inngangen til Tx/Rx-inngangen mellom Jetson og Pi, originalt ttyS0. Et av forsøkene på å løse dette var å legge brukeren til i gruppene som styrte filen, men etter kort tid ble rettighetene der også inndratt. Forsøk nummer to var å sende kommandoen «sudo chmod 666 ttyS0» hvert halve sekund, men dette var heller ikke en god løsning da loopen til programmet oppdateres raskere. Man kunne prøvd å kjøre kommandoen enda hyppigere, men det ville ta enda mer prosessorkraft ut til noe unyttig som ikke skal være et problem. Vårt tredje forsøk var å feilsøke mer i detalj, der det så ut som en system-applikasjon frosset tilgangene. Utfordringen her var at man ikke fikk mulighet til å stanse applikasjonen fra å kjøre, slik at den fortsatt frosset tilgangen. Det siste forsøket som man lyktes med var å flytte ttyS0-filen til ttyS1. Siden det er en system-applikasjon som bruker ttyS0 vil man med å flytte den over til ttyS1 ikke miste tilgang da system-applikasjonen ikke benyttet seg av denne. Siden system-applikasjonen i utgangspunktet ikke skal drive med noe seriellkommunikasjon vil ikke Jetson bli påvirket av at den ikke lenger har noen fil å benytte. Løsningen krever heller ikke at man oppdaterer tilgangen kontinuerlig, men må gjøres ved oppstart.

Mottak av data

Det var problemer med å få sendt data *fra* Ardunio til Jetson, noe som gjør at det kun er fra Pi man mottar. For mottak følger man protokollen beskrevet tidligere, men må huske å skille på kommandoer og sensordata.

For sensordataen skal man motta posisjon for å videresende til Arduino. Denne vil i motsetning til kommandoer være kodet med en «P» foran den faktiske sensordataen. Dataen som mottas er en tekststreng som inneholder data separert med komma pluss at man har en P en må kvitte seg med. Løsningen er at man fra Pi har en komma etter «P» slik at en kan separere opp tekststrengen etter komma, for så å konvertere hver og en av dem til float og legge dem til i de designerte posisjonsvariablene:

```
elif FraPI[0] == "P":  #Mottak av posisjon fra HjernePI
    _,posx,posy = FraPI.split(",")
    PosX = float(posx)
    PosY = float(posy)
```

Figur 106: Splitting av tekststreng slik at man får lagret posisjonsdataen i separate variabler

Denne metoden vil være lik for hvordan Pi henter ut mottatt serielldata fra Jetson.

For start og stopp ønskes det at Jetson bekrefter på mottatte meldinger fra Pi. I motsetning til for Arduino vil Jetson fortsette med resten av programmet og ikke vente på at Pi skal sende en bekreftelse på rett kommando. Årsaken er at Jetson ikke fungerer som hjerne men bare som en blind videresender av data og beregning av avstand.

```
elif FraPI == "STOPP": #Oppstart er av
    Klar_til_sending = False
    #Kontakt_med_PI = True
    RUND = False

    feilsjekk_tilPI = serielloverforing_msg_received("STOPPET")
```

Figur 107: Ved mottak av "STOPP" fra Pi sender vi "STOPPET" i retur slik at den vet vi har mottatt kommandoen

Sending av data

Det skal sendes avstand til objekt til Pi og noen ganger bekreftelser på mottatte kommandoer. Bekreftelsen vil selvsagt ha hovedpri, noe som løses ved å ta i bruk funksjonen *in_waiting* fra pyserial som sjekker hvor mange bytes som venter på å bli sendt ut. For sending av sensordata til Pi vil programmet være avhengig av at *in_waiting* returnerer 0. I motsetning til sensordata vil bekreftelse på kommandodata sendes uavhengig. Da den forekommer først i loopen vil den også kunne sette en «sperre» på sending av sensordata

hvis det går tregt med overføringen. Unntaket for overføring av sensordata er når det har gått 20 sekunder siden sist sending, som er for mye med tanke på mulige objekter i veien.

```
if (ser.in_waiting == 0  and abs(siste_sending_ms_PI - time.process_time()) > 20)
    ser.flush()
    ser.reset_input_buffer()
    feilsjekk_tilPi = serielloverforing(avstand)
    siste_sending_ms_PI = time.process_time()
```

Figur 108: Kriterie for å sende sensordata til Pi

For Arduino er det mye lettere da det kun foregår enveis-kommunikasjon. Samtidig har man i innledningen til totalsystemet snakket om hvordan Jetson oppdateres raskere enn Arduino. Man kan derfor ikke sende data på måfå uten noen form for begrensning fra Jetson sin side. Det benyttes en allerede nevnt ledning mellom GPIO-pin på Jetson og Arduino's digitale pin. Fra Jetson sin side vil man avvente å sende mer data til Arduino til den får logisk høy/sann verdi på GPIO-pinen:

```
if PinInput_ARDU == 1: #Hvis Arduino rapporterer klar til å motta ny data
    feilsjekk_tilArdu = serielloverforing_ardu(avstand, PosX, PosY)
```

Figur 109: Sending til Arduino som avhenger av at Arduino sier den er klar til å motta ny data

Det legges ved et ekstra kriteria om at meldingspakken må være ulik den forrige. Slik sørger man for at det ikke sendes unødvendige datapakker:

```
def serielloverforing_ardu(avstand, PosX, PosY):
    global message
    hjelp = "<"+ str(int(avstand)) + ","  + str(int(PosX)) + "," + str(int(PosY)) + ">"
    if hjelp != message:
        message = hjelp
        try:
            tilArdu.write(message.encode())
```

Figur 110: Ekstra kriteria om at meldingspakken må være ulik den forrige for at Jetson skal sende

Før tok det tid å få sendt data til Arduino, som igjen ble et problem slik styringssystemet er designet. Etter løsningen oppleves det at Arduino klarer å motta data(med lite tap) opp til hvert 0.25 sekund beregnet fra Jetson, noe som gir god nok oppdatering.

3.3.5 Program som restarter ved eventuelle feil

Det ble nevnt at Jetson avslutter programmet hvis det ikke får kontakt med Pi eller Arduino via seriell. Årsaken er at man tar i bruk Python-programmet til Alex Kras som restarter et gitt program når det stopper/krasjer:

```
from subprocess import Popen
import sys

filename = sys.argv[1]
while True:
    print("\nStarting " + filename)
    p = Popen("python " + filename, shell=True)
    p.wait()
```

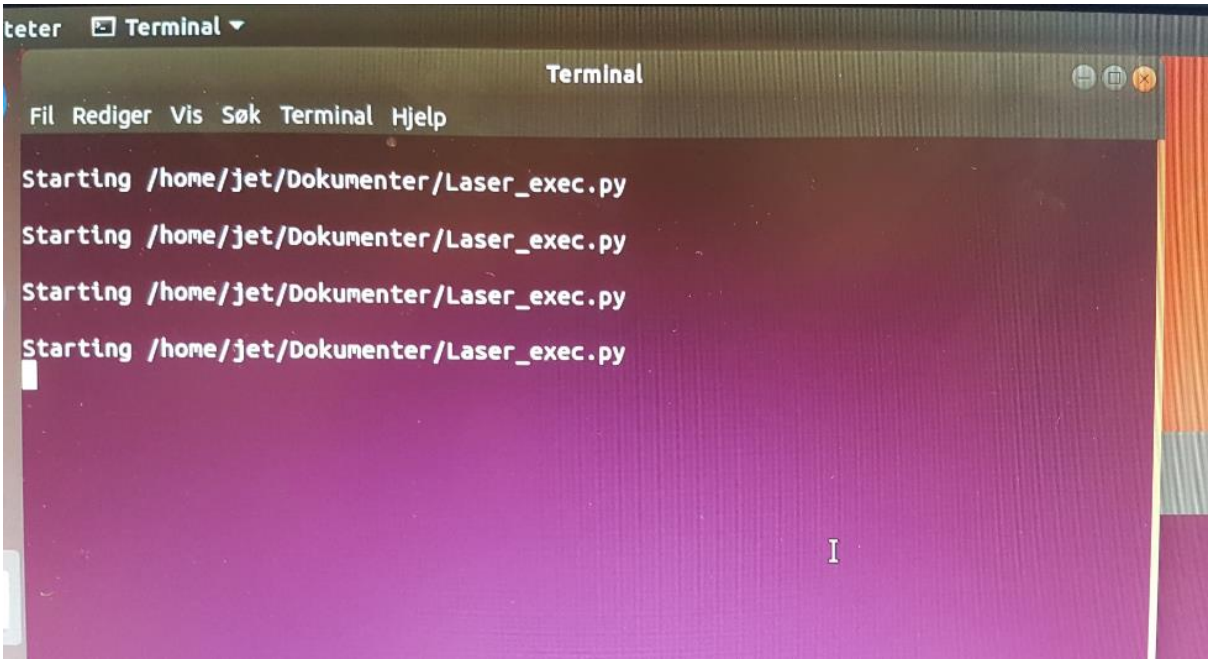
Figur 111: Enkelt Python-script som tar inn spesifisert Python-program som argument og restarter det hvis det stopper. Kilde: https://www.alexkras.com/how-to-restart-python-script-after-exception-and-run-it-forever/

Vi kaller vårt Python-script for «Laser_exec.py» og Kras' script for «Gjenoppliv.py». Etter å ha gjort Laser_exec.py executable via kommandoen «sudo chmod +x Laser_exec.py» kan man utføre følgende kommando:

```
gnome-terminal --window-with-profile=jet  -e "/home/jet/Dokumenter/Gjenoppliv /home/jet/Dokumenter/Laser_exec.py"
```

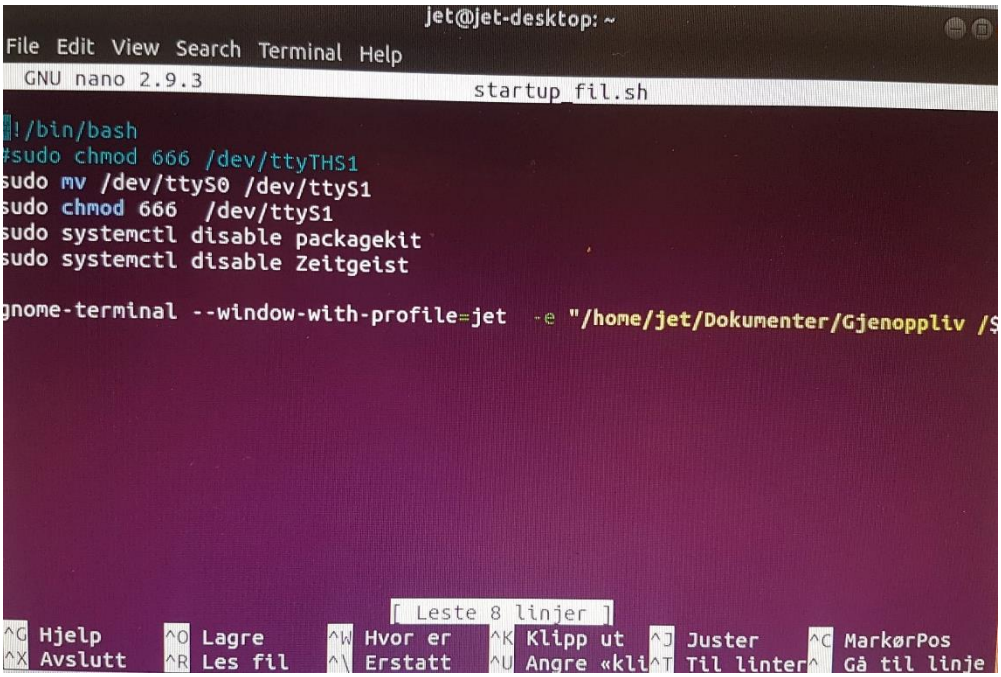
Figur 112: Kommando for å starte opp Gjenoppliv med Laser_exec som argument

Hvor gnome-terminal indikerer oppstart av et nytt terminal-vindu i Ubuntu, hvor man spesifiserer at det er vår bruker som skal utføre kommandoen(jet). «-e» gir oss mulighet for å utføre en kommando-streng i det terminalen startes opp, altså å «skrive» en kommando til terminalen. Ubuntu skal starte Gjenoppliv.py med Laser_exec.py som argument, der det er viktig at man spesifiserer filbanen til begge scriptene. Resultatet ser slikt ut:



Figur 113: Oppstart og restarting av Laser_exec.py. Bildet er tatt når DAC23 ikke var koblet til slik at scriptet avslutter

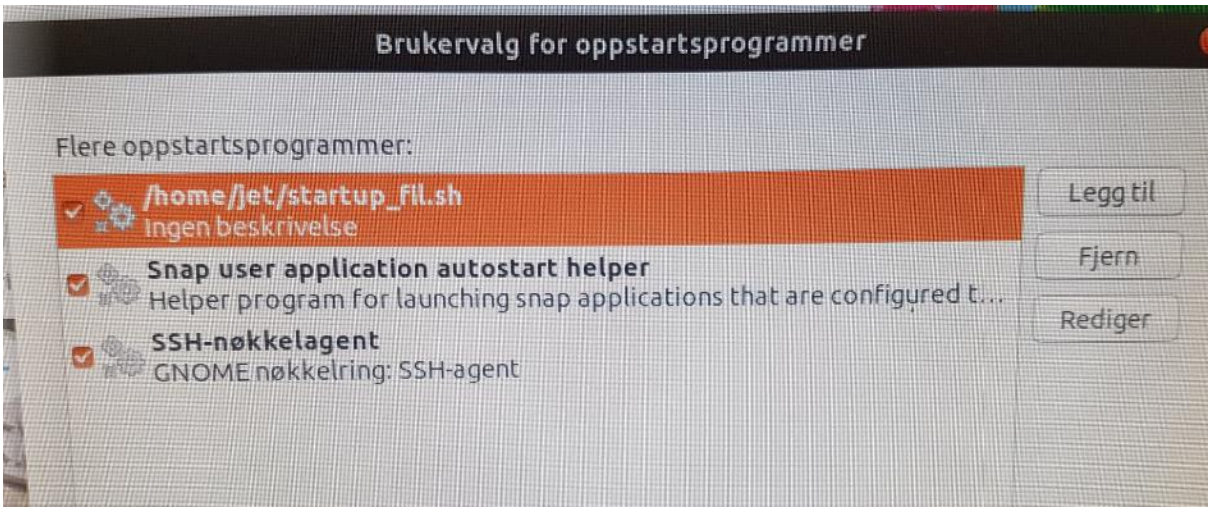
Når dronen startes opp ønskes det at kommandoen utføres automatisk, slik at man slipper å gå inn på maskinen via en skjerm og tastatur for å gjøre det. Dette gjør at dronen automatisk kan starte opp systemet når den kobles til strøm. Ubuntu har en system-applikasjon som heter «Startup Applications» der man kan legge til programmer en ønsker at skal utføres ved oppstart. Det er nevnt i 3.3.4 at ttyS0 måtte flyttes til ttyS1 ved oppstart, som også kan gjøres her. Ved å lage en tekstfil kalt: «startup_fil.sh» som gjøres executable kan man legge alle kommandoene som skal gjøres ved oppstart der:



Figur 114: Utklipp av kommandoene som kjøres ved oppstart

De to første kommandoene er for å benytte oss av ttyS1 istedenfor ttyS0. De to neste er et resultat av feilsøking da Jetson plutselig ikke hadde mer diskplass igjen, noe som skyldtes de to programmene Packagekit og Zeitgeist. Programmene sendte suksess-meldinger til log-filene på systemet hvert sekund som til slutt gjorde filen ekstremt stor. Vi var derfor nødt til å skru av programmene, da de ikke er nødvendige for Avstandsmålingen.

Det siste steget for å få scriptet og de andre kommandoene til å utføres ved oppstart er å legge filen «startup_fil.sh» til i listen til «Startup Applications»:



Figur 115: Oppstartsprogram lagt til i listen over programmer som kjører når Jetson starter

Når vi testet oppkjøringen sammen med scriptet for Pi oppsto det uforutsette problem. Hvis Jetson starter opp før Pi vil den ikke få stopp-signalet og starte opp. Videre hadde Jetson en tendens til å ikke motta deler av serielldataen etter at både Pi og Jetson var startet opp, slik at Jetson aldri utførte Avstandsmåling. Løsningen var å ta i bruk to GPIO-pins slik vi har gjort med Arduino. De to Pinsene har output ifra Pi, der den første sender logisk høy når Pi har startet opp scriptet sitt mens den andre sender logisk høy når Pi mottar start-signal fra Arduino. Den første sørger for at Jetson ikke starter opp før etter at Pi sitt script er klar:

```
#Avventer oppstart til HjernePI-program er initiert
while GPIO.input(13) == 0:
    time.sleep(1)
    print("venter på start")
```

Figur 116: Utsnitt av koden der Jetson avventer oppstart til Pi sender logisk høy

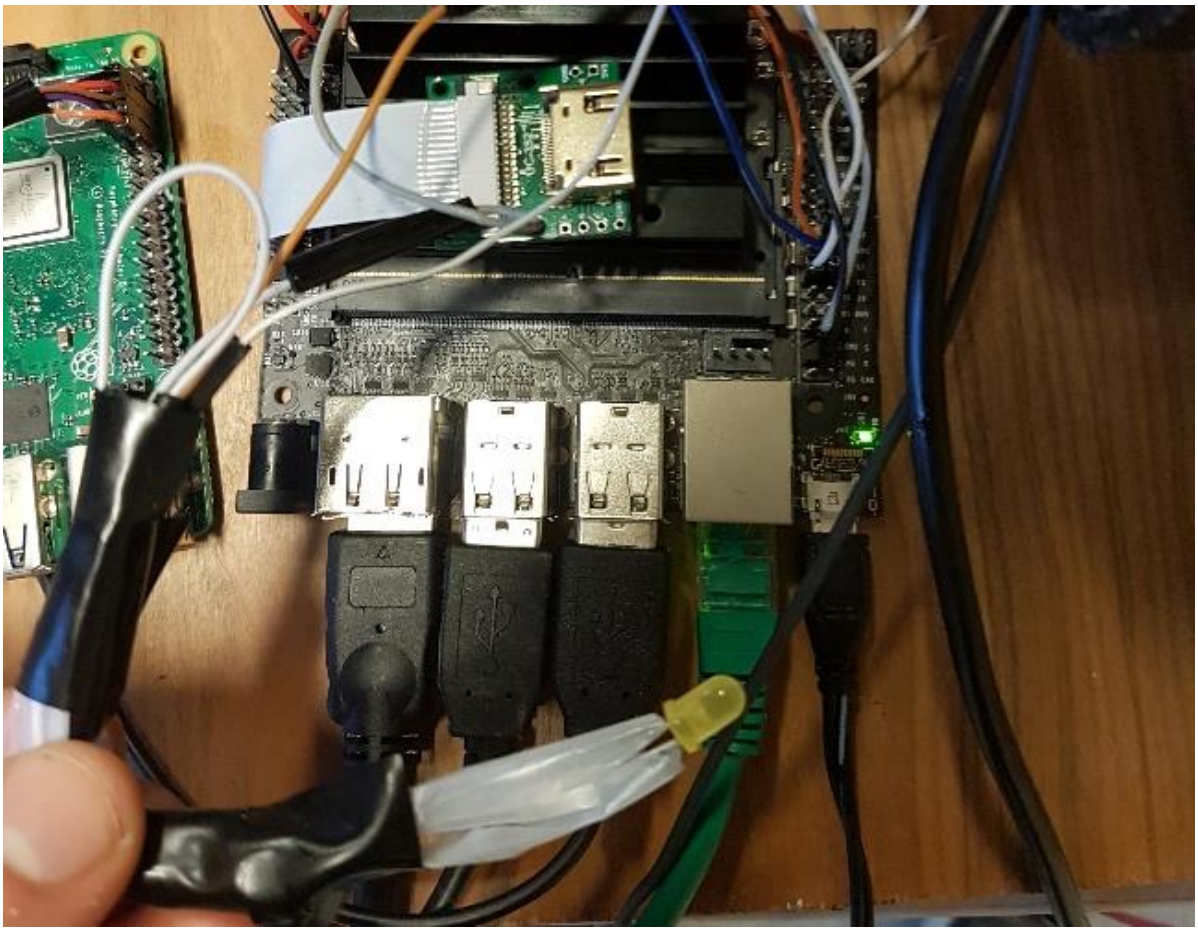
While-loopen ligger før hoveddelen av programmet utføres, slik at scriptet aldri starter opp før Pi har sendt logisk høy. Pin nummer to har til hensikt å sørge for at seriellkommunikasjonen fungerer mellom de to. Når det oppsto et slikt problem viste testingen at løsningen var å restarte Jetson. Hvis Pi sender start-melding til Jetson vil den sette pinen til logisk høy, som igjen aktiverer en boolsk variabel kalt *Kontakt_med_PI* hos Jetson:

```
#---------Feilsjekk seriell PI/JET------------#

if Kontakt_med_PI == False: #Hvis vi enda ikke har fått noen melding av PI
    print("FEIL")
    Kontakt_med_PI_teller +=1
    if Kontakt_med_PI_teller >= 100:
        GPIO.output(19,GPIO.LOW)
        time.sleep(5)
        sys.exit()  #Avsluttet og restarter programmet
#Hvis PI sender oss en melding, resetter vi variabelen
#Dette brukes så i argumentet over
if GPIO.input(16) == 1:
    feilsjekk_tilPI = serielloverforing_msg_received("START")
    Klar_til_sending = True
```

Figur 117: Når Jetson leser logisk høy settes variabelen Kontakt_med_Pi til sann. Denne aktiverer så i tur en teller som etter 100 gjennomføringer avslutter programmet

Når Jetson mottar faktisk serielldata fra Pi vil variabelen settes til usann igjen. Oppsettet fungerer da slik at Pi gir Jetson beskjed via GPIO-pin at den har sendt en melding, der Jetson avventer å motta meldingen. Hvis den ikke mottar den på 100 gjennomføringer/iterasjoner vil den avslutte programmet slik at det kan restartes. Slik vil man ved oppstart forsikre seg om at de to maskinene kommuniserer rett med hverandre når alt startes opp.

Det ble understreket at når man kobler maskinene opp til strøm så skal en unngå å måtte gå inn via skjerm og tastatur for å sjekke at alt fungerer. Det legges derfor ved en tredje Pin koblet til en led-lampe som skal indikere om Jetson har kontakt med Pi. Den settes til å lyse ved oppstart slik at man kan se at maskinen har startet scriptet sitt. Den vil derimot slutte å lyse hvis en skal avslutte programmet eller hvis alt fungerer som det skal. Merk at før man avslutter programmet vil det ha en delay på fem sekunder. Dette gjør at hvis man opplever at lyset går av i fem sekunder før det går på igjen, og at dette gjentar seg over lengre tid, betyr det at noe ikke fungerer skikkelig og ikke blir fikset. Slik kan man allerede da fastslå at noe er galt og starte med feilsøking istedenfor å innse problemet under kjøring av dronen.



Figur 118: Ledpin som lyser/ikke lyser avhengig av om ting fungerer

Hvis lyset derimot går av og forblir avskrudd, så betyr det at kommunikasjonen mellom Jetson og Pi fungerer og man kan starte opp. Når en først har en slik anordning er det greit å inkludere kriteria om at kamera må fungere for at den skal slutte å lyse.

```
ok,frame = cap.read()  #Kaprer neste bilde
#--------Feilsjekk Kamera-------------------#
if ok == False: #Vi får ikke tak i bildet
    feilmelding += "501," #Error-melding 1: Vi får ikke kapret bilde fra kamera
    #Prosedyre for å restarte programmet
    not_ok_teller +=1 #Ant ganger på rad uten bilde
    time.sleep(0.25)

    if not_ok_teller == 50: #Vi har gått for lenge uten bilde, vi resetter script
        GPIO.output(19,GPIO.LOW)
        time.sleep(5) #avventer 5 sekund
        sys.exit()
#-----------------------------------------#
```

Figur 119: Sjekk om at kamera fungerer skikkelig og gir oss bilder, med avslutning hvis det ikke er tilfelle

3.4 Software Pi

Raspberry Pi(Pi) skal ta for seg følgende oppgaver:

- Sensordata
    - Posisjonsmåling i x og y
- Logging
    - Plotting av posisjon i x, y og z
- Styring
    - Reagere på stopp/start fra Arduino
    - Styre dronen
    - Reagere på hindringer
- Kommunikasjon
    - Sende kommando
    - Mottaking av sensordata
    - Overføring sensordata

Først er det interessant å se på hvordan Pi skal løse alle disse oppgavene samtidig. Jetson skal kun hente ut sensordata og drive med seriellkommunikasjon som gjør at den enkelt kan kjøre alt i samme loop. For Arduino skal den gjøre det samme som Jetson pluss det å gi styringssignal til motorer og servoer, men dette går også fint an å kjøre i samme loop. For Pi skal den gjennomføre to oppgaver synkront og nesten helt uavhengig av hverandre. Da Pi skal fungere som hjerne for hele systemet, vil den være nødt til å gjennomføre denne oppgaven på siden imens den kontinuerlig innhenter egen og andres sensordata. Ettersom dette er en jobb som blir veldig krevende å kjøre i samme loop, er det best å ta i bruk threading.

3.4.1 Threading som oppsett

For å bruke definisjonen fra tutorialspoint.com er en thread eller tråd en mulighet for å kjøre flere programmer/oppgaver samtidig. Selv om Python ikke faktisk kjører oppgavene samtidig vil den nytte dødtid blant de ulike oppgavene til å kjøre andre oppgaver og være mer effektiv. Årsaken til at man har valgt å bruke tråd som oppsett er at de fleste av oppgavene Pi skal kjøre vil drive med mye venting. Med venting menes det at oppgavene avventer informasjon eller data fra en annen oppgave eller datamaskin for å gjennomføre oppgaven sin. Ved å tråde en oppgave som skal skru av systemet i det øyeblikket den får stopp-meldingen fra Arduino, vil den bruke lite prosessorkraft og oppdateres mye raskere enn ved vanlig loop-løsning. Tråder vil også ha enklere for å dele informasjon mellom hverandre og bruker mindre minne enn vanlige funksjoner. Videre vil det være mer oversiktlig og enklere å separere posisjonsmåleren fra resten av koden, som igjen gjør at den går mye raskere enn

hvis den skulle være inkludert i en hoved-loop. Posisjonsmåleren er på lik linje med avstandsmåleren avhengig av hyppig bildeoppdatering. I en liten programkode som for Jetson går det helt fint å kjøre den i loopen, men for posisjonsmåleren i et så stort og komplekst program som Pi skal kjøre, vil det kunne senke oppdateringen. En siste fordel som egentlig ikke blir brukt i selve løsningen var at selv om en av oppgavene skulle krasje som følge av en feil, vil resten av oppgavene fortsette å kjøre som normalt. Dette er optimalt for feilsøking og til bruk i tester som tok lang tid å gjennomføre uten å måtte starte på ny bare fordi vi mistet én oppgave.

Hver av trådene skal kjøres kontinuerlig så lenge programmet er startet opp. Dette innebærer at man mister evnen til å avslutte programmet uten å gjøre det med «kraft», altså tvinge det til avslutning. Det er derfor viktig at man ved en eventuell stopp-melding fra bruker, resetter alle variabler og nullstiller til å kunne kjøre en gang til. En må også være påpasselig med at to oppgaver kan ha aksess til samme variabel «samtidig» og dermed endre den. Vi opplevde for eksempel at en if-setning ble godkjent som følge av at variabelen som lå i kriteriet stemte overens, mens at man midt i den samme if-setningen opplevde at variabelen hadde endret verdi til noe som ikke hadde oppfylt kriteriet. Et slikt problem kunne blitt løst ved å ta i bruk ulike «låser» som hindrer endring av variabler når andre bruker dem. Men det ble heller valgt å begrense hvilke variabler som ble brukt av flere oppgaver.

Utenom å hente inn biblioteket for threading trenger man Queue. Hensikten er å holde orden på prosessene slik at de venter til det blir deres tur, og at hovedprogrammet derfor ikke krasjer. Disse to er nok til at man kan drive å tråde diverse oppgaver.

3.4.2 Sensordata

Raskere bildebehandling

Pi skal hente ut posisjonsdata i x- og y-retning. På lik linje med avstandsmåling nyttes OpenCV og Python for oppgaven. Men som beskrevet i 2.2 ble maskinvaren endret fra en Pi til Jetson da den oppnår ekstremt mye bedre hastighet og dermed bildeoppdateringsrate. Vi har ikke mer enn én jetson og må nøye oss med å bruke en Pi til denne operasjonen, men kan allikevel gjøre et par endringer for å bedre oppdateringsraten. Da det å hente ut, behandle og vise bilder krever mye datakraft og tid vil slike dataprosesser gå tregt på små enkortsmaskiner som Pi. Når prosessene bruker lengre tid, vil også loopen gå tregere som resulterer i at man klarer å lese færre bilder per sekund. Dette betyr at bildeoppdateringsraten blir treg, noe som ikke er gunstig for posisjonsmåleren som trenger god oppdateringsrate.

Ved å flytte oppgaven med å lese ut bilde til en egen tråd og dermed fordele de tunge oppgavene vil en kunne få prosessene til å gå raskere. Vi har tatt i bruk artikkelen til Najam Syed[7] som beskriver problemet godt:

> "*While it's important that the image processing portion of a video processing pipeline be streamlined, input/output(I/O) operations also tend to be a major bottleneck. One way to alleviate this is to split the computational load between multiple threads."(Syed, 2018)*

I samme artikkel blir det foreslått ulike måter å lage tråden på, der vi valgte å ta i bruk løsningen som innebar å lage en klasse.

Bildeprosess

Det må i hovedprogrammet defineres et klasseobjekt:

```
#POSTRACK
vid_get = VideoGet(0).start()
```

Figur 120: Definerer et nytt objekt fra tråden som henter ut bildet

I 2.3 ble det beskrevet grovt hvordan posisjonsmåleren skal beregne pixelendring. Løsningen baserer seg på å sammenligne forrige bilde med neste bilde for å se hvor visse like objekter/områder befinner seg.

Det finnes ulike typer Feature Matching(FM), og vi tar i bruk en som heter Brute-Force Matcher(BFM). Oppskriften for å få dette til er hentet fra OpenCV sine egne sider. For å forstå hvordan det fungerer er det viktig å ha en forståelse av hva en deskriptor er. I 2.3 nevnes det at FM lagrer informasjon om de ulike punktene/områdene i bildet og så sammenligner informasjonen i neste bilde. Denne informasjonen er lagret i en såkalt deskriptor, som er en vektor med konstant lengde. Hvilken informasjon som lagres avhenger av typen FM man velger, der oppgaven benytter BFM med en deskriptortype som heter Oriented-fast and Rotated Brief(ORB).

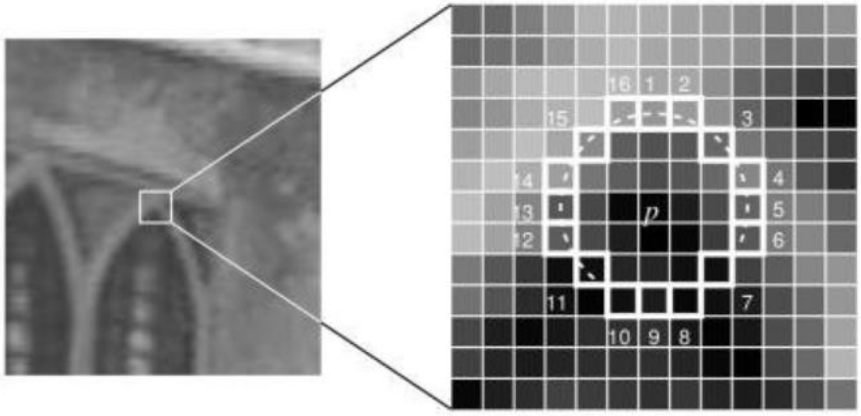
```
orb = cv2.ORB_create()
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

Figur 121: Konstruksjon av ORB og BFM, legg merke til bruken av Hamming og Crosscheck som blir forklart senere

ORB deskriptor

ORB er basert på to teknikker som heter FAST og BRIEF. FAST er den delen som velger ut hvilke punkter som skal være nøkkelpunkter og lagres for sammenligning. Måten punktene velges ut på er at FAST går gjennom hver pixel *p* i bildet og sammenligner lysstyrken til pixelen med en sirkel av 16 pixler som er i en sirkel rundt *p*. FAST kategoriserer så de 16

pixlene inn i grupper basert på om de er lysere, like eller mørkere enn p. Hvis 8 eller flere havner i lysere eller mørkere vil *p* kategoriseres som nøkkelpunkt, som vi kaller *n.* Dette er spesielt brukt til å lokalisere kanter og ender i bilder, som er enkle å finne igjen.



Figur 122: pixel p i fra et bilde, med de 16 pixlene som er i en sirkel rundt p. Kilde: https://medium.com/@deepanshut041/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

Etter at FAST har lokalisert nøkkelpunktene, er det BRIEF som skal konstruere deskriptoren til hver av dem. Ut fra nøkkelpunkt *n*, vil BRIEF definere et område rundt *n* som kalles for en patch. Dette er det området som BRIEF vil konstruere en deskriptor ut fra.

Figur 123: Utsnitt av et bilde der man fokuserer på et nøkkelpunkt. Bildet til høyre viser det definerte området rundt nøkkelpunktet(patch). Kilde: https://medium.com/@deepanshut041/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

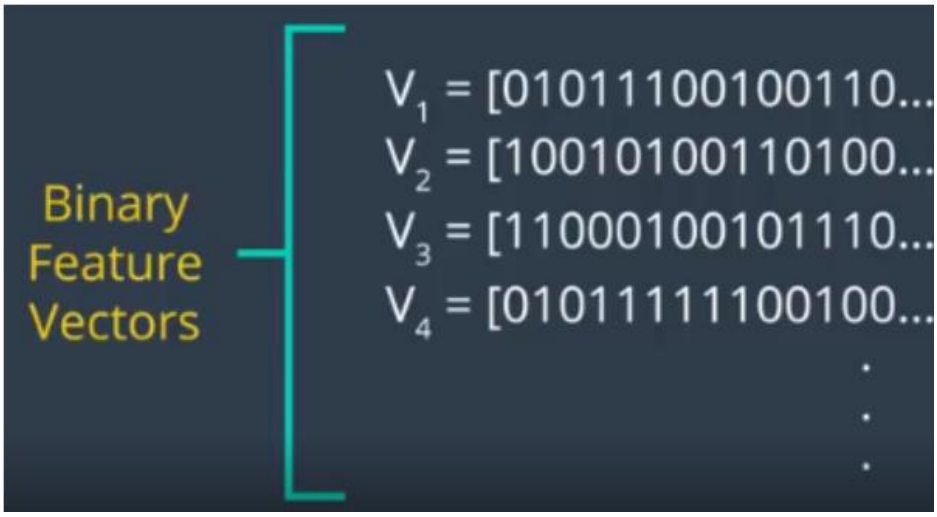
I patchen vil BRIEF velge ut to og to pixler for sammenligning. Det finnes ulike mønstre for å plukke ut parene, enten ved ulike tilfeldige utvelginger eller ved synkrone mønster. BRIEF bruker en form for tilfeldig utvalg(normaldistrubusjon) basert på at det gir best resultater, men

er ikke noe mer som gås i detalj på her. Et eksempel på to slike pixler vises til høyre i bildet over.

Etter å ha valgt ut et mønster bestående av alle pixel-parene(x,y) vil BRIEF sammenligne dem, og produsere følgende vektor:

$$\boldsymbol{A} = \begin{cases} 1 \ når \ p(x) < p(y) \\ 0 \ når \ p(x) \geq p(y) \end{cases}$$

Hvor p(x) og p(y) er lysintensiteten i de to pixel-parene og A er en binær vektor i størrelse 128-512 bits. Eksempel på slike vektorer ses under:



Figur 124: Eksempel på binære vektorer som lages ut ifra en sammenligning mellom pixel-parene vi har laget. Kilde: https://medium.com/@deepanshut041/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

Dette er feature-delen av deskriptoren. De resterende delene av den er en vektor som tar høyde for rotasjon i bildet, og den siste er informasjon om hvordan mønsteret brukt over ser ut.

Rent kodemessig brukes funksjonen detectAndCompute() til å beregne nøkkelpunkter og tilhørende deskriptorer for gammelt bilde og nytt bilde.

```
kp1, des1 = orb.detectAndCompute(frame, None)
kp2, des2 = orb.detectAndCompute(gammel_frame, None)
```

Figur 125: Uthenting av nøkkelpunkt(kp) og deskriptorer

Hvor *kp* er et objekt over alle nøkkelpunkter i x og y, mens *des* er en tabell bestående av vektorer som inneholder informasjonen om nøkkelpunktet.

Brute-Force Matching BFM

Beskrivelsen av BFM fra OpenCV's egne sider:

> *«It takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.»*

Det finnes ulike metoder å sammenligne deskriptorene på. For ORB anbefaltes det å bruke Hamming distance. Med distance, menes ikke avstand i pixler men heller ulikheten imellom deskriptorerene. Hamming utføres i to steg. Steg én er en *eksklusiv-eller(XOR)* operasjon mellom deskriptoren fra første bilde mot samtlige deskriptorer i andre bilde. Ønsket resultat fra XOR er:

$$Resultat = \begin{cases} 1\ hvis\ Deskriptorene\ har\ ulik\ verdi \\ 0\ hvis\ Deskriptorene\ har\ lik\ verdi \end{cases}$$

Som for eksempel:

$$Deskriptor_{bilde1} = [1011]\ \ og\ \ Deskriptor_{bilde2} = [0011]$$

$$Deskriptor_{bilde1} \oplus Deskriptor_{bilde2}$$

$$1011 \oplus 0011 = 1000$$

For steg to summeres antall ulikheter imellom deskriptorene:

$$\sum Deskriptor_{bilde1} \oplus Deskriptor_{bilde2}$$

Som for eksempel:

$$1 + 0 + 0 + 0 = 1$$

Som i tilfellet til eksempelet er de to deskriptorene veldig like. Dette var et eksempel mellom én av deskriptorene som er i bilde nummer to og kun for fire bits. I virkeligheten vil OpenCV kjøre gjennom samtlige deskriptorer i bilde 2 for hver av deskriptorene i bilde 1, der vektorstørrelsene er mellom 128 og 512 bits.

I figur 125 nevnes det at crosscheck skal nyttes. Crosscheck sørger for at de eneste matchene som returneres er de der deskriptor i bilde 1 og bilde 2 har den motsatte som beste match. Med andre ord, hvis deskriptor *a* i bilde 1 har sin beste match som deskriptor *b* i bilde 2 og *b* har sin beste match som *a,* så vil matchen returneres. Hvis ikke, vil den forkastes. Dette sikrer oss mer presise og solide resultater i sammenligningen.

```
try:
    matches = bf.match(des1,des2)
    matches = sorted(matches, key = lambda x:x.distance)
    matching_result = cv2.drawMatches(frame, kp1, gammel_frame, kp2, matches[:5], None, flags=2)
    matches = matches[:5]
except:
    matches = []
```

Figur 126: Koden for sammenligning av de to bildenes deskriptorer, sortering etter lavest distance og avgrensning til kun de 5 beste resultatene

Som bildet viser sammenlignes de to deskriptorene og matchene lagres i variabelen *matches*. I 2.3 nevnes det at man begrenser antall nøkkelpunkter til de fem beste, hvor med best menes de som har minst forskjell i distance, altså mest like.

Beregning av pixelendring

I 2.3 ble det målt hvor bra snittet, den med minst endring og den beste matchen presterer. Man valgte ut fra målingene å bruke den beste matchen til posisjonsmåling. Så man trenger en metode for å finne igjen nøkkelpunktet den beste matchen tilhører, noe man finner i objektet matches:

What is this Matcher Object?

The result of `matches = bf.match(des1,des2)` line is a list of DMatch objects. This DMatch object has following attributes:

- `DMatch.distance` - Distance between descriptors. The lower, the better it is.
- `DMatch.trainIdx` - Index of the descriptor in train descriptors
- `DMatch.queryIdx` - Index of the descriptor in query descriptors
- `DMatch.imgIdx` - Index of the train image.

Figur 127: Innholdet til objektet "matches" som er har laget ved bruk av BFM. Kilde: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html

Her finner man altså den tilhørende indeksen til deskriptoren som korrelerer direkte til nøkkelpunktet. Merk her at bilde 1 kalles «train image» og bilde 2 kalles «query image». Ved å bruke indeksen har man tilgang til nøkkelpunktet i bilde 1, og det samme nøkkelpunktet i bilde 2.

Nøkkelpunktet *kp* er som sagt et objekt med ulik informasjon:

Public Attributes

float	**angle**
int	**class_id** object class (if the keypoints need to be clustered by an object they belong to) More...
int	**octave** octave (pyramid layer) from which the keypoint has been extracted More...
Point2f	**pt** coordinates of the keypoints More...
float	**response** the response by which the most strong keypoints have been selected. Can be used for the further sorting or subsampling More...
float	**size** diameter of the meaningful keypoint neighborhood More...

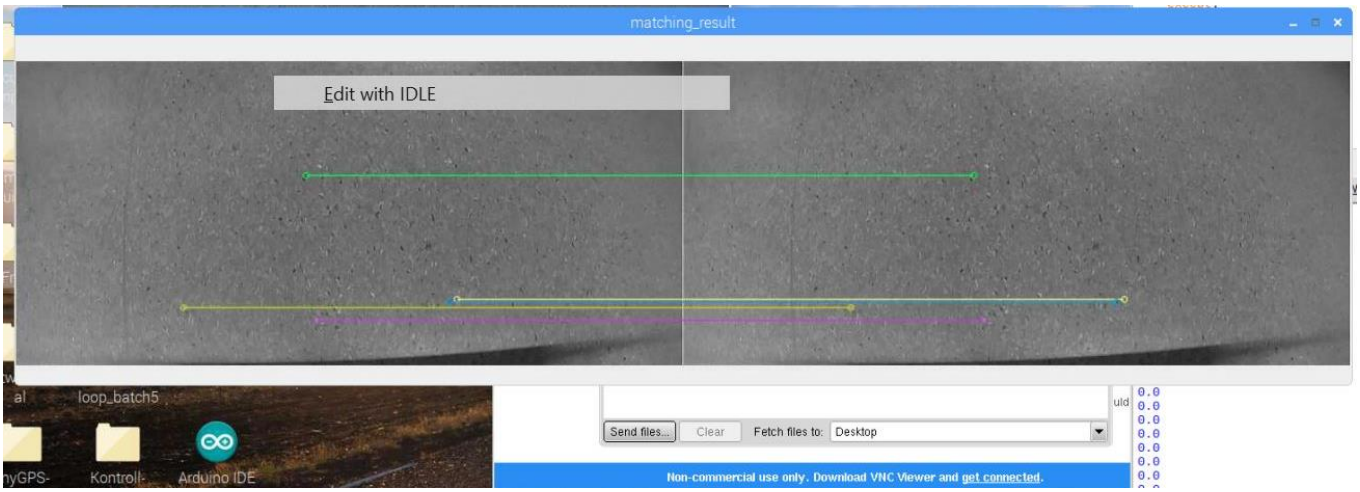
Figur 128: Vi merker oss attributen "pt" som inneholder pixelkoordinatene til nøkkelpunktet. Kilde:https://docs.opencv.org/master/d2/d29/classcv_1_1KeyPoint.html#ae6b87d798d3e181a472b08fa33883abe

Man har altså tilgang til pixelkoordinatene i *pt*, og har nå alt som trengs for å beregne pixelendring imellom de to nøkkelpunktene. Man må huske å beregne med algoritmen for korrekt posisjonsmåling som ble konstruert i 2.3.

```
pixel_pr_cm = 243.42 * (sensor_avstand_bunn)**(-0.942)
try:
    x1 = kp1[matches[0].queryIdx].pt[0]
    x2 = kp2[matches[0].trainIdx].pt[0]

    y1 = kp1[matches[0].queryIdx].pt[1]
    y2 = kp1[matches[0].trainIdx].pt[1]

    PosX += (x2-x1)/pixel_pr_cm
    PosY += (y2-y1)/pixel_pr_cm
except:
    pass
```

Figur 129: Uthenting av pixelkoordinater for kp1 og kp2, med påfølgende konvertering til cm før det legges til posisjonen

Man har nå en fullverdig posisjonsmåler som kan levere posisjonsdata for x og y. Her er et eksempel:



Figur 130: Utklipp av posisjonsmålingen. Man ser at de tre nøkkelpunktene har ingen endring, slik en ser i skallet nede til høyre der man har posisjon lik 0

Viktig moment for implementering

Styringssystemet er bestemt ut ifra x og y, men for posisjonsmåleren vil posisjonsendring fremover alltid være i retning x. Når man designer styringssystemet og loggingen er dette et moment å huske på, og gjøre nødvendige endringer avhengig av hvilket steg man befinner seg på.

3.4.3 Logging

I utgangspunktet var det ønskelig med to typer logging. Den første og viktigste er plotting av posisjonsdata i x,y og z. Den andre og litt mindre essensielle er å logge ulike sensordata som temperatur, tid i vannet, hastighet osv. Denne delen tar for seg begge løsningene, men grunnet tidsbruk vil man ikke bruke logg nummer to.

Logg for posisjonsdata

Målet er å få dataen plottet inn i en tre-dimensjonal graf som man kan se på i etterkant. Dette plottet er direkte knyttet opp til testen av styringssystemet og posisjonsmåleren i vannforhold, da den vil vise hvor bra dronen evner å styre i forhold til hva som er ønsket. For Python finnes det et bibliotek som heter Matplotlib. Vi vil ta i bruk dette for å plotte dataen korrekt. Siden loggingen er i bruk når man styrer dronen, passer det best at også funksjonene legges i samme system. Siden man skal ta i bruk tre-dimensjonalt plot trenger en også å hente inn en ekstra funksjon kalt Axes3D.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

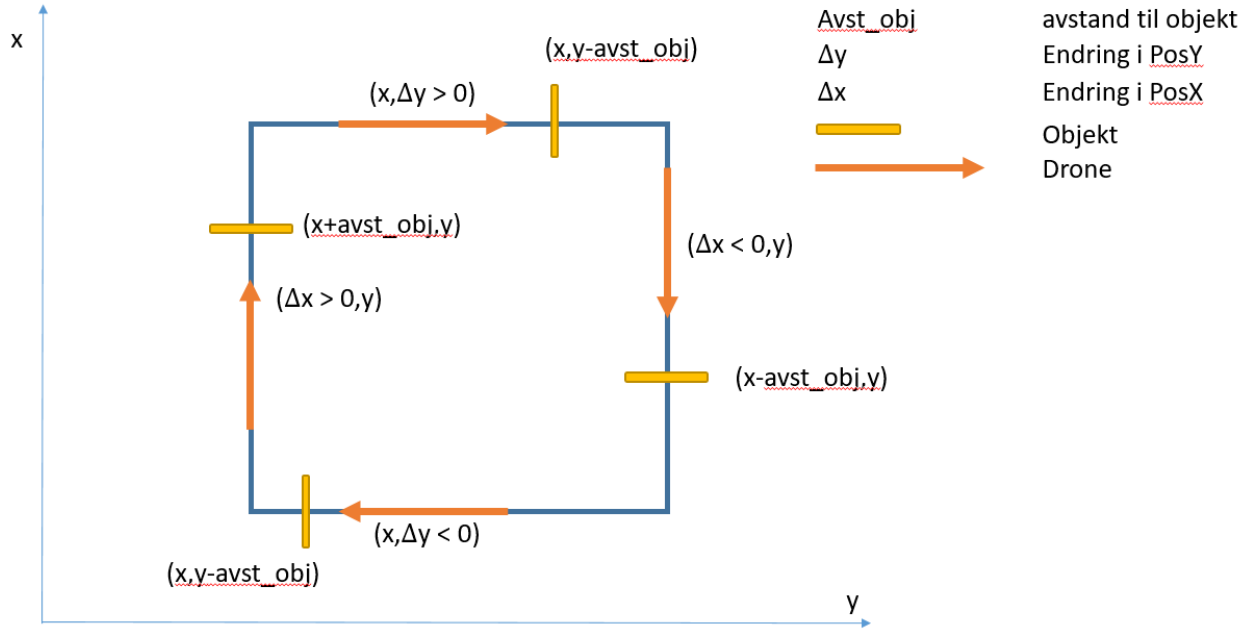
Figur 131: Importering av nødvendige biblioteker

Det er ønskelig å plotte både egen posisjon samt eventuelle objekter som er i veien for dronen. Metoden for å lagre posisjoner er at man lagrer tre egne tabeller x,y og z som fores med ny posisjonsdata kontinuerlig. Til slutt vil programmet konstruere et nytt plott som tar inn de tre tabellene og deres verdier. Siden man ønsker å ha med posisjonen til eventuelle objekter lages tre ekstra tabeller for å plotte dem i. Forskjellen ser slik ut:

```
x_track = np.append(x_track, PosX)
x = np.append(x, PosX + avst_obj)
```

Figur 132: Tillegg av ny posisjonsdata og ny posisjon på et objekt

Hvor x_track er tabellen for posisjonslogging, mens x er tabellen for eventuelle objekter. Merk at posisjonen til x vil avhenge av hvilket steg i styringssystemet dronen befinner seg på. Kjører dronen på steg én vil det se ut som i bildet med objektet foran seg i verdi, men kjører man for eksempel motsatt retning må en trekke ifra avst_obj, her bedre illustrert:



Figur 133: Illustrasjon av løype for dronen og hvordan objekter plottes ulikt av steg

Når dronen er tilbake til utgangsposisjonen lages et nytt 3D-plott med verdiene man har lagret i tabellene:

```
ax.plot(y_track,x_track,z_track,linewidth=3)
ax.scatter(y,x,z,'gray')
plt.savefig('Navigert_rute.png')
```

Figur 134: Laging av nytt 3D-plott

Merk her at man for posisjonsdata ønsker å ha en linje som baserer seg på all posisjonsdataen, mens for objektene kun ønsker rene punkter. Man må derfor bruke to ulike funksjoner for å lagre dataen korrekt i plottet, som til slutt lagres som .png-fil.

Logging for ulik sensordata

For at denne skal aktiveres må det bli spesifisert av brukeren via en kommando på Arduino. Denne kommandoen ble frigitt til annet bruk mot testing av fremdrift til dronen, slik at man skrinlegger bruken av slik loggdata helt fra Pi sin side.

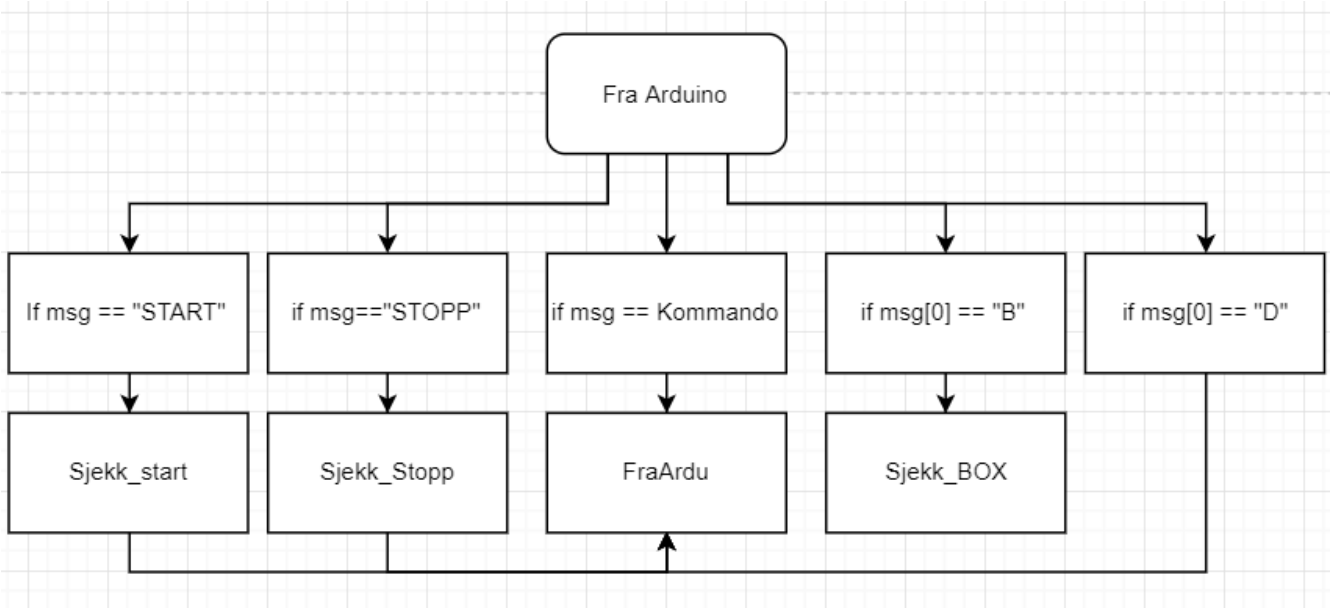
3.4.4 Kommunikasjon

For å bedre forstå styringssystemet er det mer oversiktlig å starte med å ta for seg kommunikasjonen. Variablene som styrer mottak og sending av data er for det meste dem som kan endres av mer enn én tråd, noe som krever at man er mer obs på hvordan en har oppsettet. Metoden for protokoll følger 3.1, der programkoden som brukes her er helt lik den for Jetson og vil ikke bli forklart på nytt. Den eneste forskjellen er at man mottar data fra to maskiner som gjør at en trenger to slike funksjoner hvorav én for Jetson og en annen for Arduino.

Det er designert egne variabler og funksjoner for sending og mottakelse av data fra Jetson og Arduino. Disse vil kun endres av én funksjon direkte, noe som gjør at andre funksjoner kan bruke variablene til å sjekke om deres data har blitt sendt eller eventuelt er klar for sending og om dataen de ønsker er mottatt. Dette gjør programmet mer fleksibelt og mer sporbart for feilsøking.

Mottak av data fra Arduino

Pi skal motta ulike kommandosignal fra Arduino som kommer fra HMI, og samtidig motta sensordata på avstand til bunn og dybde i vannet. For å holde det mer oversiktlig blir de ulike mulighetene delt opp i henhold til strukturen under:



Figur 135: Strukturen for de ulike funksjonene som behandler ulike oppgaver avhengig av hva som kommer fra Arduino

Det er altså fem ulike metoder på hvilken data som kan komme fra Arduino når man utelukker loggingen. Hvis meldingen er «START», «STOPP» eller begynner med «B» kommer meldingen fra Arduino via HMI. Årsaken til at det brukes «B» og «D» som første melding skiller seg ut fra protokoller og funksjoner man har hatt tidligere. Årsaken er at Arduino sender både informasjon om BOX-funksjonen og sensordata, slik at en må klare å skille dem fra hverandre da de ellers vil være like i form.

Man har en felles global boolsk variabel som heter *oppstart_arduino*, som bestemmer om programmet skal være i gang eller ikke. Både sjekk_start og sjekk_stopp har tilgang til variabelen og kan endre den avhengig av hva meldingen som kommer fra Arduino inneholder.

```
def sjekk_start():
    global oppstart_arduino
    global melding_fra_ARDU

    if melding_fra_ARDU == "START":
        print("Oppstart")

        if oppstart_arduino == False:
            try:
                tilPi = "<START>"
                PI2JET.write(tilPi.encode())
            except:
                PI2JET.reset_input_buffer()
                PI2JET.flush()
                pass
        oppstart_arduino = True
```

```
def sjekk_stopp():

    global oppstart_arduino
    global melding_fra_ARDU
    global Logfil
    global Stoppet


    if melding_fra_ARDU == "STOPP":
        Stoppet = True
        melding_fra_ARDU = ""
        oppstart_arduino = False
```

Figur 136: De to sjekk-funksjonene som styrer om programmet skal være i gang eller ikke

Merk at start-signalet til Jetson sendes via sjekk_start men at stopp-signalet ikke går via sjekk_stopp. Dette har med å gjøre at når Jetson er i gang vil den motta posisjonsdata fra Pi for å videresende, noe som gjør det vanskelig å få igjennom stopp-signalet kontra når den ikke mottar noe annen data som ved start-signalet. For det andre er det også kun ønskelig å sende start-signalet én gang, noe som skjer i skiftet fra usann til sann for *oppstart_arduino*. Det er ingenting annet som sendes i dette øyeblikket og skal ikke kreve noen form for bekreftelsesmelding.

Den tredje funksjonen sjekk_BOX skal ta imot ytterpunktene til BOX-funksjonen som tastes inn via HMI fra bruker. Dataen lagres så i de globale variablene *BOX_X* og *BOX_Y* til bruk i styringssystemet. Metoden for å separere meldingen som kommer er lik som for Jetson og blir ikke gjengitt her.

Videre har man sensordataen og bekreftelse på kommando. For kommando-bekreftelsen vil Pi sammenligne innkommende melding med meldingen den sendte ut. Hvis de er like vil den sette variabelen *melding_korrekt* til sann slik at resten av programmet vet at Arduino har mottatt rett data. For sensordataen vil man kreve at meldingen starter med «D» slik at man behandler sensordata og ikke BOX-begrensningene. Videre er en nødt til å dele opp meldingen ifra en tekststreng til de separate sensorvariablene man har. Metoden er lik som for Jetson der en separerer meldingen etter komma, for så å konvertere strengen til verdi.

Alle de fem ulike funksjonen samles i en og samme tråd kalt «FraArdu»:

```
while True:
    ArduKom()
    sjekk_stopp()
    if oppstart_arduino == True and len(melding_fra_ARDU) > 0:
        if melding_fra_ARDU == til_ARDU and til_ARDU != "999":
            melding_korrekt = True
        elif melding_fra_ARDU[0] == "A" and Logmode == True:
            Logfil.write(melding_fra_ARDU + "\n")
        elif melding_fra_ARDU[0] == "D":
            try:
                _,dybde_fra_ard, avstand_bunn_fra_ard = melding_fra_ARDU.split(",");
                sensor_dybde = float(dybde_fra_ard)
                sensor_avstand_bunn = float(avstand_bunn_fra_ard)
            except:
                pass  #Feil i uthenting av dybde, avstand_bunn
    else:
        sjekk_start()
        sjekk_logg()
        sjekk_BOX()
```

Figur 137: FraArdu-tråden som behandler all data fra Arduino

Sending av data til Arduino

Det er her det fort blir mer komplisert i forhold til for de andre maskinene. Man har stadfestet at alle kommandoer til Arduino skal bekreftes tilbake, og at en samtidig skal ha et fungerende

styringssystem som evner å reagere på hendelser. Dette innebærer at tråden som styrer utsendinger til Arduino skal enten sende bekreftelsesmeldingen «999» eller ny kommando. Videre må man og ha en måte å bekrefte eventuelt stopp-signal sendt fra Arduino, men denne er det eneste som skal sendes når *oppstart_arduino* er usann som gjør at man ikke trenger ta hensyn til det.

Det velges derfor å ha en prioritert rekkefølge på hva som skal bli sendt til Arduino, der bekreftelsesmeldingen troner øverst:

```
if oppstart_arduino == True:

    #Enten ønsker BOX_Kontroll å sende kommando:                          send_kommando = True
    #Ellers vil vi gi beskjed til Arduino at den har mottatt rett kommando:  melding_korrekt = True

    if send_kommando == True or melding_korrekt == True:


        if melding_korrekt == True:
            til_ARDU = "999"
            melding_korrekt = False

        else:
            til_ARDU = kommando
            #print("n", til_ARDU, "   ", PI2ARDU.in_waiting)
        sendt_message = "<" + til_ARDU + ">"


        try:
            if PI2ARDU.in_waiting == 0:
                PI2ARDU.write(sendt_message.encode())
```

Figur 138: Rekkefølgen på hva som skal sendes til Arduino

Løsningen gjør at tråden for styringssystemet er den eneste som endrer på variabelen *kommando.* Dette hindrer problemet med at flere tråder endrer på samme variabel, og gir også styringssystemet mulighet til å sjekke om kommandoen har blitt sendt ved å se på *til_Ardu*.

Mottak av data for Jetson

For Jetson deles det ikke opp i funksjoner da man ikke skal være klar for å motta stopp-signal fra den. Det eneste en skal motta er bekreftelsesmeldinger på at man er stoppet og sensordata, som gjør det mye enklere enn for Arduino. Man har kun én type sensordata som sendes, så man trenger ikke ha en bokstavkode i forkant. Utenom det vil overføringen til rett variabel gjøres på lik linje som tidligere. Når det kommer til å sørge for at Jetson har mottatt stopp-signal har man en variabel kalt *Stoppet_JETSON* som kun settes til usann når Pi mottar bekreftelse på stopp fra Jetson. Slik vil Pi fortsette å sende stopp-signal helt til Jetson mottar det og bekrefter tilbake.

```
while True:
    PI2JETKom()

    if oppstart_arduino == True:
        Stoppet_JETSON = False

        if PI2JET.is_open == False:
            PI2JET.open()


        if len(melding_fra_JET) > 0:
            try:
                avstand_obj = float(melding_fra_JET)
            except:
                pass
            ##print(avstand_obj)

    else:

        if Stoppet_JETSON == False:
            tilPi = "<STOPP>"
            ##print(len(tilPi), "  ", PI2JET.inWaiting())
            if PI2JET.inWaiting() < len(tilPi):
                PI2JET.write(tilPi.encode())
        if melding_fra_JET == "STOPPET":
            Stoppet_JETSON = True
        elif melding_fra_JET == "RESTART" and Stoppet_JETSON == True:
            Stoppet_JETSON = False
```

Figur 139: Funksjonen som tar seg av mottak av data fra Jetson og eventuell sending av stopp-signal helt til vi får bekreftelse

Sending av data til Jetson

Samtidig som Pi mottar avstand fra Jetson skal den også sende posisjonsdata tilbake. I funksjonen som tar seg av kommunikasjon mellom de to enhetene setter man en prioritert rekkefølge som er slik at Pi kun sender data *til* Jetson hvis meldingskøen *fra* Jetson er tom.

Sammendrag av kommunikasjon

Programmet er designet for å være fleksibelt og mest mulig robust mot kluss som følge av flere tråder som endrer samme variabel. Med å ha egne funksjoner som sender og henter serielldata, kan andre tråder sjekke opp i om man har sendt eller mottatt dataen de venter på. Det gjør og sendingen mer oversiktlig når Pi nå har en klar prioritering på hva som skal sendes.

3.4.5 Styringssystemet

Styringssystemet, eller BOX_kontroll(BOX) skal kombinere de andre trådene og oppgavene i programmet til å gi ut kommandoer og samtidig motta input for å kunne reagere på

eventuelle endringer. Her ser man en klar fordel med trådløsningen da vi kan ha ting mer oversiktlig enn tidligere.

BOX er som spesifisert i 2.8 delt opp i fire steg, herunder fire satte kurser/kommandoer. Fra bruker via HMI vil man få spesifisert en dybde man skal gå langs samt BOX-begrensningene i x og y. Brukerinputen hentes som beskrevet fra andre tråder/oppgaver, og er gjort tilgjengelig her.

Det er viktig at alle tre maskinene vet hva som skjer til enhver tid. Man er derfor avhengig av at BOX avventer videre styring og endring til den har forsikret seg om at både Jetson og Arduino henger med. Løsningen er å bruke en god del while-looper. Ved å benytte disse må man være bevisst på å legge inn avbrytningskriterier skulle en ønske å stoppe og komme seg ut av loopene. Vi definerer derfor fire variabler som skal styre kommunikasjonen med Jetson og Arduino via kommunikasjonstrådene:

```
global kommando_JET   #Endres her
global kommando        #Endres her
global send_kommando   #Endres her
global send_kommando_JET #endres her
```

Figur 140: De fire variablene som styrer kommunikasjon. De to øverste er tekststrenger og de to nederste er boolske variabler

De to øverste variablene i bildet er tekststrenger som lagrer kommandoen Pi ønsker å sende fra BOX. De to nederste er boolske variabler og er essensielle for å ha flyt i seriellkommunikasjonen. Vi ser for oss at vi skal sende en ny kommando til både Jetson og Arduino, og befinner oss i en while-loop som kun brytes av når begge maskinene har bekreftet kommandoen. Da kan det være at for eksempel Jetson svarer med en gang på bekreftelsen, mens Arduino bruker lengre tid. For Arduino vil man da prøve å sende kommandoen på nytt, men for Jetson behøver man det ikke. Det er også ikke naturlig å forvente at de to bekrefter synkront, slik at man må legge til rette for at vi forblir i loopen til begge har svart uavhengig av *når* de svarer.

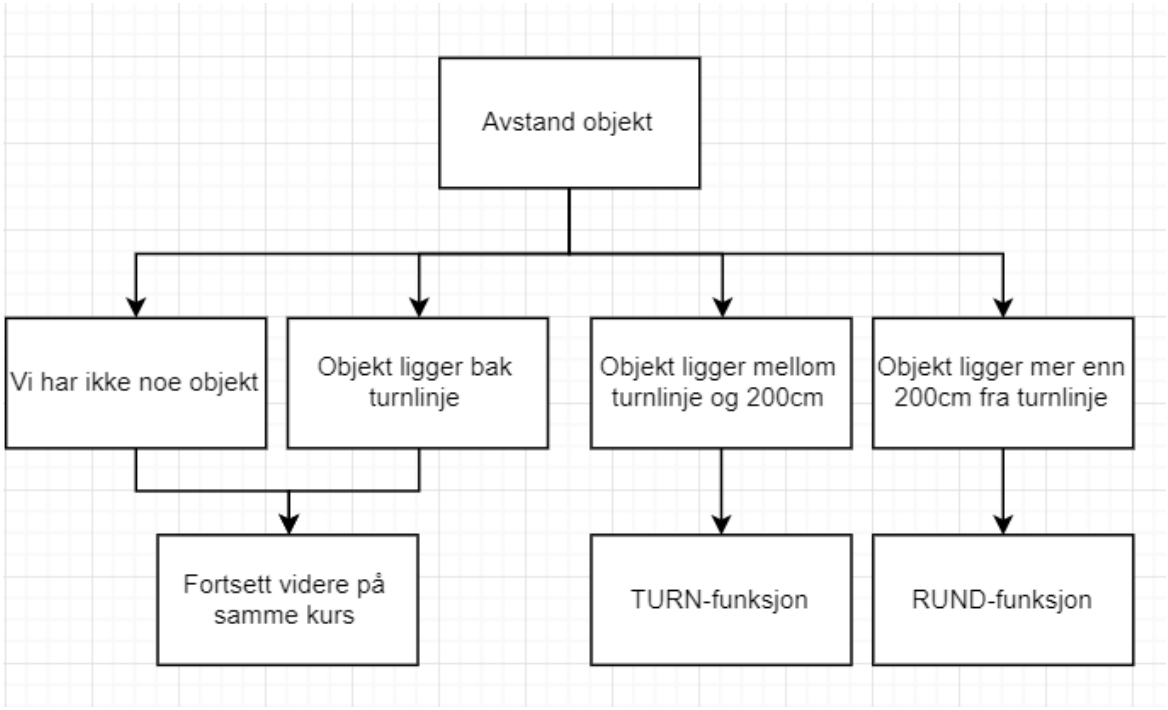
```
send_kommando = True
send_kommando_JET = True


while not(send_kommando == False and send_kommando_JET == False) and oppstart_arduino == True:

    if kommando == melding_fra_ARDU:
        send_kommando = False
    print(melding_fra_JET, "  ", melding_fra_ARDU)
    if kommando_JET == melding_fra_JET:
        send_kommando_JET = False
```

Figur 141: Utklipp fra koden for hvordan man bruker loopen til å vente på at begge bekrefter kommandoen

Man ser at de to boolske variablene opererer uavhengig av hverandre, og endres kun ved bekreftelse fra sin maskin. While-loopens avbrytes først når begge variablene er satt til usann igjen, slik at man kan få bekreftelsen på ulike tidspunkt og samtidig slutte å sende kommandoen på ny. Merk også at et fast kriterie for at loopen skal kjøres er at variabelen *oppstart_arduino* er sann, slik at man alltid kan avbryte programmet. Slike while-looper vil nyttes i BOX der en skal sende kommandoer, og har så og si samme form med kanskje noen små endringer.

For å spare på størrelsen til koden legges samtlige steg inn i samme hoved-loop. Man trenger derfor fire boolske variabler som avgjør om hvert steg er utført. Det er nyttig å ta en gjennomgang av hva som er ulikt for hvert av stegene. For det første vil kommandoene være ulik, herav kurs 0, 90, 180 og 270. For det andre vil posisjonsmåleren beregne x og y på samme måte uavhengig av hvilken kurs dronen holder. Det gjør at endringen mellom hver av stegene avgjør hvordan x og y plottes i loggen. For det tredje er turnpunktene for hver av stegene ulikt. Til slutt skal også dronen dykke opp og slå seg av når den nærmer seg slutten på siste steg.

Man har ulike røtter systemet kan bevege seg i avhengig av om det er et objekt foran seg, og om man skal snu til ny turnlinje.



Figur 142: Valgtreet til styringssystemet/BOX_kontroll

Fra Jetson har man pre-definert intet objekt som 360cm. Men det trenger ikke være at det mest hensiktsmessige tidspunktet å unngå objektet på er på slike avstander. Det er en mulighet for at objektet er forbipasserende og ikke vil befinne seg i veien senere, og da trenger det heller ikke å ageres på. Vi setter derfor distansen for reaksjon for dronen til 250cm, som skal være mer enn nok avstand til å unngå objektet. Når man har en lavere distanse enn 250cm vil valgtreet ha tre muligheter som nevnt i 2.8:

- Overse
- Turn-funksjon
- Rund-funksjon

Kriteriene som aktiverer de tre mulighetene baserer seg på avstanden til neste turnlinje og avstanden til objektet. Først har man oppgaven som overser objektet:

```
if avstand_obj < 250:

    if (avstand_obj - turn_avstand) >= 50:  #Objektet er minst 50cm etter turnpoint, overse
        pass
        print("ignorerer")
```

Figur 143: Muligheten som gjør at dronen overser objektet

Turn-funksjonen

Den neste er Turn-funksjonen. Her tar man i bruk mye av det som har blitt nevnt om kommunikasjon og uthenting av sensordata. Vi starter med å sende kommandoen om at Turn er aktivert til Arduino og Jetson:

```
elif int(turn_avstand-avstand_obj) in range(-50,200): #TURN

    kommando = "TURN"
    send_kommando = True
    kommando_JET = "TURN"
    send_kommando_JET = True
```

Figur 144: Kriterie for å aktivere turn samt sending av kommando'

Box tar så i bruk en lik while-loop fra tidligere for å verifisere at dataen er sendt korrekt. Fra Pi sin side er det neste steget å vente på at Arduino har gjort sin jobb med å styre unna objektet og fortsette til neste turnlinje. I mellomtiden vil programmet plotte oppdaterte posisjonsdata hvert andre sekund:

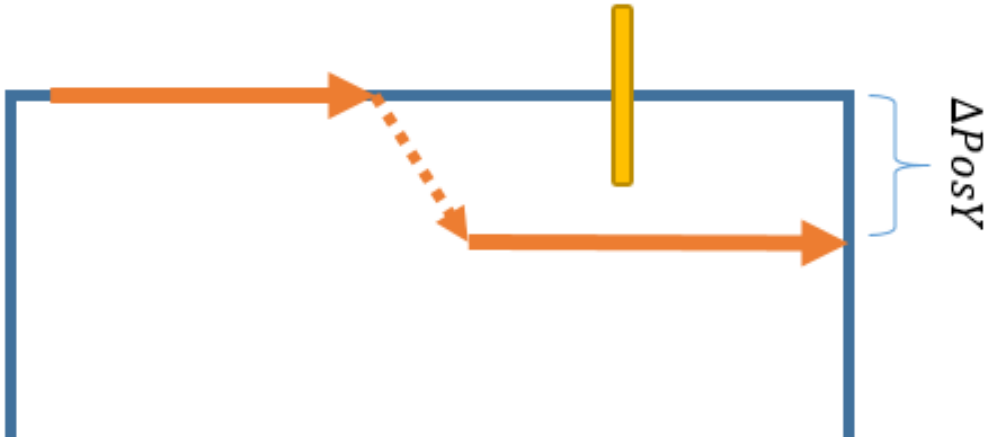
```
while melding_fra_ARDU != "DONE" and oppstart_arduino == True:
    if stegl == True and steg2 == False and steg3 == False and steg4 == False:
        x_track = np.append(x_track,PosX)
        y_track = np.append(y_track,PosY)
    elif stegl == True and steg2 == True and steg3 == False and steg4 == False:
        x_track = np.append(x_track,BOX_X - PosY)
        y_track = np.append(y_track,PosX)
    elif stegl == True and steg2 == True and steg3 == True and steg4 == False:
        x_track = np.append(x_track,BOX_X - PosX)
        y_track = np.append(y_track,BOX_Y - PosY)
    elif stegl == True and steg2 == True and steg3 == True and steg4 == True:
        x_track = np.append(x_track,PosY)
        y_track = np.append(y_track,BOX_Y - PosX)
    z_track = np.append(z_track,sensor_avstand_bunn)
    time.sleep(2) #Vi oppdaterer plottet bare hvert 2 sekund
```

Figur 145: Venting på at oppgaven med å turne er gjennomført av Arduino

Merk at man her har fire ulike metoder for å plotte dataen riktig på slik vi nevnte over, avhengig av hvilket steg dronen er på. Z-aksen vil derimot være lik for alle steg.

Det siste turn-funksjonen må gjøre er å nullstille posisjonene i x og y. Dette er basert på tidligere forklaringer om x alltid er i fremover retning mens styringssystemet er x og y-basert. Ved bytte av turnlinje er det også et bytte på hvilken av aksene som dronen beveger seg rundt, noe som innebærer skifte av x. Men turn-funksjonen har også beveget seg ut ifra

turnlinjen noe som betyr at man har en verdi for posisjonen i y.



Figur 146: Når dronen når ny turnlinje med Turn-funksjonen, vil startverdien til nye PosX ikke være 0

Resultatet er at dronen ikke treffer turnlinjen akkurat i turnpunktet, slik at nye PosX har en startverdi ulik 0! Verdien må derfor lagres og overføres når programmet nullstiller. I tråden brukes en boolsk variabel kalt *nullstill_pos* som så leses av tråden for posisjonsmåling. Det er da den som nullstiller og legger til eventuell verdi til nye PosX. Dette er mer fleksibelt da det kan være andre faktorer som gjør at PosX har en startverdi utenom turn-funksjonen, noe man helgarderer for. Fra tråden til posisjonsmåling har man:

```
if nullstill_pos == True: #Ny turn, vi nullstiller
    hjelp = PosY #Hvis PosY har verdi, må det med i nye PosX
    PosX = hjelp
    PosY = 0
    nullstill_pos = False  #Resetter variabelen
```

Figur 147: Nullstilling av posisjon og ny startverdi for nye PosX

Rund-funksjonen

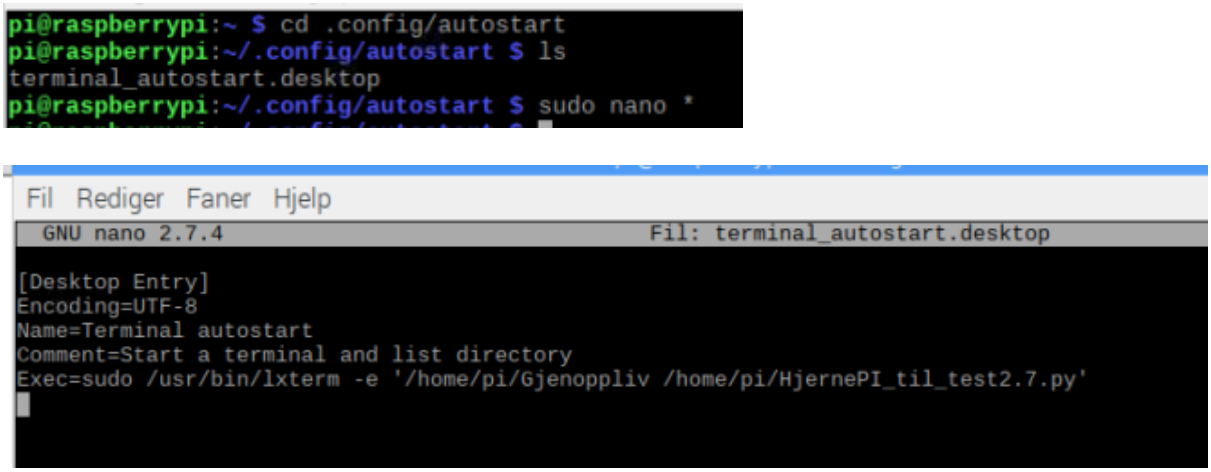
Til slutt har man Rund-funksjonen som skiller seg fra Turn med at den returnerer til samme turnlinje istedenfor å gå til neste. Oppsettet for Rund er dermed helt likt utenom at etter at Pi får melding fra Arduino på at dronen er rundt objektet, så skal den ikke nullstille posisjonen siden man enda befinner seg på samme linje.

3.4.6 programmet starter ved oppstart og restarter ved feil

I motsetning til for Jetson kjører Pi på Debian som operativsystem. Man har ikke en egen applikasjon som gir oss mulighet til å velge hvilke programmer som automatisk skal starte ved oppstart. På Raspberry Pi's eget diskusjonsforum finnes en løsning for å kjøre programmet ved oppstart. Først må man gjøre programmet «executable» eller kjørbart. Så

må man lage en textfil som inneholder kommandoer i tekstform. Det er da tekstfilen som leses ved oppstart slik at kommandoene blir utført.

Det er ønskelig at programmet skal restarte seg selv hvis det krasjer eller stopper uforventet på lik linje med Jetson. Man tar igjen i bruk Gjenoppliv-scriptet fra 3.3:



Figur 148: Kommandoer i terminal for at programmet skal starte opp ved oppstart

Programmet vil nå automatisk starte opp ved oppstart og restarte automatisk ved feil/krasj.

4.Tester og Resultater

Testingen har flere del-komponenter som alle må fungere godt og med god nøyaktighet for at hele systemet til dronen skal tilfredsstille kravene våre. Derfor ser vi det som viktig å få testet disse komponentene hver for seg, isolert fra hverandre først slik at man kan verifisere kapabiliteten til dem. Testene av de elektroniske komponentene hver for seg vil primært være en såkalt “tørrtest”, der testingen og målingen skjer på land. Vurderingen er at det å separere komponentene helt fra hverandre er enklest å gjøre i klasserom og på land. Samt at hvis vi skulle gjennomført testene under vann ville vi måtte demontert og montert opp dronen flere ganger, med nye innstillinger for hver del-komponent. Dette gjelder derimot ikke for de isolerte testene der komponentene ikke er elektroniske, som for eksempel tetthetstesten.

Oppsettet til dette kapitlet vil starte med samtlige del-tester der vi har en kort introduksjon av komponenten og hva vi forventer av resultater. Til slutt vil vi ha en total systemtest som innebærer en måling av hvor bra dronen klarer å tilfredsstille hovedmålet og besvare problemstillingen.

4.1 Testing av Avstandsmåler

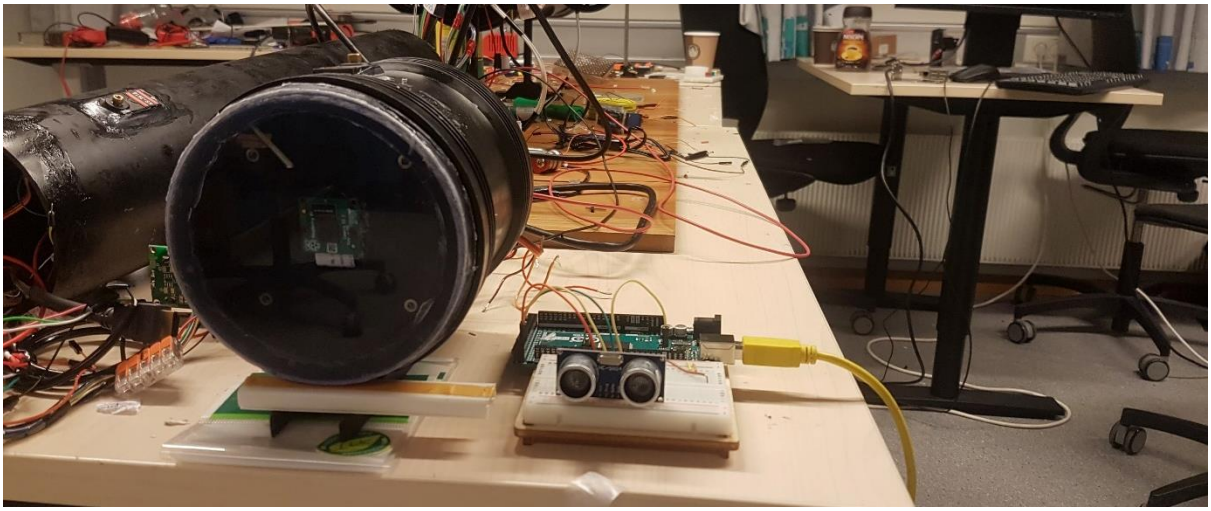
Avstandsmåleren er en del-komponent som ved bruk av kamera og lasere skal måle avstanden til objekter som befinner seg foran dronen. For at testen vår skal ha validitet må vi ha en metode for å måle faktisk nøyaktig avstand til et objekt, slik at vi kan sammenligne hvor bra og nøyaktig måleren vår er. I masteroppgaven til Henriksen[2] brukes det en avansert type posisjoneringssystem koblet til dronen for å gjengi sann avstand. Dette er utstyr vi ikke har tilgang til, og må velge noe som gir nøyaktige nok resultater og er tilgjengelig til bruk. Hvis dronen skulle blitt konstruert til å fungere over land ville antageligvis en billig avstandsmåler blitt brukt, som for eksempel en HC-SRO4(figur 153).

Figur 149: HC-SR04, Kilde: https://www.makerlab-electronics.com/product/ultrasonic-sensor-hc-sr04/

Denne er både billig, tilgjengelig og har god nok nøyaktighet for å tilsvare kravene våre. Med dette mener vi at hvis avstandsmåleren vår oppnår samme resultater vil den tilfredsstille de gitte kravene. Bruk av HC-SR04 krever liten tid å sette opp da alt av bibliotek og oppkobling finnes på nettet.

4.1.1 Test av nøyaktighet

Vi setter HC-SR04 på linje med kameraet som brukes til avstandsmåling, som vist i figur 151.



Figur 150: Oppsett for måling med komponenten for avstandsmåling til venstre og HC-SR04 til høyre

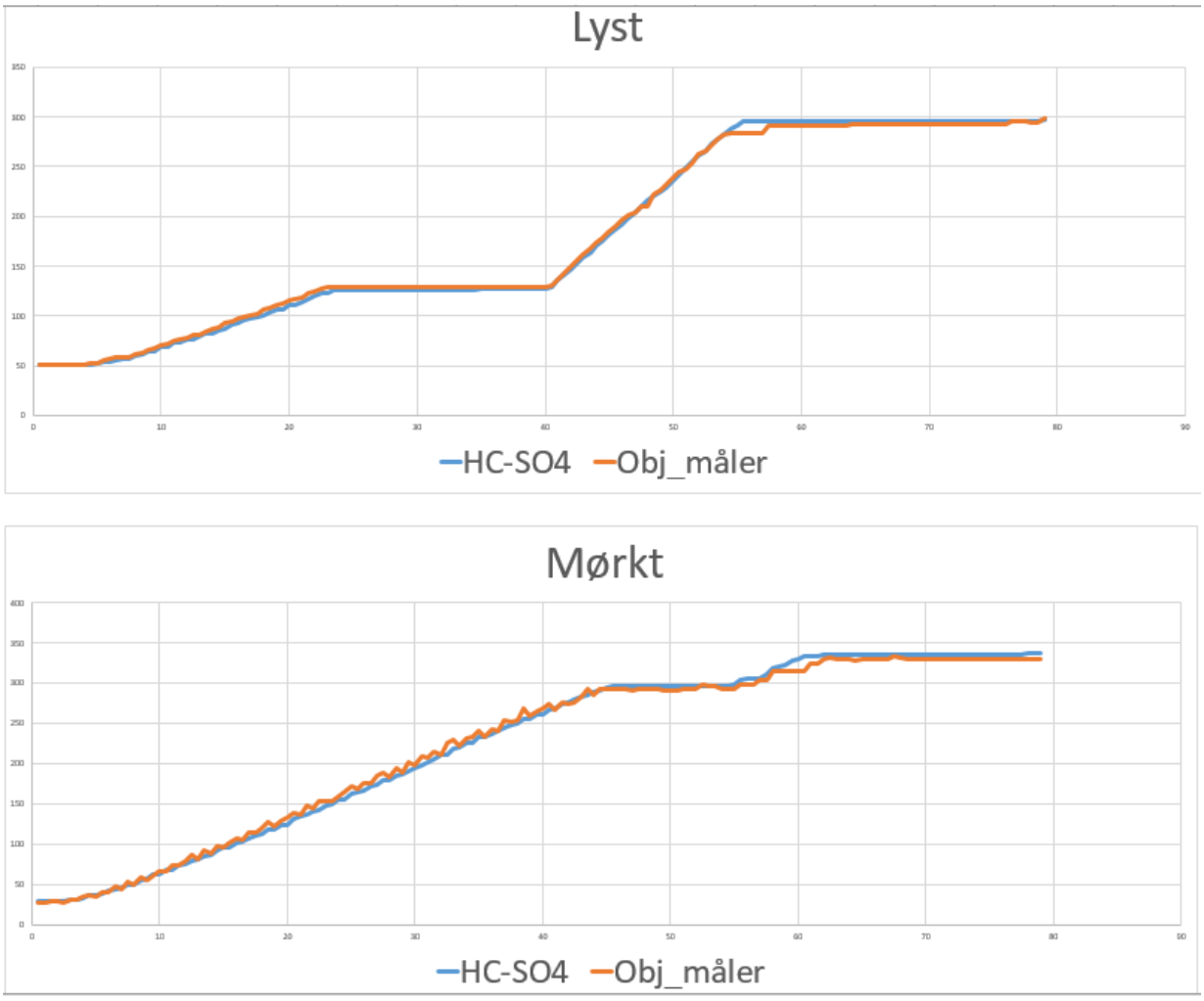
Det som kan ses på bildet er at komponenten ikke kan beveges frem og tilbake da den er festet til ulike ledninger som sitter fast i dronen. For å få data på ulik avstand til et objekt, ble det objektet som skulle bevege seg frem og tilbake. Det ble tatt i bruk et objekt med en stor flat side som vist i figur 151 som er mulig å flytte på. Flaten var da stor nok til at begge sensorene fikk utslag på samme flate. Det er viktig å påpeke at dataen for HC-SR04 og avstandsmåling kommer fra to ulike systemer. Løsningen ble å innføre en lik treghet i oppdateringen mellom de to systemene, slik at de var samkjørte. Dette var fordi de to systemene ikke hadde samme oppdateringsrate i utgangspunktet.

Figur 151: Måling av posisjon i to ulike lysforhold lyst og mørkt

Som beskrevet i 3.2, vil maskinens evne til å detektere objektene variere med lysstyrken i bildet. Vi vil dermed gjennomføre testene i både lyst og mørkt klasserom (figur 152, høyre bilde).

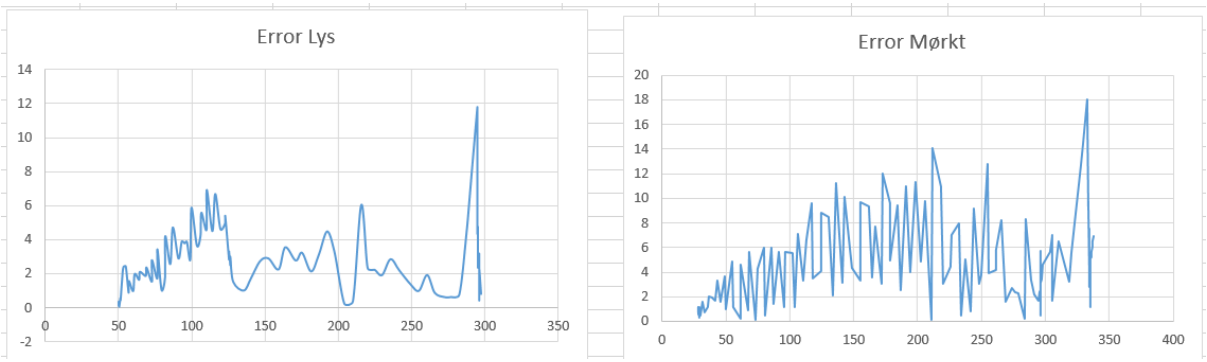
Vi må adressere feilkildene vi har i testen. For det første vil plasseringen av både kamera og HC-SR04 kunne være ulik i forhold til objektet som skal måles til og ikke helt korrekt innstilt. Videre burde begge sensorene stå vinkelrett på objektet, men dette kan være feilaktig både ved plasseringen av dem og når vi beveger objektet frem og tilbake. Til slutt har vi eventuell unøyaktighet i oppdateringsraten til de to systemene, som i teorien skal være gjort lik.

Objektet ble så flyttet frem og tilbake både i lyse og mørke forhold og dataen plottet i Excel (figur 153):



Figur 152: Test av avstand til objekt både med vår avstandsmåler og HC-SR04

Merk at objektet har ulike start/sluttpunkt og dermed ulik bevegelse for de to testene gjort i lys og i mørke. Vi ser at vår avstandsmåler evner å holde følge og beregne tilnærmet lik data på avstanden til objektet for samtlige avstander. Men, ser vi nærmere på forskjellene mellom HC-SR04 og avstandsmåleren i forhold til faktisk avstand (figur 153), ser vi at beregning i mørke er mer ujevn og har større feil enn i lyse forhold.



Figur 153: differanse i målinger mellom HC-SR04 og avstandsmåler

4.2 Testing av posisjonsmåling

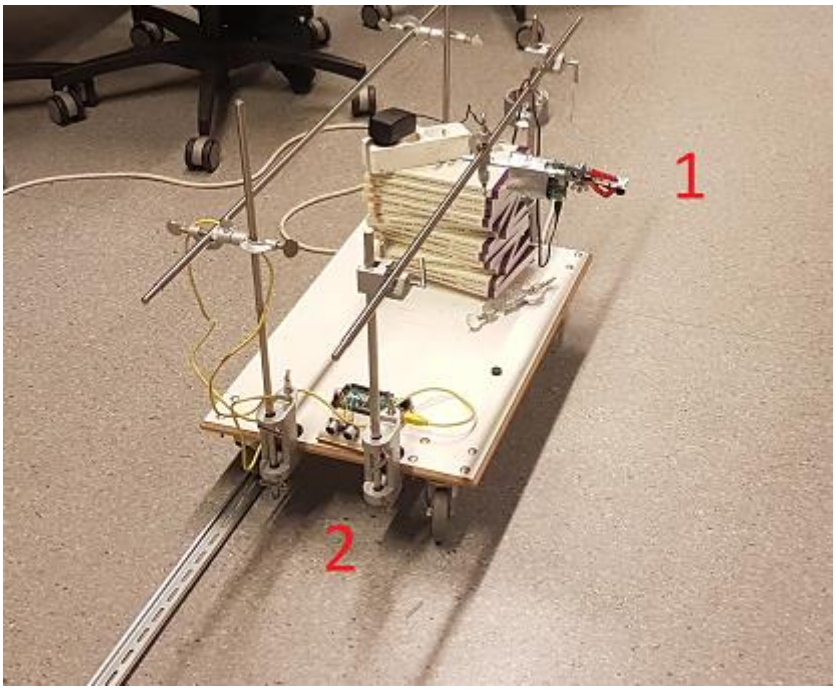
Posisjonsmåling er en del-komponent som ved bruk av kamera og sonar skal måle posisjonen til dronen i forhold til et startpunkt. På lik linje med forrige test må vi ha en måte å samle faktisk posisjonsdata for å sammenligne med dronen. Denne testen skal gjøres i klasserom som utelukker bruk av GPS, men vi kan fortsatt ta i bruk HC-SR04 med samme argumentasjon som i forrige test med at den ville kunne bli brukt skulle vi bygget dronen over vann. Eneste forskjellen fra tidligere er at vi må legge til noe egenprodusert kode i den ferdige pakken vi brukte tidligere, der programmet må sette et nullpunkt/referansepunkt. Slik vil HC-SR04 også starte i null, og notere ned posisjonsendringen med å logge avstanden den har til et objekt. Det ble videre også målt opp faktisk distanse dronen skulle gå ved hjelp av en standard laser avstandsmåler.

Figur 154: Laser avstandsmåler brukt til å måle faktisk avstand

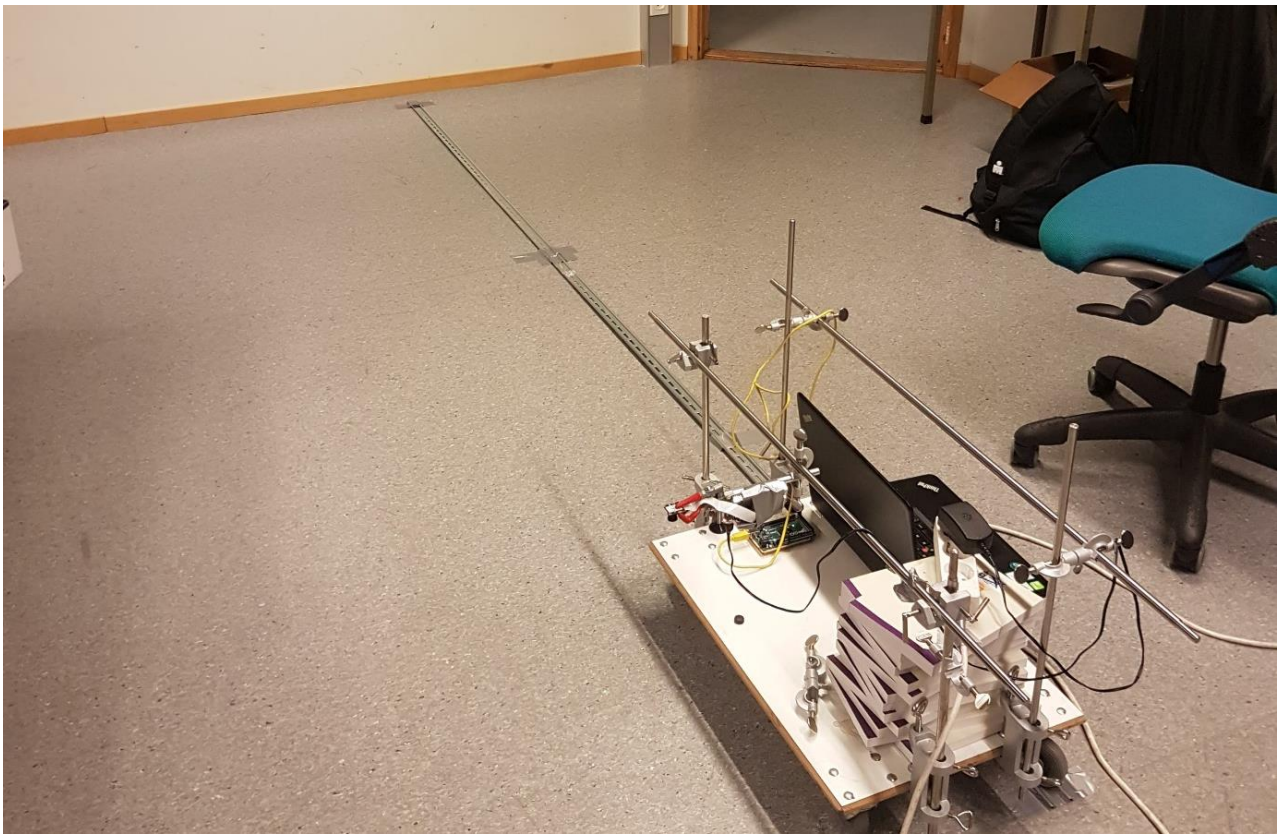
For testen måtte en fast avstand til bunn benyttes for testingen, da sonaren som blir brukt kun fungerer skikkelig i vann. For testen festes kameraet ca 40cm ifra gulvet.

4.2.1 Test av nøyaktighet

Vi ønsker å bevege en form for kjøretøy frem og tilbake langs en fast akse for å måle nøyaktigheten til posisjonsmåleren. Det er flere feilkilder til stede for denne testen som må adresseres. Før det første har vi kameraets avstand til gulvet, som er målt med en avstandsmåler og vil bringe med seg noe unøyaktighet i forhold til algoritmen vi bruker for posisjonsmåling. For det andre har vi kameraets vinkel i forhold til gulvet som ideelt sett skal stå vinkelrett på. Kameraet festes i et stativ som vi ikke kan garantere at har 90 grader i forhold til gulvet. Videre på dette har vi også HC-SR04 sin vinkel i forhold til objektet den ser mot, som heller ikke kan garanteres å være vinkelrett. For det tredje vil kjøretøyet gå på en skinne for å sørge for at kameraet kun beveger seg i én akse (Figur 156 og Figur 157). Skinnen vi har brukt er litt bredere enn hjulet til kjøretøyet og kan gjøre at kjøringen ikke blir jevn. Til slutt vil vi også her kunne ha utfordringen med at dataen hentes fra to ulike systemer og kan ha ujevn oppdateringsrate.



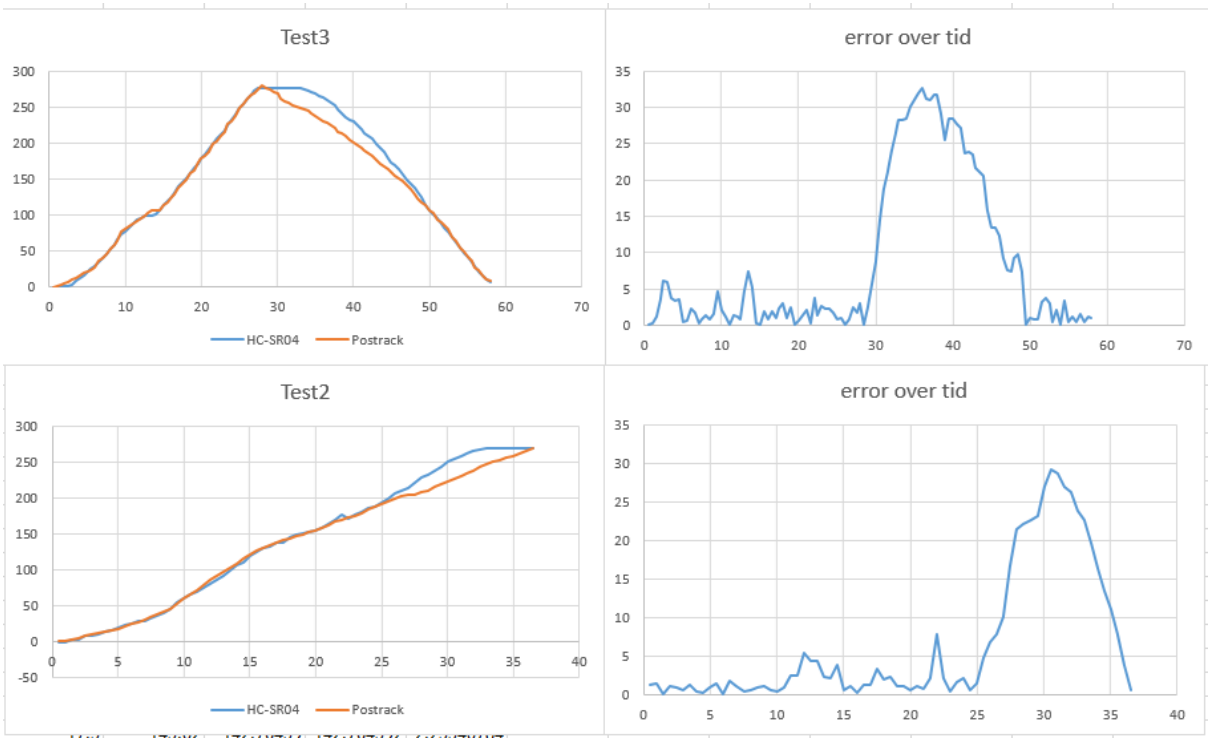
Figur 155: Montering av kjøretøyet som har posisjonsmåler(1) og HC-SR04(2) montert på



Figur 156: Kjøretøyet med skinnen den skal følge for å ha mest mulig nøyaktighet i målingen, samt veggen som HC-SR04 bruker til referanse

Vi har gjort flere målinger for denne testen i den hensikt å prøve å oppdage om posisjonsmåleren vil gi stabil data og om eventuelle hendelser som skjer i løpet av kjøringen gir ekstra utslag. Det første målingen som ble gjennomført hadde HC-SR04 tre ganger så mange målinger per sekund som posisjonsmåleren. Dette gjorde igjen at målingen måtte forkastes da målingene ikke stemte overens med hverandre, understreket derfor viktigheten av å være bevisst på feilkildene vi kan ha i målingene.

Vi har her lagt ved de neste to målingene for å vise til et interessant funn, der første måling gjennomføres kun den ene retningen imens måling nummer to kjører frem og tilbake på skinnen (Figur 157):

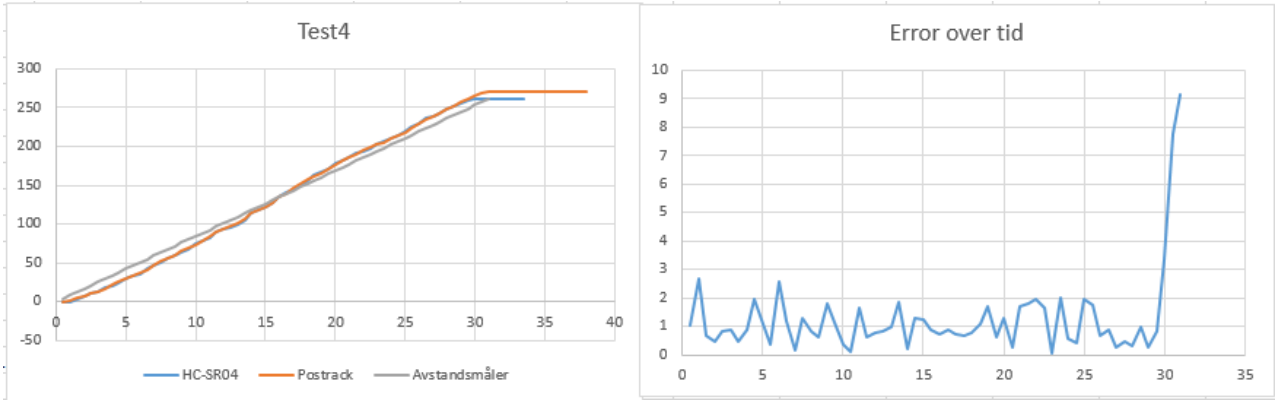


Figur 157: Plott av målinger tatt under kjøretøyets bevegelse. Test 3 er frem og tilbake mens test 2 kun er frem

Vi ser at de to systemene har en kraftig endring i differanse mellom hverandre rundt enden av skinnen, altså når vi har kjørt frem med kjøretøyet. Ser vi på test 3 er det tydelig at differansen mellom de to sensorene blir liten når kjøretøyet igjen får bevegelse. Det indikerer at differansen oppstår når kjøretøyet går fra å være i bevegelse til å stoppe opp, og at minst én av sensorene får feil data som en følge av dette.

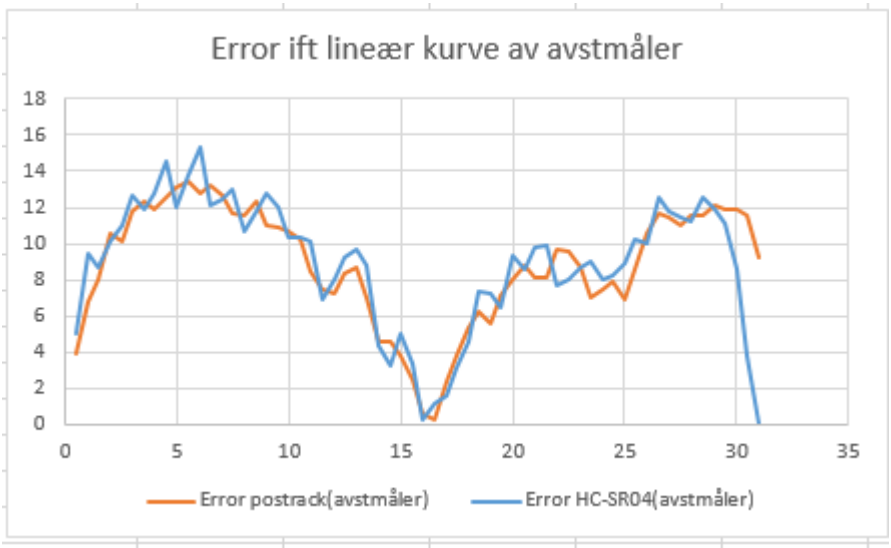
Vi vil prøve å finne ut av problemet gjennom å eliminere muligheter gjennom flere tester. Vår første antagelse er at feilen ligger i posisjonsmåleren, da bråstopp og vibrasjoner som en følge ikke skal forårsake noen problemer for HC-SR04. Vi har nevnt i 4.1 at det også finnes en usikkerhet knyttet til HC-SR04, og vi vil dermed legge til en tredje og mer nøyaktig måler. Vi tar i bruk lasermåleren og måler avstand til veggen fra kameraets posisjon og måler igjen når vi når enden av skinnen. Slik vet vi den faktiske avstanden kjøretøyet har beveget seg enda mer nøyaktig.

I neste måling ble det laget en lineær kurve basert på avstanden kjøretøyet har beveget seg og tiden den brukte målt av laser avstandsmåler og stoppeklokke. Hvis kjøretøyet så evner å holde en jevn fart skal både HC-SR04 og posisjonsmåleren ideelt sett ha en lineær linje lik den vi konstruerte. Hvis HC-SR04 følger linjen indikerer det at den er nøyaktig, og styrker antagelsen på at det er posisjonsmåleren som har problemer når vi stopper opp. Samtidig legges det på en ekstra forsinkelse på HC-SR04 for at begge systemene skal ha likt antall målinger per sekund. Resultatene vises i figuren under.



Figur 158: Test 4 der vi har forsøkt å holde jevn fart frem med kjøretøyet, med påfølgende differanse mellom de to sensorene

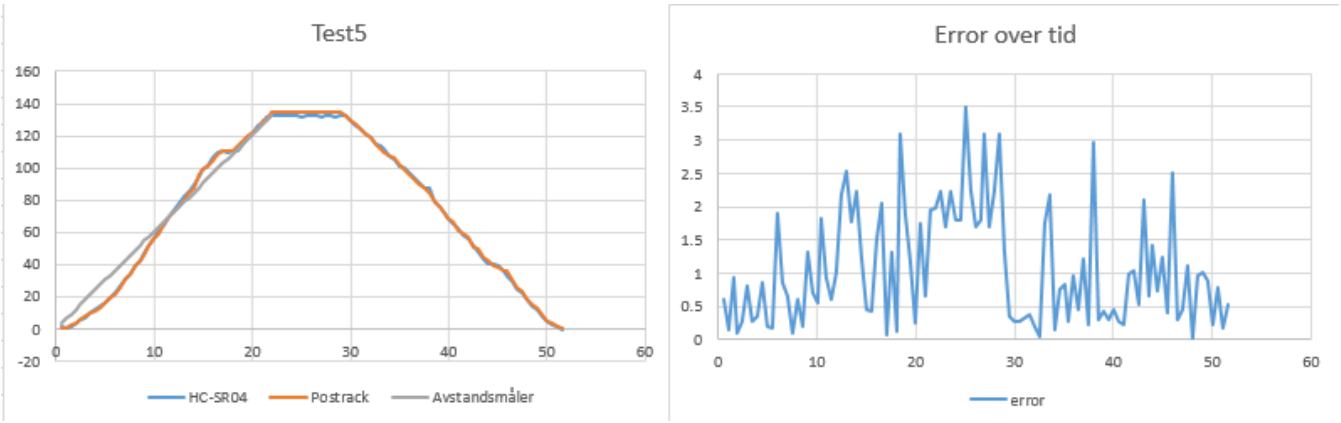
Ut fra resultatene følger de to sensorene hverandre med feil opptil 3cm helt til kjøretøyet stopper, da vi igjen får en stor ulikhet. På bildet til høyre virker det som at vi har klart å holde en ganske konstant fart i distansen vi har dratt kjøretøyet, slik vi ønsket. Dette med bakgrunn i at begge sensorene holder godt følge med den lineære kurven vi beregnet ut ifra avstandsmålerens data. Tilbake til antagelsen om at det er posisjonsmåleren som skaper en feil når kjøretøyet går fra bevegelse til ro. Ut ifra målingene ser man på bildet til venstre at HC-SR04 stopper på tilnærmet samme posisjon som avstandsmåleren, imens posisjonsmåleren er noe høyere. Ved å se på målingene følger de to sensorene hverandre helt til 260cm, der posisjonsmåleren fortsetter videre til 270cm mens HC-SR04 flater ut til 261cm som tilsvarer avstanden målt med laser avstandsmåler, som vist i figuren over.



Figur 159: Absoluttverdi av Differanse mellom den lineære linjen fra avstandsmåleren og de to sensorene hver for seg

Det er interessant å se på ulikheten mellom de to sensorene i forhold til den lineære linjen sett tidligere i Figur 159. Merk her at plottingen av grafen i Figur 160 er absoluttverdi av differansen. De to «bølgene» stammer mest sannsynlig fra at vi ikke har holdt helt konstant fart hele veien, og har en litt ulik bevegelse. Det er tydelig fra dataen at de to sensorene følger hverandre veldig bra så lenge vi er i bevegelse, men at posisjonsmåleren har nesten 10cm feil når vi stopper opp. Vi bedømmer etter å ha vurdert målingene at posisjonsmåleren ser ut til å få en større unøyaktighet når den går fra å være i bevegelse til å stå i ro. Dette kan fort være en feilkilde, som gjør at vi ønsker å sjekke om det er noe som kan forårsake uregelmessighetene.

Det første vi gjorde var å ta en nærmere titt på festingen til kameraet, da det antas å være en stor nok feilkilde til å kunne kludre med målingene våre. Referert til avsnittet om feilkilder, skal kameraet stå vinkelrett i forhold til gulvet for at målingen skal være nøyaktig. Derfor er det tenkelig at en brå endring i bevegelse som når vi stopper kjøretøyet, kan få kameraet til å oppfatte underlaget på en annen måte hvis kameraet ikke står vinkelrett på. Ved nærmere undersøkelser var kameraet litt feilposisjonert i forhold til gulvet. Vi snakker om en forskjell på maks 5 grader, men vi vil kjøre en ny måling for å sjekke om vi får noe feil utslag. For målingens effekt dras kjøretøyet frem og tilbake, slik at vi kan måle om den fortsatt holder følge der den har gjort det tidligere til tross for ny posisjon på kamera.

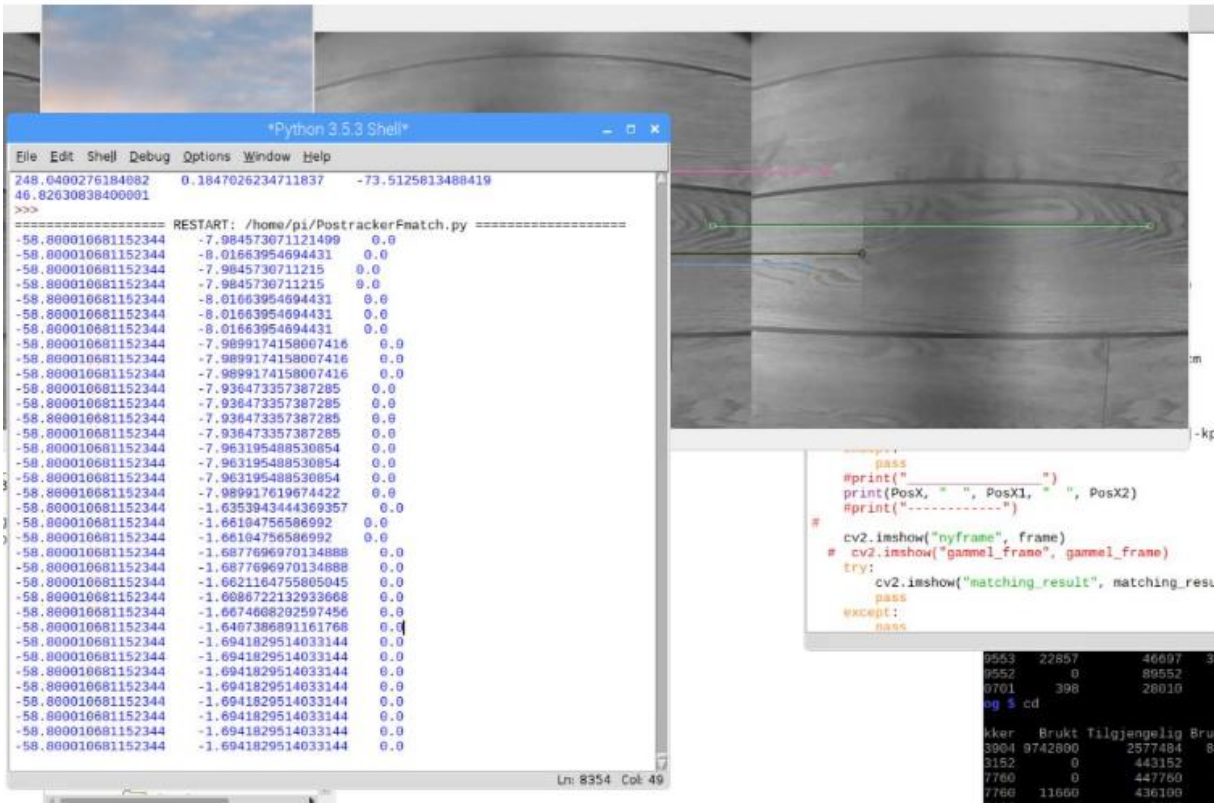


Figur 160: Nye målinger etter at vi har endret vinkelen på kamera

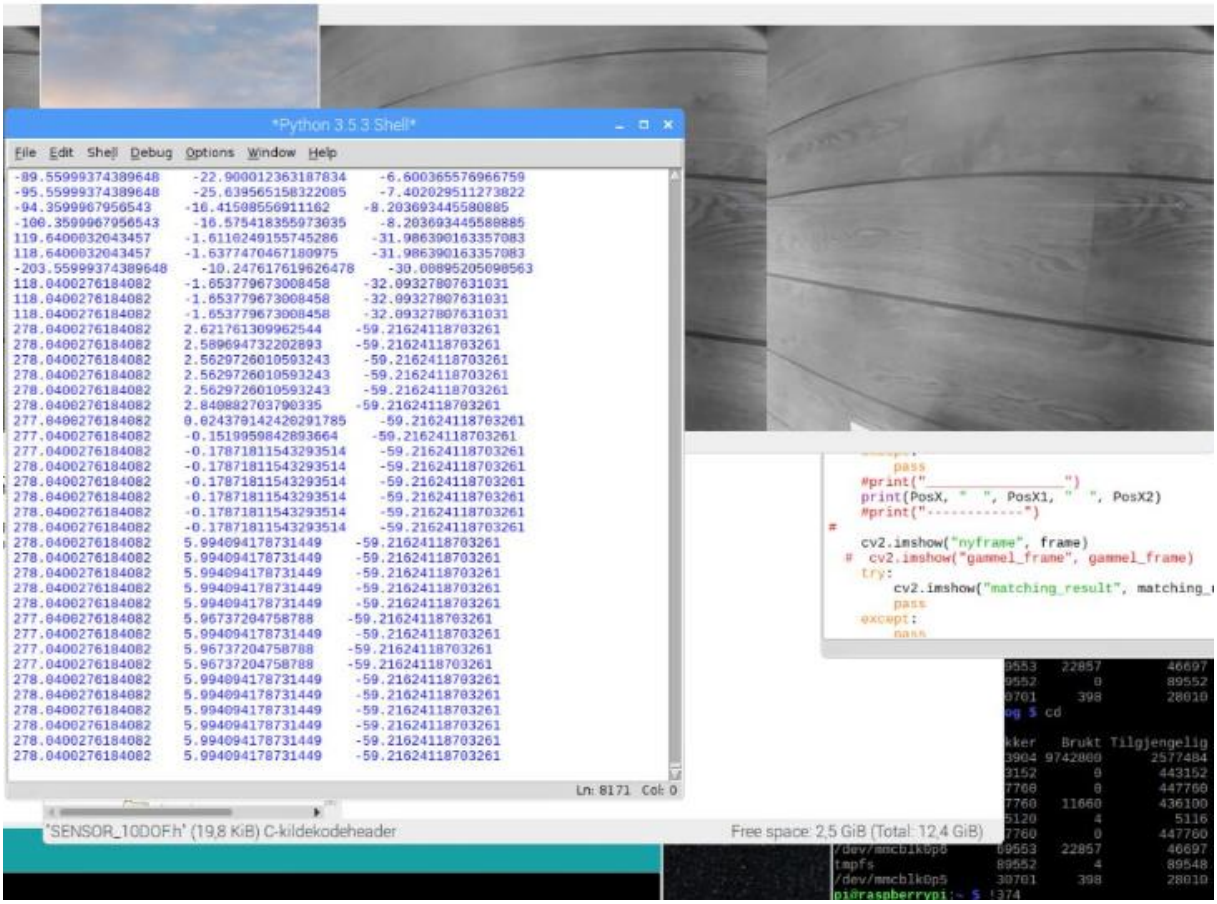
Målingene ble denne gang betydelig bedre enn i figur 159, og vi får ikke den store differansen som vi har hatt tidligere. Antagelsen vår er at kameraet har stått skjevt slik at det har fått med seg deler av veggen i det øyeblikket det nådde enden av skinnen, og dermed har feilberegnet pixelendringen.

For å se nærmere på hva vinkelen mellom kamera og bunnen kan ha å si for nøyaktigheten, ble det gjort to tester der dronen sto i ro. Den ene hadde kamera og bunn vinkelrett på

hverandre, mens den andre bevisst hadde en vinkelforskjell. Hensikten var å se om vinkelen spilte mye inn på nøyaktigheten slik vi hadde observert tidligere (Figur 156 og Figur 157):



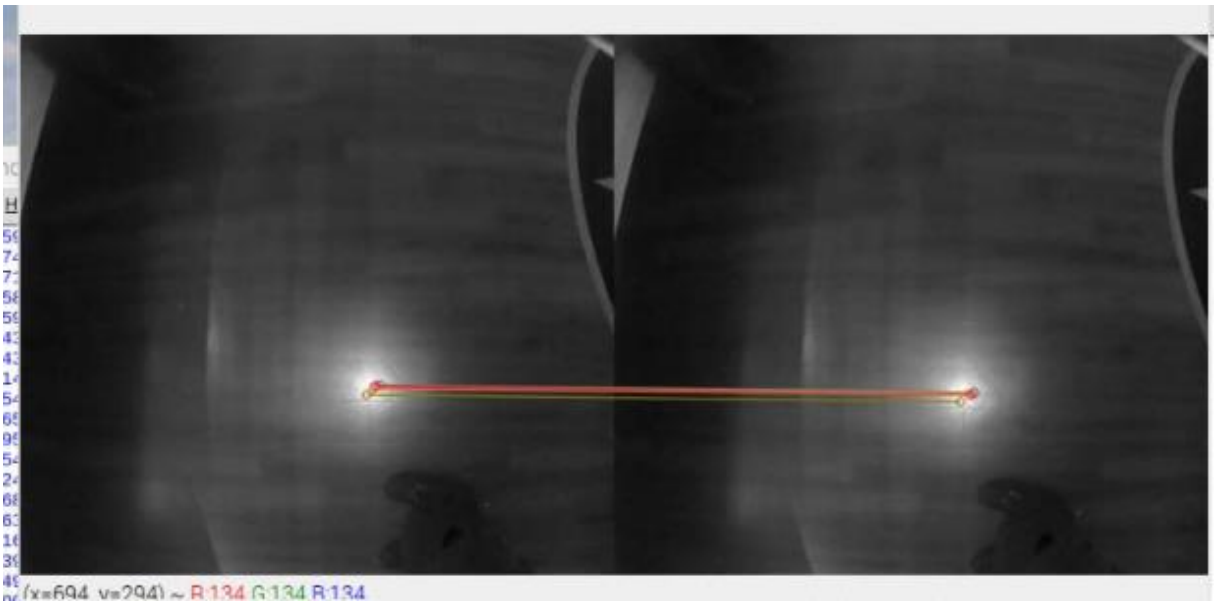
Figur 161: Kamera er vinkelrett på bunnen/gulvet. Legg merke til den ytterste av de tre verdiene som leses ut holder seg i 0 hele tiden



Figur 162: Kamera er ikke vinkelrett i forhold til bunnen/gulvet. Den ytterste verdien er ikke lenger 0 slik den burde

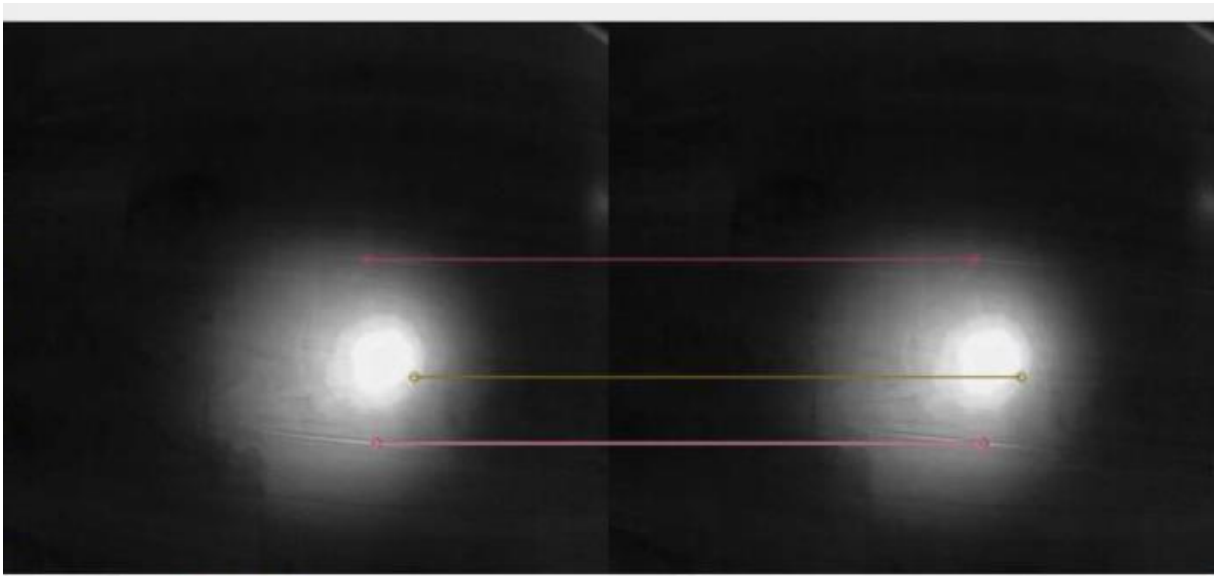
4.2.2 Test i mørke forhold og ved større avstand

Det er ønskelig posisjonsmåleren i dårligere forhold. Med dårligere forhold menes det at man både skal sjekke for forhold med mindre lys og for større avstand til bunn. Her forventes det en større unøyaktighet da man vil ha dårligere sikt til bunnen. Den første testen går ut på å sjekke om lyset som er montert fra 2.3 vil være godt nok til å belyse bunnen for posisjonsmåleren. Testen ble gjennomført i aulaen på Sjøkrigsskolen der det er ganske lite belysning, med avstander til gulvet alt fra 40cm til 2 meter (Figur 164):



Figur 163: Utsnitt fra posisjonsmåleren når dronen er 2 meter fra bunn. Lyset i bildet kommer fra dronen

Ut ifra bildet og observasjonene vi gjorde er det tydelig at lyskilden til dronen ikke klarer å lyse opp gulvet skikkelig, men heller forblir en lysflekk i bildet. Det gjør at posisjonsmåleren fokuserer på lysflekken som et nøkkelpunkt, som igjen aldri beveger seg relativt i forhold til dronen. Det gjør at posisjonsmåleren beregner endringen til å være null! Det er også viktig å bemerke den dårlige bildekvaliteten som gjør det vanskeligere å beregne andre nøkkelpunkt. Dette var for avstand på to meter, for kortere avstand ble problemet enda tydeligere, som vist i Figur 165:



Figur 164: Utsnitt fra posisjonsmåleren når dronen er 40cm fra gulvet. Lysflekken er her enda tydeligere

Ut ifra bildene kan det og tyde på at oppløsningen til kameraet også gjør det vanskelig å få god oppløsningen til gulvet. Videre ser man at to av de tre nøkkelpunktene på bildet er på sprekker i gulvet, men at den siste følger lysflekken fra kilden til dronen.

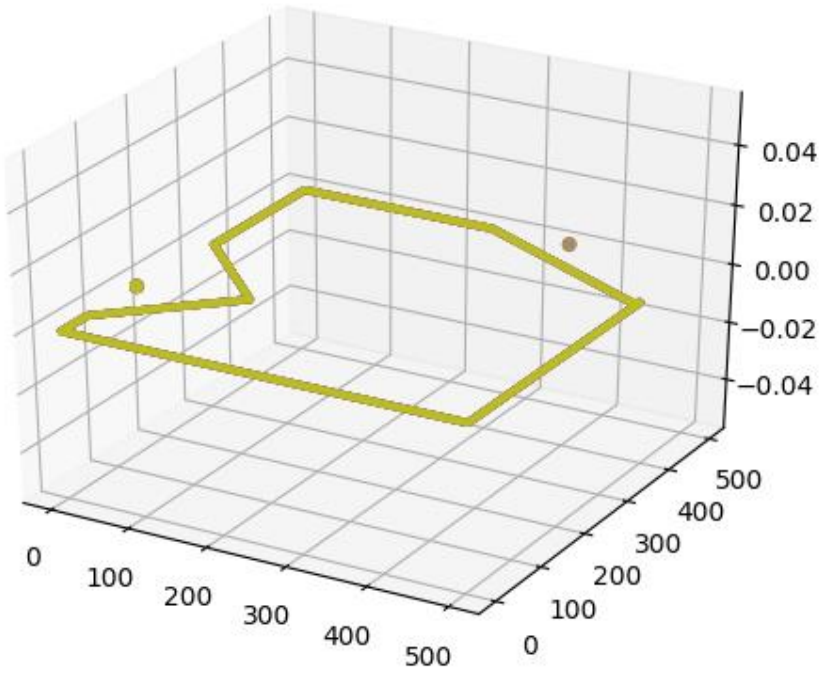
4.3 Test av styringssystem

For å teste styringssystemet før det går i vannet, slik at vi kan luke ut eventuelle feil og mangler før vi er i bassenget. Vi gjennomfører to ulike tester. Den første er en ren simulering innad i programmet der posisjonen plottes og objekter blir simulert inn. Den andre vil bruke avstandsmåleren til å gi avstand til objektet til styringssystemet, som så igjen sender ut kommando til Arduino. I test nummer to vil altså motor og posisjonsmåler være koblet av, og dronen dras manuelt bortover.

4.3.1 Simulering av styringssystem

Vi simulerer ved å koble ut posisjonsmåleren og avstandsmåleren. Så simuleres posisjon samt det legges inn en avstand til objektet når det passerer. Hensikten med testen er å sjekke selve BOX-forflytningen, Rund-funksjonen og Turn-funksjonen som er beskrevet i 2.8. BOX-forflytningen får begrensning 500 i x-aksen og 500 i y-aksen, slik at vi forventer disse målene i plottet i etterkant. Vi velger å legge ved to objekter som legges på henholdsvis steg 1 og 2 i BOX-funksjonen, ref 2.8.

På steg 1 legger vi et objekt som ligger langt nok unna turnpunkt til at vi kan gjennomføre Rund-funksjonen. På steg 2 legges et objekt som derimot er for nært turnpunktet slik at vi får testet Turn-funksjonen. Resultatet er plottet i Figur 166:

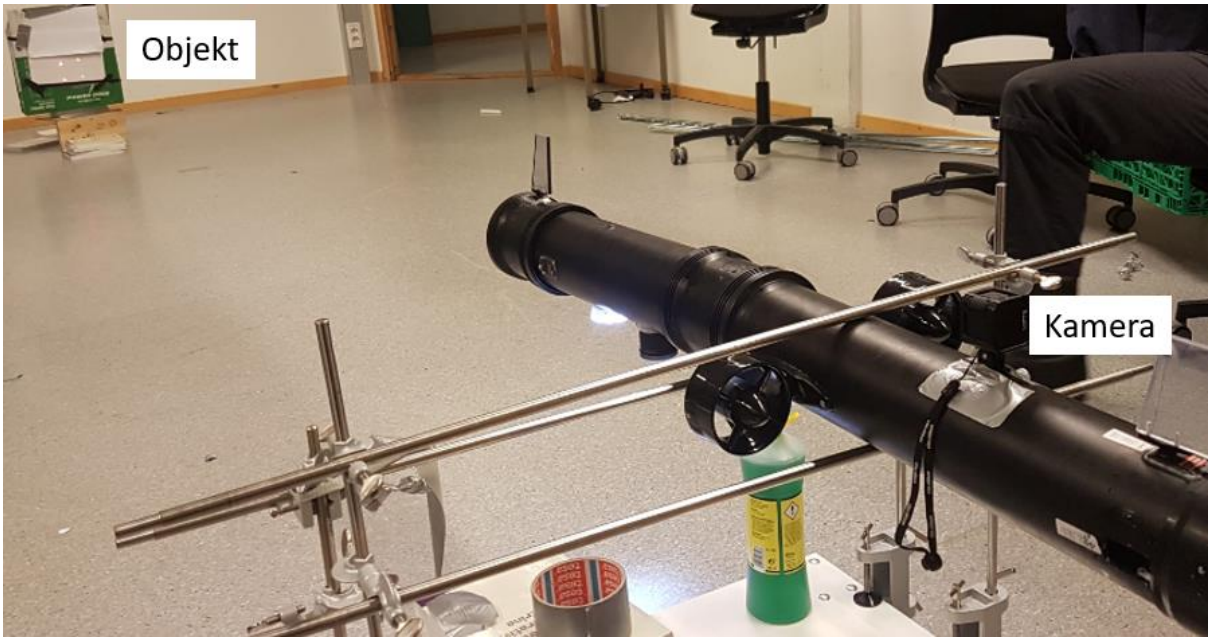


Figur 165: Simuleringen i Matplotlib

Vi noterer oss at dronen plotter kursen til BOX korrekt og samtidig får med plasseringen til objektene. Videre viser plottet at systemet unngikk første objekt ved bruk av Rund-funksjonen, mens den gikk direkte til neste turnlinje for objekt nummer to.

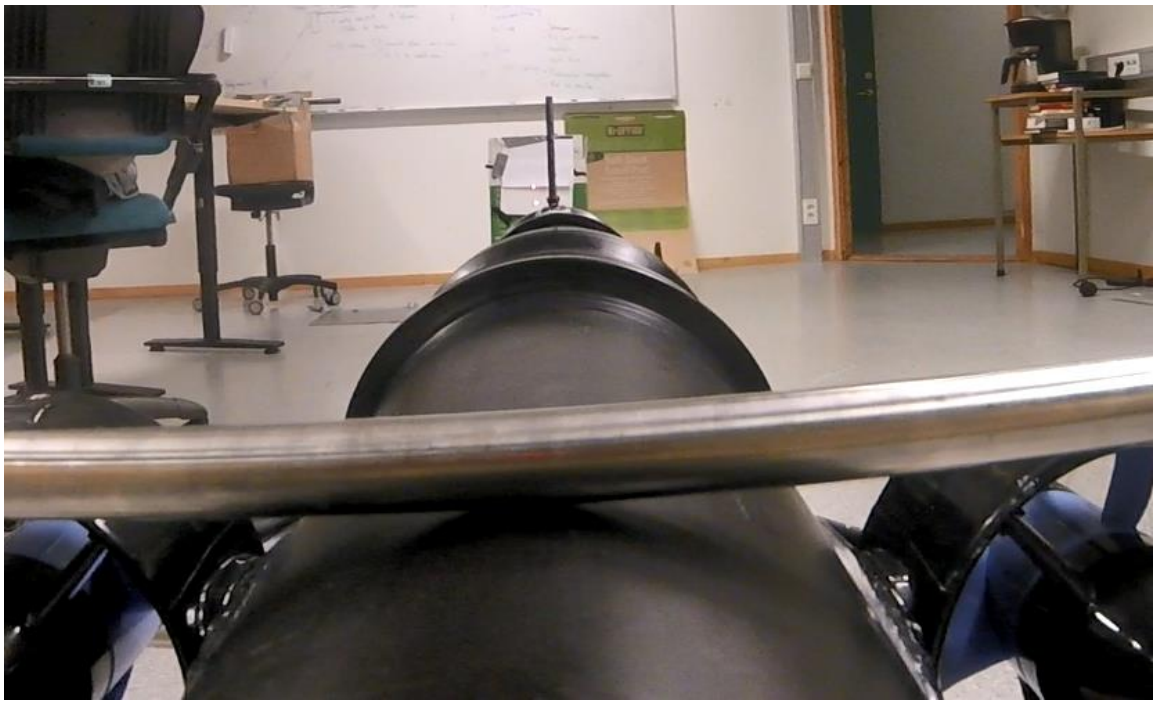
4.3.2 Test av objektunngåelse

Vi tar i bruk det samme kjøretøyet som vi brukte til å teste posisjonsmåling(vist i Figur 156 og Figur 157). Hensikten med testen er å teste de to første stegene av objektunngåelsen til systemet, herav å oppdage et nytt objekt og så forskyve seg styrbord helt til vi ikke lenger ser objektet, som ble behandlet i avsnitt 2.8. I motsetning til testen for posisjonsmåleren må vi derfor bevege oss langs to akser, der vi først etter beste evne prøver å dra donen mot objektet til det er innenfor 360cm. Etter at dronen oppdager objektet, vil finnene gi fullt rorutslag mot styrbord.

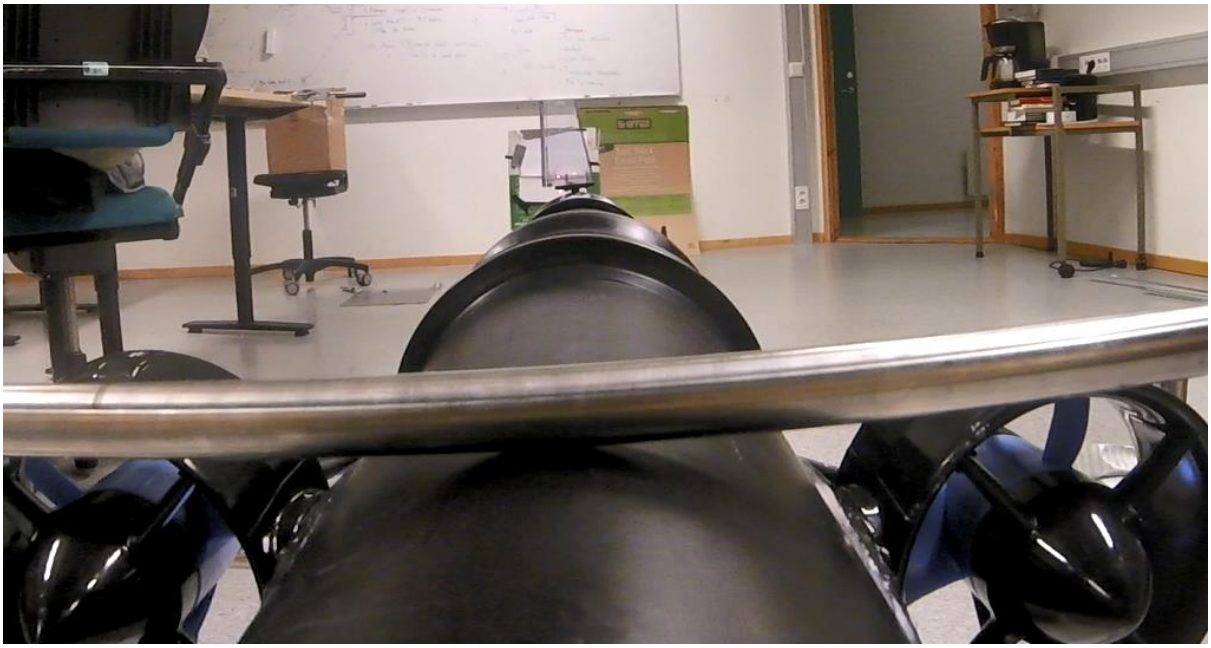


Figur 166: Kjøretøy med drone montert og objekt i bakgrunnen. Kamera er festet bakpå for å filme finnen foran

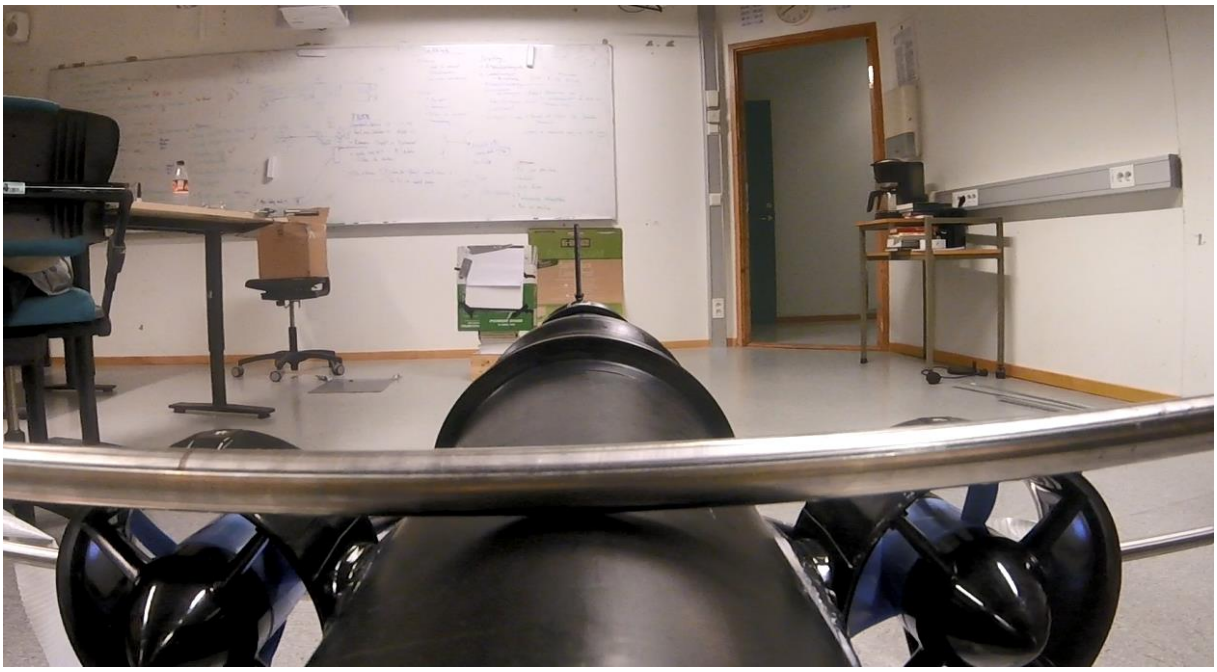
For å isolere testen mest mulig har vi fjernet sensorinput for yaw til PID-kontrolleren, slik at den alltid har som utgangspunkt å ha finnene i 0 grader. Da målingene for testen skjer via et kamera, er det viktig å få frem forskjellen når den ser objektet og når den ikke lenger gjør det. Videre er det viktig å nevne at vi teknisk sett ser et nytt objekt(her veggen) når vi har gått styrbord av det første objektet. Men dette vil være mer enn 360cm unna noe som gjør at vi fint kan teste de to første stegene med avstandsmåleren. Hadde vi derimot fortsatt på samme kurs ville vi begynt å gå styrbord igjen. Ut fra videoen viser vi tre bilder av testen, herav før oppdagelse (Figur 168), ved oppdagelse(Figur 169) og etter å ha mistet objektet igjen (Figur 170:



Figur 167: Steg1 der vi enda er for langt unna objektet(mer enn 360cm)



Figur 168: Dronen oppdager objektet og starter å forskyve seg styrbord over



Figur 169: Dronen ser ikke lenger objektet, og fortsetter rett frem

Som bildene viser evner dronen å oppdage objektet, forskyve seg styrbord over helt til den ikke ser det lenger, og så til slutt fortsette på samme originalkurs. Fra Jetson har vi de lagrede bildene for kamera-feed(Figur 171) og maske(Figur 172):

Figur 170: Bilde fra kamera-feed i det man oppdaget objektet.

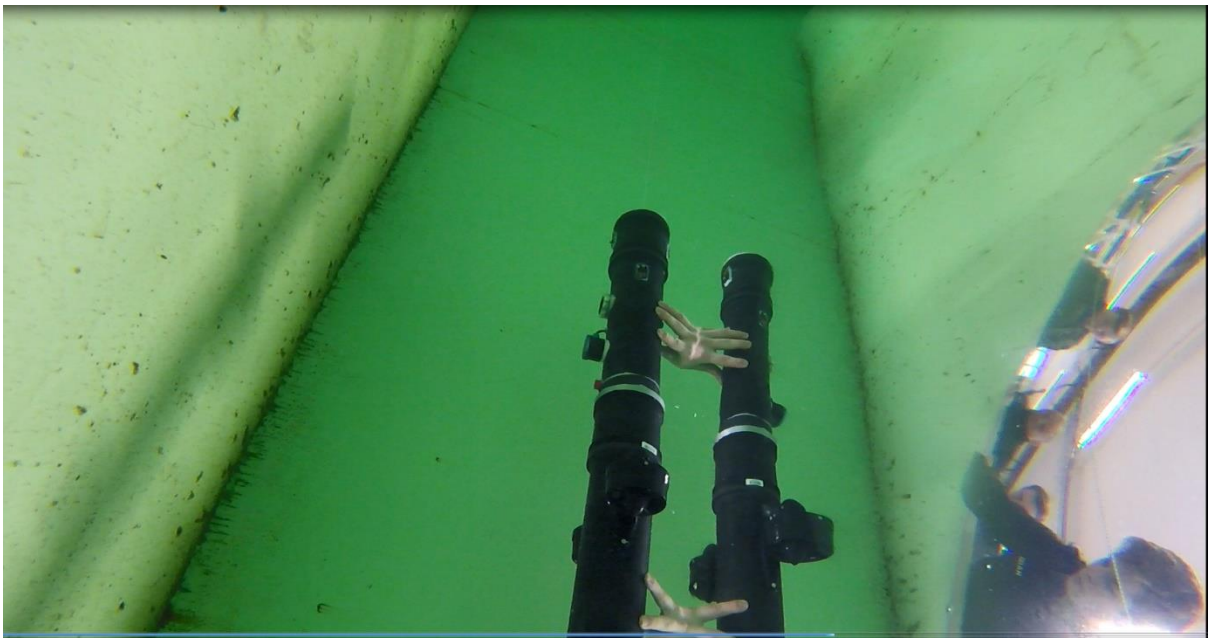
Figur 171: Maske fra bildet over, vi ser at filterer har godt resultat da det bare er laserpunktene som er igjen

4.4 Vanntester

4.4.1 Tetthetstest

I henhold til problemet med servoene der løsningen ble å benytte en annen type vanntette servoer som trekker mindre strøm, ønsket vi å gjennomføre en tetthetstest av dronen. Testen har til hensikt å avdekke eventuelle lekkasjer *før* vi gjennomfører mer kompliserte og lange tester. Årsaken er at vi kan risikere å skade elektronikk i dronen hvis den begynner å ta inn vann. I motsetning til tidligere tester har vi ikke noen sensorer som kan benyttes til å måle hvor tett skroget er. Derfor vil vi selv bedømme hvor det ser ut til å lekke ved å se etter bobler når vi har dronen under vann, og ved å sjekke hvor det er vann inne i dronen etterpå. I tillegg vil vi filme når dronen er under vann slik at vi kan gå tilbake og bedømme igjen hvor det eventuelt boblet.

Testen ble gjennomført i bølgetanken til maskinlaben på Sjøkrigsskolen, der vi holdt dronen under vann i noen sekunder for å se etter lekkasjer. På forhånd var det antatt at problemområdet ville være området rundt servoene. Med problemområde mener vi området der vi forventer/frykter at det ikke er tett, og dermed vil ta inn vann. Området baserer seg på at vi har byttet ut servoer med et noe billigere alternativ, selv om fabrikanten sier at de skal være vanntett.



Figur 172: Dronen holdes under vann, vi kan se tegn til bobler rundt sølvteipen

Etter testen var gjennomført var det noen bobler å spore rundt den ene teipen vi hadde festet på. Når dronen ble åpnet var det ikke noe vann i dette området, som muliggjør at årsaken var en luftlomme i selve teipen. Derimot var det en liten vannlekkasje i den ene servoen slik vi antok på forhånd kunne skje, som et resultat av dette var noen dråper med vann inne i dronen. Som en konsekvens kan ikke dronen sies å være vanntett, som egentlig burde ført til ingen videre testing før servoene kunne byttes.

Men grunnet tidspress var det kritisk at videre testing av andre komponenter i vann ble gjennomført. Selv om konklusjonen av testen var at dronen ikke var vanntett, anså vi mengden vann i dronen som liten nok til å fortsette videre målinger.

Ved siste vanntest ble alle servoene byttet ut til Power-HD servomotorer som er levert av norsk forhandler. Disse var betydelig tyngre enn de kinesiske som ble testet tidligere og var også betydelig tettere. Testen viste at ved seilas på 15 minutter var det kun mindre dråper med vann på innsiden av dronen. Med dette kan det konstateres at det er kun Power- HD servomotorer som er vanntette og at teknikken som er brukt til vanntetting av dronen har vært riktig.

4.4.2 Test av fremdrift og objektunngåelse

For at dronen skal fungere effektivt må vi finne ut hvilken hastighet på motorene som gir god fremkommelighet og evne til å rotere om dronen. Da hastigheten skal settes av operatøren via HMI, vil testen gå ut på å teste ulike mengder motorkraft for å se hva som er bra nok. I henhold til kravene som er satt skal dronen kunne ha fremdrift og rotere selv i motstrøm. I

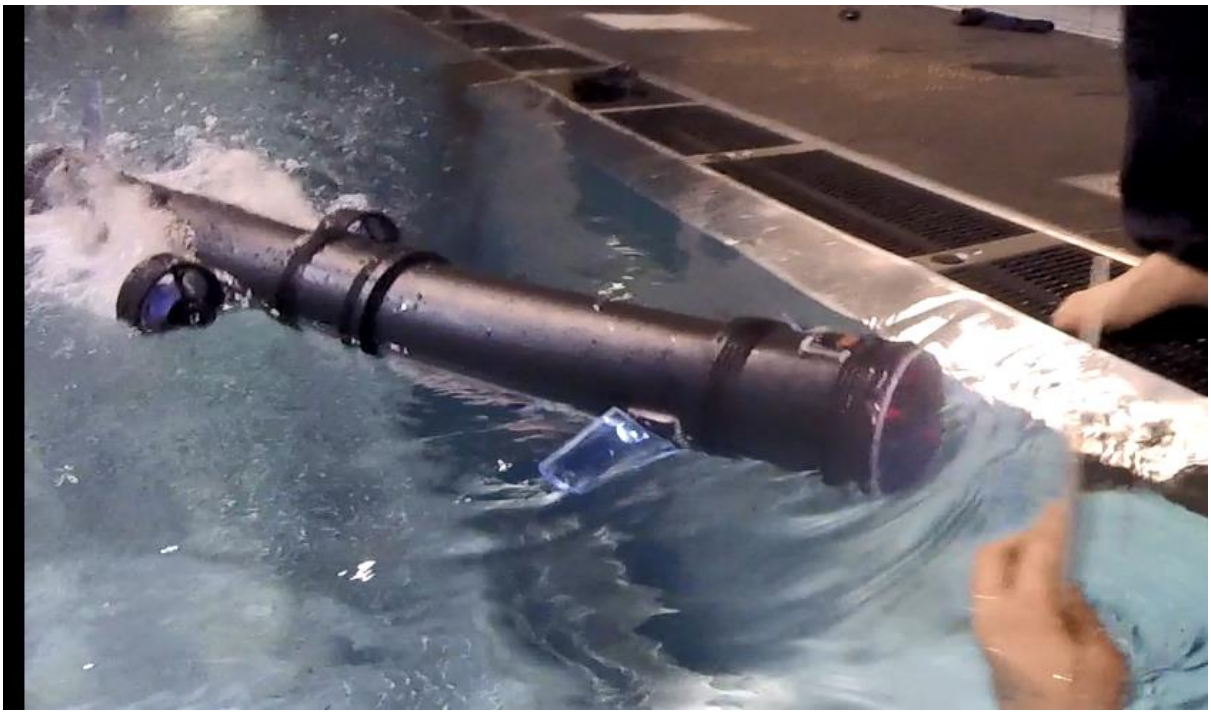
svømmebassenget får vi ikke skikkelig testet med motstrøm eller annen ytre påvirkning, noe som krever tilgang til en annen type testfasilitet vi ikke har.

Da dronen per nå ikke hadde nok vekt i seg til å kunne gå under vann ble den testet ved å ligge i vannoverflaten. Samtidig som vi ønsker å teste fremkommelighet, ville vi benytte muligheten til å teste objektdeteksjonen. Da filmingen ble gjort av en av oss som befant seg i vannet vil kameraet bevege seg og holdes noe ustabilt. Det er derfor vanskelig å vise med bilder når dronen er under vann om den har fremdrift forover eller bakover. Derimot vil strømningen fra dronen være tydelig i overflaten slik vi ser på Figur 174:

Figur 173: Det er tydelig ut fra bildet at dronen beveger seg bakover ved å se på strømningen fra motorene

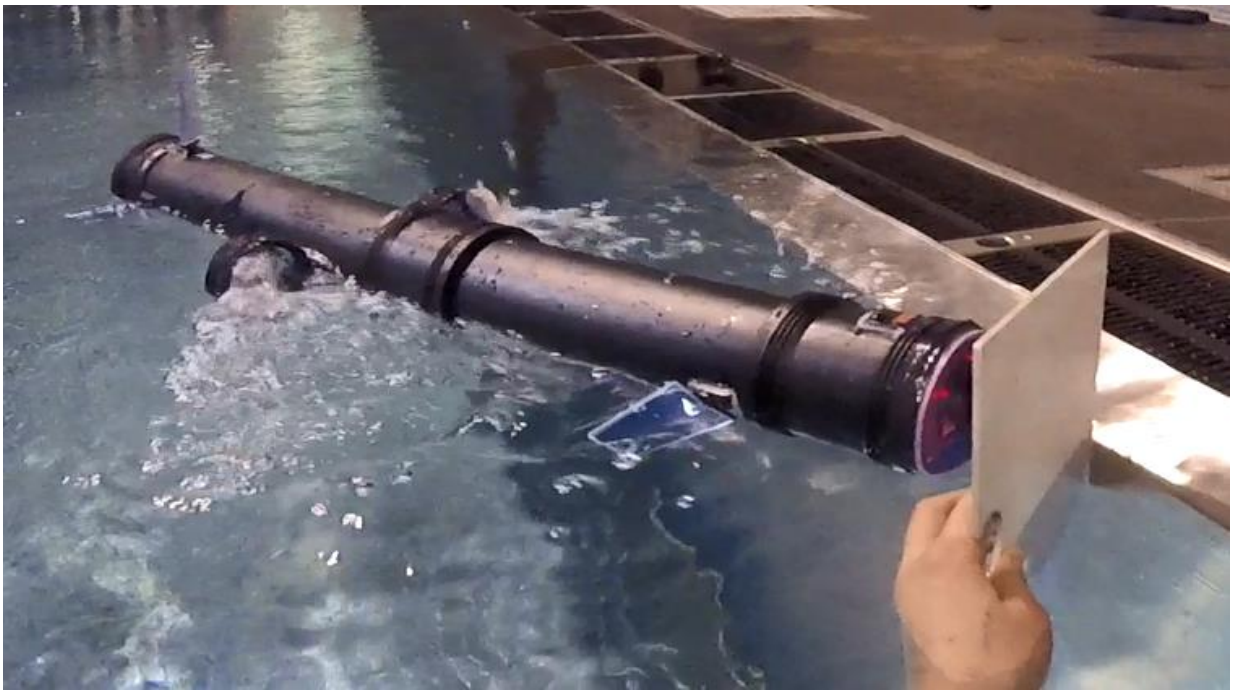
Bildet over var fra første test av dronens objektdeteksjon. Dessverre var kameraet montert feil slik at vi ikke rakk å starte det i tide, som gjorde at vi ikke får med oss objektet som holdes foran dronen. Testen startet med at dronen kjørte rett fremover til et objekt ble plassert ca 2 meter foran den. Slik vi kan se på bildene står finnen for yaw i 60 grader. Det vil si at dronen oppdaget objektet og startet oppgaven med å unngå objektet med å gå styrbord over. Da dronen ligger i overflaten vil vi ha manglende motorkraft slik at dronen ikke klarer å forflytte seg slik den skal. Som en konsekvens klarte ikke dronen å styre unna objektet selv om den prøvde slik den var programmert. Men programmet har og «failsafen» fra 3.2 som gjør at motorene går i revers og styrer unna objektet, slik vi kan se i Figur 176.

Vi gjennomførte en ny test, der kameraet denne gangen var klar til bruk. Nok en gang klarte ikke dronen å forflytte seg styrbord over slik den skal og vi var på vei rett inn i objektet Figur 174:



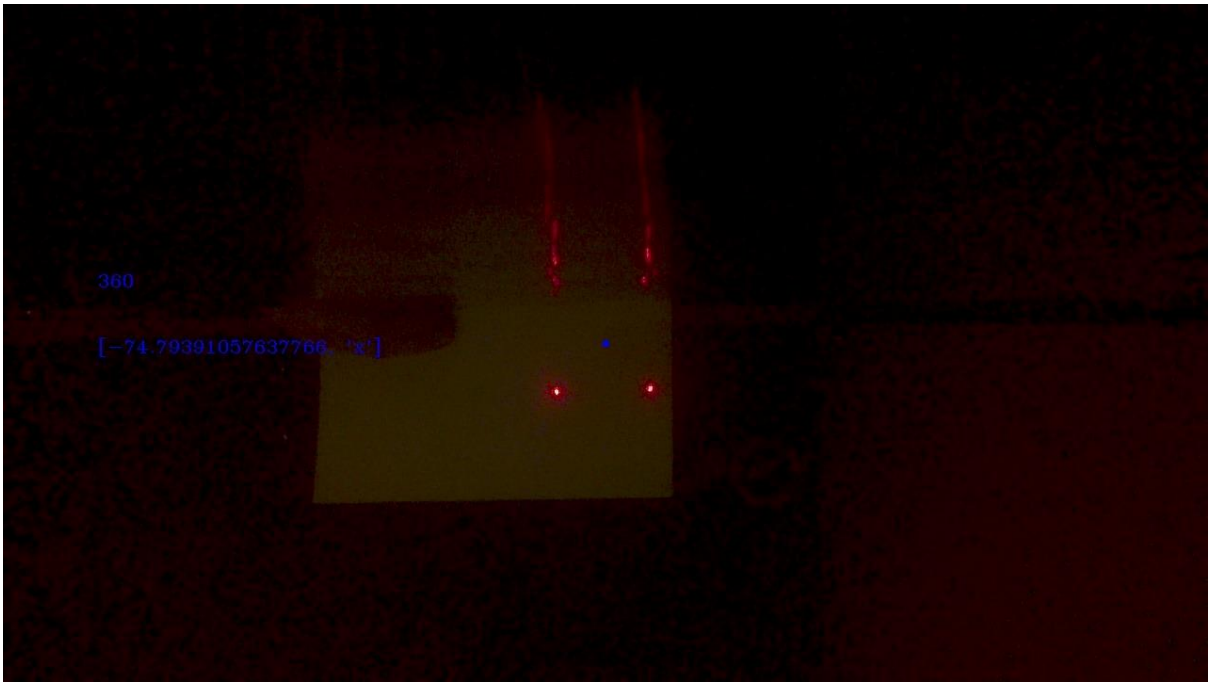
Figur 174: Dronen klarer ikke svinge unna og er på vei inn i objektet. Vi klarer så vidt å tyde at den bakerste finnen for yaw har snudd seg for å forskyve mot styrbord.

Nok en gang kom «failsafen» inn og hindret dronen fra kollisjon, selv om det var i siste liten. Vi ser her enda tydeligere den bakerste finnen som prøver å gå styrbord (Figur 175):

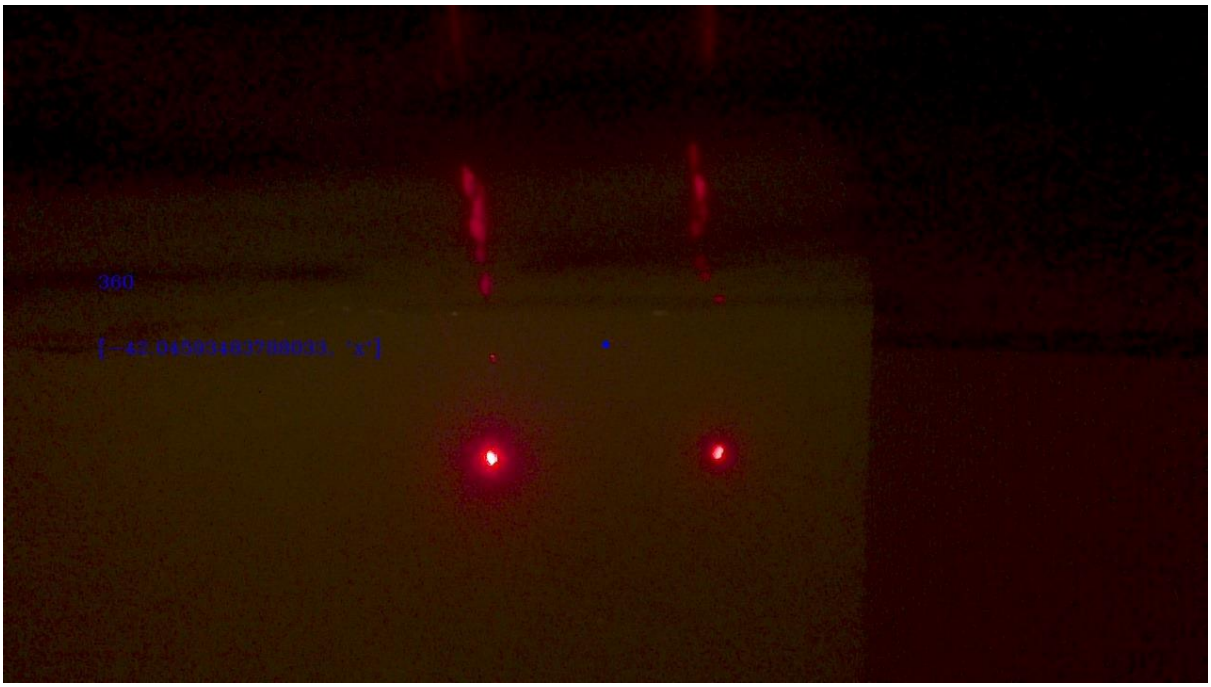


Figur 175: Failsafen setter dronen i revers og hindrer kollisjon, merk bakerste finne

Vi rakk bare å gjennomføre to tester før det ble notert for mye vann i dronen til at det var trygt for elektronikken å fortsette. Samtidig var det flere av servoene som ikke lenger fungerte skikkelig, noe som lover dårlig for å få gjennomført en full funksjonstest. Da dronen ble tatt opp på land og demontert kunne vi hente ut loggbildene vi hadde lagret på Jetson. Det er viktig å påpeke at bildene tas i det øyeblikket dronen oppdager objektet(Figur 177):



Figur 176: Lagret bilde på Jetson av test nummer én. Vi merker oss at de to øverste laserpunktene befinner seg over vannlinjen



Figur 177: Lagret bilde på Jetson av test nummer to

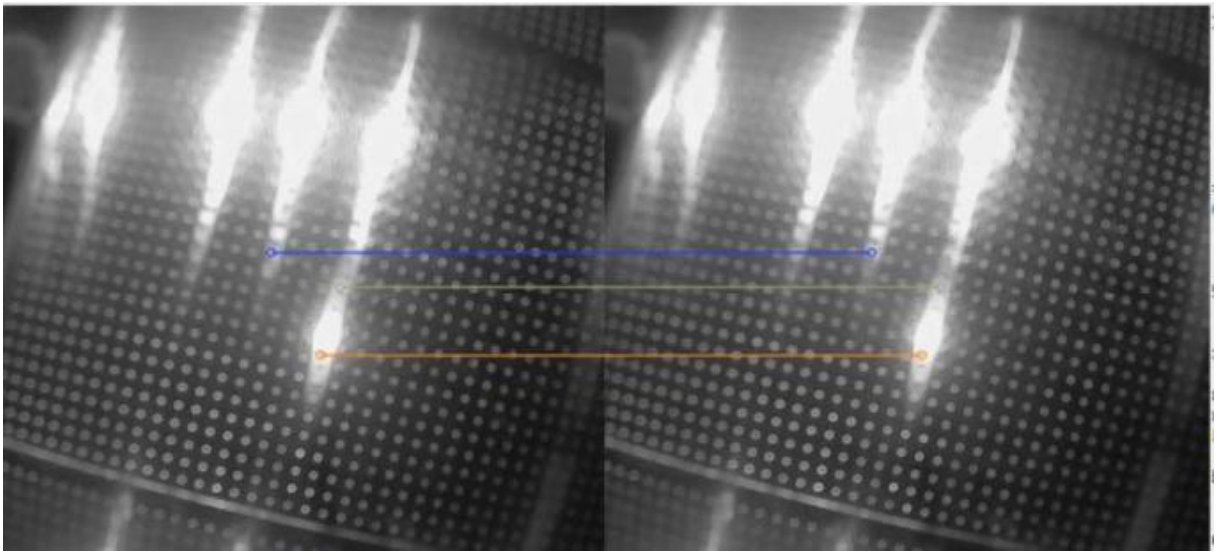
Det er viktig å påpeke at bildet ser såpass mørkt ut grunnet eksponeringstiden og forsterkningen vi har endret i systemet til Jetson. Bilde skal egentlig være helt lyst og krystallklart.

Ut ifra bildene kan vi tyde at laserpunktene er veldig tydelige og skarpe i lyssterke omgivelser (Figur 178) og på et hvitt objekt. For systemet innebærer det at det evner å oppdage og detektere laserpunktene selv i lyse forhold, noe som betyr at den også vil klare det i mørkere forhold der vi bruker enklere filtre.

4.4.3 Test av kamera til posisjonsmåler i vann

I 4.2.2 så man at posisjonsmåleren hadde utfordringer i mørke forhold både på større og små avstander til gulvet. Noe av grunnen til at vanntesten skulle foregå i bassenget på Haakonsvern var den flate bunnen med mange konturer/detaljer. Vi sjekket først dronen på

1.4 meters avstand til en godt belyst bunn ved hjelp av lyskildene til bassenget (Figur 178):

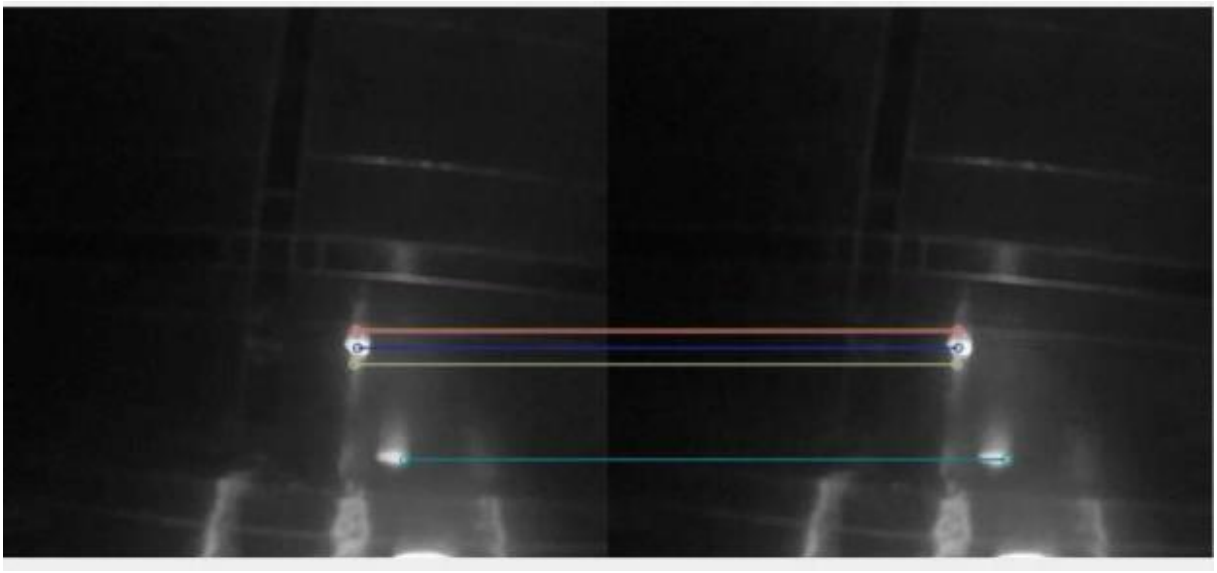


Figur 178: Utsnitt av kamera til posisjonsmåler ved 1.4 meters avstand til bunn. Man ser tydelig at nøkkelpunktene finnes ved de sterke lyskildene

Ut fra bildene ser man flere lysflekker i kamera-feeden. Den som følges av den oransje linjen er dronens lyskilde, mens de andre kommer fra lyskildene til bassenget. Selv om det er flere detaljer og konturer i bassenget er de svakere enn lysflekkene, og blir derfor ikke brukt. For sammenligning har man tatt et bilde med en mobilkamera fra overflaten(Figur180):

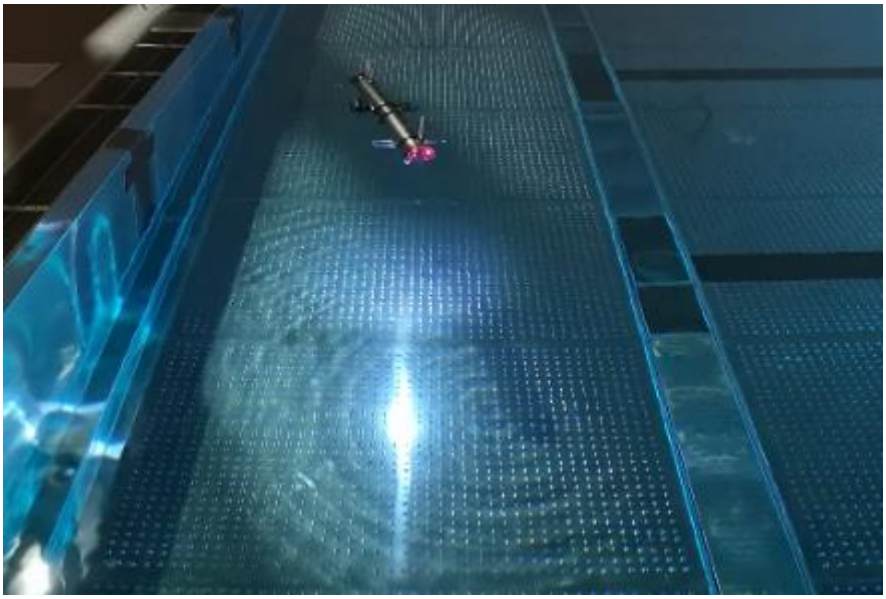
Figur 179: Bilde av dronen ovenfra. Ut fra bildet ser man detaljene enda bedre og lysflekkene enda dårligere

Hvis oppløsning tilsvarende mobilkamera hadde blitt brukt, ville posisjonsmåleren hatt et tydeligere bilde og ikke bare mange lysflekker å forholde seg til. Man ser og at lyskilden til dronen faktisk klarer å gi ganske god opplysning av bunnforholdene. Det ble også en test på 4.5 meters avstand til bunnen for å sammenligne (Figur 181):



Figur 180: Utsnitt fra kamera-feed der avstand til bunn er 4.5 meter. Det er tydelig at kvaliteten på bildet er ganske dårlig

Hvor det er tydelig at oppløsningen til kamera er for dårlig til at posisjonsmåleren kan fungere optimalt. Samtidig ble det også gjennomført en ny test på 1.4 meters avstand der bassengbelysningen ble slått av(Figur 182):



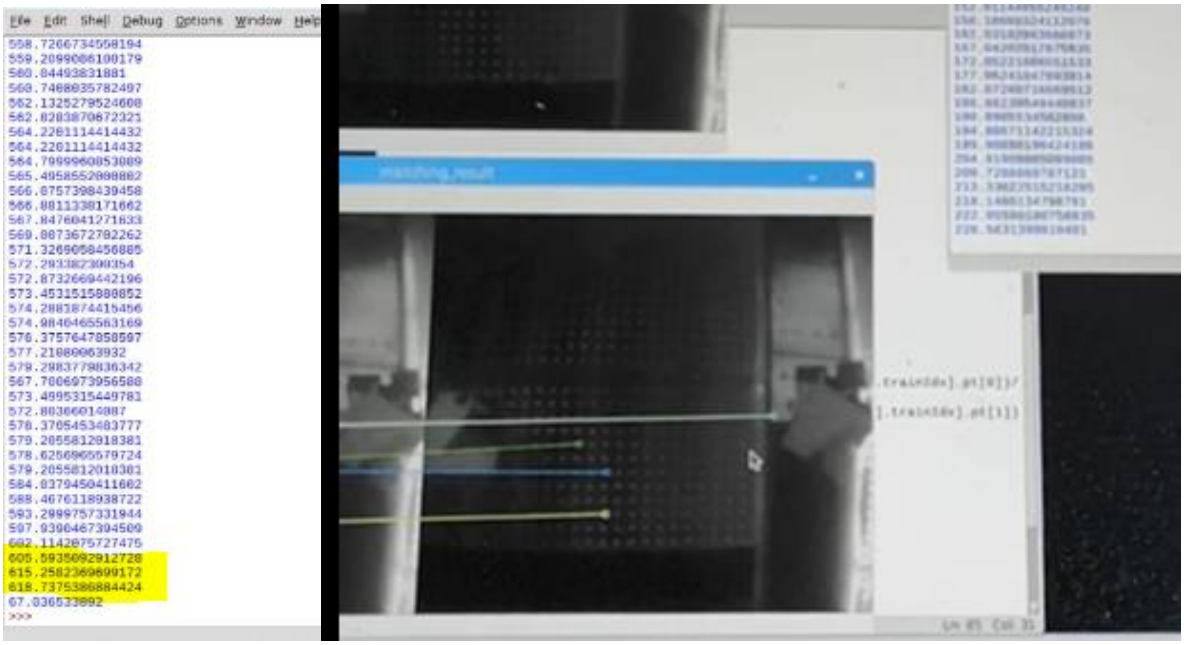
Figur 181: Bilde av dronen uten basseng-belysning. Her ser man at også mobilkamera ville hatt problem med å få gode detaljer

Det er ut ifra bildet tydelig at selv et bedre kamera ville hatt utfordringer med å ha gode detaljer fra bassengbunnen. Det betyr at lyskilden til dronen heller ikke fungerer optimalt.

4.4.4 Ny test av kamera til posisjonsmåler med bedre forhold

For å sjekke opp resultatene valgte vi å legge flere objekter på bunnen av bassenget for å bedre forholdene til posisjonsmåleren. Det ble samtidig gjort ved dagslys med bedre lysforhold i bunnen av bassenget, herav mindre lysflekker. Hensikten var å sjekke opp om den gode nøyaktigheten man fikk fra tørrtestene, også ville gjelde i vann med de rette forholdene og forutsetningene.

Dronen ble beveget langs en linje på ca 6.25 meter, med forbehold om noe unøyaktighet. Langs linjen var det lagt objekter på bunn slik at den skulle ha bedre evne til å få gode målepunkter. Resultatet er vist i Figur 183:

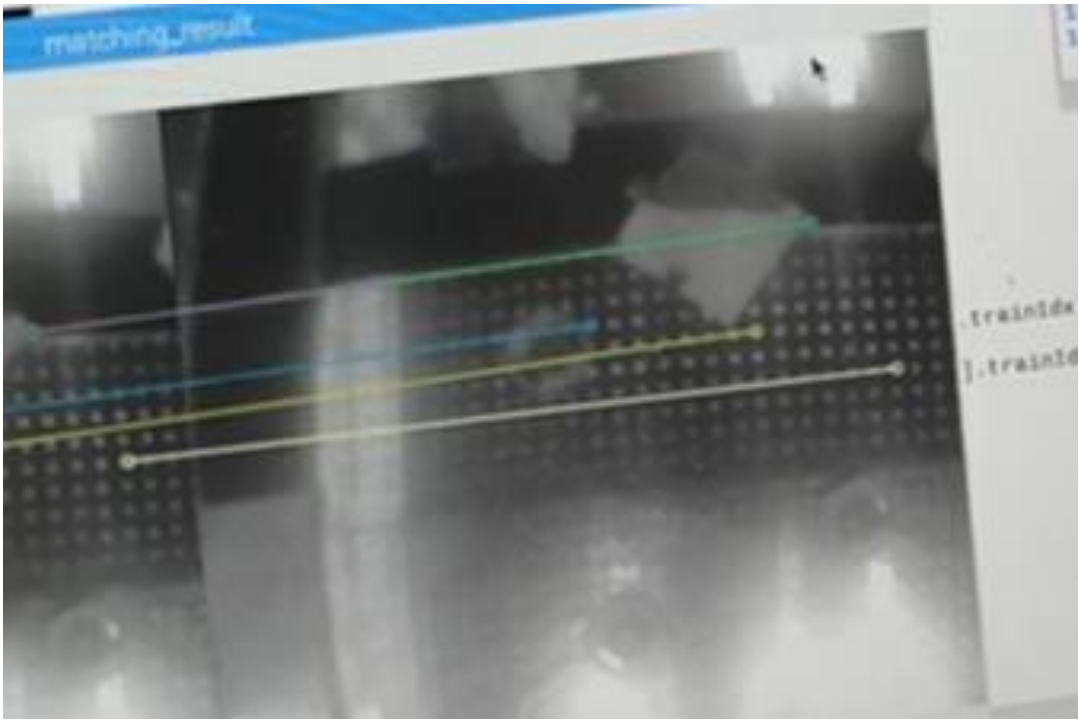


Figur 182: Resultatet fra posisjonsmåleren etter å ha blitt beveget 6.25 meter. Merk at det siste tallet tilhører et mål på hvor lenge man hadde prosessen i gang i programmet. Til høyre ser man at den klarer å velge ut bedre nøkkelpunkter enn tidligere

Man bommer altså med 7cm ut ifra en strekning på 6.25 meter, noe som tilsvarer en feil på 1.12%. Samtidig er dronen blitt beveget av en person, noe som kan tilsvare at den ikke har gått på rett kurs hele tiden, og at man ikke har noen HC-SR04 til hjelp. Det var også viktig å få teste om posisjonsmåleren klarte å «tracke» faktiske objekter på bunnen, og at det ikke bare var heldige målinger. Det ble derfor tatt video av dataskjermen som gjennomfører prosessen på neste måling. Et utsnitt av feeden ses i bildet over, der det ble observert at den bedre klarte å detektere nøkkelpunkter og skille ut detaljer i bunnen.

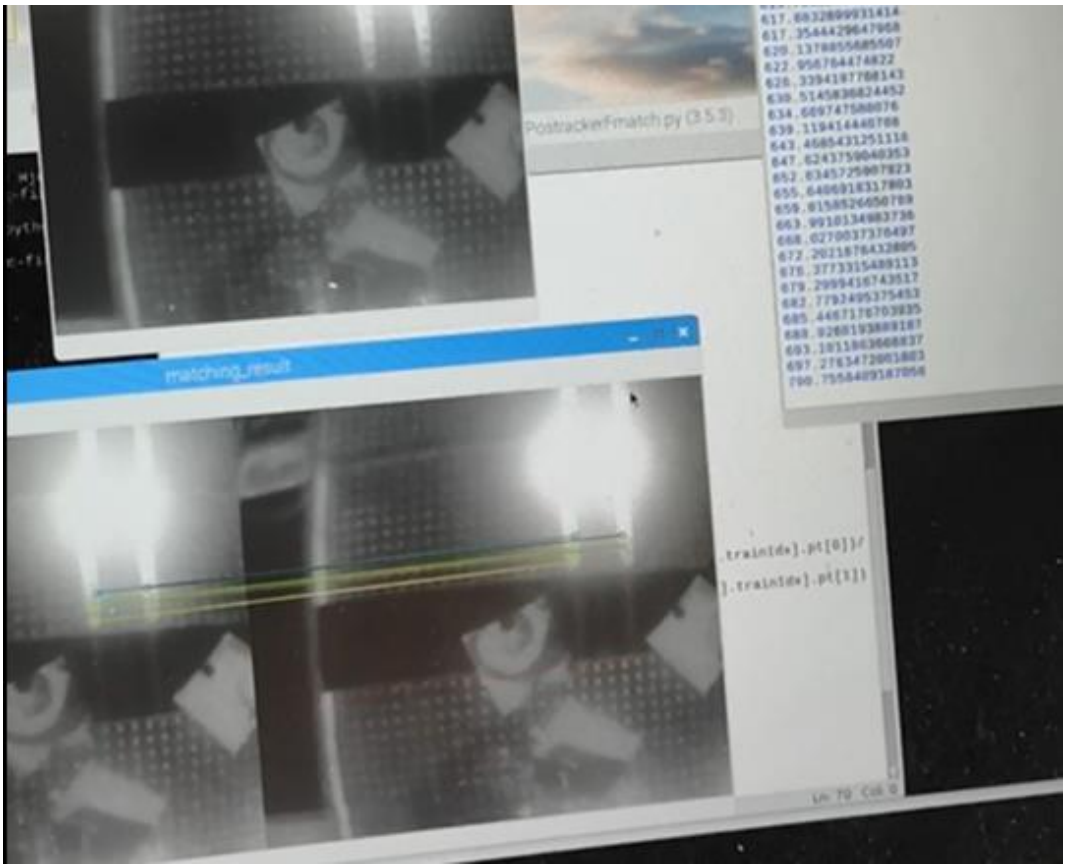
Det ble testet med å skru på mer belysning i bassenget for å se hvor grensen for hva den klarer med tanke på lysflekker i kamera-feeden. Man kjørte test fra midten av bassenget til

enden, noe som tilsvarer 8 meter. Ved noen anledninger observerte man at posisjonsmåleren evnet å velge nøkkelpunkter fra bunnen:



Figur 183: Ved visse anledninger ser man at posisjonsmåleren evnet å detektere faktiske bunndetaljer i feeden

Men da dronen kom til områder der refleksjonen fra lampene i bassenget var sterke, ville også posisjonsmåleren velge ut lysflekkene som nøkkelpunkter. Som en konsekvens måler man 7 meter på en 8 meters lang strekning, noe som er mer unøyaktig som vist i Figur 184:



Figur 184: Feed fra posisjonsmåler. Måleren velger lysflekken istedenfor de detaljerte objektene noe som gir 0 endring i posisjon. Som en konsekvens bommer man med en meter

5.Diskusjon

Dette kapitlet vil ta for seg del-komponentene sett i lys av resultatene fra testene og utviklingen, samt greie ut om eventuell videreutvikling. Videre vil ulike utfordringer som har direkte påvirket utviklingen diskuteres, før til slutt systemet som helhet vil bli drøftet med gjennomgang av systemkravene satt i 2.1.

5.1 Utvikling og testing av del-komponenter

5.1.1 Posisjonsmåler

Første punkt er test av posisjonsmåleren i 4.2, basert på utviklingen som er gjort i 2.3. Der produserte den gode resultater både ved å «følge» HC-SR04, men også når man sammenlignet med faktisk distanse(ved hjelp av laser avstandsmåler) til tross for noen utfordringer med skjevt kamera. Selv om testen ga oss enda bedre resultater på

nøyaktigheten til posisjonsmåleren, er det tydelig at effektiviteten til måleren avhenger av at kamera er plassert nøyaktig. Videre er det viktig at vi unngår å få for mye krengning på dronen, da det igjen vil føre til feilberegning av posisjon. Men slik det ser ut vil en eventuell feil i vinkel først forårsake problemer når vi støter på kanter, som for eksempel bassengkanten i svømmehallen vi skal teste i. Selv om det kan skape unøyaktigheter, viser målingene ellers at posisjonsmåleren er svært nøyaktig og klarer å beregne posisjonen til dronen. Når det kommer til usikkerheten i målingene på opp til 30cm er dette fortsatt innenfor det vi vurderer som nøyaktig nok sett i lys av hvilke komponenter som er blitt brukt.

Man så at forskjellen mellom posisjonsmåleren og HC-SR04 aldri var større enn 3.5cm når vi endret stillingen på kameraet. Samtidig kan man ikke utelukke at det skyldes andre faktorer at den ga større unøyaktigheter tidligere, som for eksempel vibrasjon. Det er nevnt i 3.4 at man har dårligere oppdateringsrate på kamera-feeden ved å bruke en Pi. Det gjør posisjonsmåleren mer utsatt for «rykninger» i bildet, med andre ord at man får en kraftig endring i feeden ved for eksempel en bråstopp. Da vil dårlig bildeoppdateringsrate kunne gjøre at man ikke registrerer hele bevegelsen, som igjen gir dårlige målinger.

Men man må også ta for oss faktumet at testen er gjennomført i et klasserom hvor forholdene er annerledes enn hva man kan forvente for operasjonsområdet. For det første er det ikke noe vann imellom linsen og bunnen/gulvet, som igjen gir klar og god sikt for kameraet til enhver tid. For det andre er lysforholdene optimale med god belysning fra taklampene som igjen gir et klart bilde av gulvet. For det tredje er testen designet for bevegelse langs en akse(ref 4.2), som gjør at posisjonsmåleren ikke får noen påvirkning/forstyrrelse fra y-aksen. Det hindrer og at man utsetter måleren for noen form for vibrasjon, som igjen kan forventes når den skal operere. Når en isolerer testen slik det gjøres, bør man også forvente høyere nøyaktighet. Dette da de ulike forstyrrelsene som er nevnt vil bidra til å svekke nøyaktigheten enda mer, som kan føre til at posisjonsmåleren blir for unøyaktig.

Ut ifra testene av posisjonsmåleren i dårligere lysforhold og sikt(ref 4.2.2 og 4.4.3) blir konsekvensene av å bruke de valgte komponentene enda tydeligere. Testene viser at posisjonsmåleren får et bilde med dårlige detaljer til bunn når man befinner seg i mørke forhold eller ved større avstander til bunn. For det første skyldes det at oppløsningen til kamera-feeden blir dårlig grunnet bruk av Pi, som gjør at man ikke klarer å velge ut gode konturer og områder i bunnen. Dette kan man begrunne med bakgrunn i hvor mye bedre oppløsning et bilde fra en vanlig telefon ga oss i 4.4.3 enn den tilsvarende kamera-feeden. For det andre viste også testen i et mørkere basseng at selv et telefon-kamera vil ha utfordringer med å få detaljer med lyskilden til dronen. Dette gjør at selv en bedre

kameraoppløsning vil bruke lysflekken fra lyskilden som nøkkelpunkt. Da lyskilden er fastmontert på dronen vil den relativt i forhold til kamera alltid stå i ro, som gjør at posisjonsmåleren måler ingen bevegelse. Man må altså ha en bedre lyskilde til dronen for å alltid kunne gi gode bilder av bunnen. For å summere vil disse to svakhetene i systemet gjøre at posisjonsmåleren med nåværende komponenter kun klarer å tilfredsstille kravet når man har gode lysforhold og kort avstand til bunnen.

I utgangspunktet kan man da argumentere for at kravet i sin helhet ikke tilfredsstilles. Samtidig har man gode resultater å vise til *når* kamera-feeden får god oppløsning og lysforholdene er gunstige. Man kan på den ene siden ikke benytte dronen slik den er nå til å gjennomføre vilkårlige operasjoner og oppdrag, basert på de svake testene under dårligere lys- og siktforhold(ref 4.2.2 og 4.4.3). Men på den andre siden er oppgaven ment å vise et konsept som kan videreutvikles, herav det er nødvendig å bruke mer penger og tid for at det skal kunne bli et ferdig produkt. Ser man derfor kun på de nøyaktige testene som er gjort ved de rette forholdene både tørt(ref 4.2) og i vann ved 4.4.4 kan man argumentere for at posisjonsmåleren fungerer. De andre testene ved dårligere forhold kan brukes til å argumentere for at problemene som oppstår er direkte korrelerbart til komponentene som er valgt. Det er basert på hvor bra målingene var når man oppnådde god oppløsning.

I motsetning til andre sensorer og komponenter finnes det ikke en guide på nettet til hvordan man skal designe posisjonsmåleren, men er noe vi har klart å konstruere basert på inputs og ideer fra andre konsepter. Det er en mangel på gode og nøyaktige posisjonsmålere for undervannsdroner. Derfor kan det tenkes at et relevant bruksområde for del-konseptet vårt er å kunne nyttes til bruk på kommersielle enheter som ønsker en slik egenskap. I tillegg til dette, er mulighetene der for at posisjonsmåleren kan videreutvikles til bruk for større operasjoner hvis mer robuste og nøyaktige komponenter tas i bruk. Det hadde vært interessant å se eventuelle tester med en Jetson istedenfor, for så å sjekke om den var bedre rustet mot såkalte «rykk» og dårligere forhold der man trenger bedre oppløsning.

Bruk av GPS for oppdatert posisjon

En mer direkte og billig utvikling hadde vært å benyttet oss av GPS'en vi har tilgjengelig. Hensikten var at dronen skulle ved mulighet eller ved faste punkt(som for eksempel turnpunktene) dykke opp til overflaten og bli værende der til den hadde en GPS-fix. Årsaken til at man ikke har benyttet seg av GPS er at man driver tester inne, både ved tørrtestene og ved vanntestene. GPS'en ga en unøyaktighet på noen meter når man befant seg inne. Med tanke på at man i BOX-systemet skal teste i et svømmebasseng som er rundt 16x25m, vil det være en stor unøyaktighet.

Videre ønsket man å lagre differansen mellom posisjonsmåler og GPS-posisjon for å se om det var en form for trend eller konstant feil som man eventuelt kunne drevet videre arbeid med å luke ut. En videre utvikling kunne også implementert maskinlæring som benyttet nettopp denne informasjonen til å forbedre posisjonsmåleren enda mer. Bruk av GPS fusjonert med posisjonsmåleren ville vært interessant å se videre på.

5.1.2 Avstandsmåler

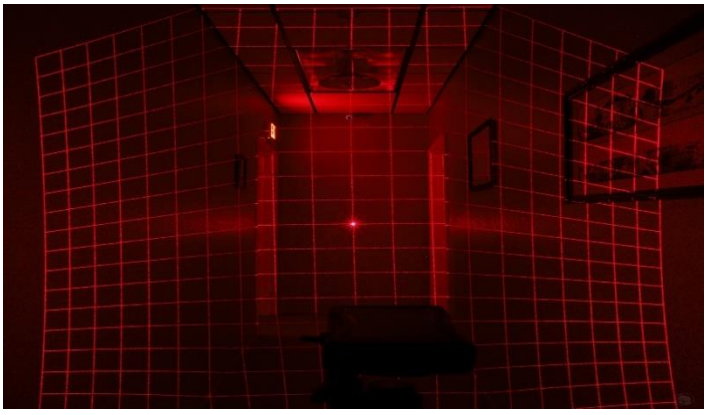
Det er viktig at man tar en vurdering basert hva en definerer som «nøyaktig nok», for å kunne vurdere om testen av avstandsmåler i 4.1 gir tilfredsstillende resultater. I utviklingen av avstandsmåleren(ref 2.2) har man snakket om at den skal brukes til å unngå objekter ved at man gir styringssystemet avstand til objekter. Der ble det videre nevnt at en unøyaktighet på noen cm ikke vil skape store feil, da det bare vil føre til at dronen turner litt tidligere. Dette er selvsagt med forbehold om hvor store feilen blir, noe som igjen avgjør om man kan si at testene indikerer et nøyaktig nok avstandsmålesystem. Som sagt så er styringssystemet ikke avhengig av om dataen som gis ut er for eksempel 90cm istedenfor 100cm, da den vil agere når objektet kommer innenfor en viss avstand. Resultatet av differansen i målingene viser en maks forskjell på 18cm i mørke og 12cm i lyset, men at de har et snitt som ligger tilnærmet rundt 6 og 4cm.

Begge målingene gir den maksimale forskjellen på store distanser mellom kamera og objekt. Dette indikerer at prediksjonene våre om økende unøyaktighet ved større avstander som en kombinasjon av algoritme og ujevn montering stemmer. En maksimal forskjell på 18cm skjer altså ved avstander over 300cm, som ikke utgjør noe store problem for styringssystemet slik det er satt opp. Bakgrunnen er at styringssystemet får input om avstanden til objektet slik at den kan bedømme når den er ved siden av det, ref 2.8. En unøyaktighet på 20cm vil ikke hindre avstandsmåleren fra å oppdage objektet, men vil kunne gjøre at styringssystemet feilbedømmer når dronen er på linje. Hvis dronen bedømmer det til å være lengre unna enn det er, vil det bare gi et slingringsmonn i det dronen skal turne tilbake til turnlinjen(hvis den benytter Rund-funksjonen, ref 2.8). Derimot vil en bedømming av at objektet ligger nærmere enn det faktisk gjør, kunne gjøre at dronen vil turne innover mot turnlinjen for tidlig. Men som beskrevet i 2.8 forskyver dronen styrbord ved objektoppdagelse, som gjør at den også beveger seg fremover i den perioden den enda ser objektet. Beregningen av når den er på linje gjøres *etter* at den har mistet objektet, slik at dronen har et forventet slingringsmonn som antas å være større enn 20cm.

Selv om man i testen har en maks forskjell på under 20cm, er disse testene gjennomført isolert og tørt på et flatt objekt. Man har her også en utfordring med at kamera ikke eksponeres for skittent og uklart vann, eller vann i det hele tatt. Der posisjonsmåleren trenger

god oppløsningen til bunnen, er ikke det en nødvendighet for sensoren da man kun er avhengig av å skille ut laserpunktene fra resten. Ref 2.2 vilmani ha større utfordringer hvis en møter på objekter med andre fasonger og ujevnheter. Det vil gjøre at feilen i målingene kan bli større og mer unøyaktig, som igjen forårsaker større feil i styringssystemet.

Slik designet er nå vurderer vi at det trengs utvikling på flere punkter for at det skal kunne nyttes i kommersielle droner. For det første må laserne monteres slik at de står vinkelrett på hverandre for å unngå feilen som systemet vårt har. Det har blitt foreslått i 2.6 ulike metoder for hvordan dette kan gjøres, men det krever at det gjennomføres en konstruksjon som koster mer enn vanlig 3D-print. For det andre vil en videre utvikling mot et gridsystem(se figur 186) av lasere gi systemet mulighet til å detektere flere objekter med ujevne flater, og til å kunne kartlegge konturene mer nøyaktig. En annen ting er at laserne trenger å gjenspeiles godt og kraftig på objektet det ser. Valget av rødlys er dermed ikke det beste basert på at det enkelt brytes i vann. For videreføring trengs det dyrere og sterkere lasere, men disse krever godkjenning av norske myndigheter for å benytte. En løsning kan her også være å bytte ut kamera og lasere med sine infrarøde kollegaer, slik at alt foregår i det infrarøde spekteret. Vi har ikke kompetanse nok på feltet til å vite om det vil bedre kvaliteten eller ikke, men er noe som kan sjekkes ut og om mulig utvikles.



Figur 185: Eksempel på et gridsystem av lasere. Kilde:https://www.ghoststop.com/Laser-Grid-GS1-p/laser-lasergrid-gs1.htm

En slik løsning setter enda sterkere krav til nøyaktig montering og oppkobling, men vil som resultat kunne gjøre dronen enda mer robust og samtidig utvikle nye bruksområder for dronen.

Alt i alt har vi et system som med en unøyaktighet opp på ca 20cm kan bedømme avstand til objekter ved hjelp av billige komponenter. Konsekvensene av unøyaktigheten kunne vært at styringssystemet feilberegner posisjonen, men vi vurderer unøyaktigheten som liten nok. I henhold til kravet vi satt for avstandsmåling i 2.1 ender vi opp med et deteksjonsområde på maks 360cm. Vår vurdering er at det skal være innenfor det man klassifisere som god

margin, da dronens videre krav til manøvrering bør evne å unngå objekter med en såpass tidlig varsel.

5.1.3 Styringssystem og fremdrift

I 4.3 testet vi styringssystemet. For simuleringen var det mer en sjekk på at programkoden var korrekt og gjorde det vi forventet av den, og er ikke så mye mer å diskutere. Det som er mer interessant å se på er hva dronens egenskaper blir gjennom et slikt avgrenset styringssystem, ref 2.8. Ser man tilbake på hovedmålet i 1.2 skulle dronen operere på lik linje med en vanlig kommersiell kablet drone. Grunnet tid vil man ikke klare å lage en stor nok «pakke» for styringssystemet som gjør den kapabel til å reagere på alle mulige utfall samt ha flere ulike moder å operere på. På den ene siden vil en slik avgrensning gjøre at dronen ikke vil ha de samme egenskapene navigasjonsmessig som en kablet drone. Men hvis vi fokuserer på andre delen av målet vil en slik avgrensning kunne vise at systemet *kan* evne å utvikles til å oppnå samme muligheter. Bakgrunnen er at vi valgte å fokusere på at styringssystemet skal vise at hver del-komponent kan jobbe sammen og gjennomføre jobben de er designet for å gjøre. Selv med en slik avgrensning, er styringssystemet designet for å kunne teste avstandsmåler, posisjonsmåler og styringsevne samtidig.

Ut ifra testen i 4.3 vil dronen evne å reagere slik vi ønsker på objekter. Utfordringen ligger i om den fysisk klarer det. Vi har satt finnene slik at den skal forskyve seg styrbord over helt til den ikke lenger ser objektet. I testen utføres forflytningen av oss ved at vi beveger kjøretøyet sidelengs. Det er derfor essensielt å teste om dronen fysisk klarer å forflytte seg slik vi ønsker når den befinner seg i vann. I 4.4 ga dronen indikasjoner på at det å forflytte seg var en utfordring i hvertfall når den ligger i vannoverflaten, noe som ikke tilfredsstiller krav 4 ifra 2.1. I henhold til målet om samme egenskaper som en kablet drone, er de fleste kommersielle droner i stand til å forflytte seg slik operatøren ønsker, i overflaten.

Det finnes flere mulige utviklinger for dronen som eventuelt kan løse dette. En mulighet er å erstatte motorer og finner med thrustere som vi ikke har tilgjengelig, ref 1.3. Thrustere ville evnet å forskyve dronen styrbord over uten videre utfordringer. Kriteriene for å forskyve seg styrbord har vi vist i 4.4 at fungerer slik de skal, noe som betyr at vi for overgang til thrustere kun måtte bytte ut kommandoene Arduino sender ut(med forbehold om eventuelle programmeringsfeil). I stedet for å måtte bytte ut deler og gjøre om programmet kunne vi fått dronen til å dykke ned og ha tilgang til alle fire yaw-finnene. Da kunne den manøvrert seg bort til ønsket plass for så å dykke opp igjen. Selv om løsningen kunne fungert bøter det ikke på det faktum at dronen da heller ikke er manøvrerbar i overflaten.

I sakens kjerne er det også viktig å benevne at «failsafen» vi har implementert i 3.2 virket med begge testene i vann. Det betyr at styringssystemet har evnen til å unngå objektet selv

med manglende forskyvningsevne. Utfordringen ser derfor ut til å ligge mer i skrogdynamikken enn i software-delen ut ifra testene i 4.3 og 4.4.

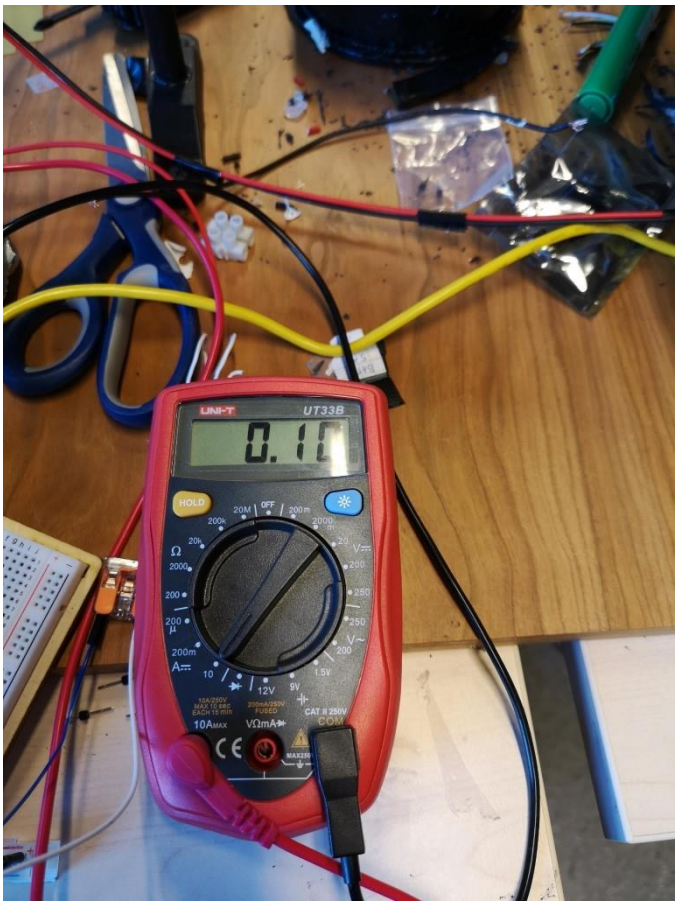
5.1.4 Skrogtetthet

I 3.4 ble dronen vanntestet ved maskinlabben ved Sjøkrigsskolen. Vi fant da mindre lekkasjer og tettet disse. Ved neste test av dronen viste det seg at dronen hadde tatt inn vann i områderundt servomotorer. Etter grundigere undersøkelse viste det seg å være de nye servomotorene som hadde blitt satt inn. Denne typen servomotor var oppgitt fra fabrikk at de var vanntette, noe de over hode ikke var. Den sikreste indikasjonen på at det var noe feil med servomotorene var da skroget ble trykksatt som følge av at frontdomen ble satt på, rant det ut vann rundt akslingen. Denne delen skulle vært tettet av produsent, men ble ikke tettet. Dette har derfor skapt store utfordringer for oppgaven også grunnet den sene leveransen av servomotorene.

5.2 Utfordringer

5.2.1 Utfordringer med hardware

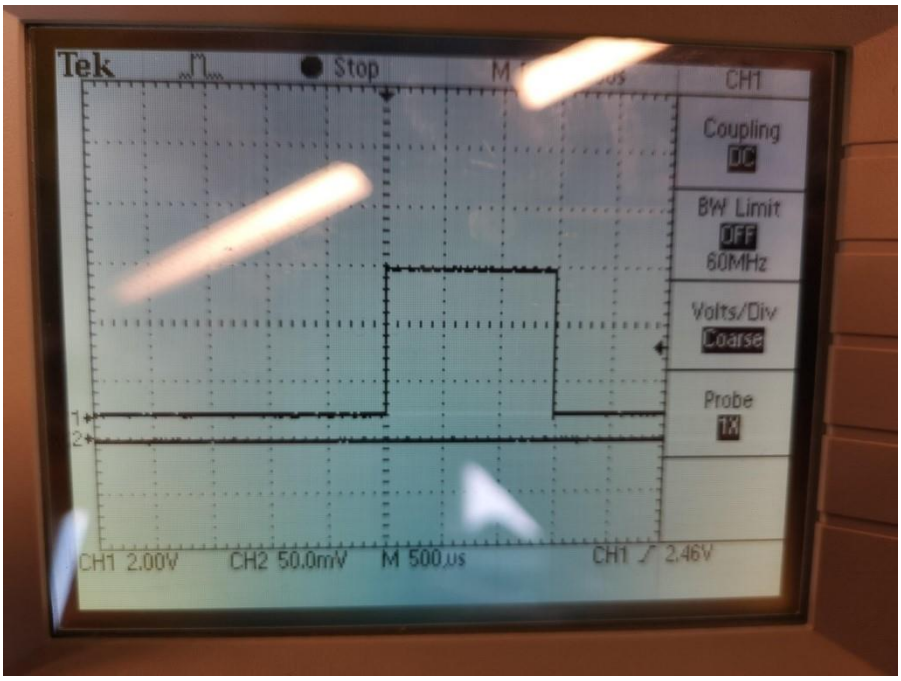
Under alle prosjekter skjer tilbakefall og uhell. Dette skjedde også dette prosjektet mot slutten av prosjektperioden da dronen ble mistet i bakken fra 40cm under en test. Under forsøkt oppstart kunne vi se at LCD-skjermen var intakt og gav ut informasjon fra dronen som tiltenkt. Batteriene leverte også en jevn spenning over alle fire cellene som tydet på at



de ikke hadde fått noen problemer etter fallet. Videre koblet vi opp alle sensorer og datamaskiner opp igjen og fikk med en gang problemer med kommunikasjon til servomotorene. Servomotorene trakk 0.1 amper uten at de hadde fått signal om endring i posisjon og visuelt var det heller ikke noen endringer i posisjon. En annen indikator på at noe var feil var at servomotorene endret posisjon beskrevet som «rykninger» når for mange servomotorer endret posisjon.

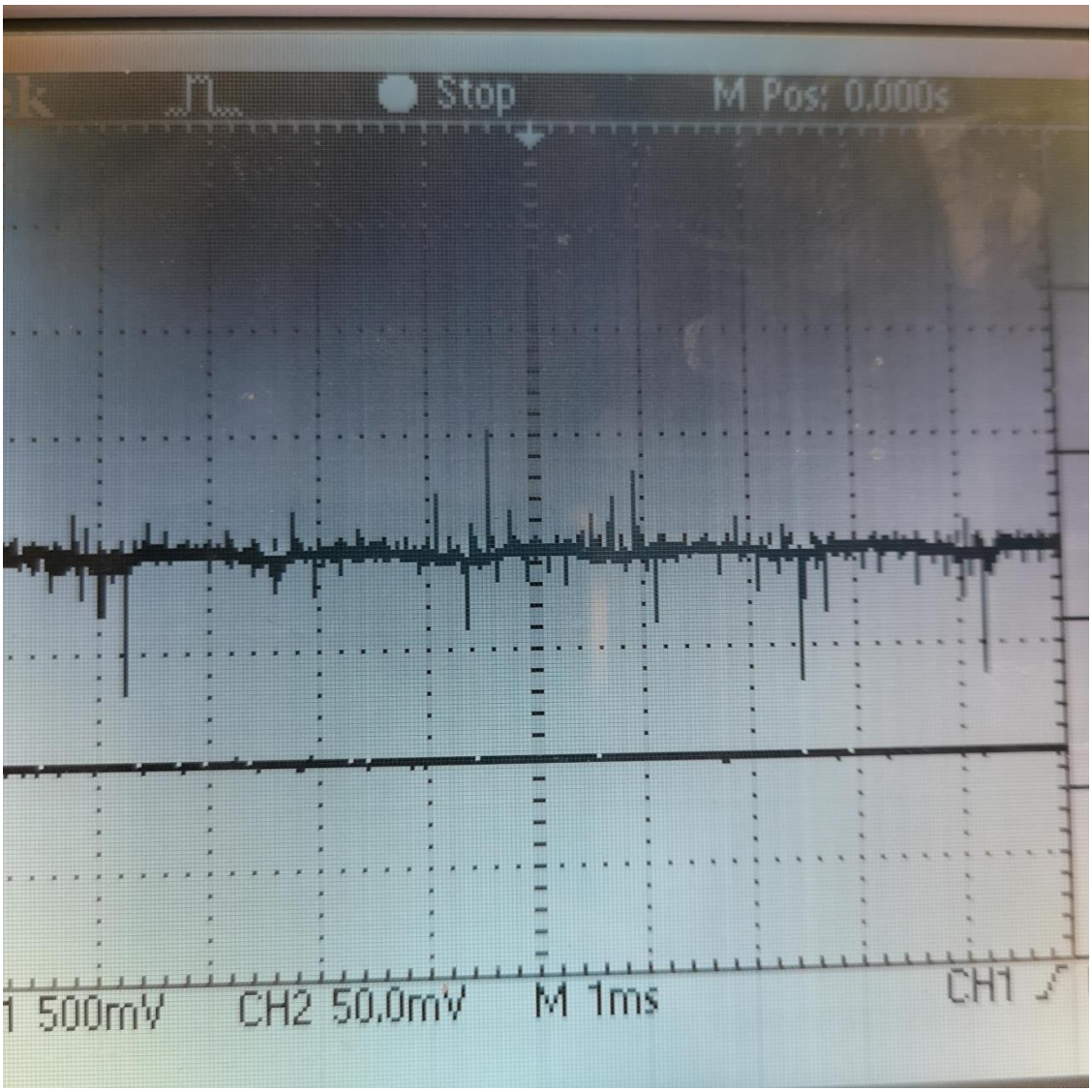
Servomotorene er koblet i parallell fra en og samme omformer som rettet søkelyset mot denne komponenten. Omformeren er svært kompleks med en mengde kondensatorer, spoler og dioder for å kunne levere korrekt spenning ved ulike belastninger. Første søk ble derfor å sjekke hvor mye strøm omformeren kunne levere uten å gå under 4,8V som er grenseverdien oppgitt fra leverandør. For å simulere lasten ble det brukt noen mindre elektromotorer som vi regulerte motstanden for hånd. Måleren viste 7 amper da spenningen gikk under 4,8V. Dermed kunne det fastslås at det ikke var effekten som omformeren leverte som var problemet.

Neste område vi trodde det var en feil på var signalet ut fra Arduinoen. Arduino leverer 8-bit pulsmodulerte signaler som kan levere 256 steg med oppløsning. Er det derimot en feil på utsignalet fra de digitale utgangene kunne det skape en feil tolkning for servomotorene slik at de hele tiden ville bytte mellom to posisjoner som svarer til problemet med «rykninger» på servomotorene. For å teste dette ble et oscilloskop koblet til mellom GND(jord) og hver av de digitale utgangene. Alle inngangene varblitt satt til en satt posisjon som skulle gi ut samme signal på hver utgang. Hver enkelt utgang ble så målt på lik måte og resultatet ble helt likt på alle utgangene. Dermed kunne vi se bort ifra feil på signalet ut fra signalkilden som problemet.

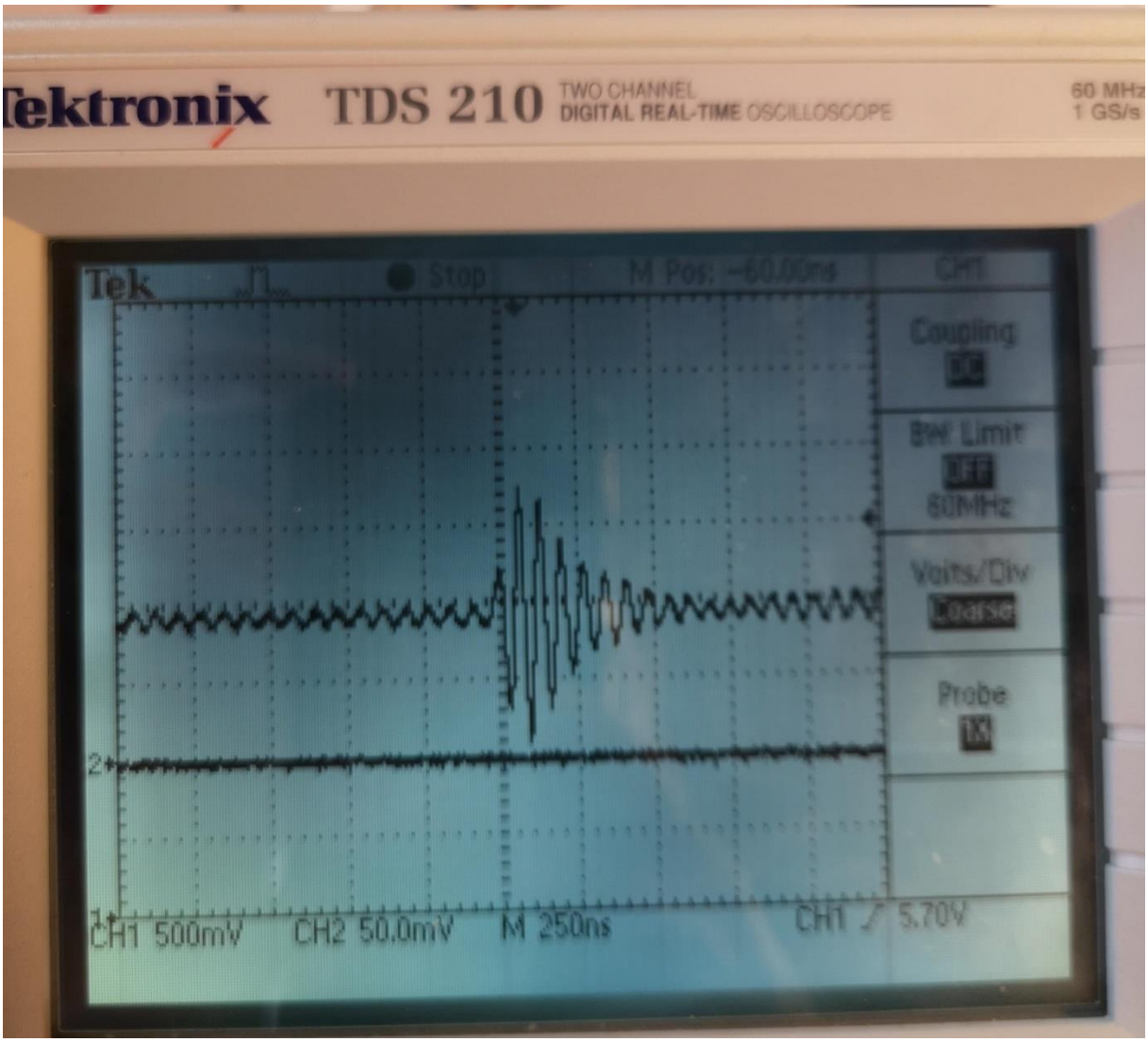


Figur 187: bildet viser det pulsmodulerte signalet som arduinoen leverer til servomotorene.

Siden det ikke var signalkilden som separat var problemet var det en mulighet at omformeren leverte ujevnheter i spenningssignalet som skapte «rykkingen» i servomotorene. Måten dette ble testet var ved å se på signalet som omformeren leverte ved moderat last. Altså last tilsvarende den som vi forventet at servomotorene skulle inneha. Dette ble da simulert med en middels stor motor som trakk 1,4 amper ved 5,7 volt. Spenningen som ble målt hadde en god del ujevnheter, men ikke nok til å skape så stor virkning på servomotorene.



Figur 188: Skjermbilde av grafen viser en endring på opp mot 0,5V i begge retninger.

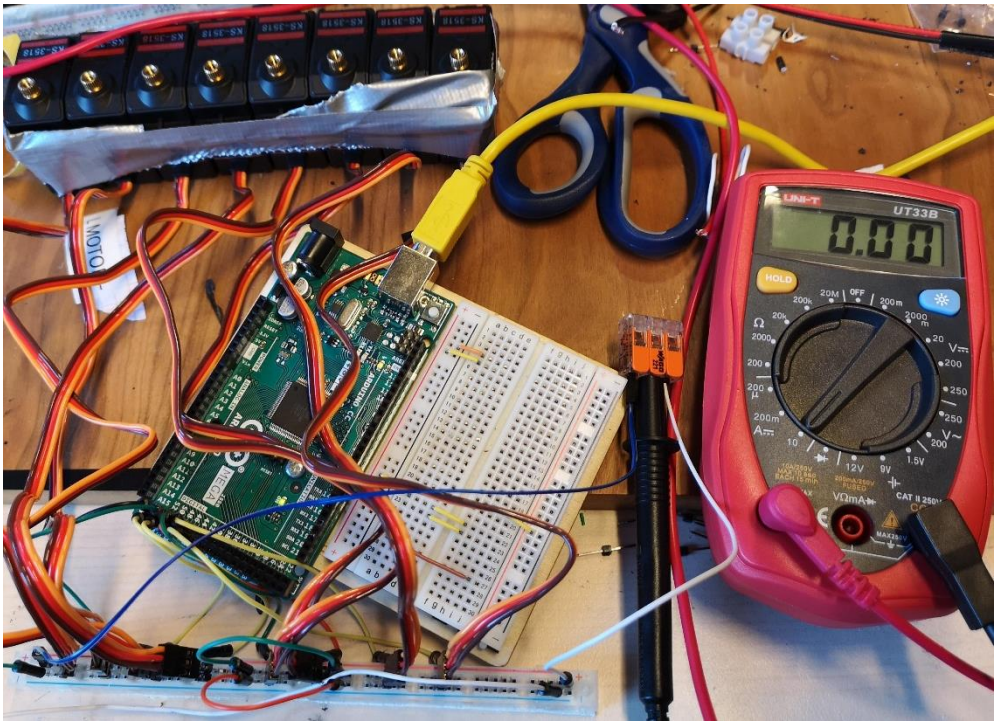


Figur 189: samme signal som figur 176, men forstørret mye. et kvadrat er 250ns.

Resultatet fra bilde to viser også at signalet som arduinoen leverer ikke deler karakteristikk med signalet sett i figur 177. Denne slutningen er fattet på bakgrunn av lengden på signalene som blir sendt ut fra både arduinoen og omformeren har for stor forskjell i både lengde og i amplitude slik at de ikke på noe som helst måte kan forveksles. Signalet fra omformeren er for høyfrekvent til at servomotoren skal kunne bruke signalet. Et signal som omformeren leverer vil være over 1 million ganger mer høyfrekvent enn det pulsmodulerte signalet fra arduinoen leverer og servomotoren klarer å lese.

Neste og siste på listen var dermed å se på servomotorene. Hypotesen bak denne delen av feilsøket var mulig at en av servomotorene hadde blitt skadet i fallet. Mye tydet på dette siden servomotorene trakk 0.1 amper da de sto i fastposisjon. Vi testet denne hypotesen og fant ut at de nye servomotorene ikke trakk noe strøm i fast posisjon. Servomotorene ble derfor koblet opp på samme vis som de tidligere servomotorene (i parallell) og koblet til

pulsmodulert signal fra arduinoen. Dette nye oppsettet trakk 1,7 amper ved bevegelse på alle servomotorene.



Figur 190: multimeteret viser at systemet ikke trekker noe strøm ved fast posisjon.

5.2.2 Utfordringer som følge av bestillingssystemet til skolen

I 1.3 nevnte vi bestillingssystemet til skolen som en begrensning for utviklingen av konseptet vårt. I løpet av utviklingen måtte vi bytte ut servoene mot noen som skulle være vanntette, samt å bytte ut nevnte Pi med en Jetson(ref 3.3). For Jetson trengte vi et nytt SD-kort som var stort nok, som måtte bestilles inn. Bestillingen ble levert i tidlig september men kortet kom ikke frem før mot slutten av oktober. På dette tidspunktet vurderte vi det som unødvendig å måtte vente på delen og gikk heller og bestilte det selv. Lignende scenarioer med små deler som koster under 1000 kroner kan løsningen vår fungere, men ikke når det kommer til de komponentene som er dyrere. Dette var en utfordring i forhold til servoene som brukte en måned på å bli levert til skolen, som igjen gjorde at vi ikke fikk testet systemet i vann som planlagt og måtte utsette. Fra vår side er en slik forsinkelse en ekstrem utfordring for utviklingen av konseptet vårt, med bakgrunn i hvor kort tid vi har avsatt til oppgaven. Skulle en annen av de store og dyrere komponentene ha røket i oktober eller senere, ville vi vurdert at oppgaven ikke kunne fullføres grunnet tregheten i bestillingssystemet til skolen.

Det kan argumenteres fra skolen sin side at man som ansvarlig for et prosjekt må bedre evne å forutse slike situasjoner, og dermed prosjektere for det når man bestiller varer. Men

dette er første prosjektet vi styrer og drifter noe som gjør oss lite kapabel til å beregne på forhånd hvilke deler og komponenter vi måtte bestille ekstra av. Samtidig er det et kostnadsspørsmål inne i bildet, der vi for eksempel ikke har penger i budsjettet til å kjøpe inn et ekstra batteri i tilfelle det ene skulle ryke. Vår vurdering er at bestillingssystemet har hemmet utviklingen av konseptet, og ført til at flere ønskede forbedringer til systemet måtte legges på is grunnet manglende tid.

5.2.3 Utfordring med varmgang inne i dronen

Vi har tre maskiner, der to av dem trekker mye strøm som igjen skaper mye varme. Der Pi kommer uten noe form for kjøleelement eller vifte, har derimot Jetson en varmeveksler som skiller varmen vekk ifra selve kortet. Men selv om vi har én maskin med varmeveksler og vi hadde montert vifte eller lignende i dronen ville det ikke hjulpet. Da dronen skal være tett, krever det også at det ikke slipper ut noe luft fra innsiden. Dette gjør selvsagt at vi aldri får kvittet oss med varmen slik systemet er designet. Fra starten av var vi klar over utfordringen og har alltid beregnet med at det må installeres/konstrueres en varmeveksler mot vannet eller en annen form for kjølesystem. Årsaken er at vi begrenser operasjonstiden ekstremt mye ved at det blir for varmt og dermed uforsvarlig å drifte dronen over tid.

På den andre siden er det fortsatt mulig å kjøre dronen i minst 30 minutter uten at komponentene/ledninger ser ut til å bli ødelagt. Batteriene har god nok levetid til å holde mer enn to timer, slik at det eneste som til nå har hindret oss fra videre drift over 30 minutt er at vi vil være på den sikre siden.

5.3 Hele systemet

Man har til nå i drøftingen sett på del-systemene og hvordan de er utviklet og fungerer. I 5.1.3 har man vist hvordan de fungerer sammen, der en har vist at den reagerer slik man ønsker men ikke nødvendigvis har forutsetningen for å kunne forflytte seg korrekt. I 5.1.4 har er det beskrevet mangelfull tetthet på skroget, som igjen gjør det vanskelig å drive vanntester i miljø der man kunne testet og sjekket opp forflytningsevnen bedre. Dette er en utfordring til tross for at siste test med gode servoer ga betydelig bedre indikasjoner. Til slutt har man i 5.2 beskrevet ulike begrensninger som gjør at tiden ikke gir mulighet for å fullføre de siste relevante testene man trenger.

Det gjør at det stilles en del spørsmål til dronens evne til å kunne tilfredsstille hovedmålet. På den ene siden har man tørrtestene som gir gode målinger på sensorene, der en har god nøyaktighet. Dette gjør at man i et landmiljø har en drone som ville fungert slik man ønsket. Med landmiljø menes at hvis man setter på hjul og styring ville dronen fungert godt i henhold til testene. På den andre siden har vanntestene vist at dronen ikke er helt tett og lekker inn vann over tid, selv om det ble mye bedre ved bytte av servoer. I vannoverflaten har den og

vist å ikke være god nok til å tilfredsstille alle kravene som igjen supplerer hovedmålet, da man mangler forflytningsevne sideveis. For å svare på om dronen fungerer optimalt som et hel-system må man få sjekket om dronen evner å forflytte seg når den er under vann. Da tettheten er en direkte utfordring og man på dette tidspunktet nærmet seg tidsfristen, skaper det en del utfordringer som tvinger oss til å ta et veivalg.

For å understreke, så vil dronens elektroniske komponenter «tåle» at det ligger litt vann i bunnen av dronen. Med å tåle menes at elektriske ledninger er isolerte og at elektroniske komponenter ligger plassert over bunnen av skroget, som ikke fører til at de kommer i direkte kontakt med vannet så lenge dronen ikke roteres. Men jo lengre man har dronen i vann, og spesielt når hele dronen kommer under vann vil mer kunne lekke inn. Samtidig vil de øverste servoene lokalisert nært elektroniske komponenter på innsiden kunne ta inn vann, som igjen ødelegger dem. Veivalget vi snakker om er en av to ting. Enten droppe å utføre undervannstester for dronen slik at den enda fungerer og heller stole på at tørrtestene gir godt nok svar. Det andre valget er å utføre testen og risikere at dronen tar inn vann og ikke lenger fungerer skikkelig. Der det første valget utgir ingen risiko slik at delene kan brukes videre og enda fungerer, vil valg nummer to risikere å ødelegge delene som igjen er vanskelig å anskaffe nye av. Valg nummer to vil også kunne ødelegge dronen helt, noe som ikke er ønskelig fra vår side.

En hybridløsning ble valgt som man kan se ut fra resultatene av nye posisjonstester i 4.4.4. Man valgte å gå bort fra videre tester under vann for at dronens komponenter ikke skulle risikere å bli ødelagt. Men det ble gjort nye tester av posisjonsmåleren i vannoverflaten med bedre lysforhold og ekstra objekter på bunnen til hjelp. Når man som vist i testene oppnår god nøyaktighet også der er det vår vurdering at del-konseptet har potensiale ved videre utvikling og gode komponenter.

Oppsummert vurderes det at dronen på den ene siden har del-systemer som kan i de rette forholdene levere nøyaktig sensordata slik at man kan oppnå hovedmålet. Men på den andre siden vil man ikke stole på tettheten til dronen, som igjen gjør at man ikke får testet ut vurderingen i praksis.

5.4 Gjennomgang av systemkrav

For å samle drøftingen er det nyttig å gå gjennom de ulike systemkravene man har satt i 2.1 som er basert på hovedmålet beskrevet i 1.2.

Første krav er at dronen må ha et system som gjør den kapabel til å oppdage objekter og bedømme avstanden til dem. I utviklingen og testingen har det vært mye fokus på at et slikt system skal brukes til objektunngåelse, som gjør det lett å glemme at det har videre

utviklingsmuligheter. Basert på testene vi har gjort og hva vi har erfart evner del-systemet å oppdage og bedømme avstand til objekter med god nøyaktighet, noe som gjør at den tilfredsstiller krav 1.

For krav nummer to trenger dronen å ha informasjon på hvordan sin egenrotasjon om alle tre aksene. Dette kravet er så og si universalt for alle droner på markedet. Til tross for store utfordringer med magnetometeret og støy som ga unøyaktighet på yaw, har man klart å få sensorene til å fungere med god nøyaktighet. Vi vurderer at dronen har god nok kontroll på egenrotasjon til at det tilfredsstiller krav 2.

Det tredje kravet var nok det som i utgangspunktet ville være vanskeligst å implementere sett i lys av at det ikke eksisterer en slik løsning tilgjengelig på nettet eller markedet. Kontroll på egen posisjon er det som i vår vurdering skiller vår drone mest ut ifra vanlige kommersielle droner. Det gir også enorme muligheter for videre utvikling av drone-konseptet, både her ved skolen og ute på markedet. Selv om vi forventet større unøyaktighet på målingene, ga systemet ekstremt nøyaktige resultater. Samtidig var det en utfordring med ujevnhet i kamera i forhold til bunn, som fort kan bli en hindring for videre bruk av dronen. Videre er testene gjennomført med klart syn til bunn, noe som ikke nødvendigvis er tilfellet for andre operasjonsområder. Til tross for ulempen, vurderer vi at kravet for posisjonsmåling er tilfredsstilt med bakgrunn i gode målinger og mulighetene for utvikling med mer penger vil gi bedre robusthet.

Det fjerde kravet er det kravet man vurderer som mest mangelfullt i forhold til å kunne si at hovedmålet er oppnådd. Årsaken ligger i at vi her var nødt til å avgrense mye for å komme i mål med å få testet systemet sammensatt. Men dronen har samtidig vist at den klarer å innhente informasjon gjennom fra nødvendige sensorer, bearbeide informasjonen og beregne rett fremgangsmåte. Den har også klart å styre og etterjustere for ytre påvirkninger, selv om det kun er testet i rolige og klare forhold. Samtidig er det ingen tvil om at systemet vårt er for enkelt for å kunne si at det matcher egenskapene til en operatørstyrt drone når det kommer til navigasjon. Det kreves en videre utvikling for at det skal kunne brukes i operasjoner og eventuelt sammenlignes med kommersielle droner. Men selv om den ikke når helt opp vurderer vi at den tilfredsstiller *deler* av kravet, herunder å kunne reagere raskt og effektivt på forstyrrelser og hinder.

Krav nummer fem er på mange måter nedprioritert sett i sammenheng med de andre del-komponentene og kravene. Grunnet manglende ventilering for dronen og mange komponenter som trekker mye strøm, har vi begrenset med operasjonstid. Selv om batteriene klarer å levere nok til et par timer, vil det ikke være trygt å bruke dronen lenger

enn 30 minutter grunnet varmgangen som følge av komponentene. Men det er også nok til å tilfredsstille kravet til operasjonstid, selv om det er langt ifra ideelt.

Vi har sett at dronen har utfordringer med å klare å benytte finnene når den er i overflaten, noe som gjør at den ikke evner å rotere om aksene. Det i seg selv tilfredsstiller ikke krav nummer seks. Basert på vurderingene i 5.3 kan man heller ikke verifisere at forflytningsdesignet vil evne å rotere dronen og gi den fremdrift i alle forhold. Det konkluderes med at man ikke har god nok tetthet i dronen til å svare på om krav seks er tilfredsstilt.

6. Konklusjon

Gjennom å ha oppsummert kravene i 5.4, kan man vurdere om hovedmålet er oppnådd som igjen kan svare på problemstillingen vår:

Hvordan kan man konstruere en autonom undervannsdrone med begrensede midler tilgjengelig?

For å summere har man evnet å lage sensorer som måler dronens posisjon, avstand til objekter og kontroll på egenrotasjon. Sammen med et styringssystem klarer dronen å samle inn de ulike datatypene og bearbeide og behandle dem for å navigere under vann. Samtidig har de fleste sensorene et utviklingspotensiale. Det er også krav fra 5.4 som ikke er fult ut oppnådd eller forblir ubesvart, som igjen er viktig for at vi skal kunne oppnå hovedmålet. Bakgrunnen er at målet sier at vi skal kunne manøvrere under vann med alle forutsetninger på lik linje med en kommersiell drone. Ut ifra det avgrensede designet til styringssystemet vil vi ikke fult oppnå en slik bred egenskap. Til tross for dette har oppgaven lagt til rette for at med videre utvikling er muligheten der for at oppgaven kan gjøre det, noe som svarer godt på delen av hovedmålet om base for videreutvikling.

Som et resultat kan problemstillingen besvares. Man *kan* lage en autonom undervannsdrone med komponenter tilgjengelig til relativt lave kostnader fra markedet. Dette skjer ved å benytte og samle ulike løsninger og forsøk fra nettet, samt å bruke egne kunnskaper. Slik vil dronen ha del-komponenter som gir nøyaktig nok data til at den klarer å unngå objekter og å navigere unna, samt holde kontroll på posisjonen, gitt de rette forholdene. Men samtidig trenger konseptet en videre utvikling for å kunne oppnå de samme egenskapene innenfor navigering og posisjonering som dagens kommersielle/militære droner.

Anbefaling til videre arbeid

Vi har vært tydelige gjennom utviklingen, drøftingen og her i konklusjonen at styringssystemet fungerer, men at det er avgrenset og har flere muligheter for å videreutvikles. Vår anbefaling til videre arbeid er å utvikle styringssystemet. Med utvikling

mener vi å utforske to muligheter: videreføre det nåværende styringssystem med flere kriterier og mulighetsrom, eller utforske muligheten for å implementere maskinlæring. Begge to vil nok kreve en bachelor-oppgave å gjennomføre, men redskapene er tilrettelagt ved bruk av komponentene vi har utviklet. En utvikling på en av disse to områdene, samt en oppgradering på visse komponenter mener vi vil gjøre dronen i stand til å ha de samme forutsetningene og egenskapene som kommersielle kablede droner.

Et videre utviklingsområde er å bytte ut skrog, finner og motor med en mer komplett pakke bestående av thrustere og mulighet for varmeutveksling. En slik løsning ville gjort dronen enda mer robust og tilrettelagt for flere muligheter og egenskaper for dronen, som å designe oppdragsspesifikke program med bruk av sensorene og komponentene vi har utviklet. Med den nye ordningen på skolen der MV-kadetter og MM-kadetter leverer bachelor samtidig, kunne denne videreutviklingen vært en fin tverrfaglig oppgave.

Bibliografi

Fra internett

1. Hvordan en gressklipper fungerer
   https://www.tek.no/nyheter/guide/i/704nMK/slik-fungerer-en-robotklipper
2. REMUS undervannsdrone – Kongsberg
   https://www.kongsberg.com/maritime/products/marine-robotics/autonomous-underwater-vehicles/AUV-remus-100/
3. Blueeye undervannsdrone – Blueeye Robotics
   https://www.blueye.no/?gclid=EAIaIQobChMI6JH9g8Ls5QIVQuaaCh00fw4tEAAYASAAEgJF0_D_BwE
4. Eksponentiell regresjon
   https://www.varsitytutors.com/hotmath/hotmath_help/topics/exponential-regression
5. Tracking ved hjelp av akselerometer
   https://x-io.co.uk/gait-tracking-with-x-imu/
6. Feature Matching
   https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html
7. Gyroskop forklart
   https://en.wikipedia.org/wiki/Gyroscope
8. Fusjonering mellom akselerometer og gyroskop
   http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1422&context=eesp
9. Hvorfor Akselerometer ikke kan måle yaw
   https://www.quora.com/Why-an-Accelerometer-cant-measure-Yaw
10. Tilt-kompensert magnetometer:
    https://www.instructables.com/id/Tilt-Compensated-Compass/
11. Kalkulator for utregning av motstand og effekt i LED-dioder
    http://led.linear1.org/1led.wiz

12. Servo effektbruk
    https://robotics.stackexchange.com/questions/16252/servo-motor-power-consumption-issue
13. PID-kontroller
    https://en.wikipedia.org/wiki/PID_controller
14. Jetson Nano vs Raspberry Pi spesifikasjoner
    https://www.tomshardware.com/news/jetson-nano-features-price,38856.html
15. I2C-bibliotek for Arduino
    https://www.arduino.cc/en/reference/wire
16. MPU9250 Register
    https://www.invensense.com/wp-content/uploads/2015/02/RM-MPU-9250A-00-v1.6.pdf
17. Konvertere GPS-koordinater til x- og y-format
    https://www.codeproject.com/Questions/626899/Converting-Latitude-And-Longitude-to-an-X-Y-Coordi
18. Bibliotek for BMP280 temperatur
    https://github.com/adafruit/Adafruit_BMP280_Library
19. Seriell-protokoll fra Arduino-forum
    https://forum.arduino.cc/index.php?topic=288234.0
20. Effektbruk Jetson Nano vs Raspberry Pi
    https://www.pidramble.com/wiki/benchmarks/power-consumption
21. Hente ut bilde fra kamera-feed for Jetson Nano
    https://devtalk.nvidia.com/default/topic/1061428/how-capture-video-from-camera-on-jetson-nano-in-h264-/
22. Hvordan restarte et Python-script når det avsluttes
    https://www.alexkras.com/how-to-restart-python-script-after-exception-and-run-it-forever/
23. Multithreading med Python
    https://www.tutorialspoint.com/python/python_multithreading.htm
24. Brute Force Matcher basics
    https://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html
25. Hamming distance forklart
    https://en.wikipedia.org/wiki/Hamming_distance

26. Matplotlib forklaringer og funksjoner
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.html
27. Automatisk oppstart av program på Raspberry Pi
https://www.raspberrypi.org/forums/viewtopic.php?t=66206
28. Hvordan I2C fungerer
https://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/
29. Pitch/Roll/Yaw
https://en.wikipedia.org/wiki/Aircraft_principal_axes

Rapporter og Artikler

1. Karras, George, Panagou, Dimitra J, Kyriakopolos, Kostas J(sept 2006)

Target-referenced Localization of an Underwater Vehicle using a Laser-based Vision System.

https://ieeexplore.ieee.org/abstract/document/4098908
2. Henriksen, Andreas Viggen (2016)

Camera-assisted Dynamic Positioning of ROVs

https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2501083
3. Winer, Kris(Aug 2017)
Simple and effective Magnetometer Calibration
https://github.com/kriswiner/MPU6050/wiki/Simple-and-Effective-Magnetometer-Calibration
4. Ozyagcilar, Talat(2015)
Implementing a Tilt-Compensated eCompass using Accelerometer and Magnetometer Sensors
https://cache.freescale.com/files/sensors/doc/app_note/AN4248.pdf
5. Even, R et al(1970)

Admittance matrix synthesis with RC common-ground networks and grounded finite-gain phase-inverting voltage amplifiers

6. Mallick, Satya(Juli 2018)
Find the Center of a Blob(Centroid) in OpenCV

https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/

7. Syed, Najam R

   Multithreading with OpenCV-Python to improve video processing performance

   https://nrsyed.com/2018/07/05/multithreading-with-opencv-python-to-improve-video-processing-performance/

Komponenter

1. MPU9250 + BMP280 10DOF
   https://www.banggood.com/MPU9250BMP280-10DOF-GY-91-Acceleration-Gyroscope-Compass-Nine-Shaft-Sensor-Module-For-Arduino-p-1100982.html?rmmds=myorder&cur_warehouse=CN
2. NEO 6-series GPS
   https://www.u-blox.com/en/product/neo-6-series
3. Ping Sonar Altimeter and Echosounder
   https://bluerobotics.com/store/sensors-sonars-cameras/sonar/ping-sonar-r2-rp/
4. Bar30 Dybdemåler
   https://bluerobotics.com/store/sensors-sonars-cameras/sensors/bar30-sensor-r1/
5. T200 thruster motor
   https://bluerobotics.com/store/thrusters/t100-t200-thrusters/t200-thruster/
6. HC-SR04
   https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/
7. KS-3518 vanntett servo
   https://www.banggood.com/KS-3518-Digital-Servo-20KG-Waterproof-180-Metal-Gear-Large-Torque-For-RC-Robot-Arm-p-1422906.html?cur_warehouse=CN
8. Arduino Mega 2560 mikrokontroller

https://www.robotshop.com/media/files/pdf/arduinomega2560datasheet.pdf-.

9.

Vedleggsoversikt

Vedlegg A – 3D-PRINT

Vedlegg B – Bilde av Komponenter

Vedlegg C – Programkode: Avstandsmåler Python (Jetson Nano)

Vedlegg D – Programkode: Posisjonsmåler og Styringssystem Python (Raspberry Pi)

Vedlegg E1 – Programkode: Styring av motorer/servo og Uthenting av Sensordata og HMI C#/C++ (Arduino)

Vedlegg E2 – Programkode: Egenlaget Bibliotek i Arduino for Sensordata C#/C++ (Arduino)

Vedlegg F1 – Ufiltrert Akselerometerdata med Fart (Excel-fil, leveres elektronisk)

Vedlegg F2 – Filtrert Akselerometerdata med Fart og Posisjon (Excel-fil, leveres elektronisk)

Vedlegg G – Konvertering av kraftmålinger til prosent for Arduino-kode

Vedlegg H – Kalibrering og beregning av algoritme for Posisjonsmåler

Vedlegg I – Kalibrering og beregning av algoritme for Avstandsmåler

Vedlegg J – Tester av Posisjonsmåler (Excel-fil, leveres elektroniskl)

Vedlegg K – Tester av Avstandsmåler (Excel-fil, leveres elektronisk)