Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Datateknologi / Automatisering og elektronikkdesign	Vårsemesteret 2021 Åpen
Forfattere: Ask S. Ramberg, Simen L. Røisi, Christian B. Leirvåg	
Fagansvarlig: Tormod Drengstig Veileder: Tormod Drengstig	
Tittel på bacheloroppgaven: Utvikling av oppgaver og løsningsforslag til bruk med Lego EV3 og Matlab/Python Engelsk tittel:	
Studiepoeng: 20	
Emneord: Matlab, Python, Lego Mindstorms EV3	Sidetall: 153 + vedlegg/annet: Stavanger 15. mai 2021

Innhold

Sammendrag

Emnet ELE130 *Anvendt matematikk og fysikk i robotprogrammering* i 2. semester består bl.a. av et Legoprosjekt hvor LEGO Mindstorms EV3 brukes sammen med enten Matlab eller Python. Som en del av prosjektbeskrivelsen er det gitt forslag til kreative prosjekter i tillegg til de obligatoriske prosjektene numerisk integrasjon, derivasjon og filtrering.

Denne oppgaven går ut på å lage forslag til løsning til et utvalg av de kreative prosjektforslagene. I tillegg oppfordres det til å foreslå nye prosjekter som kan demonstrere hva Legoutstyret kan brukes til, og på den måten fungere som motiverende eksempler for studentene som tar ELE130.

Kapittel 1

Introduksjon

1.1 Oppgavebeskrivelse

Denne oppgaven går ut på å lage løsningsforslag til et utvalg av prosjektene fra den kreative delen i emnet ELE130 *Anvendt matematikk og fysikk i robotprogrammering*. I tillegg skal det foreslås nye prosjekter som demonstrerer hva legoutstyret kan brukes til. Dette er et fag alle studentene ved datateknologi, og automatisering og elektronikkdesign tar i 2. semester.

1.2 Emnet: ELE130

Som en del av vurderingen i faget må studentene gjennomføre et gruppeprosjekt der de anvender matematikk- og fysikkunnskaper i programmering av Lego-roboten til å løse forskjellige praktiske problemstillinger.

Gruppeprosjektet består av en obligatorisk og en frivillig / kreativ del. Den kreative delen består av en rekke små og store prosjekter, i denne bacheloroppgaven har vi tatt for oss noen av de større prosjektene. I listen nedenfor er disse markert med fet tekst.

1.2.1 Større prosjekter

- **Dokumentering av ultralydsensor**
- **PID-regulator**
- Automatisk kjøring med PID-regulator
- **Tegn Bane**
- Rotasjonshastighet ved bruk av enkoder
- Estimer hastighet
- Katapult
- **Sykkelcomputer**
- **Frekvensrespons**
- Nedfolding
- Adaptiv cruisekontroll
- Pappeskeareal
- **Robotstøvsuger**

For å løse prosjektene krever det at studentene kombinerer sine kunnskaper i matematikk, fysikk og programmering.

I tillegg til prosjektene i liste 1.2.1 ovenfor, har vi foreslått et nytt prosjekt kalt "Modellering av pendel".

1.3 Programmeringsspråk

Legoroboten kan programmeres i både Matlab og Python. I denne oppgaven bruker vi begge deler ettersom enkelte funksjoner ikke er tilgjengelige i begge språkene.

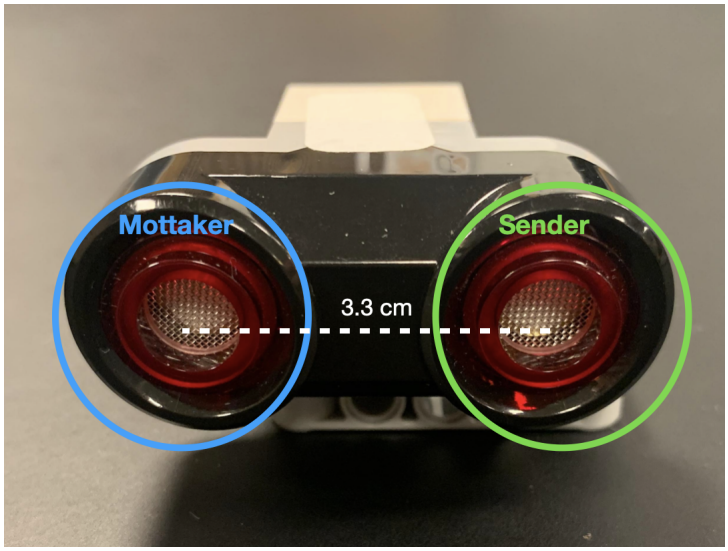
1.4 Legokonstruksjon

Robotbilen ble bygd etter bruksanvisningen til LEGO Mindstorms[2], og er delvis modifisert i noen av prosjektene. Denne ble valgt ettersom det er en av de offisielle konstruksjonene til EV3'en, og det er slik studentene i ELE130 bygger roboten.

Kapittel 2

Dokumentering av ultralydsensoren

Lego Mindstorms er et robotsett som kommer med en rekke sensorer, deriblant en ultralydsensor. Denne kan ses i figur 2.1



Figur 2.1: Lego Mindstorms ultralydsensor sett forfra. Sensoren har en sender og en mottaker. Distansen mellom disse to er 3.3 cm.

Dokumentering av ultralydsensoren

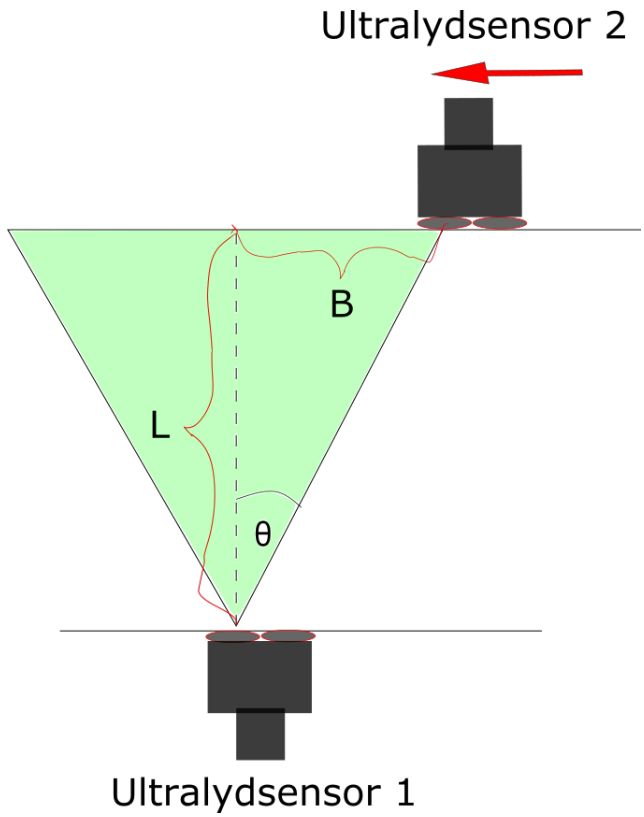
Formålet ved dette kapittelet har vært å dokumentere egenskapene til Lego Mindstorms sin ultralydsensor, ettersom flere av prosjektene i faget ELE130 innebærer å bruke denne sensoren. Erfaringen med dette er at det oppstår en del underlige resultater som vi ønsker å forstå. Det er også forskjell på hvordan sensoren kan brukes i Matlab og Python.

Vi har dokumentert følgende:

- Horisontal og vertikal utsendingvinkel på ultralyden.
- Hvordan målingene påvirkes av formen på objektet som det måles avstand til.
- Hvor skrå innfallsvinkel et objektet kan ha i forhold til ultralydsensoren, og fremdeles bli detektert.
- Hvorvidt deteksjonsbredden bedrer seg når et objekt er siktet direkte mot sensoren.
- Minste nødvendige størrelse på objekt som reflekterer ultralyd.
- Hvorvidt presisjonen i hjørnemålinger bedres av et et rør rundt mottakeren.
- Hvordan man løser problemet med crosstalk.

2.1 Dokumentering av utsendingsvinkel

I dette delkapittelet har vi dokumentert hvor stor den horisontale og vertikale utsendingsvinkelen på lydbølgene til ultralydsensoren er. Dette ble gjort ved å benytte to ultralydsensorer. Den ene sensoren ble satt til å sende ut ultralyd, mens den andre sensoren ble satt til å lytte etter ultralyd. En skisse av forsøksoppsettet er vist i figur 2.2.



Figur 2.2: Skisse av forsøksoppsettet. θ er den utregnede vinkelen. Aksene er **L** for lengde, og **B** for Bredde. Den røde pilen viser at ultralydsensor 2 blir gradvis flyttet innover til den mottar et signal, og returnerer '1'.

2.1 Dokumentering av utsendingsvinkel

Ultralydsensoren har to moduser i programmeringsspråket Python. Henholdsvis `distance` og `presence` [3]. Modusen `distance` måler avstand mellom sensoren og et objekt, mens modusen `presence` registrerer innkommende ultralyd, og returnerer enten **true** / **1** eller
false / **0**. Kodeutdraget 2.1 viser hvordan de forskjellige modusene brukes.

Kode 2.1: Utdrag fra `P13_UltralydDokumentasjon.py`

```
1              myUltrasonicSensor1 = UltrasonicSensor(Port.S4)
2              myUltrasonicSensor2 = UltrasonicSensor(Port.S4)
3
4              myUltrasonicSensor1.distance()
5              myUltrasonicSensor2.presence()
```

I linje 1 og linje 2 blir sensorene definert som variabler, og i linje 4 og linje 5 blir sensorene satt i modusene distance og presence. Modusen presence er kun tilgjengelig når man programmerer i Python. Det vil si at man i Matlab kun kan bruke ultralydsensoren til å måle avstand.

Målingene i dette delkapittelet ble repetert på tre forskjellige distanser. Henholdsvis 29.7 cm, 59.4 cm og 89.1 cm. Hvor den siste tilsvarer lengdene av tre A4 ark. Årsaken til at disse distansene ble valgt skyldes at vi ønsket å ha muligheten til å notere ned punkter på bordet. Det var også viktig at ultralydsensorene sto parallelt med hverandre slik at vi fikk presise målinger. I figur 8.14 ser vi et bilde av målingsoppsettet med disse arkene.

2.1 Dokumentering av utsendingsvinkel

Figur 2.3: Øverst til høyre står sensoren som lyttet etter ultralyd, mens nederst til venstre står sensoren som sendte ut ultralyd.

Ultralydsensor 1 var teipet fast i bordet slik at den sto i en fast posisjon, med senderen i senterlinjen. Når vi begynte en måling sendte denne kontinuerlig ut ultralyd. Denne sensoren ble styrt av en EV3 som kjørte på Matlab.

Ultralydsensor 2 kjørte derimot på Python, og var satt i modusen `presence` slik som vist i kodeutdrag 2.1. Da vi begynte en måling sto denne parallelt med, og utenfor rekkevidden til **ultralydsensor 1**. Deretter ble den skjøvet sakte innover bordet. Da **Ultralydsensor 2** så returnerte '1', ble dette punktet markert som bredden, **B**.

Dette ble repetert for alle de tre lengdene, **L**, fra både høyre- og venstreside for ultralydsensor 1. Ultralydsensor 1 ble så rotert 90 grader mot venstre, slik at senderen hadde kontakt med bordet. Se figur 2.4. Deretter ble nye målinger foretatt. Disse målingene vil da tilsvare de vertikale utsendingsvinklene til sensoren. Henholdsvis overside og underside. Overside blir da målt til venstre for ultralydsensor 1, og underside til høyre.

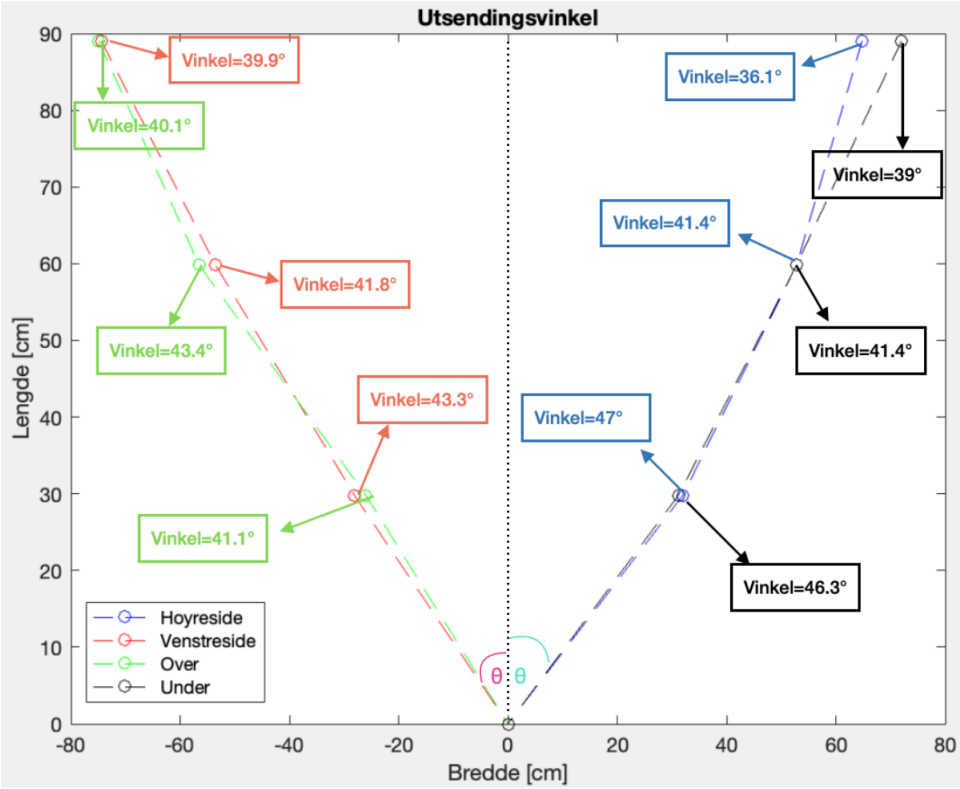
Figur 2.4: Ultralydsensor 1 rotert 90 grader slik at senderen har kontakt med bordet.

Da alle målingene var tatt, beregnet vi utsendingsvinklene ved bruk av likning 2.1.

$$\theta = \arctan\left(\frac{B}{L}\right) \tag{2.1}$$

2.1.1 Resultat av målingene

De beregnede utsendingsvinklene vises i figur 2.5.

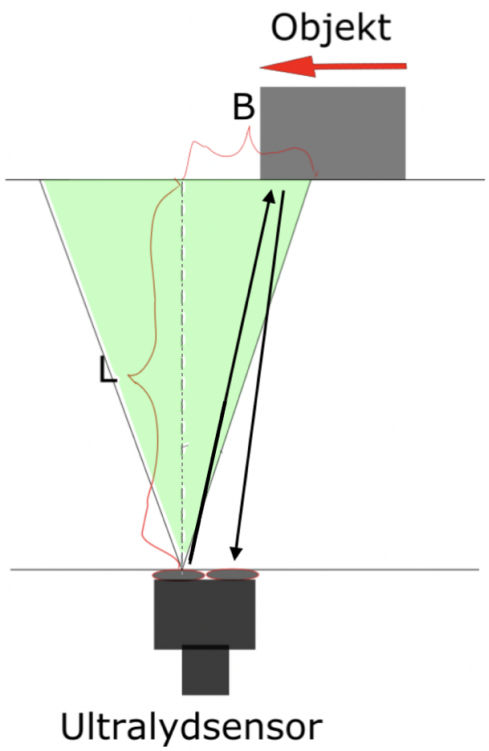


Figur 2.5: Den horisontale og vertikale lydkjeglen vises her i fugleperspektiv.

Fra figur 2.5 ser vi at lydkjeglen er relativt symmetrisk. Venstreside og overside er tilnærmet like, og høyreside og underside er tilnærmet like. De største avvikene finner vi ved lengde 29.7 cm, der differansen mellom høyreside og venstreside er 3.7 grader, mens den er 5.2 grader for oversiden i forhold til undersiden. Avviket ved lengde 89.1 cm viser at utsendingsvinkelen på høyreside er mellom 2.9 og 4 grader mindre enn de andre utsendingsvinklene. Det er vanskelig å si noe om årsaken til disse ulikhetene, men vi ser at utsendingsvinkelen minsker med avstanden. Dette kan skyldes at dess lenger bort refleksjonen foregår, dess mindre ultralyd reflekteres tilbake på grunn av spredningen.

2.2 Deteksjon av forskjellige former

For å undersøke hvor mye formen på et objekt påvirker deteksjonsbredden til ultralydsensoren(Bredden, **B**, fra senterlinjen) benyttet vi oss av totalt seks forskjellige objekter. Figur 2.6 viser en skisse av forsøksoppsettet.



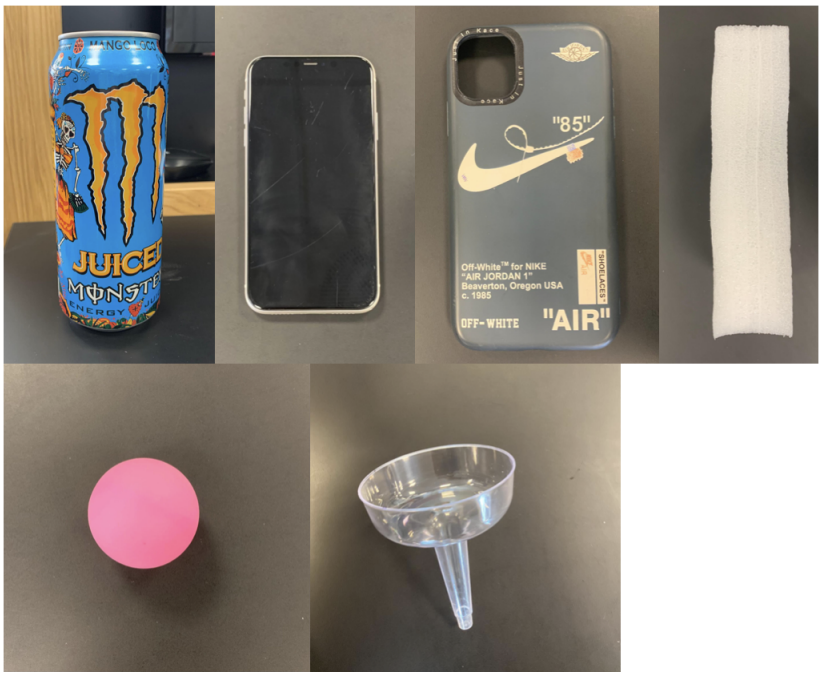
Figur 2.6: Aksene er **L** for lengde, og **B** for Bredde. Den røde pilen viser at objektet blir gradvis flyttet innover til det blir detektert av ultralydsensor 1.

Objektene som ble benyttet i målingene:

- Brusboks
- Mobil, glass-side
- Mobildeksel (Plastikk)

- Isoporflate
- Ping-pong ball
- Champagne glass (Plastikk)

Objektene ble valgt basert på formen de har, og tilgjengeligheten til objektene. Figur 2.7 viser hvordan objektene ser ut.



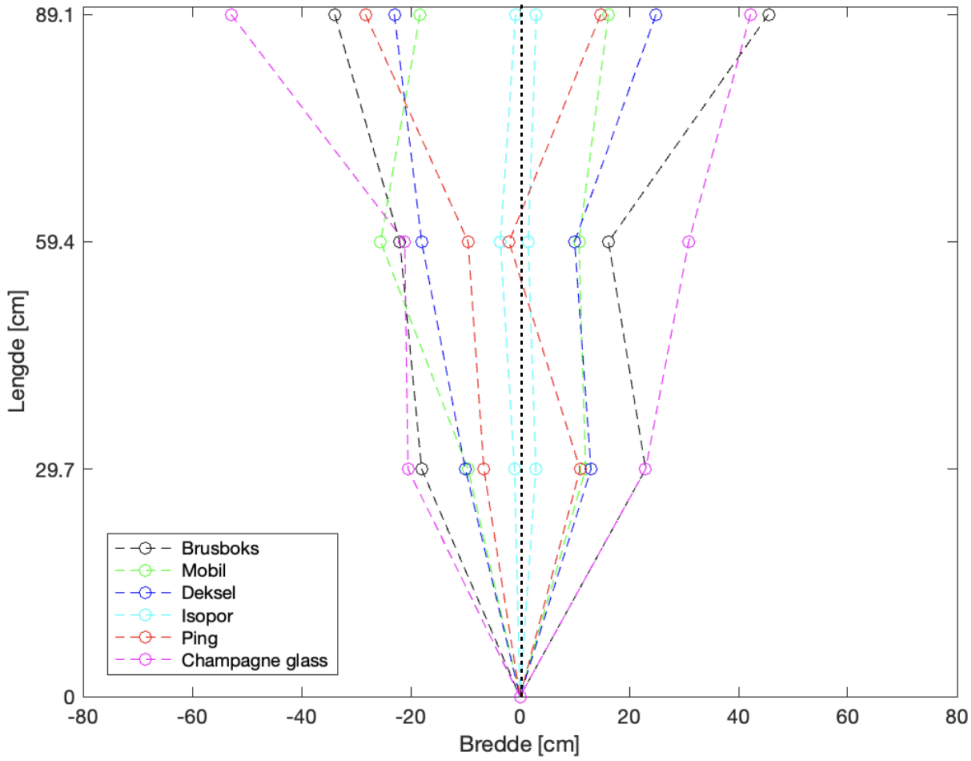
Figur 2.7: En brusboks, en mobil, et mobildeksel, en isoporflate, en pingpong ball og et 'champagne' glass

Denne dokumenteringen ble foretatt på samme måte som dokumenteringen i delkapittel 2.1. Ultralydsensoren var teipet fast i bordet og målte avstand. Et objekt ble så skjøvet innover mot senterlinjen til ultralydsensoren sin sender. Når ultralydsensoren så detekterte objektet, ble dette punktet reverifisert slik at ultralydsensoren hadde en stabil avstandsmåling hver gang. For å være sikker på at det var objektet som det ble målt avstand til, og ikke hånden til vedkommende som foretok målingene, ble objektet så stående

mens vedkommende gikk helt ut av deteksjonssonen til ultralydsensoren. I dette delkapittelet har vi kun målt horisontalt.

2.2.1 Resultat av målingene

Målingene vises i figur 2.8.

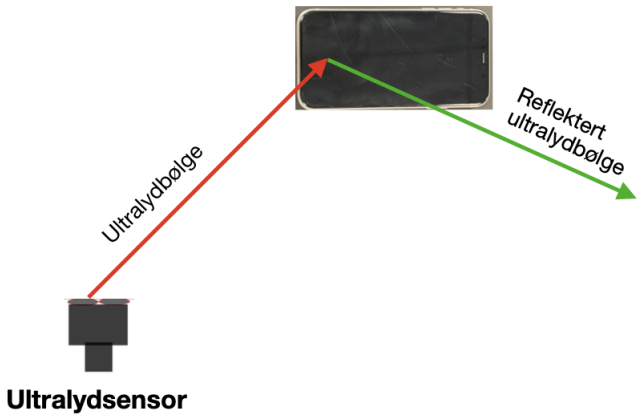


Figur 2.8: Målingene er fremstilt i fugleperspektiv for å gi en bedre oversikt.

Det kommer frem fra figur 2.8 at målingene på høyreside ved lengde 59.4 cm "knekker" inn mot senterlinjen. Dette gjelder alle objektene bortsett fra champagne-glasset. På venstresiden blir deteksjonsbredden større for alle objektene, men økningen er allikevel minimal for flere av objektene. Ved lengde 89.1 cm øker deteksjonsbredden for alle objektene bortsett fra isoporflaten på venstreside.

Brusboks:
I figur 2.8 er dette objektet representert ved fargen svart. Denne er sylinderformet, relativt høy i forhold til de andre objektene, og laget av aluminium. Det kan tenkes at brusboksen sin sylinderformede overflate, vil føre til at ultralydenbølgen bretter seg som et bånd rundt overflaten, og reflekteres tilbake i et bredt spekter av retninger. En av disse vil derfor være motsatt rettet, og i samme retning bølgen kom fra. Høyden på brusboksen gir den i tillegg et større overflateareal sammenlignet med de andre objektene.

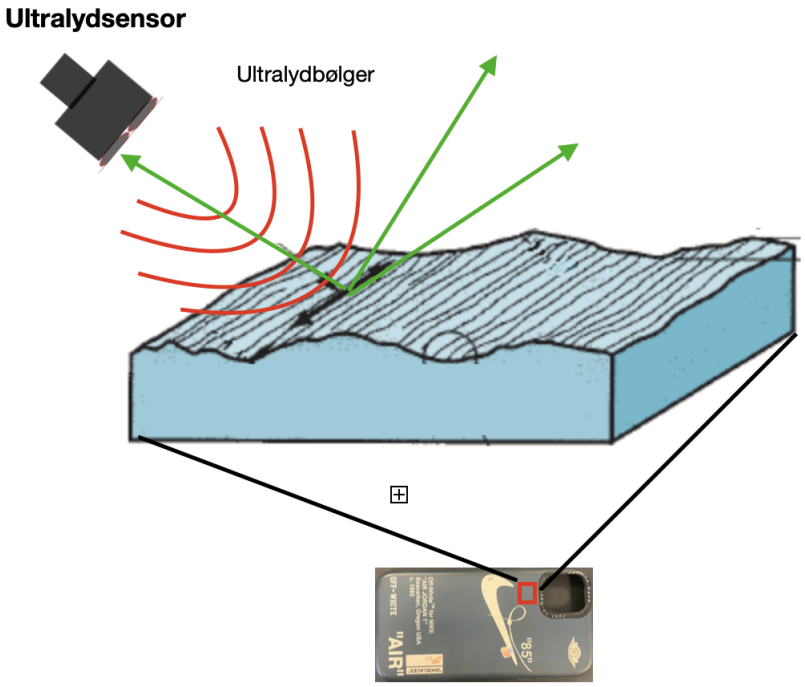
Mobil:
I figur 2.8 er dette objektet representert ved fargen grønn. Mobilen er rektangelformet, flat, og mobilskjermen er laget av glass som gjør dette til et av de glatteste objektene vi testet. Bortsett fra et avvik på venstreside-målingen ved lengde 59.4 cm, har dette objektet en av de lavere deteksjonsbreddene. Dette skyldes antakeligvis at lydbølgene blir reflektert vekk i feil retning, ettersom mobilen står parallelt med sensoren, kombinert med den glatte overflaten. Mobilen må derfor stå nærmere senterlinjen til senderen for å reflektere tilbake til mottakeren på sensoren. Se figur 2.9 for en skisse av dette.



Figur 2.9: Ultralydsensoren sender ut lydbølger som reflekteres vekk når bølgene treffer den glatte overflaten.

Deksel:
I figur 2.8 er dette objektet representert ved fargen blå. Mobildekselet er rektangelformet, flat, og overflaten er delvis ru og laget av gummi. Den har

lik form som mobilskjermen, og fra figur 2.8 har de relativt lik deteksjonsbredde. Forskjellen mellom de to objektene viser seg i målingene ved lengden 89.1 cm. I denne lengden har dekselet en noe bredere deteksjonsbredde, som kan skyldes den rue overflaten på dekselet. Se figur 2.10.



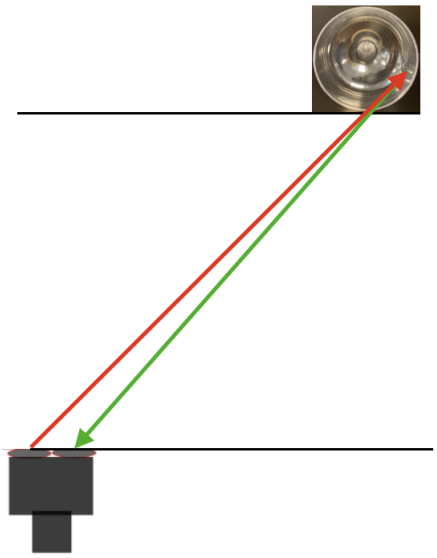
Figur 2.10: Den rue overflaten representerer en veldig liten seksjon av dekselet. Ultralydsensoren sender ut lydbølger (rød bølger) som reflekteres (grønn pil) i flere retninger når bølgene treffer den rue overflaten. [7]

Når lydbølgene treffer en ru overflate vil de treffe små ujevnheter og dermed reflektere i et bredt spekter av retninger.

Isoporflate:
I figur 2.8 er dette objektet representert ved fargen cyan. Isoporflaten har den desidert minste deteksjonsbredden i denne undersøkelsen. Isoporflaten måtte stå nesten rett foran ultralydsensoren for å bli detektert. Årsaken er ikke kjent, men det kan tenkes at lydbølgene trenger igjennom materialet grunnet den lave tettheten.

Pingpong-ball:
I figur 2.8 er dette objektet representert ved fargen rød. Pingpongballen har en sfæriske form og et lite overflateareal. Grunnet den spesielle formen, vil den reflektere ultralydbølgene i mange ulike vinkler. Det er derfor vanskelig å si om enkelt av disse vil reflekteres tilbake til ultralydsensoren eller ikke. Dette kan vi se på figuren ved at det er stor variasjon i deteksjonsbredden. På høyreside ved lengde 59.4 cm måtte ballen helt over senterlinjen før den ga en kontinuerlig avstandsmåling.

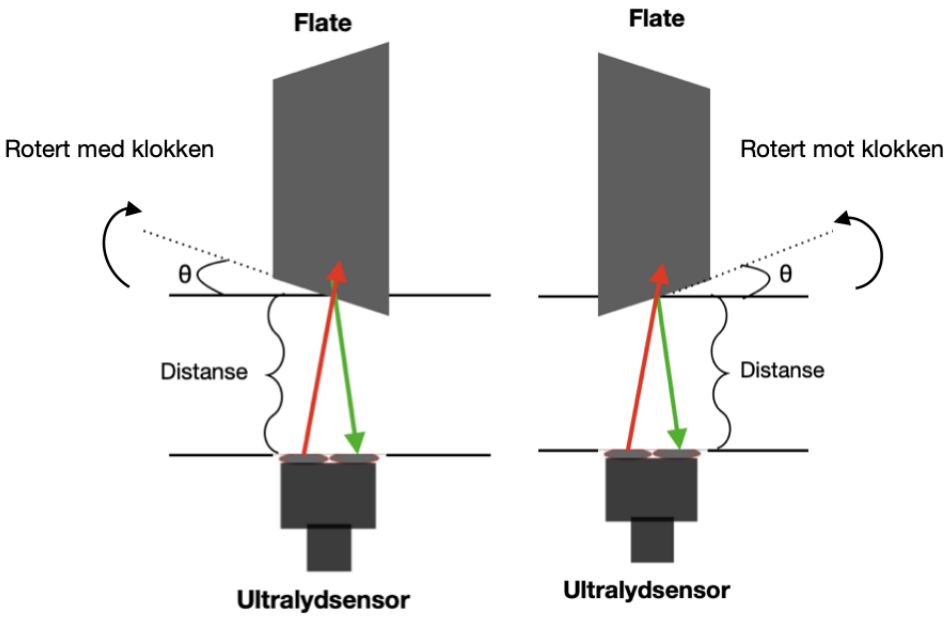
Champagne glass:
I figur 2.8 er dette objektet representert ved fargen rosa. Glasset er formet som et parabol, og er laget av plastikk. Formen på glasset er en fordel siden innsiden av glasset er formet slik at lydbølgene treffer en buet flate som retter seg tilbake mot sensoren. Se figur 2.11.



Figur 2.11: Ultralydsensoren sender ut lydbølger (rød pil) som reflekteres (grønn pil) tilbake når bølgene treffer rett vinkel på glasset.

2.3 Måling av flate med skrå innfallsvinkel

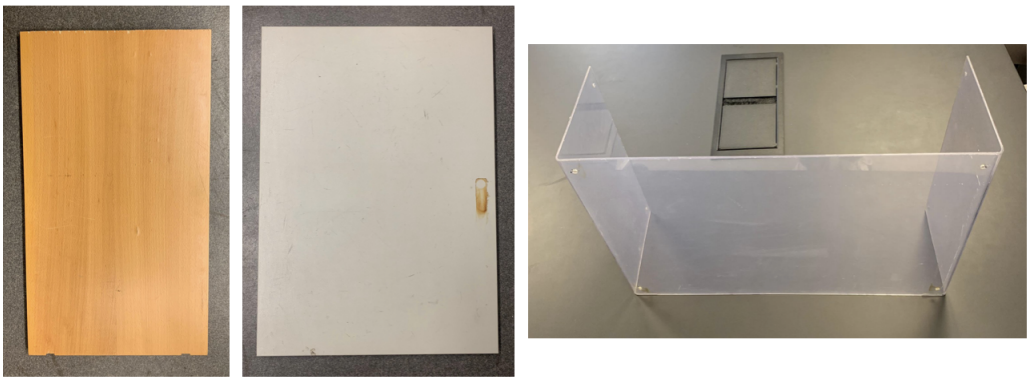
Vi ønsket å dokumentere den maksimale vinkelen en flate kunne stå i, i forhold til sensoren, slik at ultralydsensoren fremdeles klarte å måle avstanden til flaten foran seg. Se figur 2.12.



Figur 2.12: Skisse av forsøksoppsett som viser vinkelen, θ som blir målt.

Det ble totalt testet tre flater. Se figur 2.13 for bilder av disse flatene.

- Treplanke
- Metallplate
- Plastikkplate



Figur 2.13: De tre flatene, en treplate, en metallplate og en plastikkplate.

Målingene ble foretatt for distansene 10 cm og 30 cm. For hvert av forsøkene sto en flate foran sensoren som vist i skisse 2.12. Platen ble så rotert gradvis helt til ultralydsensoren ikke lenger detekterte en avstand. Deretter ble vinkelen dette utgjorde målt med en gradskive. Forsøket ble så repetert med motsatt rotering av flaten.

2.3.1 Resultat av målingene

Tabell 2.1 viser dataene vi har dokumentert.

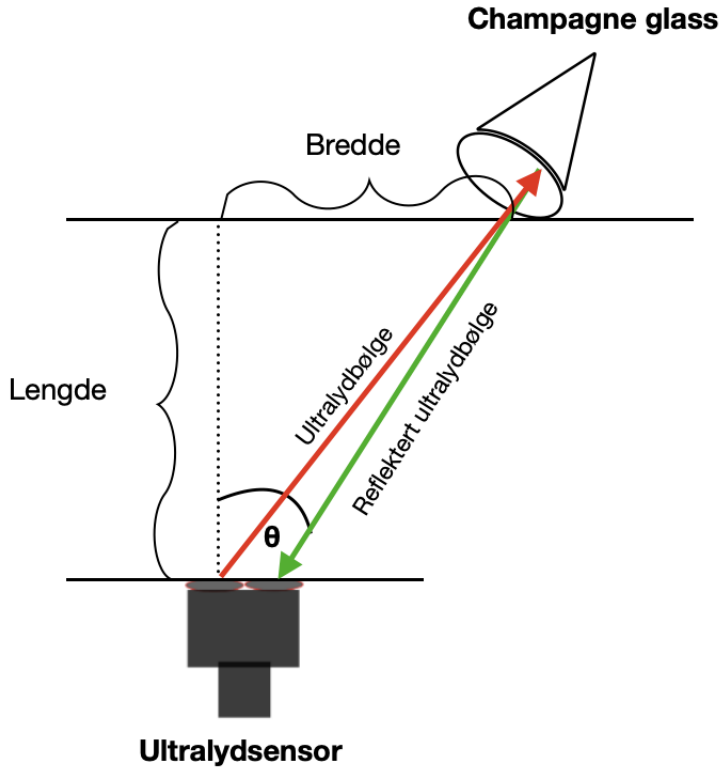
Distanse [cm]	Vinkel med klokken	Vinkel mot klokken
	Treplanke	
10	42°	59°
30	50°	65°
	Metallplate	
10	44°	59°
30	50°	64°
	Plastikkplate	
10	48°	54°
30	45°	46°

Tabell 2.1: Tabellpresentasjon av den maksimale innfallsvinkelen.

Fra tabell 2.1 kan det konkluderes med at flater som blir rotert mot klokken utgjør den største vinkelen som reflekterer ultralyd. Målingene for treplanken og metallplaten er tilnærmet like. Plastikkplaten sine dokumenterte vinkler er derimot mindre enn de to andre, utenom ved 10 cm og en rotasjon med klokken.

2.4 Parabol rettet mot ultralydsensor

I denne undersøkelsen ble champagne glasset som vi benyttet i delkapittel 2.2, rettet mot ultralydsensoren da vi foretok målingene. Formålet med dette forsøket var å dokumentere hvor langt ut et objekt som er rettet mot sensoren kan stå, og fremdeles bli detektert av ultralydsensoren. Se figur 2.14



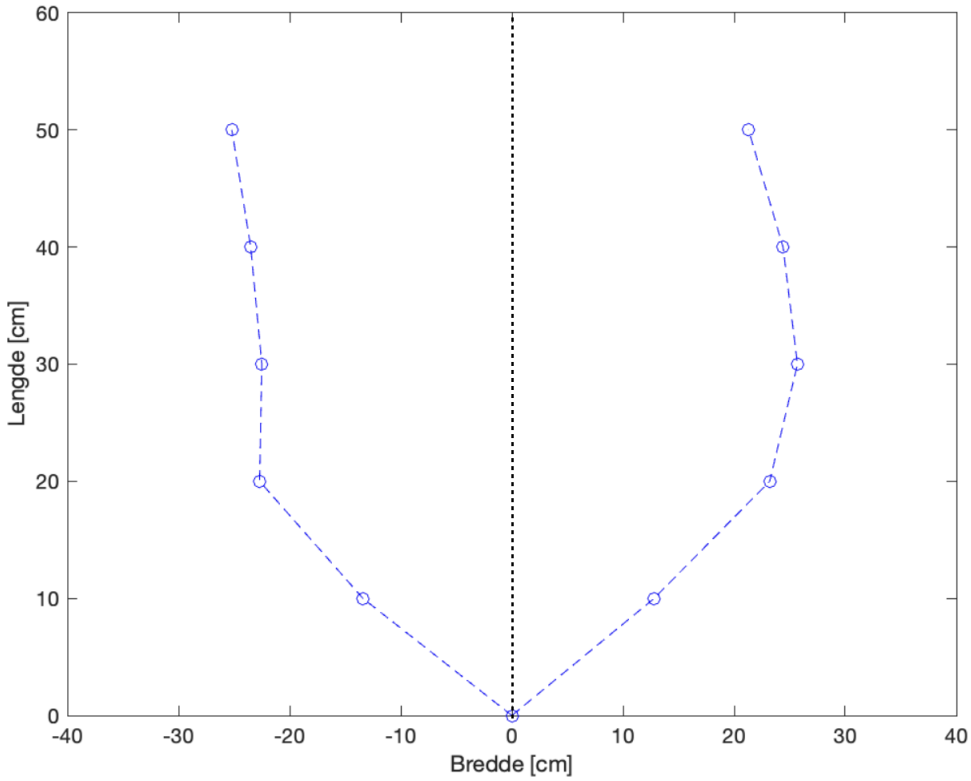
Figur 2.14: Skisse av forsøksoppsettet.

Årsaken til at vi valgte dette glasset skyldes at den har formen til et parabol. Dette ble sett på som en fordel ettersom denne formen kunne vise til en god deteksjonsbredde i delkapittel 2.2. Figur 2.15 viser hvordan glasset ser ut. Bildet til venstre i denne figuren viser måten glasset ble rettet mot sensoren.

Figur 2.15: Champagne-glasset. Bildet til venstre er tatt mot innsiden av glasset, mens bildet til høyre er tatt ovenfra.

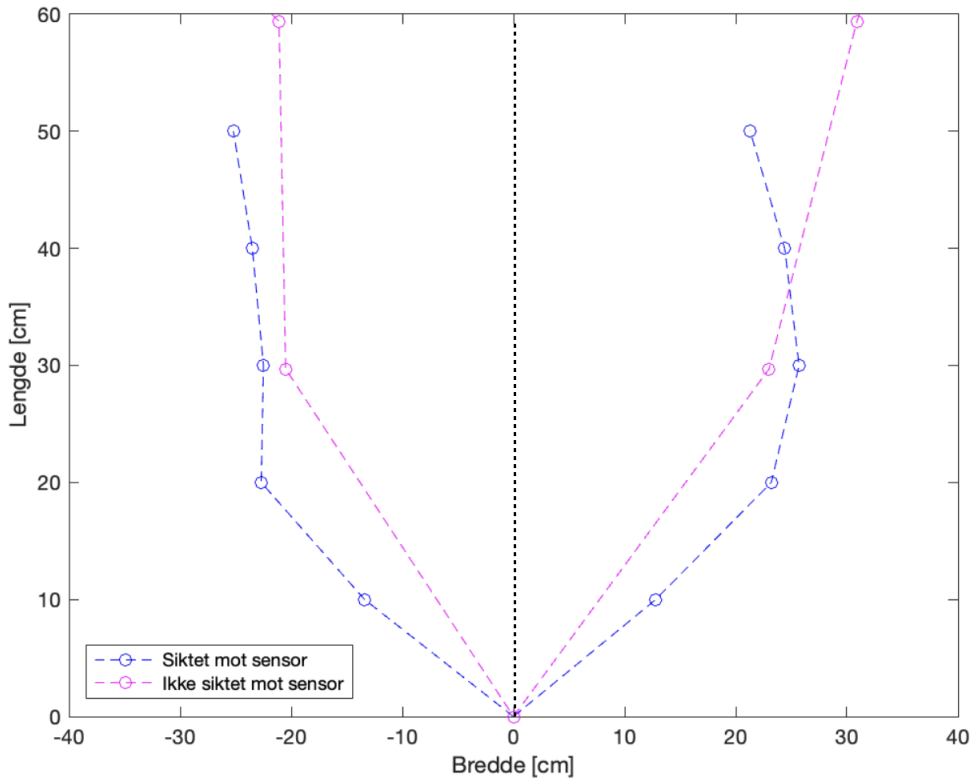
2.4.1 Resultat av målingene

Målingene er fremstilt i figur 2.16.



Figur 2.16: Grafisk fremstilling av målingene

Fra figur 2.16 kommer det frem en asymmetri mellom målingene på høyre- og venstreside. På høyreside er deteksjonsbredden formet som en kurve som bretter seg innover mot senterlinjen. Med ytterste deteksjonsbredde ved Lengde(30). På venstresiden snevrer deteksjonsbredden seg smått innover mot senterlinjen ved Lengde(30), før den bretter seg ut igjen ved Lengde(40) og Lengde(50). I figur 2.17 er målingene fra dette delkapittelet sammenlignet med målingene av glasset fra figur 2.8 i delkapittel 2.2.

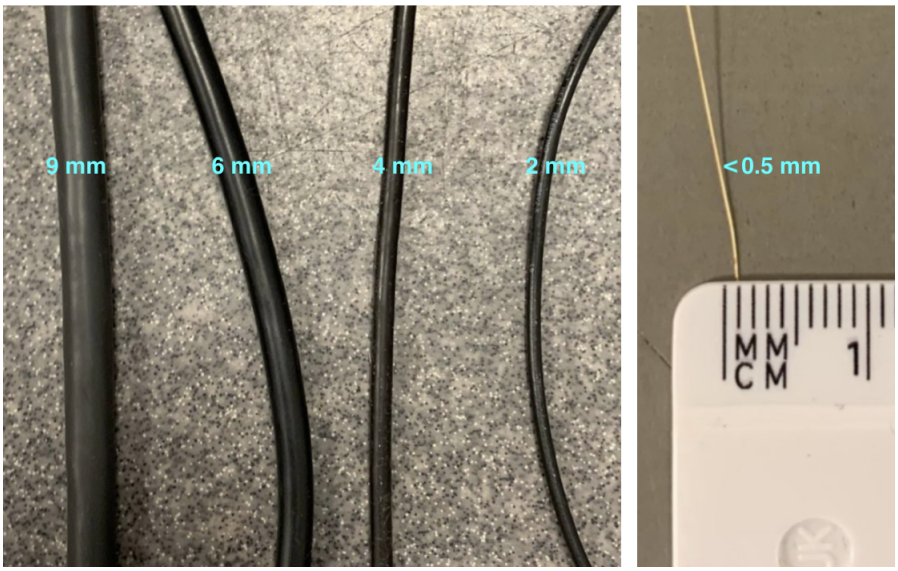


Figur 2.17: Sammenligning med glasset som står parallelt med sensoren, i figur 2.8

Fra figur 2.17 ser vi at deteksjonsbredden av glasset som er siktet mot sensoren er noe bredere enn glasset som står parallelt med sensoren. På høyreside for Lengde(30) og oppover har derimot glasset som står parallelt med sensoren fra delkapittel 2.2 større deteksjonsbredde enn glasset som er rettet mot sensoren. Vi kan dermed konkludere med at det i dette tilfellet ikke utgjorde noen vesentlig forbedring i deteksjonsbredden når objektet er siktet mot sensoren, kontra når objektet står parallelt med sensoren.

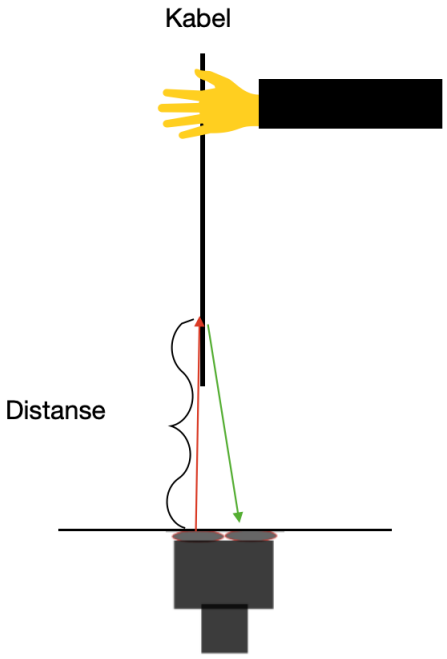
2.5 Minste nødvendige størrelse

I denne undersøkelsen har vi dokumentert hvor smalt et objekt kan være, og fremdeles bli detektert av ultralydsensoren. Det har blitt testet totalt fire kabler av ulik tykkelse. Se figur 2.18.



Figur 2.18: De fire forskjellige kablene som ble testet.

Kablene i figur 2.18 har tykkelse 9, 6, 4 og 2 mm. Figur 2.19 viser hvordan disse målingene ble tatt.

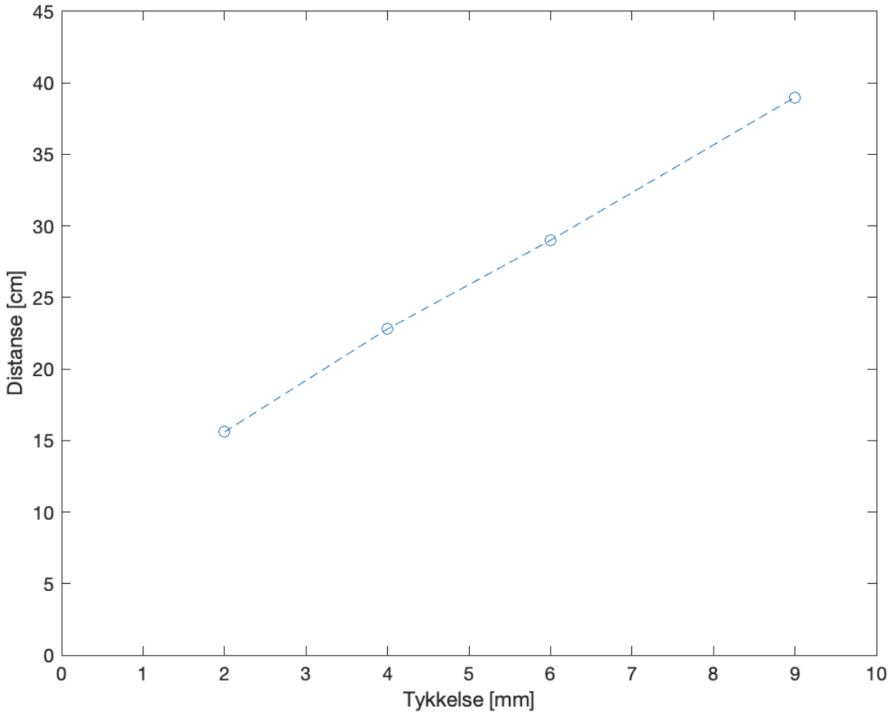


Figur 2.19: Skisse av forsøksoppsett

Kabelen ble holdt ovenifra, og utenfor ultralydsensoren sitt deteksjonsområde. Deretter ble den flyttet gradvis nærmere til den ble registrert av ultralydsensoren.

2.5.1 Resultat av målingene

Figur 2.20 viser den maksimale avstanden der kabelen detekteres av sensoren i y-aksen, mot tykkelsen på kabelen langs x-aksen.

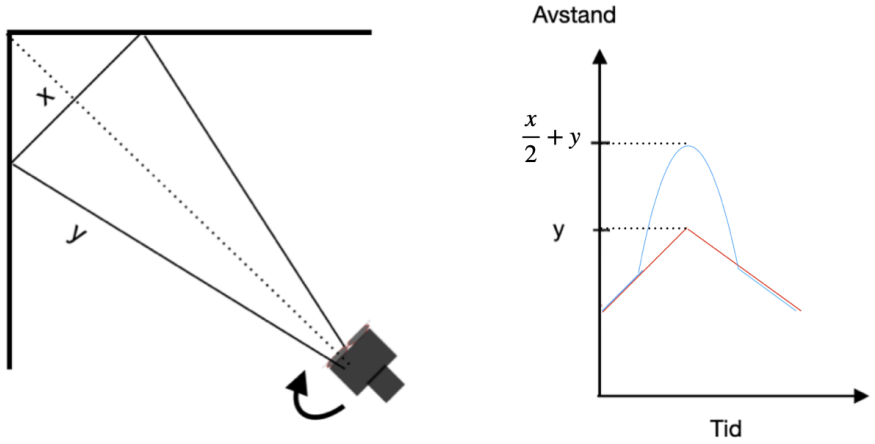


Figur 2.20: Detektert distanse mot tykkelse

Fra figur 2.20 kommer det frem en lineær sammenheng mellom tykkelsen på kabelen og den maksimale detekterte distansen til kabelene. Tykkelser mindre enn 2 mm blir ikke detektert av ultralydsensoren.

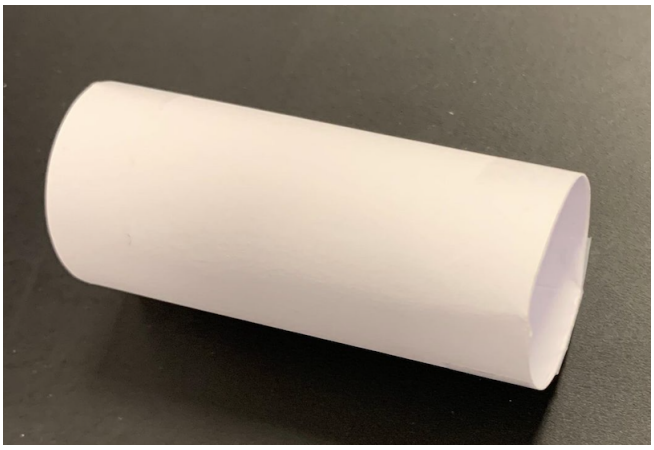
2.6 Hjørnemåling

Et kjent problem når ultralydsensoren skal måle avstand til et hjørne er at ultralyden spretter fra vegg til vegg, før den returnerer til sensoren. Dette fører til at sensoren måler en lengre avstand enn det som faktisk er tilfelle. Se figur 2.21.



Figur 2.21: Skisse av problemstillingen ved hjørnemålinger. Grafen til høyre viser ønsket måling, y, og uønsket måling, $\frac{x}{2} + y$. Legg også merke til pilen som viser at sensoren ble rotert fra venstre til høyre.

En tenkt løsning på dette var å montere et rør på mottakeren til ultralyd-sensoren. På denne måten skal røret fange opp ultralyden som reflekteres fra første trefning, og gi en mer presis avstandsmåling av hjørnet. I figur 2.22 ser man et av rørene vi benyttet.



Figur 2.22: Røret er laget av et A4 ark som er rullet til et sylinder. Deretter er det brukt teip slik at røret holder formen, og slik at det ikke stikker ut noen kanter inni røret.

Forsøkene ble gjort inne i en stor pappeske som kan ses i figur 2.23. Ultralydsensoren var montert på en motor som var festet til robotbil-konstruksjon. Motoren gjorde at ultralydsensoren fikk en konstant bane og hastighet da den ble rotert rundt innsiden av hjørnet.

Figur 2.23: Oppsettet med et av rørene festet til mottakerøyet på ultralydsensoren. Her står sensoren i posisjon A med ultralydsensoren siktende inn i hjørnet.

Kodeutdraget 2.2 viser Matlab-koden som styrer motoren.

Kode 2.2: Utdrag fra `SikteiBoks.m`

```
168         start(motorA)
169         if Tid(k) ≥ 2
170             if VinkelPosMotorA(k) ≥ 90
171                 stop(motorA);
172             else
```

```
173                     motorA.Speed = 7;
174                     start(motorA);
175                 end
176             end
```

Vi ønsket at målingene skulle ha en identisk start for å lettere kunne sammenligne målingene med ulik lengde på rørene. Fra kodeutdraget 2.2 vil motoren begynne å rotere ultralydsensoren etter det har gått to sekunder. Når motoren så har rotert 90 grader stopper motoren, og målingen avsluttet. Se så kodeutdrag 2.3 for definering av avstandsmåling.

Kode 2.3: Utdrag fra `SikteiBoks.m`

```
112                 Avstand(k) = double(readDistance(mySonicSensor));
113                 Avstand_filtrert(k) = ...
                        IIR_filter(Avstand_filtrert(k-1), ...
                        Avstand(k), 0.8);
```

Fra Kodeutdraget 2.3 i linje 112 blir avstandsmålingen definert. I linje 113 blir avstandsmåling med IIR filtrering brukt, der vi har brukt en alfa på 80%. Dette vil si at "nåværende" måling er vektet 80%, mens forrige måling blir vektet 20%.

Rørlengder:

- Uten rør
- 4 cm langt rør
- 6 cm langt rør

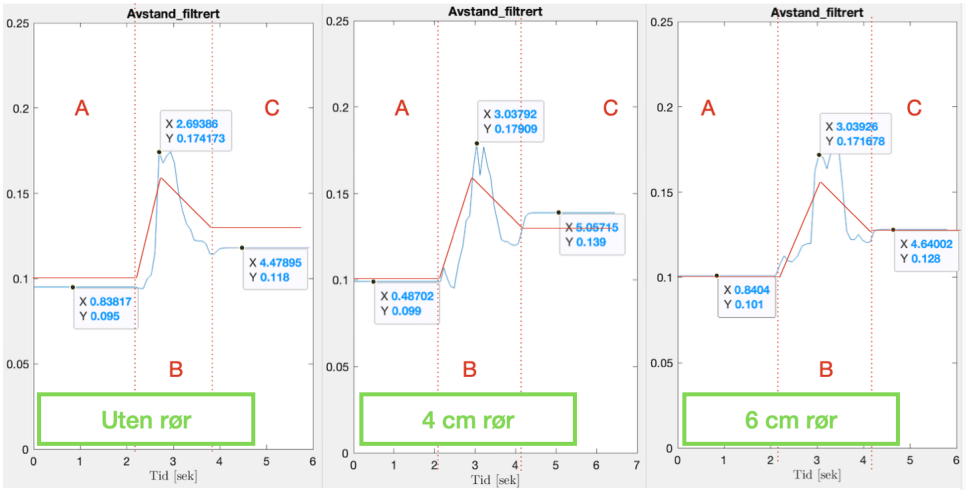
Årsaken til at vi valgte disse lengdene var at noe kortere rør enn 4 cm ikke gjorde noen vesentlig forskjell. Og noe lenger enn 6 cm førte til at røret kom nær veggen.

2.6.1 Resultat av målingene

Målingene ble tatt ved to forskjellige avstander, som vi har valgt å kalle for **posisjon A** og **posisjon B**. Ved posisjon A er avstanden til veggen rett foran ultralydsensoren 10 cm, mens den er 19 cm ved posisjon B. Figur 2.24 viser målingene fra Posisjon A med de forskjellige rørene. Det røde signalet er den reelle avstanden mellom sensoren og veggene. Mens det blå signalet er den målte avstanden med ultralydsensoren.

Posisjon A:

- Seksjon A: Ultralydsensoren står i startposisjon og sikter rett frem. Avstand til veggen = 10 cm
- Seksjon B: De to sekundene har gått, og sensoren roteres 90 grader med klokken. Avstand til hjørnet = 16 cm
- Seksjon C: Sensoren har nådd sluttposisjon, og sikter i veggen. Avstand til veggen = 13 cm



Figur 2.24: Målingene fra Posisjon A, den sensormålte avstanden er i fargen blå, mens den avstanden målt med målebånd er i rød. Figurene er delt opp i tre seksjoner, A, B og C. Nederst til venstre i hver graf står hvilket rør som er benyttet.

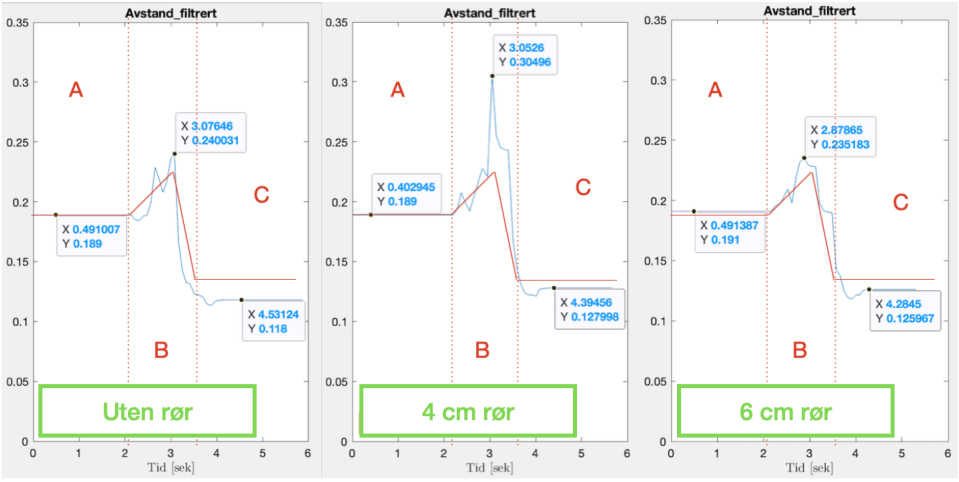
Ut i fra figur 2.24 kan det ikke tydes noen bedring i presisjonen ved bruk

av et rør på mottakeren. Uten rør måler sensoren 17.4 cm til hjørnet. Med et 4 cm langt rør måles det 17.9 cm til hjørnet, og og med et 6 cm langt rør måles det 17.2 cm til hjørnet. Hjørnemålingen uten rør gjør det altså tilnærmet like bra som hjørnemålingene med rør.

Figur 2.25 viser målingene tatt i posisjon B.

Posisjon B:

- Seksjon A: Ultralydsensoren står i startposisjon og sikter rett frem. Avstand til veggen = 19 cm
- Seksjon B: De to sekundene har gått, og sensoren roteres 90 grader med klokken. Avstand til hjørnet = 23 cm
- Seksjon C: Sensoren har nådd sluttposisjon, og sikter i veggen. Avstand til veggen = 13 cm



Figur 2.25: Målingene fra Posisjon B. Den sensormålte avstanden er i fargen blå, mens avstanden målt med målebånd er i rød. Nederst til venstre i hver graf står hvilket rør som er benyttet.

Fra figur 2.25 måles avstanden til hjørnet uten et rør til 24 cm. Med et 4 cm langt rør måles det 30.5 cm til hjørnet, og med et 6 cm langt rør måles det 23.5 cm til hjørnet.

Det kan dermed konkluderes med at et rør rundt mottakeren på ultralydsensoren ikke utgjør noen vesentlig forskjell for hjørnemålingene.

2.7 Crosstalk

Crosstalk er når det oppstår interferens mellom ultralyden fra to ulike sensorer, samt når ultralyden fra den ene sensoren mottas av den andre, og motsatt. Ettersom avstandsmålingene beregnes ut i fra tiden det har gått siden ultralydpulsen ble sendt ut, vil feil tid gi feil avstand. Se likning 2.2.

$$Avstand = \frac{Lydens\ hastighet \cdot Tid}{2} \tag{2.2}$$

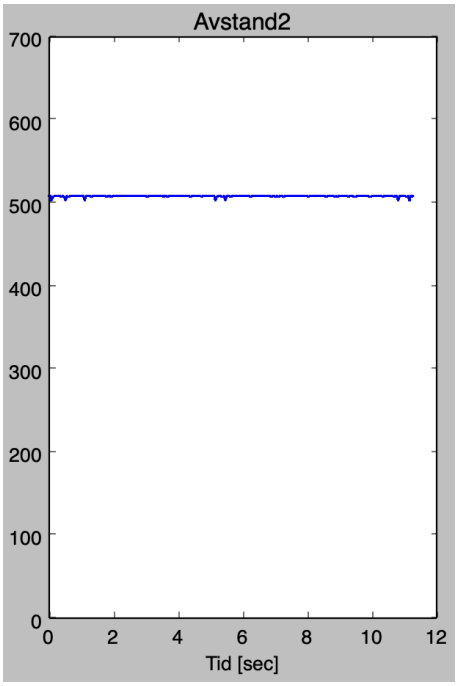
Dette er derfor noe man må unngå dersom man benytter flere ultralydsensorer.

I dette kapittelet har vi utarbeidet en multipleks løsning for bruk av flere ultralydsensorer på en gang. Multipleksing vil si at ultralydsensorene tar målinger i en alternerende rekkefølge. På denne måten unngår man at én sensor registrerer en annen sensor sin ultralyd.

For å dokumentere crosstalken, ble det benyttet to ultralydsensorer som sto side om side i modusen `distance`(ref kodeutdrag 2.1). Sensorene ble plassert side om side, og siktet mot en flate rundt 50 cm på avstand. Se figur 2.26.

Figur 2.26: Oppsettet viser to ultralydsensoren som sikter mot en flate.

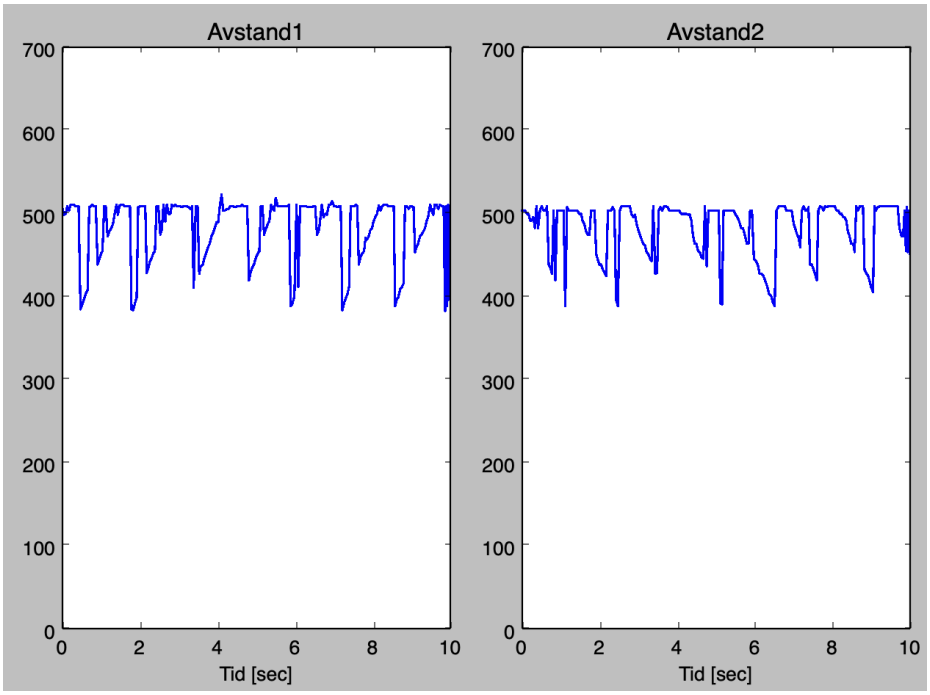
Da kun den ene av sensorene var påskrudd og sendte ut ultralyd, ga dette følgende måling. Se figur 2.27.



Figur 2.27: Måling fra den ene ultralydsensoren.

Som forventet får vi ingen crosstalk når vi kun bruker én ultralydsensor.

Til sammenligning fikk vi følgende resultat da begge ultralydsensorene ble påskrudd og sendte ut ultralyd. Se figur 2.28.



Figur 2.28: Enheten langs y-aksen er i millimeter. Avstandsmålingene bærer tydelig preg av crosstalk.

I figur 2.28 hopper avstandsmålingen mellom 40 og 50 cm, uten å få en konstant avstandsmåling. Dette beviser at crosstalk er et problem, og det er derfor noe en må være varsom med dersom det skal utføres eksperimenter med flere ultralydsensorer.

2.7.1 Multipleks løsning

Modusen `distance` har en parameter som heter `silent`. Den er som standard satt til `False`, men kan settes til `True` om ønskelig. Settes den til `True` vil sensoren skru seg av etter den har tatt én måling [3]. I kodeutdrag 2.4 vises det hvordan dette gjøres.

Kode 2.4: Utdrag fra `P14_Crosstalk.py`

```
171             sensor1 = ...
                    myUltrasonicSensor1.distance(silent=True)
```

I dokumentasjonen til ultralydsensoren [3] opplyses det om at for hyppig bruk av denne parameteren vil føre til at sensorene henger seg opp. Dette er noe vi opplevde mange ganger da vi testet programmet, og det gir følgende feilmelding i terminalen, se figur 2.29

```
  File "/home/robot/Prosjekt0X_BeskrivendeTekst/P14_Crosstalk.py", line 176, in main
OSError: [Errno 5] EIO:

Unexpected hardware input/output error with a motor or sensor:
--> Try unplugging the sensor or motor and plug it back in again.
--> To see which sensor or motor is causing the problem,
    check the line in your script that matches
    the line number given in the 'Traceback' above.
--> Try rebooting the hub/brick if the problem persists.
```

Figur 2.29: Feilmeldingen som kan oppstå når man setter sensorene til `silent=True` for hyppig

Linje 176 som feilmeldingen refererer til er der den ene ultralydsensoren settes til å ha `silent=True`. Dette kommer vi til i kodeutdrag 2.5. Dersom denne feilmeldingen oppstår må man enten koble ut og inn igjen sensoren, eller restarte EV3'en.

Kode 2.5: Utdrag fra `P14_Crosstalk.py`

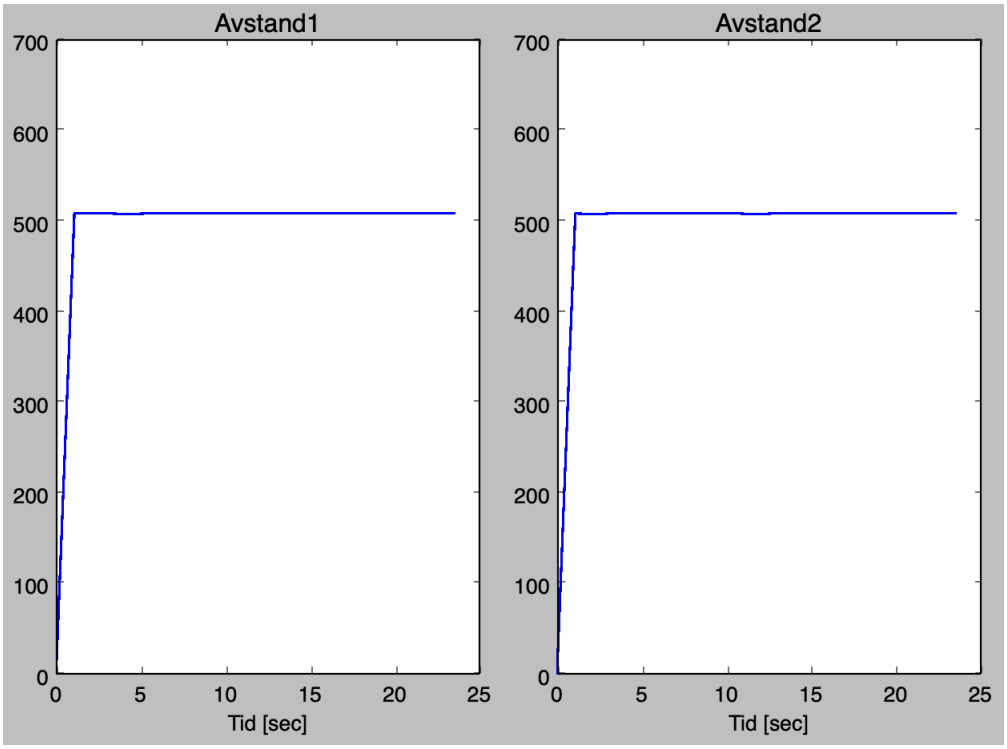
```
171             sleep(0.2)
172             sensor1 = ...
                    myUltrasonicSensor1.distance(silent=True)
173             Avstand1.append(sensor1)
174
175             sleep(0.2)
176             sensor2 = ...
                    myUltrasonicSensor2.distance(silent=True)
177             Avstand2.append(sensor2)
```

Kodeutdraget 2.5 viser hvordan en kan utføre multipleksing med ultralydsensoren. Ettersom hele programmet kjøres i en `While=True` løkke, vil koden bli kjørt for hver iterasjon av k.

På linje 171 er funksjonen `sleep` benyttet. Denne pauser programmet i antall sekunder spesifisert i funksjonskallet. I dette tilfellet har vi brukt 0.2 sekunder ettersom en kortere pause enn dette førte til at sensorene hengte seg opp slik som vist i figur 2.29.

På linje 172 definerer vi variabelen `sensor1` som setter ultralydsensor1 i "stillemodus", og i linje 173 legges denne avstandsmålingen til listen `Avstand1`.

På linje 175 til 177 repeteres dette for ultralydsensor 2. I figur 2.30 ses målingene tatt med den multiplekse løsningen.



Figur 2.30: Multipleksing

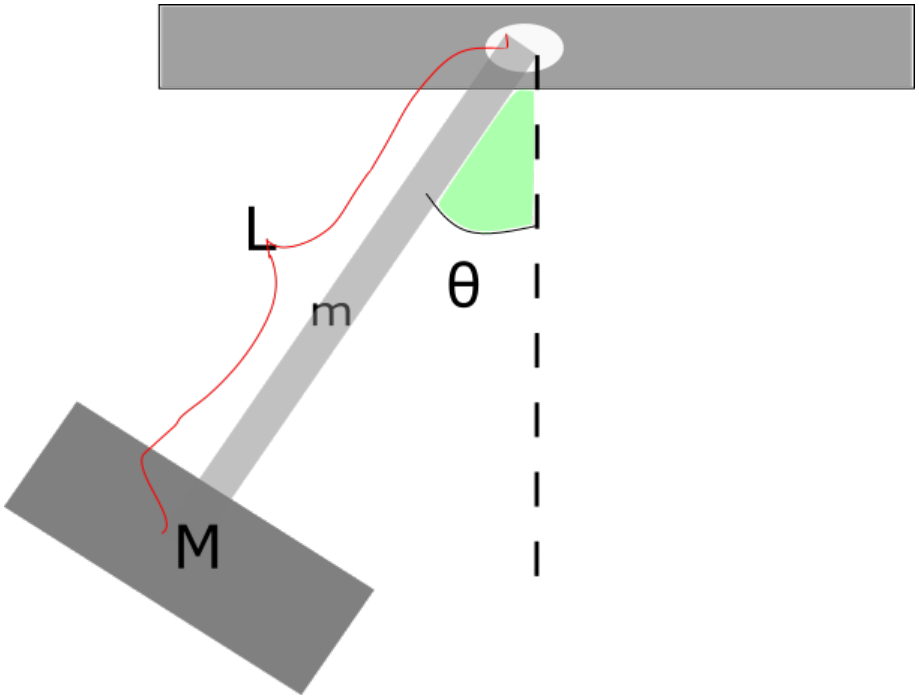
Avstandsmålingene i figur 2.30 er tatt med samme oppsett som vist i figur 2.26. Avstandsmålingen er konstant på litt over 50 cm, og det er ikke lenger tegn til interferens. Vi kan derfor konkludere med at den multiplekse løsningen fungerer, men på grunn av det påtvungne intervallet på 0.2 sekunder mellom hver måling er den ikke effektiv til alle formål. Årsaken til at målingene ser ut til å starte på 0

Kapittel 3

Modellering av pendel

I dette kapittelet har en matematisk modell blitt tilpasset en fysisk modell av et pendel. Formålet med dette prosjektet er å demonstrere hvordan man modellerer et roterende mekanisk system ved hjelp av Matlab og EV3'en.

En illustrasjon av pendelen vi konstruerte til det fysiske eksperimentet kan sees i figur 3.1. Pendelen består av en metallstang med en last i enden bestående av EV3'en, gyrosensoren samt noen metallklosser for ekstra vekt og stabilitet.



Figur 3.1: Skisse av pendelet. Her er '**m**' massen til stangen, '**M**' er massen til lasten, og '**L**' er lengden fra rotasjonspunktet til gravitasjonspunktet for pendelet. 'θ' er endringen i vinkel fra likevektspunktet.

3.1 Utstyrsliste

- Lego mindstorms EV3 med gyrosensoren.
- Metall stang (1.00m, 0.180kg)
- Metall last (0.370kg)

3.2 Konstruksjonen av pendelet

For å verifisere den matematiske modellen, ble pendelet i figur 3.2 konstruert.

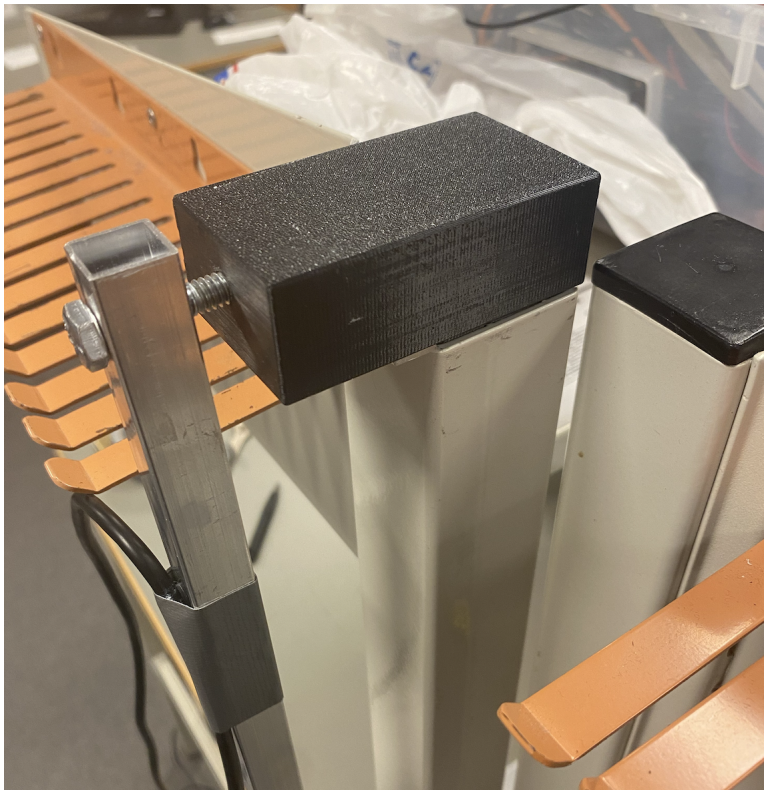
Figur 3.2: Her ser vi pendelen hengt opp (til venstre) og liggende på bordet (til høyre).

Nedenfor er et nærmere bilde av lasten i enden av pendelen.

Figur 3.3: Her ser vi lasten i enden av metallstangen i form av EV3'en, gyrosensoren og metallklossene som er teipet fast.

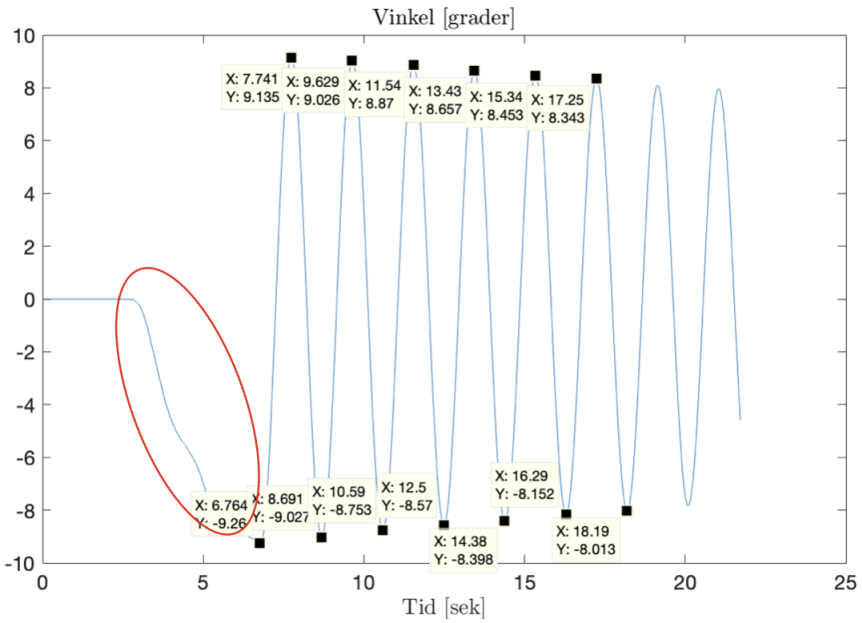
Nederst på metallstangen har vi montert EV3'en sammen med gyrosensoren, og tre metallklosser. Metallklossene ble plassert på andre siden av EV3'en. Dette var for å ha en motvekt for EV3'en og gyrosensoren, slik at den hang mest mulig stabilt. På denne måten hindret vi at den svingte på tvers av fartsretningen.

Metallstangen er festet i en kloss med en skrue. Se figur 3.4.



Figur 3.4: Her ser vi hvordan metallstangen er festet.

Nedenfor i figur 3.5 vises en graf av pendelens frie bevegelse i rommet.



Figur 3.5: Pendelens vinkelposisjon som en funksjon av tiden. Sirkelen i rødt viser posisjonen til pendelen når den blir flyttet manuelt ut til siden.

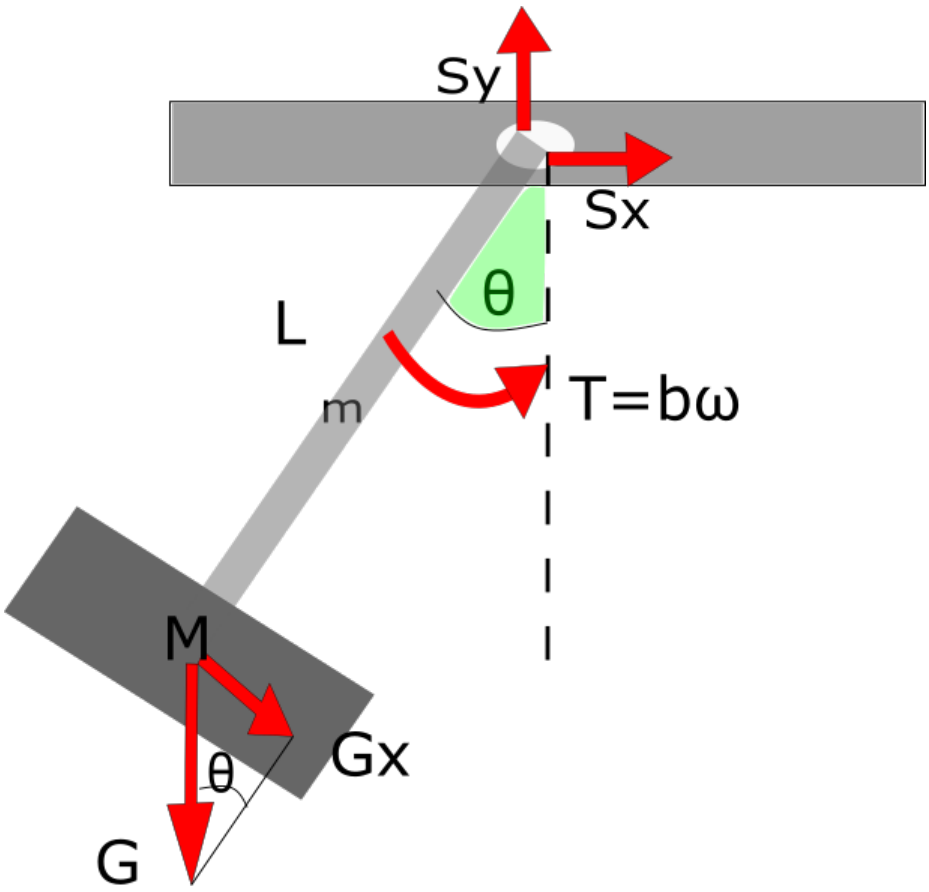
Grafen i figur 3.5 viser pendelens vinkelposisjon som en funksjon av tiden. Pendelen ble løftet fysisk ut til en av sidene, for så å bli sluppet i dens naturlige bevegelse etter 6.76 sekunder. Topp- og bunnpunktene er tatt med for videre beregninger i delkapittelet 3.3 om den matematiske modelleringen. Det er også denne grafen(figur 3.5) vi skal bruke for å verifisere den matematiske modellen i slutten av kapittelet.

3.3 Matematisk modellering av en pendel

I denne delen vil vi vise hvordan forståelsen for fysiske prinsipper lot oss utlede likningene for den matematiske modellen. Først vil vi se på en modell av et enkelt pendel.

Som vist i fritt-legeme diagrammet i figur 3.6 så er kreftene som gjør et arbeid på pendelet, gravitasjonskraften, og reaksjonskreftene i opphengspunktet. Vi har også inkludert en kraft, $\mathbf{T}$. Dette er kraften som kommer

fra friksjonen i opphengspunktet og luftmotstanden. Vi antar en viskøs friksjonsmodell for denne kraften.



Figur 3.6: Fritt-legeme diagram for pendelet. Her er **G** gravitasjonskraften, **T** de viskøse friksjonsmomentet, og **Sy**/**Sx** er friksjonen i opphengspunktet.

For å utlede en bevegelseslikning for pendelet, summerer vi alle momentene ved pendelet. Den letteste måten å gjøre dette på er å summere momentene rundt rotasjonspunktet. På denne måten blir momentene Sx og Sy kansellert. Fra kapittel 2 (likning 2.45) i bok? får vi at momentbalansen til systemet er gitt ved formel 3.1.

$$J\frac{d^2\theta}{dt^2} = \sum T_i \tag{3.1}$$

Der T_i er de ulike momentene. Summert rundt opphengspunktet får vi følgende momentbalanse gitt av formel 3.2.

$$J\frac{d^2\theta}{dt^2} = -(M+m)\cdot g \cdot Lg \cdot \sin(\theta) - b\frac{d\theta}{dt} \tag{3.2}$$

Her er Lg lengden fra opphengspunktet til gravitasjonspunktet (ikke nødvendigvis lik lengden L til pendelet), og J er treghetsmomentet. Omrokkerer vi litt på ligningen ender vi opp med følgende andre ordens differensial likning 3.3.

$$J\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + (M+m)\cdot g \cdot Lg \cdot sin(\theta) = 0 \tag{3.3}$$

Som bare har lineære faktorer, bortsett fra $\sin(\theta)$. For å løse dette bruker vi en liten-vinkel tilnærming. Liten-vinkel tilnærmingen sier at for små vinkler er $\sin\theta \approx \theta$. Derfor blir $\sin\theta$ erstattet av θ i likning 3.3. På denne måten blir vi kvitt den ikke-lineariserte faktoren som inneholder $\sin\theta$, og ender opp med den lineariserte likningen 3.4. Grunnen til at vi ønsker en ligning med bare lineære faktorer er at vi kan sette opp ligningen på en standard form.

$$\frac{d^2\theta}{dt^2} + \frac{b}{J}\frac{d\theta}{dt} + \frac{(M+m)\cdot g \cdot Lg}{J}\theta = 0 \tag{3.4}$$

Parametrene M og m fåes ved å veie utstyret vi har brukt. Mens b, J og Lg må regnes ut. Dersom vi antar at massen til pendelet er konservert på enden av pendelet vil treghetsmomentet og massesenteret være som vist i ligningene nedenfor.

$$J = (M+m)L^2 \tag{3.5}$$

$$Lg = L \tag{3.6}$$

I vårt tilfellet er ikke massen konsentrert rundt enden av pendelet. Vi antar at massen er fordelt jevnt over metallstangen som er brukt (m), mens

massesenteret til pendelet er sentrert i midten av objektet. Dette gir oss følgende ligninger for utregning av J og Lg.

$$J = \frac{mL^2}{3} + ML^2 \tag{3.7}$$

$$Lg = (Ml + 0.5ml)/(M + m) \tag{3.8}$$

Hvis massen M i systemet er mye større enn m, vil ligningene 3.5 og 3.6 være tilstrekkelig. Det er ikke tilfellet for vårt system, og vi benytter oss derfor av ligningene 3.7 og 3.8.

Det gir oss følgende verdier for parametrene J og Lg

$$J = \frac{0.180 \cdot 0.935^2}{3} + 0.654 \cdot 0.935^2 = 0.624 \tag{3.9}$$

$$Lg = (0.654 \cdot 0.935 + 0.5 \cdot 0.180 \cdot 0.935)/(0.654 + 0.180) = 0.834. \tag{3.10}$$

Standard form for 3.4 er gitt ved

$$\theta'' + 2\zeta\omega_n\theta' + \omega_n^2\theta = 0 \tag{3.11}$$

Setter vi denne opp mot differensiallikningen i 3.4 får vi følgende likninger for parametrene våre:

$$2\zeta\omega_n = \frac{b}{J} \tag{3.12}$$

$$\omega_n^2 = \frac{(M + m) \cdot g \cdot Lg}{J} \tag{3.13}$$

I figur 3.1 har vi notert verdiene i hvert av topp- og bunnpunktene. Disse verdiene har vi brukt til å regne ut den naturlige frekvensen. Perioden til funksjonen kan approksimeres ved å ta differansen i x-verdien(tiden) på to etterfølgende toppunkt (x_n, y_n) og (t_{n-1}, y_{n-1}) (her er t_n og t_{n-1} tiden ved de to toppunktene (t_n, y_n) og (t_{n-1}, y_{n-1})). Siden periodetiden mellom toppunktene varierer, bruker vi den gjennomsnittlige periodetiden til hele signalet. Dette har vi gjort ved å ta differansen (i tid) mellom det siste og første toppunktet, for så å dele på totalt antall toppunkter, minus en. Formelen for periodetiden blir som vist i ligningen nedenfor.

$$T = \frac{1}{n-1} * (t_n - t_1) \tag{3.14}$$

Dette gir oss en periodetid på $T \approx 1.9s$. Den dempede egenfrekvensen er gitt ved

$$\omega_d = \frac{2\pi}{T} \tag{3.15}$$

En periodetid på 1.9 gir da en dempet egenfrekvens på $\omega_d \approx 3.3$. Dempekoeffisienten til et andreordens system kan bli tilnærmet ved formelen nedenfor.

$$\zeta = \frac{-ln(OS)}{\sqrt{\pi^2 + \ln OS^2}} \tag{3.16}$$

Her er OS(oversving) forholdet mellom y-verdien i bunn-/toppunktet (t_n, y_n) og det påfølgende topp-/bunnpunktet (t_{n+1}, y_{n+1}).

$$OS = \frac{|y_{n+1}|}{|y_n|} \tag{3.17}$$

Studerer vi grafen på figur 3.1 ser vi at bunnpunktene ser ut til å ha en større dempingskoeffisient enn toppunktene (bunnpunktene avtar fortere i absoluttverdi enn toppunktene). Vi behandler derfor topp- og bunnpunktene hver for seg. På bakgrunn av dette må vi dele høyresiden av ligning 3.16 på to, siden perioden vi tar forholdet mellom har doblet seg. For å få et nøyaktig estimat av ζ, summerer vi alle kalkulerte verdier for ζ for de forskjellige ekstremalpunktene, for så å ta gjennomsnittet av disse. Dette gjør at vi ender opp med en ligning for ζ som vist i ligningen nedenfor

$$\zeta_{topp/bunn} = \frac{1}{n-1} \sum_{n=1}^{n} \frac{-ln(OS_{topp/bunn})}{2 \cdot \sqrt{\pi^2 + (\ln(OS_{topp/bunn}))^2}} \tag{3.18}$$

Denne brukte vi som sagt en gang for toppunktene, for så å bruke den igjen på bunnpunktete. Deretter summerte vi de to verdiene, og delte på to (for å få gjennomsnittet av disse). I vårt eksperiment er $n = 6$ for toppunktene og $n = 7$ for bunnpunktene. Dette gir oss et estimat på $3.37 \cdot 10^{-}3$ for ζ. Basert på estimatene våre av ζ og ω_d kan vi regne oss fram til et estimat for ω_n ved bruk av følgende relasjon

$$\omega_n = \frac{\omega_d}{\sqrt{1 - \zeta^2}} \tag{3.19}$$

Med en estimert verdi for ζ så lavt som i vårt eksperiment vil $\omega_n \approx \omega_d$. Videre kan vi regne oss fram til estimater av parametrene J og b ved bruk av ligningene 3.6 og 3.11 på forrige side. Fra disse kommer det frem følgende sammenhenger

$$J = \frac{(M + m) \cdot g \cdot Lg}{\omega_n^2} \tag{3.20}$$

$$b = 2\zeta\omega_n J \tag{3.21}$$

Med våre estimater av de forskjellige variablene, ender vi med et treghetsmoment på $J \approx 0.627$ og friksjonskoeffisienten $b \approx 0.0139$. Det er verdt å merke seg at den matematiske tilnærmingen for treghetsmomentet (J = 0.624),som vi regnet ut tidligere i kapittelet, ser ut til å være en god approksimasjon basert på det fysiske eksperimentet.

Satt inn i ligning 3.5 får vi følgende differensiallikning for bevegelsen av pendelet.

$$\frac{d^2\theta}{dt^2} + \frac{0.0139}{0.627}\frac{d\theta}{dt} + 10.88\theta = 0 \tag{3.22}$$

Denne skal vi videre verifisere ved bruk av Matlab.

3.4 Kode for plotting av pendelen og verifisering av den matematiske modellen

Først skal vi se på hvordan vi fikk plottet vinkelposisjonen til pendelet som en funksjon av tiden. Som sagt tidligere har EV3'en en tilhørende gyrosensor som kan måle vinkelposisjonen til EV3'en i forhold til startsposisjonen. Problemet med denne funksjonen til gyrosensoren, er at den bare har en oppløsning på 1°. Så når vi prøvde å bruke den til å plotte vinkelposisjonen, fikk vi topp- og bunnpunkter som var avkappet. Det var altså en alt for liten nøyaktighet på målingene. Gyrosensoren har også en annen funksjon som regner ut vinkelhastigheten til sensoren ved et gitt punkt. Det var denne funksjonen vi endte opp med å bruke til å regne oss tilbake til vinkelposisjonen. Dette kan ses i kodeutdraget nedenfor.

Kode 3.1: Utdrag fra `Prosjekt05_Pendel.m`

```
118         if k==1
119             tic
120             Tid(1) = 0;
121             Vinkel(1) = 0;
122             resetRotationAngle(myGyroSensor);
123             GyroRate(k) = ...
                    double(readRotationRate(myGyroSensor));
124         else
125             Tid(k) = toc;
```

```
126             GyroRate(k) = ...
                    double(readRotationRate(myGyroSensor));
127             Vinkel(k) = Vinkel(k-1) + ...
                    GyroRate(k)*(Tid(k)-Tid(k-1));
128             sinusfunc(k) = ...
                    A*exp(-zeta*Tid(k))*cos(omega*Tid(k));
```

Linje 127 viser hvordan vinkelen regnes ut basert på den forrige verdien pluss endringen av vinkelposisjonen. Gyrorate gir oss hvor endringen av vinkelen i $°/s$. For å finne endringen av vinkelen siden forrige måling multipliserer vi derfor denne endringen med tidsendringen siden forrige måling (Tid(k)-Tid(k-1)). Dette ga oss en mye mer nøyaktig måling av vinkelposisjonen. Verifisering av den matematiske modellen er gjort ved koding i Matlab. Mer bestemt har vi benyttet oss av de innebygde funksjonene for løsing av ODE (Ordinary Differential Equation) Matlab har å by på. For å lese mer om hvordan disse funksjonene fungerer, gå til nettsiden [6]. Koden for løsing av differensiallikningen kan ses i kodeutdraget under.

Kode 3.2: Utdrag fra `Prosjekt05_Pendel.m`

```
42 syms y(x)
43 Dy = diff(y);
44 ode = diff(y,x,2) + (0.0139/0.627)*Dy +10.88*y == 0;
45 cond1 = y(0) == -9.26;
46 cond2 = Dy(0) == 0;
47 conds = [cond1 cond2];
48 ySol(x) = dsolve(ode,conds);
49 ySol = simplify(ySol)
50 tsone = [6.764 20];
```

Her er *ode* differensiallikningen som vi kom fram til i 3.22. Videre er $cond1$ og $cond2$ initialverdiene for vinkelposisjonen θ og vinkelhastigheten $\frac{d\theta}{dt}$. $ySol$ er funksjonen vi ender opp med for θ. Fra figur **??** kommer det fram at pendelen blir sluppet, og starter sin bevegelse etter 6.764s. $Tsone$ (intervalltiden for signalet) er derfor satt til å gå fra 6.764 til 20. Dette må vi også ha i tankene når vi skal plotte signalet. Nedenfor kan du se hvordan dette er implementert i plottingen.

Kode 3.3: Utdrag av plottingen fra `Prosjekt05_Pendel.m`

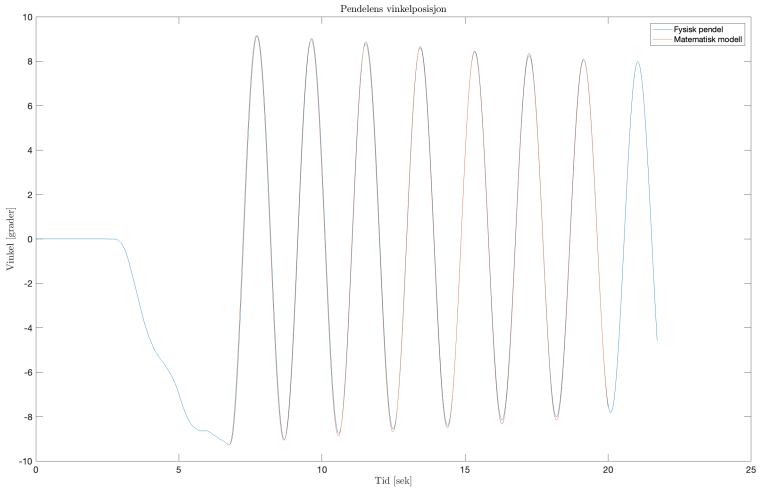
```
205
```

```
206       plot(Tid(1:k-1),Vinkel(1:k-1));
207       title('Vinkel [grader]')
208       xlabel('Tid [sek]')
209       hold on
210       fplot(ySol(x-6.764), tsone );
211       %plot(tiden2 + 6.8, Test1, 'r--');
212       legend('Fysisk pendel', 'Matematisk modell')
```

Siden pendelen blir sluppet ved $t = 6.764s$, må vi faseforskyve utrykket vårt for θ. Dette er gjort i linje 210 i kodeutdraget.

3.5 Resultat

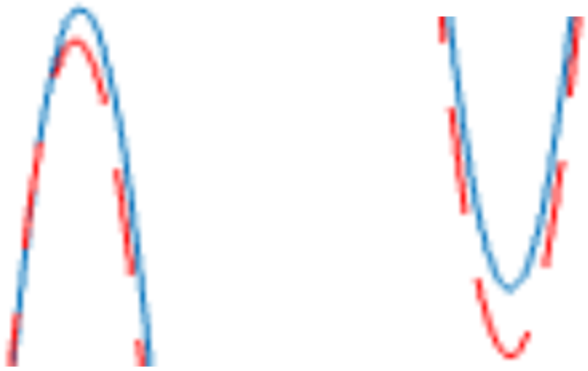
I figuren nedenfor er pendelens vinkelposisjon plottet sammen med den matematiske modellen for systemet.



Figur 3.7: Pendelens virkelige posisjon (blå kurve) plottet med den matematiske modellen for systemet (rød kurve)

Som vi ser er den matematiske modellen for systemet veldig lik den faktiske bevegelsen til pendelet. Hvis man studerer figuren nøye, kan man dog se

noen små endringer. Den matematiske modellen har lavere topp- og bunnpunkt enn grafen til den fysiske pendelen. Noe som ikke kom som noen overraskelse. Når vi regnet ut dempekoeffisienten så vi at systemet så ut til å ha en større demping mot en av sidene (som kom til tyde ved at differansen på etterfølgende bunnpunkt var større enn for etterfølgende toppunkt). Det gir da mening at gjennomsnittet av disse dempekoeffisientene vil gi et resultat som overdempet i forhold til den ene (toppunktene), og underdempet i forhold til den andre (bunnpunktene). En mulig årsak til dette kan være at systemet vi konstruerte hadde større friksjon mot den ene veien. Kanskje en mer trolig løsning var at likevektspunktet ble forskjøvet litt under hevelsen av pendelet. Uansett er dette snakk om så små forskjeller at vi ikke vil bruke mer tid på å finne ut av det. På figuren nedenfor er forskjellene mellom topp- og bunnpunkt ytterligere demonstrert.



Figur 3.8: Forskjellen mellom den matematiske modellen og pendelens virkelige posisjon ved et utvalgt toppunkt (venstre) og et utvalgt bunnpunkt (høyre)

Kapittel 4

Tegne bane

4.1 Problemstilling

I dette prosjektet har målet vært å kunne kartlegge banen LEGO-roboten tilbakelegger under kjøring. Ved at roboten lagrer hvordan den kjører kan vi presentere ett kart over hvor den har kjørt. For å få dette til er vi avhengig av å finne strekningen og retningen roboten har tatt. Basert på verdier fra sensorer som roboten er koblet til kan vi regne oss frem til både strekning og retning. Med verdier for strekning og retning kan vi matematisk sett plotte posisjonen i en graf. Vi har valgt å hente verdier gjennom bruk av både *gyrosensoren*, og *vinkelposisjonen* til motoren. Slik har vi satt opp problemstillingen i to ulike deler.

- Del en: Plotte banen til roboten basert på *gyro-sensoren*
- Del to: Plotte banen til roboten basert på *vinkelposisjonen* til motorene.

I resultat delen 4.3 vil vi presentere plottet med *gyrosensoren* og *vinkelposisjonen* sammen. Dette gjør vi for å avgjøre hvilken metode som er mest nøyaktig. Motivasjonen for å gjennomføre prosjektet var for at plotet skal kunne brukes videre i prosjektet *Robotstøvsuger* 8.

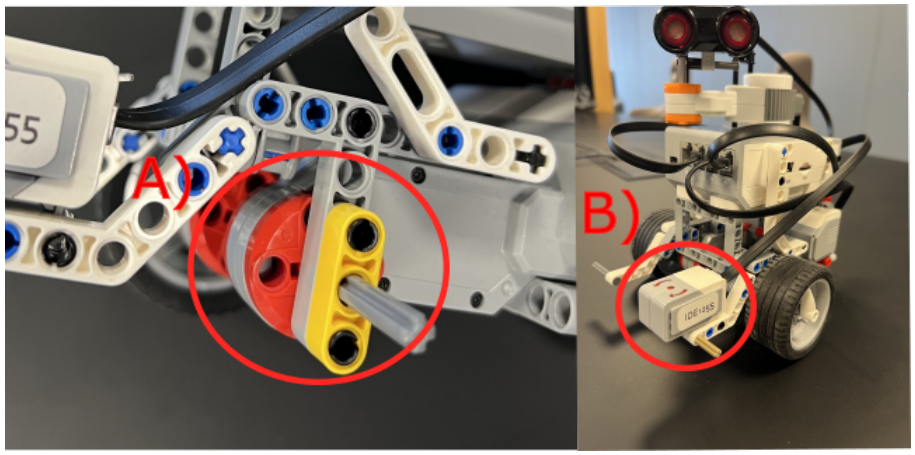
4.2 Forslag til løsning

I dette delkapittelet vil vi presentere vårt forslag til løsning på hvordan tegne banen til LEGO-roboten. Dette inkluderer:

- Kort om LEGO-konstruksjon, sensorer og målinger
- Matematikken og koden for å lage plottet basert på *gyro-sensoren*
- Matematikken og koden for å lage plottet basert på *vinkelposisjonen* til motoren

4.2.1 LEGO-konstruksjon, sensorer og målinger

For å kunne tegne banen LEGO-roboten tilbakelegger er vi avhengig av en robot som kan kjøre. Igjennom alle prosjektene har vi valgt å bruke samme konstruksjon av LEGO-roboten når den skal kunne kjøre. Konstruksjonen er forklart nærmere i vedlegget om LEGO-konstruksjonen **??**. Roboten er satt opp slik at vi har koblet til flere sensorer. I dette prosjektet skal vi i all hovedsak fokusere på *gyrosensoren* og sensoren som måler *vinkelposisjonen* til motorene. LEGO-konstruksjonen vises nedenfor, hvor vi har markert begge disse sensorene.



Figur 4.1: Bilder av LEGO-roboten. A) Bilde av motoren til roboten uten hjul. Her er den sensor som registrerer vinkelposisjonen til hjulet. B) Gyro-sensoren som registrerer endring i vinkel.

Ved å ha tilkoblet *gyrosensoren* vil vi kunne oppdage endringen i rotasjonsbevegelse. *gyrosensoren* vil gi verdier i form av grader. Det vil si om roboten kjører en full sirkel mot høyre vil det resultere i en endring i rotasjon på *360[grader]*. Rotasjoner mot venstre gir negative verdier. Merk at *gyrosensoren* klarer bare å gi oppgi grader i form av heltall.
Når det kommer til sensoren for å måle endring i *vinkelposisjonen* til motorene oppgis dette også i *grader*. Det vil si om endring i vinkelposisjon er *360[grader]*, vil hjulet gått en full runde fremover. Går hjulet bakover vil det gi negative verdier. Ved å kunne måle endring i vinkelposisjonen til motorene kan vi kalkulere strekningen roboten tilbakelegger.
Ved å ha målinger fra både endring i rotasjon og avstand er vi i stand til å beregne og plotte posisjonen til roboten. Vi vil begynne med den enklere måten hvor vi benytter oss av gyrosensoren.

4.2.2 Matematikken og koden for å lage plottet basert på gyro-sensoren

I denne seksjonen vil vi presentere matematikken og koden for å plotte banen til LEGO-roboten basert på målinger fra gyro-sensoren. Plotet vil bestå av en todimensjonal graf. Slik vil det være en X og Y variabel. Det vil også

være en ligning for både *X & Y*. Disse ligningene kan vi konstruere basert på målingene fra gyro-sensoren og endring i vinkelposisjonen til motorene. Først må vi definere en ligning for *strekningen* roboten tilbakelegger, som vist under.

$$Strekning = \frac{(\Delta VinkelPosMotorA + \Delta VinkelPosMotorB) \cdot 0.17}{720} \quad (4.1)$$

Her er ligningen for strekning definert ved å ta den kombinerte endringen i vinkelposisjon fra begge motorene og gange den med hjulets omkrets(17 cm), dette er så delt på 720 grader($2 \cdot 360$), fordi det er to motorer.
Når vi skal implementere ligningen for strekning til kode må vi først definere endringen i vinkelposisjon (Δ VinkelPosMotor) mellom vært tidsskritt. Det gjør vi som vist i kodeutdraget under.

Kode 4.1: Kode for endring av vinkelposisjon til motorene

```
1 Delta_VinkelPosMotorA = VinkelPosMotorA(k) - VinkelPosMotorA(k-1);
2 Delta_VinkelPosMotorB = VinkelPosMotorB(k) - VinkelPosMotorB(k-1);
```

Her ser vi at endringen i vinkelposisjon til motorene er definert som **Delta_VinkelPosMotorA** og **Delta_VinkelPosMotorB**. Dette er endringen i vinkelposisjon siden vi tar den nåværende vinkelposisjonen minus forrige vinkelposisjonen. Ved å ha en definert variabel for endring av vinkelposisjon for hver av motorene kan vi videre implementere ligningen for strekning 4.2.2 til koden.

Kode 4.2: Kode for beregning av strekning

```
1 Strekning(k) = ((Delta_VinkelPosMotorA + ...
      Delta_VinkelPosMotorB)*0.17)/(720);
```

I praksis bruker vi samme ligning som vi definerte i tidligere for strekning 4.2.2. Den eneste forskjellen er at "strekning" er en variabel som blir definert for hvert tidskritt (k). Etter å ha definert et mål for *strekning* vender vi til matematikken. Ved å bruke geometri kan vi avgjøre posisjonen i både X og Y plan.

$$X = Strekning * sin * Vinkel \quad (4.2)$$

$$Y = Strekning * cos * Vinkel \quad (4.3)$$

Her er *strekning* den samme som vi definerte i ligning 4.2.2, Og *vinkel* er verdien vi får fra gyro-sensoren. Med geometri vil vi få et punkt som vi kan sette for posisjonen.
For at vi skal kunne anvende dette i koden er vi nødt til å først definere en variabel for gyro-sensoren. Vi kaller *vinkel* i dette tilfelle **GyroAngle(k)** i koden. Koden for å initialisere gyro-sensoren vises under.

```
1 GyroAngle(k) = double(readRotationAngle(myGyroSensor));
```

GyroAngle(k) vil få ny verdi for hver iterasjon når sensoren avleser vinkelen [grader]. For at vi skulle implementere ligningene for X og Y i koden bruker vi vinkelen fra **GyroAngle(k)**, og **Strekning(k)** som vi definerte i kodeutdrag strekning 4.2. Vi kan dermed definere variablene for X og Y i koden.

Når vi bruker denne ligningen i koden, må vi spesifisere "vinkelen" til radianer som vist i koden nedenfor.

```
1 x(k) = x(k-1) + (strekning(k)*sin(pi*GyroAngle(k)/180));
2 y(k) = y(k-1) + (strekning(k)*cos(pi*GyroAngle(k)/180));
```

Siden LEGO-roboten leser inn nye verdier for hver iterasjon legger vi til verdiene fra forrige tidskritt X(k-1) og Y(k-1).
Med definerte variabler for **x(k)** og **y(k)** kan vi plotte posisjonene, som vist i kodeutdraget under.

Kode 4.3: Kode for plotting av posisjon

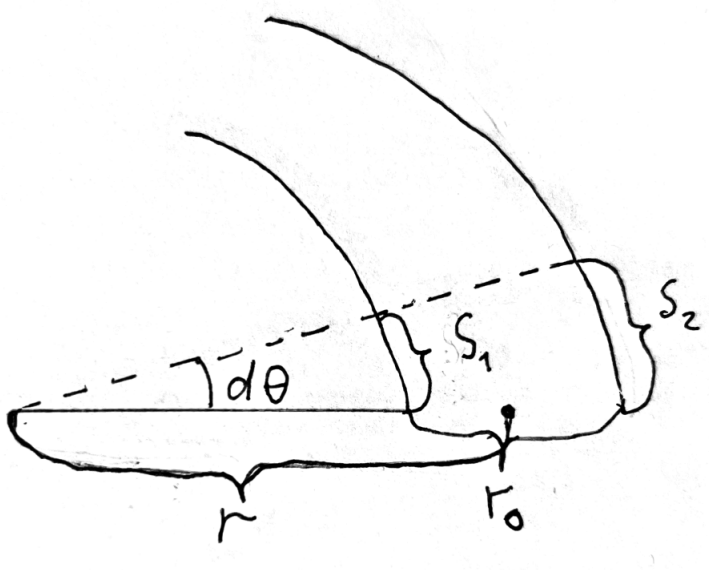
```
1 plot(x(1:k-1),y(1:k-1));
2 title('Posisjon')
3 xlabel('avstand[m]')
```

Vi vil presentere resultatet av denne metoden med gyrosensoren for å tegne banen i kapitelet resultater 4.3.

4.2.3 Matematikken for å lage plottet basert på vinkelposisjonen til motorene

I denne seksjonen vil vi presentere hvordan vi kan kun bruke vinkelposisjonen til motorene for å kunne tegne banen til roboten. Dette vil si at vi ikke er avhengig av å bruke gyrosensoren for å oppnå et mål på endring i rotasjonbevegelse. Måten vi oppnår et mål på endring i rotasjon er gjennom å matematisk regne det frem basert på vinkelposisjonen til motorene. Slik bruker vi fortsatt de definerte uttrykket for *strekning*, X og Y. Den eneste forskjellen er at vi finner *vinkelen* gjennom vinkelposisjonen til motorene.

For å bruke vinkelposisjonen til hjulene til å bestemme vinkelen til roboten, var vi nødt til å komme fram til en matematisk formel. For å komme fram til dette så vi på ett veldig lite (uendelig lite) intervall hvor roboten svinger. Vi vil da få en situasjon som på figuren nedenfor.



Figur 4.2: En skisse over at roboten svinger, hvor r_0 er avstanden mellom hjulene. Svingen danner en sirkelformet bevegelse, hvor$r$ er radiusen fra midten av bilen til midten av sirkelen. Hvor $d\theta$ er endringen i vinkelen. S_1 og S_2 vil være strekningen for hver av hjulene tilbakelegger.

Hvis vi ser på et lite nok intervall, vil S_1 og S_2 være tilnærmet rette strek-

ninger. Vi kan da bruke endring i vinkelposisjonen til hjulene for å regne ut disse avstandene. Vi får da følgende ligning for S_1 og S_2.

$$S_{1/2} = \frac{\Delta Vinkelpos Motor \cdot 0.17}{360} \tag{4.4}$$

Vi vet at en seksjon av sirkelens omkrets er gitt ved den bestemte vinkelen, ganger med sirkelens radius. Dette har vi brukt til å komme fram til følgende formel for strekningene S_1 og S_2

$$S_1 = d\theta \cdot (r - \frac{r_0}{2}) \tag{4.5}$$

$$S_2 = d\theta \cdot (r + \frac{r_0}{2}) \tag{4.6}$$

Bytter vi ut S_1 og S_2 i likningene ovenfor med høyresiden av ligning 4.4, får vi følgende likningssett.

$$\frac{\Delta Vinkelpos Motor_A \cdot 0.17}{360} = d\theta \cdot (r - \frac{r_0}{2}) \tag{4.7}$$

$$\frac{\Delta Vinkelpos Motor_B \cdot 0.17}{360} = d\theta \cdot (r + \frac{r_0}{2}) \tag{4.8}$$

Løser vi likningssettet med hensyn på $d\theta$, finner vi et utrykk for endringen av vinkelen

$$d\theta = \frac{(\Delta Vinkelpos MotorA - \Delta Vinkelpos MotorB) \cdot 0.17}{360 \cdot r_0}(rad) \tag{4.9}$$

For å få endringen i grader må vi gange uttrykket ovenfor med $\frac{180}{\pi}$. Dette resulterer i at vi ender opp med likningen vår for endring av vinkelen, målt i grader.

$$d\theta = \frac{(\Delta Vinkelpos MotorA - \Delta Vinkelpos MotorB) \cdot 0.17}{2 \cdot \pi \cdot r_0}(grader) \tag{4.10}$$

Slik har vi et mål på endring av grader vi kan bruke i koden vår. Dette vil gjøres for hver iterasjon av koden, slik at intervallene mellom målingene blir så små som mulig.

4.2.4 Koden for å lage plottet basert på vinkelposisjonen til motorene

Ved å nå ha definert ett mål på endring i grader basert på vinkelposisjonen til motorene i ligning 4.10 kan vi implementere det videre til kode. Først trenger vi å finne avstanden mellom hjulene (r_0). Når vi målte opp avstanden, fant vi at det var ca 12,2 cm mellom midten av det ene hjulet til midten av det andre hjulet.

Ved å målt opp avstanden mellom hjulene til å bli 12,2 cm kan vi sette det inn i for r_0 i ligning 4.10. Slik kan vi nå definere ligningen for endring av vinkel.

$$d\theta = \frac{(\Delta VinkelposMotorA - \Delta VinkelposMotorB) \cdot 0.17}{2 \cdot \pi \cdot 0.122}(grader) \quad (4.11)$$

Vi kan direkte bruke denne ligningen når vi gjør det om til kode. Vi velger å kalle $d\theta$ for `delta_vinkel` i koden. Koden for endring av vinkel vises under.

Kode 4.4: Kode for endring av vinkel basert på vinkelposisjonen til motorene

```
1 Delta_vinkel = ((Delta_VinkelPosMotorA - ...
      Delta_VinkelPosMotorB)*0.17)/(2*pi*0.125);
```

Videre blir **vinkel(k)** i koden definert ved å ta forrige verdi av vinkelen(k-1) summert med **delta_vinkel**, som vist i kodeutdraget under.

```
1 vinkel(k) = vinkel(k-1) + Delta_vinkel;
```

Når vi regner frem vinkelen basert på vinkelposisjon til motoren må vi avgjøre hvilken retning den tar. Sett langs kjøreretningen til roboten har vi bestemt at om den svinger til høyre vil den være positiv, og negativ mot venstre. Slik vil motor på høyre side være A, og venstre være B. Vi finner om vinkelen er positiv eller negativ ved å sjekke om **Delta_VinkelposMotorA**er større enn **Delta_VinkelposMotorB**. Dette gjøres med en "IF-statement".

Kode 4.5: Kode for å sjekke om det positiv eller negativ vinkel

```
1 if Delta_VinkelPosMotorA > Delta_VinkelPosMotorB
2     Delta_vinkel = ((Delta_VinkelPosMotorA - ...
          Delta_VinkelPosMotorB)*0.17)/(2*pi*0.125);
3     vinkel(k) = vinkel(k-1) + Delta_vinkel;
```

Denne delen av "IF-statementen" avgjør at vinkelen blir positiv dersom **Delta_VinkelPosMotorA** er større enn **Delta_VinkelPosMotorB**, altså svinger mot høyre. I "else" delen av "IF-statementen" setter vi at vinkelen blir negativ dersom **Delta_VinkelPosMotorA** er mindre enn **Delta_VinkelPosMotorB**.

```
1 Delta_vinkel = -((Delta_VinkelPosMotorB - ...
      Delta_VinkelPosMotorA)*0.17)/(2*pi*0.125);
2 vinkel(k) = vinkel(k-1) + Delta_vinkel;
```

Koden over vil være et tilfelle hvor roboten svinger mot venstre, hvor endringen i vinkelen blir negativ.
Vi kan nå definere X og Y variablene i koden på samme måte som vi gjorde med gyrosensoren, ved at vi istedenfor bruker **vinkel(k)** fra kodeutdraget over. Da blir koden for X og Y slik:

```
1 x(k) = x(k-1) + (strekning(k)*sin(pi*vinkel(k)/180));
2 y(k) = y(k-1) + (strekning(k)*cos(pi*vinkel(k)/180));
```

Plottingen vil fungere på samme måte som tidligere, som vist i kodeutdrag 4.3

4.3 Resultater

I dette kapitelet blir resultatene fra posisjons plotet presentert. Vi vil presentere resultater ved bruk av både gyro sensor og vinkelposisjonen til motoren. Ved å la roboten kjøre i ulike baner vil vi verifisere at plottet stemmer med den faktiske banen roboten tilbakelegger. Dette gjør vi ved å sammenligne koordinatene fra plottet med egne utregninger. For å sikre at roboten holder riktig bane lagde vi en PID-regulator på begge motorene, som forklart i kapitelet PID-regulator 7. Først vil vi presentere resultatene ved å få roboten

til å kjøre rett frem. Deretter vil vi få roboten til å ta en sving, og til slutt vil vi få roboten til å kjøre i en slags "S" bane.

4.3.1 Kort om PID-regulatoren

For å få roboten til å kjøre rett fremover valgte vi at begge hjulene skulle holde en hastighet på 88[grader/s]. Dette gjør vi ved å sette referansehastigheten i koden til 88[grader/s].

```
1 HastighetRefA = 88;
2 HastighetRefB = 88;
```

Ved at vi har satt en PID-regulator på hver av motorene vil regulatoren gi motorpådrag slik at hjulene holder en hastighet på 88[grader/s]. Grunnen til at vi setter en referansehastigheten til 88[grader/s] er fordi vi regnet frem passende regulator parametre som fungerer best ved denne hastigheten. Dette er forklart nærmere i kapitelet om regulering 7.

4.3.2 Plottet ved å la roboten kjøre rett fremover

Vi ønsker å validere om plottet vårt er nøyaktig når roboten kjører rett fremover. For å validere plottet velger vi å lese av endelig `Y` verdi, som representerer lengden roboten har kjørt. Vi sammenligner `Y` verdien mot egne beregninger for å bekrefte hvor nøyaktig plottet er.

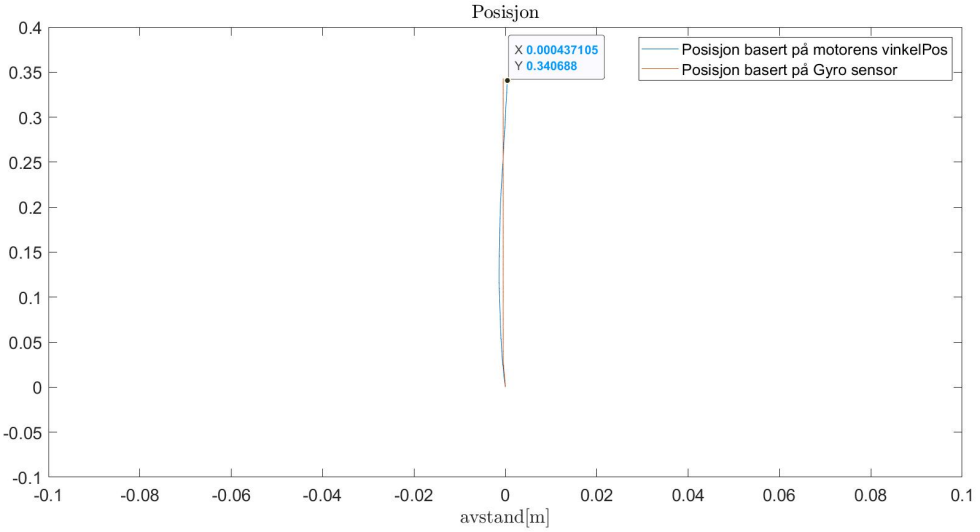
Kode for å kjøre roboten rett fremover

Vi ønsker å få roboten til å kjøre rett fremover inntil hjulene har gått to ganger rundt. For å få roboten til å kjøre rett fremover bruker vi koden under.

Kode 4.6: Kode for å at roboten kjører fremover frem til hjulene har gått to ganger rundt

```
1  if VinkelPosMotorB(k) < 720
2      PowerB(k) =  PB(k) + IB(k) + DB(k-1);
3  else
4      PowerB(k) = 0;
5  end
6
7  if VinkelPosMotorA(k) < 720
8      PowerA(k) =  PA(k) + IA(k) + DA(k-1);
9  else
10     PowerA(k) = 0;
11 end
```

Kort fortalt gis det ett motorpådrag fra regulatoren så lenge vinkelposisjonen til motorene er under 720 grader. Dette gjør at roboten kjører rett fremover til hjulet har gått to runder rundt.
Koden for å plote er vist i kodeutdrag til plotet 4.3. Resultatet fra å plote kjøringen rett fremover vises i figuren under.



Figur 4.3: Posisjonen LEGO-roboten tilbakelegger etter at hjulene har gått to runder. Her vises det to linjer, hvor den ene er basert på gyrosensoren, og den andre på vinkelposisjonen til motorene.

Fra figuren ser vi at avlest `Y` verdi er 0.34, altså 34cm. Da forteller plottet at roboten har kjørt 34cm rett fremover. Vi vil nå gjøre egne beregninger

for å validere dette.

Validere plottet av kjøring rett fremover

Siden omkretsen til hjulene er 17cm, vet vi at roboten har kjørt 17cm etter en runde. Da blir lengden etter to runder vist i ligningen under.

$$Lengde = 17cm \cdot 2 = 34cm \tag{4.12}$$

Vi konkluderer dermed at plotet klarer nøyaktig å kartlegge lengden til roboten. Videre ønsker vi å validere om plotet klarer å nøyaktig kartlegge posisjonen når roboten svinger.

4.3.3 Plottet når roboten svinger mot høyre

Vi ønsker å validere om plotet er nøyaktig når roboten svinger mot høyre. Ved å lese av både `Y` og `X` verdiene fra plotet vil vi sammenligne de mot egne utregninger av koordinatene.

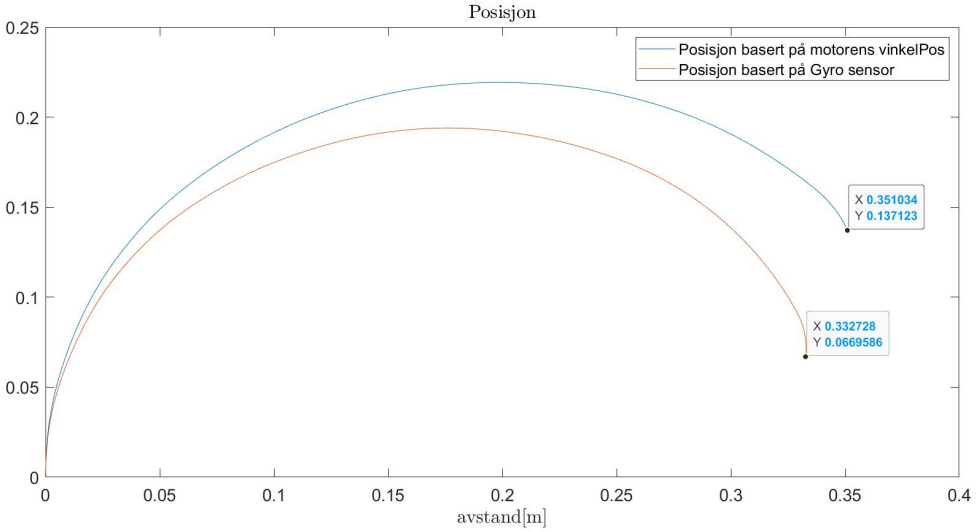
Kode for at roboten tar en sving mot høyre

Vi ønsker at roboten skal kjøre slik at den svinger mot høyre. Det gjør vi ved å kjøre koden under.

Kode 4.7: Kode for at robot svinger mot høyre

```
1  HastighetRefA = 44;
2  HastighetRefB = 88;
3  if VinkelPosMotorB(k) < 1440
4      PowerB(k) =  PB(k) + IB(k) + DB(k-1);
5  else
6      PowerB(k) = 0;
7  end
8
9  if VinkelPosMotorA(k) < 720
10     PowerA(k) =  PA(k) + IA(k) + DA(k-1);
11 else
12     PowerA(k) = 0;
13 end
```

Sett langs kjøreretningen til roboten vil `MotorA` være på høyreside, og `MotorB` være på venstresiden. I koden setter vi at `MotorA` skal nå en vinkelposisjonen på 720, og `MotorB` skal nå en vinkelposisjon på 1440. `MotorB` settes til å ha en `referansehastighet` på 88[grader/s], som er dobbel så raskt som `MotorA` sin `referansehastighet`. Når denne koden kjøres vil roboten svinge mot høyre. Resultatet fra å plotte kjøringen med sving mot høyre vises i figuren under.



Figur 4.4: Posisjonen LEGO-roboten tilbakelegger etter å ha kjørt kode for å svinge mot høyre 4.7. Her representerer blå linje posisjonen basert på motorenes vinkelposisjon, og oransje linje posisjonen basert på gyrosensor.

Fra figuren ser vi avleste verdier fra de endelige koordinater:

- Posisjonen basert på motorenes vinkelposisjon gir oss:

$$\begin{aligned} X &= 0.351 \\ Y &= 0.137 \end{aligned} \tag{4.13}$$

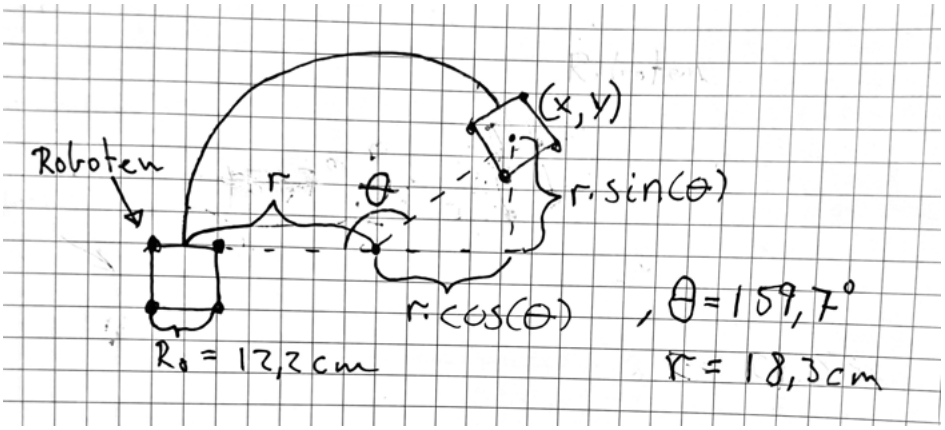
- Posisjonen basert på gyrosensor gir oss:

$$\begin{aligned} X &= 0.333 \\ Y &= 0.067 \end{aligned} \tag{4.14}$$

Vi vil bruke disse koordinatene når vi verifisere opp mot egneutregninger av koordinatene.

Validere plottet av sving mot høyre

Måten vi verifiserer plottet er ved å regne oss frem til X og Y koordinatene og sammenligne de mot de avleste koordinatene fra figur 4.4. For å regne frem slutt-koordinatene til roboten når den svinger mot høyre brukte vi følgende figur.



Figur 4.5: Skisse over at roboten svinger mot høyre.

Måten vi har regnet oss fram til x- og y-koordinatene, er ved å bruke ligning 4.10. Siden endringen av vinkelen er lik for hele tidsperioden, kan vi bruke denne til å regne oss fram til den totale endringen av vinkelen. Setter vi inn 720 for $VinkelposMotorA$, og 1440 for $VinkelposmotorB$, får vi at endringen av vinkelen blir lik *159.7* grader. Alt vi trenger da er radiusen til den sirkelformede bevegelsen. Snur vi om på ligning 4.7, får vi følgende uttrykk for radiusen.

$$r = \frac{\Delta VinkelposMotorB \cdot 0.17}{360 d\theta} - \frac{r_0}{2} \tag{4.15}$$

Det vi må være obs på ved bruk av denne ligningen er at $d\theta$ i denne likningen er endringen i vinkelen, målt i radianer. 159.7° i grader tilsvarer en verdi

på *2.79* i radianer. Satt inn i ligningen ovenfor gir dette en radius på *18.3* cm.

Utifra illustrasjonen 4.5 kan vi regne oss fram til de endelige x- og y-koordinatene. Y-verdien til det nye punktet får vi ved å ta sinus til vinkelen (159.7) ganger med radiusen (0.183m). For x-verdien må vi ta cosinus til 180 minus vinkelen ganger med radiusen, pluss radiusen. De endelige koordinatene blir da.

$$X = \cos(180^\circ - 159.7^\circ) \cdot 0.183m + 0.183m = 0.35463m$$
$$Y = \sin(159.7^\circ) \cdot 0.183m = 0.06348m \tag{4.16}$$

Ved at vi har regnet oss frem til X og Y koordinatene vil vi sammenligne de mot de avleste koordinatene fra figur 4.4. Alle koordinatene vises i tabellen under.

Metode	X	Y
Basert på motorenes vinkelposisjon	0.351m	0.137m
Basert på gyrosensor	0.333m	0.067m
Egneutregning	0.355m	0.063m

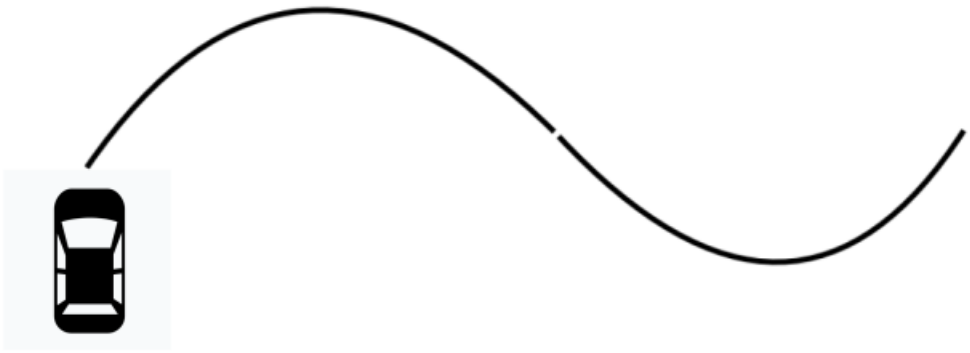
Fra tabellen ser vi at plottet basert på gyrosensoren er mest nøyaktig, med en usikkerhet på ett par centimeter. Y verdien basert på motorenes vinkelposisjon bommer med 7,4cm. Dette kan skyldes av at hjulene "spinner" når roboten kjører. Videre ønsker vi å bekrefte at metoden med gyrosensoren er mer nøyaktig enn plottet basert på motorenes vinkelposisjon. Dette gjør vi ved å få roboten til å kjøre i en "S" bane. Dette gjør vi også for å se om plottet klarer sving mot både høyre og venstre.

4.3.4 Plottet når roboten kjører i en S bane

Vi ønsker å validere om plotet er nøyaktig når roboten svinger mot både høyre og venstre. Ved å lese av både `Y` og `X` verdiene fra plotet vil vi sammenligne de mot egne utregninger av koordinatene.

Kode for å kjøre roboten i en S bane

Vi ønsker å få roboten til å kjøre i en slags S bane som vist i bilde under.



Figur 4.6: Kjøre banen vi ønsker at roboten skal kjøre for å validere at plottet er nøyaktig ved sving mot både høyre og venstre.

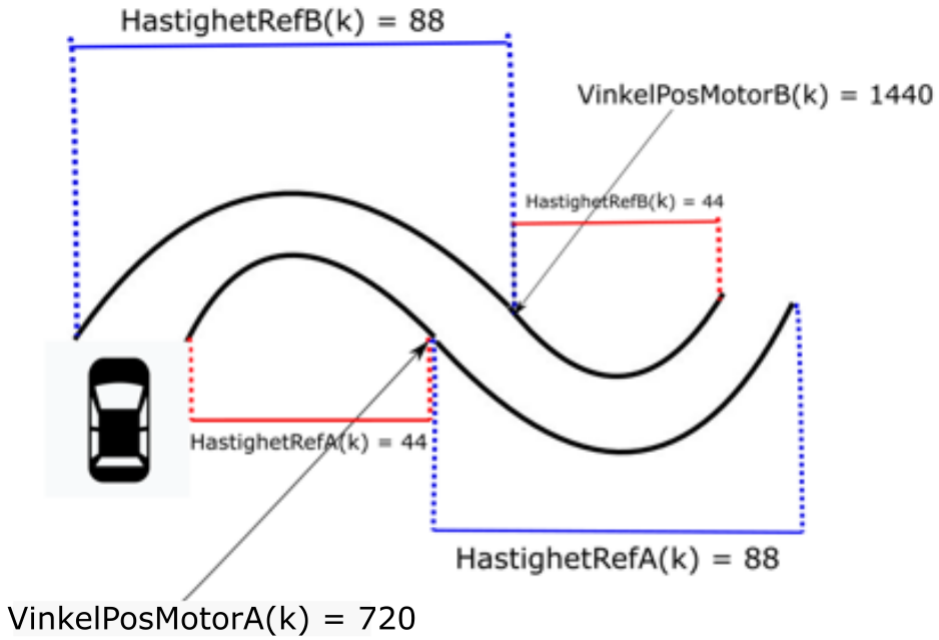
Måten vi får roboten til å kjøre som på bildet over er ved å kjøre koden under.

Kode 4.8: Kode for at robot kjører i en S-bane

```
1  if VinkelPosMotorA(k) > 720 && VinkelPosMotorB(k) > 1440
2      HastighetRefA(k) = 88;
3      HastighetRefB(k) = 44;
4  else
5      HastighetRefA(k) = 44;
6      HastighetRefB(k) = 88;
7  end
8
9  if VinkelPosMotorB(k) < 2160 && k>1
10     PowerB(k) =  PB(k) + IB(k) + DB(k-1);
11 else
12     PowerB(k) = 0;
13 end
14
15 if VinkelPosMotorA(k) < 2160 && k>1
```

```
16        PowerA(k) =  PA(k) + IA(k) + DA(k-1);
17    else
18        PowerA(k) = 0;
19    end
```

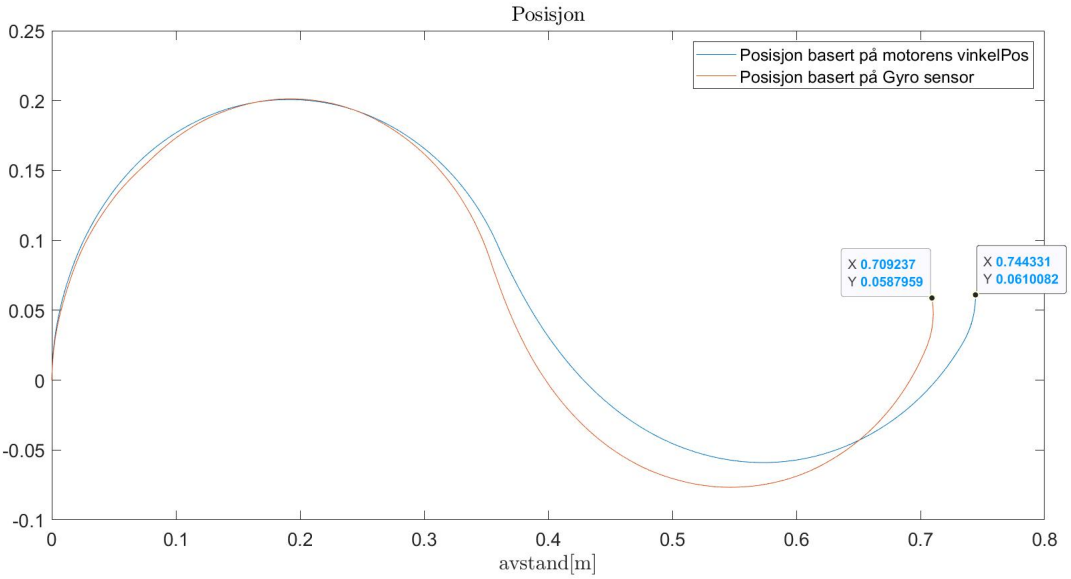
Denne koden er baseres seg på koden for å svinge mot høyre 4.7. Dette fordi roboten vil først ta en slik sving mot høyre, for deretter å ta en slik sving mot venstre. Svingen skjer ved at referansehastigheten(`HastighetRef(k)`) endrer seg når motorene når vinkelposisjonen (`VinkelPosMotor(k)`) 720 og 1440. Bildet under illustrerer når endringen i referansehastighet(`HastighetRef(k)`) skjer på MotorA og MotorB.



Figur 4.7: Illustrasjon av roboten som kjører i en S-bane. Målet er å illustrere endringen i referansehastighet når motorene når ulike vinkelposisjoner.

Fra figuren ser vi at MotorA starter første sving med en referansehastighet på 44[grader/s], og MotorB starter med en referansehastighet på 88[gra-

der/s]. Siden MotorB har en dobbelt så rask hastighet vil den nå vinkelposisjon 1440, like fort som MotorA når en vinkel på 720. Fra dette punktet bytter Motorene referansehastighet, og når vinkelposisjon 2160 samtidig. Dette utgjør en S-bane. Resultatet fra å plotte kjøringen av denne S-banen vises i figuren under.



Figur 4.8: Posisjonen LEGO-roboten tilbakelegger etter å ha kjørt koden for S-bane 4.8. Her vises det to funksjoner, hvor oransje er posisjonen basert på gyrosensoren, og den blå er posisjonen basert på vinkelposisjonen til motorene.

Fra figuren ser vi avleste verdier fra de endelige koordinater:

- Posisjonen basert på motorenes vinkelposisjon gir oss:

$$\begin{aligned} X &= 0.744 \\ Y &= 0.061 \end{aligned} \tag{4.17}$$

- Posisjonen basert på gyrosensor gir oss:

$$\begin{aligned} X &= 0.709 \\ Y &= 0.059 \end{aligned} \tag{4.18}$$

Vi vil bruke disse avleste koordinatene når vi verifisere opp mot egneutregninger av koordinatene.

Validere plottet ved kjøring i en S-bane

Vi vil igjen verifiserer plottet ved å regne oss frem til X og Y koordinatene og sammenligne de mot de avleste koordinatene fra figur 4.8. Vi kommer frem til de endelige koordinatene ved å bruke logikk.

- X-verdien. Når det kommer til X-verdien vil den øke like mye for hver sving. Ved første sving er X-verdien 0.355. Siden det er to svinger vil endelige X-verdien bli:

$$X = 2 \cdot 0.355 = 710 \tag{4.19}$$

- Y-verdien. Ved at roboten først tar en sving mot høyre vil Y-verdien gå opp, men Y-verdien vil gå like mye ned igjen når den tar samme sving motsatt vei. Slik vil det endelige Y-verdien bli 0.

$$Y = 0 \tag{4.20}$$

Ved at vi har funnet frem til X og Y koordinatene vil vi sammenligne de mot de avleste koordinatene fra figur 4.8. Alle koordinatene vises i tabellen under.

Metode	X	Y
Basert på motorenes vinkelposisjon	0.744m	0.061m
Basert på gyrosensor	0.709m	0.059m
Egneutregning	0.710m	0m

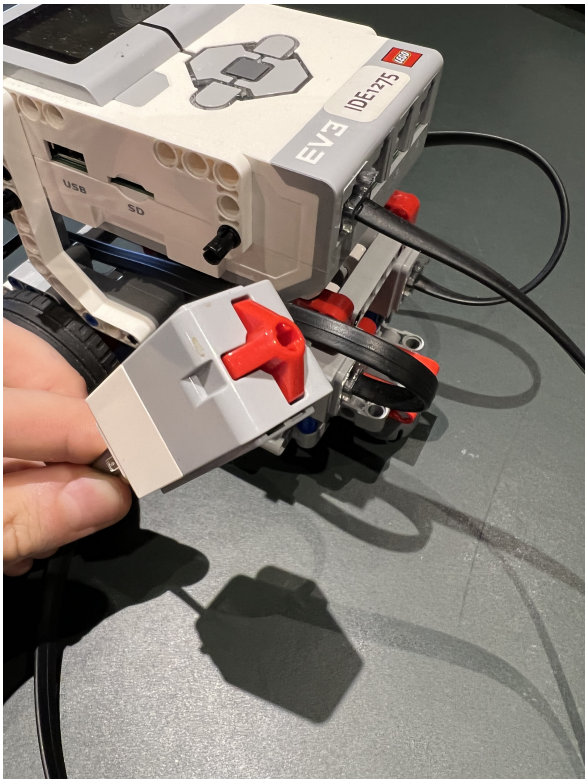
Fra tabellen ser vi igjen at koordinatene basert på gyrosensoren er mer nøyaktig enn koordinatene basert på motorenes vinkelposisjon. Men gyrosensoren har fremdeles usikkerhet når den plotter kjøreveien til roboten. Dette kan skyldes at gyrosensoren ikke er god nok, hvor den kun har oppløsning på én grad. Metoden basert på gyrosensoren ga igjennom kjøringene på det meste en feil på 6cm, hvor metoden basert på motorenes vinkelposisjon ga på det meste en feil på 7,4cm. Totalt sett har gyrosensoren gjort det best. Derfor konkluderer vi med å bruke gyrosensoren når vi skal plotte kjørebanen til robotstøvsugeren 8.

Kapittel 5

Sykkel Computer

5.1 Problemstilling

I dette kapitellet har målet vært å simulere en sykkelcomputer. I en standard sensorbasert sykkelcomputer vil du se at den bruker en magnet montert på en spile, og en magnetbryter på gaffelen som registrerer at hjulet går rundt. I dette prosjektet blir ikke slike sensorer brukt. Vi vil derimot simulere denne situasjonen ved bruk av en trykkbryter og MATLAB. Vi ønsker å måle gjennomsnittshastigheten, tilbakelagt strekning og akselrasjonen til sykkelen. Trykkbryteren vises på bildet under.



Figur 5.1: Bildet av trykkbryter koplet til EV3'en

Trykkbryteren vil gi et datasett bestående av 0'ere og 1'ere. Trykkbryteren gir 0 om den ikke er trykt inn, og 1 om den er trykt inn. Vi vil simulere at hjulet går rundt når trykkbryteren blir trykt inn og registrerer 1. Noen ganger registreres flere 1'ere etter hverandre, noe som betyr at EV3'en rekker å spør mange ganger mens du trykker. Det vil derfor være nødvendig å komme med en løsning hvor vi skiller unike trykk fra trykk hvor sensoren rakk å måle flere ganger mens bryteren fremdeles var trykt ned. En videre utfording å registrere når sykkel har stoppet og står i ro. Grunnen til dette er at sykkel computeren er bygd opp ved at den registrerer runder basert på at magneten eller trykkbryteren settes til 1 ved en passert runde. Slik vil den ikke få registert runden hvor sykkelen står i ro.

5.2 Forslag til løsning

I dette kapitellet skal vi presentere vårt forslag til sykkelcomputeren. Dette inkluderer kode og beregninger av:

- Validering av registerte trykk
- Momentanhastighet og IIR-filtrert hastighet
- Gjennomsnittshastighet siden start
- Tilbakelagt strekning
- Akselerasjon (ut ifra IIR-filtrert hastighet)

Til slutt vil presentere vår kode for å registrere når sykkelen står i ro.

5.2.1 Trykkbryter, og validere trykk

Vi vil nå presentere koden og variabler for å validere registrerte trykk. Ved starten av programmet vil koden se slik ut:

Kode 5.1: Kode ved start av sykkelcomputeren [8]

```
1 Tid(1) = 0;
2 Bryter(1)  = double(readTouch(myTouchSensor));
3 Runder = 1;
```

Her henter vi første avlesing fra trykkbryteren(`myTouchSensor`). Variabelen ble kalt `Bryter(k)`. Ved start blir også første runden til hjulet startet. Slik ble variabelen `Runder` definert, `Runder` vil være variabelen for å lagre antall unike trykk fra sensoren. Variabelen Tid(k) vil være tiden, og er 0 ved start.

For at variabelen `Runder` skal kunne registrere riktig antall runder hjulet har gått rundt, må vi validere trykkene fra trykkbryteren. Dette må vi gjøre siden det kan bli registrert flere 1'ere etter hverandre når trykkbryteren er trykt ned. Vi validerer trykkene for å forhindre at mange runder blir

registrert, når hjulet i realiteten bare har gått en runde. Valideringen består av å skille ut uønskede 1'ere, slik at det blir registrert kun ett unikt trykk. Kriteriene for ett unikt trykk er som følgende:

- Trykkbryteren må registrere verdi på 1.
- Den forrige målingen av trykkbryteren kan ikke være lik 1, da vil dette være det samme trykket som allerede er målt.

Logikken til disse to uttrykkene implementerer vi til kode. Pseudokoden vises i kodeutdrag 5.2.

Kode 5.2: Utrykk

```
1 HVIS (Bryter(k) er lik 1) OG (Bryter(k-1) er lik 0)
```

Her sjekkes det først om bryteren er aktivert med at `Bryter(k)` er lik 1. Deretter sjekkes det om at den forrige målingen ikke er presset ned, altså lik 0. Vi sjekker forrige måling med at `Bryter(k-1)` er lik 0. For at trykket skal bli registrert som er validert trykk må begge kravene oppnås.
Ett validert trykk vil si at hjulet har gått en runde rundt. Ut ifra dette kan økes variabelen `Runder`, dermed ville summen av antall trykk fortelle oss hvor mange runder hjulet har gått. Koden for å validere trykk og registrere ny `Runder` vises i kodeutdraget under.

Kode 5.3: Validere trykk og registrere runde.[8]

```
1 if (Bryter(k) == 1) && (Bryter(k-1) == 0)
2     Runder = Runder + Bryter(k);
3 end
```

Her blir det brukt en IF-statement for å validere trykket, er trykket validert legges det til en `Runder`. Ved å ha en validering på når hjulet har gått en runde, vil vi utregne hastigheten ved bruk av omkretsen til hjulet og rundetiden.

5.2.2 Utregninger

Vi vil nå presentere vår løsning på hvordan sykkel computeren vil kunne vise:

- Momentanhastighet og IIR-hastighet
- Gjennomsnittshastighet siden start
- Tilbakelagt strekning
- Akselerasjon

Formellen for momentanhastighet finner vi fra fysikken og er vist i ligning 5.2.2.

$$Momentanhastighet = \frac{\Delta Strekning}{\Delta Tid} \tag{5.1}$$

I denne formellen vil Δ Strekning være omkretsen av hjulet. Siden strekningen alltid vil være fast, trenger vi bare tiden hjulet bruker på en runde for å finne ut av momentanhastigheten.

For at vi skulle kunne måle tiden mellom hvert trykk, valgte vi å lagre tidspunktet registrert ved hvert nye trykk. Dette gjorde vi ved at vi lagde en ny variabel `Tid_R(Runder)`, og satt den til å være lik tiden i det bestemte tidspunktet `Tid(k)`. `Tid_R` blir satt til tidspunktet `Runder` blir inkrementert med en. `Rundetiden` finner vi ved hjelp av `Tid_R`. `Rundetiden` forteller oss hvor lang hjulet brukte på runden. Dette fant vi ved å ta den nyeste målingen på `Tid_R` og trekke i fra den forrige målte verdien. Utregningen av `RundeTid(Runde)` vises i kodeutdraget under.

Kode 5.4: If-Statement med utregning av rundetid [8]

```
1 if (Bryter(k) == 1) && (Bryter(k-1) == 0)
2     Runder = Runder + Bryter(k);
3     Tid_R(Runder) = Tid(k);
4     RundeTid(Runder-1) = Tid_R(Runder) - Tid_R(Runder-1);
5 end
```

Med `Rundetid` definert, regner vi deretter ut hastigheten på hjulet. Distansen per runde vil være omkretsen til hjulet. Vi har satt distansen til å være

omkretsen av sirkelen, dette vises i 5.2.2. Vi valgte å bruke en radius på 0.35 meter, som tilsvarer radiusen til et standard sykkelhjul.

$$Sirkelomkrets = 2\pi r \tag{5.2}$$

Ut ifra å ha omkretsen til hjulet bruker vi distansen til å kalkulere hastigheten til hjulet. Dette gjør vi gjennom å sette en variabel som heter `Hjul`, som er omkretsen til hjulet. Videre deler vi omkretsen på `RundeTid`. For at sykkelcomputeren vår skal kunne vise kilometer i timen multipliserer vi hastigheten med 3.6, som er forholdet mellom m/s og km/h. Koden til regne frem hastigheten vises i kodeutdrag 5.5.

Kode 5.5: Beregnet hastighet [8]

```
1 Hastighet(Runder-1) = (Hjul / RundeTid(Runder-1)) * 3.6;
```

Her blir Hastigheten regnet ut for forrige runde. Videre blir hastigheten regnet ut for hver gang ett validert trykk blir registrert. Siden hastigheten blir endret momentant for hvert trykk vil en filtrert versjon gi et bedre bilde over funksjonen til hastigheten, som vist i kodeutdrag 5.6.

Kode 5.6: Hastighet

```
1 Hastighet_IIR(k) = IIR_filter(Hastighet_IIR(k-1), ...
      Hastighet(k-1),alfa);
```

Vi velger å filtrere ved bruk av et IIR filter, som filtrerer basert på forrige måling.

Den total distansen kan bli regnet ut på flere måter. Vi valgte å multiplisere antall runder i "Runder" med omkretsen på hjulet. Denne utregningen ble gjort med variabelen `TotalDistanse(Runder)` som vist i kodeutdrag 5.7. Gjennomsnittshastigheten ble regnet ved at ta gjennomsnittet av alle hastighetene fra start. Dette gjorde vi gjennom å bruke funksjonen `Mean`, som er en innebygd MatLab funksjon som finner gjennomsnittet. Vi kalte variabelen `Gjennomsnittfart`, som vist i kodeutdraget 5.7.

Kode 5.7: Kode for totaldistanse og gjennomsnittsfart

```
1 TotalDistanse(Runder) = Runder * Hjul;
2 Gjennomsnittsfart(k) = mean(Hastighet_IIR);
```

Her finner vi gjennomsnittsfarten siden start ved å ta gjennomsnittet av de filtrerte hastighetene.

Akselrasjonen finner vi ved å derivere hastigheten. Vi valgte å deriverer den filtrerte hastigheten siden den gir et bedre bilde på hvordan hastigheten oppfører seg i realiteten. Vi bruker funksjonen `Derivation`, som vist i kodeutdrag 5.8

Kode 5.8: Derivation

```
1 function [Secant] = Derivation (FunctionValues, Timestep)
2 R = numel (FunctionValues);
3 Secant = ((FunctionValues(R)-FunctionValues(R-1))/Timestep(R-1));
4 end
```

`Derivation` er en funksjon for numerisk derivasjon. Videre bruker vi denne funksjonen for å avgjøre akselerasjonen til hjulet. Vi definerer akselerasjon ved å deriverer den filtrerte hastigheten, som vist i kodeutdrag 5.9.

Kode 5.9: Akselrasjon

```
1 Akselrasjon(k) = Derivation(Hastighet_IIR(k-1:k), Ts(k-1));
```

Her er akselrasjon definert ved å derivere den filtrerte hastigheten. Den deriveres ved bruk av den forrige filtrerte hastigheten og tidskrittet.

5.2.3 Forslag til å avslutte runden

Ved at koden er konstruert på en måte hvor bryteren trenger å bli trykt inn før en ny runde blir registrert, er det ingen måte å forsikre at runden blir avsluttet på. Slik det er nå vil også hastigheten kun bli oppdatert når ett trykk blir registrert. Det skaper et feil bilde av hvordan hastigheten egentlig er. For å få ett mer realistisk bilde valgte vi å lage variabelen `AlternativHastighet(k)`, vist i kodeutdraget under.

Kode 5.10: AlternativHastighet

```
1 AlternativHastighet(k) = (Hjul/(Tid(k) - Tid_R(Runder)))*3.6;
2 if AlternativHastighet(k) < Hastighet(k)
3     Hastighet(k) = AlternativHastighet(k);
4 end
```

`AlternativHastighet(k)` er hastigheten til hjulet dersom det skulle blitt registrert ett trykk i det bestemte tidspunktet. Videre satt vi at `Hastighet(k)` settes lik `AlternativHastighet(k)` dersom `AlternativHastighet(k)` er mindre enn `Hastighet(k)`. Slik blir hastigheten mer realistisk. Med ett mer realistisk bilde av hastigheten valgte vi å avslutte runden dersom hastigheten går under 0.5[km/t], som vist i kodeutdraget under.

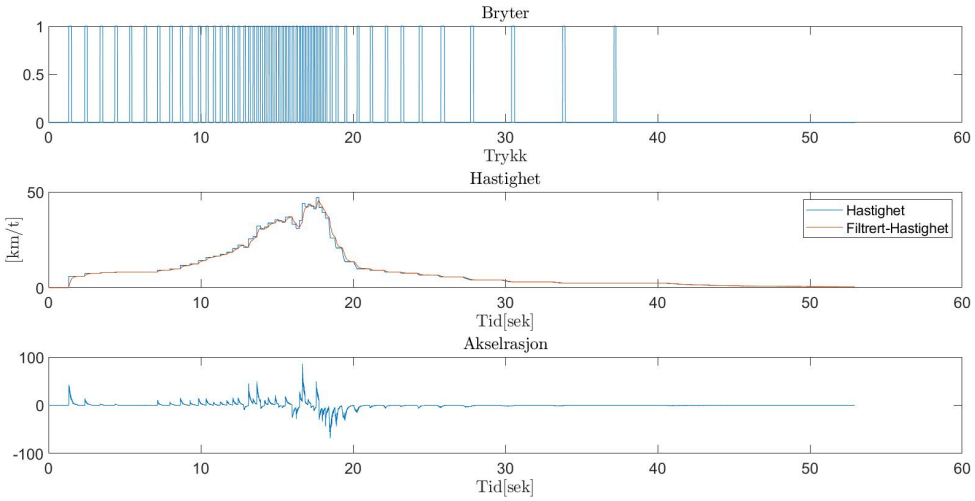
Kode 5.11: AlternativHastighet

```
1 if Hastighet(k) < 0.5 && Hastighet(k-1) > 0.5
2     break
3 end
```

Vi valgte å avslutte runden når hastigheten er mindre enn 0.5[km/t], dette fordi vi mener det er en såpass lav hastighet at det er tilnærmet med å stå stille.

5.3 Resultater

I dette kapitellet vil vi presentere resultatene fra sykkel computeren. Vi simulerer rotasjonene til et sykkelhjul ved å trykke på trykkbryteren. Ved å øke og senke frekvensen på trykkene vil vi vise endring i hastighet og akselrasjon. Plottet vil bestå av trykkbryteren, hastigheten og akselrasjonen. Vi vil også presentere nåverdiene til gjennomsnittsfarten og tilbakelagt strekning. Resultatet av trykkene vises i figuren under.



Figur 5.2: Simulering av sykkelcomputer, hvor runden avsluttes til slutt ved at hastigheten blir under . Plottet viser trykkbryteren, både filtrert og ufiltrert hastighet, og akselrasjonen.

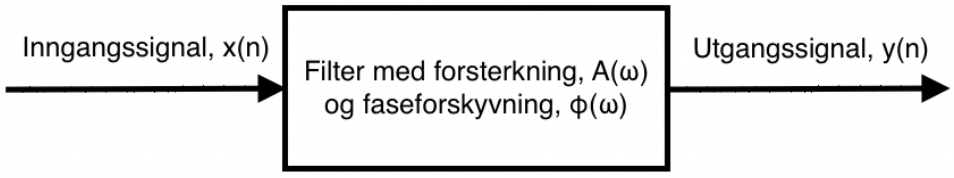
Fra denne figuren ser vi at trykkene gradvis øker før de igjen avtar. Vi ser at hastigheten samsvarer med trykkene, hvor hastigheten gradvis øker med frekvensen til trykkene. Hastigheten avtar også når frekvensen på trykkene minker. Basert på endringen mellom trykkene ser vi at akselrasjonen økes når tiden mellom trykkene blir mindre. Vi ser også at akselrasjonen blir negativ når tiden mellom trykkene blir større. Vi leser av nåverdiene og finner at:

- $$gjennomsnittsfarten = 8.642[km/t] \tag{5.3}$$
- $$TotalDistanse = 123.2m \tag{5.4}$$

Kapittel 6

Frekvensrespons

I dette kapittelet ble et *chirp-signal* filtrert gjennom forskjellige filtre. Formålet med prosjektet var å demonstrere effekten av de forskjellige filtrene, og å studere hvordan dette signalet endret seg i form av fase og amplitude da signalet ble filtrert.



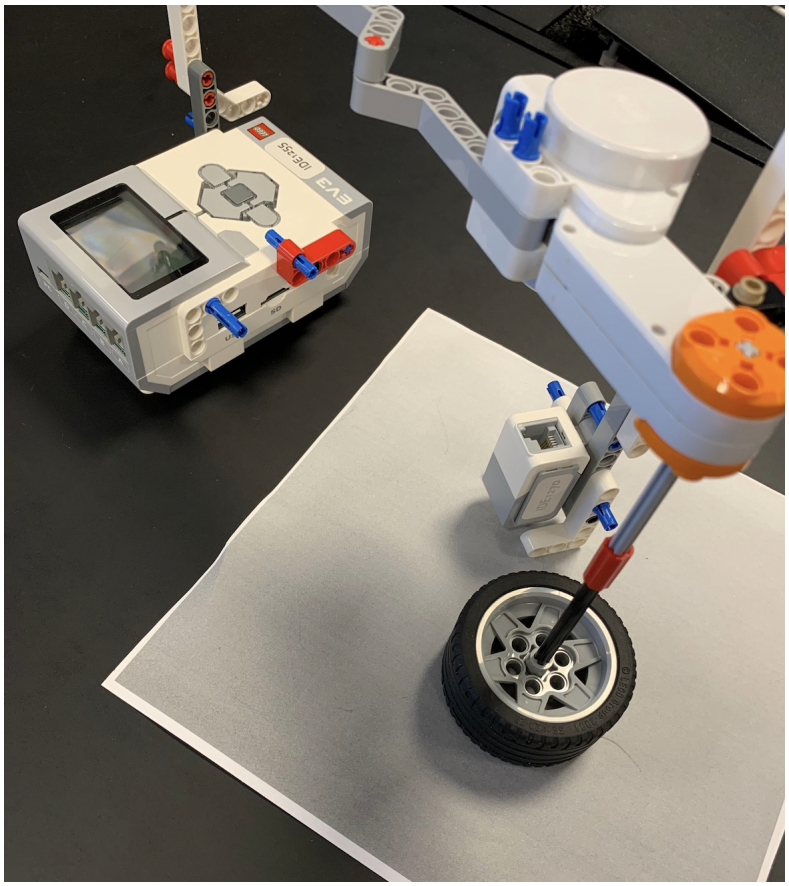
Figur 6.1: Inngangssignalet filtreres gjennom systemet, og gir ut et utgangssignal med forsterkning og faseforskyvning.

6.0.1 Utstyrsliste:

- Lego Mindstorms EV3 kloss med en motor og lyssensor
- Gråskala ark

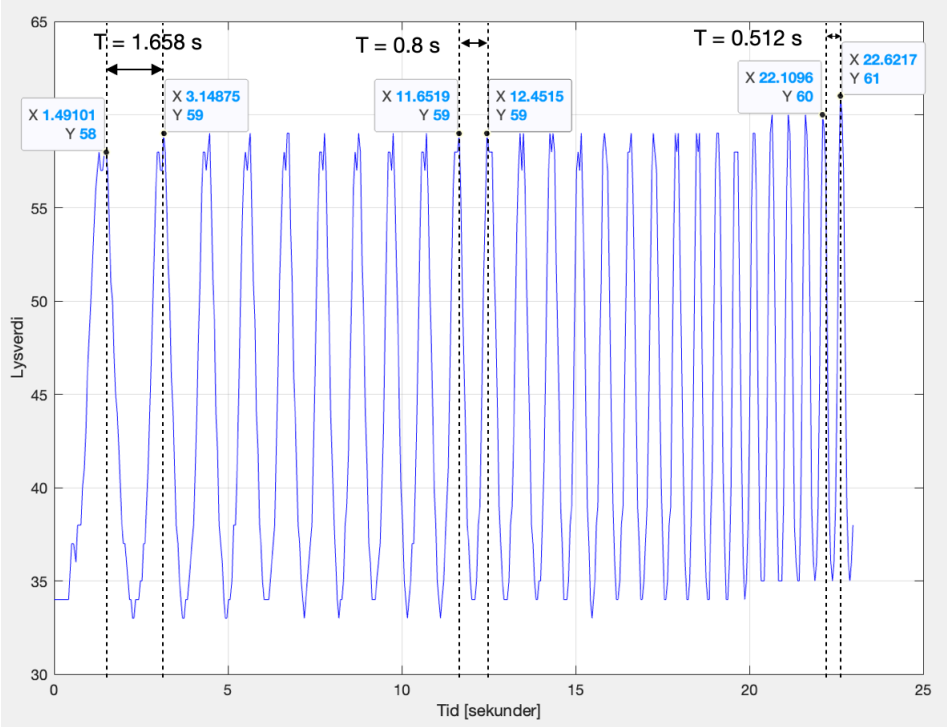
6.1 Sampling av signal

Før signalene skulle samples ble det bygd en lego konstruksjon der et hjul ble festet til en motor. Lyssensoren ble festet slik at den sto friksjonsfritt over et gråskala ark. Da motoren ble gitt et pådrag, gjorde friksjonen mellom arket og hjulet at de ble rotert sammen over den glatte overflaten på bordet. Lyssensoren registrerte da verdier mellom 0 og 100, der svart tilsvarer verdien 0, og hvit verdien 100. Forskjellige gråskalaverdier ligger da inni dette intervallet. Et chirp-signal øker i frekvens som funksjon av tiden. For å skape denne effekten ble motorens pådrag gradvis økt fra minimum til maksimum ved hjelp av styrestikken. Figur 6.2 viser både lego-konstruksjonen og gråskala arket som lego-konstruksjonen står på.



Figur 6.2: Konstruksjonen som ble benyttet for å sample lyssignalene.

Det samplede sinussignalet fra lyssensoren kan sees i figur 6.3.

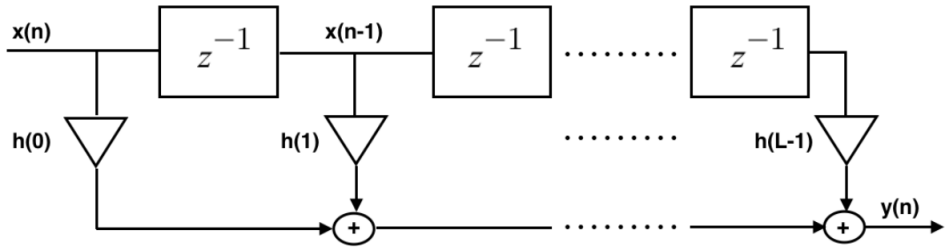


Figur 6.3: Det samplede chirp-signalet

Vi ser her at perioden mellom toppunktene minsker etterhvert som tiden øker. Amplituden holder seg forholdsvis konstant, og frekvensen øker fra 0.6 til 1.95 Hz.

6.2 Anvendelse av filtre

For å anvende diverse filtre på *chirp*-signalet vårt krever det filtre av svært høy orden. Dette er fordi at det er så liten differanse mellom den laveste frekvensen og den høyeste frekvensen, noe som forutsetter at helningen på filteret må være svært bratt for at signalet skal bli dempet "i tide". Figur 6.4 viser et blokkskjema av et høyere ordens FIR-filter.



Figur 6.4: Blokkskjema av et FIR filter. Inngangssignalet multipliseres med en filterkoeffisient og summeres med en endelig mengde tidligere inngangssignaler som er multiplisert med tilhørende filterkoeffisienter.

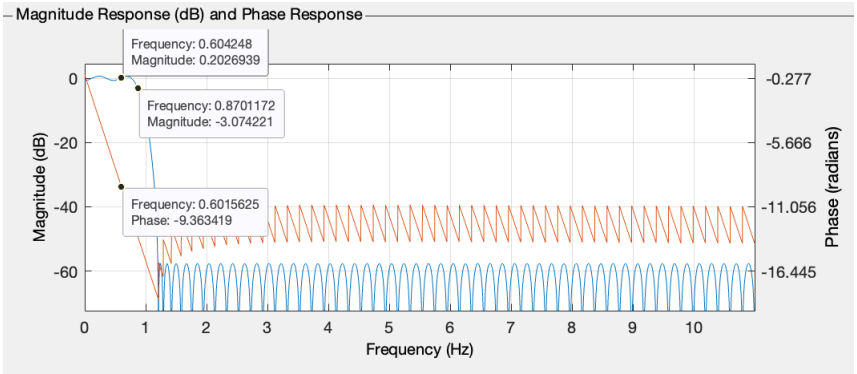
Differenslikningen til filteret i figur 6.4 er anvist i likning 6.1.

$$y(n) = \sum_{k=0}^{L-1} h(k) \cdot x(n-k) \tag{6.1}$$

Det er mange måter å lage digitale filtre på. I dette prosjektet har vi valgt å bruke MatLab's verktøy $filterDesigner$. Her spesifiseres hva slags filter man ønsker å lage, deretter spesifiseres spesifikasjonene på filteret. Etter dette eksporterte vi filteret som et objekt til MatLabs workspace, og plottet signalet der.

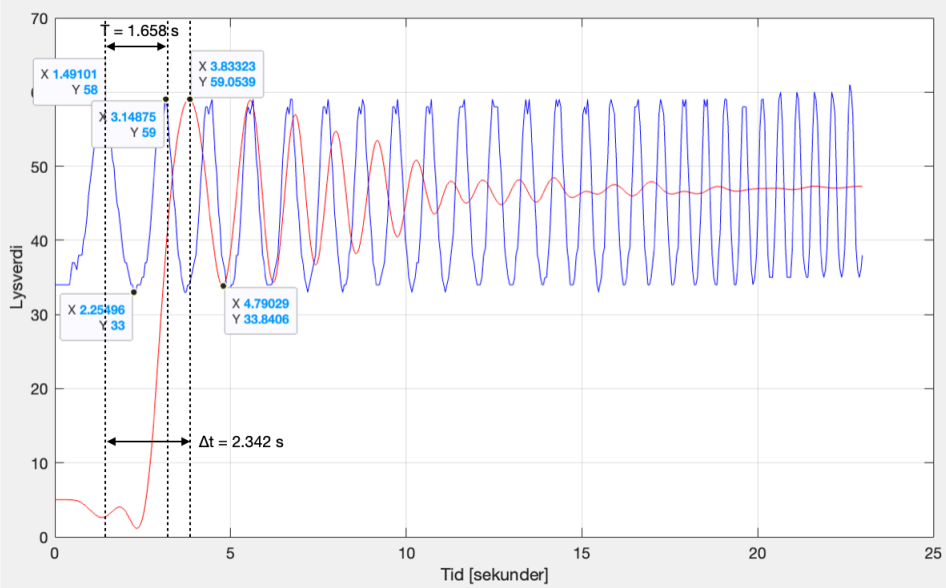
6.2.1 Lavpass filter

Det ble designet et Lavpass-FIR filter av 109. orden. I filterDesigner er spesifikasjonene Fstop = 0.8 Hz og Fpass = 1.2 Hz. Det vil si at ved 0.8 Hz skal dempingen være -3 dB, og for høyere frekvenser enn dette blir dempingen eksponentielt sterkere. Figur 6.5 viser bode diagrammet til filteret.



Figur 6.5: Det blå signalet er magnituderesponsen til filteret, mens det røde signalet er faseresponsen.

Fra bode diagrammet ser vi at knekkfrekvensen er 0.87 Hz, og ved 0.6 Hz er forsterkningen 0.2 dB og faseforskyvningen -9.36 rad/s. I figur 6.6 ser vi inngangssignalet og utgangssignalet.



Figur 6.6: Inngangssignalet er i blått, mens utgangssignalet er i rødt.

Det filtrerte signalet i rødt slipper gjennom de lavere frekvensene, og demper de høyere frekvensen i chirp-signalet.

For å verifisere filterets amplitudeforsterkning(A) og faseforskyvning(ϕ) må vi regne ut amplituden(A), perioden(T) og forsinkelsen(Δt) til signalene.

Amplitudeforsterkningen er gitt ved likning 6.2

$$A = \frac{Y}{U} \rightarrow \frac{\frac{59.05-33.84}{2}}{\frac{58-33}{2}} \rightarrow \frac{12.605}{12.5} = 1.0084 \rightarrow 20log(1.0084) = 0.073\ dB \tag{6.2}$$

I likning 6.2 tilsvarer Y, amplituden til utgangssignalet, og U, amplituden til inngangssignalet. Fra faseresponsen i figur 6.7 ser vi at amplitudeforsterkningen ved 0.6 Hz er 0.202 dB, den beregnede amplitudeforsterkningen er altså litt lavere,

Perioden er gitt ved likning 6.3

$$T = 3.149\ s - 1.491\ s = 1.658\ s \tag{6.3}$$

Forsinkelsen er gitt ved likning 6.4

$$\Delta t = 3.833\ s - 1.491\ s = 2.342\ s \tag{6.4}$$

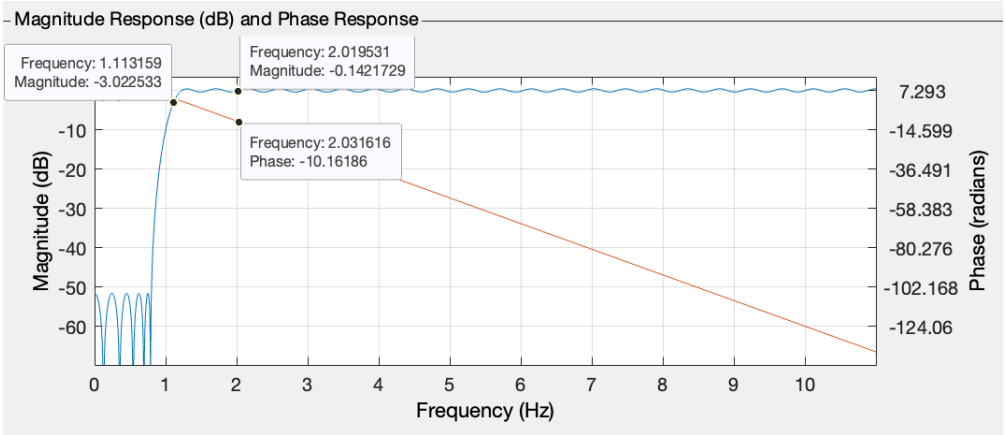
Faseforskyvningen er gitt ved likning 6.5:

$$\phi = \frac{360° \cdot \Delta t}{T} \rightarrow \frac{360° \cdot 2.342}{1.658} = -508.52° \cdot \frac{\pi}{180} = -8.88\ rad/s \tag{6.5}$$

Ettersom utgangssignalet henger etter inngangssignalet, vil faseforskyvningen, ϕ, i likning 6.5 være negativ. Fra bodediagrammet i figur 6.5 ser vi fra faseresponsen at faseforskyvningen til filteret er -9.36 rad/s ved frekvensen 0.6 Hz. Den beregnede faseforskyvningen er altså litt større.

6.2.2 Høypass filter

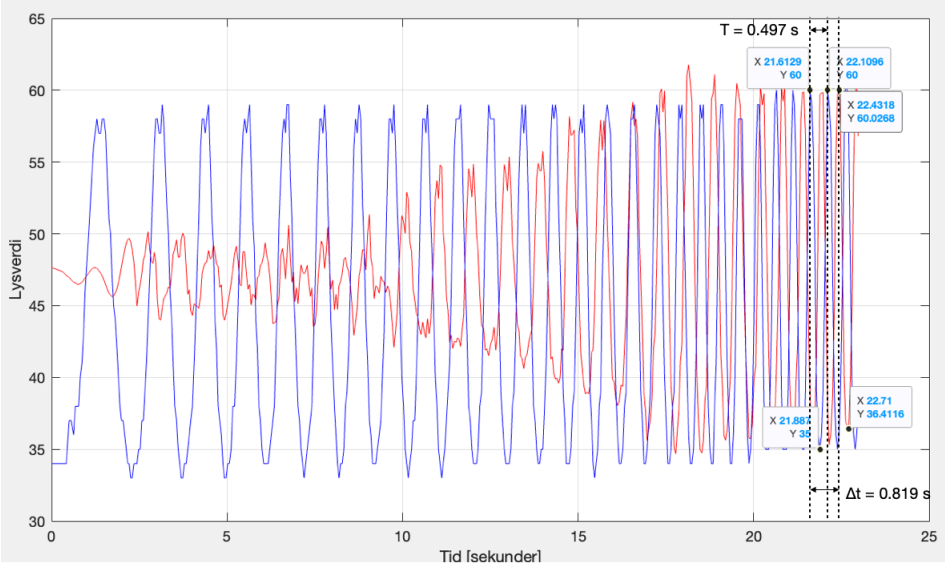
Her har vi designet et høypass-FIR filter av 100. orden. I filterDesigner er spesifikasjonene Fstop = 0.8 Hz og Fpass = 1.2 Hz. Det vil si at ved 1.2 Hz skal dempingen ideelt sett være -3 dB, og ved lavere frekvenser enn dette blir dempingen eksponentielt sterkere. Ved 0.8 Hz er dempingen -50 dB. Figur 6.7 viser bodediagrammet til filteret.



Figur 6.7: Knekkfrekvensen er omtrent 1.1 Hz, ved 2 Hz er amplitudeforsterkningen -0.14 og faseforskyvningen -10.16 rad/s.

I figur 6.8 ser vi resultatet av å filtrere chirp-signalet med høypassfilteret.

De lavere frekvensene i utgangssignalet blir ikke helt dempet, men vi ser en klar tendens til høypassfiltrering.



Figur 6.8: Inngangssignalet er i blått, mens utgangssignalet er i rødt.

For å verifisere filterets amplitudeforsterkning(A) og faseforskyvning(ϕ) må vi regne ut amplituden(A), perioden(T) og forsinkelsen(Δt) til signalene.

Amplitudeforsterkningen er gitt ved likning 6.6

$$A = \frac{Y}{U} \rightarrow \frac{\frac{60.03-36.41}{2}}{\frac{60-35}{2}} \rightarrow \frac{11.81}{12.5} = 0.945 \rightarrow 20log(0.945) = -0.491\ dB \tag{6.6}$$

Fra bodediagrammet i figur 6.7 ser vi at amplitudeforsterkningen ved 2 Hz er -0.142 dB, den beregnede amplitudeforsterkningen er altså litt lavere.

Perioden er gitt ved likning 6.7

$$T = 22.11\ s - 21.613\ s = 0.497\ s \tag{6.7}$$

Forsinkelsen er gitt ved likning 6.8

$$\Delta t = 22.432\ s - 22.613\ s = 0.819\ s \tag{6.8}$$

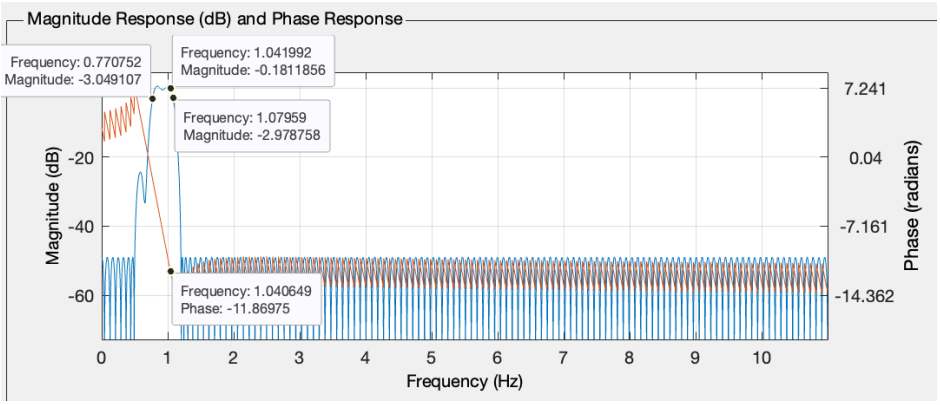
Faseforskyvningen er gitt ved likning 6.9:

$$\phi = \frac{360° \cdot \Delta t}{T} \rightarrow \frac{360° \cdot 0.819\ s}{0.497\ s} = -593.24° \cdot \frac{\pi}{180} = -10.35\ rad/s \quad (6.9)$$

Utgangssignalet henger etter inngangssignalet, dermed blir faseforskyvningen negativ. Fra faseresponsen i bodediagrammet 6.7 ser vi at faseforskyvningen ved 2 Hz for høypassfilteret er -10.16 rad/s. Dette samsvarer med faseforskyvningen vi har regnet ut.

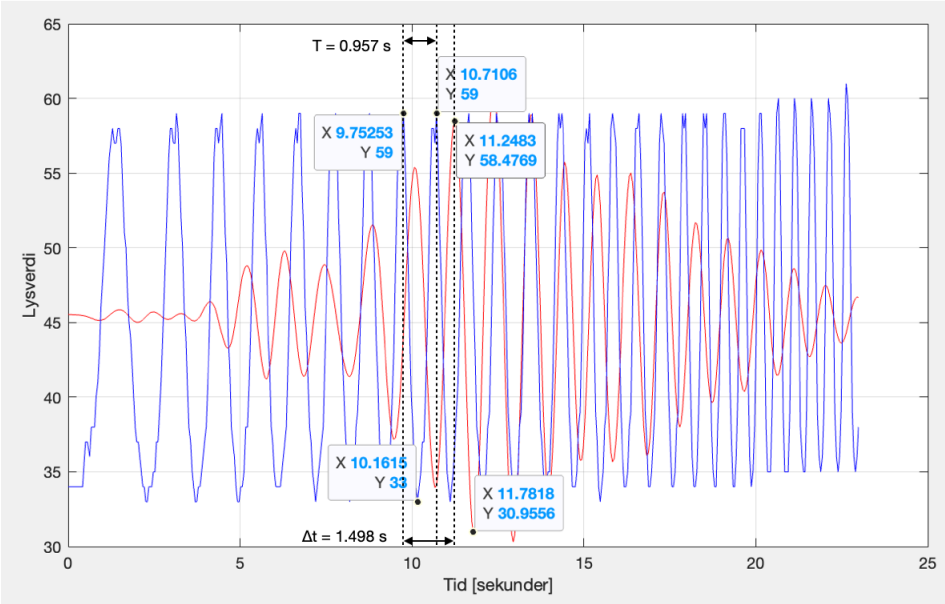
6.2.3 Båndpass filter

Figur 6.9 viser bodediagrammet til et båndpass FIR-filter av 249. orden. Differansen mellom passbåndet og stoppbåndet så liten at det krever et filter av svært høy orden for at helningen skal rekke å nå dempningsverdien. Filterets båndbredde ble designet for intervallet 0.8 Hz til 1.05 Hz. Den reelle båndbredden kan vi lese av datapunktene til å være mellom 0.77 Hz og 1.08 Hz.



Figur 6.9: De to knekkfrekvensene avleses til 0.77 og 1.08 Hz. Ved 1.05 Hz er faseforskyvningen -11.87 rad/s og forsterkningen -0.18 dB.

I figur 6.10 ser vi effekten av båndpassfilteret. De nedre og øvre frekvensene blir dempet, mens frekvensene i båndbredden 0.69 til 1.0 Hz slipper gjennom.



Figur 6.10: Inngangssignalet er i blått, mens utgangssignalet er i rødt.

For å verifisere filterets amplitudeforsterkning(A) og faseforskyvning(ϕ) må vi regne ut amplituden(A), perioden(T) og forsinkelsen(Δt) til signalene.

Amplitudeforsterkningen er gitt ved likning 6.10

$$A = \frac{Y}{U} = \frac{\frac{58.48-30.95}{2}}{\frac{59-33}{2}} = \frac{13.77}{13} = 1.059 \rightarrow 20log(1.059) = 0.498\ dB \quad (6.10)$$

Fra figur 6.9 ser vi at amplitudeforsterkningen ved 1.04 Hz er -0.181 dB, den beregnede amplitudeforsterkningen er altså litt større.

Perioden er gitt ved likning 6.11

$$T = 10.71\ s - 9.753\ s = 0.957\ s \quad (6.11)$$

Forsinkelsen er gitt ved likning 6.12

$$\Delta t = 11.248\ s - 9.75\ s = 1.498\ s \quad (6.12)$$

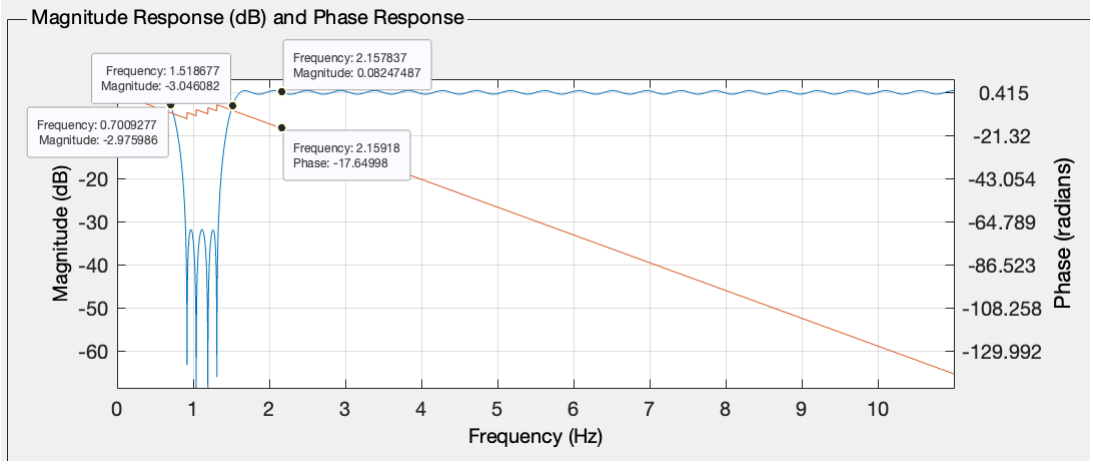
Faseforskyvningen er gitt ved likning 6.13:

$$\phi = \frac{360° \cdot \Delta t}{T} \rightarrow \frac{360° \cdot 1.498}{0.957} = -563.51° \cdot \frac{\pi}{180} = -9.835\ rad/s \quad (6.13)$$

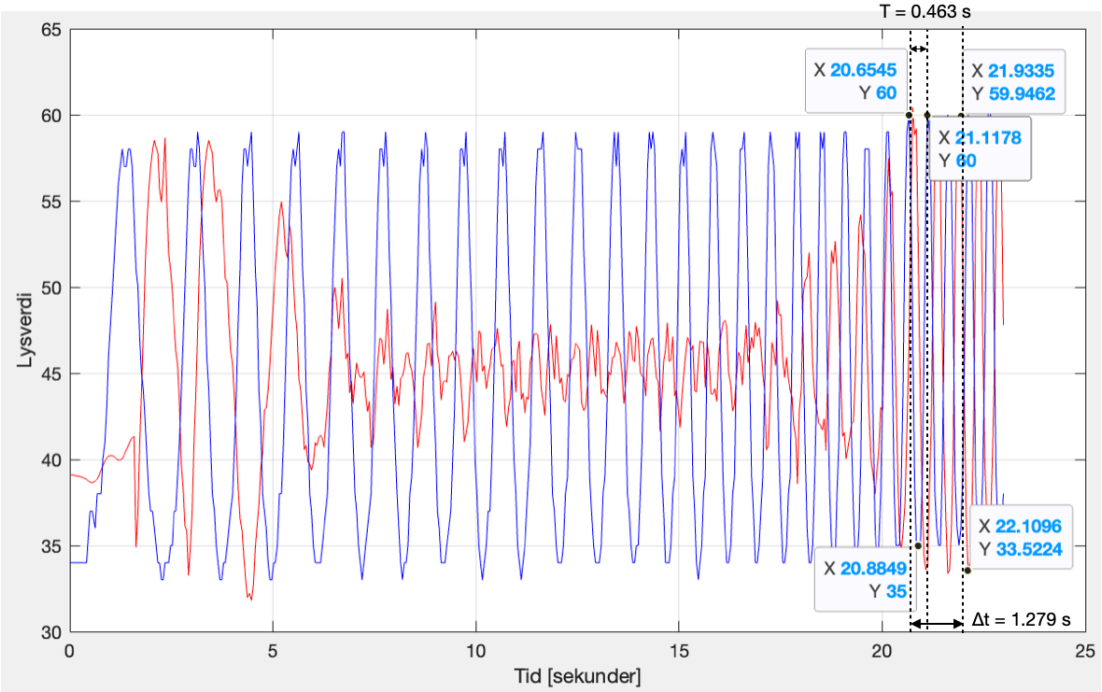
Også her henger utgangssignalet etter inngangssignalet, dermed blir faseforskyvningen negativ også i dette tilfellet. Fra faseresponsen i bodediagrammet 6.9 ser vi at faseforskyvningen ved 1.04 Hz er -11.87 rad/s. Den utregnede faseforskyvningen er altså litt større.

6.2.4 Båndstopp filter

Det siste filteret vi designet var et båndstopp-FIR filter av 98. orden. Filterets stoppbånd ble designet for båndbredden 0.9 til 1.3 Hz, og fra bodediagrammet i figur 6.11 ser vi at stoppbåndets reelle båndbredde er 0.7 til 1.52 Hz.



Figur 6.11: Stoppbåndets båndbredde er 0.7 til 1.52 Hz, amplitudeforsterkningen er 0.08 dB og faseforskyvningen -17.65 rad/s ved 2.16 Hz.



Figur 6.12: Inngangssignalet(U) er i fargen blå, mens utgangssignalet(Y) er i fargen rød.

For å verifisere filterets amplitudeforsterkning(A) og faseforskyvning(ϕ) må vi regne ut amplituden(A), perioden(T) og forsinkelsen(Δt) til signalene.

Amplitudeforsterkningen er gitt ved likning 6.14

$$A = \frac{Y}{U} \rightarrow \frac{\frac{59.95-33.52}{2}}{\frac{60-35}{2}} \rightarrow \frac{13.22}{12.5} = 1.058 \rightarrow 20log(1.058) = 0.490\ dB \tag{6.14}$$

Fra figur 6.11 ser vi at amplitudeforsterkningen ved 2.16 Hz er 0.082 dB, den beregnede amplitudeforsterkningen er altså litt høyere.

Perioden er gitt ved likning 6.15

$$T = 21.118\ s - 20.655\ s = 0.463\ s \tag{6.15}$$

Forsinkelsen er gitt ved likning 6.16

$$\Delta t = 21.934\ s - 20.655\ s = 1.279\ s \tag{6.16}$$

Faseforskyvningen er gitt ved likning 6.17:

$$\phi = \frac{360^\circ \cdot \Delta t}{T} \rightarrow \frac{360^\circ \cdot 1.279\ s}{0.463\ s} = -994.47^\circ \cdot \frac{\pi}{180} = -17.36\ rad/s \tag{6.17}$$

Utgangssignalet henger etter inngangssignalet, dermed blir faseforskyvningen negativ. Fra faseresponsen i bodediagrammet 6.9 ser vi at faseforskyvningen ved 2.16 Hz er -17.65 rad/s. Den utregnede faseforskyvningen samsvarer godt med faseforskyvningen fra bodediagrammet.

6.3 Konklusjon

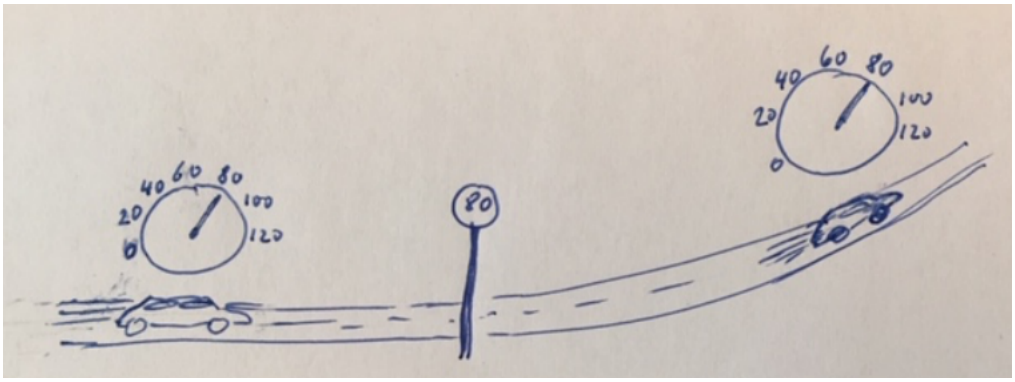
Vi har i dette kapittelet vist hvordan diverse filtre kan benyttes for å dempe uønskede frekvenser fra et signal. Filtrenes bodediagrammer ble verifisert mot de filtrerte signalenes amplitude og faseforskyvning, og resultatene fra dette samsvarer bra med hverandre.

Kapittel 7

PID-regulator

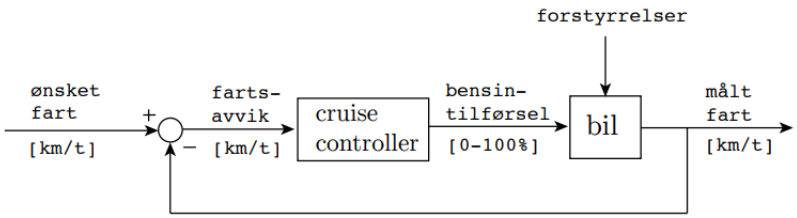
7.1 Problemstilling

I dette prosjektet har målet vært å lage en turtallregulator for motorene på LEGO-roboten. En turtallregulator har vært nødvendig i de prosjektene der vi bruker to motorer for å kjøre roboten, blant annet i prosjektet robot-støvsugeren 8. Funksjonen til turtallregulatoren er å opprettholde den ønskede hastigheten på hjulene, selvom den møter motstand. Et eksempel på motstand kan være i form av en motbakke, som vist i 7.1.



Figur 7.1: En bil med turtallregulator som vil automatisk holde ønsket hastighet, uansett om bilen kjører rett frem eller i en motbakke [9]

For å oppnå slik turtallregulator må roboten måle den nåværende hastighet (*målt fart*), og sammenligne den mot den ønskende hastigheten (*ønsket fart*). Hvor forskjellen på den nåværende hastigheten og den ønskede hastigheten blir et *farts-avvik* som enten er positivt eller negativt. Utifra *avviket* vil regulatoren automatisk regulere *pådraget* den skal gi til motorene for å oppnå den ønskede hastigheten. Dette er prinsippet bak *cruise controllen* på moderne biler, som vist i figur under.



Figur 7.2: Blokkskjema som viser prinsippet bak en *cruise controller* i en bil. Hvor *farts-avviket* er forskjellen mellom *ønsket-fart* og *målt-fart*. *farts-avviket* avgjør hvor mye *motor-pådrag* som skal gis til motorene. I en bil vil *motor-pådraget* justeres ved *bensintilførselen*. *Forstyrelser* vil være alt som minsker eller øker motstand på bilen. Eksempler på dette kan være medvind eller motbakke.[9]

I de prosjektene vi har brukt en *turtallregualtor* har *motstanden* vært iform av friksjon fra bakken. Det har vist seg at når LEGO-roboten skal kjøre rett fremover, hvor samme *pådrag* blir gitt til begge motorene kan roboten ta en uventet bane. Dette kan skje siden hjulene vil ikke møte på lik friksjon langs kjøreveien. Ved å bruke en *turtallregulator* har vi sikret oss at roboten holder den planlagte banen ved at begge hjulene holder *lik* hastighet, uansett hva slags *friksjon* hjulene skulle møte. Vi har derfor lagd en *PID-regulator* til motorene som har funksjonaliteten til å automatisk holde ønsket hastighet. Vi vil beskrive/dokumentere:

- Oppsett av robot og testing
- Kode for beregning av vinkelhastighet (*målt-fart*)
- Testing uten turtallregulator
- Sammenheng mellom pådrag og vinkelhastighet

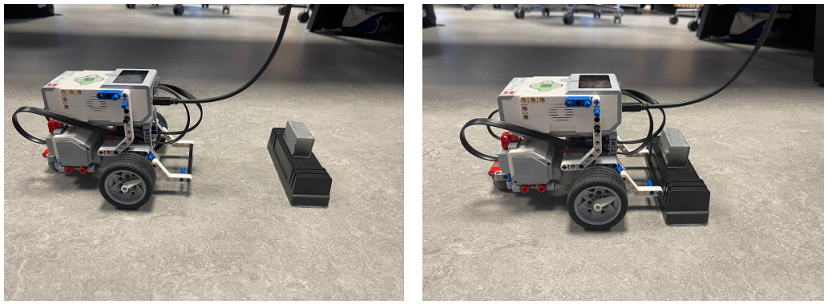
- Kode for å implementere PID-regulatoren
- Regne frem passende regulator parametre
- Testing av turtallregulatoren
- Resultater

7.2 Forslag til løsning

I dette delkapittelet presenterer vi vårt forslag til løsningen av turtallregulatoren. Dette inkluderer oppsettet for hvordan vi utsatte LEGO motorene for relativt lik motstand gjennom forsøkene. I tillegg presenteres koden for å implementere PID-regulatoren. Deretter vil vi regne frem passende regulator parametre.

7.2.1 LEGO-konstruksjonen

For å teste turtallregulatoren valgte vi å bygge LEGO-konstruksjonen etter instruksjonene fra LEGO-mindstorms sine nettsider [2]. Konstruksjonen til roboten vises i bildet 7.3. For praktisk testing av turtallregulatoren valgte vi gjennomføre to eksperimenter. I det første eksperimentet valgte vi å teste om turtallregulatoren får roboten til å kjøre rett fremover når begge hjulene settes til å ha samme hastighet. I det andre eksperimentet testet vi videre om turtallregulatoren klarte å holde ønsket hastighet selvom vi påførte økt motstand. Dette gjennomførte vi ved å kjøre roboten på bakken for at den så skal treffe på en kloss som gir økt motstand, som vist i figur 7.3.



Figur 7.3: Oppsett av LEGO-roboten for å teste turtallregulatoren med friksjon fra bakken og videre økt motstand

Vi velger å utføre testingen ved å kjøre roboten på bakken fordi vi ønsker å teste turtallregulatoren i de samme omstendighetene som prosjektet 4. Dette gjør vi for å kunne regne frem de passende regulator parametrene slik at regulatoren kan brukes i de nevnte prosjektene. Selve testingen av turtallregulatoren vil være å teste om roboten holder rett bane, selvom hjulene vil oppleve ulik friksjon fra bakken. Deretter teste om turtallregulatoren øker motor-pådraget når roboten opplever økt motstand med klossen, slik at roboten holder samme hastighet som før møte med klossen.

7.2.2 Kode for beregning av vinkelhastighet

I dette delkapittelet vil vi først presentere koden for å gi pådrag til LEGO-motorene, deretter hvordan vi beregner vinkelhastigheten utifra vinkelposisjonen til motorene.

For å kjøre LEGO-roboten fremover eller bakover gis det et motorpådrag mellom -100 og 100, som oppgis i [%] Her vil en positiv verdi drive motoren fremover, og en negativ verdi vil drive motoren bakover. Under uttestingen av turtallregulatoren valgte vi å sette pådraget til motorene i selve koden. Ved å ha variabel for tid kunne vi sette pådraget for den bestemte tiden vi ønsket. I koden under vises et eksempel på at vi velger at motorpådraget skal settes til 10% i tretten sekunder, etter at den først har stått stille i to sekunder. Grunnen til at den først står stille i to sekunder er for å tydeligere se motorpådraget i plottet.

Kode 7.1: Kode for motorpådrag

```
1  if Tid(k) ≤2
2      PowerB(k) = 0;
3      PowerA(k) = 0;
4  elseif Tid(k) ≤ 15
5      PowerB(k) = 10;
6      PowerA(k) = 10;
7  else
8      PowerB(k) = 0;
9      PowerA(k) = 0;
10 end
```

Selve koden vil gjøre at roboten kjører fremover etter to sekunder og at den stopper etter tretten sekunder. Etter utført kode **??** vil roboten ha kjørt fremover i tretten sekunder med ett pådrag på 10%. I testingen av regulatoren vil vi hovedsak bruke dette oppsettet hvor roboten kjører i 13 sekunder etter å ha stått stille i to sekunder.
Beregningen av vinkelhastigheten [grader/sek], regnes frem ved å derivere avlesningen fra vinkelposisjonen [grader] til motorene. Vi derivere ved å bruke en funksjon for numerisk derivasjon, som vist i kodeutdraget under 7.2

Kode 7.2: Kode for å numerisk derivasjon

```
1 function [Secant] = Derivation (FunctionValues, Timestep)
2 R = numel (FunctionValues);
3 Secant = ((FunctionValues(R)-FunctionValues(R-1))/Timestep(R-1));
4 end
```

Hvor

- FunctionValue(k-1:k) (funksjonsverdien som skal bli integrert).
- FunctionValue(k-1) er funksjonsverdien for forrige indeks.
- FunctionValue(k) er nåverdien til funksjonen.
- TimeStep (tidsforskjellen mellom den indeksen k og k-1)

Med en funksjon for *numerisk derivasjon* er det mulig å derivere *vinkelposisjon målingene* til motorene. Men siden *vinkelposisjon målingene* i utgangspunktet er utsatt for en del *støy* velger vi å *filtrere* de først siden ellers vil

det gi ubrukelig resultater. Vi filtrer målingene ved bruk av ett *IIR filter*. Den nødvendige koden for å beregne vinkelhastigheten vises i 7.3

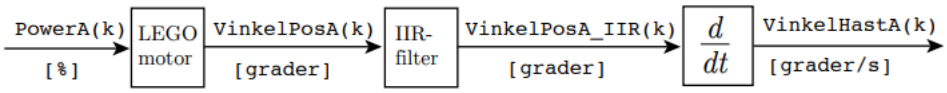
Kode 7.3: Kode for å beregne vinkelhastigheten

```
1  alfa = 0.3
2  VinkelPosMotorA(k) = double(motorA.readRotation);
3  VinkelPosMotorB(k) = double(motorB.readRotation);
4  VinkelPosA_IIR(k) = ...
       IIR_filter(VinkelPosA_IIR(k-1),VinkelPosMotorA(k),0.3);
5  VinkelPosB_IIR(k) = ...
       IIR_filter(VinkelPosB_IIR(k-1),VinkelPosMotorB(k),0.3);
6  VinkelHastighetA(k-1) = Derivation(VinkelPosA_IIR(k-1:k),Ts(k-1));
7  VinkelHastighetB(k-1) = Derivation(VinkelPosB_IIR(k-1:k),Ts(k-1));
```

Hvor

- `alfa`, Verdien på `alfa=0.3` i linje 6 ble funnet etter endel prøving og feiling.
- `VinkelPosMotorA(k)` og `VinkelPosMotorB(k)` er avlesningen av vinkelposisjonene til `MotorA` og `MotorB` .
- `VinkelPosAIIR(k)` og `VinkelPosBIIR(k)` er den filtrert vinkelposisjon.
- `VinkelHastighetA(k-1)` og `VinkelHastighetB(k-1)` er beregnet vinkelhastighet.

Oppsummert kan kodeutdraget 7.3 fremstilles i blokkdiagram 7.4.

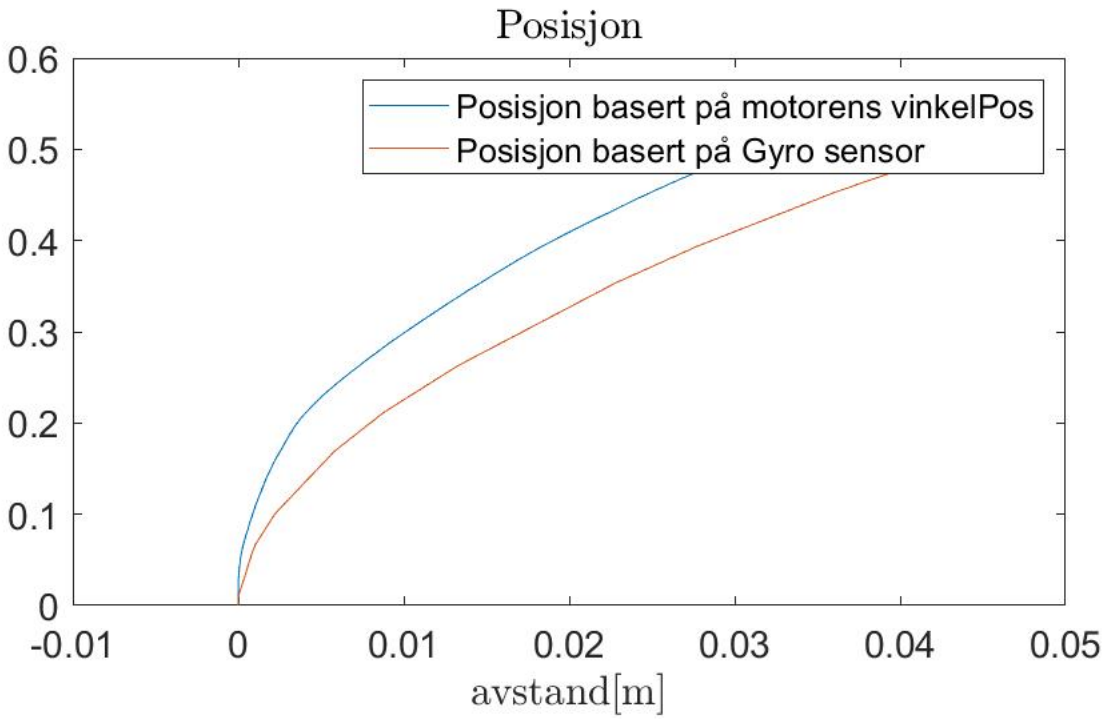


Figur 7.4: Blokkskjema over koden for beregning av vinkelhastighet [9]

7.2.3 Testing uten turtallsregulator

Eksperiment holde rett bane

For å finne ut av hvordan LEGO-moterene håndterte friksjon fra bakken uten turtallsregulatoren valgte vi å la roboten kjøre fremover med samme pådrag på begge hjulene. Dette gjorde vi fordi vi ønsket å se om roboten kjørte rett fremover eller om den svingte til en av sidene på grunn av ulik friksjon mellom hjulene. Vi gjennomførte testingen med ett motorpådrag på 10%. Vi valgte å sette motorpådraget til 10% siden det er det vi har valgt som standard i de prosjektene hvor roboten skal kjøre. Dette gjør vi for å sikre oss at regulator parametrene blir mest mulig nøyaktige, dette blir nærmere forklart i 7.2.6. Koden for motorpådraget er vist og forklart i 7.1. Et typisk resultat fra eksperimentet kan ses i figur 7.5.

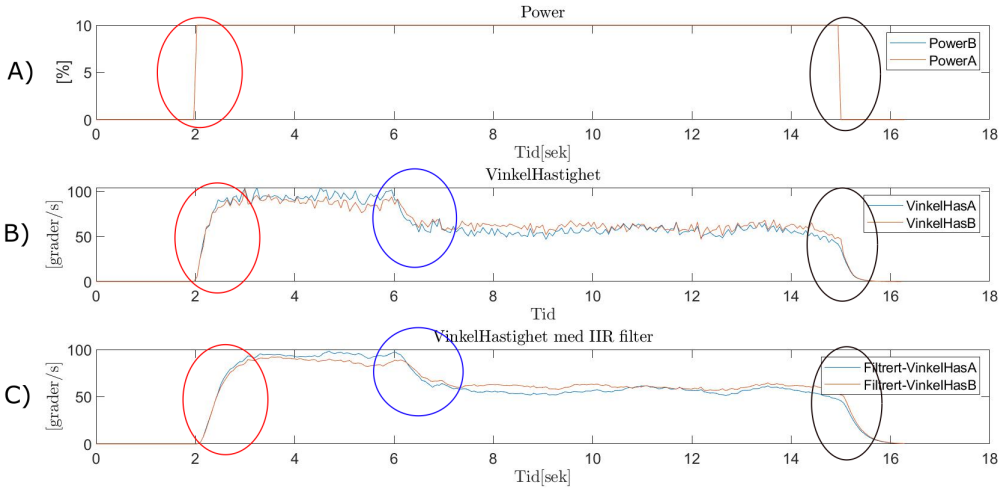


Figur 7.5: Testing av eksperiment holde rett bane uten turtallregulator. Hvor roboten kjører fremover med samme motorpådrag på begge hjulene. Hvor oransje linje viser posisjonen til roboten basert på gyro sensoren, og blå linje er posisjonen basert på vinkel posisjonen til motoren. Koden til plotene er vist og forklart i kapitellet *tegne bane* 4

Fra figuren ser vi banen som roboten tilbakelegger etter å ha et motorpådrag på 10% på begge hjulene i 13 sek. Figuren viser to ulike posisjons grafer, hvor den oransje er regnet frem ved bruk av gyro, og den blå ved vinkelposisjonen til motorene. Fremgangsmåten til hvordan vi plotter posisjonen gjennom gyro og vinkelposisjonen til motorene forklares og vises i kapitellet Tegne bane 4. Selve figuren viser at roboten tar en uventet bane mot venstre. Den svinger omtrent `6cm` mot venstre fra startpunkt. En av grunnene til at roboten svinger mot venstre er fordi hjulene opplever ulik friksjon fra bakken, derfor er det nødvendig med en turtallregulator for å sikre at roboten holder rett bane.

Eksperiment bevare hastighet ved møte med motstand

For å finne ut hvordan LEGO-motorene håndterte økt motstand uten turtallregulator gjennomførte vi eksperimentet hvor roboten møter på en kloss som den må skyve, som vist i 7.3. Koden for å gi motorpådraget er vist og forklart i kodeutdraget 7.1. Et typisk resultat for dette eksperimentet vises i figur 7.6



Figur 7.6: Testing av eksperiment bevare hastighet ved møte med motstand uten turtallregulator, hvor roboten kjører fremover og møter på økt motstand i form av en kloss den må skyve. I figuren ser vi at de røde sirklene markerer hvor pådraget til motorene settes til 10. De blå sirklene markerer hvor roboten møter på klossen som gir økt motstand. De sorte sirklene markerer hvor pådraget på 10 til motorene avsluttes. A) Pådragsverdier til LEGO-motorene. B) Vinkelhastigheten til motorene ufiltrert. C) Vinkelhastigheten til motorene filtrert med IIR-filter.

Utifra de markerte punktene av de blå sirklene i delfigur B og C ser man roboten sitt møte med klossen. De blå sirklene markerer at vinkelhastigheten går ned, og dette kan være grunnet til at klossen gir økt motstand som senker vinkelhastigheten. Med en PID-regulator vil vinkelhastigheten kunne opprettholdes selvom roboten opplever økt motstand.

7.2.4 Sammenheng mellom pådrag og vinkelhastighet

For å kunne oppnå den ønskede hastigheten er pådraget til motorene nødt til å bestemmes utifra avviket mellom ønsket og nåværende hastighet. Dette blir en sammenligning hvor man trekker fra *målt fart* fra *ønsket fart*, som vist i ligning under.

$$fartsavvik = onsketfart - maltfart \tag{7.1}$$

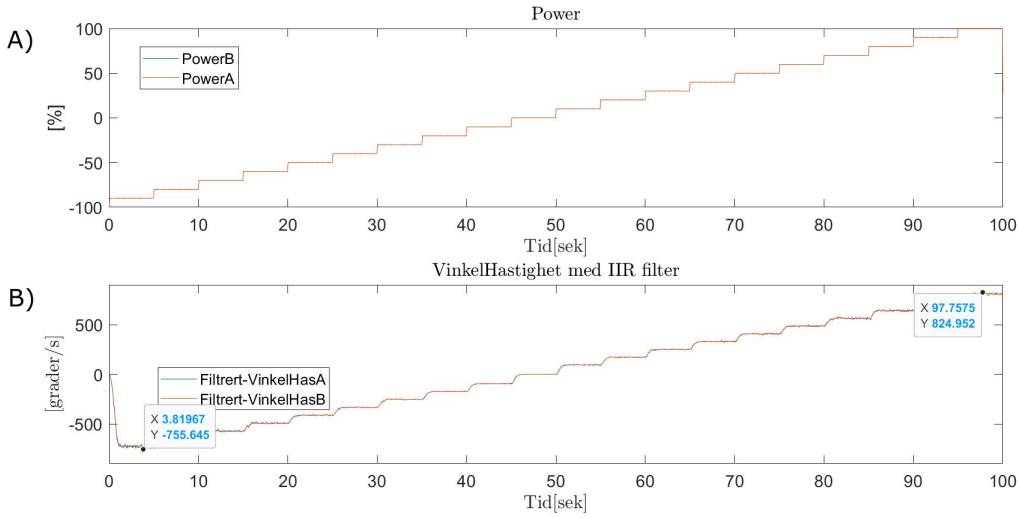
Dersom avviket er positivt (det vil si at ønsket hastighet er større enn nåværende hastighet), vil regulatoren øke pådraget til den nåværende hastigheten er lik den ønskede (og motsatt dersom avviket er negativt). En forutsetning for å beregne fartsavviket er at de er på samme benevning, vi valgte å bruke og regne med benevningen *[grader/s]* for både *målt hastighet* og *ønsket fart*.

LEGO-roboten har en grense på hvor stor fart den kan maksimalt oppnå både fremover og bakover, dette blir grensene for oppnåelige verdier for ønsket fart. Ved å gi LEGO-roboten et motor-pådrag fra -100% til +100%, kan vi spesifisere den maksimale oppnåelig vinkelhastigheten. Kodeutdraget 7.4 setter motorpådraget for begge hjulene fra -100% til +100% på intervaller på fem sekunder med en økning på 10 på motorpådraget. Kodeutdraget er vist og forklart koden under 7.4.

Kode 7.4: Kode for å finne maksimal oppnåelig ønsket hastighet

```
1  if Tid(k) ≤ 0
2      PowerB(k) = -100;
3      PowerA(k) = -100;
4  elseif Tid(k) ≤ 5
5      PowerB(k) = -90;
6      PowerA(k) = -90;
7  ....
8      PowerB(k) = 0;
9      PowerA(k) = 0;
10 elseif Tid(k) ≤ 60
11     PowerB(k) = 10;
12     PowerA(k) = 10;
13 ....
14 elseif Tid(k) ≤ 130
15     PowerB(k) = 100;
16     PowerA(k) = 100;
17 else
18     PowerB(k) = 0;
19     PowerA(k) = 0;
20 end
```

Med koden fra 7.4 kan vi utføre ett eksperiment hvor motorene til LEGO-roboten får ett jevnt pådrag fra -100 til 100. Dette eksperimentet utføres uten at hjulene utsettes for friksjon. Resultatet fra kan ses i figur under.

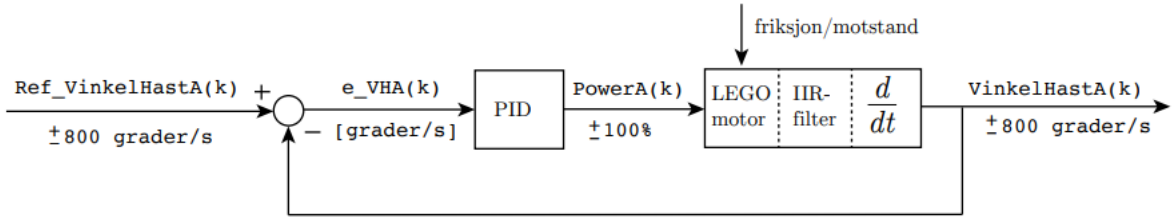


Figur 7.7: Resultatet fra forsøket hvor det ble gitt motorpådrag fra -100% til +100%. Forsøket viser sammenhengen mellom motorpådrag(`Power`) og beregnet vinkelhastighet. A) `Power` som funksjon av tid. B) IIR-filtert vinkelhastighet som funksjon av tid.

Som vi kan se fra avlesningene gir 100% i motorpådrag en vinkelhastighet på ca 857 [grader/s]. Men et fullt negativt pådrag gir ca -736 [grader/s]. Med tanke på at vi lager denne PID-regulatoren til prosjekter som skal hovedsaklig kjøre fremover velger vi å avrunde, og bestemmer at maksimal oppnåelig hastighet er +800 [grader/s](uten friksjon). Slik vil 100% motorpådrag gi +800 [grader/s].

$$PowerA = 100[\%] \longrightarrow VinkelHastA = 800[grader/s]$$

Slik kan vi ta utgangspunktet i blokkskjema-et for "cruise controlleren" **??** og presentere ett blokkskjema for turtallregulatoren vi ønsker å implementere.



Figur 7.8: Blokkskjema over turtallregulatoren hvor `Ref_VinkelHastA` representerer den ønskede hastigheten som kan være mellom ±800 grader/s. `e_VHA(k)` er avviket mellom ønsket og målt vinkelhastighet. Utifra avviket gis det et motorpådrag(`PowerA(k)`) mellom ±100%. `friksjon/motstand` er tegnet inn som en forstyrrelse på motorblokken. `VinkelHastA(k)` er den målte hastigheten. Legg merke til at dette er en lukket sløyfe.[9]

Vi vurderte også å forenkle bruken av turtallregulatoren ved å normalisere vinkelhastigheten mellom +-100%. Dette hadde gjort det betraktelig enklere dersom vi byttet mellom forskjellige benevninger, for eksempel mellom pådrag i [%] til [grader/s]. Men siden vi kun brukte roboten til automatisk kjøring med benevingen [grader/s], bestemte vi oss for å la være og normalisere vinkelhastigheten.

7.2.5 Kode for turtallregulator

I dette delkapittelet vil vi presentere koden for turtallregulatoren. Koden er basert på prinsippet til en standard industriell PID-regulator, som kan uttrykes matematisk sett i ligning 7.2

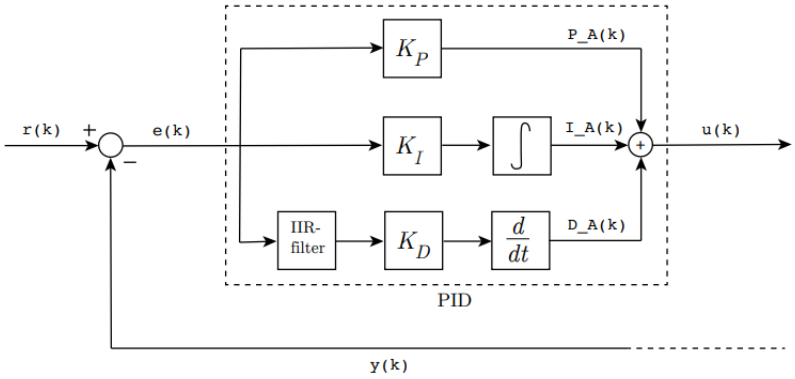
$$u(t) = u_0 + \underbrace{K_P \cdot e(t)}_{\text{P}} + \underbrace{K_I \int_0^t e(\tau)d\tau}_{\text{I}} + \underbrace{K_D \cdot \frac{d}{dt} e_f(t)}_{\text{D}} \tag{7.2}$$

[9]
hvor

- u(t) er det totale pådraget beregnet.
- For turtallregulatoren vil u_0 bli satt til 0. Dette grunnet til at u_0 er basispådraget.

- $K_P, K_I og K_D$ er regulatorparametre, de vil vi regne frem i neste delkapittel 7.2.6.
- e(t) er reguleringsavviket som er forskjell mellom ønskede og målte hastigheten.
- $e_f(t)$ er reguleringsavviket filtrert.
- Bokstavene P, I og D står for henholdsvis proporsjonaldel, integraldel og derivatdel.

[9] Videre kan vi presentere variablene og parametrene fra ligning 7.2 skjematisk som en PID-blokk i blokkskjema-et over turtallregulatoren fra figur 7.8. Blokkskjema-et over turtallregulatoren med en PID-blokk vises i figur 7.9.



Figur 7.9: Blokkskjema over turtallregulatoren hvor det er satt inn en PID-blokk basert på ligningen 7.2. *P_A(k)*, *I_A(k)* og *D_A(k)* tilsvarer bidragene P, I og D i ligning 7.2.

[9]

Viktige kodeutdrag for turtallregulatoren

Den LEGO-konstruksjonen som vi anvender i prosjektene våre benytter seg av to hjul, slik vil det derfor være en PID-regulator på hver av motorene til

hjulene. Dette fordi hjulene opplever ulik friksjon fra bakken, og vil derfor trenge ulike motorpådrag for å opprettеholde ønsket fart. I koden har vi valgt å skille motorene ved å gi variabelnavn med A og B til slutt. Her vil A være motoren som er koblet til port A på EV3'en, og det samme gjelder for motor B. Dette betyr at motorene vil ha egne verdier for *vinkelhastighet*, *referansehastighet* og *motorpådrag*. For å enklere forklare koden vil vise koden til motor A, legg merke til at koden er helt lik for motor B. Detaljer om koden er som følger:

- `HastighetRefA(k)` er den ønskede hastigheten. I koden vi har skrevet kan den settes til verdier på mellom -800 til 800[grader/s], siden det er de oppnåelige hastighetene som roboten kan oppnå. Reguleringsavviket `e_TurtallA(k)` er differansen mellom `HastighetRefA(k)` og målt hastighet(`VinkelHastighetA(k-1)`. I kodeutdraget 7.5 vises et eksempel hvor `referanse hastigheten` er satt til 88[grader/s] og hvor `e_TurtallA(k)` beregnes.

Kode 7.5: Kode for sette referanse hastigheten og beregne reguleringavviket

```
1       HastighetRefA(k) = 88;
2       e_TurtallA(k) = HastighetRefA(k) - VinkelHastighetA(k-1);
```

Legg merke til at `VinkelHastighetA(k-1)` er definert for forrige tidsskritt `k-1`. Dette er grunnet fordi `VinkelHastighetA` beregnes ved å derivere `Vinkelposisjonen` med henhold på `tid`, som baseres på forrige tidskritt. Vi gjør en antagelse om at beregnet vinkelhastighet i forrige tidsskritt `k-1` kan brukes til å beregne reguleringsavviket i nåværende tidsskritt `k`.

- `P_A(k)` Proporsjonaldelen forsterker reguleringsavviket `e_TurtallA(k)` ved å multiplisere med K_p parameteren som vi regner frem i neste kapitel 7.2.6. K_p defineres slik i koden:

Kode 7.6: Kode for beregne `P_A(k)` Proporsjonaldelen

```
1       P_A(k) = Kp*e_TurtallA(k);
```

Proporsjonaldelen vil være null dersom den ønskede hastigheten har blitt nådd, det kan vi se ved at reguleringsavviket også er `e_TurtallA(k)` = 0.

- `I_A(k)` integratordelen her blir produktet av reguleringsproduktet og integrasjonsforsterkningen K_i integrert, vist i kodeutdraget under.

Kode 7.7: Kode for beregne integratordelen **I_A(k)**

```
1       I_A(k) = ...
            EulerForward(I_A(k-1),Kia*e_TurtallA(k-1),Ts(k));
```

Legg merke til at vi bruker integrasjonsfunksjonen `EulerForward`. Dette gir regulatoren en verdi for hvordan regulerings avviket har vært totalt under hele prosessen. For å unngå anti-windup setter vi en grense på hva integrasjonsverdien kan være, vist i kodeutdraget under.

Kode 7.8: Kode for unngå anti-windup

```
1       if I_A(k) > umax
2           I_A(k) = I_A(k-1);
3       elseif I_A(k) < umin
4           I_A(k) = I_A(k-1);
5       end
```

Med en if-løkke forhindres anti-windup ved å ikke la integrasjons-verdien gå over/under en max/min verdi. Skulle den gå over/under max/min verdi bruker den verdien fra forrige tidsskritt.

- `D_A(k)` derivatdelen, i dette leddet deriverer vi det filtrerte avviket som gir regulatoren en verdi på hvor fort den er på vei mot eller vekk fra ønsket hastighet. Vi velger først å filtrere reguleringsavviket med et IIR-filter med en $\alpha - verdi$ på 0.1. Deretter deriverer vi med funksjonen `Derivation`, som vist i 7.2. Kodeutdraget for derivatdelen vises under.

Kode 7.9: Kode for derivatdelen

```
1       eA_Filtrert(k) = ...
            IIR_filter(eA_Filtrert(k-1),e_A(k-1),0.1);
2       D_A(k-1) = Derivation(Kd*eA_Filtrert(k-1:k),Ts(k));
```

Legg merke til derivasjonsforsterkningen K_d som er et parameter for å justere påvirkningen av D-leddet på PID-regulatoren. Denne regnes frem i neste delkapitel 7.2.6.

- Til slutt kan de samlede delpådragene `P_A, I_A og D_A` summeres sammen slik at det utgjør total pådraget til LEGO-roboten.

Kode 7.10: Kode for totalpådraget

```
1        PowerA(k) = P_A(k)+I_A(k) + D_A(k);
```

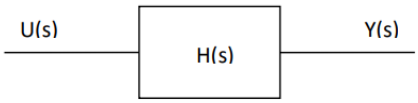
For å kunne optimalisere turtallregulatoren er det nødvendig og regne frem nøyaktige verdier for regulator parametrene $K_p, K_i og K_d$. Det gjør vi i neste delkapitel ved å bruke **Skogestads metode**.

7.2.6 Regulator parametre

I dette delkapitelet regner vi frem passende regulator parametre ved å bruke **Skogestads PID-tuning metode**. Dette er en metode hvor regulatorparametrene uttrykkes som funksjoner av prosessmodellens parametre[**?**]. Ved å finne **Transfer funksjonen** til kontroller systemet kan vi med **Skogestad's formuler** velge passende regulator parametre.

Transfer funksjon

En transfer funksjon er definert som forholdet mellom et systems output og input, vist i blokkdiagram 7.10. Benevningen vil være avhengig av benevningene til input og output[5].

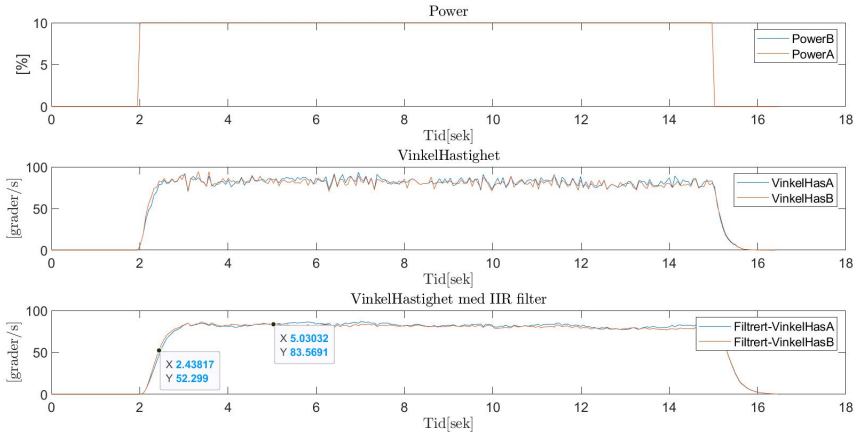


Figur 7.10: Blokkdiagram for transfer funksjon. Hvor U(s) er inngangsignalet, H(s) er transfer funksjonen, og Y(s) er utgangsignalet.

[5] Dette kan også uttrykes matematisk sett i ligningen under.

$$H(s) = \frac{Y(s)}{U(s)} \tag{7.3}$$

Vi valgte å finne transfer funksjonen til prosessen eksperimentelt. Det vil si at vi fant forholdet mellom motorpådraget(*inngangsignalet*) og vinkelhastigheten(*utgangsignalet*) ved å gjennomføre ett eksperiment hvor vi brukte koden7.1 som gir et motorpådrag på 10% (*inngangsignalet*), for at vi så leser av den filtrerte vinkelhastigheten(*utgangsignalet*). Vi gjennomførte eksperimentet ved å la LEGO-roboten kjøre på bakken. Dette var nødvendig siden vi ønsker å stille inn regulator parametrene for kjøring på bakken. Resultatet av eksperimentet er vist i figuren under



Figur 7.11: Resultat ved at LEGO-roboten får et motorpådrag på 10%(*inngangsignalet*) i 13 sekunder. Hvor den filtrerte vinkelhastigheten[grader/s](*utgangsignalet*) er avlest.

Utifra buen til grafen `VinkelHastighet med IIR-filter` antar vi at den er av typen førsteordens transfer function. Den generaliserte transfer funksjon av første orden kan uttrykes matematisk sett slik:

$$\frac{K}{T_s + 1} \tag{7.4}$$

Deretter kan vi fra skogestads's tabell finne formlene for $K_p, K_i og K_d$. Skogestad's tabell med formler kan ses i figuren under.

[?]

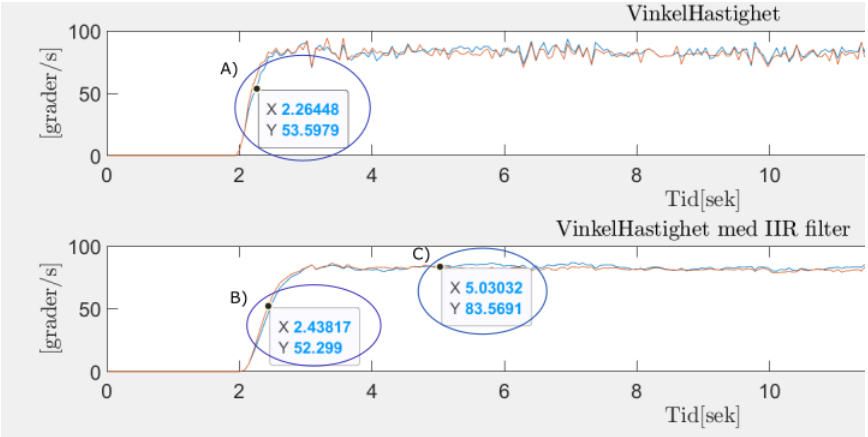
$H_p(s)$ (process)	K_p	T_i	T_d
$\frac{K}{s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$k_1\,(T_C+\tau)$	0
$\frac{K}{Ts+1}e^{-\tau s}$	$\frac{T}{K(T_C+\tau)}$	$\min\left[T,\ k_1\,(T_C+\tau)\right]$	0
$\frac{K}{(Ts+1)s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$k_1\,(T_C+\tau)$	T
$\frac{K}{(T_1s+1)(T_2s+1)}e^{-\tau s}$	$\frac{T_1}{K(T_C+\tau)}$	$\min\left[T_1,\ k_1\,(T_C+\tau)\right]$	T_2
$\frac{K}{s^2}e^{-\tau s}$	$\frac{1}{4K(T_C+\tau)^2}$	$4\,(T_C+\tau)$	$4\,(T_C+\tau)$

Figur 7.12: Skogestad's tuning tabell. Her vises de generaliserte transfer funksjonene, og formlene for regulator parametrene. A) Typen førsteorden transfer funksjon. B) Formel for å finne K_p for førsteorden. C) Formel for å finne K_i for førsteorden. D) Derivasjonsforsterkningen K_d i en førsteordens transfer funksjon er satt til null

For at vi skal kunne bruke formlene markert i ringer *B, C og D* fra Skogestad's tabell i figur 7.12, må vi først finne tidskonstantenT_s. Tidskonstanten, tiden det tar for systemets *steg respons* å nå 63,3 slutt verdi[5]. Fra figur 7.11 kan vi estimere at sluttverdien med en stegrespons på 10% motorpådrag er ca. 83[grader/s]. Videre kan vi regne frem tidskonstanten, som vist i ligningen under.

$$T = sluttverdi \cdot 0.63 \tag{7.5}$$

Vi vet da etter en tidskonstant så vil vinkelhastighet ha den verdien.



Figur 7.13: Resultat ved at LEGO-roboten får et motorpådrag på 10%(*inngangsignalet*) i 13 sekunder. Hvor vinkelhastigheten vises ufiltrert og filtrert. A)Avlest punkt som er 63% av sluttverdi på ufiltrert vinkelhastighet. B) Avlest punkt som er 63% av sluttverdi på IIR-filtrert vinkelhastighet. C) Sluttverdien til den IIR-filtrerte vinkelhastigheten ved et motorpådrag på 10%

Vi velger å bruke punkt A fra figur 7.13 når vi skal regne frem tidskonstanten. Dette punktet er 63% av sluttverdien til **ufiltrert** vinkelhastighet. Vi velger å bruke den ufiltrerte vinkelhastigheten siden filtreringen gjør funksjonen litt *tregere*. Slik blir tidspunktet til punktet 2.265sek, Slik blir tidskonstanten:

$$T = 2.265 - 2 = 0.265 \tag{7.6}$$

Med en tidskonstant T på 0.265 kan vi nå bestemme ønsket T_c. T_c avgjør hvor aggressiv reguleringen er. Vi definerte T_c ved å dele tidskonstanten T på 2,5. Ligningen under viser T_c.

$$T_c = \frac{0.265}{2.5} = 0.106 \tag{7.7}$$

Ved å ha funnet T og T_c og at $K = 0.265$ kan vi bruke formlene fra skogestad's tabell 7.12. Fra skogestad's tabell finner vi formlen for K_p markert med B og grønn ring. Vi setter inn verdiene for variablene, som vist i ligningen under.

$$Kp = \frac{0.265}{8.357 \cdot 0.106} = 0.299 \tag{7.8}$$

Videre kan vi avgjøre K_i, den finner vi fra tabellen 7.12 markert C med gul ring. Her velger vi minste verdien av Tidskonstanten T, og $T_c \cdot 1.44$. 1.44 er

en standard vi bruker, den kan gi raskere forstyrrelseserstatning. Med T på 0.265 blir da 0.152 den minste verdien. Vist i ligningen under.

$$K_i = 0.106 \cdot 1.44 = 0.152 \tag{7.9}$$

Slik bestemmer vi at $K_i = 0.152$. Parameteren for derivasjons delen er som sagt tidligere 0, utifra skogestad's tabell. Vi vil i neste kapitel vise resultatene ved bruk av disse parametrene i regulatoren ved å utføre de to forsøkene for turtalltestingen som beskrevet i testing uten regulator 7.2.3.

7.3 Resultater

I dette kapitelet vil vi ta for oss resultatene ved å bruke turtallregulatoren vi presenterte i forrige kapitel 7.2.5 når vi gjennomfører forsøkene for testing 7.2.3. Fra det første forsøket ønsker vi å se om regulatoren gjøre at LEGO-roboten holder en rett bane selvom hjulene opplever ulik friksjon. Fra den andre forsøket ønsker vi å se om regulatoren gjør at LEGO-roboten klarer å holde ønsket hastighet selvom den møter på økt motstand. Forsøkene ble utført med følgende parametre vi regnet frem i forrige kapitel 7.2.6: $K_p = 0.299$, $K_i = 0.152$ og $K_d = 0$.

Kode for å testing med turtallregulator

For å finne ut av hvordan LEGO-moterene håndterte friksjon fra bakken uten turtallsregulatoren valgte vi sist gang å la roboten kjøre fremover med samme pådrag på begge hjulene. Denne gangen vil vi istedenfor sette *referanse hastigheten* på begge motorene til å være den samme. Derfor endrer vi på koden for motorpådrag fra kodeutdrag 7.1, til kodeutdraget under.

Kode 7.11: Kode for testing med regulator

```
1  if Tid(k) ≤ 2 && k > 1
2          PowerB(k) = 0;
3          PowerA(k) = 0;
4      elseif Tid(k) ≤ 15 && Tid(k) > 2
5          PowerB(k) = PB(k) + IB(k) + DB(k-1);
6          PowerA(k) = PA(k) + IA(k) + DA(k-1);
```

```
7       else
8           PowerB(k) = 0;
9           PowerA(k) = 0;
10  end
```

Koden 7.11 vil i utgangspunktet gjøre det samme som koden for motorpådrag 7.1. Det vil si at LEGO-roboten kjører fremover i 13sekunder etter å først stått stille i 2sekunder. Hvor det tidligere ble gitt et motorpådrag på 10%, gir regulatoren nå pådraget gjennom koden `PowerA(k) = PA(k) + IA(k) + DA(k-1)`. Får at dette skal gå må vi angi en referanse hastigheten (`HastighetRefA(k)`). Vi bestemte å sette referanse hastigheten (`HastighetRefA(k)`) til å være 88[grader/s], dette fordi det tilsvarer omtrent 10% motorpådrag. Merk at vi har valgt å utføre prosjektene med en hastighet på rundt 80[grader/s], grunnen er at vi regnet frem PID-regulator parametrene med et steg på 10% motorpådrag. Slik vil parametrene fungere best rundt en hastighet rundt 80[grader/s], siden det er 10% av maksimal oppnåelig hastighet på 800[grader/s]. Koden for å sette `HastighetRefA(k)` er vist og forklart under.

Kode 7.12: Kode for å sette (*HastighetRefA(k)*)

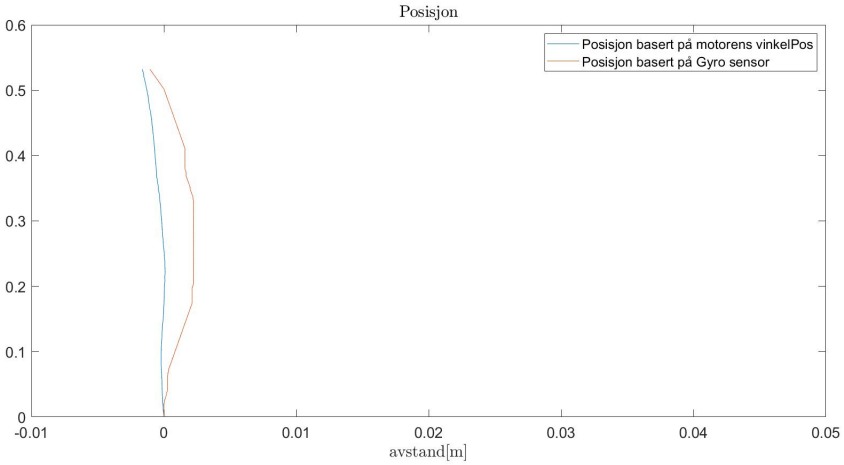
```
1  Tid(k) = toc;
2  if Tid(k) >2
3      HastighetRefA(k) = 88;
4      HastighetRefB(k) = 88;
5  else
6      HastighetRefA(k) = 0;
7      HastighetRefB(k) = 0;
8  end
```

Her settes `HastighetRefA(k)` til å være 88[grader/s] etter 2sekunder, dette fordi den først står stille. Det vil si at vi har satt ved start at `HastighetRefA(k)` til 0[grader/s]. Både kodeutdraget 8.117.11 og Kode 8.12 7.12 brukes til å kjøre roboten fremover i begge eksperimentene ved testing av regulatoren.

Eksperiment holde rett bane

Ved å kjøre koden 7.12 vil roboten kjøre fremover i 13sek med en referanse hastighet på 88[grader/s]. Vi ønsker å se om roboten klarer å holde banen sin bedre med regulatoren, enn uten. Dette fordi om hjulene oppleve for-

skjellig friksjon vil de forsatt holde lik hastighet og dermed kjøre "rettere". Resultatet fra forsøket vises i figuren under.

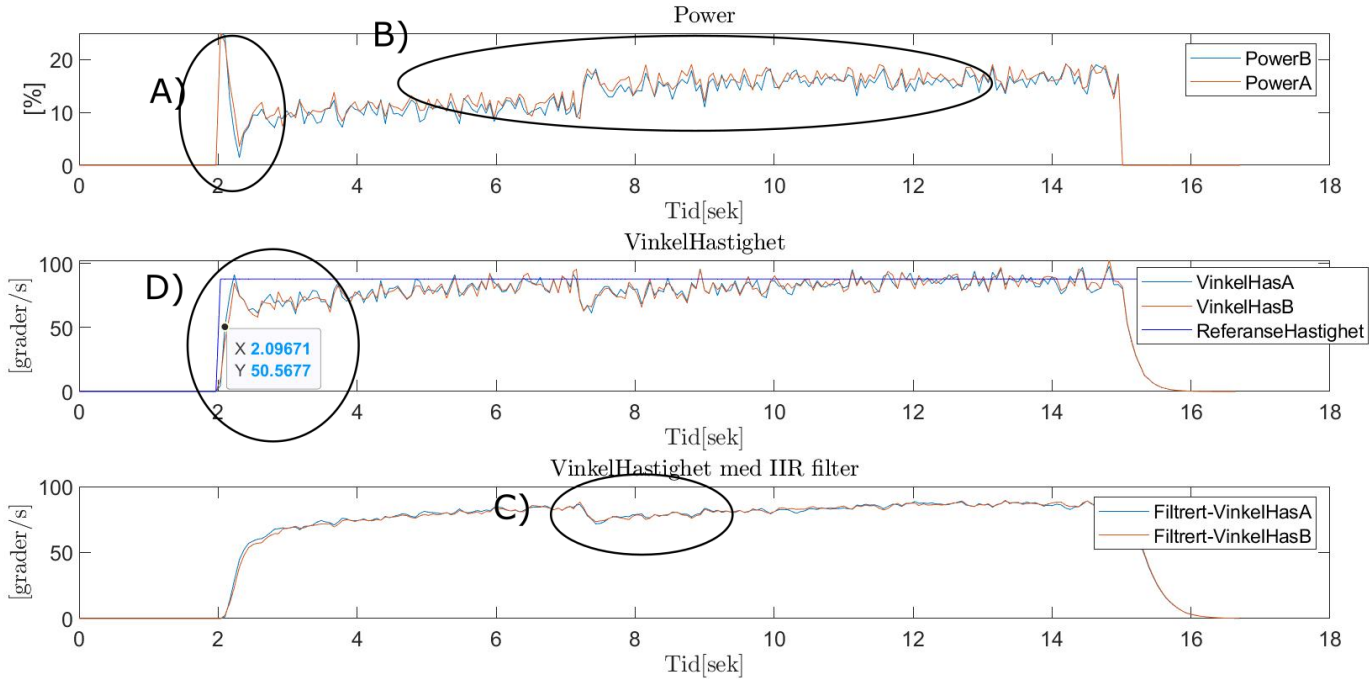


Figur 7.14: Resultat fra eksperiment holde bane med regulator

Fra figuren kan vi konkludere med at regulatoren fungerer slik som den skal. Dette kan vi si siden viss vi ser på figur 7.2.3 som var testing uten regulator ser vi at LEGO-roboten tar å kjører 5cm mot høyre. Men med regulatoren kjører den nærmest rett frem.

Eksperiment bevare hastighet ved møte med motstand

I dette eksperimentet kjører vi også koden 7.11 som vil kjøre roboten fremover i 13sek hvor den underveis møter på en kloss som gir økt motstand, som vist på bilde 7.3. Vi ønsker å se om regulatoren klarer å øke pådraget når roboten møter på klossen slik at roboten opprettholder hastigheten. Resultatet fra forsøket vises på bilde under.



Figur 7.15: Resultat fra eksperiment bevare hastighet ved møte med motstand med regulator. A) Hvor regulatoren begynner å gi pådrag. B) Punktet hvor roboten møter klossen og motorene får økt pådrag. C) Punktet roboten møter klossen hvor vinkelhastigheten går noe ned før det stabiliserer seg. D) Referansehastighet, og avlest koordinater ved 63% av sluttverdi.

Vi konkluderer med at eksperimentet fungerte bedre med regulatoren. Dette fordi vi ser at motorene får økt pådrag ved møte med klossen sammenlignet med forsøket uten hvor vinkelhastigheten gikk ned ved møte med klossen, se figur 7.11. Vi leser av verdien ved en tidskonstant, og ser at det stemmer overens med T_c

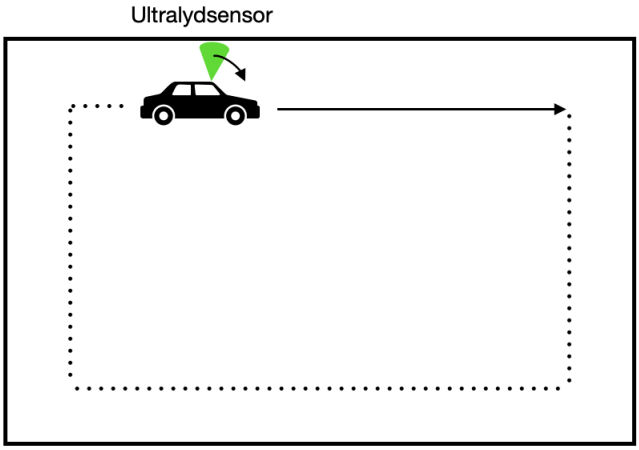
Kapittel 8

Robotstøvsuger

8.1 Problemstilling

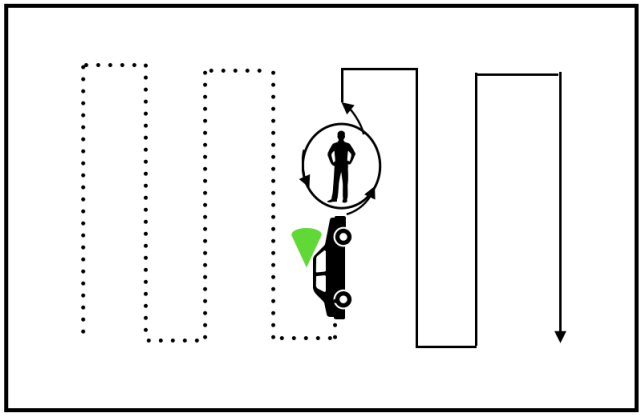
Formålet med dette prosjektet har vært å få roboten til å oppføre seg som en robotstøvsuger. Det vil si kjøre rundt i rommet samtidig som den kartlegger omgivelsene. Motivasjonen for dette prosjektet var å benytte opparbeidet kunnskap fra tidligere kapitler til å vise en mer avansert måte å bruke roboten på. I dette prosjektet har vi satt opp problemstillingen i to deler.

- Del en: Kjøre langs veggen til rommet slik at den får kartlagt utkanten av området den skal "støvsuge". Når den kommer tilbake til startspunktet skal den begynne på neste sekvens. Dette er forsøkt illustrert ved figuren nedenfor.



Figur 8.1: Illustrasjon av kartlegging av veggen. Bilen er ment å være roboten, og den grønne kjegleformede biten er ultralydsensoren

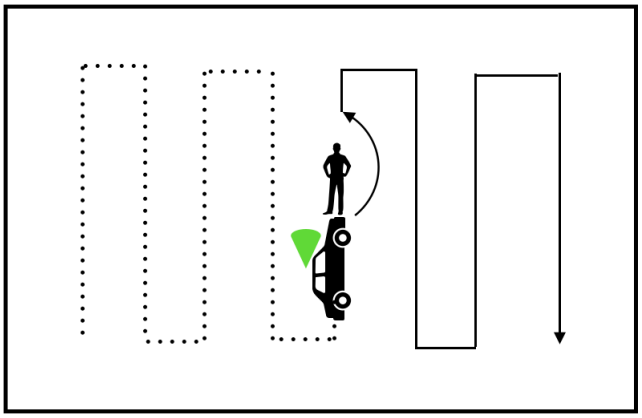
- Del to: Når den er ferdig å kartlegge utkanten av rommet skal den kjøre i siksakk frem og tilbake i rommet slik den får dekt hele arealet av rommet. Hvis den møter på et objekt skal den kjøre rundt objektet, samtidig som den kartlegger hvor objektet befinner seg i rommet (illustrert i figuren nedenfor)



Figur 8.2: Illustrasjon av støvsugingen og første møte med objekt. Mannen i figuren er i dette et tilfellet et objekt.

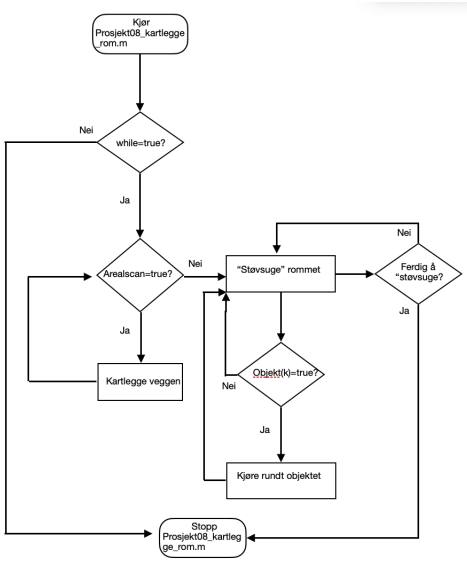
Hvis den allerede har kartlagt objektet den møter på, skal den bare kjøre

rundt til andre siden av objektet (se figur 8.3).



Figur 8.3: Illustrasjon av støvsugingen og møte med objekt som allerede er blitt detektert

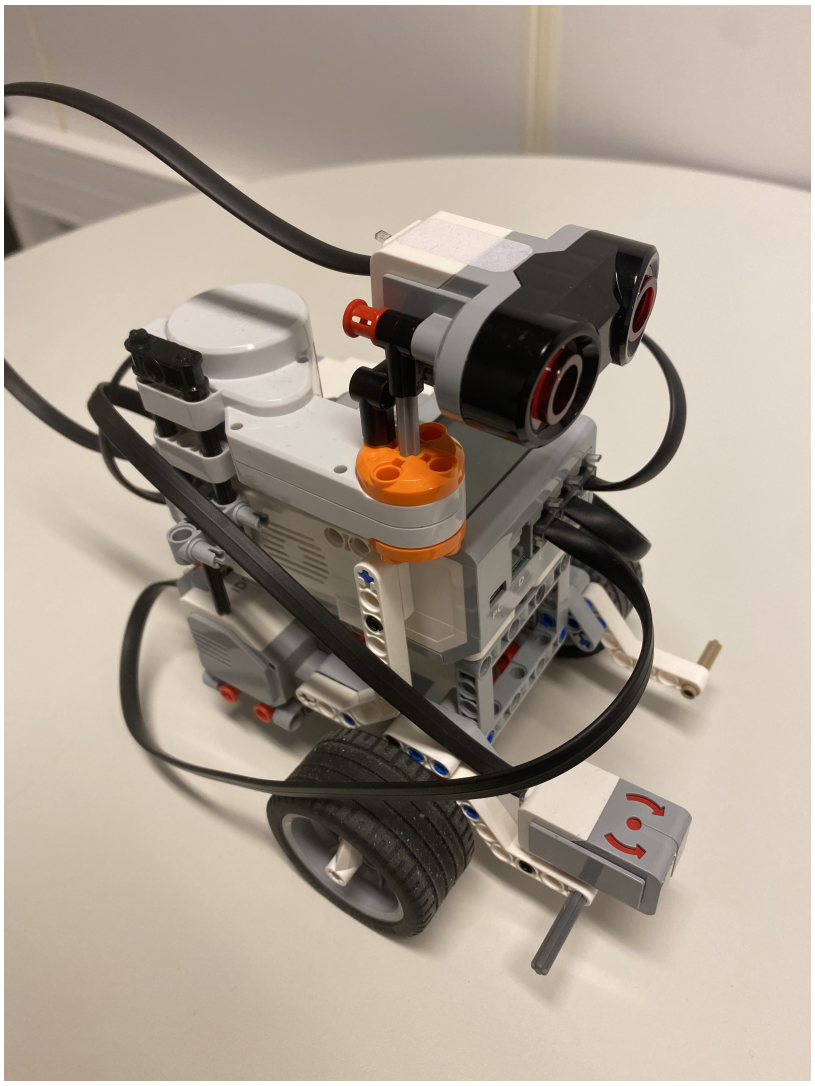
Dette er forsøkt illustrert i flytskjemaet nedenfor.



Figur 8.4: Kontrollflyten til prosjekt08_kartlegge_rom.m. 'While=true?'-blokka er true så lenge det ikke har kommet et brukertrykk på skyteknappen til styrestikken.

8.1.1 Legokosntruksjon

Hovedstrukturen av legokonstruksjonen er på samme måte som i tidligere kapitler. Det er derimot gjort små modifikasjoner på denne. Den største forskjellen er at det er montert en ekstra motor på toppen av roboten. På denne er det i tillegg montert en gyrosensor. Dette kan ses på figuren nedenfor.



Figur 8.5: Bilde av legokonstruksjonen

Sensoren og den ekstra motoren som er montert gjør det mulig for roboten å få avstandsmålinger fra et større område rundt seg. Fra kapittel 2 vet vi at ultralydsensoren har en liten deteksjonsvinkel på enkelte objekter. Ved å montere den på en motor, har vi muligheten til å rotere den slik at den får detektert et større område.

I dette prosjektet tar vi i bruk tre motorer. To til å få roboten til å bevege seg, og en for rotasjon av avstandsmåleren. Vi har tatt i bruk en ultralydsensor for avstandsmålinger, og en gyrosensor for vinkelmålinger.

Vi prøvde også å bruke to ultralydsensorer plassert vinkelrett på hverandre. Dette ga derimot ikke informasjon om et stort nok området, og den hadde problemer med å bevege seg rundt hjørner.

8.2 Kode brukt for kartlegging av rommet

Koden brukt i dette prosjektet er skrevet i en hovedfil i Matlab, i tillegg til tre egendefinerte funksjoner. Før vi går igjennom hovedfilen for koden vil vi ta for oss de tre funksjonene. Dette gjør det enklere å forstå seg på hovedfilen når vi kommer oss dit.

8.2.1 Funksjonen scan

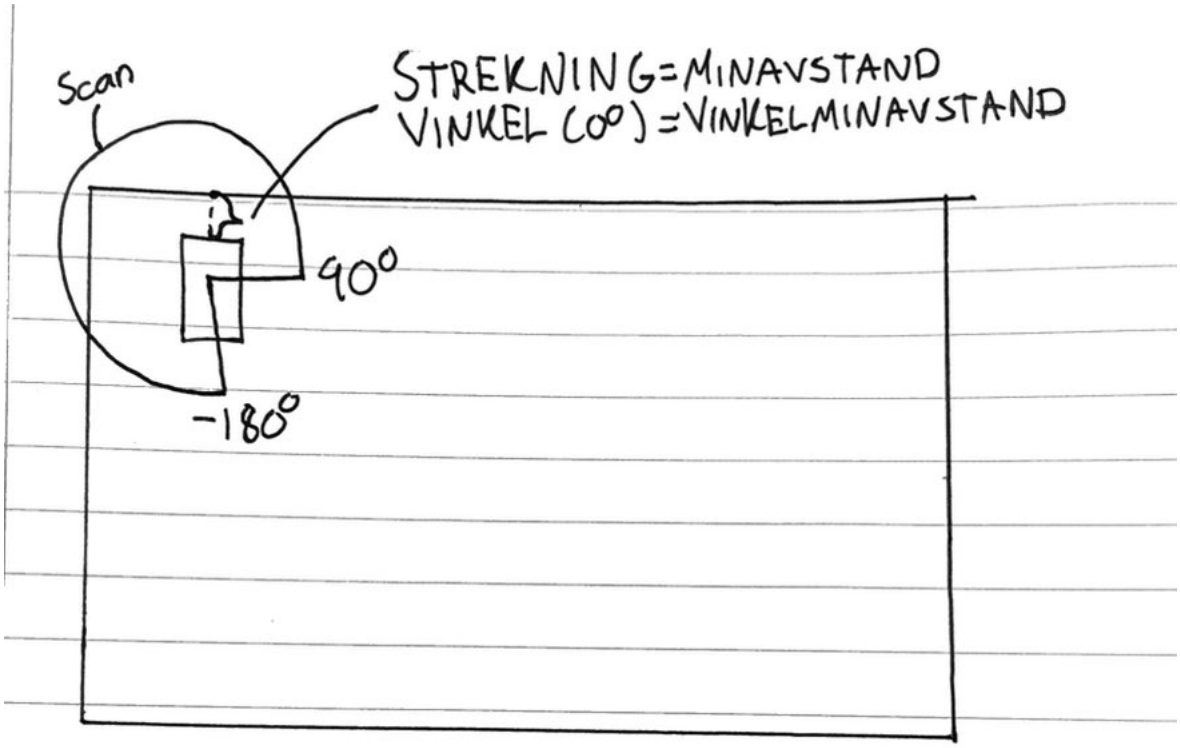
Den første funksjonen vi tar for oss er funksjonen vi har kalt for `scan`. Denne har som formål å ta en scan av omgivelsene slik at vi får informasjon om hvor det eventuelt befinner seg et objekt eller en vegg. Dette gjøres ved å rotere ultralydsensoren montert oppå roboten i en vinkel på totalt 270 grader. Hvordan funksjonen er definert kan ses i kodeutdrag 8.1.

Kode 8.1: Definering av funksjonen **scan**

```
1 function [minavstand, vinkelmaksavstand] = scan(motorA, ...
      motorB, motorC, mySonicSensor)
```

`Minavstand` er den minste avstanden detektert av ultralydsensoren under

scanningen. Mens `vinkelminavstand` er vinkelen som denne avstanden ble detektert på. Dette er vist i figur 8.6 nedenfor.



Figur 8.6: Når funksjonen `scan` kalles, roteres ultralydsensoren fra 90 grader til høyre for roboten, til en vinkel på 180 grader til venstre for roboten. Hvis det hadde vært en situasjon som illustrert ovenfor ville `vinkelminavstand` fått verdien 0, mens `minavstand` ville blitt avstanden fra roboten til veggen rett foran seg. Vi har definert positiv vinkel til å være med klokken.

Hvordan koden for funksjonen ser ut kan ses i kodeutdrag 8.2. På linje 2-7 i kodeutdraget initialiseres det en tellevariabel `i`, farten til `motorB` og `motorC` (motorene koblet til hjulene) settes til å være lik null, samtidig som `motorA.speed` settes til verdien 20 (får ultralydsensoren til å rotere).

På linje 8 startes en while-loop som kjøres så lenge variabelen `scaner` på linje 5 er satt til `true`. Linje 9 leser avstanden til ultralydsensoren, og lagrer den i variabelen `avstand2`.

På linje 10 defineres `VinkelPosMotorA` til å være vinkelposisjonen til `motorA`. Det er denne som blir brukt som et mål på vinkelen til avstandene detektert i `avstand2`.

Linje 31 sørger for at tellevariabelen `i` inkrementeres med 1 for hver iterasjon av while-loopen. Når vinkelposisjonen til ultralydsensoren blir større eller lik 95 grader, vil if-statementen på linje 11 bli `true`. Dette setter variabelen `scan2` til å være `true` på linje 12. While-løkken på linje 13 vil kjøre helt til denne blir satt til å være `false`.

Etter while-løkken blir `motorA` satt til å bevege seg motsatt retning (linje 14). Deretter blir `avstand2` og `VinkelPosMotorA` oppdatert. Motoren vil fortsette å rotere ultralydsensoren helt til vinkelen blir mindre eller lik -180 grader. Variablene `scan1` og `scan2` vil da settes til å være false på linje 18 og 19. På denne måten vil while-løkkene på linje 8 og 13 avsluttes.

På linje 20 lagres den minste avstanden detektert mens koden har blitt kjørt. Denne lagres i variabelen `minavstand`, sammen med indeksen til denne verdien. På linje 21 blir vinkelen assosiert med den minste avstanden lagret i variabelen `vinkelminavstand`. While-løkken på linje 22-24 sørger for at `motorA` stilles tilbake til startsposisjonen. Når dette er gjort, stoppes motoren på linje 25. Funksjonen er nå ferdig, og de to variablene `minavstand` og `vinkelminavstand` sendes til workspace.

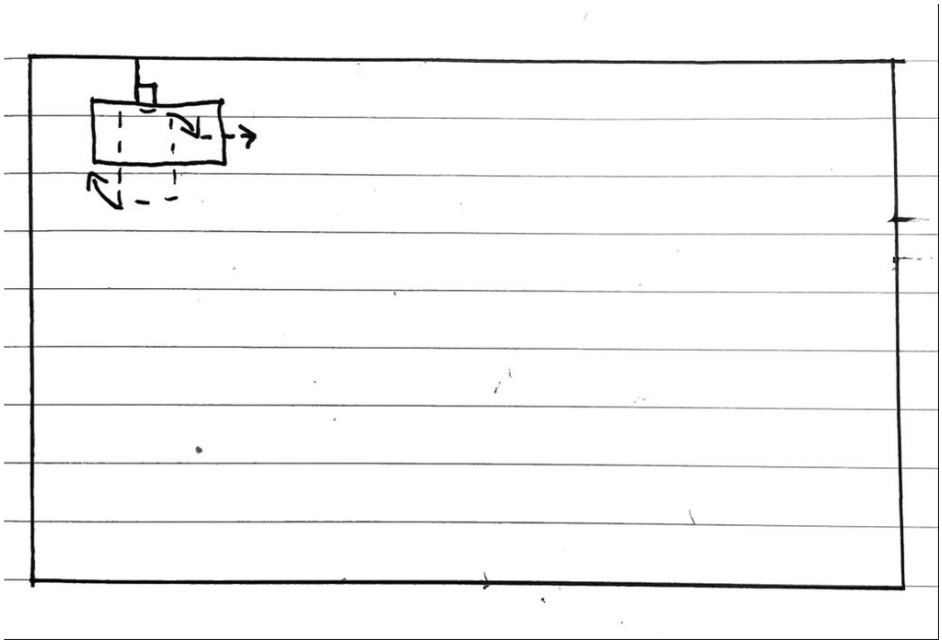
Kode 8.2: Funksjonen scan

```
2  i = 1;
3  motorB.Speed = 0;   %Stopper motorene som beveger roboten
4  motorC.Speed = 0;
5  scaner = true;
6  motorA.Speed = 20;  %Setter farten til "scanneren" som er ...
       drevet av motorA
7  start(motorA);
8  while scaner
9      avstand2(i) = double(readDistance(mySonicSensor));
10     VinkelPosMotorA(i) = double(motorA.readRotation);   ...
           %Leser avstanden hved hver iterasjon av i, ...
           samtidig som hvilken vinkel motoren st?r i
11     if VinkelPosMotorA(i) ≥ 95   %N?r motoren har rotert ...
           95 grader
12         scan2 = true;
13         while scan2
14             motorA.Speed = -20;  %Begynner ? scanne andre ...
```

```
                veien
15              avstand2(i) = double(readDistance(mySonicSensor));
16              VinkelPosMotorA(i) = double(motorA.readRotation);
17              if VinkelPosMotorA(i) ≤ -180
18                  scan2 = false;
19                  scaner = false;
20                  [minavstand, index] = min(avstand2);  ...
                        %Lagrer minste registrerte avstand ...
                        m?lt p? hele scannen
21                  vinkelmaksavstand = ...
                        VinkelPosMotorA(index);   %Hvilken ...
                        vinkel denne avstanden tilsvarer
22                  while double(motorA.readRotation) ≤ 0
23                      motorA.Speed = 20;
24                  end
25                  stop(motorA)
26              else
27                  i = i+1;
28              end
29          end
30      else
31          i = i+1;
32      end
33  end
```

8.2.2 Funksjonen kjoer_gyro

En annen funksjon som brukes mye i koden er funksjonen `kjoer_gyro`. Denne funksjonen brukes når roboten møter på et objekt eller en vegg. Den får da roboten til å stille seg parallelt med objektet, for så å kjøre en kort distanse fremover. Hvis roboten er i en situasjon som i figur 8.6, blir den rotert som i figuren nedenfor.



Figur 8.7: Illustrasjon av bevegelsen til roboten ved kjøring av funksjonen `kjoer_gyro`. Roboten vil først roteres slik at den er parallelt med veggen, før den kjører en kort distanse fremover.

På linje 1 i kodeutdrag 8.3 defineres funksjonen `kjoer_gyro`. `x` og `y` er koordinatene til roboten når funksjonen blir kalt. `Vinkelminavstand` er vinkelen til den minste avstanden registrert av funksjonen `scan` beskrevet i seksjonen ovenfor. `Retning` er enten satt til `1` eller `2` avhengig av om roboten skal ha veggen på venstre- eller høyresiden av seg. Hvis retningen er satt til `1` skal veggen være på venstresiden av roboten, mens `2` betyr at den skal være på høyresiden.

Kode 8.3: Funksjonen kjoer_gyro

```
1 function [ny_x, ny_y, beregnet_posisjon3] = ...
      kjoer_gyro(motorB, motorC,  x, y, vinkelminavstand, ...
      myGyroSensor, retning)
2 j = 1;
3 VinkelPosMotorvenstre(1) = double(motorB.readRotation);
4 VinkelPosMotorhoyre(1) = double(motorC.readRotation);
5 startsvinkel = double(readRotationAngle(myGyroSensor));
6 kjoer = true;
```

```
7  Δ_x = [0];
8  Δ_y = [0];
9  beregnet_posisjon2 = [0];
```

Funksjonen går så inn i en while-løkke på linje 10 i kodeutdrag 8.4. Inne i while-løkken er det lagt inn fire if-statements (ser bare den første av disse). Dette er for å skille mellom hvilken vei den skal rotere avhengig av `vinkelminavstand` (minste vinkel registrert), og `retning` (om veggen/-objektet skal være på høyre eller venstre side av roboten). På linje 11 i kodeutdraget kjøres den første if-statementen. Denne kjøres hvis det er et tilfelle der den har registrert et objekt, og at det er i et område mellom vinkelrett til venstre for seg (-90 grader) og vinkelrett til høyre for seg. Siden retningen er satt til `1`, skal roboten også ha objektet på venstresiden av seg. For å få dette til på minst mulig rotasjon vil linje 13 og 14 i kodeutdraget sørge for at roboten roteres til høyre.

På linje 15-24 gjøres beregninger for å finne roboten sin nye posisjon og vinkel. Her er `vinkelforskjell` den totale vinkelendringen til roboten. Når denne er blitt mer enn 90 grader mer enn `vinkelminavstand`, vet vi at objektet/veggen er parallelt med roboten. Dette er lagt inn som en if-statement på linje 24 i 8.4. Siden `vinkelminavstand` vil være -90 når objektet er vinkelrett til venstre for roboten, og 0 når objektet er rett foran, er det grunnen til at `vinkelminavstand` er addert med `90` på if-statementen på linje 24.

Kode 8.4: Kodeutdrag av funksjonen **kjoer_gyro**

```
10 while kjoer == true
11     if vinkelminavstand > -90 && retning == 1
12         j = j+1;
13         motorB.Speed = 10;
14         motorC.Speed = -10;
15         VinkelPosMotorvenstre(j) = ...
               double(motorB.readRotation);
16         VinkelPosMotorhoyre(j) = double(motorC.readRotation);
17         Δ_vinkel_b = VinkelPosMotorvenstre(j) - ...
               VinkelPosMotorvenstre(j-1);
18         Δ_vinkel_c = VinkelPosMotorhoyre(j) - ...
               VinkelPosMotorhoyre(j-1);
19         beregnet_posisjon2(j) = ((Δ_vinkel_b + Δ...
               _vinkel_c)*0.17)/(720);
20         angle = double(readRotationAngle(myGyroSensor));
21         Δ_x(j) = Δ_x(j-1) + ...
```

```
            (beregnet_posisjon2(j)*sin(pi*(angle)/180));
22          Δ_y(j) = Δ_y(j-1) + ...
            (beregnet_posisjon2(j)*cos(pi*(angle)/180));
23          vinkelforskjell(j) = angle - startsvinkel;
24          if vinkelforskjell(j) ≥ vinkelminavstand + 90
25              VinkelPosMotorB2 = double(motorB.readRotation);
26              VinkelPosMotorC2 = double(motorC.readRotation);
```

Etter roboten har rotert til riktig vinkel, skal den kjøre en kort distanse rett frem. Dette blir gjort i linje 27-40 i kodeutdrag 8.5. While-løkken sørger for at roboten kjører rett frem helt til vinkelposisjonen til hjulene har endret seg med 240 grader. Endringen i posisjonen til roboten regnes ut på samme måte som tidligere på linje 39 og 40.

Etter alt dette er gjort blir motorpådragene satt til null, relevante variabler lagret, og sendt tilbake til workspace. Koden for dette har vi valgt å ikke ta med, da det er ganske rett frem hvordan dette gjøres.

Kode 8.5: Kodeutdrag av funksjonen **kjoer_gyro**

```
27              l = 1;
28              while double(motorB.readRotation) ≤ ...
                    VinkelPosMotorB2 + 240 && ...
                    double(motorC.readRotation) ≤ ...
                    VinkelPosMotorC2 + 240
29                  VinkelPosMotorhoyre(j+l) = ...
                        double(motorC.readRotation);
30                  VinkelPosMotorvenstre(j+l) = ...
                        double(motorB.readRotation);
31                  Δ_vinkel_b = VinkelPosMotorvenstre(j+l) - ...
                        VinkelPosMotorvenstre(j+l-1);
32                  Δ_vinkel_c = VinkelPosMotorhoyre(j+l) - ...
                        VinkelPosMotorhoyre(j+l-1);
33                  beregnet_posisjon2(j+l) = ((Δ_vinkel_b + Δ...
                        _vinkel_c)*0.17)/(720);
34                  motorB.Speed = 20;
35                  motorC.Speed = 20;
36                  start(motorB)
37                  start(motorC)
38                  angle = ...
                        double(readRotationAngle(myGyroSensor));
39                  Δ_x(j+l) = Δ_x(j+l-1) + ...
                        (beregnet_posisjon2(j+l)*sin(pi*(angle)/180));
40                  Δ_y(j+l) = Δ_y(j+l-1) + ...
                        (beregnet_posisjon2(j+l)*cos(pi*(angle)/180));
```

Videre i koden er det tre andre, relativt like if-statements som i kodeutdrag 8.4, basert på `vinkelminavstand` og `retning`. Disse er veldig lik det som er gjennomgått, bare med noen små endringer.

8.2.3 Funksjonen snu_til_vinkel

Den siste funksjonen vi lagde i dette prosjektet heter `snu_til_vinkel`. Denne funksjonen sin oppgave er å få roboten til å snu seg i en bestemt vinkel i forhold til startsposisjonen til roboten. I kodeutdrag 8.6 vises koden som gjorde dette mulig. På linje 1 i kodeutdraget defineres funksjonen samt inngangsverdiene til funksjonen. Disse er `vinkel`, som er den ønskede vinkelposisjonen til roboten, `motorB` og `motorC`, som er motorene til roboten, `GyroAngle`, som er vinkelen til roboten når funksjonen blir kalt, og `myGyroSensor`, som har alle funksjonene til gyrosensoren som er koblet til. Så hvis man ønsker at roboten skal snu seg 180 grader fra startsretningen, blir funksjonskallet slikt:

`snu_til_vinkel(180, motorB, motorC, GyroAngle(k), myGyroSensor)`.

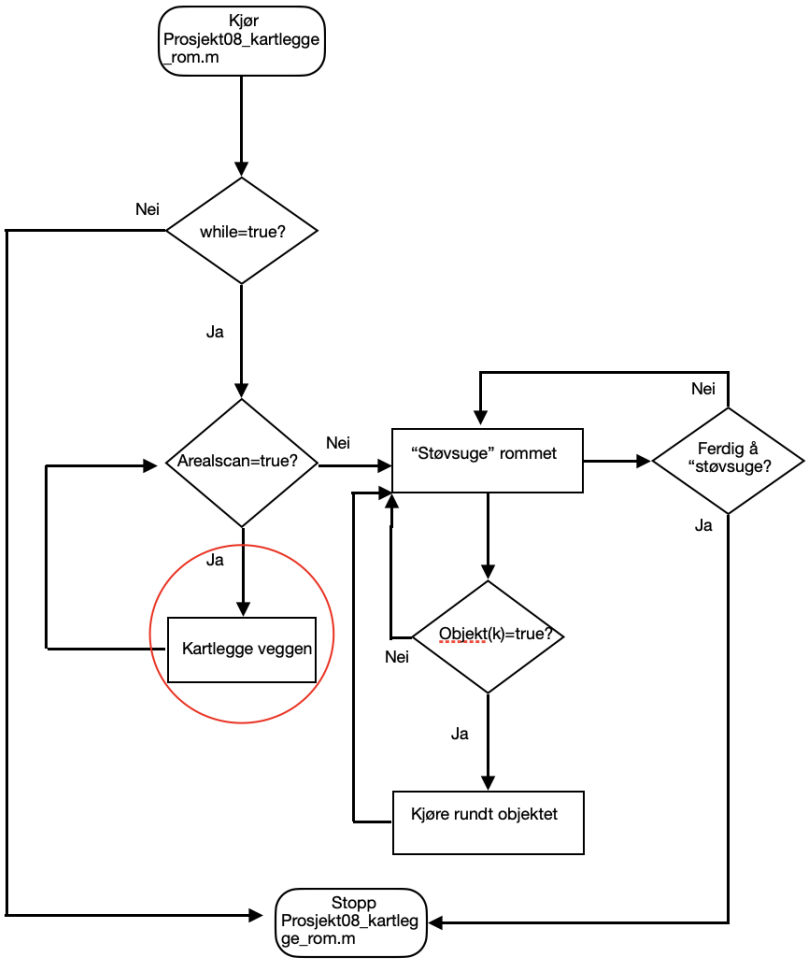
På linje 3 i kodeutdrag 8.6 blir det lagd en variabel (`rest`), som tilegnes den ønskede vinkelen sin verdi i første omdreining. Dette er siden vinkelen fra `GyroAngle` kan ha verdier utenfor området mellom 0 og 360. Linje 4 regner ut differansen mellom den ønskede vinkelen og den nåværende vinkelen, og lagrer den i variabelen `differanse`.

På linje 5 - 39 kjøres det en while-loop som kjøres så lenge variabelen `kjoer` er true. I while-loopen kjøres det en if-setning på linje 6 - 38. Linje 6, 14, 22 og 30 viser fire "if-statements" som bestemmer hvilken retning roboten skal kjøre, avhengig av differansen mellom ønsket vinkel og nåværende vinkel. Metoden for å få den til å svinge til ønsket vinkel er relativt lik ved de fire forskjellige tilfellene. På linje 7 og 8 settes motorene, `motorB` og `motorC`, til å ha `10` og `-10` i hastighet. Dette vil få roboten til å rotere mot høyre. På linje 9 kjøres en ny if-statement som sjekker om robotens nåværende vinkel er lik ønsket vinkel. Hvis dett er tilfellet settes motorpådraget til motorene til null, og funksjonen avsluttes.

Kode 8.6: Funksjonen snu_til_vinkel

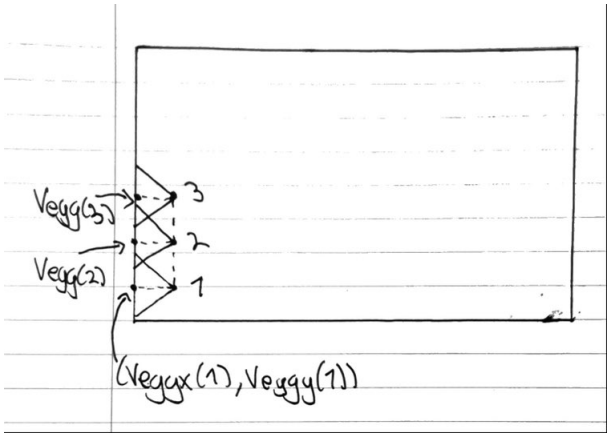
```
function snu_til_vinkel(vinkel, motorB, motorC, GyroAngle, ...
    myGyroSensor)
kjoer = true;
rest = mod(GyroAngle, 360);
differanse = vinkel - rest;
while kjoer
    if (180 > differanse) && (differanse ≥ 0)
        motorB.Speed = 10;
        motorC.Speed = -10;
        if  double(readRotationAngle(myGyroSensor)) - ...
            GyroAngle ≥ differanse
            motorB.Speed = 0;
            motorC.Speed = 0;
            kjoer = false;
        end
    elseif (180 ≤ differanse) && (differanse ≤ 360)
        motorB.Speed = -10;
        motorC.Speed = 10;
        if  double(readRotationAngle(myGyroSensor)) - ...
            GyroAngle ≤ 360 - differanse
            motorB.Speed = 0;
            motorC.Speed = 0;
            kjoer = false;
        end
    elseif (-180 > differanse) && (differanse ≥ -360)
        motorB.Speed = 10;
        motorC.Speed = -10;
        if  double(readRotationAngle(myGyroSensor)) - ...
            GyroAngle ≥ differanse + 360
            motorB.Speed = 0;
            motorC.Speed = 0;
            kjoer = false;
        end
    elseif (-180 ≤ differanse) && (differanse < 0)
        motorB.Speed = -10;
        motorC.Speed = 10;
        if  double(readRotationAngle(myGyroSensor)) - ...
            GyroAngle ≤  differanse
            motorB.Speed = 0;
            motorC.Speed = 0;
            kjoer = false;
        end
    end
end
```

8.2.4 Del en: kartlegging av veggen



Figur 8.8: Nåværende posisjon i flytskjema

Første del av prosjektet handler om å få kartlagt veggen. Det vil si at roboten skal kjøre langs veggen, mens den foretar målinger av veggen. Dette skal den gjøre helt til den har kommet tilbake til hvor den startet kartleggingsrunden. Hvordan dette er løst er illustrert i figuren nedenfor.



Figur 8.9: Punktene 1, 2 og 3 er robotens posisjon etter første, andre og tredje iterasjon av koden. Mellom hvert punkt foretas det også en scan som gir oss informasjon om hvor veggen befinner seg i forhold til robotens posisjon. Det gjør at vi får ett nytt koordinat for veggen for hvert av de tre punktene. `Veggx(1)` og `Veggy(1)` er de første x- og y-koordinatene til veggen.

Hvis roboten etter en stund havner på samme koordinater som den startet på, vet vi at den har gjort seg ferdig med kartleggingsrunden. Det samme kan sies om koordinatene til veggen. Når hovedfilen kjøres for første gang initialiseres det en del ulike variabler. Noen av disse kan ses i kodeutdraget nedenfor (8.7). I koden definerer vi variablene `Objekt`, `Arealscan` og `Vegg` til å være henholdsvis `false`, `true` og `true`. Variabelen `Objekt` er en variabel som sier oss om roboten har møtt på et objekt eller ikke. `Arealscan` brukes til å ha en oversikt over når roboten er ferdig med kartleggingsrunden, mens `Vegg` er en variabel som settes lik true når roboten møter på en vegg. Siden roboten alltid starter med å scanne veggen er denne satt til å være true ved første iterasjon av koden.

Kode 8.7: Definering av initialverdier i `Prosjekt08_kartlegge_rom`

```
138             Objekt(1) = false;
139             Arealscan = true;
140             Vegg = true;
```

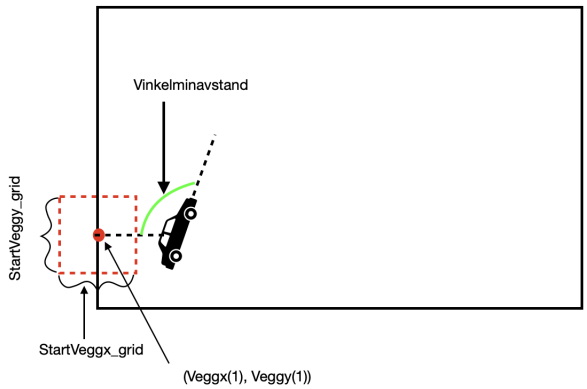
Etter dette tar roboten sin første scan av omgivelsene ved bruk av funksjonen `scan` (linje 141 i kodeutdrag 8.8. Deretter hentes ut de første koor-

dinatene til veggen på linje 142-143 ved bruk av sinus og cosinus (`x(1)` og `y(1)` er startskoordinatene til roboten, og settes lik null tidligere i koden). `StartVeggx_grid` og `StartVeggy_grid` defineres så til å være et område rundt startskoordinatene til veggen. Dette skal senere brukes til å verifisere når roboten har kommet tilbake til startsposisjonen sin igjen.

Kode 8.8: Definering av initialverdier i `Prosjekt08_kartlegge_rom`

```
141             [Minavstand(1), Vinkelminavstand(1)] = ...
                    scan(motorA, motorB, motorC, ...
                    mySonicSensor); %Kjoerer en scan av ...
                    omgivelsene for aa faa foerste punkt til ...
                    veggen
142             Veggx(1) = x(1) + ...
                    Minavstand(1)*sin((pi*Vinkelminavstand(1))/180);
143             Veggy(1) = y(1) + ...
                    Minavstand(1)*cos((pi*Vinkelminavstand(1))/180);
144             StartVeggx_grid = [Veggx(1) - 0.15, Veggx(1) + ...
                    0.15];
145             StartVeggy_grid = [Veggy(1) - 0.15, Veggy(1) + ...
                    0.15];
```

I figuren nedenfor er det forsøkt å illustrere de ulike variablene.



Figur 8.10: Figuren viser en illustrasjon av de ulike variablene i kodeutdrag 8.8. Den sorte bilen er ment å være robotstøvsugeren. Her vises det også hvordan `Vinkelminavstand` er vinkelen fra robotens kjøreretning og den minste detekterte avstanden til veggen.

Neste steg i kartleggingen blir å ta nye målinger av veggen helt til den har kjørt rundt hele veggen, og endt opp på startspunktet igjen. Koden brukt for å oppnå dette kan ses i kodeutdrag 8.10. Funksjonen `scan` og `kjoer_gyro` brukes i linje 249 og 250 til å ta målinger av veggen, stille seg opp parallelt med veggen, og kjøre en kort distanse rett fremover. På linje 251-256 oppdateres en del variabler basert på målingene og endringen av posisjonen til roboten. Dette gjøres for hver iterasjon av koden, helt til `Arealscan` settes til å være `false`

Kode 8.9: Kode for kjøring langs vegg samt kartlegging.

```
248             if Arealscan == true %Sjekker om den er p? ...
                    runden hvor den skal kartlegge veggen
249                 [Minavstand(k), Vinkelminavstand(k)] = ...
                        scan(motorA, motorB, motorC, ...
                        mySonicSensor); %Sjekker den minste ...
                        avstanden rundt roboten, og den ...
                        tilh?rende vinkelen
250                 [ny_x, ny_y, beregnet_posisjon2(k)] = ...
                        kjoer_gyro(motorB, motorC, x(k-1), ...
                        y(k-1), Vinkelminavstand(k), ...
                        myGyroSensor, 1); %Stiller seg opp ...
                        paralelt med veggen og kj?rer fremover.
251                 x(k) = ny_x;
252                 y(k) = ny_y;
253                 VinkelPosMotorB(k) = ...
                        double(motorB.readRotation);
254                 VinkelPosMotorC(k) = ...
                        double(motorC.readRotation);
255                 Veggx(k) = x(k-1) + ...
                        Minavstand(k)*sin((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180);
256                 Veggy(k) = y(k-1) + ...
                        Minavstand(k)*cos((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180);
```

For å sjekke om roboten er ferdig med kartleggingen av veggen sammenligner vi koordinatene til roboten og veggen (på nåværende tidspunkt) med startskoordinatene. Koden vi har brukt for dette er vist i kodeutdrag 8.10. På linje 258 i kodeutdraget har vi sagt at `k` må ha en større verdi enn 10 for at resten av koden skal bli kjørt. Årsaken til dette er at roboten skal kjøre en hvis distanse før den sjekker om den har kommet rundt veggen. If-setningene på linje 259-260 sjekker om den nåværende posisjonen til roboten er innenfor et område på $10cm^2$ rundt startskoordinatene til roboten (0,0). På linje 266-267 sjekker vi på samme måte om de nåværende ko-

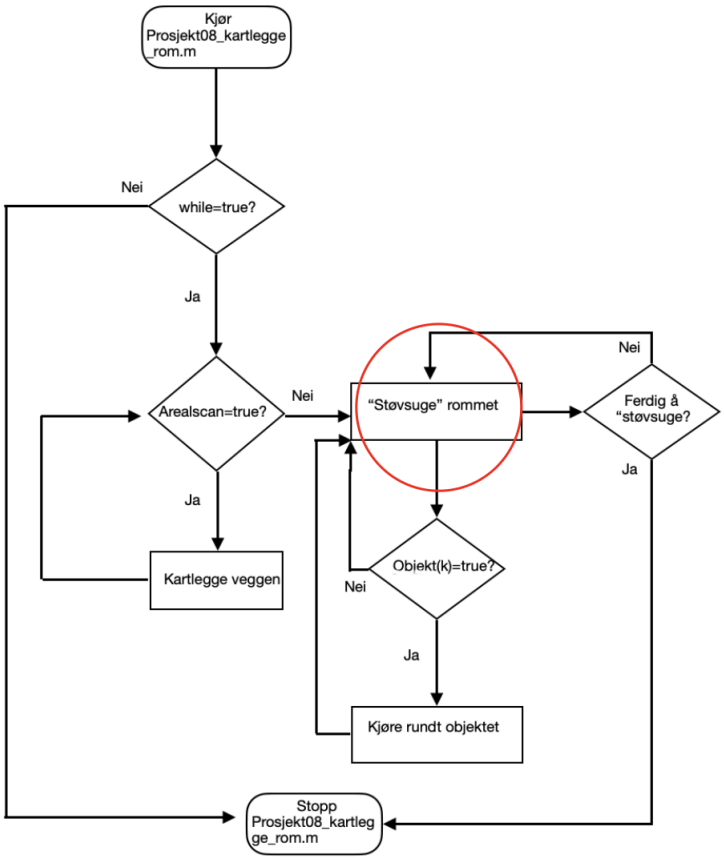
ordinatene til veggen er innenfor et område gitt av `StartVeggx_grid` og `StartVeggy_grid` som vi definerte tidligere i kapittelet. Noe som er verdt å merke seg er variabelen `Strekning`, som settes til `1` på linje 262 og 269. Dette er en variabel som vi bruker til å holde styr på hvilken retning roboten har i forhold til startsposisjonen.

Kode 8.10: Kode for å sjekke om kartleggingen av veggen er ferdig

```
258                 if k > 10
259                     if x(k) > -0.05 && x(k) < 0.05
260                         if y(k) > -0.05 && y(k) < 0.05 % ...
                                Sjekker om roboten har kommet ...
                                til startsposisjonen igjen
261                             Arealscan = false;
262                             Strekning = 1;
263                             Veggx(k+1) = Veggx(1);
264                             Veggy(k+1) = Veggy(1);
265                         end
266                     elseif Veggx(k) > StartVeggx_grid(1) ...
                            && Veggx(k) < StartVeggx_grid(2)
267                         if Veggy(k) > StartVeggy_grid(1) ...
                                && Veggy(k) < ...
                                StartVeggy_grid(2) % Sjekker ...
                                ogs? om veggen som den scanner ...
                                har samme koordinater som den ...
                                f?rste scannen av veggen
268                             Arealscan = false;
269                             Strekning = 1;
270                             Veggx(k+1) = Veggx(1);
271                             Veggy(k+1) = Veggy(1);
272                         end
273                     end
274                 end
```

8.2.5 Del 2: Støvsuging av rommet

Når roboten er ferdig med kartleggingen av veggen går den videre til å støvsuge rommet. Nåværende posisjon i flytskjema er vist i figuren nedenfor.



Figur 8.11: Nåværende posisjon i flytskjema markert med en rød ring

Det neste som skjer i koden er vist i kodeutdrag 8.11. Her sjekker vi først om den nettopp er ferdig med kartleggingsfasen med en if-statement på linje 315. Som nevnt tidligere er `Strekning` en variabel for å holde orden på hvor i prosessen roboten befinner seg. På linje 290 blir denne variablen inkrementert med en, og koden i kodeutdraget vil dermed bare kjøres én gang.

På linje 279 og 280 settes motorpådraget til null for å være sikker på at roboten står i ro. Videre brukes funksjonene `scan` og `kjoer_gyro` (linje 282 og 283) på samme måte som tidligere for å stille opp roboten parallelt

med "startsveggen". På linje 284 settes variabelen `Vegg` til å være false. Roboten er nå ferdig med kartleggingen av veggen, og vi antar dermed nå at ved neste deteksjon av noe foran roboten, så vil dette være et objekt, og ikke veggen. Mer om dette senere i koden. På linje 286 brukes funksjonen `snu_til_vinkel` til å rette opp roboten slik at den står i lik vinkel som den startet i (0 grader). Resterende linjer i kodeutdrag 8.11 brukes på å oppdatere ulike variabler.

Kode 8.11: Innstilling etter roboten er ferdig med kartleggingsfasen

```
277             elseif Arealscan == false %Naar den er ferdig ...
                    med kartleggingen av veggen
278                 if Strekning == 1 %Hvis den nettopp har ...
                        blitt ferdig ? kartlegge veggen
279                     motorB.Speed = 0;
280                     motorC.Speed = 0;
281                     % Stiller seg opp parallelt med ...
                            "startsveggen"
282                     [Minavstand(k), Vinkelminavstand(k)] = ...
                            scan(motorA, motorB, motorC, ...
                            mySonicSensor);
283                     [ny_x, ny_y, beregnet_posisjon2(k)] = ...
                            kjoer_gyro(motorB, motorC, x(k-1), ...
                            y(k-1), Vinkelminavstand(k), ...
                            myGyroSensor, 1);
284                     Vegg = false;
285                     GyroAngle(k) = ...
                            double(readRotationAngle(myGyroSensor));
286                     snu_til_vinkel(0, motorB, motorC, ...
                            GyroAngle(k), myGyroSensor);
287                     x(k) = ny_x;
288                     y(k) = ny_y;
289                     VinkelPosMotorB(k) = ...
                            double(motorB.readRotation);
290                     VinkelPosMotorC(k) = ...
                            double(motorC.readRotation);
291                     Strekning = Strekning + 1;
```

Neste steg i programmet blir å få roboten til å kjøre rett frem helt til den møter på et objekt eller veggen.

Møte med objekt eller vegg

Det første roboten gjør når den møter på et objekt eller en vegg er at den scanner omgivelsene sine ved bruk av funksjonen `scan`. Dette er vist på linje 295 i kodeutdrag 8.12. Linjen over (294) viser betingelsen for at funksjonen skal kjøres. Det skal enten være et objekt mindre enn 25cm foran roboten, eller så skal den ha detektert et objekt på forrige iterasjon av koden.

På linje 332 i kodeutdrag 8.12 sjekkes det om det er første gangen roboten møter på objektet/veggen. Dette er for å videre ta en sjekk for om det er en vegg eller objekt den har møtt på. Etter å ha satt `Vegg` til å være false på linje 333, opprettes det to variabler `punktx` og `punkty`. Dette er koordinatene til hindringen den har møtt på (regnes ut på samme måte som koordinatene til veggen tidligere i koden). Når disse er opprettet vil vi sjekke om koordinatene befinner seg innenfor veggen sine koordinater.

Før vi kan sjekke om koordinatene er innenfor veggen sine rammer, må vi sjekke om roboten er på vei oppover eller nedover i forhold til startsposisjonen. Dette gjøres ved koden på linje 299 i kodeutdrag 8.12. Her brukes den innebygde funksjonen `mod`, som i dette tilfellet returnerer 0 når roboten er på vei oppover, og 1 hvis den er på vei nedover. På linje 302 og 306 bruker vi funksjonen `inpolygon` til å sjekke om koordinatene er innenfor området avgrenset av veggen. Vi har trukket ifra 0.1, og lagt til 0.1, avhengig om den er på vei oppover eller nedover. Dette er en form for sikkerhetsmargin, slik at roboten ikke tror at noe detektert ti centimeter eller mindre innenfor veggen er et objekt.

Funksjonen `inpolygon` returnerer 1 hvis punktet gitt er innenfor den gitte rammen, og 0 hvis ikke. På linje 308-310 bruker vi dette til å sette `Vegg` til å være true hvis funksjonen returnerte 0. Hvis den returnerer 1 vil `Vegg` være false på bakgrunn av koden tidligere i prosjektet.

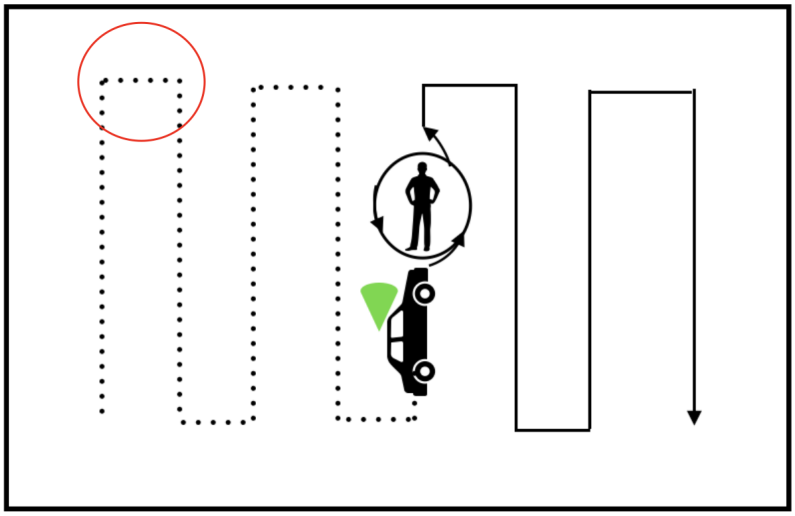
Kode 8.12: Kode for å skille mellom objekt og vegg

```
292
293                 if Avstand(k) < 0.25 || Objekt(k-1) == ...
                        true %Mooter paa objekt/vegg
294                     if Objekt(k-1) == false
295                         [Minavstand(k), ...
                                Vinkelminavstand(k)] = ...
```

```
                    scan(motorA, motorB, motorC, ...
                    mySonicSensor);
296                 Vegg = false;
297                 Punktx = x(k) + ...
                    Minavstand(k)*sin((pi*(GyroAngle(k)+Vinkelminavstand(k)))/18
298                 Punkty = y(k) + ...
                    Minavstand(k)*cos((pi*(GyroAngle(k)+Vinkelminavstand(k)))/18
299                 if mod(Strekning, 2) == 0 % ...
                    Sjekker om den er paa vei ...
                    oppover eller nedover
300                 % Hvis den er paa vei oppover ...
                        sjekker den om
301                 % "objektet" er 10cm fra veggen.
302                 in = ...
                        inpolygon(Punktx,Punkty,Veggx,Veggy ...
                        - 0.10);
303                 else
304                 %Samme med nedover, maa bare ...
                        bytte fortegn til
305                 %"+0.10"
306                 in = ...
                        inpolygon(Punktx,Punkty,Veggx,Veggy ...
                        + 0.10);
307                 end
308                 if in < 1
309                     Vegg = true;
310                 end
```

Møte med veggen

Når det viser seg at hindringen roboten møter på er en vegg vil vi at roboten skal bevege seg som illustrert ved figuren nedenfor.



Figur 8.12: Ønsket bevegelse for roboten ved møte med en vegg. Den røde ringen i figuren viser den ønskede kjørebanen til roboten når den møter på en vegg.

Første del av løsningen for dette er vist i kodeutdrag 8.13. Her er det verdt å merke seg at funksjonen `kjoer_gyro` brukes på to forskjellige måter avhengig av om den kjøres oppover eller nedover. Dette kan sees på linje 318-322. Hvis roboten er på vei oppover (som i figuren ovenfor), settes det siste argumentet i funksjonskallet til å være 1, og 2 hvis den er på vei nedover. På denne måten stiller roboten seg med veggen på venstresiden når den kjører oppover, mens veggen blir på høyresiden når den kjører nedover. På denne måten vil den kjøre sikksakk fram og tilbake.

Kode 8.13: Kodeutdrag fra `Prosjekt08`$_{k}artlegge_{r}om$

```
315                         else
316                             Objekt(k) = false;
317                             [Minavstand(k), ...
                                    Vinkelminavstand(k)] = ...
                                    scan(motorA, motorB, motorC, ...
                                    mySonicSensor);
318                             if mod(Strekning, 2) == 0
319                                 [ny_x, ny_y, ...
                                        beregnet_posisjon2(k)] = ...
                                        kjoer_gyro(motorB, motorC, ...
                                        x(k-1), y(k-1), ...
                                        Vinkelminavstand(k), ...
```

```
                                    myGyroSensor, 1);
320                             else
321                                 [ny_x, ny_y, ...
                                        beregnet_posisjon2(k)] = ...
                                        kjoer_gyro(motorB, motorC, ...
                                        x(k-1), y(k-1), ...
                                        Vinkelminavstand(k), ...
                                        myGyroSensor, 2);
322                             end
323                             x(k) = ny_x;
```

På linje 325 i kodeutdrag 8.14 sjekkes det om roboten er på vei oppover. Hvis det er tilfellet, brukes funksjonen `snu_til_vinkel` på linje 327 til å snu roboten til den står vendt 180 grader fra vinkelen den startet på, altså vendt rett nedover. Hvis den er på vei nedover, vil den samme funksjonen brukes, bare nå til å vende roboten til 0 grader.

Kode 8.14: Kodeutdrag fra `Prosjekt08_kartlegge_rom.m`

```
324                             y(k) = ny_y;
325                             if mod(Strekning, 2) == 0 %Hvis ...
                                    den er p? vei oppover, og har ...
                                    m?tt p? en vegg. Da skal den ...
                                    snu seg til motsatt retning
326                                 GyroAngle(k) = ...
                                        double(readRotationAngle(myGyroSensor));
327                                 snu_til_vinkel(180, motorB, ...
                                        motorC, GyroAngle(k), ...
                                        myGyroSensor);
328                             else
329                                 GyroAngle(k) = ...
                                        double(readRotationAngle(myGyroSensor));
330                                 snu_til_vinkel(0, motorB, ...
                                        motorC, GyroAngle(k), ...
                                        myGyroSensor);
331                             end
332                             Strekning = Strekning +1;
333                             Vegg = false;
```

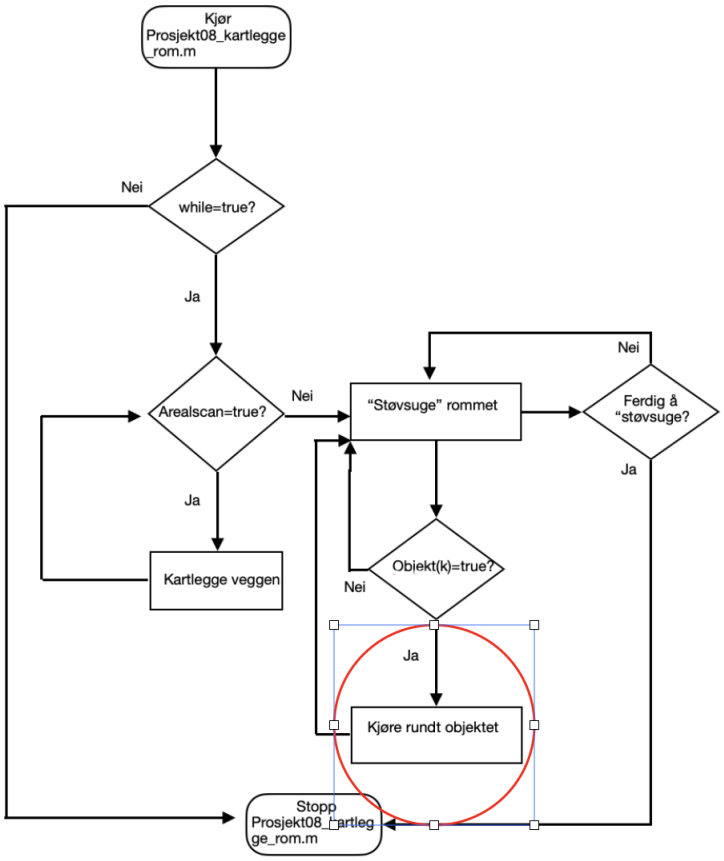
På linje 335-343 sjekkes det om roboten er ferdig med å "støvsuge" hele rommet. Dette gjøres ved å sjekke om de nåværende x- og y-koordinatene til roboten er innenfor et område på 10cm av høyeste registrerte x-koordinat og høyeste/laveste y-koordinat. Hvis dette er tilfellet vil `JoyMainSwitch` settes til å være lik 1, som vil avslutte programmet ved neste iterasjon av koden.

Kode 8.15: Kodeutdrag fra `Prosjekt08`$_{k}artlegge_{r}om$

```
335                             if x(k) > max(x) -0.10
336                                 if y(k) > max(y) -0.10
337                                     JoyMainSwitch=1;
338                                     disp('ferdig')
339                                 elseif y(k) < min(y) +0.10
340                                     JoyMainSwitch=1;
341                                     disp('ferdig')
342                                 end
343                             end
```

Møte med objekt

Når roboten møter på et objekt, blir variabelen `Objekt(k)` satt til å være true. Det som skjer deretter avhenger av om det er første gangen den møter på objektet, eller om objektet har blitt kartlagt tidligere. I forhold til flytskjemaet vil roboten være i situasjonen nedenfor.



Figur 8.13: Nåværende posisjon i flytskjemaet

I kodeutdrag 8.16 er det vist hvordan vi sjekker om det er første gangen roboten møter på objektet. På linje 391 blir det kjørt en sjekk om objektet ble registrert ved nåværende iterasjon av koden, eller om den allerede er i gang med å kartlegge objektet. Etter å ha oppdatert en del variabler på linje 392-396, blir `Punktx` og `Punkty` satt til å være koordinatene til det første punktet registrert på objektet.

Kode 8.16: Kodeutdrag fra `Prosjekt08`$_{k}artlegge_{r}om$

```
389          if Objekt(k) == true % Sjekker om den har moett ...
                 paa et objekt
```

```
390                 %Sjekker om det er foerste gangen den moeter ...
                        paa objektet
391                 if Objekt(k) == true && Objekt(k-1) == false
392                     Antall_objekt = Antall_objekt + 1;
393                     Allerede_objekt = false;
394                     Startspunkt_objekt = [x(k), y(k)];
395                     %Setter startstidspunktet naar den moeter ...
                            paa objektet
396                     Startstidspunkt = k;
397                     %Skaffer de foerste koordinatene til ...
                            objektet (punktx,
398                     %punkty)
399                     Punktx = x(k) + ...
                            Minavstand(k)*sin((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180);
400                     Punkty = y(k) + ...
                            Minavstand(k)*cos((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180);
```

På linje 401 startes en for-loop som går igjennom alle objektene i matrisen `Objekter`, som inneholder alle de tidligere registrerte objektene. Siden alle objektene har både x- og y-koordinater, blir de lagret i matrisen slik at hvert objekt tar opp to rader i matrisen. Det vil si at det første registrerte objektet vil ta opp både rad 1 og rad 2 i matrisen `Objekter`. Derfor har vi på linje 402 gjort det slik at resterende del av kodeutdraget bare kjøres hvis `k1` (som øker fra 1 til antall rader i Objekter) er et oddetall.

På linje 403 i kodeutdrag 8.17 defineres `k2` til å være en liste med indeksene til alle nonzero elementer i rad nummer k2 i matrisen Objekter. Dette må vi gjøre siden matrisen fylles på med nuller når den utvides med en rad som er lengre enn de som ligger der fra før. Altså hvis det første objektet har 7 x- og y-koordinater, også legges det et nytt objekt til i matrisen med 9 x- og y-koordinater, vil matrisen utvides til å være en 4x9 matrise. De to siste kolonnene i rad 1 og 2 (det første objektet) vil da inneholde tallet null, da det ikke var noe der fra før av.

Funksjonen `boundary` blir i linje 406 brukt til å lage et omrisset av punktene hentet ut fra det relevante objektet (mer om funksjonen kan leses her: [1]). På linje 418 og 419 blir det opprettet to variabler `x_koords` og `y_koords` som inneholder koordinatene av omrisset til det nåværende objektet. For å sjekke om startspunktet er inni, eller på dette omrisset bruker vi funksjonen `inpolygon` (Mer om denne funksjonen her: [4]) på linje 422. Kort fortalt returnerer denne 1 hvis det relevante punktet er inni, eller på et avgrenset område definert av de to siste inngangene til funksjonen. På linje 414-418

bruker vi dette til å sette variabelen `Allerede_objekt` til å være true hvis dette er tilfellet.

På denne måten får vi gått igjennom alle objektene i listen med objekter, og sjekket om startspunktet for det nye objektet allerede finnes i disse.

Kode 8.17: Kode for å sjekke om objektet har blitt registrert tidligere

```
401                 for k1 = 1:size(Objekter,1) %Gaar igjennom ...
                        listen med objekter, med hensyn paa radene
402                     if mod(k1, 2) ≠ 0 % Sjekker om k1 ikke ...
                            er delelig med to. Ettersom ...
                            koordinatene til objektene kommer ...
                            i par (x- og y-koordinater)
403                         k2 = find(Objekter(k1,:)); %Finner ...
                                alle nonzero entries i objektet
404                         %Definerer boundary til ? v?re ...
                                omr?det avgrenset av
405                         %objektet
406                         boundary = ...
                                boundary(Objekter(k1,k2)', ...
                                Objekter(k1+1,k2)', 0.1);
407                         %Definerer x- og y-koordinatene ...
                                til ytterkanten av
408                         %objektet
409                         x_koords = Objekter(k1, boundary);
410                         y_koords = Objekter(k1 +1, boundary);
411
412                         %Sjekker om startspunktet allerede ...
                                finnes i objektet
413                         [in2, on] = ...
                                inpolygon(Punktx,Punkty,x_koords, ...
                                y_koords);
414                         if in2 > 0 || on > 0
415                             Allerede_objekt = true;
416                         else
417                             continue
418                         end
419                     end
420                 end
```

Allerede registrert objektet

Hvis objektet allerede er blitt registrert tidligere i prosjektet, er målet at den skal kjøre rundt dette, for å så fortsette sin opprinnelige kjørebane. Hvordan vi har valgt å implementere dette i koden kan sees i kodeutdrag 8.18. Linje 422 og 423 brukes til å oppdatere variablene `Objekt` og `Antall_objekter`. For å få roboten til å vite når den er kommet rundt objektet, definerer vi i linje 434 `Startspunkt_x` til å være x-koordinatet til roboten i møte med objektet. Resten av kodeutdraget viser koden brukt for å kjøre rundt objektet når roboten er på vei oppover. Siden koden brukt når kjøreretningen nedover nesten er identisk, er ikke denne inkludert i kodeutdraget.

Når roboten kjører rundt objektet, vil den kjøre rundt på høyresiden (siden vi har definert retningen i `kjoer_gyro` til å være 1). Vi vet da at hvis robotens x-koordinat er mindre enn verdien til `Startspunkt_x`, vil det si at roboten er kommet på andre siden av objektet.

Hvis `x(k)` er større enn starsposisjonen (linje 432), vil roboten fortsette å kjøre rundt objektet ved bruk av funksjonene `scan` og `kjoer_gyro`, som tidligere. Når `x(k)` er blitt mindre enn startsposisjonen, vil koden på linje 438-441 bli kjørt. `Allerede_objekt` settes først til å være false igjen, og roboten rettes opp ved bruk av `snu_til_vinkel`.

Kode 8.18: Kode for å kjøre rundt objektet

```
421                 if Allerede_objekt
422                     Objekt(k) = false;
423                     Antall_objekt = Antall_objekt - 1;
424                     if mod(Strekning, 2) == 0 %Sjekker om ...
                            den er paa vei oppover
425                         Startspunkt_x = x(k);
426                         %Tar foerste maaling for aa kjoere ...
                                rundt objektet.
427                         [Minavstand(k), ...
                                Vinkelminavstand(k)] = ...
                                scan(motorA, motorB, motorC, ...
                                mySonicSensor);
428                         [ny_x, ny_y, ...
                                beregnet_posisjon2(k)] = ...
                                kjoer_gyro(motorB, motorC, ...
                                x(k), y(k), ...
                                Vinkelminavstand(k), ...
```

```
                                myGyroSensor, 1);
429                         x(k) = ny_x;
430                         y(k) = ny_y;
431                         while Allerede_objekt
432                             if x(k) > Startspunkt_x ...
                                    %Kjorer rundt objektet
433                                 [Minavstand(k), ...
                                        Vinkelminavstand(k)] = ...
                                        scan(motorA, motorB, ...
                                        motorC, mySonicSensor);
434                                 [ny_x, ny_y, ...
                                        beregnet_posisjon2(k)] ...
                                        = kjoer_gyro(motorB, ...
                                        motorC, x(k), y(k), ...
                                        Vinkelminavstand(k), ...
                                        myGyroSensor, 1);
435                                 x(k) = ny_x;
436                                 y(k) = ny_y;
437                             else %Hvis den har kjoert ...
                                    rundt objektet
438                                 Allerede_objekt = false;
439                                 GyroAngle(k) = ...
                                        double(readRotationAngle(myGyroSensor));
440                                 snu_til_vinkel(0, motorB, ...
                                        motorC, GyroAngle(k), ...
                                        myGyroSensor); %Rettes ...
                                        opp til ? ha riktig ...
                                        retning
441                                 disp('kjoert rundt objekt')
442                             end
443                         end
```

Første møte med objektet

Hvis `Allerede_objekt` i kodeutdrag 8.17 ikke settes til true, vil det si at det er første gangen roboten møter på objektet. Det vil da være koden i kodeutdrag 8.19 som blir kjørt. Her opprettes de første koordinatene til det nye objektet (`Objekt_koordinater_x og Objekt_koordinater_y`. På linje 470 og 471 opprettes det som kan sees på som et firkantet område rundt startspunktet der det er 10cm x-retning og 20cm i y-retning (litt utydelig kanskje). Dette vil senere bli brukt til å bestemme når roboten har kjørt rundt hele objektet. Etter å ha brukt `kjoer_gyro` til å begynne å kjøre rundt objektet, legges de nye detekterte koordinatene i listen med koordinater på

linje 475 og 476.

Kode 8.19: Kode for første møte med objektet

```
466                 else %Hvis objektet ikke har blitt ...
                        detektert tidligere
467                     Objekt_koordinater_x = [Punktx];
468                     Objekt_koordinater_y = [Punkty];
469                     Startx_grid = [Startspunkt_objekt(1) - ...
                            0.1, Startspunkt_objekt(1) + 0.1];
470                     Starty_grid = [Startspunkt_objekt(2) - ...
                            0.2, Startspunkt_objekt(2) + 0.2];
471                     [ny_x, ny_y, beregnet_posisjon2(k)] = ...
                            kjoer_gyro(motorB, motorC, x(k-1), ...
                            y(k-1), Vinkelminavstand(k), ...
                            myGyroSensor, 1);
472                     x(k) = ny_x;
473                     y(k) = ny_y;
474                     [Minavstand(k), Vinkelminavstand(k)] = ...
                            scan(motorA, motorB, motorC, ...
                            mySonicSensor);
475                     Objekt_koordinater_x = ...
                            [Objekt_koordinater_x, x(k) + ...
                            Minavstand(k)*sin((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180)];
476                     Objekt_koordinater_y = ...
                            [Objekt_koordinater_y, y(k) + ...
                            Minavstand(k)*cos((pi*(GyroAngle(k)+Vinkelminavstand(k)))/180)];
477                 end
```

Allerede i gang med å kartlegge objektet

Neste situasjon vi vil ta for oss er når den allerede er i fasen hvor den kjører rundt objektet. Den skal da kjøre rundt hele objektet, mens den tar vare på koordinatene til objektet.

For å sjekke om roboten har kommet rundt objektet, sammenligner vi koordinatene med starskoordinatene. For sikkerhets skyld lar vi det derfor gå tre iterasjoner av koden siden den først møtte på objektet (linje 485 i kodeutdrag 8.20). Slik at den ikke tror den er kjørt rundt objektet med en gang den møter på det.

Kode 8.20: Kode for å sjekke om roboten er ferdig med å kjøre rundt objektet

```
485                 if k > Startstidspunkt + 3
486                     if x(k) > Startx_grid(1) && x(k) < ...
                            Startx_grid(2)
487                         if y(k) > Starty_grid(1) && y(k) < ...
                                Starty_grid(2)
488                             Stoerrelse = size(Objekter,2); ...
                                    %Finner antall kolonner i ...
                                    Objekter
489                             Lengde = ...
                                    length(Objekt_koordinater_x);
490                             if Stoerrelse > Lengde
491                                 Forskjell = Stoerrelse - ...
                                        Lengde;
492                                 Objekter(Stoerrelse+1:Stoerrelse+2,:) ...
                                        = ...
                                        [Objekt_koordinater_x ...
                                        zeros(1, Forskjell) ; ...
                                        Objekt_koordinater_y ...
                                        zeros(1, Forskjell)];
493                             else
494                                 Forskjell = Lengde - ...
                                        Stoerrelse;
495                                 Objekter(Stoerrelse+1:Stoerrelse+2,1:Lengde) ...
                                        = ...
                                        [Objekt_koordinater_x ...
                                        ; Objekt_koordinater_y ];
496                             end
```

Møte med objekt

På linje 488 i kodeutdraget nedenfor defineres `Stoerrelse` til å være antall rader i matrisen med objekter (`Objekter`), og `Lengde` til å være lengden på listen med x-koordinatene (som selvfølgelig er like lang som den med y-koordinatene). Måten man legger til koordinatene i matrisen med objekter avhenger av hvilken av disse størrelsene som er størst. Hvordan dette gjøres kan sees på linje 490-496. Etter dette tømmes listene med koordinater og `Objekt(k)` settes til false igjen.

Kode 8.21: Kode for å sjekke om det er første møte med objektet

```
485                 if k > Startstidspunkt + 3
```

```
486                         if x(k) > Startx_grid(1) && x(k) < ...
                                Startx_grid(2)
487                             if y(k) > Starty_grid(1) && y(k) < ...
                                    Starty_grid(2)
488                                 Stoerrelse = size(Objekter,2); ...
                                        %Finner antall kolonner i ...
                                        Objekter
489                                 Lengde = ...
                                        length(Objekt_koordinater_x);
490                                 if Stoerrelse > Lengde
491                                     Forskjell = Stoerrelse - ...
                                            Lengde;
492                                     Objekter(Stoerrelse+1:Stoerrelse+2,:) ...
                                            = ...
                                            [Objekt_koordinater_x ...
                                            zeros(1, Forskjell) ; ...
                                            Objekt_koordinater_y ...
                                            zeros(1, Forskjell)];
493                                 else
494                                     Forskjell = Lengde - ...
                                            Stoerrelse;
495                                     Objekter(Stoerrelse+1:Stoerrelse+2,1:Lengde) ...
                                            = ...
                                            [Objekt_koordinater_x ...
                                            ; Objekt_koordinater_y ];
496                                 end
497                                 %Plasserer de nye objektenes ...
                                        koordinater i
498                                 %listen med objekter
499                                 %"Tommer" listene for ...
                                        koordinater saa de er klar
500                                 %til neste objekt
501                                 Objekt_koordinater = [];
502                                 Objekt_koordinater_x = [];
503                                 Objekt_koordinater_y = [];
504                                 Objekt(k) = false;
505                                 disp('Ferdig aa kartlegge objekt')
506                                 if mod(Strekning, 2) == 0 ...
                                        %Sjekker om den er paa vei ...
                                        oppover
```

Resterende linjer i `prosjekt08_kartlegge_rom` brukes til å få roboten til å kjøre rundt objektet, samt plotting. Koden vi har brukt for å få roboten til å kjøre rundt objektet er lik som i kodeutdrag 8.18, og blir forklart tidligere i kapittelet.

8.2.6 Plotting

Det siste i koden er selve plottingen av området. Her har vi valgt å plotte veggen, kjørebanen i tillegg til eventuelle objekter. Hvordan vi har valgt å løse det kan sees i kodeutdrag 8.22. For-løkken i kodeutdraget går igjennom hvert objekt i matrisen med objekter, for å så plotte koordinatene ved hjelp av `boundary` funksjonen i matlab.

Kode 8.22: Kode for å sjekke om det er første møte med objektet

```
584     plot(x(1:k),y(1:k));
585     title('Vegg og kjoerevei')
586     xlabel('avstand[m]')
587     hold on
588     plot(Veggx(1:numel(Veggx)),Veggy(1:numel(Veggy)));
589     legend('Posisjon ved Gyro', 'Veggen')
590     antall = 0;
591     for k3 = 1:size(Objekter,1) %G?r igjennom listen med ...
            objekter
592         if mod(k3, 2) ~= 0
593             antall = antall +1;
594             k2 = find(Objekter(k3,:));
595             boundary1 = boundary(Objekter(k3,k2)', ...
                    Objekter(k3+1,k2)', 0.1);
596             x_koords2 = Objekter(k3, boundary1);
597             y_koords2 = Objekter(k3 +1, boundary1);
598             plot(x_koords2, y_koords2)
599             for z = 1:antall
600                 objektlegend(z, :) = sprintf('Objekt ...
                        nummer %d', z);
601             end
602             legend('Posisjon ved Gyro', 'Veggen', ...
                    objektlegend)
603         end
604     end
```

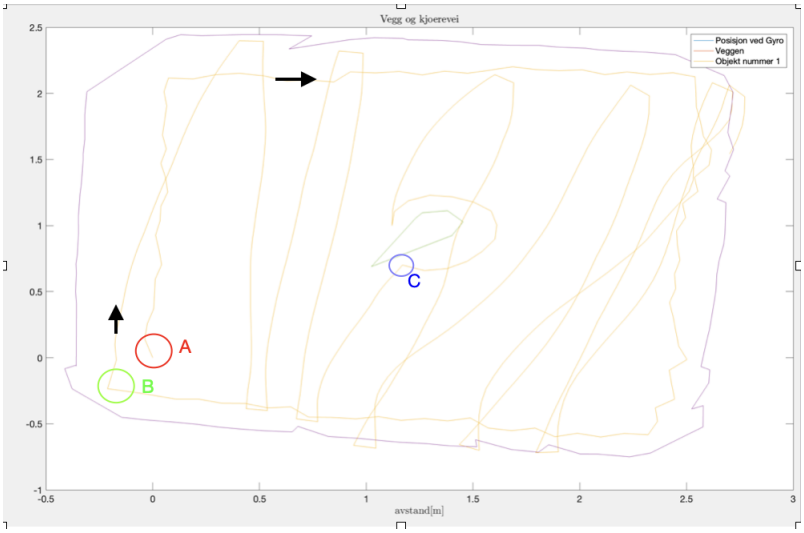
8.3 Resultat

I dette kapittelet vil vi presentere resultatet for kartleggingen av rommet. For å validere koden konstruerte vi et avgrenset område for roboten ved å sette opp bord som vist i figuren nedenfor. I midten av rommet plasserte

vi en søppelbøtte som et objekt for roboten å møte på. Området kan ses i bilde nedenfor

Figur 8.14: Område brukt for støvsuging. Søppelbøtten i midten av rommet er ment å være et objekt

Startsposisjonen til roboten satt vi til å være i et av hjørnene til det konstruerte rommet. Dette gjorde at vi endte med et plot som vist i figur 8.15.



Figur 8.15: Kartlegging/støvsuging av rommet. A: startsposisjonen til roboten. B: Posisjonen til roboten når den er ferdig å kartlegge veggen. C: Posisjonen til roboten ved møte med objektet. Pilene i figuren er for å vise robotens kjøreretning.

Noe som er verdt å merke seg er at det ser ut til at roboten sin posisjon ble noe feil etter å ha kjørt rundt objektet. Hvis man ser helt nederst på figuren, ser det ut som at roboten har kjørt igjennom veggen, da dette åpenbart ikke var tilfellet.

Årsaken til dette tror vi kom av at hjulene så til å miste grepet enkelte steder, blant annet når den kjørte rundt objektet. Dette gjorde at vinkelposisjonen til motorene endret seg, uten at posisjonen til roboten faktisk gjorde det. I koden brukte vi vinkelposisjonen til hjulene for å regne ut den tilbakelagte strekningen til roboten. Derfor vil posisjonen til roboten bli noe feil i forhold til realiteten. Dette gjorde derimot ikke at den ikke klarte å fullføre støvsugingen.

Kapittel 9

Konklusjon

Hovedmålet for oppgaven var å lage forslag til løsning for enkelte av de kreative oppgavene fra ELE130, samt komme med forslag til nye kreative oppgaver. Gjennom prosjektet har vi kommet med løsninger til flere av de store prosjektene, samt kommet med forslag til en ny oppgave. Enkelte av løsningsforslagene er nok litt over det nivået som er forventet av en student på første trinn. Det vil uansett kunne brukes til å sammenlignes med det studentene klarer å få til, slik at man ser hva som faktisk er mulig og ikke.

Bibliografi

[1] boundary. https://se.mathworks.com/help/matlab/math/creating-and-concatenating-matrices.html.

[2] Building instructions. https://le-www-live-s.legocdn.com/sc/media/lessons/mindstorms-ev3/building-instructions/ev3-rem-driving-base-79bebfc16bd491186ea9c9069842155e.pdf.

[3] Ev3 devices. https://pybricks.com/ev3-micropython/ev3devices.html#ultrasonic-sensor.

[4] inpolygon. https://se.mathworks.com/help/matlab/ref/inpolygon.html.

[5] Reguleringsteknikk sammendrag. http://mekaunikum.no/wp-content/uploads/2014/06/Reguleringsteknikk-Sammendrag.pdf.

[6] Solve differential equation. https://se.mathworks.com/help/symbolic/solve-a-single-differential-equation.html.

[7] Tsbf engineering. http://www.tsbfengineering.com/jorgen/2017/10/3/overflatefinhet-og-finish.

[8] C. Bakos. Lego mindstorms og matlab; anvendt matematikk og fysikk i skjønn forening. Technical report, Universitet i Stavanger, 2019. ING100.

[9] T. Drengstig. Lego mindstorms og matlab; anvendt matematikk og fysikk i skjønn forening. Technical report, Universitet i Stavanger, 2019. Utdelt materiale.