University of Stavanger

Ådne Hult Karlson, Filip Byberg

---

*B*achelor

Styring av lineærmotor i ROS

15. mai 2021

Sammendrag

Robotteknologi og automasjon har forandret produksjon mye de siste 10 årene, og denne utviklingen forventes å fortsette de kommende årene. Styring av ulike motorer, blant annet lineærmotorer, er ofte en viktig del i en automatisert prosess. Denne rapporten inneholder styring av lineærmotor ved bruk av operativsystemet Robot Operating System (ROS). Det blir presentert styring av en virtuell robot modell samt en ekte robot modell. Oppgaven tar for seg metoder for bruk av ROS til styring, samt hvordan ROS fungerer. Gjennom ROS vil det bli kjent hva noder, *topic* (emne), *publisher* (publiserer) og *subscriber* (abonnent) er og hvordan dette brukes til å kommunisere i ROS nettverket.

Oppgaven er basert på et vippe system, der en lineærmotor fra produsenten LinMot, styrer et stag som skal utføre manipulasjoner på en ball, som ruller på en bane. Kinematikken i vippesystemet blir fremstilt på matematisk vis, ved å beskrive hvordan de forskjellige delene i roboten påvirker hverandre, og vippens påvirkning på ballen.

For den virtuelle delen av oppgaven blir metoden for robot konstruksjon og visualisering, presentert. Her brukes program som RViz og Gazebo. RViz brukes som et hjelpemiddel for konstruksjon av roboten, mens Gazebo som et simuleringsverktøy. Gjennom dette blir også konstruksjons format *Unified Robot Description Format* (URDF) satt i fokus. Her forklares også sammenhengen mellom *links* og *joints*, og hvordan de brukes under konstruksjon. For styring av den virkelige modellen brukes EtherCAT kommunikasjon, det blir da forklart hva EtherCAT er og hvordan det kan brukes til å kommunisere med LinMot driver, samt hvordan driver leser og skriver kommandoer.

Begge modellene reguleres og beveges, det blir da vist hvordan man kan bestemme bevegelsen av roboten, med bruk av Python og ROS pakken *dynamic_ rekonfigure*. Denne pakken skaper et grensesnitt, der en kan styre roboten i sanntid. Det blir presentert to metoder for regulering i form av PID regulering og prediksjon. For PID regulering, forklares metoden brukt og matematisk bakgrunn. For den prediktive reguleringen, blir det forklart hvordan en kan forutse ballens framtidige bevegelse og da stoppe den på et gitt punkt. Med dette blir de matematiske prinsippene forklart, samt de fysiske egneskapene som gjør at ballen påvirker beregningene.

Resultatene for PID viser gjennom testing, hvordan en kan komme frem til forskjellige K parametre som oppnår gode resultat. For den virkelige modellen blir det også vist hvordan forskjellige baller påvirker resultatet Her utføres tester for en pingpong ball, en golf ball og en baseball. Disse forsøkene sammenlignes, hvor det også blir utført polanalyse på ballene. Her forklares egenskaper som oversving, dempingsfaktor og den naturlige frekvensen i systemene.

For prediktiv regulering undersøkes forholdet mellom vippevinkel og den momentane farten til ballen. Ut ifra dette, vil motoren kunne bidra med å oppnå en kontrollert ballbevegelse.

Da original oppgavetekst omhandlet en hexapod, er noe forklaring om denne tatt med som tenkt fremtidig arbeid.

En link til video av vippe mens den kjører samt kode finnes her:
Kode: https://drive.google.com/drive/folders/1v7qL92X_N8nfJNUKPInefIi-NrbTHtd1?usp=sharing
Video: https://www.youtube.com/watch?v=pg_0FV3BZZA&ab_channel=%C3%85dneHultKarlson

Forkortelser

Forkortelse	Beskrivelse(Eng)	Beskrivelse(No)
DC	Direct Current	likestrøm
DOF	Degrees of freedom	frihetsgrader
DPRAM	Dual-Port-Random-access-memory	Dual Port Tilfeldig tilgangsminne
ESC	EtherCat Slave Controller	EtherCat slavekontroller
EtherCAT	Ethernet for Control Automation Technology	Ethernet for kontrollautomasjonsteknologi
FCS	Frame Check Sequence	Rammekontrollsekvens
FreeCAD	Free Computer-Aided Design	3D tegne program
HMS		Helse, Miljø og Sikkerhet
ID	Identity	identitet
idx	Index	Indeks
Kp	Proportional gain	proporsjonal pådrag
Ki	Integral gain	integrert pådrag
Kd	Derivative gain	derivert pådrag
LSB	Least Significant Bit	minst signifikant bit
MSB	Most Significant Bit	mest signifikant bit
netif	Network Interface	Nettverksgrensesnitt
OSI	Open Systems Interconnection	Samtrafikk med åpne systemer
PC	Personal Computer	Personlig datamaskin
PDI	Process Data Interface	Grensesnitt for prosessdata
PDO	Process data object	Behandle dataobjekt
PHY	Physical Layer	Fysisk lag
PID	Proportional Integral Derivative	Proporsjonalt integrert derivat
PLS	Programmable Logic Controller	programmerbar logisk kontroller
RAM	Random Access Memory	tilfeldig tilgangs minne
RJ45	Registered Jack-45	registrert Jack-45
ROS	Robot Operating System	Robots operativsystem
RPY	Roll, Pitch, Yaw	rulle, vippe, høyde
RViz	ROS Visualization	ROS-visualisering
RxPDO	Receive Process data object	Motta prosessdataobjekt
SDF	Simulation Description Format	Simuleringsbeskrivelse Format
SOEM	Simple Open EtherCAT Master	Enkel åpen EtherCAT Master
TxPDO	Transmit Process data object	Overfør prosessdataobjekt
URDF	Unified Robot Description Format	Enhetlig robot beskrivelses format
USB	Universal Serial Bus	Universell seriell Bus
VaiGoToPos	Velocity Acceleration Interpolator Go To Position	Hastighet akselerasjon interpolator gå til posisjon
wkc	Working Counter	Arbeidsteller
XML	Extensible Markup Language	Utvidbart markering språk
YAML	Yaml Ain't Markup Language	Yalm er ikke et Utvidbart markering språk
3D	Three-dimensional space	Tre dimensjonelt område
µC	Microcontroller	Mikrokontroller
	Joint	Ledd
	Link	Lenke
	Topic	Emne

Innhold

Forord

Slik vi ser det, trenger samfunnet i dag nye ingeniører som har evnen til å forandre verden. Studier innen ingeniørkunst har for oss alltid vært appellerende da dette gir tilgang til kunnskap, som åpner dørene for å uttrykke seg kreativt på utallige måter. Dette er hovedsaklig fordi emnene baserer seg på problemløsning og kreativ tenkning.

Robotteknologi er et fag som driver oss til å utforske muligheter, da fagets bruksområder potensielt kan implementeres på alle aspekter i et samfunn. Hvordan roboter av ulike slag påvirker menneskets teknologiske standpunkt, er noe som fascinerer oss og skaper motivasjon til å strebe etter en karriere innen robotfaget.

En annen drivende faktor som trigger vår interesse, er ønsket om å skape smarte selvoperative system som kan gjøre hverdagslige utfordringer enklere. En arbeidsplass som har rom for kreativ tenking, er viktig for å skape noe nytt. Dette er en kvalitet mange arbeidsplasser innen dette faget har.

Vi har begge fått mulighet innen tidligere jobberfaring og iløpet av studiet, til å få et innblikk i hva noe av det denne typer industrier har å tilby. Her har vi fått mulighet til å jobbe sammen med utdannede ingeniører, samt andre fagfolk. Å ha assosiasjoner med slike profesjonelle folk har intensivert vårt ønske om en fremtid innen teknologi.

Instituttet for Data og Elektro (IDE) har tidligere gitt oss mulighet for å jobbe med styring av en ABB robot arm. For oss var dette en utrolig opplevelse, da denne erfaringen betydelig utvidet horisonten for læring og kunnskap. Vi gleder oss over muligheten til å jobbe med dette prosjektet og er overbevist om at erfaringen kan utnyttes i fremtiden.

Denne rapporten er skrevet av Filip Byberg og Ådne Hult Karlson, som studerer til bachelor gjennom Y-vei, spesialisering innen elektronikkdesign og automasjon ved IDE. Rapporten bygges på relevant yrkesfagligutdanning, og skal gi et systemperspektiv på ingeniørfaget med fordypning innen robotteknikk/kybernetikk, programmering og reguleringssystemer.

Filip har jobbet som industrielektriker i bedriftene Rønning elektro AS og Solland AS, med underliggende fagbrev gruppe L. Han har i tillegg erfaring innen offshore transmitterovervåkning hos ConocoPhillips. Ådne har fagbrev som elektriker gruppe L og spesialisert seg innen bolig og service gjennom bedriften Sønnico AS.

Vi vil takke sentrale personer som har vært til assistanse, både under utvikling av system, veiledning og generell hjelp. Vi vil takke:

- **Morten Mossige**: For hjelp med kode og EtherCAT kommunikasjon, samt veiledning.
- **Ståle Freyer**: For lån av vippe lineærmotor system, samt veiledning.
- **Karl Skretting**: For god veiledning, samt tips til rapportskriving.
- **Finn Byberg** For lån av kontorplass hos *Rqm Safety AS*.
- **Tormod Drengstig** For hjelp med reguleringssystem.

1 Introduksjon og motivasjon

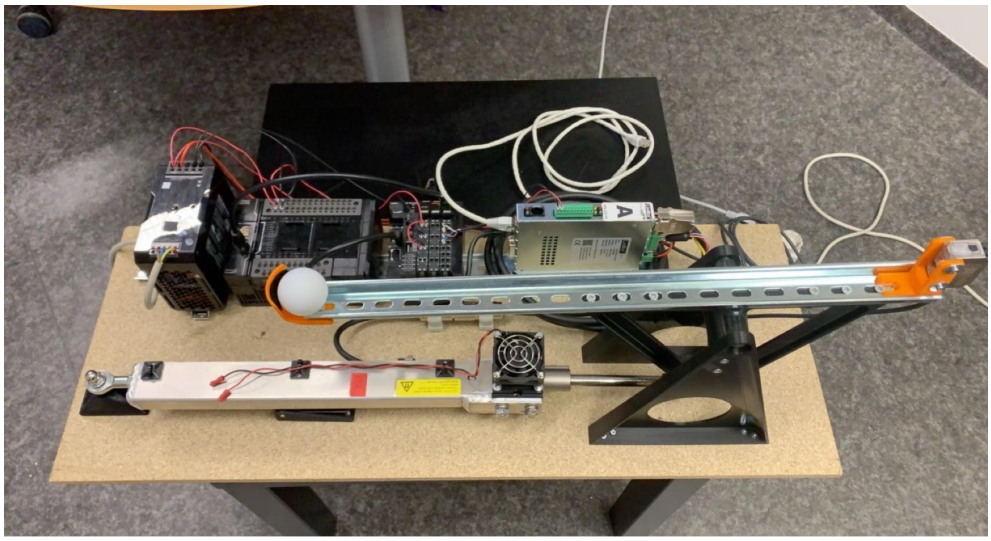
I denne oppgaven brukes *Robotic Operating System(ROS)* for å styre en lineærmotor festet til en vippe, slik at hellingsvinkel på banen kan endres. Hensikten er å kontrollere hvordan en ball, plassert på vippen, ruller på vippebanen. Det er ønsket å lage et godt kommunikasjonssystem via EtherCAT, slik at systemet kan videreutvikles til mer komplekse oppgaver.

Å kunne utvikle et system for styring av en enkel lineærmotor i ROS, har et stort potensialet for utvikling av kontrollerbare roboter. Utfordringen med dette er å lage et komplett system som enkelt kan utvides. Prosjektet skal sette et grunnlag for fremtidig arbeid med en hexapod, som har flere lineærmotorer.

Lineærmotorer er ofte brukt for å styre bevegelse i automatiseringssystemer. I automasjonsindustrien er presisjon en nøkkelfaktor for å optimalisere funksjonalitet. Ofte er det vanskelig å oppnå god presisjon, da et fysisk system som regel påvirkes av ytre feilbidrag. Av den grunn vil det være fokus på ulike feilbidrag som oppstår under testing av systemet og hvordan dette påvirker resultatet.

ROS er en plattform som blir mer og mer brukt, spesielt for enkle og rimelige system. Den store fordelen med ROS, er muligheten for å knytte sammen noder i et nettverk. Dermed kan kommunikasjonen mellom noder med ulike oppgaver, sammen skape et system med ønsket funksjonalitet. Et slikt system er svært gunstig for å lage et prosjekt som kan utvides, da funksjonalitet enkelt kan legges til, i form av noder.

ROS åpner dører for å uttrykke seg kreativt og gir brukeren mulighet til å realisere omtrent et hvert robotprosjekt. Da ROS tillater brukeren å opprette et grensesnitt for styring av fysiske motordrevne system, skaper dette et mangfold av muligheter for robotmanipulering og kontroll system. Det er ønskelig å oppnå en kontrollert styring av vippemodellen som er vist i figur 1.1



Figur 1.1: Oppsett for vippemodell brukt i oppgaven

1.1 Original oppgavebeskrivelse fra november 2020

Dette prosjektet er en bacheloroppgave med oppgavetittel "*Robot Operating System (ROS) modellering og visualisering av hexapod og vippesystem*". Prosjektet skal deles inn i to deler, hvor målet for del 1, er å oppnå kommunikasjon mellom ROS og ekstern driver, samt lage et grunnlag for simulering og visualisering av vippesystem i ROS. Del 2 vil bygge på del 1, da seks av samme driver skal brukes og prinsippet holdes det samme, men mer kompleks.

Hentet fra oppgavebeskrivelse på Studentekspedisjonen:

> *I denne oppgaven ønsker vi at studentene skal sette seg inn i ROS og modellering, utvikling og visualiseringsverktøy. Oppgaven er å utvikle modell av vår (demonterte) hexapod robot, og styre og visualisere denne modellen. I tillegg til ROS kan RViz, Gazebo og MoveIt være nyttige verktøy. Vi har fra tidligere et program i Matlab som visualiserer hexapod og der framover og invers kinematikk er løst.*

Oppgaven skal inneholde i del 1:

- Utviklet grafisk modell av lineær motor
- Styring og visualisering av grafisk modell
- Implementering av ROS-kommunikasjon med driver.

Oppgaveinnhold del 2:

- Utviklet grafisk modell av hexapod
- Styring og visualisering av hexapod
- Implementering av ROS-kommunikasjon med drivere og lage testprogram for lineærmotor

1.2 Justert oppgavebeskrivelse

Etter veiledningsmøte mellom gruppemedlemmene og veiledere, ble det mandag 22.03.21, tatt en avgjørelse av fagansvarlige, om at oppgave beskrivelse skulle endres. Dette var på grunn av at veileder ønsket mer fokus på regulering og styring av lineær motor. Dette medførte at hexapod delen falt bort fra oppgaveteksten, for å få mer fokus på vippesystemet. Dette valget ble tatt da fagansvarlig mente arbeid med vippesystem var mer enn tilstrekkelig.

Oppgavebeskrivelse ble da som følger:

> *I denne oppgaven ønsker vi at studentene skal sette seg inn i ROS og modellering, utvikling og visualiseringsverktøy. Oppgaven er å utvikle modell av vårt vippesystem, og styre og visualisere denne modellen. I tillegg til ROS kan RViz, Gazebo og MoveIt være nyttige verktøy. Det er ønskelig at studentene lager et testprogram med regulering for styring av modell. For så å opprette et grunnlag for testing av systemet via EtherCAT.*

Tema for oppgaven, hentet fra Studentekspedisjonen:

> *Robot Operating System (ROS) er et rammeverk for utvikling av programmer for roboter. Særlig for småskalarobotisering er det, det foretrukne verktøy. UiS har tidligere hatt flere oppgaver der ROS har blitt brukt, og vi har også ROS i en Spurv", en selvkjørende liten bil fra KVS.*

Oppgaveinnhold:

- Utvikle grafisk modell av lineær motor
- Styre og visualisere grafisk modell
- Implementere EtherCAT-kommunikasjon med drivere og lage testprogram for vippesystem
- Lage et velfungerende reguleringssystem for å oppnå gode resultater
- Fremtidig arbeid for hexapod

Del I

Innledning

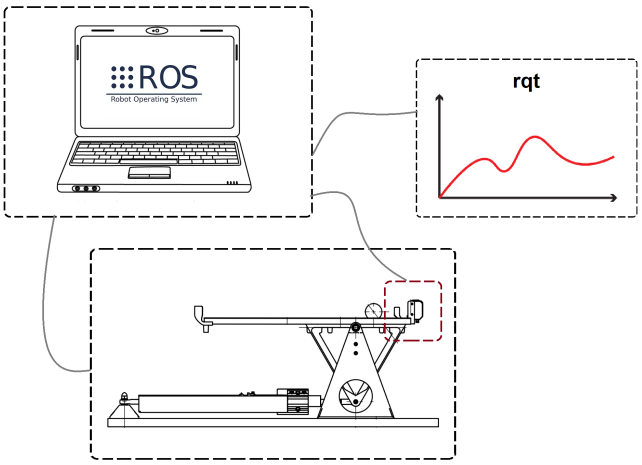
I dette kapittelet vil den viktigste bakgrunnskunnskapen presenteres. Her blir det satt fokus på ting gruppen ikke har gjort selv, men har brukt som hjelpemidler og utgangspunkt for å løse oppgaven.

2 Bakgrunn

Denne delen av oppgaven beskriver en mekanisme, som består av en bane plassert på et roterende ledd i et vippesystem. Vippesystemet styres av en motor med lineær forskyvning. En ball er plassert på banen og vil rulle en bestemt retning langs banen, ved endring av vippevinkel. Ballens posisjon vil dermed kunne manipuleres ved hjelp av motoren i samspill med en avstandssensor for posisjonsrapportering. Det er ønskelig å lage en virtuell simulering av systemet før det utføres tester på vippemodellen. Dette vil hjelpe med å etablere en god forståelse om hvordan responsen til de fysiske komponentene i systemet fungerer og opprette et grunnlag for testregulering. Samtidig gir dette mulighet til å utforske ulikheter mellom ideelt system og realistisk system.

Prosjektets hovedfokus vil ligge på å opprette god kommunikasjon mellom ROS og eksterne drivere, samtidig som det blir satt søkelys på hvordan man oppnår god styring i systemet. I denne sammenheng blir ulike metoder for regulering satt på prøve. Her må alle komponenter, samt programvare i prosjektet samspille, for å oppnå best mulige resultater.

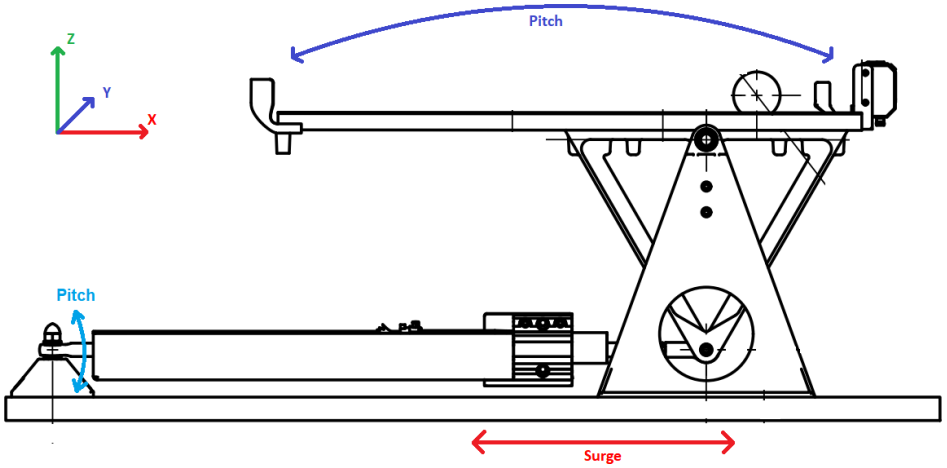
Systemet skal ha mulighet for en sanntids parameteroppdatering, slik at innstillinger og verdier kan endres under drift. I tillegg kunne loggføre ulike statusverdier til senere analyse.



Figur 2.1: ROS knyttet til vippesystem, avstandssensor og sanntidsplot. (ref. Ståle Freyer)

2.1 Definisjon av bevegelse

Oppsettet for vippesystemet er konstruert med et overgangsledd, som transformerer en "*surge*" forflytning, til en "*pitch*" bevegelse. Fleksibel bevegelse for motoren i vertikal retning, blir kompensert for, ved bruk av et vippeledd på enden av motoren.

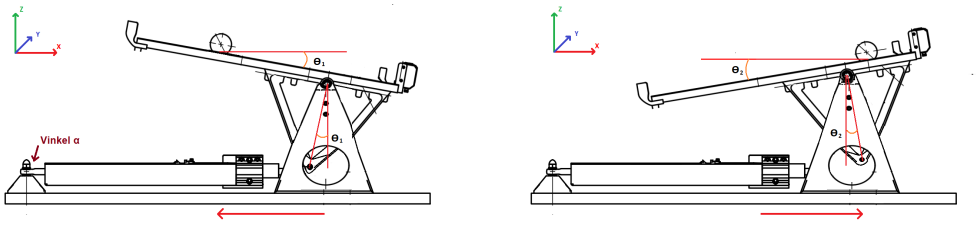


Figur 2.2: Bevegelse i vippesystem (ref. Ståle Freyer)

En "*surge*" bevegelse på motorstaget følges av en "*pitch*" bevegelse på banen. Dette resulterer i en endring av banens vinkelposisjon. Som årsak av vinkelendring, vil banen fungere som et skråplan og ballen vil rulle i retningen som korresponderer med vinkelposisjonen. Vippeleddet som motoren er festet til, vil tillate mer fleksibel bevegelse i z-aksen for festepunktet mellom motorstag og vippe, som vist med lyseblå linje, i figur 2.2.

2.2 Motorens påvirkning

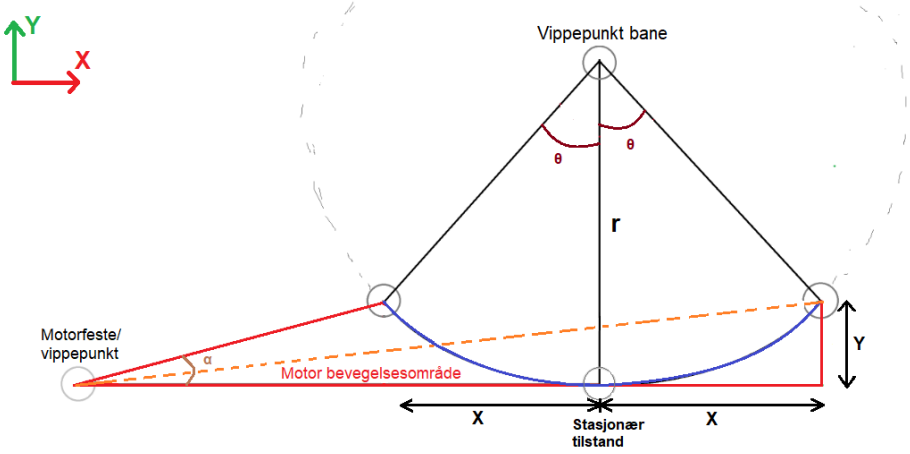
Lineærmotorens funksjon i systemet er å gi pådrag, med hensikt i å stabilisere ballen på et valgt settpunkt. Motorens forskyvning vil utgjøre en vinkel θ på banen. Vinkel θ vil avgjøre ballens akselerasjon, og hvilken retning den vil trille. En gitt "*surge*" bevegelse fra motoren gir et korresponderende utslag på banevinkel θ.



Figur 2.3: Motorens påvirkning av banevinkel. (ref. Ståle Freyer)

Den geometriske sammenhengen i festepunkt mellom motorstag og vippe kan beskrives med en semisirkulær bevegelse, vist med blå linje i figur 2.4. Dermed kan festepunktets posisjonsverdi beskrives ut ifra funksjonen til en sirkel: Hentet fra: [s.38, ref. [1]]

$$r^2 = (x-a)^2 + (y-b)^2 \qquad r \rightarrow Sirkelradius \qquad a\ og\ b \rightarrow Forskyvning \tag{2.1}$$



Figur 2.4: Geometrisk sammenheng mellom bevegelige ledd

For å få stasjonær tilstand av sirkelbevegelsen i origo, settes sirkelens forskyving i y-akse, a = r og sirkelforskyvning i x-akse, b = 0. Ut fra dette, vil funksjonen 2.2, beskrive festepunktets bevegelse med hensyn til x.

$$Y(x) = r - \sqrt{r^2 - x^2} \tag{2.2}$$

Ved harmoni mellom motorvinkel α og stagforskyvning fra motor, vil denne bevegelsen kunne realiseres. Sirkel radiusen r, vil i dette tilfelle representere avstand mellom vippepunkt på bane og motorstagets festepunkt. Dette blir brukt som et utgangspunkt for videre utregninger.

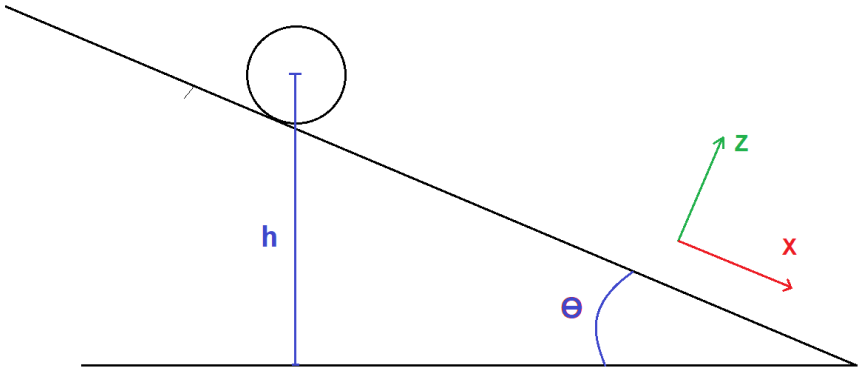
2.3 Ballens bevegelse

Fra Galileo Galileis bevegelseslære i boken «*Samtale om de to nye vitenskaper*» [2], kan man finne læren om rullende kuler på skråplan. I dette tilfelle kan skråplanet ha en varierende vinkel. Den kinetiske energien til et objekt som ruller, kan beskrives med tyngdepunktsbevegelsen og rotasjonsenergien, som vist i ligning 2.3. [kap. 9.4, ref. [3]]

$$K = \frac{1}{2}mv^2 + \frac{1}{2}I_0\omega^2 = \frac{1}{2}(1+c)mv^2 \tag{2.3}$$

$\omega \longrightarrow Vinkelhastighet, \quad c \longrightarrow Rulle\ friksjon\ koeffesient, \quad I_0 = cmr^2 \longrightarrow Treghetsmoment$

Rulle friksjons koeffesienten indikerer hvor stor motstand det skapes for en gitt normalkraft, mellom ballen og overflaten som ballen ruller på. Mer informasjon om dette finnes: [4].



Figur 2.5: Ball på skråplan

Når ballen ruller langs et skråplan med helningsvinkel θ og beveger seg en avstand $h = xsin(\theta)$ i tyngdefeltet, en lengde x langs banen vist i figur 2.5, gis bevaringen av mekanisk energi.

$$mgxsin(\theta) = \frac{1}{2}(1+c)mv^2 \tag{2.4}$$

Ut ifra denne ligningen er det mulig å finne generell formel for hastighetsøkning som funksjon av distansen vist i ligning 2.5. For så å løse differensialligningen og finne hastigheten som funksjon av tiden vist i ligning 2.6, samt finne formel for akselerasjon.

$$v(x) = \sqrt{\frac{2gxsin(\theta)}{(1+c)}} = \frac{dx}{dt} \tag{2.5}$$

$$v(t) = \frac{gsin(\theta)}{2(1+c)} \cdot t^2 = \frac{1}{2}at^2 \tag{2.6}$$

$$a = \frac{gsin(\theta)}{1+c} = \frac{dv}{dt} \tag{2.7}$$

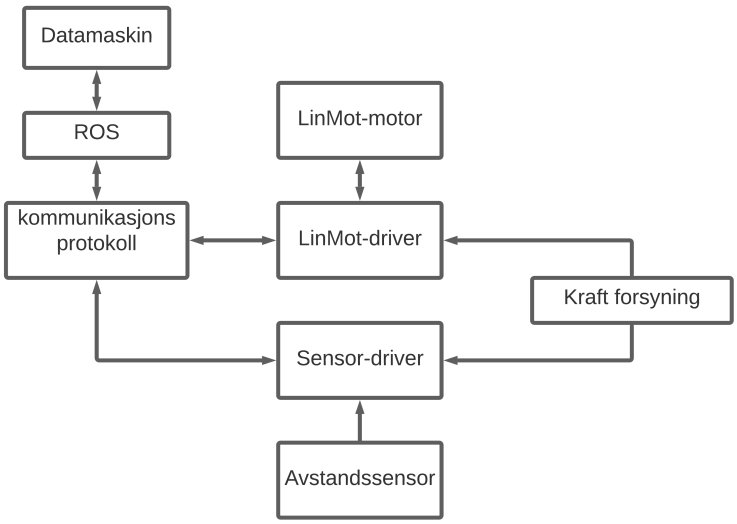
Senere vil formel for hastighet i ligning 2.5, brukes til videre utledning, mens ligning 2.6 og 2.7 brukes av gruppen for å få en bedre forståelse av bevegelsen. Videre kan sammenhengen mellom ballens fart og vinkel θ beskrives med ligning 2.8. Denne ligningen brukes senere i oppgaven.

$$\theta = \arcsin(\frac{V^2 \cdot (1+c)}{2 \cdot g \cdot x}) \tag{2.8}$$

Vær oppmerksom på at distansen x her, ikke er samme som forklares i kapttel 2.2.

2.4 Utstyr

Alt utstyr var ferdigmontert på vippemodellen, og dermed var innkjøp, montering og valg av utstyr ikke nødvendig. Det er mulighet for PLS-styring av systemet, men siden oppgaven omhandler ROS, falt valget på direkte styring fra PC via EtherCAT.



Figur 2.6: Kommunikasjonslinjer i systemet

Figur 2.6 er en oversikt over kommunikasjonslinjer, samt signalene i systemet. Det vil bli brukt samme strømtilførsel til begge driverene koplet i parallell.

2.4.1 Motor til vippesystemet

Intitutt for Data og Elektro(IDE) har kjøpt lineære motorer av typen *PS01-23x80F-HP-R20* fra LinMot [5]. Motoren har som oppgave å styre all bevegelse i systemet.

Motoren styres av en LinMot-driver av typen *1150-EC-XC-0S* [6]. Denne har mulighet til å motta og sende informasjon under drift, som vil være gunstig da systemet skal kjøres via sanntidskommunikasjon.

Viktige egenskaper til motoren, relevant til realisering av oppgaven:

Min lengde	Maks lengde	Maks hastighet	Maks kraft	Presisjon
0mm	90mm	7.3m/s	67N	±0.05mm

2.4.2 Sensor på vippen

Sensorens oppgave er å rapportere ballens posisjon til en hver tid. Denne er plassert på enden av banen, slik at systemet kan tilpasse motorverdiene ut ifra målt posisjon. Sensoren er av type *E3AS-HL500M*[7], med en tilhørende driver av type *NXECC202*[8]. Viktige egenskaper til avstandssensor, relevant for realisering av oppgaven:

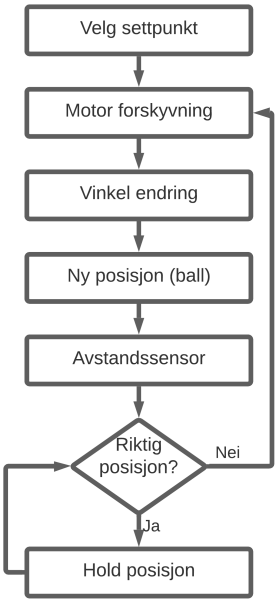
Min rekkevidde på vippe	Maks rekkevidde på vippe	Presisjon	Maks antall målinger pr sekund
35mm	500mm	1mm	660

2.4.3 Kraft forsyning

Kraft forsyning til systemet er kablet fra en *Omron S8VK-S12024* [9], med en utgangsspenning til drivere på 24V DC. Dette medfølger at man ikke trenger høyere kapslingsgrad, da *IDE* godkjenner bruk av utstyr på laboratorium, som leverer under 50V.

2.4.4 Hvordan styre systemet?

Det er tenkt å bruke sanntidskommunikasjon mellom ROS og drivere, via EtherCAT. Kommunikasjonsprotokollen vil sendes over en 8-pin *CAT5* kabel. Et forenklet funksjonsdiagram er gitt under for å gi en oversikt over basis funksjonaliteten til vippesystemet uten regulator.



Figur 2.7: Forenklet funksjonsskjema av vippesystem.

Videre vil dette bli brukt som utgangspunkt før implementering av reguleringssystem.

2.4.5 Utstyrsliste

Navn	Type	Antall	Beskrivelse
LinMot driver	1150-EC-XC-0S	1	Driver til motor [6]
Lineær motor	PS01-23x80F-HP-R20	1	Motor til å styre vippen [5]
Power supply Omron	S8VK-S12024	2	Kraft forsyning [9]
Sensor driver	NXECC202	1	Driver til sensor [8]
Sensor	E3AS-HL500M	1	Avstandssensor [7]
Kabel	CAT5e	2	Kommunikasjonskabel til driver
USB/VGA	RS232	1	Overgang fra USB til VGA
Kabel	K05-Y/R-2	1	Tilførsel og kommunikasjon fra driver til motor
Tilførselskabel	3x1.5 PFSP	1	Tilførselskabel til kraft forsyning
Vippesystem	–	1	Vippesystem lånt av Ståle Freyer

2.5 Robot Operating System

ROS er et open-source robotoperativsystem, skapt for å forenkle konstruksjonen av kompleks robot design og funksjonalitet. ROS gir brukeren tilgang til virtuelle bibliotek og verktøy, for å tilpasse og spesifisere robot programvare. Mer informasjon om ROS finnes på deres hjemmeside [10].

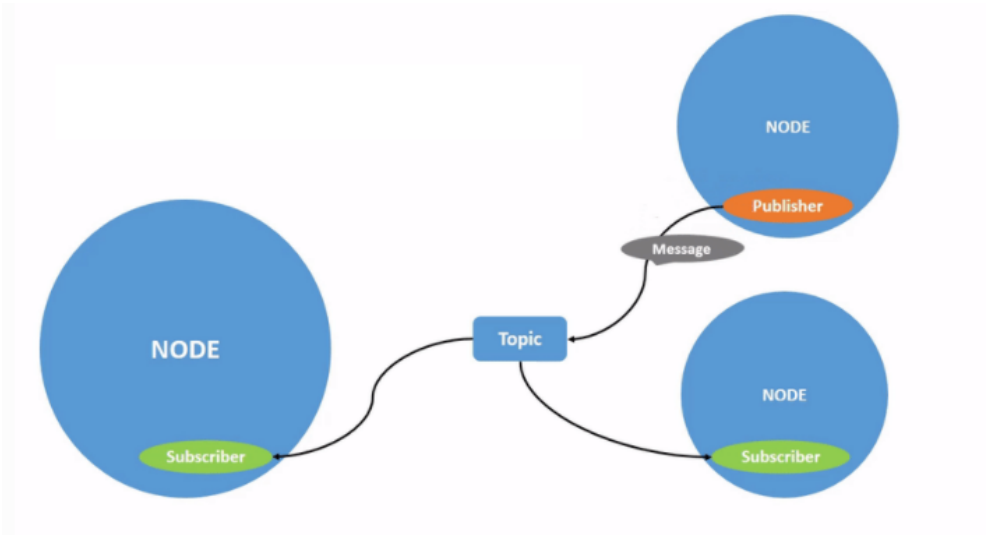
Det finnes ulike varianter av ROS med ulik funksjonalitet. Valg av variant brukt til å realisere hvert prosjekt, ble tatt på grunnlag av variantens brukervennlighet i forhold til hensikt.

- **ROS Melodic** - Grafisk visualisering og simulering
- **ROS Neotic** - Styring over EtherCAT-kommunikasjon

Det er mulig å finne informasjon om de forskjellige ROS versjonene på deres forum: [11].

2.5.1 Robot Operating System funksjonalitet

For å forstå hvordan ROS fungerer, er det viktig å sette seg inn i hvordan data blir fraktet mellom noder. Flere noder utgjør sammen et ROS nettverk. Kommunikasjonen i et ROS nettverk består hovedsakelig av ulike noder som frakter informasjon via *topics*. Hver node har muligheten til og hente informasjon (*subscriber*) og publisere informasjon(*publisher*).



Figur 2.8: Node komunikasjon (ref:[12])

Det finnes ulike metoder å opprette noder og *topics*. Flere ferdigdefinerte noder kan hentes og defineres fra pakker i ROS sitt virtuelle bibliotek. Det er også mulighet for å spesiallage noder og *topics* i Python, C++ osv. med funksjon etter behov.

For å realisere et system vist i figur 2.7, ved bruk av ROS funksjonalitet, vil det være viktig å spesifisere noder som inneholder de nødvendige posisjonsverdiene, samt funksjonalitet. I tillegg må tilstrekkelig med *topics* defineres. Dette sørger for at kommunikasjonen mellom de ulike nodene realiseres.

2.5.2 Hvordan starte et nytt prosjekt?

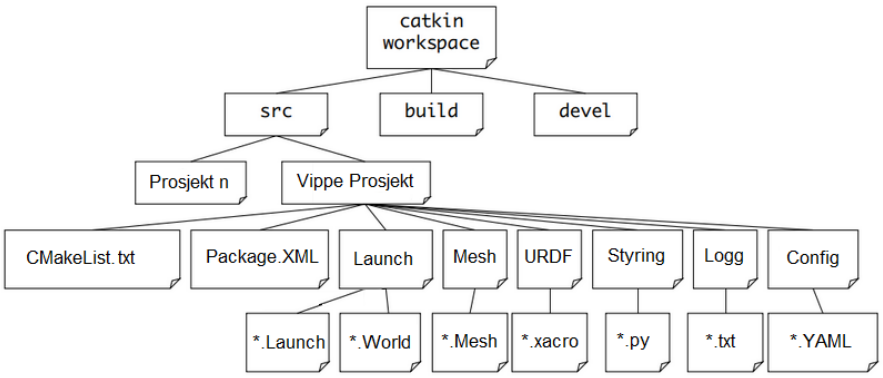
Et nytt prosjekt i ROS, oppretter en virtuell arbeidsplass for robot manipulering. Alle prosjektene som lages i oppgaven, er gjort i et eget *Catkin workspace*. Dette er nødvendig for å lage prosjektet i *Unified Robot Description Format (URDF)*, som er standardformat i ROS for visualisering og bygging av robotstrukturer. Et *Catkin workspace* skaper en avgrenset arbeidsplass i form av et virtuelt bibliotek, hvor brukeren har mulighet til å opprette flere prosjekter og pakkesystem.

Som forklart i kapittel 2.5, skal det lages to prosjekter med ulik hensikt, i to forskjellige varianter av ROS. Dermed er det viktig å holde disse i adskilte *workspace*. Organisering av filer i ROS og tilhørende pakker, samt oppretting av et *Catkin workspace* for et prosjekt, følges etter standard prosedyre og er beskrevet i [14].

Det ble installerte følgende pakker:

- ***dynamic_reconfigure*:** Tillater sanntids parameter oppdatering til en node [15]
- ***rosgraph_msgs*:** Logging av grafer i ROS [16]
- ***rospy*:** Oppretter et grensesnitt mellom Python og ROS [17]
- ***std_msgs*:** Muliggjør bruk av ulike meldingstyper i ROS [18]
- ***urdf*:** XML spesifikasjoner for robot modeller [13]
- ***controller_manager*:** Muliggjør sanntids-kontroller for robotmekanismer [19]
- ***joint_state_controller*:** Publiserer informasjon om hvert ledd i roboten [20]
- ***robot_state_publisher*:** Publiserer robotstatus [21]

Det vil være mulighet for å legge til pakker ved bruk av *CMakeList.txt* og *Package.XML*, om nødvendig. En oversikt over opprettet *Catkin workspace*, er vist i figur 2.9. Dette blir brukt for simulering og visualisering i ROS Melodic.



Figur 2.9: *Workspace* i ROS Melodic

Catkin workspace for styring over EtherCAT i ROS Neotic, vil ha en annerledes og mer komplisert oppbygning. Oppbygning og innholdet i Neotic *workspace* finnes i link til prosjektet.

2.5.3 MoveIt

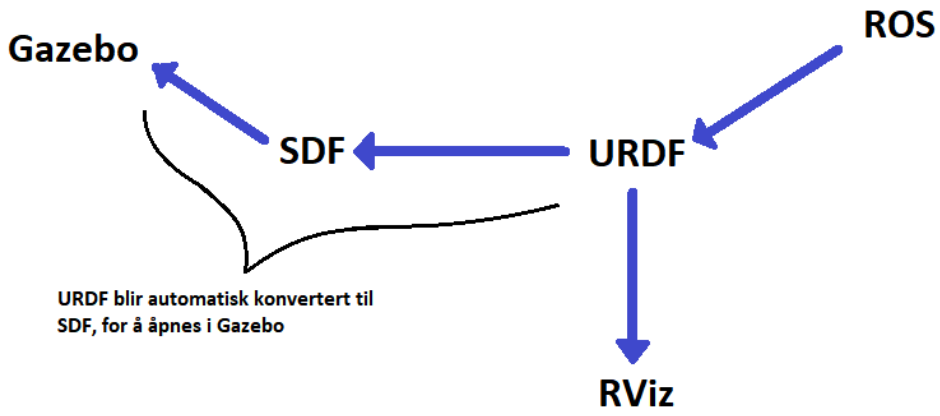
For å starte robot manipulasjon i et nytt prosjekt, ble det praktiske verktøyet MoveIt brukt. MoveIt gir brukeren blant annet mulighet til å enkelt konfigurere robotformat i et ROS nettverk. Info om MoveIt bruksområde og installasjonsveiledning finnes i MoveIt dokumentasjon: [22].

MoveIt fungerer i samspill med robotformatet URDF. Dette tillater brukeren å importere grafiske 3D-objekt, for å bygge komplekse roboter i visualiseringsprogrammet RViz. Videre kan man utføre planlagte bevegelser, som skaper et visuelt grunnlag for modellen. Dette forklares videre i kapittel 3. En liste av programmer brukt for å realisere et komplett system i *workspace*, kompatibelt med MoveIt er oppgitt i punktlisten:

- **ROS-melodic-Moveit**: Robot manipulator
- **Python**: Programmeringsspråk
- **Catkin**: ROS pakke manipulator
- **Visual Studio**: Filmanipulerings program
- **Chocolatey**: Windows pakke manipulator [23]

`Installasjonsveiledning til pakker brukt av gruppemedlemmene:`[24]

I denne oppgaven vil MoveIt sin funksjonalitet brukes til visualiseringsdelen. Selve simuleringen vil bli utført i simuleringsprogrammet Gazebo. Dette blir videre forklart i kapittel 4.

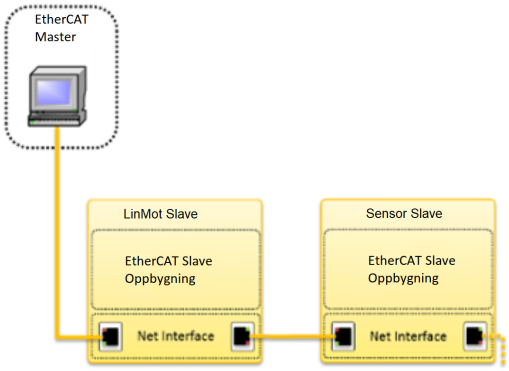


Figur 2.10: Lesing av URDF til visualisering eller simulering

Figur 2.10 viser hvordan de ulike programmene henter informasjon om modellen i URDF, fra *workspace* i ROS. Se også figur 2.9. I *workspace* vil en URDF fil defineres, med robotbeskrivelser. Videre kan man importere robotbeskrivelsen til enten Gazebo eller RViz, avhengig om man ønsker å simulere eller visualisere modellen.

2.6 EtherCAT

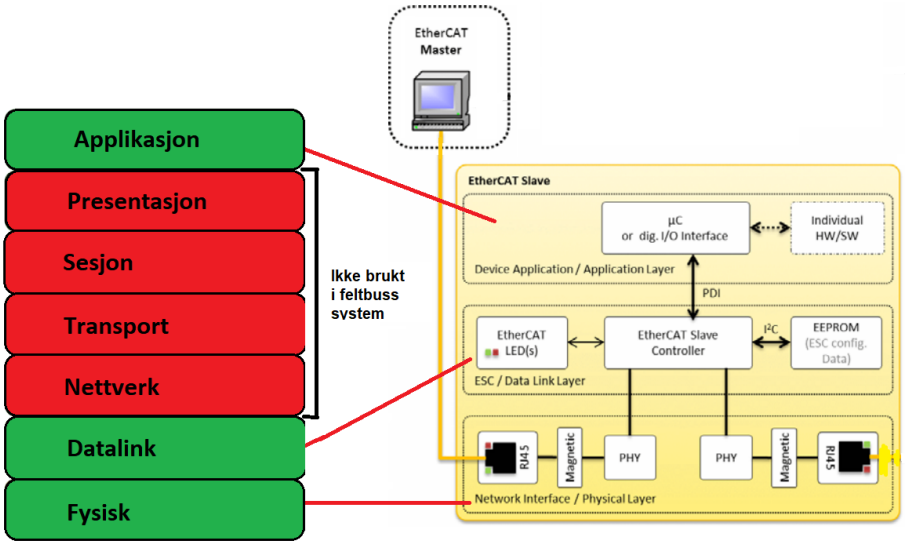
EtherCAT er et Ethernet-basert databussystem, utviklet av *Beckoff Automation*. I dette prosjektet brukes EtherCAT kommunikasjon, fordi kommunikasjonsprotokollen er svært gunstig for sanntidskommunikasjon innen automasjonsteknologi. Modellen for EtherCAT-kommunikasjon i vippesystemet, mellom master og slave er vist i figur 2.11.



Figur 2.11: EtherCAT generell modell (ref:[25], s.9)

Fordelen med EtherCAT, er at master vil sende data gjennom hver slave i systemet, samt samle all data "On the fly", før informasjonen blir returnert tilbake til master. Dette er noe som krever lav båndbredde i komplekse systemer og vil derfor være en god løsning, dersom prosjektet skal utvides i senere tid.

EtherCAT bruker *Open Systems Interconnection* (OSI)-nettverkmodellen. Denne viser hvordan man deler inn og bygger opp komunikasjonsprotokoller. Ved å «hoppe over» bruk av OSI lag 3-6 under sanntids kommunikasjon, vist i figur 2.14, oppnår systemet drastisk raskere syklustider og komunikasjonshastighet. Informasjon om OSI-modellen til EtherCAT ved sanntidskommunikasjon finnes: [26]



Figur 2.12: EtherCAT OSI (ref:[25], s.10)

Fysisk del

Den fysiske delen er et nettverksgrensesnitt som behandler feltbussignaler. Nettverksdata fra master kommer inn via standard *Registered Jack-45* (RJ45) innganger. Dataen blir konvertert til binære signaler, for så å sendes til *EtherCAT Slave Controller* (ESC), for håndtering. Se figur 2.12.

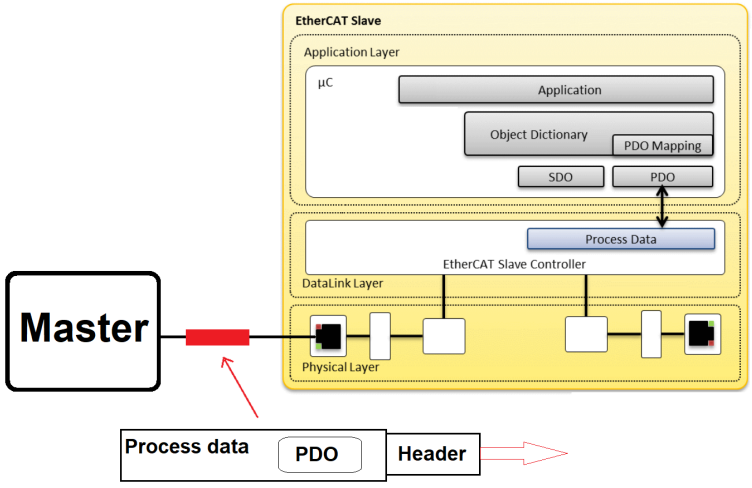
Data link del

I data link delen, vil ESC være hovedkomponenten som håndterer EtherCAT protokollen i sanntid. Data blir behandlet "On the fly", og det skapes et prosess data grensesnitt (PDI), for datautveksling mellom EtherCAT nettverk og slavens lokale applikasjonskontroller. Prosess data er utvekslet via en *Dual Ported Random Access Memory* (DPRAM) i ESC. Dette tillater grensesnittet å utføre flere datautvekslinger samtidig.

Applikasjons del

Applikasjon delen inneholder en micro kontroller (µC) som kommuniserer med ESC via PDI. µC har som oppgave å utveksle data med slave applikasjonen. Dette innebærer å kontrollere slave enheten som i dette tilfellet er lineærmotoren, med data sendt fra master og behandlet i ESC. Samt rapportere applikasjonsstatus tilbake til ESC. µC leser og skriver data i form av prosess data objekt (PDO), fra/til ESC. Input data i µC, måles opp mot slavens objekt ordbok og utfører kommandoer, basert på objekt ordbokens beskrivelse. LinMot driver objekt ordbok finnes i LinMot dokumantasjon: [27].

Nærmere beskrivelse av EtherCAT nettverk med fokus på slave arkitektur finnes i: [25].



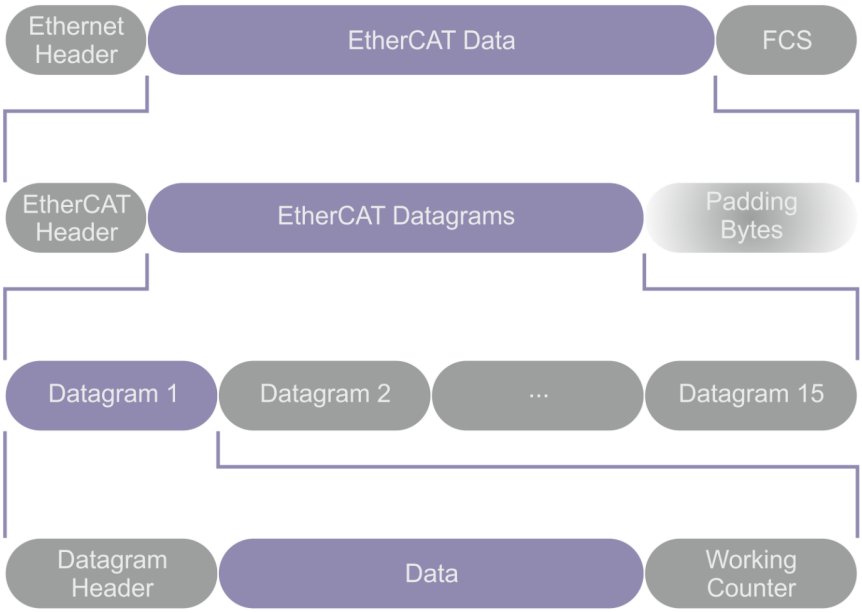
Figur 2.13: PDO kommunikasjon fra master til slave (ref:[25], s.24)

Figur 2.13 viser en forenklet tegning av hvordan et EtherCAT datagram sendes fra master til slavens ESC, via en RJ45 i den fysiske delen. PDO leses fra process data, for så å behandles av µC, til styring av applikasjon. µC skriver så status fra applikasjonen tilbake til prosess data på samme måte. Alt dette skjer "On the fly". Prosess data er en del av EtherCAT-rammens oppbygning. Denne blir videre forklart nærmere.

Visualisering av datautveksling "On The Fly" vist i lenken:[28]

2.6.1 Protokoll kommunikasjons prinsipp

Systemets kommunikasjonsprotokoll er av typen syklisk datautveksling. Denne typen protokoll, utveksler data i form av PDO mellom master og slaver. For å forstå hvordan EtherCAT data er transportert, er det viktig å sette seg inn i EtherCAT-rammens oppbygning. For at EtherCAT-rammer skal kunne kjøre kontinuerlig uten å stoppe på hver slave, må den inneholde spesifikke komponenter, vist i figur 2.14.



Figur 2.14: EtherCAT telegram (ref:[29])

EtherCAT datagram (prosess data)

En enkel EtherCAT-ramme kan inneholde opp til 15 datagrammer, som vist i figur 2.14. Antallet datagrammer avgjøres av hvor mye informasjon som skal utveksles i en kommunikasjons syklus. Siden protokoll for syklisk datautveksling brukes i oppgaven, vil et datagram representere prosess data. Den totale mengden informasjon fordelt over datagrammene kan beskrives som:

$$d_i = [12\ ,\ 1498]\ Byte \qquad \sum(d_1, d_2, ..., d_n) \leq 1498\ Byte \qquad d_i \rightarrow Vilkårlig\ datagram \tag{2.9}$$

Hvert datagram inneholder en datagram header som peker på "linker" i objekt-ordboken, og indikerer hvilken kommando datagrammets innhold refererer til.

EtherCAT data

Datagrammet frakter en eller flere PDO. Disse fungerer som parametre til kommandoen, som header peker på i objekt-ordboken. Det finnes to typer PDO, avhengig av om data skal sendes eller mottas.

- **TxPDO:** brukes til å sende data
- **RxPDO:** brukes til å motta data

Helt til slutt i datagrammet, vil det være en *working counter*.

Working counter:

Hvert datagram har en teller. Telleren økes i verdi med 1 dersom en slave blir adressert, samt en operasjon ble utført. Dersom ingen operasjon blir utført, tilsier dette en verdi på 0. Dette gir masteren mulighet til å sjekke om den planlagte datautvekslingen var en suksess. Ved å sammenligne den forventa verdien til telleren, med den faktiske verdien til telleren, etter informasjonen har passert gjennom alle slavene, sjekker master om kommunikasjonen har vært en suksess eller ikke.

Kommando	Resultat	Teller
Les kommando	Suksess	+1
	Ikke suksess	Ingen endring
Skriv kommando	Suksess	+1
	Ikke suksess	Ingen endring
Skriv/Les kommando	Lesing suksess	+1
	Skriving Suksess	+2
	Skriving og lesing suksess	+3
	Ikke suksess	Ingen endring

For mer informasjon om protokollprinsipp: [29]

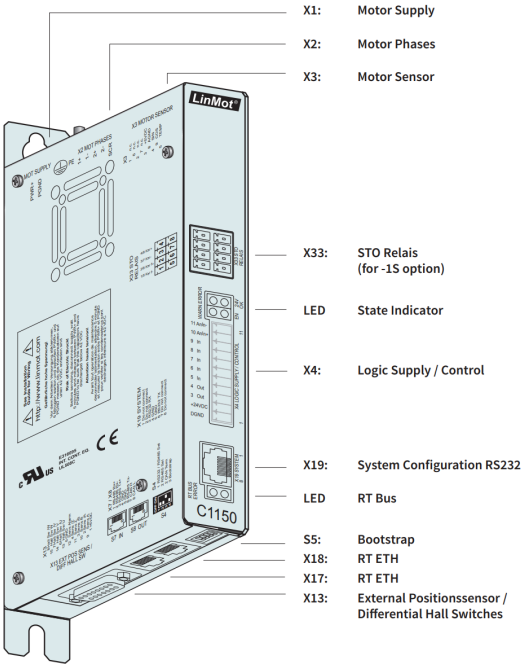
Morten Mossige fra IDE, har laget en modul for kommunikasjon mellom EtherCAT-master og slaver, samt et fungerende nettverk for kommunikasjonsprotokollen. Dette ble videre brukt i oppgaven, for å realisere kommunikasjon mellom ROS og drivere.

Python script fra Morten Mossige samlet i en komprimert mappe finnes:

(https://drive.google.com/drive/folders/1XcRGO27QCqfFxyG65JNJE2ygFlOB4YgI?usp=sharing)

2.7 EtherCAT til LinMot driver

Lineærmotoren er styrt av en LinMot-driver av type C1150-EC-XC-0S, som skal kontrolleres over EtherCAT. All sanntids-kommunikasjon vil gå via X18/X17, mens konfigurasjon av driver vil skje via tilkobling av X19, vist i figur 2.15. Mer informasjon om LinMot-driver, kan finnes i datablad: [30].



Figur 2.15: LinMot driver, [s.30, ref. [30]]

For kontrollering av LinMot-driver, ble programmvaren LinMot-Talk tatt i bruk. Ved hjelp av dette programmet er det mulig å konfigurere driver, samt overvåke kommunikasjonsprosessen. Realisering av motorens funksjon etter hensikt, krevde kun standard bruk av objekt ordboken. Dermed var ingen konfigurasjon av spesialkommandoer nødvendig.

Figur 2.16 viser standard PDO-kartlegging i LinMot driver. TxPDO vil være output fra driver til master, mens RxPDO vil være input, fra master til driver.

TxPDO

Index	Size [Byte]	Byte Offset	Name	Data Type
0x1B00	***18***	***-***	***Variables***	***RECORD***
0x1B62:00	2	0	StateVar	Uint16
0x1D51:00	2	2	StatusWord	Uint16
0x1D8E:00	2	4	WarnWord	Uint16
0x1B8A:00	4	6	DemandPosition	Int32
0x1B8D:00	4	10	ActualPosition	Int32
0x1B93:00	4	14	DemandCurrent	Int32

RxPDO

Index	Size [Byte]	Byte Offset	Name	Data Type
0x1700	***24***	***-***	***Variables***	***RECORD***
0x1D52:00	2	0	ControlWord	Uint16
0x1DB0:00	2	2	MotionCommandHeader	Uint16
0x1E40:00	4	4	MotionCommand Par 1	Word32
0x1E41:00	4	8	MotionCommand Par 2	Word32
0x1E42:00	4	12	MotionCommand Par 3	Word32
0x1E43:00	4	16	MotionCommand Par 4	Word32
0x1E44:00	4	20	MotionCommand Par 5	Word32

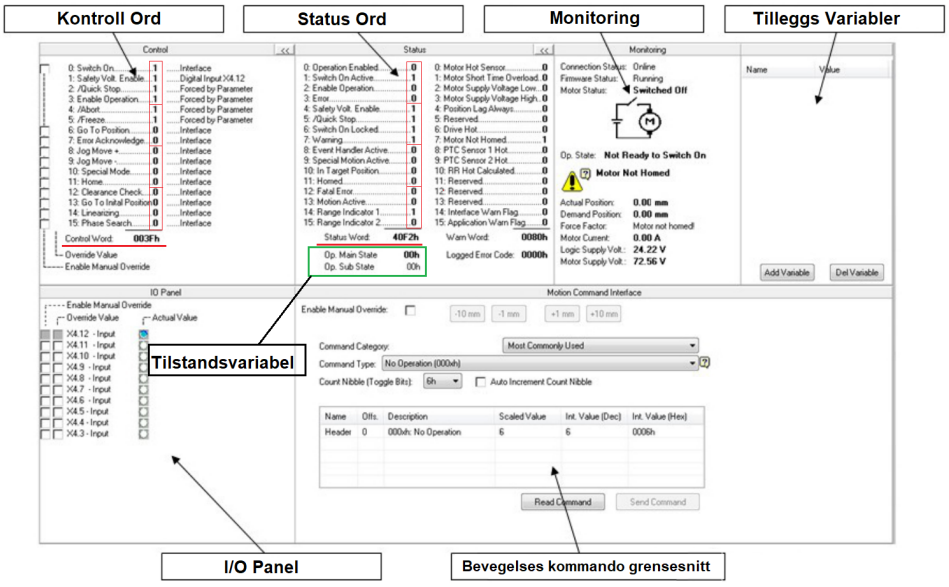
Figur 2.16: Standard kartlegging av PDO i LinMot driver [31, s. 10]

Kontroll ord

Prinsippet for kontrollering av LinMot-driver er bygget på en tilstands maskin, som vist i figur 2.18. Her brukes kontroll ord for å sette driverens hovedtilstand. Når systemet har fått klargjøring av kontroll ordet, vil maskinen gå videre til neste tilstand. Som figur 2.17 viser, er kontrollordet på 16 bit. Hvert bit har en betydning for endring i tilstanden til LinMot-driver.

Status ord

Status ord er som vist i figur 2.17, også på 16 bit. Denne vil fungere som en sjekkliste på driverens operative status. Dette sørger for at man enkelt kan finne ut om det er feil i systemet. Hvis for eksempel bit 10 er satt lik 1, er motoren i ønsket posisjon.



Figur 2.17: LinMot-Talk brukergrensesnitt (ref. [32], s.8)

Tilstandsvariabel

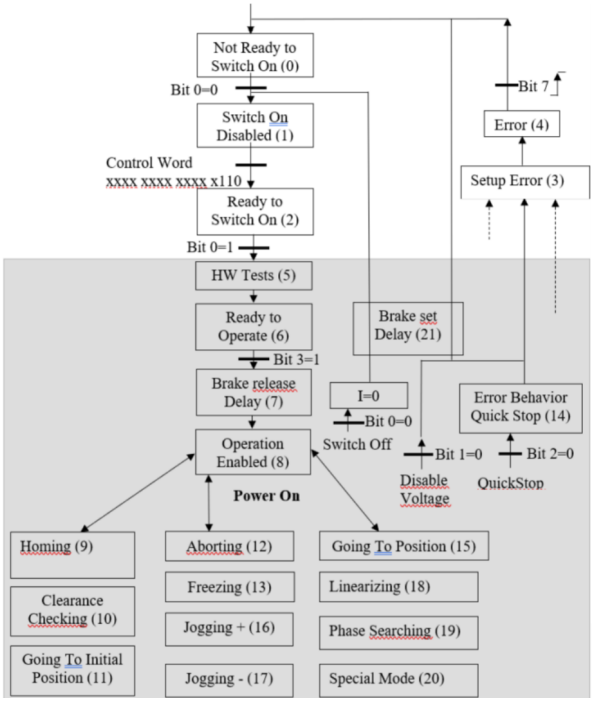
Tilstandsvariabelen beskriver systemets operative tilstand og er delt inn i to tilstander; hovedtilstanden mest signifikante bit (MSB) og undertilstand minst signifikante bit (LSB), vist i figur 2.17. Undertilstandens betydning varierer etter hva hovedtilstanden er. Tabellen under viser en oversikt over hvilke bit som korresponderer til hver tilstand.

Tilstands variabel															
Hovedtilstand (MSB)								Undertilstand (LSB)							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

En oversikt over de ulike hovedtilstandene med korresponderende undertilstandene er vist i LinMot dokumentasjon: [27, s.13].

Tilstands maskin

De ulike tilstandene i tilstandsmaskinen er vist i figur 2.18. Hver tilstand er definert i systemet ved hjelp av tilstandsvariabelen. Endring av tilstand blir utført når driver får klargjøring av kontroll ord.



State Var	
Main State	Sub State
00: Not Ready To Switch On	0
01: Switch On Disabled	0
02: Ready To Switch On	0
03: Setup Error	**Error Code which will be logged**
04: Error	**Logged Error Code**
05: HW Tests	0 (Not yet defined)
06: Ready To Operate	0 (Not yet defined)
07: -	
08: Operation Enabled	**Bits 0..3: Motion Command Count** **Bit 4: Event Handler Active** **Bit 5: Motion Active** **Bit 6: In Target Position** **Bit 7: Homed**
09: Homing	**0Fh: Homing Finished**
10: Clearance Check	**0Fh: Clearance Check Finished**
11: Going To Initial Position	**0Fh: Going To Initial Position Finished**
12: Aborting	Not yet defined
13: Freezing	Not yet defined
14: Quick Stop (Error Behaviour)	Not yet defined
15: Going To Position	**0Fh: Going To Position Finished**
16: Jogging +	**01h: Moving positive** **0Fh: Jogging +Finished**
17: Jogging -	**01h: Moving negative** **0Fh: Jogging -Finished**
18: Linearizing	Not yet defined
19: Phase Search	Not yet defined
20: Special Mode	Not yet defined

Figur 2.18: Tilstandsmaskin og tilstandsvariabler ([27], s.12/13)

Bevegelses kommando grensesnitt

Hvis hovedtilstanden er i bit 8, tilsier bit 0-3 i undertilstanden at driver settes i *motion command count* modus. Dette tillater driveren å sende bevegelses kommandoer til motor, basert på verdier gitt av master. For enkel posisjonsendring av motor, brukes bevegelseskommandoen *VAI Go To Pos*. Denne kommandoen er hentet fra objekt ordboken [27, s.29].

4.3.9 VAI Go To Pos (010xh)

Name	Byte Offset	Description	Type	Unit
Header	0	010xh: VAI Go To Pos	UInt16	-
1. Par	2	Target Position	SInt32	0.1 um
2. Par	6	Maximal Velocity	UInt32	1E-6 m/s
3. Par	10	Acceleration	UInt32	1E-5 m/s^2
4. Par	14	Deceleration	UInt32	1E-5 m/s^2

Figur 2.19: VAIGoToPos bevegelses kommando ([27], s.29)

Som vist i figur 2.16, vil driver ta imot RxPDO fra master, med et kontrollord som spesifiserer hovedtilstand til *operation mode* (bit 8), se figur 2.18. En header vist i figur 2.19, peker mot en unik kommando i ordboken. Headeren til bevegelseskommandoen defineres ved hjelp av tre ledd.

- **Master ID:** Spesifiserer kommando gruppen.
- **Sub ID:** Identifiserer ulike kommandoer i samme kommando gruppe.
- **Command counter:** Utfører ny kommando hvis denne verdien har endret seg.

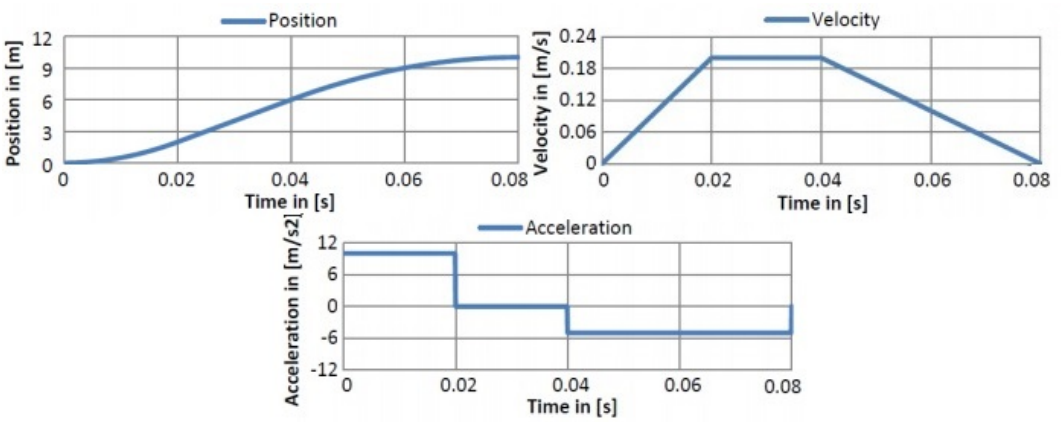
Headeren tillater driveren å forstå hvilke kommando som skal utføres. For å faktisk utføre kommandoen, må parametrene vist i figur 2.19 defineres. Dette tilfredsstiller kommandoens inputverdier og driveren vil dermed kunne bruke dette til styring av applikasjon. En liste over alle mulige kommandoer som kan brukes til posisjons endring er gitt i objekt ordboken [27, s.22-27].

I Morten sin modul, er dette definert som vist i listing 1

Listing 1: Vai Go To Pos kommando (ref. Morten Mossige)

```
103     def ExecuteVAIGoToPos(self):
104         self._motionCmdPar1 = int( self._targetPos * 10000)
105         self._motionCmdPar2 = int( self._maxVel * 1000000 )
106         self._motionCmdPar3 = int( self._accel *100000 )
107         self._motionCmdPar4 = int( self._decel * 100000 )
108 
109         self._motionCmdHeader = VAIGoToPos | self.GetCountNibble()
```

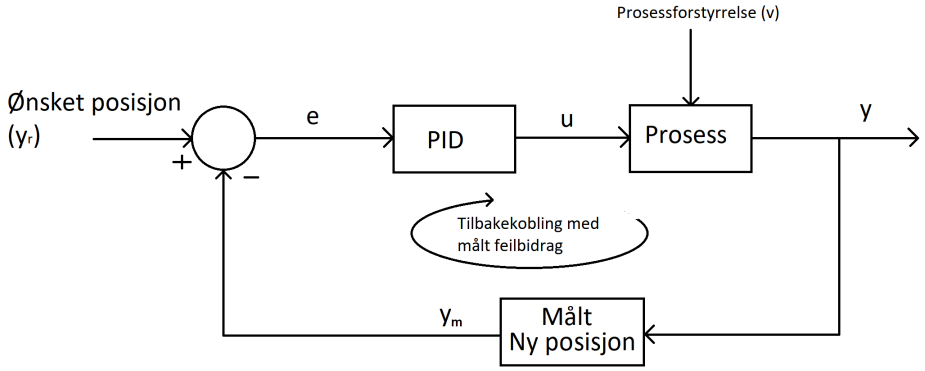
Denne kommandoen er i katergorien «*Velocity Acceleration Interpolator (VAI)*». Dette betyr at driveren opptrer med egen interpolator. Med utgangspunkt i ønsket posisjon, maks hastighet og akselerasjon, generere driveren en planlagt bevegelseskurve, motoren skal bevege seg etter. Et eksempel på dette er vist i figur 2.20. Hvis driver mottar en ny posisjonskommando, mens motoren følger forrige bevegelsekurve, vil driveren prioritere den nye isteden. Mer informasjon om driverens interpolator finnes i dokumentasjon ([27], s.81).



Figur 2.20: VA-interpolator graf (ref. [44], s.109)

2.8 Regulering med negativ tilbakekobling

Problemstilling for regulering setter søkelys på en hurtig og stabil prosess. Systemet skal bruke et settpunkt som referanse og regulerer pådraget til motoren ut fra ballens posisjonsavvik fra settpunkt. En god løsning på dette er bruk av tilbakekoblingsmetoden. En PID-regulator blir benyttet for å regulere pådraget, se figur 2.21.



Figur 2.21: PID tilbakekoblings regulator

Ved hjelp av tilbakekoblings metoden, er det mulig å finne reguleringsavviket innenfor akseptale grenser. Pådraget *(u)* regnes ut med ballens avvik *(e)* fra ønsket verdi.

$$e = y_r - y_m \qquad y_r \rightarrow \textit{Ønsket ballposisjon} \qquad y_m \rightarrow \textit{Målt ballposisjon} \tag{2.10}$$

Totalt pådrag defineres som:

$$u = u_0 + u_e \qquad u_0 \rightarrow \textit{Nominelt pådrag} \qquad u_e \rightarrow \textit{Pådrag etter avvik} \tag{2.11}$$

Nominelt pådrag er ikke ønsket her. Dette er fordi vippen skal stå i ro når ballen er stanset i ønsket posisjon og vil derfor neglisjeres. Det er ønsket å bruke en PID-regulator, da denne typen regulator bruker tilbakekoblingsmetoden. En slik regulator består av tre ledd, som sammen kan oppnå ønsket regulering i prosessen.

2.8.1 P-regulator

En P-regulator vil gi en reguleringsforsterkning (Kp), proporsjonalt med avviket (e). Ved å øke/senke reguleringsforsterkningen Kp, vil vippevinkelens utslag endres relativt til ballens posisjonsavvik.

$$u_e = K_p \cdot e \tag{2.12}$$

Stasjonært avvik rundt e = 0, kan unngås ved å stille K_p til en tilstrekkelig stor verdi, men kan resultere i brå bevegelser.

2.8.2 PI-regulator

PI-regulatoren vil i tillegg til P-delen inneholde et integrerende ledd (I), som integrerer opp avviket fra t = 0s, med et pådrag på K_p/T_i.

$$u_e = K_p \cdot e + \frac{K_p}{T_i} \int_0^t e(\tau) d\tau \tag{2.13}$$

I-delens bidrag består av et resultat av oppsamlet gjennomsnittlig avvik. Dette hjelper med å forminske avvik, ved å tilpasse seg bevegelsesmønsteret.

2.8.3 PID-regulator

En komplett PID-regulator tar i bruk et siste D-ledd, som har som funksjon å derivere avviket, for å raskere innregulere prosessen ved brå endringer. Pådraget bestemmes av $K_p \cdot T_d$.

$$u_e = K_p \cdot e + \frac{K_p}{T_i} \int_0^t e(\tau) d\tau + K_p \cdot T_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.14}$$

For å realisere dette vil Pythons standard PID-regulator *simple pid*, benyttes. Formeler hentet fra: [33]. Instilling av forsterkningsverdier er basert på: [34].

2.8.4 Måling av presisjon

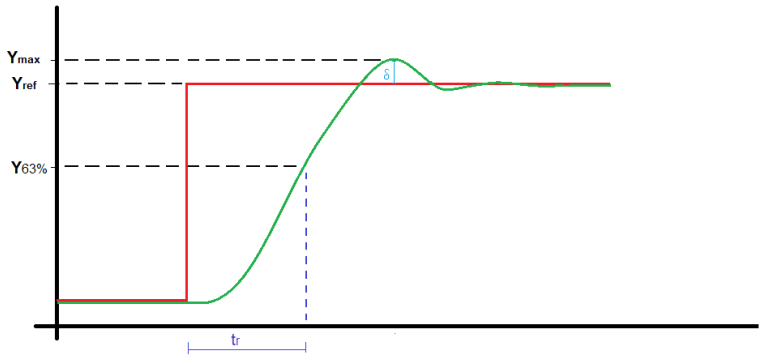
Etter resultat av balanseringstester er optimalisert, er det ønskelig å utføre en systemidentifikasjon av resultatene. Dermed kan man analysere egenskaper til responsen. Gitt at systemet er av 2.orden, kan overføringsfunksjonen brukes til å visualisere ulike egenskaper i responsen. Standardform for 2.ordens system er gitt i ligning 2.15. Formel hentet fra: [35]

$$H(s) = \frac{\omega_0^2}{s^2 + 2 \cdot \zeta \cdot \omega_0 \cdot s + \omega_0^2} = \frac{1}{(\frac{s}{\omega_0})^2 + 2 \cdot \frac{\zeta}{\omega_0} \cdot s + 1} \tag{2.15}$$

En viktig faktor i overføringsfunksjonen, vil være dempnings ratio ζ. Ut ifra et grafisk resultat fra reguleringen, vil det være mulig å kartlegge systemets dempningsegenskaper som vist:

$$\begin{aligned} \zeta > 1 \quad &\Rightarrow \quad \textit{Overdempet} \\ \zeta = 1 \quad &\Rightarrow \quad \textit{Kritisk dempet} \\ 0 < \zeta < 1 \quad &\Rightarrow \quad \textit{Underdempet} \end{aligned}$$

Ved hjelp av disse kriteriene, kan man bruket verdier fra responsen til å kategorisere systemet. I figur 2.22 vises et eksempel på en underdempet respons. Denne brukes videre for å sette grunnlag på analysen av virkelig system.



Figur 2.22: 2.ordens respons

Ved å undersøke grafens innsvingning, kan man finne dempnings ratioen i systemet. Dette ved bruk av oversvingsfaktoren δ. Denne forklarer prosentandelen av oversving, i forhold til referanseverdi. Ut fra målene i figur 2.22, kan man bruke formel for oversvingsfaktor, for så å bruke verdien til å finne faktoren ζ, som vist i ligning 2.16 hentet fra [35]

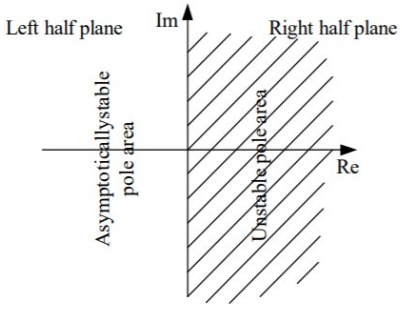
$$\delta = \frac{y_{maks} - y_{ref}}{y_{ref}} \qquad \delta = e^{\frac{-\zeta\pi}{\sqrt{1-\zeta^2}}} \Rightarrow \zeta = \sqrt{\frac{\ln\delta^2}{\ln\delta^2 + \pi^2}} \tag{2.16}$$

Formel for den naturlige frekvensen ω_0 i et underdempet systemet kan finnes med hjelp av responstiden t_r, vist i figur 2.22. Dette er tiden det tar, før responsen oppnår 63% av referanseverdi.

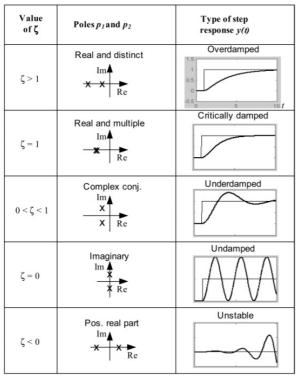
$$\omega_0 = \frac{1.5}{t_r} \tag{2.17}$$

Ut ifra formel for overføringsfunksjonen, vist i ligning 2.15, kan man enkelt finne polene i systemet. Når nevner settes lik 0, gir dette en andregradsfunksjon som kan løses for den imaginære verdien s. Dette gir en generell formel for de kompleks konjugerte polene s_1 og s_2. Hentet fra: [36].

$$s_1, s_2 = -\zeta \cdot \omega_0 \pm j\omega_0\sqrt{1-\zeta^2} \tag{2.18}$$



Figur 2.23: Polområde (hentet fra: [37])



Polenes medfølgte egenskaper (hentet fra: [33])

Ved å bruke utregna dempingsfaktor og den naturlige frekvensen, kan man avgjøre polplasseringer. Disse plasseringene beskriver hvor raskt systemet er med plassering på den reelle aksen. I tillegg til stabiliteten, med plassering på den imaginære aksen.

Del II

Konstruksjon

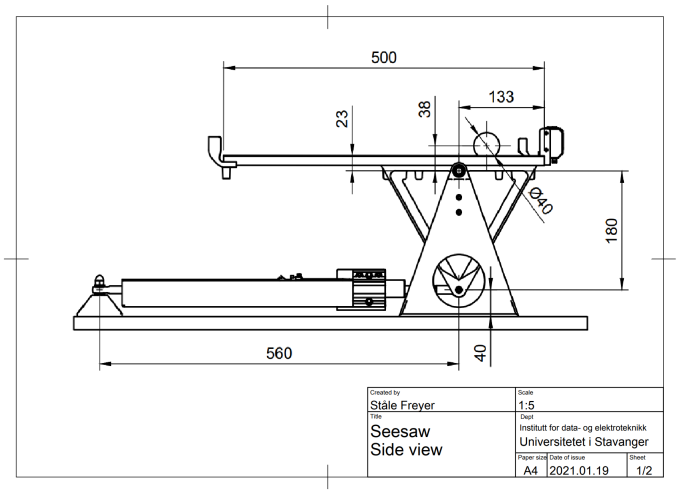
I konstruksjons delen vil løsningsmetoder brukt, for å realisere oppgaven presenteres. Her fremstilles det studentene har gjort selv i prosjektet. Dette innebærer en redegjørelse for grafisk konstruksjon av vippesystemet og implementering av EtherCAT-kommunikasjon mellom ROS og driver. I tillegg vil utdyping av ulike reguleringssystemer brukt for å oppnå ønsket kontrollering, være et sentral brikke i konstruksjonen. Det blir satt fokus på fremgangsmåter og metoder brukt for å tilrettelegge for eksperimentelle oppsett.

Det er ønsket å opprette en virtuell simulering av vippesystemet. Dermed bør det visualiseres en grafisk modell av vippen, som kan styres ved hjelp av et simuleringsverktøy. Ut fra dette, kan man lage et testprogram for styring av modellen, for å opprette et grunnlag for testing av fysisk modell. Da dette er utført, kan kommunikasjonsprotokollen implementeres til ROS, for videre styring og regulering via EtherCAT.

3 Visualisering

Før virtuell simulering settes i gang, bør systemet visualiseres. Dette skaper et utgangspunkt i robotformat, som senere kan styres ved å implementere bevegelse til de ulike leddene.

For å lage en god visualisering av vippesystemet ved bruk av ROS og MoveIt, er det nødvendig å designe en grafisk tredimensjonal modell, med presise mål. For å realisere dette brukes RViz, da dette er et standardprogram i MoveIt og operativt i samspill med ROS.



Figur 3.1: Mål fra vippemodellen (ref. Ståle Freyer)

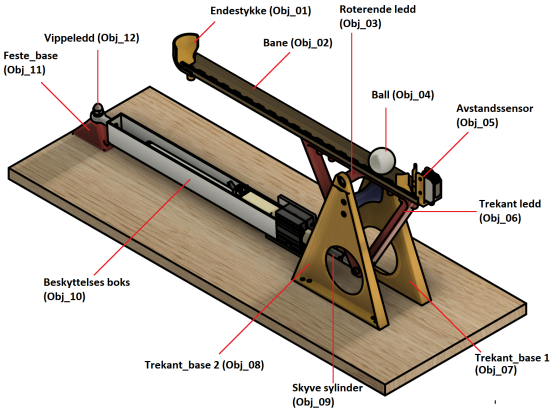
Figur 3.1 viser oversikt over nødvendige mål av vippe-oppsettet. Denne tegningen er brukt som referanse under deldesign og setter grunnlaget på den grafiske modellen implementert, ved bruk av URDF.

3.1 Grafisk design av deler, FreeCAD

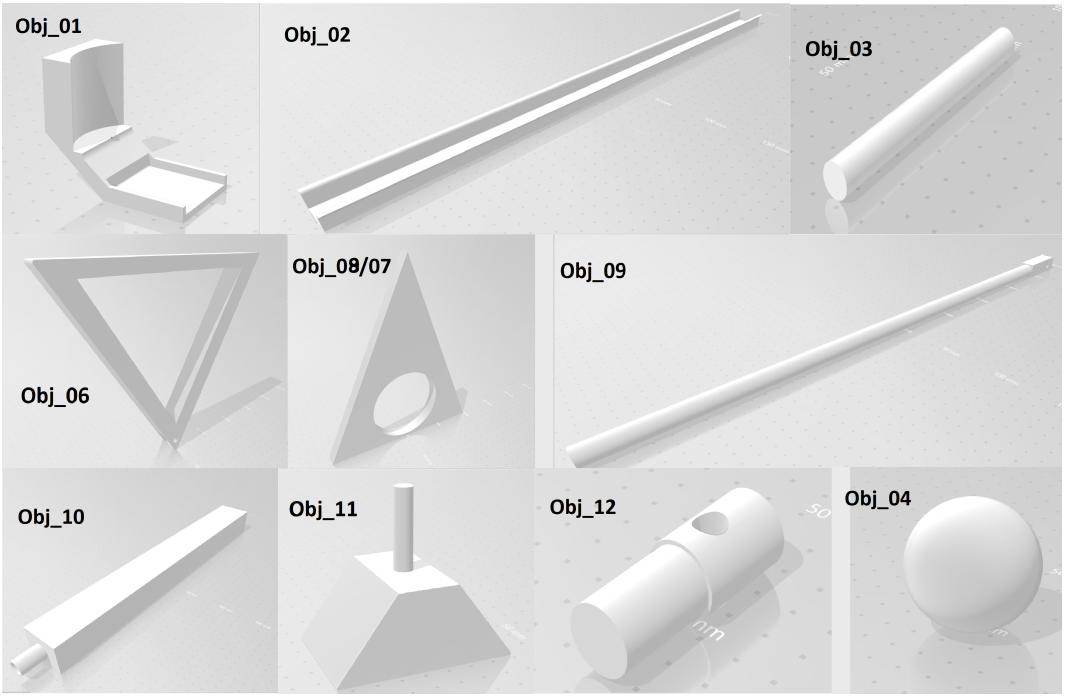
En av utfordringene knyttet til konstruksjon av den grafiske modellen, er design av komplekse komponenter. URDF har som standard, kun mulighet til å definere enkle geometriske figurer. For å unngå klumsete og upresist design, bør det importeres *.mesh* filer, med spesiallagde deler fra et 3D-modellerings program. Figur 3.2 viser en planskisse av de sammensatte ulike komponentene som skal lages. Denne vil refereres til videre i delkapittelet.

For å konstruere grafiske spesialdeler, ble 3D-design platformen FreeCAD tatt i bruk. Med dette verktøyet er det mulig å designe deler med formkompleksitet og spesifiserte mål etter behov. Hver del kan importeres inn til URDF, for videre bruk i grafisk robotmanipulering. Nedlasting av FreeCAD, ble gjort via deres hjemmeside: [38]

Alle deler ble laget i FreeCAD, med unntak av baseplaten og avstandssensor. Baseplaten ble definert som en grunnstruktur i form av en rektangulær kube, mens avstandssensor ble neglisjert fra visualisering, forutsett at ballens posisjon kunne hentes direkte ut ved hjelp av et *topic* i ROS. Figur 3.3 viser alle delene designet i FreeCAD.



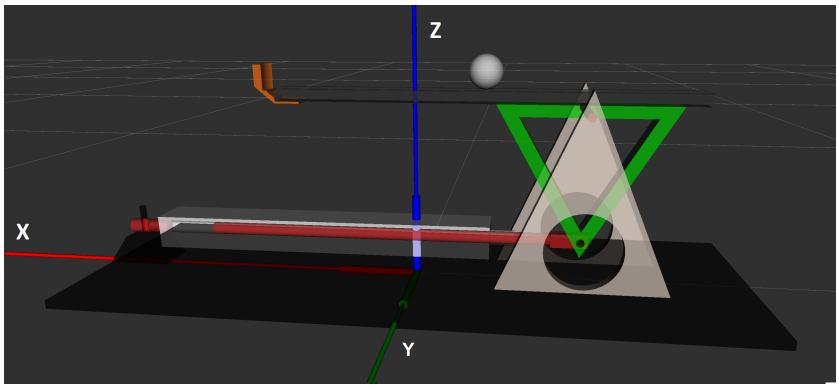
Figur 3.2: Vippe oversikt (Ref. Ståle Freyer)



Figur 3.3: Oversikt over delene laget i FreeCAD

3.2 Visualiseringsverktøy, RViz

Visualisering av modellen er nyttig for å skape et godt utgangspunkt for simulering. I ROS er det mulighet for å importere definerte modeller til 3D visualiseringsverktøyet RViz. Hver komponent i robotstrukturen må settes sammen, ledd for ledd i URDF. For så å åpnes i RViz for inspeksjon.



Figur 3.4: Modell av vippesystem i RViz

3.2.1 Modellering i RViz

URDF tillater bygging av modell i "gren struktur". Deler defineres og settes sammen ved hjelp av flere *links* og *joints*.

Link (lenke)

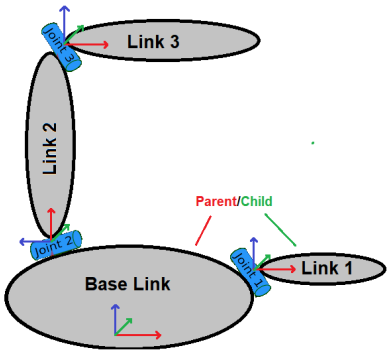
En *link* representerer èn enkel komponent i modellen. Den defineres med et unikt navn, et eget aksesystem og plassering i simulert verden. Samtidig defineres komponentens fysiske egenskaper. Dette innebærer blant annet skala, kollisjon og masse. Ettersom modellen består av flere unike *links*, lages en generell klasse for definering av disse. Ved bruk av denne klasssen, er det mulig å importere delene fra FreeCAD, for så å tilpasse komponentenes plassering i modellen. Hvert objekt fra figur 3.2, representerer en *link*. Den generelle klassen for definering av ulike *links*, er vist i figur 3.5.

```
1  <xacro:macro name="n_link_mesh" params="name origin_xyz origin_rpy meshfile meshscale mass ixx ixy ixz iyy iyz izz color">
2      <link name ="${name}">
3          <inertial>                                                                                          Masse
4              <mass value="${mass}"/>                                                                         Plassering
5              <origin rpy="${origin_rpy}" xyz="${origin_xyz}"/>                                               Treghet (inertia)
6              <inertia ixx="${ixx}" ixy="${ixy}" ixz="${ixz}" iyy="${iyy}" iyz="${iyz}" izz="${izz}"/>
7          </inertial>
8          <collision>
9              <origin rpy="${origin_rpy}" xyz="${origin_xyz}" />                                              Kollisjon
10             <geometry>
11                 <mesh filename="${meshfile}" scale="${meshscale}" />
12             </geometry>
13         </collision>
14         <visual>
15             <origin rpy="${origin_rpy}" xyz="${origin_xyz}" />                                              Visuell plassering
16             <geometry>                                                                                      Importert 3D objekt
17                 <mesh filename="${meshfile}" scale="${meshscale}" />                                        Farge
18             </geometry>
19             <material name="${color}"/>
20         </visual>
21         <gazebo reference="${name}">                                                                        Soliditet
22             <kp>100.0</kp>                                                                                  Friksjon
23             <kd>10.0</kd>
24             <mu1>100.0</mu1>
25             <mu2>10.0</mu2>
26         </gazebo>
27     </link>
28 </xacro:macro>
```

Figur 3.5: Generell klasse for definering av en *link*

Joint (ledd)

Hver definerte *link* må ha et utgangspunkt å plassere seg etter. Hvis en *link* ikke har et utgangspunkt, vil den ikke ha noen funksjon. Dette medfølger at hver *link* må bygges ut ifra en annen *link*, eventuelt fra en basestruktur i simulert verden. Dette setter grunnlaget på URDF sin metode for bygging av "gren strukturer". For å lage et slikt forhold mellom to *links*, brukes en *joint*. En *joint* fungerer som et festepunkt mellom to *links*, og oppretter dermed et nytt startpunkt, som den nye *linken* kan bygge seg ut ifra. Den nye *linken* defineres som en *child-link*, mens eksisterende *link* som skal bygges ut ifra, defineres som en *parent-link*. Dette skaper et master/slave forhold mellom to *links*, ved bruk av en *joint*. Dette visualiseres i figur 3.6.



Figur 3.6: Forholdet mellom *link* og *joint*

For å enklere kunne definere ulike *joints*, lages en generell klasse. Ved å kalle på denne klassen for definering av hvert unike *joints*, er det mulig å enkelt karakterisere hver *joint*, for å forenkle koden. Klassen for definering av *joints* vises i figur 3.7.

```
1  <xacro:macro name="n_joint" params="name type axis_xyz origin_rpy origin_xyz parent child limit_e limit_l limit_u limit_v">
2      <joint name="${name}" type="${type}">
3          <axis xyz="${axis_xyz}" />
4          <limit effort ="${limit_e}" lower="${limit_l}" upper="${limit_u}" velocity="${limit_v}"/>
5          <origin rpy="${origin_rpy}" xyz="${origin_xyz}"/>
6          <parent link="${parent}"/>
7          <child link="${child}"/>
8      </joint>
9      <transmission name="trans_${name}">
10         <type>transmission_interface/SimpleTransmission</type>
11         <joint name="${name}">
12             <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
13         </joint>
14         <actuator name="motor_${name}">
15             <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
16             <mechanicalReduction>1</mechanicalReduction>
17         </actuator>
18     </transmission>
19 </xacro:macro>
```

Bevegelses type
hvilken akse bevegelsen skal utføres
maks/min bevegelses intervall
Forskyvning av akse (ifht. parent link)
Parent/Child
Standard for bevegelse i Gazebo

Figur 3.7: Generell klasse for definering av *joints*

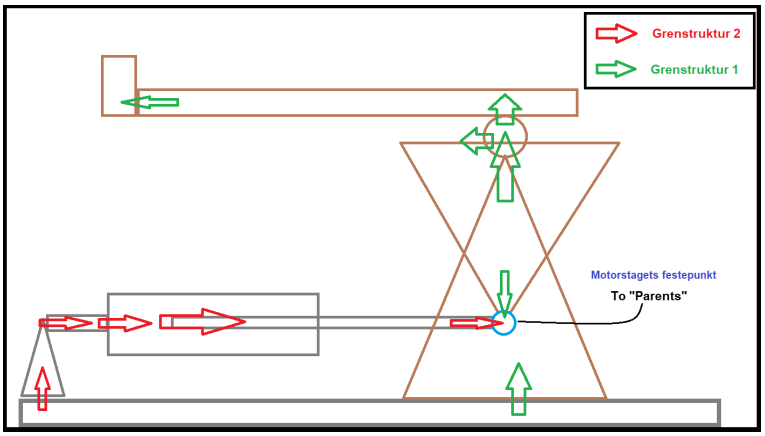
For å realisere bevegelighet i *joints*, brukes *joint type* vist i figur 3.7, linje 2. Dette tillater *child-link* å bevege seg en bestemt aksebevegelse med *parent-link* som utgangspunkt. En tabell over bevegelige ledd, med navn basert på figur 3.2, er vist i tabellen.

Child link	navn	joint type	akse	Parent link
Obj_12	Vippeledd	Revolve	Y	Obj_11
Obj_03	Roterende ledd	Revolve	Y	Obj_06
Obj_09	Motor stag	Prismatic	X	Obj_10

Joint type: Revolve i y-akse, gir en ønsket "*pitch*" bevegelse til de roterende leddene.
Joint type: Prismatic i x-akse, gir en "*surge*" forskyvning til motorstag. Med dette er krav til vippens bevegelse vist i figur 2.2, tilfredsstilt for modellen.

3.2.2 Sammenstilling i visualisering

URDF gir som nevnt, kun mulighet for å bygge i en "grenstruktur". Dette innebærer at en *parent-link* kan ha flere *child-links*, men ikke omvendt. En modell hvor en *child-links* har mer enn en *parent-link* kalles for en "*closed-loop struktur*". I vippemodellen fra figur 3.2, vil et festepunkt mellom Obj_06 og Obj_09 ikke være mulig, da denne ville fått to parents. Dette er visualisert i figur 3.8. Grunnen til at dette er et problem, er fordi modellen ikke har mulighet til å være sammenhengende uten en "*closed-loop struktur*", og dermed ikke fungere etter hensikt.



Figur 3.8: *Closed-loop* struktur visualisert

Løsningen på dette er å bruke funksjon for festepunktets bevegelse fra ligning 2.2. Dette vil sette et utgangspunkt for den geometriske sammenhengen mellom de tre bevegelige leddene nevnt i tabell s.27. Avviket fra festepunktets bevegelse i x-akse og motorstag forskyvning vist i figur 3.9, er ekstremt lite. Dette førte til at tilnærmingen vist i ligning 3.1 brukes.

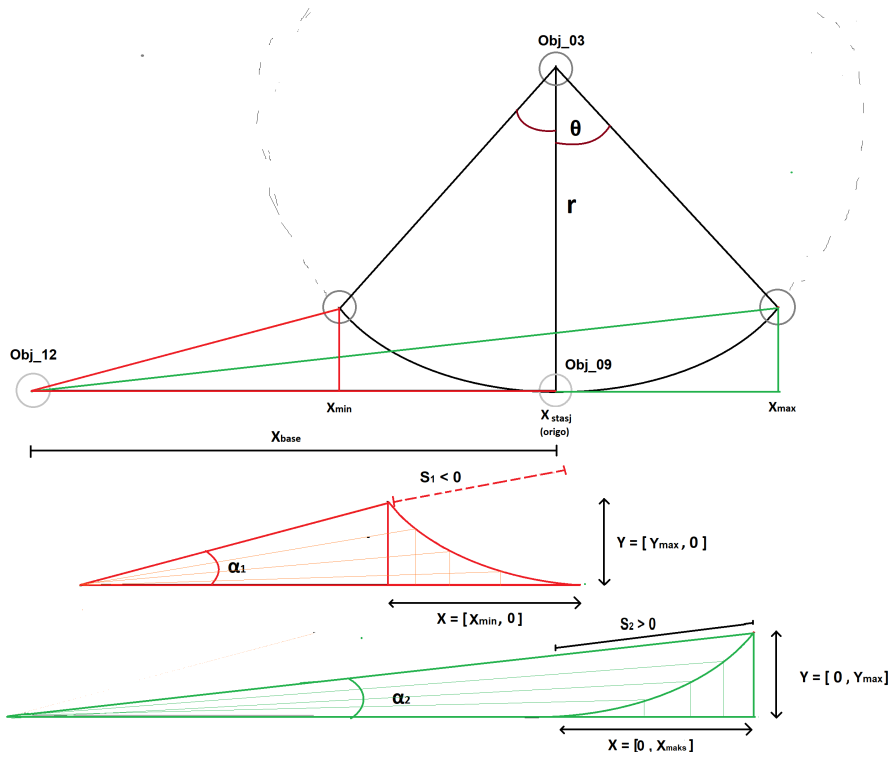
$$Y(x) = r - \sqrt{r^2 - x^2} \approx Y(s) = r - \sqrt{r^2 - S^2} \qquad S \longrightarrow Stag\ forskyvning \tag{3.1}$$

Målet var å få motorstagets festningspunkt til å passe denne bevegelsen. For å oppnå harmoniske bevegelse med de roterende leddene, slik at modellen kan simulere et festepunkt mellom stag og vippe, må vinkel θ og vinkel α brukes som funksjoner av stag forskyvning S. Ut ifra sammenhengen mellom vinklene og stagets forskyvning, brukes pytagoras læresetning [s.17, ref. [1]], for å finne følgene formler. Merk at S ha nullpunkt i X_{stasj}. Dette betyr at forskyvning kan være negativ.

$$\alpha = \arctan(\frac{Y(x)}{X_{base} + x}) \approx \arctan(\frac{Y(S)}{X_{base} + S}) \tag{3.2}$$

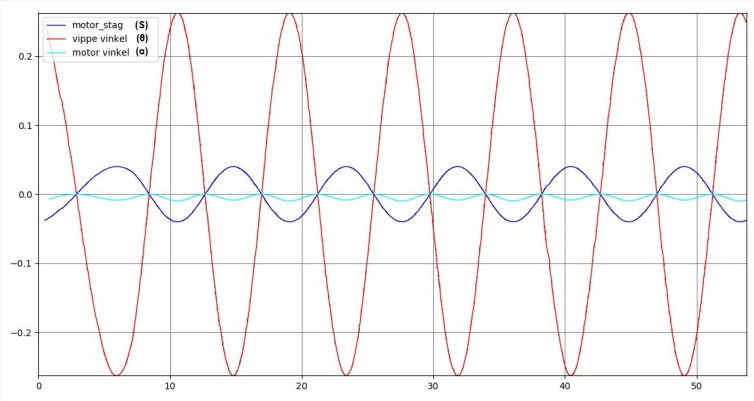
$$\theta = \arcsin(\frac{x}{r}) \approx \arcsin(\frac{S}{r}) \tag{3.3}$$

Oversikt over alle målene brukt for å skape en geometrisk sammenheng, er vist i figur 3.9. For å realisere synkroniserte bevegelser mellom alle leddene i simulering, opprettes en node i ROS ved hjelp av Python.



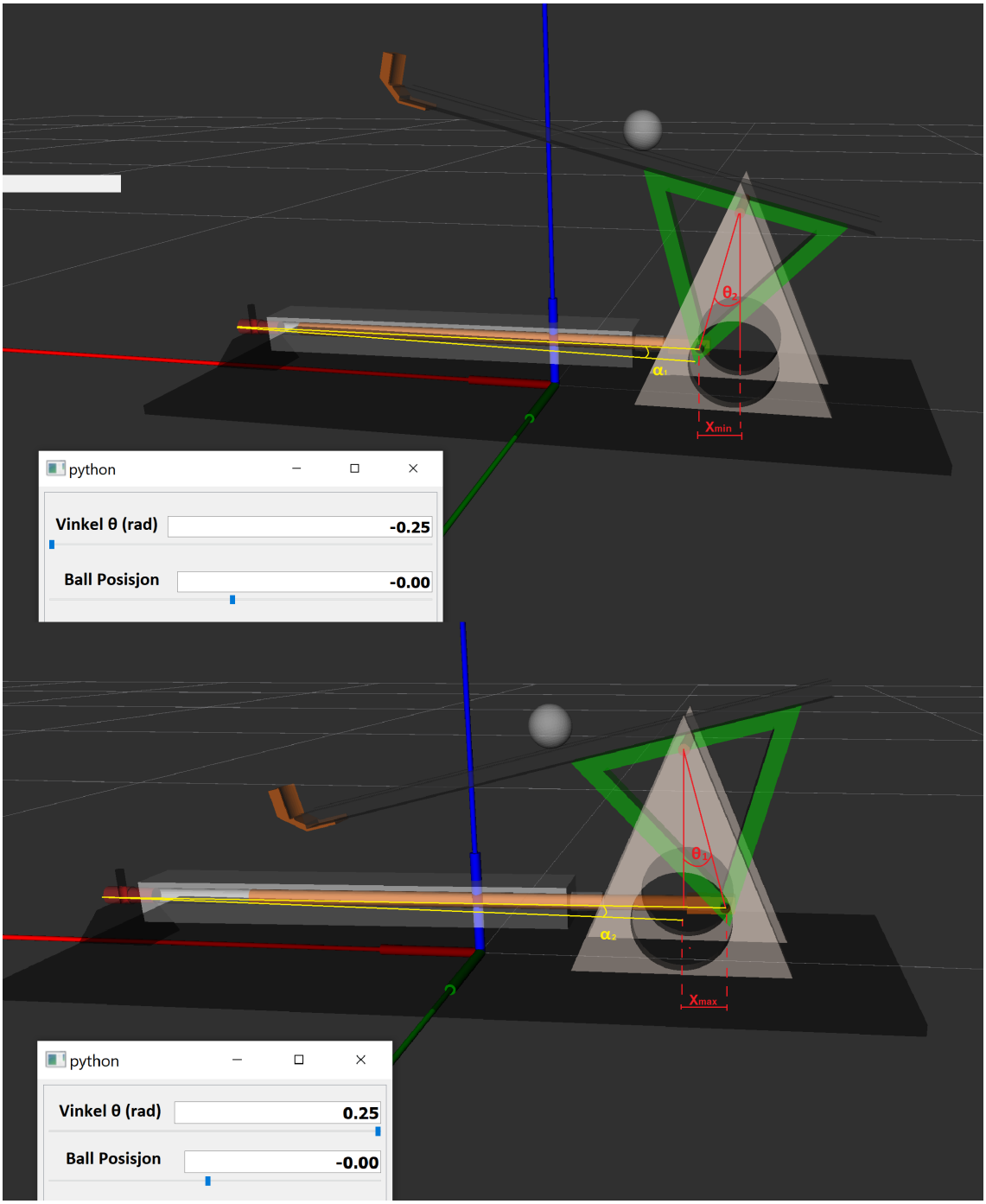
Figur 3.9: Geometrisk sammenheng

Ved å bruke utregning av vippevinklenes funksjon med hensyn på stagforskyvning i ROS noden, vil vinklene bevege seg etter staget, og dermed simulere et festepunkt. Figur 3.10, viser en kurve med oversikt over de bevegelige leddenes sammenheng under drift. Et sinus signal ble brukt som pådrag på motorstag for å demonstrere.



Figur 3.10: Sammenheng mellom bevegelige ledd

For å verifisere at verdiene stemmer overens, visualiseres resultatet i RViz. Dette er for å se hvordan sammenhengen virket estetisk sett. Resultatet er visualisert i figur 3.11.



Figur 3.11: Geometrisk sammenheng i RViz

4 Simulering

En viktig del av prosjektet er å lage en virtuell simulering av det fysiske systemet. Hensikten med dette er å utføre tester på en grafisk modell, med funksjonalitet som speiler den fysiske modellen. Dette vil skape et godt grunnlag for å sette begrensninger ut ifra modellens oppførsel og respons, samt være en nøkkelbrikke for testing av reguleringssystemet.

Det vil være viktig å gjøre de grunnleggende testene på den virtuelle modellen, før den fysiske modellen. Dette er for å unngå slitasje på motoren og bevegelige ledd, men også for å skape et utgangspunkt for enkel testing av ulike metoder. Målet i dette kapittelet, er å tilrettelegge for testing av modell i simulasjon.

4.1 Simuleringsverktøy, Gazebo

For å utføre presise tester på vippen, trengs det et bra simuleringsprogram som har mulighet til å simulere en verden og implementere fysiske egenskaper til vippemodellen. Simuleringsverktøyet Gazebo tilbyr muligheten til å presist simulere komplekse roboter, på en svært effektiv måte. I tillegg er Gazebo et standard program i ROS og operativt i samspill med URDF.

Listing 2: .Launch fil

```
<launch>
  <!--Argumenter-->
  <arg name="sim" default="false"/>
  <arg name="rviz" default="false"/>
  <arg name="rate" default="50.0"/>

  <!-- Hvis sim er satt til sann, vil simulasjon av modellen i Gazebo starte -->
  <group if="$(arg sim)">

      <!-- Initialiserer verden og starter kontrollere -->
      <include file="$(find armbane)/launch/PingPongVippe_world.launch" />
      <include file="$(find armbane)/launch/PingPongVippe_control.launch" />

      <!-- Python noder/ program for manupulering av robot -->
      <node name="Python_control_node" pkg="armbane" type="Python_control.py" output="screen" respawn="true">
          <param name="rate" value="$(arg rate)"/>
      </node>
      <node name="info_print_node" pkg="armbane" type="sub_node.py" output="screen" respawn="true" />

      <!--rqt hjelpemidler-->
      <node pkg="rqt_reconfigure" type="rqt_reconfigure" name="rqt_reconfigure"/>
      <node pkg="rqt_console" type="rqt_console" name="rqt_console"/>

  </group>

  <!-- Visuallisering i RViz -->
  <group if="$(arg rviz)">
      <!--Starter RViz launch fil-->
      <include file="$(find armbane)/launch/rviz.launch" />
  </group>

</launch>
```

Listing 2 viser ".launch" filen som implementerer URDF til enten Gazebo (linje 8-24) eller RViz (linje: 27-30). Denne filen er plassert i *Catkin workspace*, som vist i figur 2.9. Videre vil funksjonaliteten og oppbygning til sytemet beskrives ut ifra listing 2, ved å gå gjennom forskjellige noder opprettet i prosjektets virtuelle bibliotek og hvordan de kommuniserer.

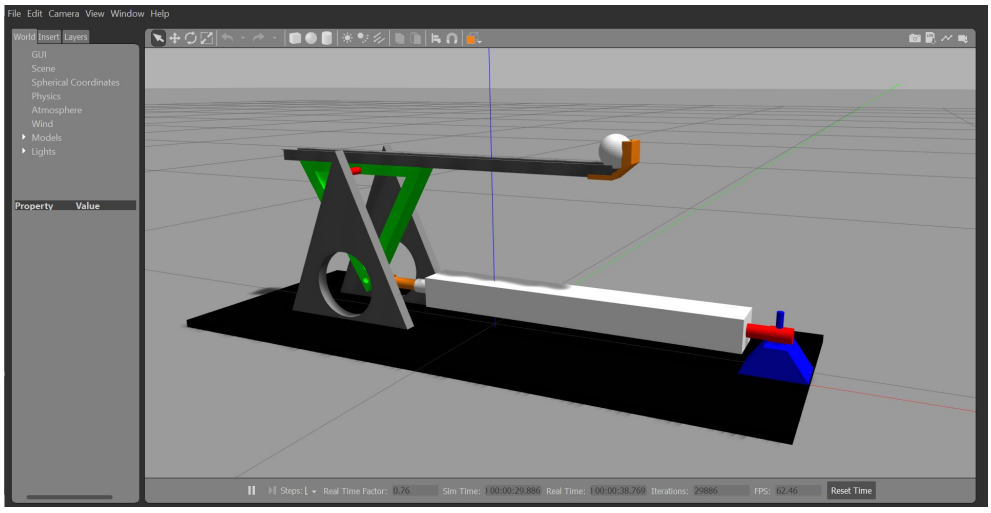
4.1.1 Implementere visuell modell til simulasjon

Hoved ".launch" filen starter med å hente data om Gazebo fra *PingPongVippe_world.launch*. Dette innebærer opprettelsen av en simulert verden. ROS finner så URDF beskrivelsen av den visualiserte modellen. Dermed importeres modellen til simulert verden. Dette vises i listing 3.

Listing 3: PingPongVippe_world.launch

```
1   <launch>
2       <!-- initialiserer tom gazebo verden(simulasjons vindu) -->
3       <include file="$(find gazebo_ros)/launch/empty_world.launch"></include>
4
5       <!-- finner Robot model (urdf) -->
6       <param name="robot_description" command="$(find xacro)/xacro --inorder
            '$(find armbane)/urdf/PingPongVippe.xacro'"/>
7       <arg name="x" default="0"/>
8       <arg name="y" default="0"/>
9       <arg name="z" default="0.005"/>
10
11      <!-- importerer robot modell til gazebo verden -->
12      <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="
            false" output="screen"
13              args="-urdf -param robot_description -x $(arg x) -y $(arg y) -z
                    $(arg z) -model ping_pong_vippe"/>
14  </launch>
```

I prosjektets *workspace*, vil URDF koden: "*PingPongVippe.xacro*" definere hele grenstrukturen, ledd for ledd. Den geometriske sammenhengen mellom de bevegelige leddene forklart i kapittel 3.2.2, appellerer til modellen i Gazebo. Resultatet av modellen implementert til simulert verden, er som vist i figur 4.1.



Figur 4.1: Vippemodell i Gazebo

4.1.2 Styring av simulert modell

Uten en kontroller node, vil modellen ikke ha noen funksjon. Dermed er det ønskelig å implementere en posisjons kontroller til hvert bevegelige ledd, slik at modellen kan styres manuelt. Systemet må også kunne rapportere tilbake posisjons status. For å realisere dette, defineres leddene som "kontrollerbare" i en *.yaml* fil, og hentes inn i hovedfilen slik at kontrolleren kan kommunisere med Gazebo. For å realisere styring av hver *joint*, brukes følgende funksjoner:

- ***JointPositionController*:** Sørger for at hver *joint* kan motta en posisjonsverdier.
- ***JointStateController*:** Publiserer hver enkelt *joint* sin tilstand til nodenettverk.
- ***RobotStatePublisher*:** Publiserer modellens fellestilstand til nodenettverk.

Listing 4: Control.launch

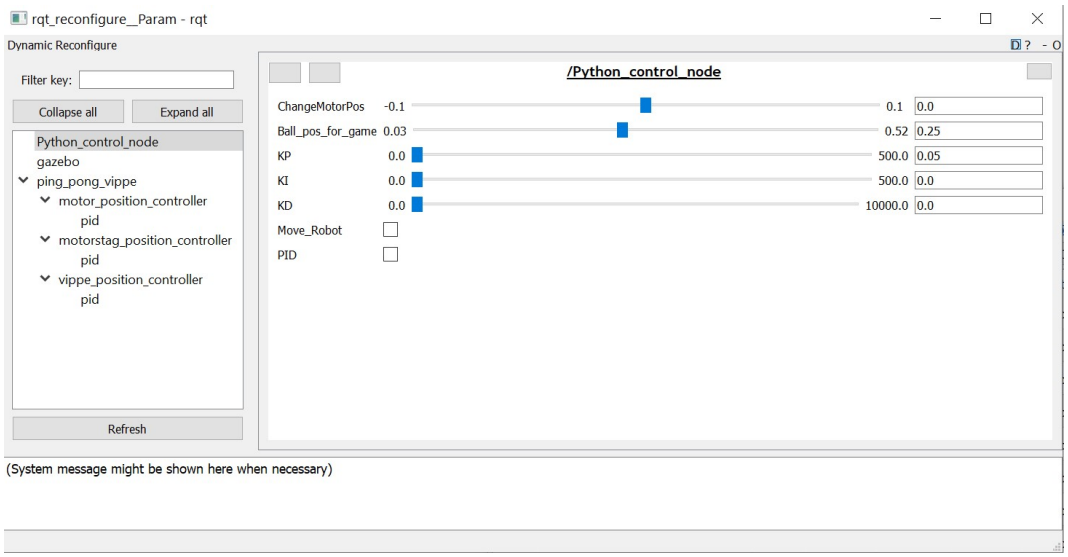
```
1   <?xml version="1.0" encoding="utf-8"?>
2 <launch>
3       <group ns="/ping_pong_vippe">
4             <!--laster in kontroller fra yaml fil-->
5             <rosparam command="load" file="$(find armbane)/config/joints.yaml"
                  />
6
7             <!--starter opp kontrollere-->
8             <node name="controller_spawner" pkg="controller_manager" type="
                  spawner"
9                   respawn="false" output="screen" ns="/ping_pong_vippe"
10                  args="--namespace=/ping_pong_vippe
11                  joint_state_controller
12                  vippe_position_controller
13                  motor_position_controller
14                  motorstag_position_controller
15                  --timeout 60">
16
17            </node>
18            <!--lager publishere til kontrollene-->
19            <node name="robot_state_publisher" pkg="robot_state_publisher"
                  type="robot_state_publisher"
20                  respawn="false" output="screen">
21                <remap from="/joint_states" to="/ping_pong_vippe/
                      joint_states" />
22            </node>
23      </group>
24
25 </launch>
```

I listing 4, lastes først *joints.yaml* inn og knytter dermed forbindelse med noden. *Controller-Spawner* starter så opp en *JointPositionController* for hver bevegelige *joint*, slik at disse kan motta kommandoer i sanntid. I tillegg opprettes en *JointStateController*, som er ansvarlig for å publisere status for hver *joint* til nodenettverket. Dette vises i linje 8-14. Til slutt publiseres modellens fellestilstand med *RobotStatePublisher*, vist i linje 19-22. Hensikten med alt dette, er å opprette noder hvor verdier kan settes og hentes i Gazebo. En kontroller tillater altså at bevegelse av modellen, kan utføres fra en hver plass i nodenettverket.

Videre i listing 2, opprettes *Python_control_node* for setting og henting av posisjonsverdier (linje 14-18). Nå som kontrollere er implementert, kan bevegelse av roboten manipuleres fra denne noden.

4.1.3 Implementering av sanntids parameterinstilling

Nå som modellen er implementert til Gazebo med kontrollerbare *joints*, er neste steg å realisere sanntids parameteroppdatering for posisjonsverdiene. Det Qt-baserte rammeverket *rqt* brukes for å endre og sette ulike parametre i et display. Pakken *dynamic_ reconfigure* muliggjør bruk av *rqt* sine tjenester i ROS, slik at modellen kan styres under drift. Figur 4.2 viser et display av de justerbare verdiene, samt ulike stater LinMot kan settes til under drift.



Figur 4.2: *rqt* display med sanntid parameter oppdatering

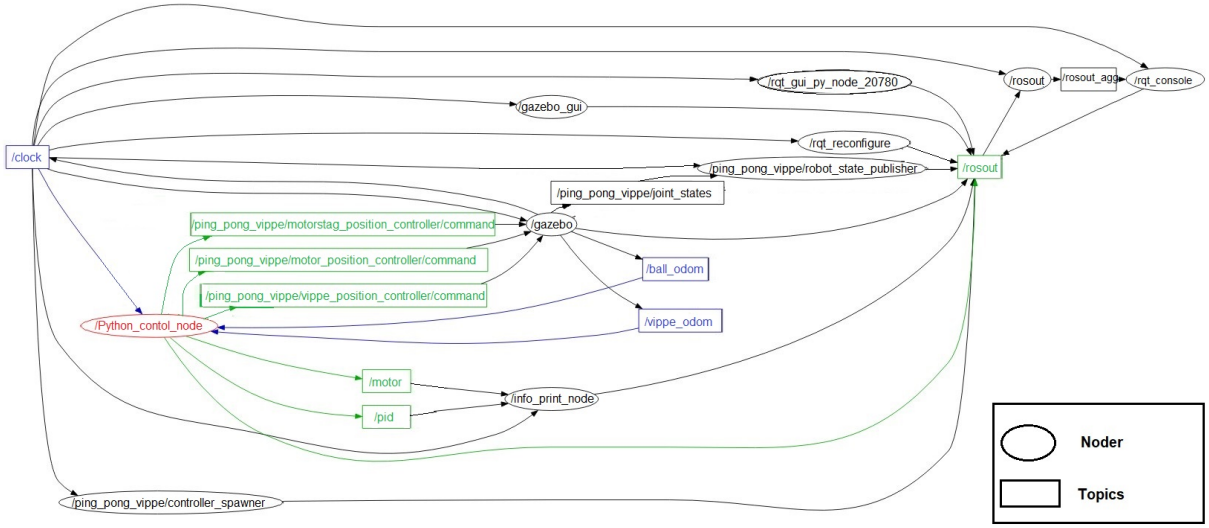
Display i figur 4.2 er en oversikt over parametere som kan endres, samt modus modellen kan settes til. K-parametre og PID modus, tilhører testprogram for balansering av ball med PID-regulator. Denne funksjonaliteten beskrives senere i rapporten.

I listen under vises en oversikt over funksjonaliteten i utviklet *rqt-display*.

- **ChangeMotorPos:** ——————> Velg motor posisjon
- **Ball_pos_for_game:** ——————> Velg posisjon ballen skal balanseres til (regulator)
- **KP:** ——————> Kp verdi (regulator)
- **KI:** ——————> Ki verdi (regulator)
- **KD:** ——————> Kd verdi (regulator)
- **Move_Robot:** ——————> Posisjonstesting modus
- **PID:** ——————> Balanserings modus (regulator)

4.1.4 Overordnet system i ROS Melodic

Til nå er vippemodellen med kontroller importert til Gazebo og klar for testing. Figur 4.3 viser oversikt over nodekommunikasjon i nodenettverket, for simulering av vippesystem i ROS Melodic. Videre vil en overordnet beskrivelse av nodenettverket kort oppsummeres.



Figur 4.3: Nodenettverk i prosjekt for simulering

Python control node står for setting av posisjonsverdier, til modellen i Gazebo. Noden mottar også ulike verdier fra Gazebo, vist med blå *topics*. Dette utgjør de visuelle bevegelsene i simuleringen. Noden sender også verdier til enet*info print node*, slik at systemet kan loggføre ønsket verdier.

Rosout vises helt til høyre i figur 4.3. Denne opptrer som et knutepunkt for logging av verdier i nettverket, og kan brukes for å hente ut enhver ønsket verdi fra de ulike nodene. Denne bidrar som et fellesregister, hvor alle noder kan hente informasjon. Blant annet vil *Pyton control node* motta verdiene innstilt i *rqt reconfigure*, for sanntids parameterinstilling. I tillegg kan all informasjon printes i *rqt console* fra *rosout*, for overvåkning av systemet.

Andre viktige komponenter i nodenettverket:

- ***Gazebo gui*:** Display for simuleringsprogrammet Gazebo.
- ***rqt reconfigure*:** Display for Sanntid parameterinstilling.
- ***Clock*:** Lager et tidsperspektiv på verdiene som sendes.
- ***rqt gui node*:** Display for visualisering av nodenettverk.
- ***Robot state publisher*:** Publiserer modellens fellestilstand til *rosout*.
- ***Controller spawner*:** Oppretter kontrollere og publiserer hver *joint* tilstand til *rosout*.

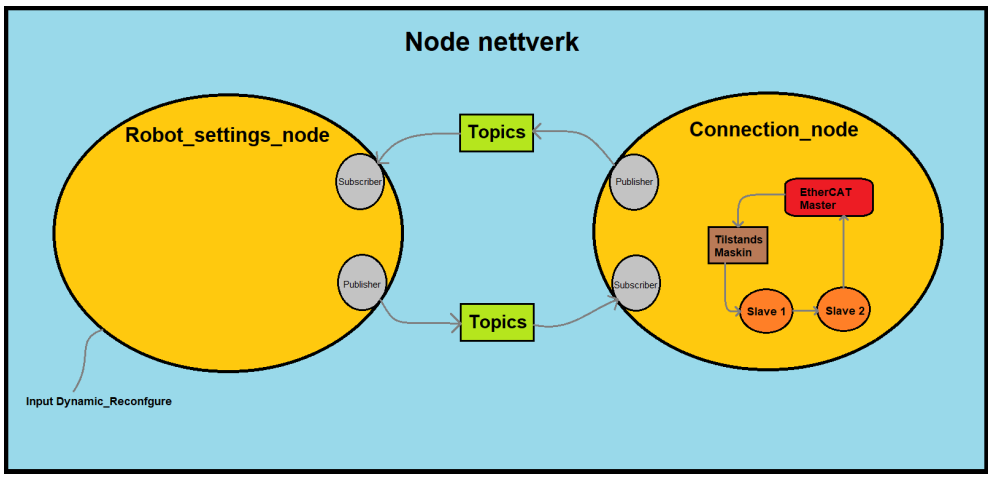
5 ROS-EtherCAT kommunikasjon

Ettersom det er opprettet en velfungerende virtuell simulering av vippesystemet, kan kommunikasjonsprotokollen via EtherCAT implementeres til ROS nettverket. Dette gjør det mulig å skape et oppsett for kontroll av det fysiske vippesystemet, i samspill med ROS funksjonalitet. Et slikt oppsett realiseres med å opprette en node, som tilrettelegger datautveksling mellom en EtherCAT-master og slaver. Denne noden må i tillegg, kommunisere med andre noder i nettverket via *topics*. Dette er med hensikt i å opprette en direkte datautveksling, mellom ROS og driver.

Den mest ryddige måten å utføre dette på, er å lage to noder i Python, med hvert sitt formål i kommunikasjonsprosessen. Ønsket funksjonalitet i nodene, oppnås ved bruk av følgende ROS-egenskaper:

- ***Publisher*:** ———> Ansvar for å sende informasjon.
- ***Subscriber*:** ———> Ansvar for å hente informasjon
- ***Topics*:** ———> Ansvarlig for frakting av data mellom noder
- **EtherCAT Master:** ———> Et senterpunkt for all EtherCAT kommunikasjon
- **Slaver:** ———> Enheter styrt av master
- **Tilstandsmaskin:** ———> Tilstandsmaskin for driver, med status oppdatering

Kommunikasjonen mellom nodene, samt realisering av EtherCAT protokollen, bygger på forkunnskaper om ROS og EtherCAT fra kapittel 2.5-2.7. Store deler av Morten sin modul er brukt for å oppnå PDO-utveksling mellom master og slaver, mens nodekommunikasjonen bygger på kapittel om ROS funksjonalitet.

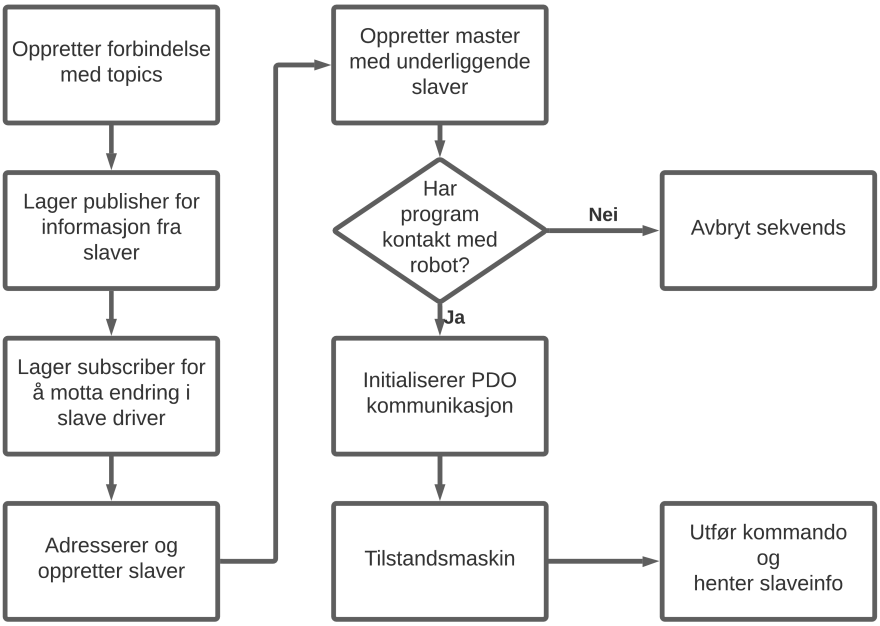


Figur 5.1: Noder for EtherCAT kommunikasjon

Figur 5.1 viser oversikt over nodene opprettet, for å oppnå datautveksling mellom ROS nettverket og drivere. *Connection_ node* har ansvar for å realisere EterCAT kommunikasjonsprotokollen, mens *Robot_ settings_ node* fungerer som et grensesnitt mellom EtherCAT kommunikasjonsprotokoll og ROS nettverket.

5.1 Node for realisering av EtherCAT protokoll

Connection_ node er ansvarlig for opprettelsen av kommunikasjonsprotokollen, samtidig som noden skal kunne hente og sende informasjon ut i nodenettverket. Et flytskjema er vist i figur 5.2, for å vise en oversikt over nodens funksjonalitet.



Figur 5.2: Sekvens for funksjonalitet i *Connection_ node*

5.1.1 EtherCAT-master

Masterens oppgave er å hente nødvendig informasjon om hver slave, samtidig som den oppretter en kontinuerlig kommunikasjons løkke for datautveksling. Hver gang master mottar data, vil hver verdi publiseres til en *topic*, spesifisert til verdien. Dette skjer ved hjelp av en *publisher*.

Funksjonaliteten til EtherCAT-master er definert i koden *SoemEtherCat.py*, laget av Morten. Dette er ved bruk av Python-pakken *pysoem* [39]. Master opprettes i *Connection_ node*, vist i listing 5. Masterens betingelser er utgang fra pc (*netif*), underliggende slaver og kommunikasjonshastighet for datautveksling og applikasjon.

Listing 5: Master (ref. Morten Mossige)

```
53          # Cat ip
54          netif = r'\Device\NPF_{62A01D95-A94C-4D74-86A3-4474A149D8A7}'
55          self.pub_GetNetif.publish(netif)
56          ecatloop = 10
57          apploop = 10
58
59          # Lager etercat master
60          etherCatMaster = SoemEtherCat.Create(netif, ecatloop/1000.0, apploop
                /1000.0, theEtherCatSlaves)
```

5.1.2 EtherCAT-slaver

Definert master har to underliggende slaver. Hver slave representerer hver av driverene brukt til å realisere systemet. Fra figur 5.1, representerer de to slavene følgene drivere:

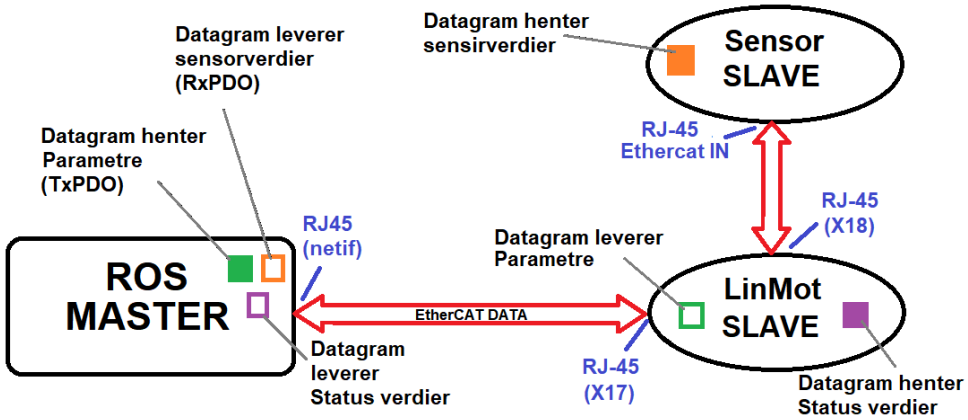
- **Slave 1:** LinMot driver *(LinMotSlave.py)*
- **Slave 2:** Sensor driver *(NXECC202.py)*

Hver av slavenes funksjonalitet er definert i egne skript, laget av Morten. Navnet til filene er vist i punktlisten over. Hver av slavene opprettes i *Connection_node* og legges så i en liste, for å samles under en master.

Listing 6: Slaver (Ref. Morten Mossige)

```
45          #Lager slaver.
46          rospy.loginfo('Creating EtherCat slaves:')
47          self.linMotSlave = LinMotSlave.Create()
48          self.inputSensors = NXECC202.Create()
49
50          #logger slaveinfo til navn.
51          theEtherCatSlaves= [ self.linMotSlave, self.inputSensors ]
```

LinMot driveren vil i korte trekk ha ansvar for å styre lineærmotoren og rapportere dens tilstand, mens sensor slavens oppgave er å rapportere ballens posisjon tilbake til masteren. I løpet av en EtherCAT syklus, vil master sende ut prosess data. Dermed settes ønsket hovedtilstand i tilstandsmaskinen, ved hjelp av kontroll ordet. Header i datagrammet sørger for å peke på hvilken kommando i undertilstand som skal utføres. Til slutt vil verdier hentet fra nodens *subscriber* til master, sendes som PDO for å tilfredsstille applikasjonens input verdier.



Figur 5.3: Forenklet kommunikasjonsmodell mellom master og slaver

LinMot driver rapporterer status til datagrammet "On The Fly", før den passerer videre til sensor slaven. Her hentes siste avstandsmålig, som datagrammet tar med seg videre og rapporterer til master. En forenklet oversikt over kommunikasjonen mellom slave og master, vises i figur 5.3.

6 Regulator

Å regulere prosessen er viktig for å få et kontrollert og presist resultat. Riktig valg av regulator er dermed avgjørende for kompensering mot forstyrrelse. Regulatorens formål i oppgaven er å oppnå og opprettholde en kontrollert bevegelse på ballen. Dette inkluderer forhindring av stor oversving, brå bevegelser og forsinkelse under balansering. Dermed oppsto følgende reguleringsproblemer:

Hvordan oppnå en stabil innsvingning av ballens posisjon til settpunkt?.

Hvordan oppnå en kontrollert ballbevegelse?

Løsning på reguleringsproblem foreslås ved bruk av to ulike reguleringsmetoder. Den første metoden, benytter seg av regulering basert på negativ tilbakekobling. Dette muliggjør en reguleringsprosess basert på feilbidrag, for at ballen skal oppnå stabil innsvingning. I metode 2, benyttes regulering basert på prediksjon. Dette er for å oppnå kontrollerte bevegelser, ut ifra ballens hastighet i et punkt. Hver metode brukt, inkluderer en redegjørelse for reguleringsystemet, innstilling av parameterverdier og beskrivelse av eksperimentelle oppsett.

6.1 Realisering av PID-regulator med tilbakekoblingsmetoden

I kapittel 2.8, forklares oppbygningen av en PID-regulator. For å realisere en slik prosess i en ROS-node, ble Pythons standard PID-regulator *simple pid* tatt i bruk [40]. Denne er basert på ligning 2.14. En viktig del av reguleringsprosessen, er valg av regulator parametre. Dette er nødvendig for å oppnå best mulig resultat. Videre i delkapittelet vil fremgangsmåte for parameter innstilling forklares. Det vil bli utført tester på tre ulike baller, med ulike fysiske egenskaper.

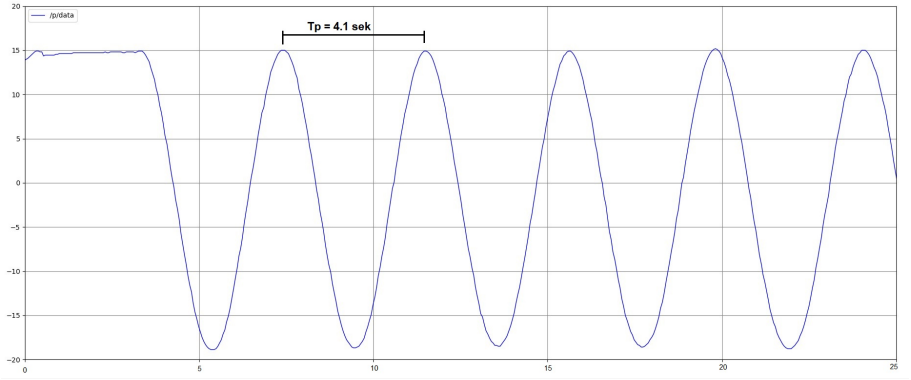
- **Pingpong ball:** Ikke massiv, glatt overflate
- **Golf ball:** Massiv, ujevn overflate
- **Baseball:** Massiv, større dimmensjon, svært ujevn overflate

Figur 6.1: Ulike baller brukt i testing

Videre vil pingpong ballen brukes for å illustrere valg av parametre, ved bruk av "prøve og feile" metoden. Resultat fra de andre ballene presenteres i resultat.

6.1.1 Valg av K_p

For å finne en god reguleringsforsterkning K_p, settes først $T_i = \infty$ og $T_d = 0$. Dette vil sette I og D leddenes bidrag lik 0. Ved bruk av denne innstillingen, kan man undersøke ballens respons ved forskjellige pådragsverdier. Målet var å få en K_p-verdi som gir en stabil svingning med fast amplitude. Dette vises i figur 6.2.

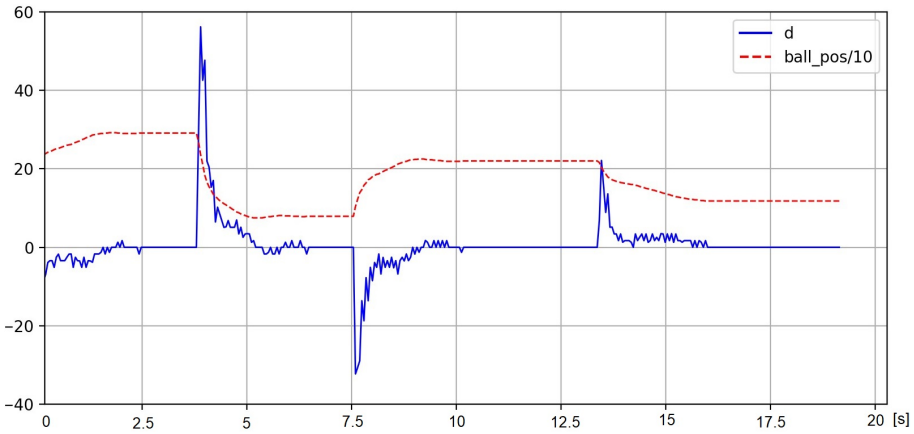


Figur 6.2: Innstilling av P-ledd

6.1.2 Valg av K_d

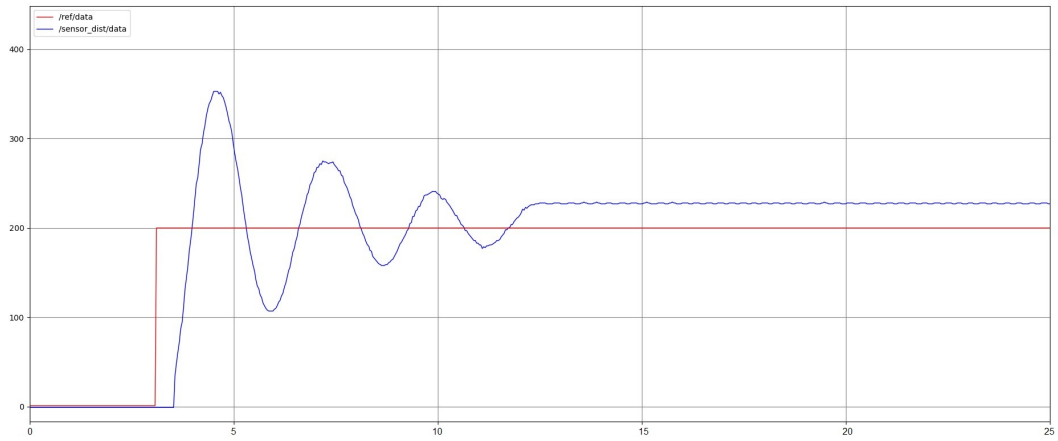
Ved en harmonisk svingning, vil ballens posisjon aldri oppnå stabilitet. Dermed er det viktig å ha et godt derivasjons-ledd. Dette leddet vil derivere ned avviket og dermed innregulere prosessen. K_d vil være en funksjon av K_p og derivasjons tidskonstanten T_d.

$$K_d = K_p \cdot T_d \tag{6.1}$$



Figur 6.3: Instilling av D-ledd

D-leddet fokuserer på farten ballen beveger seg i. Et godt D-ledd vil raskt derivere ballens hastighet ned til 0, slik at posisjonsverdien blir stasjonær, samtidig som stabil innregulering opprettholdes. Figur 6.3 viser systemets respons, under posisjonsendring av ballens settpunkt med en PD-regulator.



Figur 6.4: PD-regulator

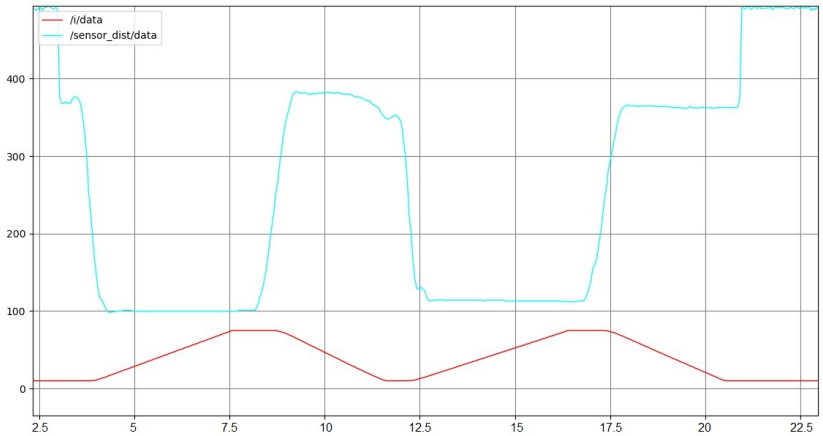
Problemet med dette er at ballen blir regulert til stasjonær tilstand, men ikke til riktig stasjonært settpunkt. Dette visualiseres i figur 6.4. For høye verdier på D-ledd, kan også resultere i ekstremt brå og raske bevegelser. For å unngå slitasje på lineærmotor, bestemmes T_d på grunnlag av en god harmoni mellom innreguleringstid og stabilitet.

6.1.3 Valg av K_i

K_i vil være en funksjon av forsterkningen K_p og integrator tidskonstanten T_i. Denne kan beskrives som vist i ligning 6.2. Formålet med I-leddet er å integrere opp avviket. Dersom ballen stabiliserer seg i feil posisjon som vist i figur 6.4, vil D-leddet sørge for at avviket blir derivert ned til 0. En slik respons medfører at vippen ikke har mulighet til å justere ballen til rett posisjon. Dette kan kompenseres for, ved bruk av et I-ledd.

$$K_i = \frac{K_p}{T_i} \tag{6.2}$$

K_i velges ved å først sette $T_d = 0$, dermed settes en ønsket posisjonsverdi som ballen skal plassere seg etter. Ballen holdes rolig et stykke unna ønsket posisjon. Dette medfører at I-leddet integrerer opp avviket, avhengig av ballens virkelige posisjon, relativt til ønsket posisjon.



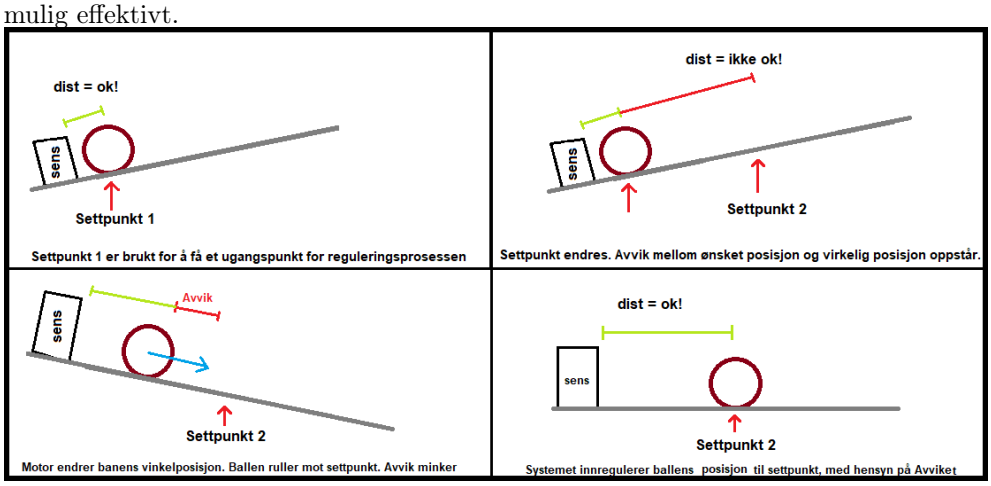
Figur 6.5: Instilling av I-ledd

I figur 6.5 er ønsket posisjon lik 200. Plottet av I-ledd viser da at pådraget integreres raskere når ballen er plassert på posisjon = 375, enn når posisjon = 100. Dette skyldes at pådrag i integrator blir større, relativt til avviket.

6.1.4 Eksperimentelt oppsett

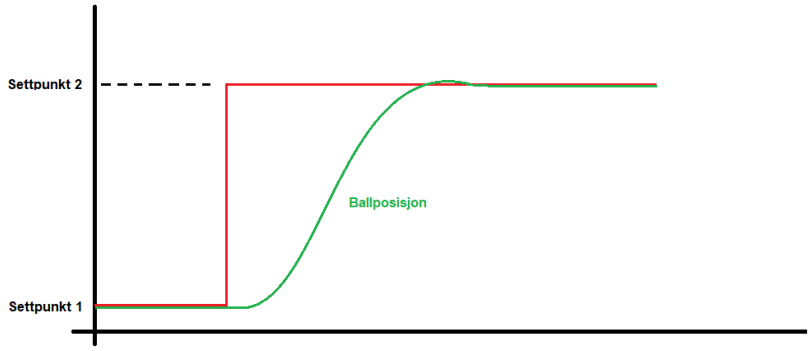
En god strategi for testing av reguleringssystemet, vil gi et godt utgangspunkt for å presentere resultater. For å kunne fremstille et godt resultat, med løsningforslag til reguleringsproblem, må parametre finjusteres under testing.

Parameter innstillinger utført til nå, gir et godt utgangspunkt for ønsket respons. Ved å utføre eksperiment med fokus på stabilitet og innreguleringstid, er det mulig å finjustere parameterne, slik at systemet gir optimal respons for hensikt. Som vist i figur 6.6 er prosedyren for testing et settpunkts skifte. Systemet skal innregulere ballens posisjon fra settpunkt 1, til settpunkt 2, mest mulig effektivt.



Figur 6.6: Testoppsett av system med PID-regulator

Et slikt eksperiment vil gi en sprangrespons, ved settpunktsskiftet. Det er ønsket å oppnå en posisjonsgraf for ballens bevegelse, med tilnærmet kritisk dempningsfaktor. En visualisering av ønsket resultat vises i figur 6.7. Ved å utføre flere tester, kan man undersøke de ulike parameternes påvirkning av responsen. Deretter finjustere hver parameter, slik at resultatet gir en respons som er tilnærmet denne.

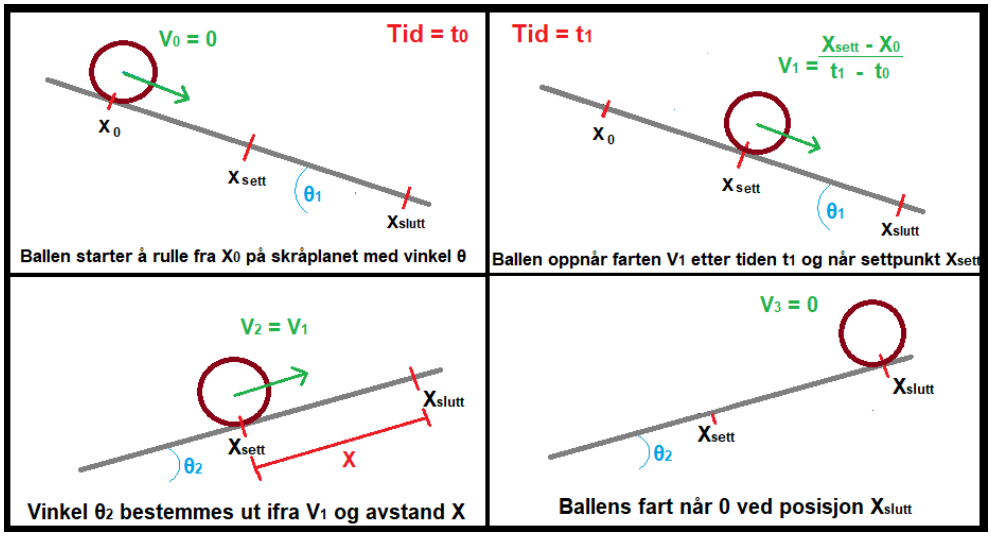


Figur 6.7: Ønsket respons, ballposisjon

6.2 Realisering av prediktiv regulering

For å tilfredsstille reguleringsproblemet for metode 2, brukes en regulator basert på prediktering av ballens bevegelse. Ved hjelp av denne type regulator kan systemet forutse ballens fremtidige posisjonering, ut ifra et målingspunkt. Her brukes den momentane farten til ballen i målingspunktet for å avgjøre helningsvinkelen på banen.

Ut ifra figur 6.8, er hensikten å redusere ballens fart til 0 i et ønsket punkt. Dette skal utføres med en enkel endring av vinkel på vippen. Ved å måle ballens fart i et målingspunkt, er det mulig å forutse ballens fremtidige bevegelse. Vinkel θ må dermed settes til en verdi, som korresponderer med ønsket bevegelse.



Figur 6.8: Eksprimentelt oppsett prediksjon

6.2.1 Beregning av ballens bevegelse

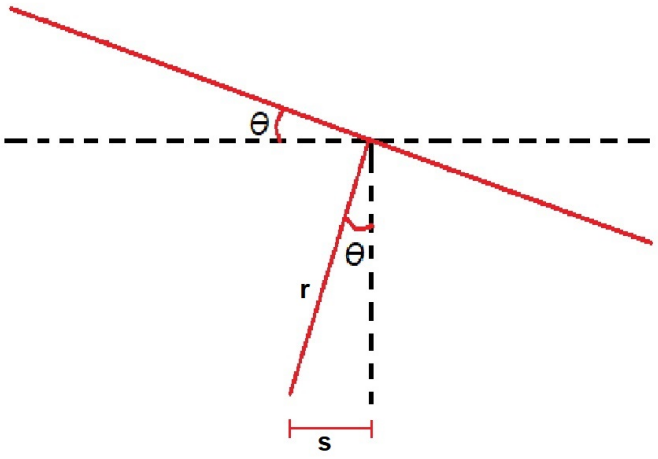
Ut ifra ligning 2.5 blir forholdet mellom endringen i distanse og endring i tid, brukt for å måle ballens fart under drift. I eksperimentet er det mest gunstig å måle farten ut ifra de to siste målingene. Dette gir formelen brukt for beregning av farten, vist i ligning 6.3.

$$V = \frac{\Delta d}{\Delta t} \qquad d \longrightarrow \textit{Distansen ballen } \textbf{har } \textit{trillet} \tag{6.3}$$

Ut ifra farten V og den ønska distansen ballen **skal** trille x, bestemmes ny helningsvinkel θ_2, se figur 6.8. Vinkel θ_2 kan finnes ut ifra ligning 2.5, og beskrives som en funksjon av farten og distansen til stoppepunkt. Denne formelen brukes til å sette vinkel som korresponderer med den momentane farten, vist i ligning 6.4.

$$\theta_2 = \arcsin\left(\frac{V^2 \cdot (1+c)}{2 \cdot g \cdot x}\right) \tag{6.4}$$

Da det var usikkerhet om valg av rullefriksjons koeffisienten c, til en pingpong-ball i kontakt med to flater på metallbanen. Det ble et det tatt et valg om å prøve ulike verdier, for å oppnå best mulig resultat. Ut ifra flere tester viste det seg at c = 1.9, var optimal for vippesystemet. Motoren har ansvar for å forskyve staget en lengde som tilsvarer utregnet vinkel θ_2. Her brukes læren om formlikhet i systemet for å sette ønsket stagforskyvning.



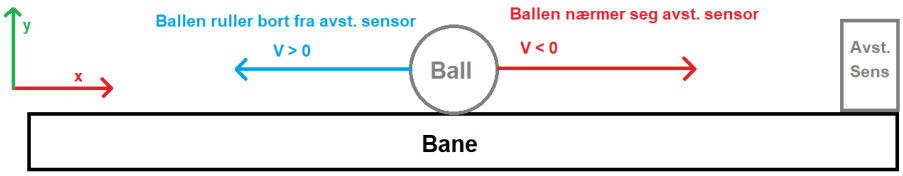
Figur 6.9: Formlikhet i vippesystem

I beregningspunktet X_{Sett} fra figur 6.8, beregnes helningsvinkel θ_2, som er nødvendig for å stoppe ballen i stoppunkt X_{Slutt}. Motorpådraget S vises i figur 6.9. Den bestemmes av ønsket θ_2, samt avstand fra stag festningspunkt og vippeleddet til banen, som er målt til $r = 190mm$.

Motorposisjon vil være satt $S = 34mm$ i stasjonær tilstand, når vippevinkel $\theta = 0$. Fra ligning 6.4 vil θ alltid være positiv, selv om farten kan gå i negativ retning i forhold til banen. Dette setter grunnlaget for initial betingelsen 6.5.

$$\begin{aligned} V > 0 \quad &\longrightarrow \quad S = 34mm - 190mm \cdot sin(\theta) \\ V < 0 \quad &\longrightarrow \quad S = 34mm + 190mm \cdot sin(\theta) \\ V = 0 \quad &\longrightarrow \quad S = 34mm \end{aligned} \tag{6.5}$$

Fartens fortegn bestemmes av hvilken retning ballen triller på banen. Dette vil føre til at farten, grafisk sett vil veksle mellom positiv og negativ verdi omkring nullpunktet. Når farten treffer null, vil dette tilsi at ballen har nådd et endepunkt.

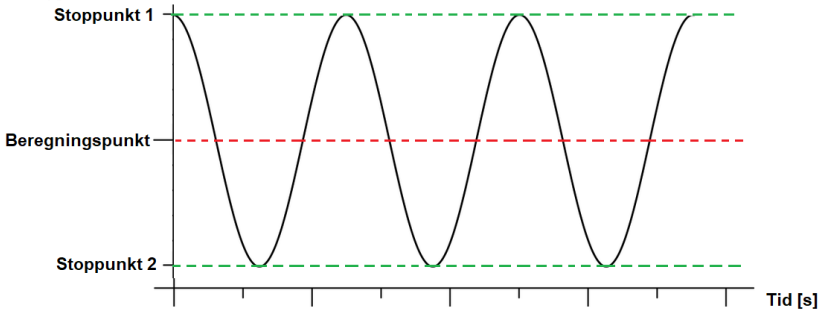


Figur 6.10: Fortegn for fart i forhold til banen

For å realisere en slik reguleringsmetode, opprettes en prediktor node, se figur 6.12. Her kalkuleres og settes alle verdier for styring i *Predict_ script.py*. Siden Ballen aldri vil stå stille (V = 0) i beregningspunktet, kan denne betingelsen neglisjeres fra koden.

6.2.2 Realisering av ekspriment

Formålet med eksperimentet er at ballens posisjonsgraf skal simulere en sinuskurve. Figur 6.8 viser det grunnleggende eksperimentelle oppsettet for å oppnå sinuskurvens amplitude. For å etablere god kontroll av ballens posisjon, brukes tre punkter på banen. To av punktene brukes som ønsket stoppunkt, her vil farten være lik 0 og vil visuelt sett være et topp/bunnpunkt i sinuskurven.



Figur 6.11: Forventet resultat av eksperiment

For å oppnå $V = 0$ i stoppunktene, bør det brukes et beregningspunkt mellom stoppunktene. Her skjer alle beregninger under drift, slik at systemet kan kalkulere en motor stagforskyvning, som tilsvarer en ønsket helningsvinkel θ. Dette medfører at ballen vil rulle opp et skråplan, med krefter som jobber mot ballens hastighet, som etterhvert vil akselerere ned farten til 0, etterfulgt av at ballens bevegelse endrer retning på banen.

6.3 Begrensninger og forutsetninger

For å unngå slitasje på vippeoppsettet, med posisjonsverdier utenfor de ulike bevegelige leddenes rekkevidde og brå bevegelser, må enkelte begrensninger bli tatt hensyn til. En vippemodell med ukontrollerte bevegelser kan resultere i en katapult. Dette vil ikke bare ødelegge komponenter i modellen, men kan også være en fare i samsvær med HMS. I tillegg kan mye risting og brå bevegelser føre til slitasje over tid. Dette er noe gruppen ønsker å unngå.

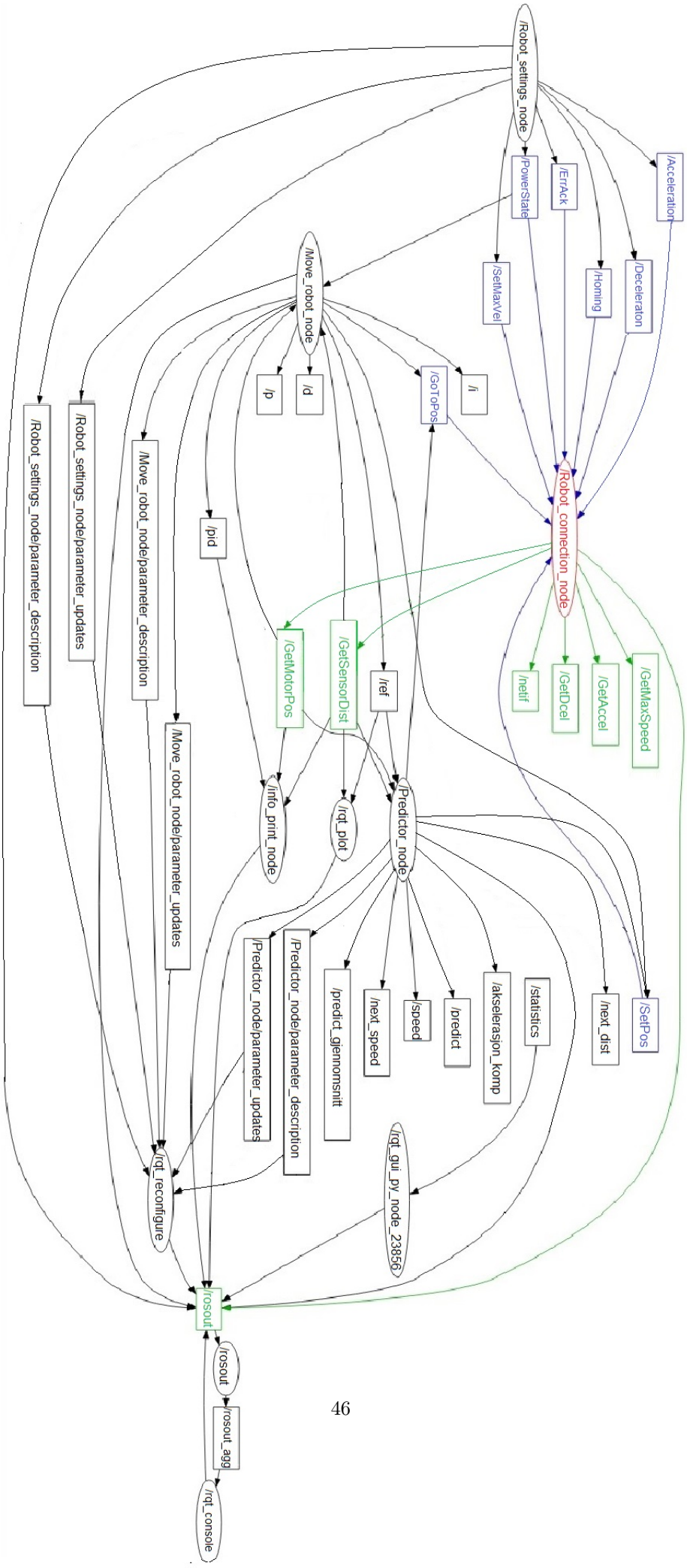
Begrensninger:

- Motorstag kan ikke overstige øvre 90mm og nedre 0mm forskyvning under predikiv regulering.
- Motorstag kan ikke overstige øvre 75mm og nedre 10mm forskyvning under PID-regulering.
- Bruker lavt pådrag for å unngå brå endringer ved stort posisjonsavvik.
- Bruker sampling rate 50 Hz for å unngå risting.

Forutsetninger:

- Systemet vet hvilken posisjon motorstaget er, til en hver tid.
- Ballens hastighet i en retning kan påvirkes i større grad ved endring av vippevinkel, når ballen er posisjonert nær enden av banen.
- Ved vippevinkel $\theta = 0$, er motorstag S = 34mm.

6.4 Overordnet system i ROS Noetic



Figur 6.12: Overordnet system for nodekommunikasjon i ROS

Over vises en oversikt over nodenettverket i ROS Neotic. Dette representerer all kommunikasjon mellom nodene, opprettet for å realisere styring av det fystiske vippesystemet via EtherCAT. Videre forklares sammenhengen i nodenettverket på et overordnet nivå.

Robot_connection_node:
Denne noden er markert med rødt i nodenettverket og har som oppgave å starte en kommunikasjonsprotokoll mellom EtherCAT master og slaver. Data fra hver slave kan hentes ut via *topics*, vist med grønne rektangler. *GetSensorDist* og *GetMotorPos*, er data som hentes av hver applikasjonsnode for styring.

Robot_settings_node:
Denne noden sørger for at *Robot_connection_node* mottar informasjon for styring av lineærmotor. Her innstilles verdier som hastighet og akselerasjon. Det er også mulighet for å sette noen av de operative tilstandene i tilstandsmaskinen, som *Power State* og *Homing*. Disse verdiene settes i *rqt_reconfigure*.

Move_robot_node:
Ved hjelp av denne noden realiseres PID regulering med tilbakekoblings metoden. Her utføres en toveiskommunikasjon med *Robot_connection_node*, hvor sensormålinger og motorposisjon hentes, og ønsket motorposisjons verdier leveres. Parametere for styring i sanntid, utføres i *rqt_reconfigure*.

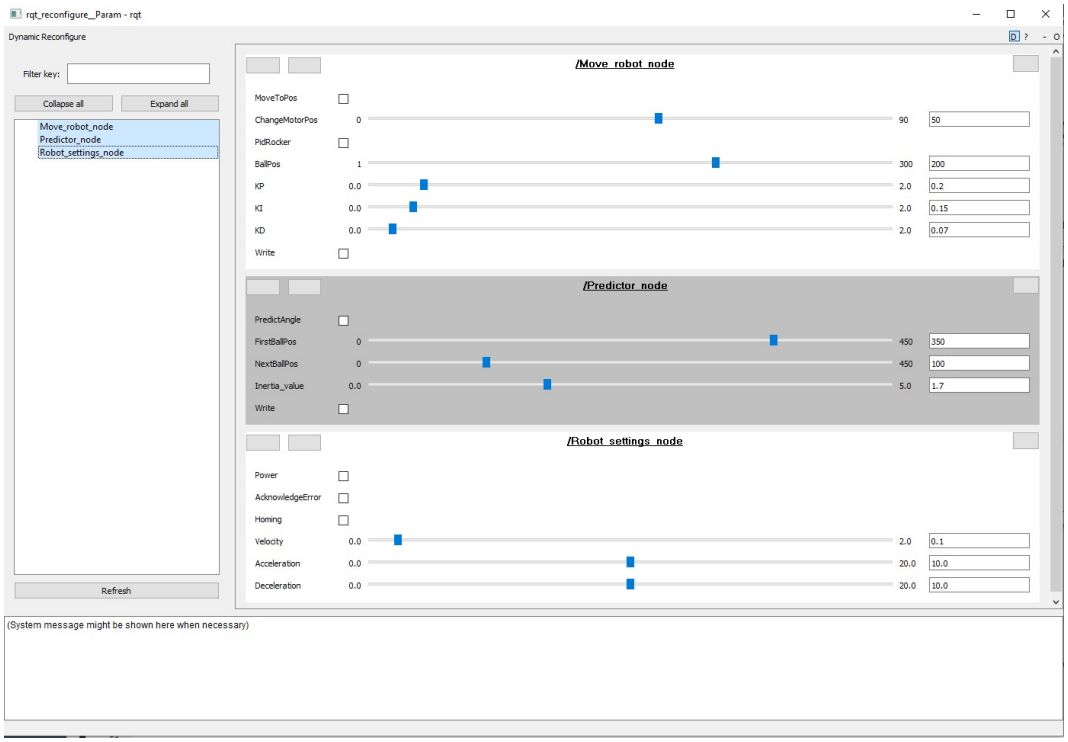
Predictor_node:
Predictor_node er ansvarlig for den prediktive reguleringen i systemet. Denne noden mottar data fra samme *topics* som *Move_robot_node*, men har ulik funksjonalitet internt. I tillegg til å sende motorposisjons verdier til *Robot_connection_node*, vil også noden sende informasjon om ønsket stoppunkt i eksperimentet. Dette kan utføres fra brukergrensesnittet *rqt_reconfigure*.

info_print_node:
For å overvåke systemet under drift, mottar *info_print_node* informasjon om sensor distanse, motor posisjon og pådrag (pid). Dersom man ønsker å hente andre verdier, kan et hvert *topic* knyttes til denne noden.

rqt_plot:
Denne noden er ansvarlig for å skape en grafisk visualisering av data som mottas fra *topics*. Dersom man ønsker å hente andre verdier, kan et hvert *topic* knyttes til denne noden. Dermed kan man enkelt analysere respons på ulike aspekter i systemet.

rosout:
Rosout fungerer som knutepunktet for logging av data i i nodenettverket. Mer info om rosout i kapittel 4.1.4.

rqt_reconfigure:
Som *rqt_plot*, er også *rqt_reconfigure* et visualiserings verktøy. Med *rqt_reconfigure* kan man enkelt endre parametre i sanntid. Det er også mulighet for å velge om *Move_robot_node* eller *Predictor_node* skal være aktiv. Dette vinduet er vist i Figur 6.13



Figur 6.13: *rqt_ reconfigure*, display for sanntids paramenteroppdatering og valg av modus.

For at systemet skal kunne starte opp, må alltid *Power* modus være aktivert. Her sendes også data fra *Robot_ settings_ node* til *Robot_ connection_ node*, vist helt nederst i figur 6.13. For kjøring av prediktiv regulering, må *PredictAngle* aktiveres. Her velges begge endepunktene, og sendes til *Robot_ connection_ node*.

PidRocker initialiserer PID regulering. Her velges ønska forsterknings verdier, som videre sendes til *Robot_ connection_ node*. Dersom man ønsker å bevege vippen manuelt, kan *MoveToPos* aktiveres og motorposisjon velges.

Write modus vist i display for *Move_ robot_ node* og *Predictor_ node*, gir mulighet for å automatisk loggføre ønska verdier for den gjeldene noden, til et tekst dokument. Verdiene printes i en liste og kan brukes for å analysere tester. Dette var nødvendig, da grafisk fremstilling av flere plott samtidig, ikke alltid var synkronisert tidsmessig.

Hvert *topic* i overordnet system, kan enkelt lenkes opp med en hver node. Dersom man ønsker å lage en grafisk fremstilling av ballens målte hastighet, kan det opprettes en *subscriber* for *rqt_ plot* noden. Dermed mottar denne noden data fra *speed topic*.

Del III

Resultat

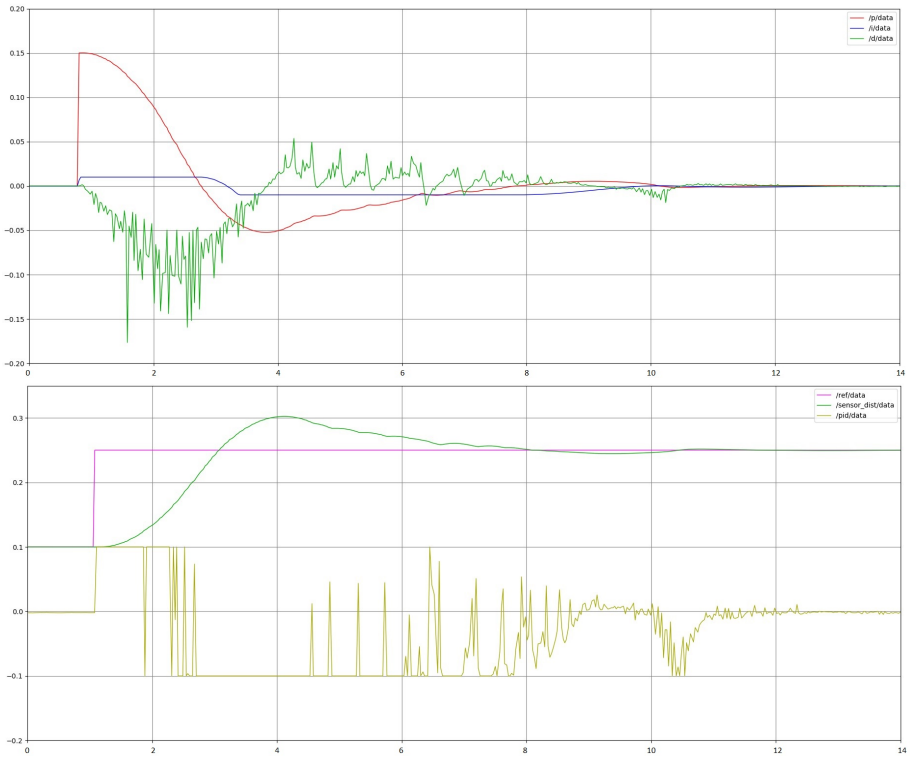
Her presenteres resultatene av eksperimentene. Det blir satt fokus på objektive, målbare resultater. Tester under ideelle forhold i Gazebo, sammenlignes med tester fra det fysiske systemet, hvor ytelse og presisjon er sentralt. Det blir også satt søkelys på ulike feilbidrag i systemet, samt hvordan disse påvirker ulike resultater.

7 Eksperiment regulator

Før testing av reguleringssystemet på lineærmotor, ble det utført flere tester i virtuell simulering. Dette skapte et utgangspunkt for forventet resultat. Det var også viktig å bruke simulering for å bestemme begrensninger, og dermed unngå slitasje på komponenter.

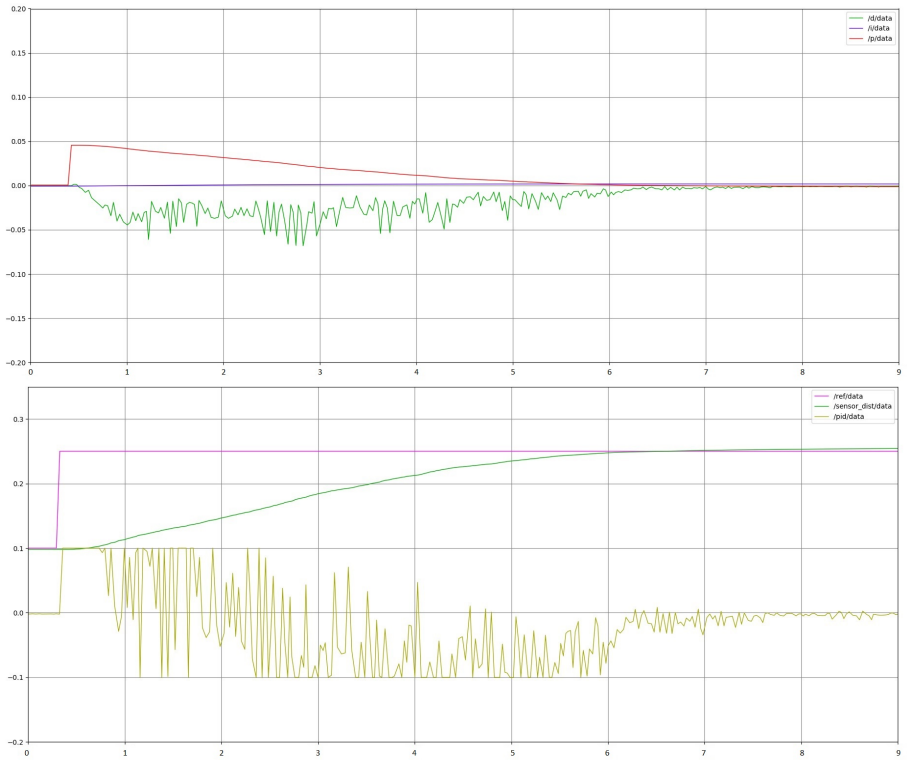
7.1 Eksperiment PID-regulering i Gazebo

For den grunnleggende testingen av balanseringsfunksjonaliteten i systemet, ble PID-regulator implementert til styring av modellen i Gazebo. Dette lager et "ideellt" utangspunkt, som responsen til det fysiske systemet kan strekke seg etter. Som en start, utføres en test, der $K_p = K_i = K_d = 1$. Dette gir en respons som kan forbedres, se figur 7.1.



Figur 7.1: PID-regulering første utkast i Gazebo

Plottet viser en innsvingningstid på omtrent 10 sekund, samt en oversvingsfaktor som tilsvarer et underdempet system. Det er ønsket å få et resultat som tilnærmes et kritisk dempet system, hvor ballen ruller i en mer kontrollert bevegelse før den stopper i ønsket posisjon. Ved å justere de ulike parameterne, var det mulig å få et overdempa system, med mer stødig ballbevegelse.



Figur 7.2: PID i Gazebo, overdempet system ($K_p = 0.3$, $K_i = 0.005$, $K_d = 1.2$)

Dette ga en raskere innregulering og mer kontrollert ballbevegelse, men på bekostning av mer brå motorbevegelser. Denne informasjonen tas med videre som baktanke og resultatene settes som utgangspunkt for testing av fysisk modell.

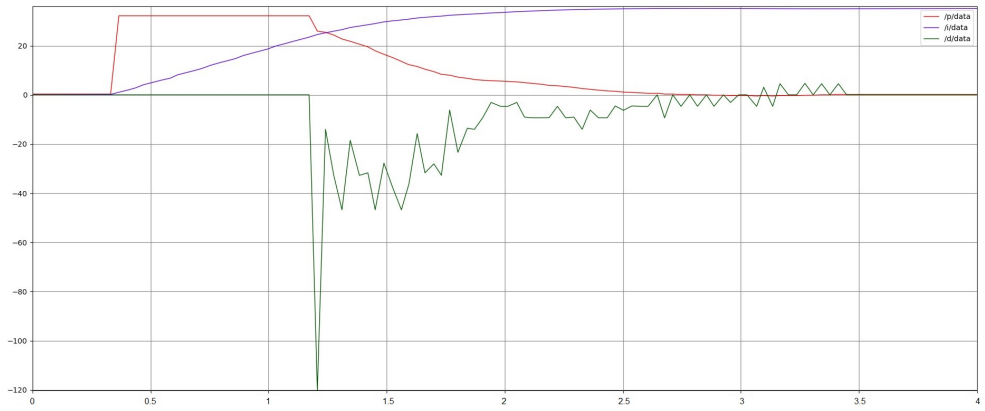
7.2 Eksperiment PID-regulator vippe

Objekter med ulike fysiske egenskaper, vil oppføre seg ulikt. For vippesystemet er dette intet unntak. Da ideelt system i Gazebo er friksjonsløst, samt ballens bevegelse i simulering vil være glidende og ikke rullende, var det interessant å se ulike objekters respons i systemet. En tung golfball med massen fordelt i hele volumet og ujevn overflate, vil oppføre seg annerledes enn en pingpong-ball med en jevn overflate overflaten. Dermed må ulike objekter, innstilles med ulike parametre. Dette er for å få best mulig resultat. Etter flere tester ble følgende parametre satt:

	K_p	K_i	K_d
Pingpong	0.16	0.145	0.145
Golf	0.17	0.1	0.125
Base	0.16	0.1	0.115

Pingpong-ball:

Pingpong-ballen vil være et bra testobjekt for reguleringen, da denne er lett og har en glatt overflate. Dette fører til at ballens rullebevegelse langs banen, vil skje med få forstyrrelser og lav friksjon. Det var forventet på forhånd at pingpong-ballen raskt kunne stabiliseres omkring ønsket posisjon, men vil ha problemer med å oppnå presis posisjon på grunn av vekten.

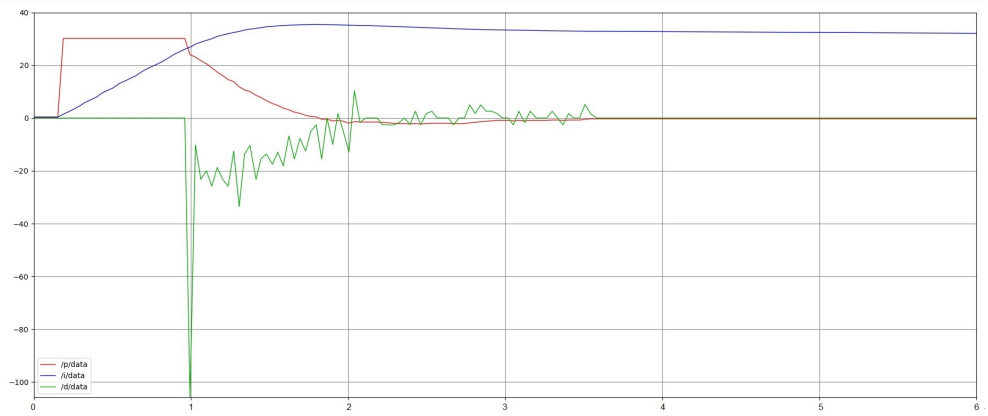


Figur 7.3: Pingpong ball: P, I og D plott

Figur 7.3 viser at D-leddet tidlig utfører brå endringer i pådrag. Dette fører til at regulatoren raskt innregulerer hastigheten til ballen. Det viser seg at hver minste bevegelse i vippen vil påvirke ballen i relativt stor grad, som forventet.

Golfball

Golfballen vil ha en annerledes oppførsel enn pingpong-ballen, da denne har en ujevn overflate. Siden massen ikke påvirker rullehastigheten, er det forventet at feilbidragene utgjør forskjellen mellom golfballens respons og pingpong-ballens respons.

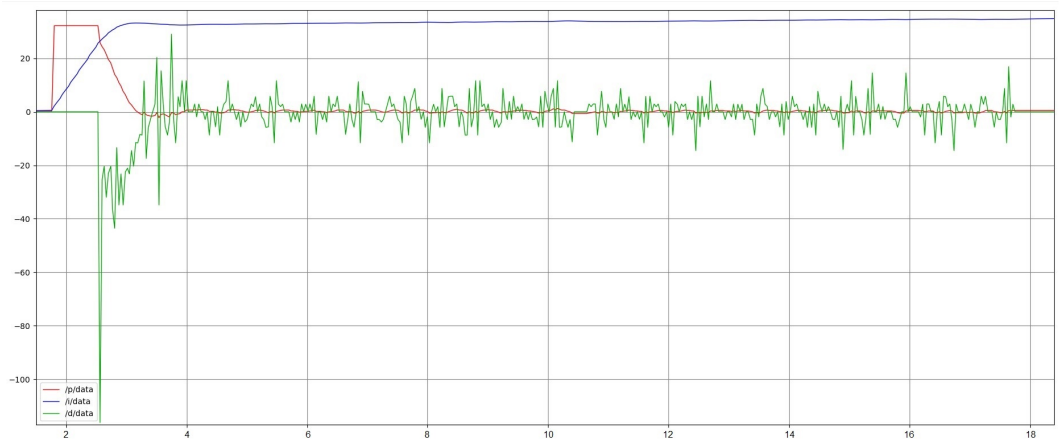


Figur 7.4: Golfball P, I og D plott

Det viser seg at responsen her gir lignende pingpong ballens respons, men med en litt tregere stabiliseringstid. Dette er forventet da, dimensjonen til disse to ballene er omtrent like. Forskjellen mellom resultatene kan komme av feilbidrag som luftmotstand, rullefriksjon og ujevnhet.

Baseball

Som et siste testobjekt, ble baseballen satt til test. Hensikten med dette var å se hvordan flere feilbidrag vil påvirke prosessen. Baseballen har store ujevnheter på sin overflate, i tillegg til at ballen er mye større og tyngre enn de andre testballene. Overflaten er laget av skinn, noe som gir ballen mer grep på banen. Med dette vil ballen ha helt andre friksjonsverdier.



Figur 7.5: Baseball P, I og D plott

Dette resulterte i at ballen aldri helt klarte å stabilisere seg i senter med 100% presisjon. Ballens overflate medførte at når ballen først begynte å rulle en retning, rullet den for langt. Dette resulterte i en gynge bevegelse rundt settpunkt. Dette visualiseres i figur 7.5, med stor ujevnhet i D-leddet.

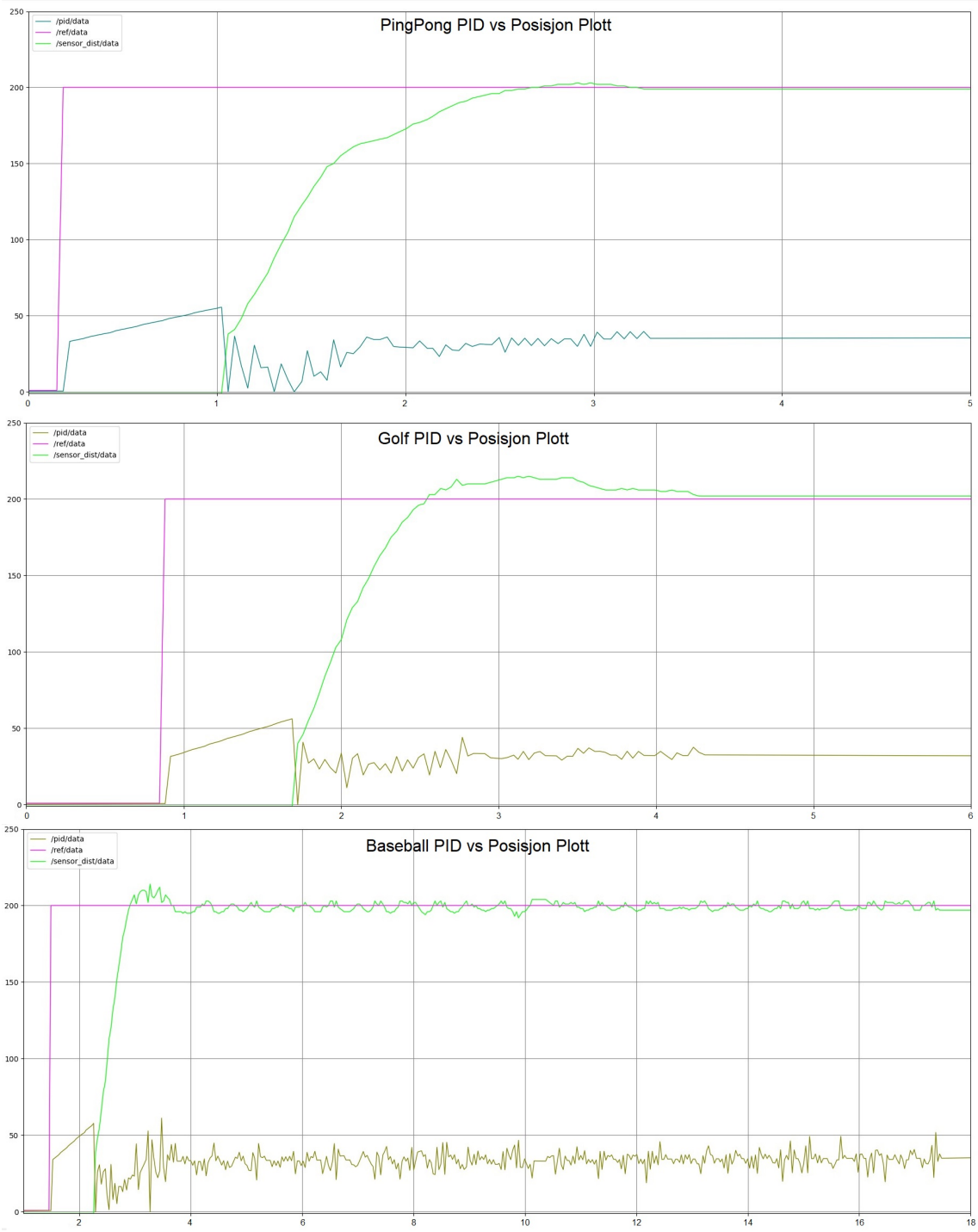
7.2.1 Resultat PID eksperiment

Figur 7.2 brukes her som et utgangspunkt for tester på fysisk vippe. Optimalisering av innsvingningstid og stabilitet, var sentralt under justering av parametre. Et mål var også at systemet skulle tilnærmes kritisk dempet. Resultatet av en enkel innsvingnings test, på hver av de ulike ballene, er vist i figur 7.6. Denne figuren refereres til videre.

Pingpong-ballen gir et tilfredsstillende resultat, da responsen har en tilsynelatende tilnærmet kritisk dempingsfaktor. Stabiliteten anses også som god, da posisjonsgrafen til ballen ikke antyder til noen brå bevegelser. Stabiliseringstiden fra sprang, til ballen stoppet på settpunkt er på ca. 3 sekunder. Dette vurderes som et godt resultat.

Golfballens respons viser noe høyere oversving, større akselerasjon tidlig i spranget og omtrent like lang stabiliseringstid. Dette viser hvordan ballens feilbidrag påvirker responsen, forutsett at golfballen har omtrent samme dimensjon som pingpong-ballen. Resultatet anses dermed bra, med en oversving som er innenfor akseptale grenser.

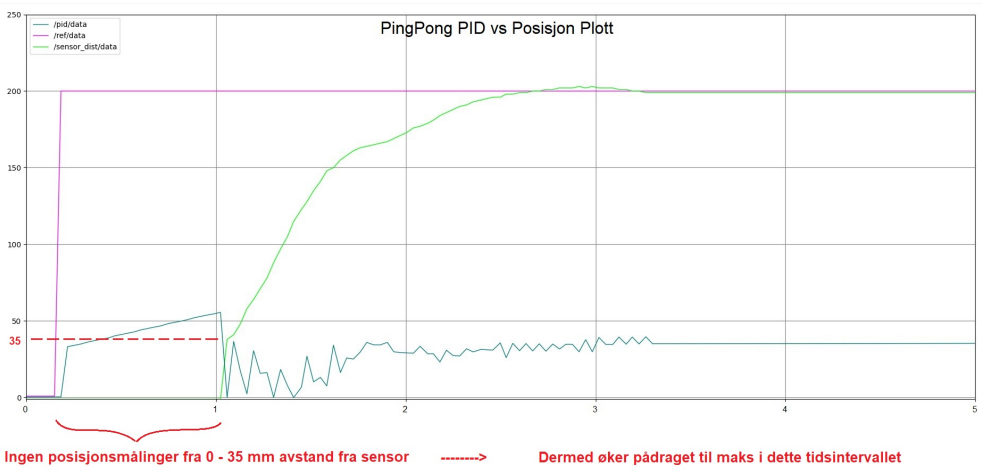
Baseballens respons anses som svært dårlig, da den aldri klarte å stabilisere seg. Dermed blir baseballen neglisjert fra videre utregning.



Figur 7.6: PID-regulator for ulike testobjekt

7.2.2 Feilbidrag fra avstandssensor

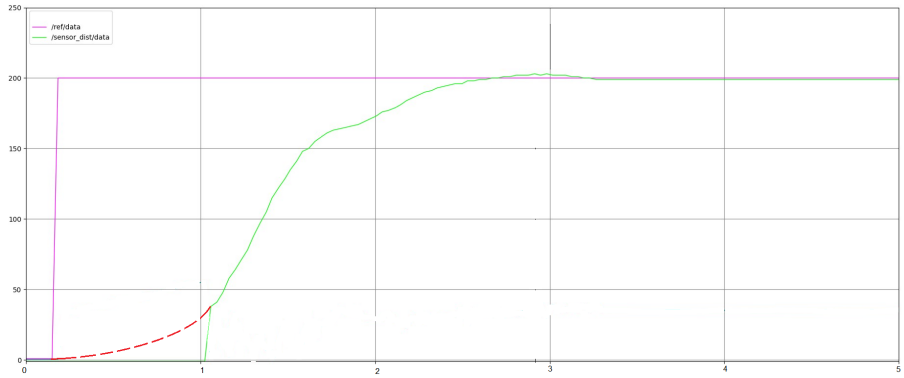
I alle grafene vises en tilsynelatende dødtid på ca. 1 sekund. Lenge ble dette sett på som en alminnelig forsinkelse i systemet. I etterklokskapens lys, kom det frem at dette var et feilbidrag fra avstandssensoren plassert på vippen.



Figur 7.7: Manglende avstandsmålinger i tidsintervall [0, 35]mm

Som vist i tabell for avstandssensor i kapittel 2.4.2, har sensoren en minimum rekkevidde på 35mm. Dette medfører at alle målinger i avstandsintervallet [0, 35]mm fra sensoren, anses som 0 i systemet. Som vist i figur 7.7, vil dette gi et uønsket utslag i pådrag og integrator, det første sekundet i prosessen. Dette vil også føre til at responsen kan ved første øyekast, se ut som en 1.ordens respons.

Når grafen registrerer sin første virkelige måling, vil målt posisjon stige fra 0mm til 35mm, iløpet av to målinger. Dermed blir derivatorens bidrag veldig stort her, og jobber på høygir for å prøve å kompansere for de feilaktige målingene. Et forventet resultat av responsen uten feilbidrag fra avstandssensor, vises med rød linje i figur 7.8.

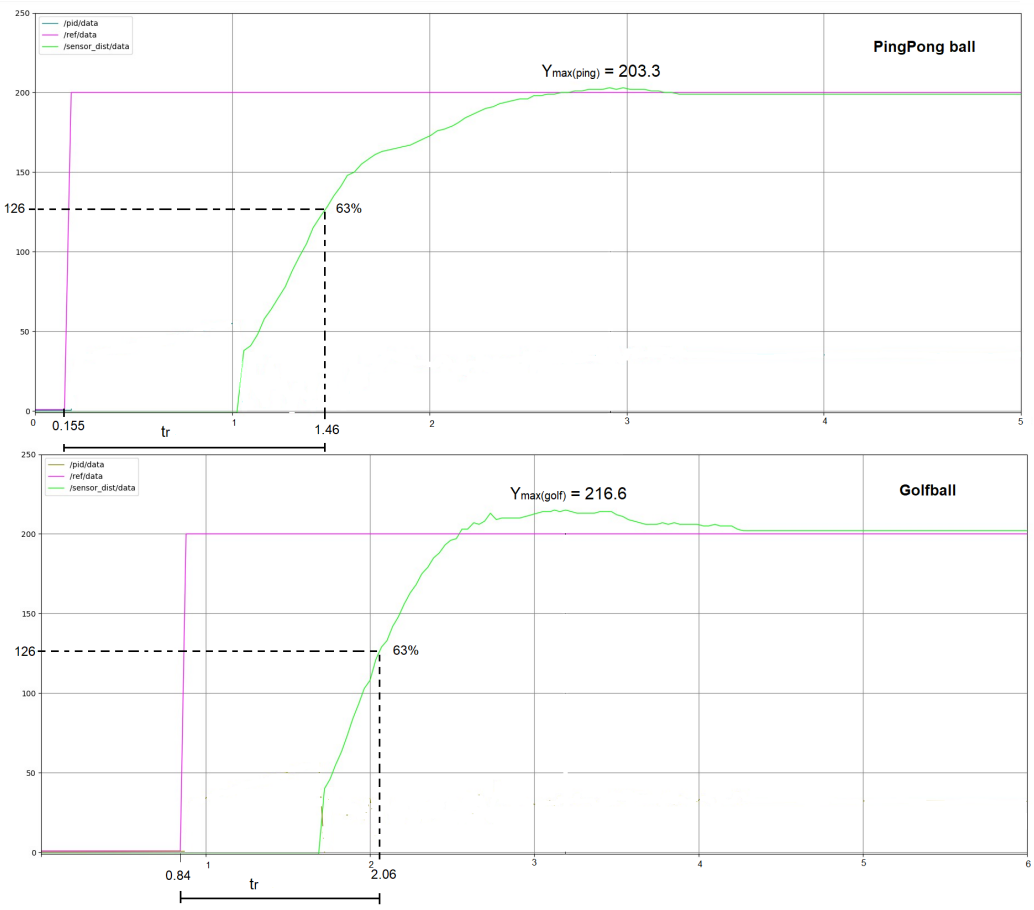


Figur 7.8: Forventet resultat uten feilbidrag fra sensor, som viser at responsen er av 2.orden

Siden dette ble oppdaget mot slutten av prosjektet, ble det tatt et valg om å bruke de opprinnelige målingene for videre analyse. Med baktanke om at dette påvirker regulatorens oppførsel.

7.2.3 Systemegenskaper og presisjon

Ved å analysere steg-responsen til systemet, for både pingpong-ballen og golfballen, kan man kartlegge egenskapene til responsene på en oversiktelig måte. Dermed kan man finne hvor stabilt og raskt systemet er.



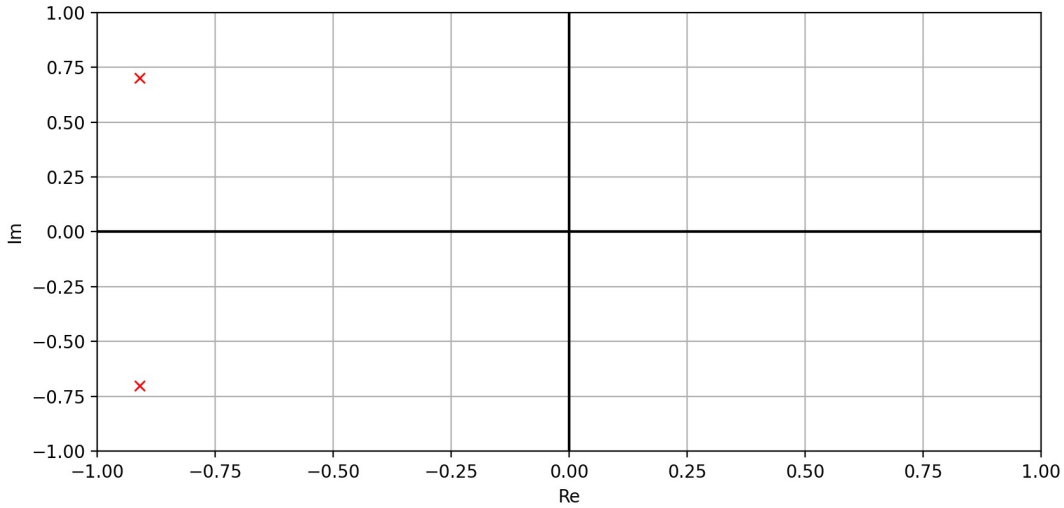
Figur 7.9: Steg-respons golfball og pingpong ball

I figur 7.9 vises steg-responsen til hver av ballene. Ut ifra disse resultatene, blir læren om 2.ordens overføringsfunksjon fra kapittel 2.8.4, brukt for å determinere systemets faktorer. En oversikt vises i tabellen:

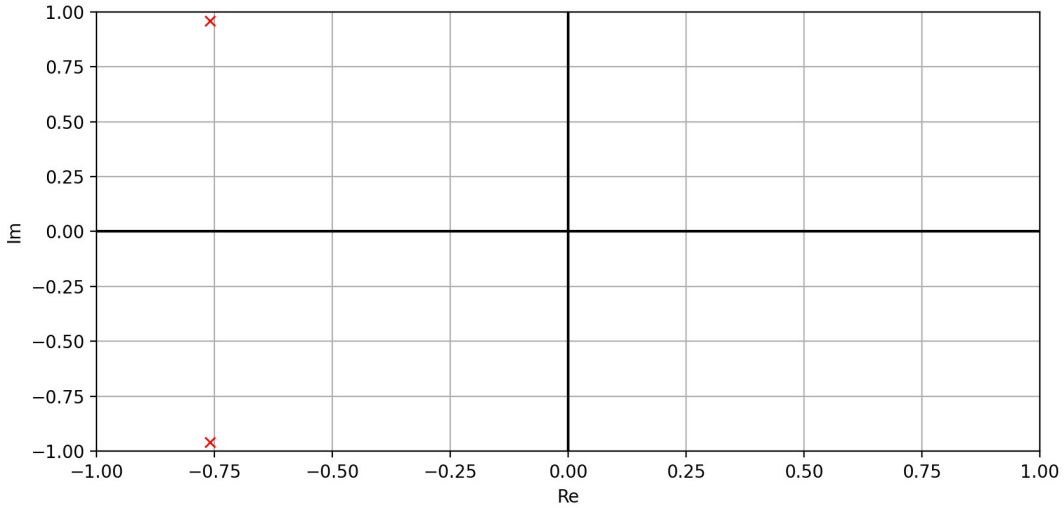
Navn	Y_{max}	δ	ζ	t_r [s]	ω_0	Dempning
Pingpong	203.3	0.0167	0.79	1.305	1.15	Underdempet
Golf	216.6	0.083	0.62	1.22	1.22	Underdempet
Hentet fra:	Fig. [7.9]	Lign. [2.16]	Lign. [2.16]	Fig. [7.9]	Lign. [2.17]	Kap. [2.8.4]

Ved å plotte verdiene inn i ligning 2.18, får man et resultat av de kopleks kojugerte polene s_1 og s_2 i hvert system.

$$\textbf{Pingpong:}\quad s_1, s_2 = -0.91 \pm 0.7j \qquad \textbf{Golf}: \quad s_1, s_2 = -0.76 \pm 0.96j \tag{7.1}$$



Figur 7.10: Pingpong ball: Poler



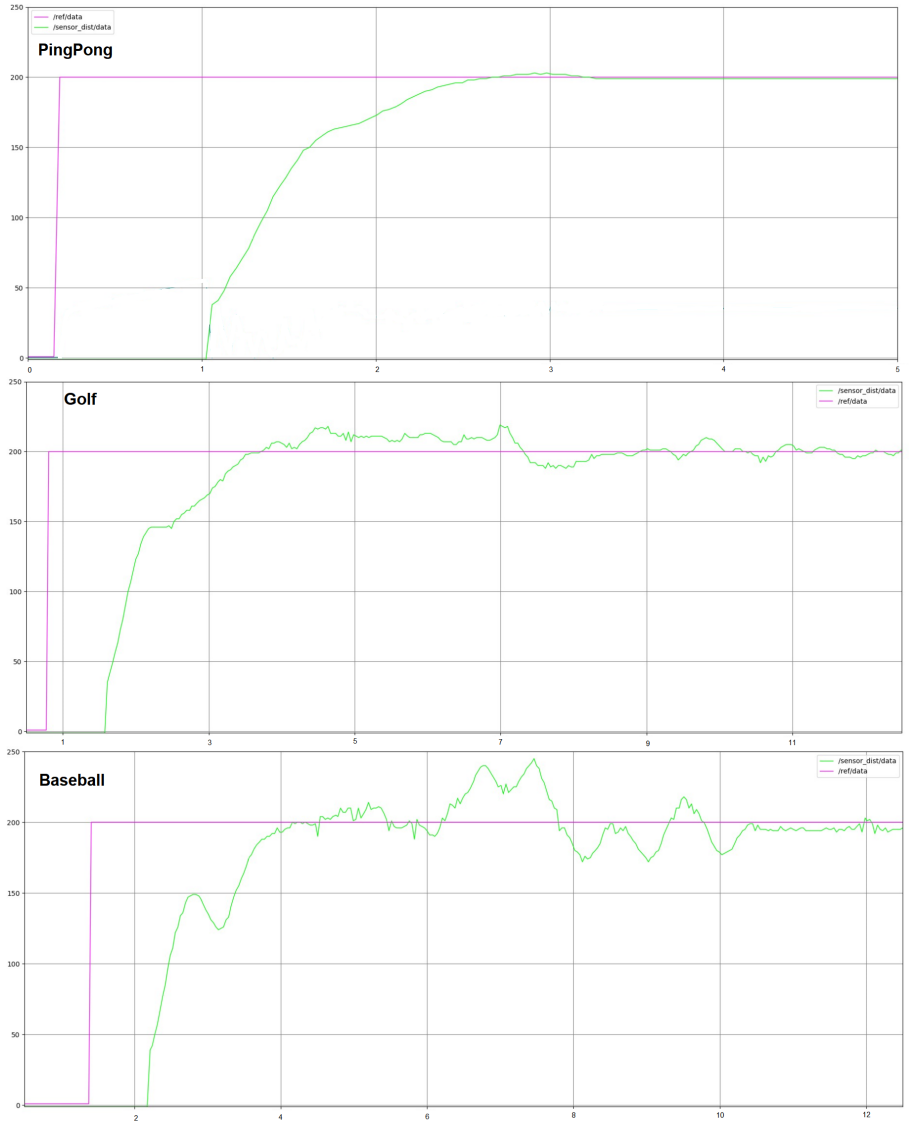
Figur 7.11: Golf ball: Poler

Polene blir grafisk presentert i figur 7.10 og 7.11. Ut fra figur 2.23, kan man se at dette er et asymptotisk stabilt system, da begge polene er i venstre halv plan.

Den reelle aksen forteller noe om hurtigheten til systemet. Jo mer negativt den reelle verdien er, desto raskere er systemet. Den imaginære verdien viser hvor dempet systemet er. Jo nermere de imaginære verdiene er 0, desto bedre er systemet dempet. Hvis de imaginære verdiene er lik 0, er responsen kritisk dempet. Ut ifra disse polene, kan man verifisere at pingpong-ballen ga best resultater, i forholdt til stabilitet og innsvingningstid.

7.2.4 Testing av ballenes feilbidrag

For å bedre forstå hvordan P,I og D leddene påvirker responsen til de ulike ballene, ble det utført en test på golfballen og baseballen, med optimale pingpong-ball parametre. Dette er for å illustrere hvordan de ulike fysiske egenskapene, medfører forstyrrelse i prosessen. I denne testen anses posisjonsplottet til pingpong-ballen, i figur 7.12 som optimal.



Figur 7.12: Pingpong vs golf vs baseball

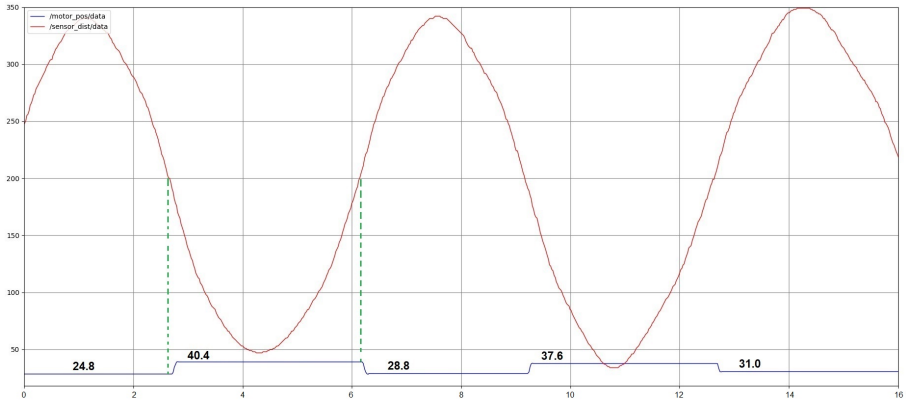
Som vist i figuren, er responsen til de to ballene dårlige. Med dette kan følgende konklusjon trekkes: For å oppnå best mulig resultat for hvert unike objekt, må reguleringsparametrene spesifiseres. Valg av forsterkningsverdier bør gjøres med henhold til objektets fysiske egenskaper.

7.3 Eksperiment prediktiv regulator

Stoppunktene velges nær endene av banen, men med hensyn til at ballen skal kunne ha litt arbeidsrom rundt stoppunktet, for å ikke treffe kanten. Dette er fordi 100% presisjon ikke er forventet. Beregningspunktet ble satt i senter av disse to punktene.

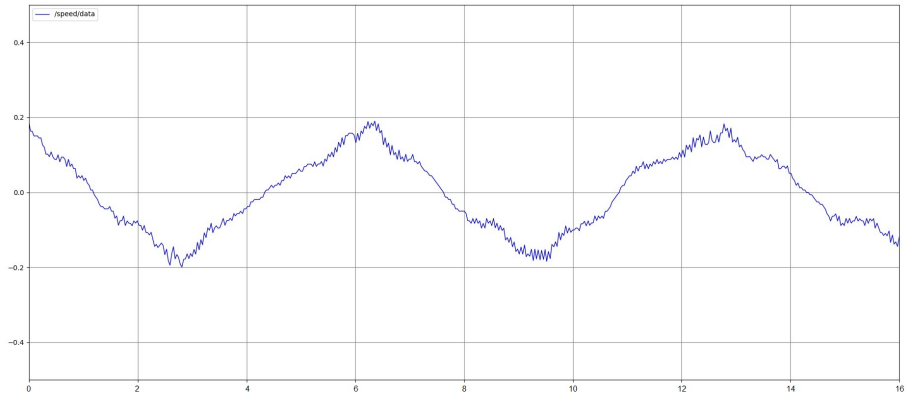
- **Stoppunkt 1:** 50 mm
- **Stoppunkt 2:** 350 mm
- **Beregningspunkt:** 200 mm

Figur 7.13 viser ballens posisjonsgraf, med tilhørende motorpådrag. Grafen viser at motorpådraget endrer verdi når ballen passerer beregningspunktet. Dette etterfølges av at ballens fart avtar, til den når 0. Dette er vist som et topp/bunnpunkt.



Figur 7.13: Ballposisjonering eksperiment

I figur 7.14 visualiseres utregnet fart under drift. Grafen svinger rundt 0, på y-aksen. Dette er fordi hvert toppunkt representerer maksfarten ballen oppnår. Deretter utføres vinkelendingen som påvirker ballens fart til å virke i motsatt retning.



Figur 7.14: Ballens fart under oscillering

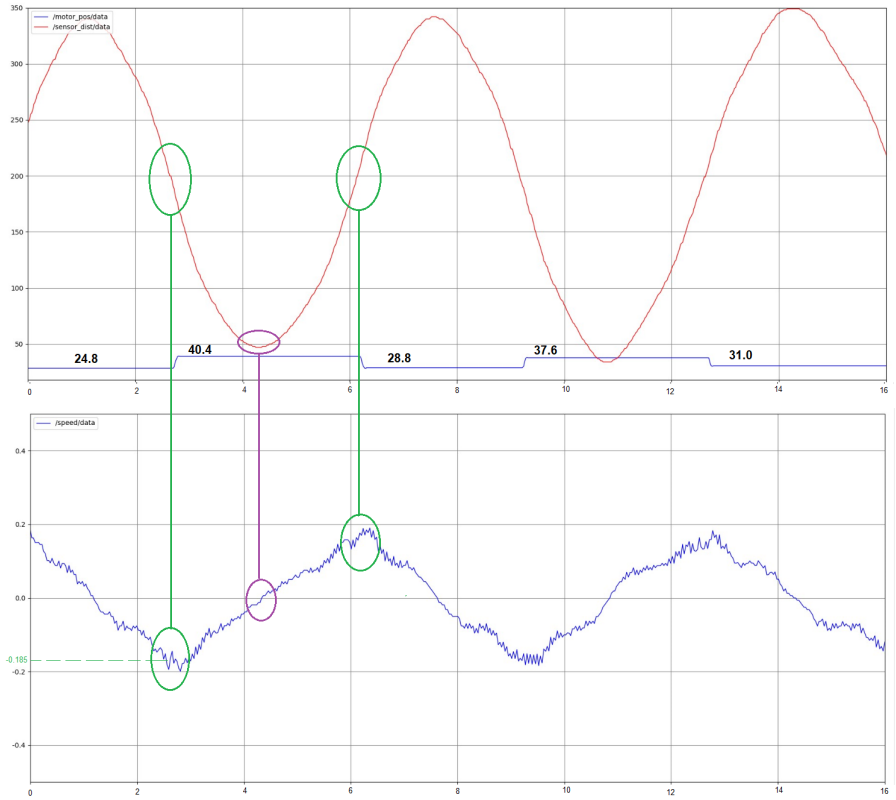
7.3.1 Verifisering av resultat

For å verifisere at systemet fungerer etter hensikt, brukes første beregning som utgangspunkt. Dette er vist i figur 7.2, med grønn oppmerking. Ved å bruke initial betingelsen for negativ fart i ligning 6.5, er det ønskelig å verifisere responsen.

Når ballen krysser beregningspunktet, settes total motorforskyvning $S = X + 34mm = 40.4mm$. Dette tilsvarer motorforkyvning fra stasjonær tilstand: $X = 6.4mm$. X representerer det siste leddet i ligning 6.5. Dermed kan man finne vinkel θ som en funksjon av X, vist i ligning 7.2:

$$\theta = \arcsin(\frac{X}{R}) = \arcsin(\frac{6.4mm}{190mm}) = 0.0337rad \tag{7.2}$$

Ved bruk av ligning 2.5 tilsvarer dette en fart på $V = 0.185\ \frac{m}{s}$. Dette resultatet verifiseres i den første målingen oppmerket med grønn farge, i figur 7.15. På bunnpunktet skal også farten være lik 0, dette vises med lilla farge.



Figur 7.15: Verifisering av posisjonsgraf

Målet var å utføre ballbevegelser som etterlignet en sinuskurve. Figur 7.15, viser dette med relativt god presisjon. Små avvik har oppstått i topp/bunn punkt. Dette skylles ulike feilbidrag som luftmotstand, ujevnheter på banen og upresise målinger.

Feilbidrag utgjør små ujevnheter i fartsmålingen. Målingen kan derfor ikke være helt tilregnelig, for å oppnå 100% presisjon. Siden responsens avvik er innenfor akseptale grenser, anses dette som et godt resultat.

Del IV

Diskusjon

Denne delen tar for seg en diskusjon rundt det ferdige prosjektet. Når det kommer til prosjekt med flere gjøremål, er det viktig å ha en plan på fremgangsmåte. Dermed bør stegene mot ferdig produkt, deles opp i delmål.

Hver metode brukt i prosjektet har blitt vurdert som den beste og mest praktiske metoden av begge gruppemedlemmene. I vurderingen har tidskrav, tilgjengelighet og testing vært sentrale faktorer. Dette kapittelet er dermed dedikert til: Å begrunne valg som ble tatt for å oppnå best resultat, alternative metoder vurdert, forslag til forbedring og fremtidig arbeid.

8 Utførelse og forbedringspotensiale

Videre er vår vurdering av fordeler og ulemper, om de ulike aspektene ved oppgaven.

1. **ROS rammeverket:** Utførelse og positive trekk

   - God struktur på *workspace*. Dette gir oss mulighet til å lage et systematisk bibliotek med god oversikt. Dette medførte en enklere feilsøking i systemet.
   - Bra pakkesystem. Et godt pakkesystem ga oss mulighet til å enkelt bruke og importere pakker til videre bruk i systemet.
   - Visuelt brukergrensesnitt. Dette gjør testing mer praktisk og brukervennlig enn ved bruk av kommandovindu.
   - Sanntidsplott. Bedre visualisering av systemets respons. Gir også mulighet for enklere feilsøking.
   - Node nettverk opprettes enkelt. Dette førte til at ekspandering av node kommunikasjon ble utført på en svært effektiv måte.
   - Visualisering i RViz. Bruk av visualisering, ga mulighet til å få oversikt over bygging av den virtuelle modellen, før den ble importert til simuleringsprogram for testing
   - Testing i Gazebo. Opprettet et godt utgangspunkt for testing av begrensninger og vippens respons, som medførte mye sikrere testing av fysisk system.
   - Presis design av deler. Hver del til den virtuelle modellen ble designet med presise mål. Dette førte til mer presise resultater og bedre visualisering.
   - Kompatibilitet med programmeringsspråk. Mulighet til å enkelt opprette noder i programmeringsspråk som Python ved bruk av *rospy*.
   - Bra nettforum. Feilsøking ble enklere da nettforumet https://answers.ros.org/ kom med gode forslag til løsning til ulike problemer.
   - God bruk av URDF. Robotformatet ble bygget svært effektivt ved bruk av importerte klasser. Dette førte til at den lange kode ble svært oversiktlig.

Forslag til forbedring og mindre positive trekk ved ROS rammeverket:

- URDF har begrenset funksjonalitet. Dette skapte problemer med bygging av en "*closed-loop struktur*". Et alternativ er bruk av *Strukturert Data Format (SDF)*, da dette formatet tillater at en "*child*" har flere *parents*". En annen metode som ble vurdert var bruken av *closed_ loop_ plugin*, [41].
- ROS *Melodic* opererer med Python 2, som ikke støtter pakken *pysoem*. Dette er en pakke som var nødvendig for styring over EtherCAT. ROS *Neotic* opererer med python 3, men hadde problemer med å importere den virituelle modellen til Gazebo. Dette førte til at simulerings del og kommunikasjons del utført i hver sin versjon. Det vil være mulig å samle alt under et prosjekt ved bruk av Linux, da ROS er mer rettet mot dette operativsystemet.
- Gazebo krever mye RAM. Ved endring av kode måtte Gazebo avsluttes, for så å starte igjen med de nye endringene. Dette var en svært tidskrevende prosess for feilsøking i programmet. Alternativt kan man bruke MoveIt og RViz til simulering i større grad enn det gruppen har gjort.

2. **EtherCAT:** Utførelse og positive trekk

- EtherCAT oppsett er lett utvidbart. Det vil være enkelt å videreutvikle, med flere EtherCAT slaver i ROS. F.eks: Legge til nye drivere til styring av 6DOF Stewart platform.
- Feildeteksjon mot LinMot-motorene. Alle feil vil komme opp som varsel i kommandovinduet. Det er også mulig å anerkjenne feil i brukergrensesnittet. Dette gir oss en forbedret testprosedyre.
- Stort utvalg i objektordbok. Dette muliggjør et stort utvalg av ferdig definerte kommandoer som skal utføres av driver.
- EtherCAT kommuniserer med en informasjonsløkke. Dette gir svært rask kommunikasjonshastighet, som er svært gunstig i sanntidskommunikasjon.
- LinMot-driver opptrer med egen interpolator. Dermed vil driveren prioritere nye bevegelseskommandoer mottatt, mens den utfører forrige kommando. Dette fører til raskere og bedre respons.

Forslag til forbedring og mindre positive trekk ved EtherCAT:

- Hver verdi hentet fra slave, har hver sin *topic* til nodenettverket. Dette kan medføre forsinkelse, når flere signaler fordeles til flere *topics*. En løsning til å forbedre dette er å korte ned på antall *topics*, og heller samle mer data på samme *topic*.
- Tidsbruken på EtherCAT i prosjektet var mye større enn forventet. Da ingen av oss hadde mye erfaring med kommunikasjonsprotokollen, ble det satt av mye tid på å sette seg inn i dette. Dermed fikk vi mindre tid til å prioritere andre viktige aspekter av oppgaven.

3. **Vippeoppsett:** Utførelse og positive trekk

- Systemet benytter seg av en stegmotor. Dette gjør at motoren har høyt holdemoment, noe som forhindrer uønska forstyrrelser. Dette vil også være praktisk for rapportering av motorens posisjon status.
- Motorens hastighet og akselerasjon velges enkelt. Dette kan gjøres i brukergrensesnittet.

- Kraft forsyningen til driver er på 24V. Dette er godkjent for studentbruk, i henhold til IDE sine HMS regler for bruk av utstyr på laboratorium.

Forslag til forbedring og mindre positive trekk ved vippeoppsett:

- Flere av de bevegelige delene er laget av plast. Dette medfører at delene kan fort bli utsatt for skade og kan knekke. Dette blir kompensert for, ved å sette begrensinger for bevegelse.

4. **Regulering:** Utførelse og positive trekk

- Fungerer etter hensikt. Reguleringen tilfredsstiller oppgavens reguleringsproblemer. Systemet gir gode resultater.
- Rask stabilisering av ball. Kort innsvingningstid.
- Enkel regulator realisert. Dette var med bruk av *simple pid*.
- Posisjonering av ulike verdier loggføres. Dette har vært til stor hjelp for sammenligning av resultater.
- Sanntid parameter innstilling. Med dette kan systemet styres og overvåkes under drift.
- Tester på ulike objekter gir ulikt resultat. Dette hjelper med å få en større forståelse av hvordan reguleringssystemet fungerer.
- Bedre respons for virkelig system, kontra virtuelt system. Responsen for styring over EtherCAT ga bedre stabilitet og hurtighet, enn simulering. Dette kom av at det ble brukt mer tid på justering av parametre for styring over EtherCAT.

Forslag til forbedring og mindre positive trekk ved regulering:

- Vippepunktet på vippen ligger forskjøvet ifht. banen. Dette gjør at ballen får en ekstra kraft som arbeider på seg i vertikal retning ved vinkelbytte på vippen under prediksjons regulering. Dette kan bli kompensert for, med å ha beregningspunktet så nær vippepunktet på banen som mulig
- Usikkerhet rundt rullefriksjon koeffisient. Dette førte til at testing av ulike verdier var nødvendig for å oppnå best resultat. Dette kan ha en påvirkning på videreutvikling av systemet. Forslag til løsning er å videre studere rullefriksjon av en ball på to metalliske overflater (banen).
- Under ideelle forhold i Gazebo, viste det seg at en PD-regulator ga bedre stabilitet og innreguleringstid enn en PID-regulator. Dette ble ikke tatt med i oppgaveteksten, da fokuset lå på å sammenligne samme regulator under ideelle forhold og reelle forhold. Resultat fra PD-regulator ble istedenfor brukt i video oppsummering av prosjektet.
- Feilbidrag fra avstandssensor. Siden avstandssensor har en minimum rekkevidde på 35mm ga dette en respons, med et stort avvik det første sekundet. Dette kan fremstå som villedende og hadde en stor påvirking på regulatoren. Etter vurdering, kom vi frem til at resultatet allikevel ga en god responstid og relativt god stabilitet, med tanke på at regulatoren måtte kompansere for feilbidraget. Det er ønsket å ta nye tester, med enten en ny avstandssensor med et større spekter for måling, eventuelt en ny sprangrespons med målinger utenfor sensorens minimum rekkevidde.

5. **Effektivitet og tidsforbruk:** Utførelse og positive trekk

- Gruppen har laget en plan for tidsforbruk. Dette medførte en mer oversiktlig fordeling av tidsbruk og planlegging av fremgang.
- Faste veiledningsmøter. Hver andre uke var det satt av tid til et veiledningsmøte, med fagansvarlige. Dermed fikk vi tilstrekkelig veiledning og forslag underveis.
- Arbeidsfordeling. Tidlig i prosjektet ble det laget en plan for arbeidsfordeling. Dermed var det enklere for hvert medlem å jobbe selvstendig for å utføre gjøremål.

Mindre positive trekk ved effektiviteten og tidsforbruk:

- Covid-19 pandemi. På grunn av pågående Covid-19 pandemi, har arbeidet med oppgaven blitt en enda større utfordring. Studentene valgte å låne kontorplass privat, da det var lite tilgjengelige plasser å sitte på skolen.
- Det oppsto en konflikt med tidsskjema når eksamen i *ING200 - Teknologiledelse* ble satt kun en uke før levering av bachelor-oppgave.
- Endring av oppgavebeskrivelse. Underveis i prosjektperioden, fikk vi beskjed om at prosjektets problemstilling skulle endres. Ønske fra fagansvarlig var å sette fokus på vippesystemet, og dermed se bort ifra den originale oppgave teksen som omhandlet styring av 6DOF hexapod. Dette viste seg å komme i konflikt med arbeidet studentene hadde gjort til den dags dato, da mange timer var lagt inn for grafisk konstruksjon av hexapod. Det ble dermed tatt en avgjørelse om at all fremgang innen hexapod prosjektet, skulle presenteres i *fremtidig arbeid* kapittelet.

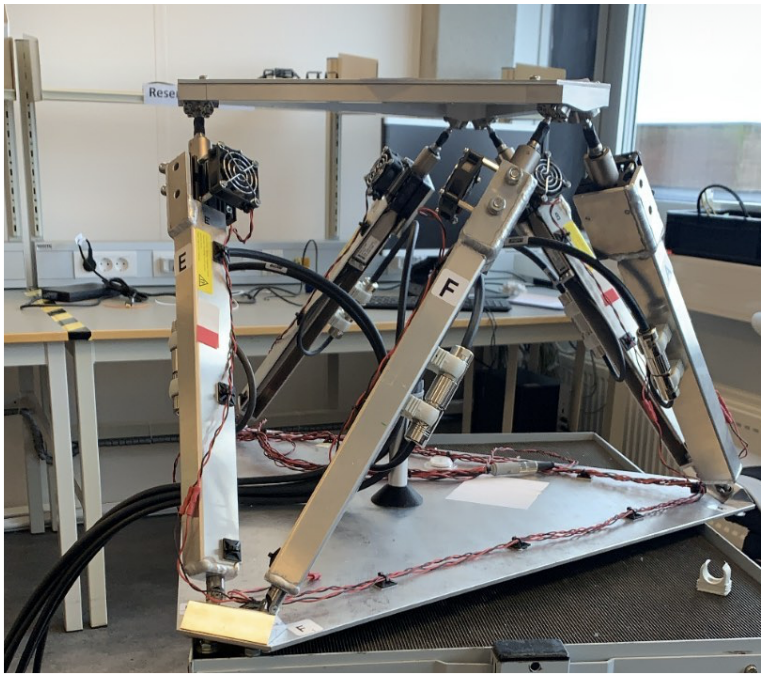
9 Fremtidig arbeid

Den originale oppgaveteksten omhandlet en 6DOF hexapod. Da det ble tatt en avgjørelse om at prosjektet skulle omhandle styring av vippesystemet, utgår arbeidet gjort på hexapod delen. Dermed ønsket fagansvarlig at alt arbeid med hexapod, inkluderes i rapporten som fremtidig arbeid.

Videre i dette kapittelet, presenteres gruppens arbeid med hexapod, samt en kort beskrivelse av systemet. I tillegg foreslås det hvordan system vårt kan bidra til en god løsning.

9.1 Bakgrunn, hexapod

Hexapod delen beskriver en mekanisme som har seks frihetsgrader, kontrollert av seks fastmonterte lineær motorer, styrt av hver sin LinMot-driver a typen *1150-EC-XC-0S*. På samme måte som med vippesystemet, kan hver lineærmotor styres via EtherCAT. Stewart platform er en form for hexapod, som blant annet brukes til å simulere flyforhold, under opplæringen av piloter.

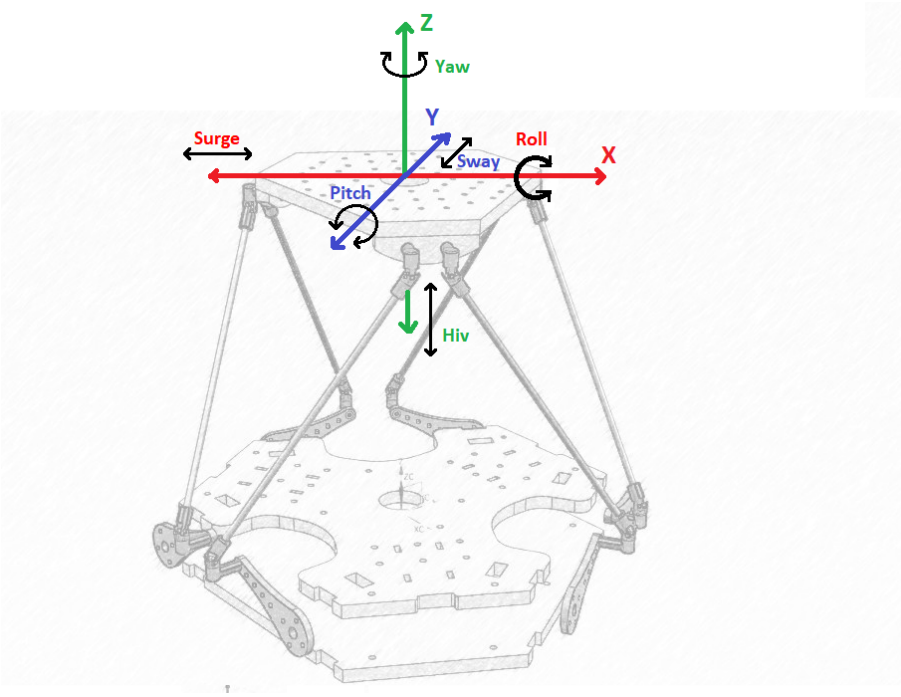


Figur 9.1: *IDE* sin hexapod

Som fremtidig arbeid er det ønskelig at det opprettes modellering og visualisering av en hexapod i ROS. Det er også ønsket å oppnå kommunikasjon mellom ROS nettverk og hver individuelle driver, ved å utvide eksisterende kommunikasjonsprotokoll. En visualisering er laget av gruppemedlemmene. Denne kan også styres i Gazebo. Det er til nå opprettet nodekommunikasjon til hver *joint* i modellen, men det finnes mulighet for forbedring.

9.2 Definisjon av frihetsgrader

En hexapod kan manipuleres i seks frihetsgrader under styring av hver motor. Tre av frihetsgradene er prismatiske, og kan oppnås ved lineær forflytning av massesenter i x, y eller z-akse. De tre andre frihetsgradene kan defineres som *roll, pitch og yaw* (RPY), og er roterende bevegelse rundt de ulike aksene.



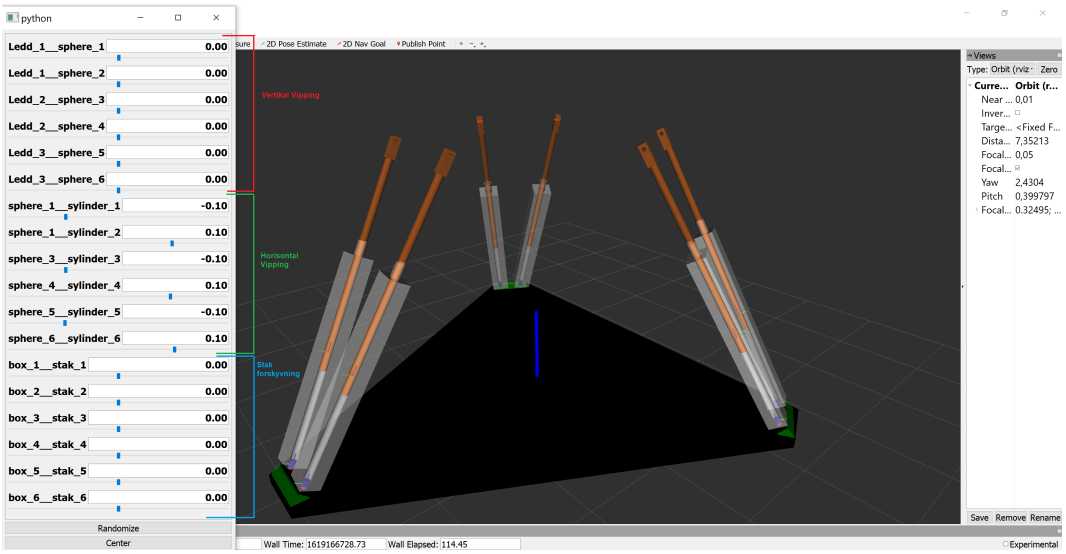
Figur 9.2: Visualisering av seks frihetsgrader på en hexapod. Hentet fra [46]

- **Surge:** Lineær forflytning i X-aksen
- **Sway:** Lineær forflytning i Y-aksen
- **Hiv:** Lineær forflytning i Z-aksen
- **Roll:** : Roterende bevegelse om X-aksen
- **Pitch:** Roterende bevegelse om X-aksen
- **Yaw:** Roterende bevegelse om X-aksen

Ved å sette lengde til de ulike beina til hexapodden, vil sentermassen av topplanet kunne nå et hvert punkt innenfor et endelig arbeidsområde.

9.3 Visualisering av hexapod

På samme måte som i vippeprosjektet, er RViz brukt for visualisering. Hver "arm" kan styres individuelt med tre forskjellige bevegelser, som vist i figur 9.3. Visualisering er laget i URDF, og er basert på fremgangsmåten til vippesystemet. Problemet med dette er at URDF har veldig begrenset funksjonalitet ved bygging av komplekse roboter.



Figur 9.3: Visualisering av hexapod i RViz.

De to største problemstillingene som dukket opp under visualisering av hexapod ved bruk av URDF, var følgende:

1. Hvordan realisere et fungerende "*closed-Loop*" system, med tanke på topp platen?
2. Hvordan oppnå friere bevegelse i *joints*, til hver arm?

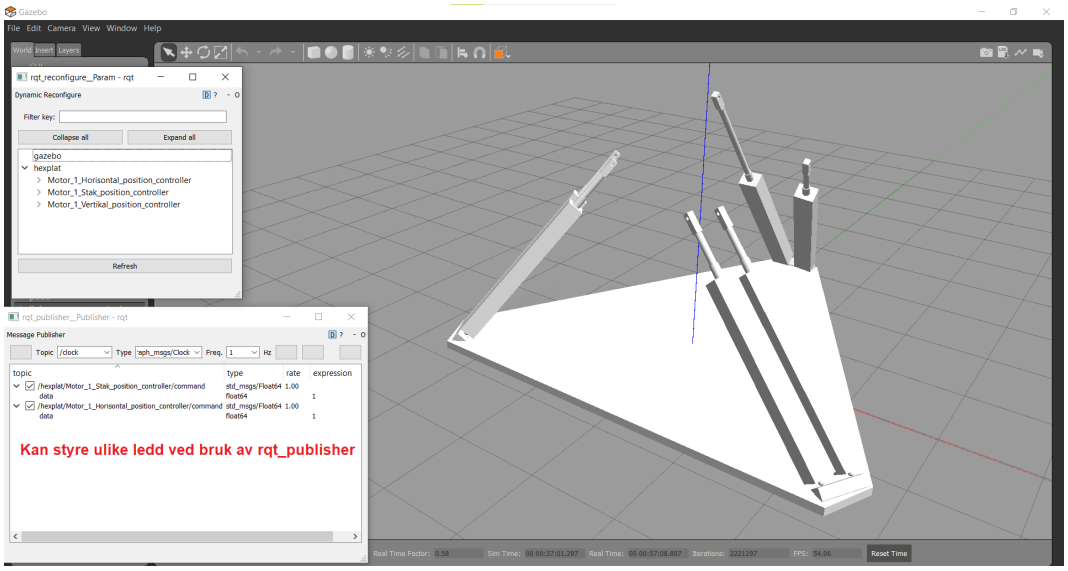
Som vist i figur 9.3, må vertikal vipping og horisontal vipping til hver arm, utføres hver for seg. Dermed er det ønsket å finne en løsning til friere bevegelse. Sammenstillingen av modellen for å oppnå simulerte festepunkt, vil være mye mer komplisert i en hexapod, sammenlignet med vippemodellen. Dermed bør en alternativ løsning brukes. Etter en grundig undersøkelse av mulighetene for en løsning, har gruppen kommet med et forslag.

Som nevnt i *diskusjon* kapittelet, vil SDF være en mer praktisk løsning for å oppnå kompleksitet i roboten. Ved hjelp av dette formatet er det mulig å bygge en "*closed-loop*" struktur i RViz. Dette skal også fungere for simulering i Gazebo. Et forslag til fremgangsmåte for bygging av en "*closed-loop*" struktur i SDF, vises i: [42].

I SDF er det også mulig å oppnå friere rotasjonsbevegelse med en *joint*. Her er det mulighet for å definere to rotasjonsbevegelser i *joint* aksesystemet, som en *link* kan feste seg til. Mer info om dette finnes i: [43].

9.4 Virtuell simuliering av hexapod

Hexapod modellen er importert til simueringsprogrammet Gazebo, hvor hvert bevegelige ledd kan styres fra *rqt_ publisher* med inputverdier, se figur 9.4. Det er i utgangspunktet mulig å utføre tester på den grafiske modellen uten en topp plattform, men vil ikke være hensiktsmessig for å få et godt overblikk over kinematikken i systemet.



Figur 9.4: Hexapod i Gazebo.

Videre i arbeidet, er det ønsket at det opprettes et testprogram for kontrollert styring av denne modellen. Eventuelt kinematikk for kontrollert bevegelse av topp plattformen. Dette kan være med på å skape et godt utgangspunkt for testing av *IDE* sin hexapod.

Hexapod prosjektet studentene lagde i URDF kan finnes med denne linken:
https://drive.google.com/drive/folders/1-cUwK5rFQc30E1RAYF1_8AzGz4fqZCBM?usp=sharing

9.5 Realisering av EtherCAT kommunikasjon

Realisering av EtherCAT kommunikasjon mellom ROS og hver indviduelle driver, kan baseres på arbeidet fremført i kapittel 5. Siden funksjonaliteten for kommunikasjonsprotokollen er utført, vil implementering av dette til hexapod systemet være en enkel oppgave.

Ved å utvide koden for definerte slaver under EtherCAT-master, vist i listing 6, kan en enkelt legge til nye LinMot slaver. Hver slave og medfølgende kommunikasjonsprotokoll må selvfølgelig defineres individuelt i koden.

10 Konklusjon

Denne oppgaven har vært svært tidskrevende. Den største utfordringen var å sette seg inn i ROS rammeverket, der hver versjon har ulik funksjonalitet.

Gruppen har også fått inntrykk av at prosjektet skal utvides på et senere tidspunkt. Dette har også påvirket valg av ROS versjon. Valg av versjon i ROS, er tatt på grunnlag av følgende funksjonalitet.

1. **Grafisk konstruksjon:** Visualisere en virtuell vippemodell for testing.
2. **Simulering:** Styre visualisert modell. Utføre test for eksperiment. Finne begrensninger og muligheter utførelse.
3. **EtherCAT kommunikasjon:** Implementere kommunikasjonsprotokoll i ROS for styring av fysisk modell.
4. **Regulering:** Kontrollert styring av både virtuell og fysisk modell. Lage eksperiment for presentasjon av resultat.

Valget falt på ROS *Melodic* for oppgaver 1 og 2. Denne versjonen viste seg å være mest praktisk for utvikling og visualisering av en virtuell modell. ROS *Neotic* ble brukt for styring av en fysisk modell. Denne var mer hensiktsmessig for kommunikasjon med eksterne enheter som LinMot-driver. Begge versjonene fremsto som brukervennlige og oversiktlige, noe som er gunstig om systemet skal utvides. Det var også mulighet for å realisere node for regulering i begge versjoner.

Oppgaven avklarer behovet om å kunne styre og visualisere vippemodellen i ROS, med forslag om bruk av RViz, Gazebo og MoveIt. I ROS *Melodic* realiseres dette på en svært tilfredsstillende måte. Det er mulighet for visualisering av modell, med MoveIt visualiseringsverktøy, RViz. Dette skaper et godt grunnlag for en eventuell utvidelse av modellen. Bruk av spesialdesignede komponenter laget i FreeCAD, gjorde at visualiseringen ble svært tilfredsstillende. Resultatet var en presis etterligning av vippemodellen, som videre kunne importeres til simuleringsprogram.

Under konstruksjon av modellen i RViz, oppsto det noen problemer med sammenstilling av ulike ledd. Dette på grunn av at URDF, kun har mulighet til å bygge modeller i en grenstruktur. Dette medførte komplikasjoner med å lage en sammenhengende struktur. Da modellen har relativ lav kompleksitet, ble dette løst med å synkronisere bevegelse i modellen. Denne endringen ga gode resultater, men dette kan være et problem for utvidelse av modellens kompleksitet. Det finnes et forslag til løsning på problemet i kapittelet *Diskusjon*.

Det ble tatt et valg om å bruke simuleringsverktøyet Gazebo, fremfor å videre utnytte simuleringsfunksjonaliteten i MoveIt. Dette var en vurdering som ble tatt underveis. Gazebos evne til å simulere fysiske krefter, viste seg å gi en bedre etterligning av realiteten. Resultatet var en simulering som fungerte svært godt etter hensikt. Ved bruk av ROS nodefunksjonalitet, var det enkelt å implementere ulike metoder for styring i sanntid, med gode resultater. Ved bruk av *rqt* pakker i ROS, kan man effektivt endre parametere for styring under drift.

Det ble lagt vekt på å lage et oversiktlig *workspace*. Dette resulterte i et ryddigere prosjekt, samt skapte muligheten for å enkelt opprette nye noder i nodenettverket. Reguleringsmetoder kan dermed opprettes på en oversiktelig måte. Problemet for styring og regulering, ble dermed løst med å opprette noder med ulik funksjonalitet, for bevegelse av modell. Ved hjelp av et *rqt* brukergrensesnitt, var det enkelt å velge hvilken måte man ønsket å styre systemet. Dette ga en svært brukervennlig metode for utførelse av eksperiment.

Kommunikasjon mellom ROS og lineærmotor, ble implementert til nodenettverket, ved å bruke en kommunikasjonsprotokoll for EtherCAT utviklet av Morten Mossige. Dermed var det mulig å utveksle data direkte mellom ROS og slaveenhetene. Det er mulighet for fremtidig utvidelse av slavenettverket i noden, men det krever at funksjonaliteten til hver slave defineres. Resultatet ble svært tilfredsstillende, da kommunikasjonsprosessen fungerte etter hensikt, uten merkbare forsinkelser.

Løsning for reguleringsproblemet ble til, ved bruk av to ulike metoder. En PID regulator for å oppnå stabilisering av ballen og en prediktiv regulator for å etablere kontroll av ballbevegelsen. Resultatene for en stabilisering av ballen, tilfredsstilte gruppens forventninger til respons. Systemet oppnådde en god harmoni mellom stabilitet og innreguleringstid. For prediktiv regulering var resultatet innenfor akseptable grenser for posisjonsavvik. Dermed anses resultatet som godt.

Flere feilbidrag i systemet ble oppdaget og undersøkt. Begrenset rekkevidde for måling med avstandssensoren brukt i vippeoppsettet, førte til et kraftig utslag på responsen under testing av PID regulator. Systemet klarte allikevel å kompensere for feilbidraget.

Et visuelt brukergrensesnitt for hvert system er utviklet og implementert. Brukergrensesnittet har mulighet til å innhente ulike systemverdier, som sensoravlesning og posisjon. Dette kan så visualiseres grafisk. Brukergrensesnittet gir også brukeren mulighet til å lagre data i en tekstfil, for senere analyse.

Gruppen har fullført alle punktene under oppgaveinnhold som er gitt i *justert oppgavebeskrivelse*, vist på side 3. Tidlig i prosjektet var det planlagt å lage et system for styring av en 6DOF hexapod. Dette ble tatt ut fra oppgaveteksten, da fagansvarlig heller ønsket at gruppen skulle fokusere på regulering av en lineærmotor i et vippesystem. Arbeid utført i sammenheng med hexapod, er dermed presentert i *Fremtidig arbeid* kapittelet. Da vippesystemet og hexapod bruker samme type lineærmotor, kan mye av systemet laget i dette prosjektet, brukes til å realisere en styring av en hexapod.

11 Vedlegg

I startfasten av prosjektet, ble det laget en plan for estimert tidsforbruk på ulike aspekter i oppgaven. Estimert arbeid, vises i figur 11.1 med fargeoppmerking. Underveis ble timeforbruk loggført.

		Ukenummer																					
1																							
2	Aktivitet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Timer Totalt	Timer Esimert	Ansvarlig
3	Planlegging																							
4	Behovspesifikasjon	1																				1	1	Begge
5	Arbeidsfordeling	1																				1	1	Filip
6	Smittevern tiltak																					5	20	Begge
7	Mek. Konstruksjon			2																		2	1	Ådne
8	Totalt																							
9	Prosjekstyring																							
10	Aktivitetsplan	1																				1	1	Filip
11	Veiledningsmøter	1		1		1		1				1		1		1			1	1		9	10	Begge
12	Gruppemøter	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		20	20	Begge
13	Forelesninger	4	4	4	4	4	4	4		8												32	30	Begge
14	Totalt																							
15	Programvare																							
16	ROS (kommunikasjon)	63	20	2	2		29	10	30			10		10								176	150	Begge
17	Rviz (visualisering)		40	80	10			20	50													200	50	Begge
18	FreeCAD		6	3																		9	30	Ådne
19	LinMotTalk/EtherCAT				10	90	35	20	30													185	150	Filip
20	Regulering											40	45	30	30	10	20					175	200	Ådne
21	Gazebo				66	10	10	10		7			10		10							123	100	Begge
22	Totalt																							
23	Maskinvare																							
24	Skjemategning			1																		1	4	Filip
25	Montering																							
26	Lineær motor			2	4																	6	10	Begge
27	Hexapod																						20	
28	Testing og feilsøking				4																	4	30	Ådne
29	Totalt																							
30	Rapport skriving																							
31	Skrive Essay								6	80	60											146	150	Begge
32	Prosjektrapport		6	8	5		5	10	5			11	17	15	20	52	65	70		68		357	300	Begge
33	Rapport av logg	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1			18	18	Filip
34	Egenstudie	4	4	4	4	4	4	4				4	4	4	4	4	4		70	2		124	150	Begge
35	Totalt																							
36	Prosjekt Totalt	**77**	**82**	109	111	111	89	81	123	97	62	68	78	62	66	69	91	72	73	72	0	1593	1446	

Figur 11.1: Logg av timebruk

Referanser

[1] Samdvold, K. E., Øgrim, S., Thorstensen, R., Thorstensen, A. K., Bakken, T., Pettersen, B., Skrindo, K.: Gyldendals formelsamling i mattematikk, 2017. Oversatt av H. Crew og A. de Salvio

[2] Galilei, G.: Dialogues concerning two new sciences. Macmillan, 1914. Oversatt av H. Crew og A. de Salvio

[3] Young, H. D. og R. A. Freedman: University Physics. Pearson Education, 14. utgave, 2016.

[4] Kurtus. R, Coefficient of Rolling Friction, 2016. hentet fra: https://www.school-for-champions.com/science/friction_rolling_coefficient.htm#.YJUR368zY2w

[5] LinMot. hentet fra: https://shop.linmot.com/E/linear-motors/linear-motors-p01-23/stators-ps01-23x80-hp/ps01-23x80f-hp-r20.htm

[6] NTI AG: MotionCtrlSW-SG5-SG7. 2019. hentet fra: https://shop.linmot.com/E/documentation/software-manuals/drive-sw-manuals/user-manual-0185-1093-(en):.htm

[7] OMRON Corporation, E3AS Series, 2020. hentet fra: http://www.ia.omron.com/products/family/3779/specification.html

[8] OMRON Corporation, NX-ECC, 2020. hentet fra: http://www.ia.omron.com/products/family/3184/specification.html

[9] Omron Corporation. 2021. hentet fra: https://industrial.omron.no/no/products/S8VK-S12024

[10] ROS: about ros. hentet fra: https://www.ros.org/about-ros/

[11] Staples, G: Distributions. 2020. hentet fra: http://wiki.ros.org/Distributions

[12] ROS 2 docs, 2021: understandig Ros2 topics. hentet fra: https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html

[13] Playfish: urdf. 2019. hentet fra: http://wiki.ros.org/urdf

[14] Lalancette, C: Creating Package. 2018. hentet fra: http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage

[15] Ferguson, M: dynamic_reconfigure. 2015. hentet fra: http://wiki.ros.org/dynamic_reconfigure

[16] Dirk, T: rosgraph_msgs. 2015. hentet fra: http://wiki.ros.org/rosgraph_msgs

[17] GvdHoorn: Rospy. 2017. hentet fra: http://wiki.ros.org/rospy

[18] Saito, I: std_msgs. 2017. hentet fra:
http://wiki.ros.org/std_msgs

[19] Okada, Y: controller_manager. 2019. hentet fra:
http://wiki.ros.org/controller_manager

[20] Reynolds, M: joint_state_controller. 2020. hentet fra:
http://wiki.ros.org/joint_state_controller

[21] GvdHoorn: robot_state_publisher. 2020. hentet fra:
http://wiki.ros.org/robot_state_publisher

[22] docs.ros : getting started. 2018. hentet fra:
http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/getting_started/getting_started.html

[23] Chocolatey Software, Inc. (2021). hentet fra:
https://docs.chocolatey.org/en-us/faqs

[24] Staples, G: 2020. Installation. hentet fra:
http://wiki.ros.org/Installation

[25] EtherCAT Technology Group : EthetCAT and EthetCAT P Slave Implementation Guide. 2018. hentet fra:
https://www.ethercat.org/download/documents/ETG2200_V3i1i0_G_R_SlaveImplementationGuide.pdf

[26] Dewesoft d.o.o. 2021. hentet fra:
https://dewesoft.com/daq/what-is-ethercat-protocol

[27] NTI AG: Motion Control SW. 2014. hentet fra:
http://www.linmot.com/fileadmin//user_upload/Downloads/software-firmware/servo-drives/linmot-talk-6/Usermanual_MotionCtrlSW_SG5_e_recent.pdf

[28] EtherCATGroup: EtherCAT Functional Principle (2D). 2016. hentet fra:
https://www.youtube.com/watch?v=z2OagcHG-UU&ab_channel=EtherCATGroup

[29] Beckhoff new automation technology: EtherCAT, General. hentet fra:
https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_io_intro/1257993099.html&id=3196541253205318339

[30] Profilex s.a: SERVO DRIVES. hentet fra:
https://profilex.be/pdf/linmot/General/www.profilex.be_LinMot_Databook_V24_EN_839-943.pdf

[31] NTI AG: EtherCAT Interface. 2014. hentet fra:
http://www.linmot.com/fileadmin/user_upload/Downloads/software-firmware/servo-drives/linmot-talk-6/Usermanual_EtherCAT_SG5_e_recent.pdf

[32] NTI AG: LinMot-Talk 6 Configuration Software. 2018. hentet fra:
https://linmot.com/fileadmin//user_upload/Downloads/software-firmware/servo-drives/linmot-talk-6/0185-1059-E_6V7_MA_LinMotTalk.pdf

[33] Halvorsen, H, P: Reguleringsteknikk med LabVIEW og MathScript eksempler. 2016. hentet fra:
https://www.halvorsen.blog/documents/automation/resources/

Reguleringsteknikk%20med%20eksempler.pdf?list=PLdb-TcK6Aqj0qo8bGea_2F0YEU0IBeiXO

[34] Haugen, F. Eksperimentell innstilling av PID-regulator http://techteach.no/fag/tel220/v06/pid_regulering/pid_innstilling.pdf

[35] Second Order and Higher Order Systems. hentet fra: http://engineering.ju.edu.jo/Laboratories/05%20-%20Second%20Order%20System%20and%20Higher%20order.pdf

[36] Elsevier B.V. Second-Order System. 2021 hentet fra: https://www.sciencedirect.com/topics/engineering/second-order-system?fbclid=IwAR0OkPtGY0DcxMXH0AjXvVy70GbZ8cj4jSJt-31NuWlJJzL_3H7bRxxYxEQ

[37] Haugen, F: Advanced dynamics and control. 2012. hentet fra: http://www.techteach.no/publications/books/advanced_dynamics_and_control/Adv_Dyn_Con_textbook.pdf

[38] The FreeCAD Team. hentet fra: https://www.freecadweb.org/

[39] Revision. B, P: Master. 2021. hentet fra: https://pysoem.readthedocs.io/en/latest/master.html

[40] Lundberg, M: simple_pid. 2021. hentet fra: https://pypi.org/project/simple-pid/

[41] angel_jj: closed loop plugin. 2019. hentet fra: http://wiki.ros.org/action/edit/Angel_jj/closed_loop_plugin

[42] Open Source Robotics Foundation: Model a 4-bar linkage in SDFormat and URDF. 2014. hentet fra: http://gazebosim.org/tutorials?tut=kinematic_loop&cat=

[43] Open Source Robotics Foundation. 2020. hentet fra: http://sdformat.org/spec?ver=1.8&elem=joint#joint_axis2

Bilde referanser:

[44] NTI AG: MotionCtrlSW-SG5-SG7. 2019. hentet fra: https://shop.linmot.com/data/import/Software/0185-1093-E_6V8_MA_MotionCtrlSW-SG5-SG7.pdf

[45] Nordfjell, T: Smallwood annual meeting. 2020. hentet fra: http://www.smallwood.eu/smallwood-annual-meeting/

[46] Ortiz, E: Stewart platform with fuzzy control and computer vision, 2019. hentet fra: http://www.innovatefpga.com/cgi-bin/innovate/teams.pl?Id=AS032