Nr. _____ av _____

Sjøkrigsskolen

Bacheloroppgave

Utvikling av autonomt system på landbasert plattform

GPS-uavhengig navigering i trange miljøer

av

Daniel Hassel, Sondre Moen & Kristian Ohma

Levert som en del av kravet til graden:

BACHELOR I MILITÆRE STUDIER MED FORDYPNING I ELEKTRONIKK OG DATA

Innlevert: Desember 2019

Godkjent for offentlig publisering

Publiseringsavtale

En avtale om elektronisk publisering av bachelor/prosjektoppgave

Kadettene har opphavsrett til oppgaven, inkludert rettighetene til å publisere den.

Alle oppgaver som oppfyller kravene til publisering vil bli registrert og publisert i Bibsys Brage når kadettene har godkjent publisering.

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

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

Plagiaterklæring

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

Dato: 03.12.2019

Daniel Finbak Hassel _____
Kadett navn Kadett, signatur

Sondre Granlund Moen _____
Kadett navn Kadett, signatur

Kristian Rosten Ohma _____
Kadett navn Kadett, signatur

Forord

Bacheloroppgaven er et krav for militære studier med fordypning i elektronikk og data ved Sjøkrigsskolen. Denne oppgaven er skrevet av Daniel Hassel, Sondre Moen og Kristian Ohma i tidsrommet Mai 2019 til Desember 2019.

Oppgaven har gitt oss mulighet til å undersøke hvordan autonomi kan se ut i et konkret system. Inspirasjonen til oppgaven er basert på en fremtidsrettet tankegang rundt hvordan autonomi kan komme til å påvirke Forsvaret og samfunnet vi lever i. Prosjektet har gitt oss økt kompetanse innenfor programmering, elektronikk og sammensatte systemer.

Vi vil gjerne rette en stor takk til Alexander Sauter for gode samtaler underveis i bachelorperioden. Samtidig vil vi også takke Frode Wikne for hjelp med 3D-printing av komponenter.

Bergen, Sjøkrigsskolen, 03.12.2019

Daniel Hassel

Sondre Moen

Kristian Ohma

Oppgaveformulering

I 2018 utviklet Martin Tande og Sondre Flaatten en testplattform for autonome systemer til bruk i undervisning og kommende bacheloroppgaver (Flaatten og Tande, 2018). Autonome systemer preger i dag den teknologiske utviklingen, derfor er det viktig å få et innsyn i hvordan denne teknologien fungerer. Med bakgrunn i dette passet testplattformen meget godt som utgangspunkt for å lage en autonom plattform. Fokuset i denne oppgaven vil derfor ligge på å lage en autonomt kjørende plattform. Videreførelse av modulariteten er også et fokus, for å kunne bli brukt i videre undervisning og/eller videreutvikling av prosjektet.

Sammendrag

I den teknologiske utviklingen er det et større fokus på hvordan autonome systemer fungerer enn tidligere. Forsvaret har økt sin satsing innen autonomi som endrer måten operasjoner planlegges og gjennomføres på. Med bakgrunn i denne utviklingen forsøker oppgaven å gi en større forståelse rundt hvordan et konkret autonomt system kan fungere.

Gjennom arbeidet av plattformen er det forsøkt å videreføre modulariteten i konstruksjonen, ved å gi en enkel og oversiktlig enhet å jobbe med. Etter anbefaling fra plattformens utviklere ble motorkontrollene erstattet med en mer robust type. Det er også fjernet moduler fra PLS og ultralyd-baserte avstandsmålere som ble overflødige, grunnet oppgradering av sensorpakken. Disse ble erstattet med en Lidar som brukes til hurtig skanning av plattformens miljø.

Raspberry Pi, PLS og ROS er brukt som styrings – og kommunikasjonsenheter. ROS er en linuxbasert programvare for utvikling av software til roboter. Videre er plattformen utvidet på hardwaresiden med en Arduino, som primært benyttes gjennom puls bredde modulasjon for å styre motorene. Disse motorkontrollsignalene blir sendt via programmeringsspråket Node-Red som fungerer som et bindeledd mellom plattformens ulike enheter. Det implementerte Lidar systemet er kompatibelt med en rekke programmeringsspråk. Etter flere vurderinger av ulike språk falt valget på Python, da det er et utbredt programmeringsspråk som er lett å integrere i ROS.

Siden oppstarten av bachelorperioden, har tanken vært å bruke systemer som brukes i utdannelsen ved Sjøkrigsskolen. Gjennom arbeidsprosessen har det blitt identifisert flere områder hvor plattformen kan videreutvikles. Det har vært en prioritet å begrense bruken av ukjente programmeringsspråk, slik at fokuset kan legges på integreringen av nye komponenter.

Plattformen kan styres på tre ulike måter, gjennom Dualshock PS4 kontroller, gjennom PLS og via ROS. Dette gjør at brukere har mulighet til selv å kunne velge hvordan plattformen skal komme seg fra A til B, og åpner muligheten til å utvide systemet fra forskjellige innfallsvinkler.

Oppgaven tar utgangspunkt i Sheridan og Verplank (Hareide, 2018) sin rangering av hvor autonomt et system er, i den hensikt å få et bilde over hva som kreves i et autonomt

system. Datamaskinen utfører plattformens handlinger automatisk, og informerer brukeren i etterkant om hva den har gjort.

Oppgaven har lyktes med å få plattformen til å kjøre autonomt innendørs. Muligheten til å kjøre autonomt utendørs med GPS-støtte ble også undersøkt. Navio2 anses å ha relevant sensorpakke og brukersystem for videreutvikling av plattformen.

Innholdsfortegnelse

Publiseringsavtale ..... **i**

Forord ..... **ii**

Oppgaveformulering ..... **iii**

Sammendrag ..... **iv**

Innholdsfortegnelse ..... **vi**

Figurer **8**

Tabeller/Diagrammer **10**

Nomenklatur / Forkortelser / Symboler **11**

1 Introduksjon **12**

1.1 Bakgrunn 12

1.2 Mål 13

1.3 Begrensninger 13

1.4 Metode 14

1.5 Struktur 14

2 Teori **15**

2.1 Autonomi 15

2.2 Raspberry Pi 17

2.3 Temperatursensor 18

2.4 Robot Operating System (ROS) 19

2.5 Pathfinding 20

2.5.1 Dijkstras algoritme og Best-First-Search 23

2.5.2 Algoritme A* 25

2.6 Kvaternioner 27

2.7 Navio2 28

2.8 Ardupilot (Mission Planner) 29

3 Konseptutvikling **31**

3.1 Plattform 31

3.2 Fremdriftssystem 32

3.3 Sensorer ... 34
3.4 System for utvikling av autonomt konsept ... 35
3.5 Programmeringsspråk ... 36
3.6 Pathfinding algoritme ... 38

4 Implementering ... 40
4.1 Systemoversikt ... 40
4.2 Karosseri ... 41
4.2.1 Endringer i karosseri ... 41
4.2.2 Konstruksjon ... 44
4.3 Fremdriftssystem ... 48
4.3.1 Arduino ... 49
4.3.2 Odrive Motorkontroller ... 51
4.4 Styringsenheter ... 56
4.4.1 Programmerbar Logisk Styring ... 56
4.4.2 Node-Red ... 62
4.5 Utvikling av kildekode ... 71
4.5.1 Hvordan fremstilles kartet ... 72
4.5.2 Implementering av A* ... 72
4.5.3 Implementering av valgfunksjonen ... 73
4.5.4 Systemets virkemåte ... 73
4.5.5 Utvikling av script for plotting ... 74

5 Tester ... 75
5.1 Tuning av nøyaktighetskonstant ... 75
5.2 Navigasjonsevne i dynamiske omgivelser ... 78
5.3 Test av turtall og fart på motor ... 81
5.4 Test av effektforbruk ... 83

6 Drøfting ... 85

7 Konklusjon med anbefaling ... 89

8 Bibliografi ... 91

A. Dokumentasjon ... 96
A.1 Arrangementstegning ... 96
A.2 Rekkeklemmetabeller ... 98
A.3 Playstation 4 kontroller ... 99
A.4 Oppstartsprosedyre ... 102

A.5 Installering av ROS 104
A.5.1 RPLIDAR ROS Pakke 106
A.5.2 HECTOR_SLAM ROS Pakke 108
A.5.3 Egen ROS pakke 111
A.6 Raspberry Pi GPIO tilkoblinger 112
A.7 Budsjett 112

B Kildekode 113
B.1 Action.py 113
B.2 aStar.py 114
B.3 directionDatabase.py 117
B.4 drawLine.py 118
B.5 main.py 119
B.6 myMap.py 121
B.7 plotter.py 124
B.8 position.py 126
B.9 queue.py 126
B.10 sendMessage.py 127
B.11 stop.py 127
B.12 Tastaturstyring 128
B.13 Kode for måling 129
B.14 mapping_rover.launch 129
B.15 rover.launch 130
B.16 rviz_rover.rviz 131

Figurer

Figur 2.1 - Oversikt over Raspberry Pi. Det finnes 40 tilkoblingspunkter som stort sett kan konfigureres fleksibelt (Carnino, 2016). 17
Figur 2.2 - Temperatursensoren PT100 (elektrofag.info, 2019), (RS-online, 2019). 18
Figur 2.3 - Noder kommuniserer ved hjelp av topics. Her kommuniserer Node_1 med Node_2 via Topic_1 og Node_2 med Node_3 via Topic_2 (Huang, 2016) 19
Figur 2.4 - Viser hvordan Master-noden fungerer (Huang, 2016) 19
Figur 2.5 - Illustrasjon av hvordan pathfinding algortimer fungerer (Patel, 2019) .. 20
Figur 2.6 - Ulike typer kart 22
Figur 2.7 - Illustrasjon av svakhetene til Dijkstra (Patel, 2019) 23
Figur 2.8 - Forskjellen på Dijkstra (venstre) og Best-First (Høyre) (Patel, 2019) ... 24
Figur 2.9 - A* på enkle oppgaver (Patel, 2019) 25
Figur 2.10 - A* på mer komplekse oppgaver (Patel, 2019) 25
Figur 2.11 - Enhetssirkler for Kvaternioner og Radianer sett med Nord som opp .. 27
Figur 2.12 - Oversiktsbilde over Navio2 på en Raspberry Pi (Emlid, 2015). 28
Figur 2.13 - Tiltenkt styring ved implementering av Ardupilot (Ardupilot Dev Team, 2018) 30
Figur 3.1 - Opprinnelige plattform (Flaatten og Tande, 2018). 31
Figur 3.2 - Ultralyd sensor HC-HC-SR04 34
Figur 4.1 - Systemoversikt 40
Figur 4.2 - Bilde av plattformen 41
Figur 4.3 - Plattform før endringer 42
Figur 4.4 – Plattform etter endringer 43
Figur 4.5 - Ultralyd sensor og Lidar 44
Figur 4.6 - Ramme (Flaatten og Tande, 2018) 45
Figur 4.7 - Tegning av 3D-deler 45
Figur 4.8 - Hovedstrømbryter 46
Figur 4.9 - DC-DC omsettere og brytere for 24V og 5V 46
Figur 4.10 - Oversikt over fremdriftssystemet til plattformen 48
Figur 4.11 - Oversiktsbilde over Arduino 49
Figur 4.12 - Oversiktsbilde over motorkontrolleren. Nummereringen forklares i tabell 4.3 51
Figur 4.13 - Viser hvordan filterkondensatorene er loddet på 52
Figur 4.14 - Odrv0 refererer til serienummeret på motorkontrolleren. 53
Figur 4.15 - Innsiden av hoverboard motor (Jaszczolt, 2017) 53
Figur 4.16 - Global variabel liste 57
Figur 4.17 - Interne variabler i PLS 58
Figur 4.18 - Tastaturblokk 58
Figur 4.19 - Switchblokk system 59
Figur 4.20 - Målingsblokk 59

Figur 4.21 - Forside 60
Figur 4.22 - PLS styreside 60
Figur 4.23 - Overvåkinger fra PLS 61
Figur 4.24 - Viser hvordan switch systemet fungerer 63
Figur 4.25 - Tilkobling mellom Node-Red og PLS 64
Figur 4.26 - Bildet viser flowen til PS4 styresystemet og hvordan det er satt sammen. 65
Figur 4.27 - Bluetooth noden sender verdier fra PS4 kontrolleren 66
Figur 4.28 - Viser utgangene til switch noden 67
Figur 4.29 - To switch noder som separerer analoge og boolske knapper 68
Figur 4.30 - Sjekker om Node-Red har fått kontakt med bluetooth 69
Figur 4.31 - Kommunikasjonen mellom Node-Red og ROS 70
Figur 5.1 - Testoppsett dynamisk miljø 78
Figur 5.2 - Turtallsmåler 81
Figur 5.3 - Multimeter av typen Amprobe 38XR-A med strøm sonde Tektronix A622 83
Figur A.1 - Oversiktsbilde med forklaring 96
Figur A.2 - RPLIDAR modul 97
Figur A.3 - Oversiktsbilde over knapper og navn 99
Figur A.4 - Subflow Dualshock PS4 kontroller 101
Figur A.5 - Viser innholdet i bluetoothminimal.py 101
Figur A.6 - Lidar laserscan visualisert i RVIZ 107
Figur A.7 - HECTOR_SLAM kartet visualisert i RVIZ 109
Figur A.8 - Forklaring av RVITZ kart 110
Figur B.1 – Tastaturstyring i PLS 128
Figur B.2 - Kode for målinger i PLS 129

Tabeller/Diagrammer

Tabell 2.1 - Sheridan og Verplanks nivå av automasjon (Hareide, 2018). 16
Tabell 3.1 - Antall søkeresultater for de respektive programmeringsspråkene 36
Tabell 3.2 - Fordeler/Ulemper ved Python (Tellez, 2019) 37
Tabell 3.3 - Fordeler/Ulemper ved C++ (Tellez, 2019) 37
Tabell 3.4 - Antall søkeresultater for de respektive algoritmene 38
Tabell 4.1 - Nummereringen i Figur 4.11 vises. 49
Tabell 4.2 - Pins fra Arduino til motorkontroll 1 & 2 50
Tabell 4.3 - Tabellen til venstre viser koblingene mellom hallsensorene og motorkontrolleren. Tabellen til høyre forklarer nummereringen på Figur 4.12 51
Tabell 4.4 - Oppkobling filterkondensatorer 52
Tabell 4.5 - WAGO modulene som er brukt med tilhørende forklaring 56
Tabell 4.6 - Oversikt over nodene som er brukt i Node-Red 62
Tabell 4.7 - Styremåter for PS4 66
Tabell 5.1 - Nøyaktighetsgrad 0.2 radianer 76
Tabell 5.2 - Nøyaktighetsgrad 0.1 radianer 76
Tabell 5.3 - Nøyaktighetsgrad 0.05 radianer 77
Tabell 5.4 - Dynamisk endring av vei, hinder 79
Tabell 5.5 - Dynamisk endring av vei, total blokade 79
Tabell 5.6 - Resultatene viser at PLS og PS4 er kalibrert på samme maks hastighet 81
Tabell 5.7 - Effektforbuk til komponentene. 84
Tabell A.1 - Knapper brukt på PS4 kontrolleren 99
Tabell A.2 - Oversiktsbilde over output og tilhørende navn 100
Tabell A.3 - Viser GPIO punktene som er i bruk. 112
Tabell A.4 - Budsjettoversikt 112

Nomenklatur / Forkortelser / Symboler

GPIO	General Purpose Input/Output
LIDAR	Light Detection and Radar
PLS	Programmerbare Logiske Styringer
GVL	Global Variable List
ROS	Robot Operating System
RVIZ	ROS Visualization
SLAM	Simultaneous Localization And Mapping
DSI	Display Serial Interface
HAT	Hardware Attached on Top
Node-Red	Programmeringsverktøy for å koble sammen ulike hardware komponenter.
RPM	Rotasjoner per minutt
Msg.payload	Melding med data i Node-Red.

1 Introduksjon

1.1 Bakgrunn

Den teknologiske utviklingen går i rekordfart, og forskningsinstitutt rundt om i verden jobber på spreng for å minimere risikoen for å tape terreng i de teknologiske fremskrittene. Dette gjør at Forsvaret er nødt til å tilpasse seg etter hvordan teknologien utvikler seg. «Teknologiutvikling innen autonomi er kommet lenger i USA enn i resten av Europa. Dette teknologigapet vil påvirke samarbeidet med NATO» (Knutsen, 2018). Det teknologiske kappløpet påvirker den sikkerhetspolitiske situasjonen i stor grad, og det er derfor viktig å være klar over hva utviklingen har å si for samarbeid innad i NATO, men også utad. Det er derfor viktig for Forsvaret å fokusere på teknologisk utvikling, og særlig innenfor autonomi.

«Sjøforsvarets intensjon med autonomi bør i hovedsak baseres på å redusere tap av menneskeliv, samt effektivisering av operasjoner der mennesket er en restriksjon» (Hareide, 2018). Hugin, Sjøforsvarets autonome undervannsfartøy, er et resultat av et langvarig samarbeid mellom Forsvaret og forskningsmiljøer. Systemet viser fordelene ved ubemannede systemer og opererer i tråd med det Hareide mener burde være Sjøforsvarets intensjon hva gjelder autonomi. Basert på dette kan det i stor grad argumenteres for at autonome systemer kommer til å prege Forsvarets operative handlemåte.

Forsvarets forskningsinstitutt (FFI) mener at autonome systemer for Forsvaret gir en helt ny måte å operere på, og lister opp eksempler på hvordan autonome systemer kan brukes i Forsvaret. Under mineryddingsoppdrag vil det være tryggere å bruke maskiner, da dette kan redde menneskeliv. Videre pekes det på at autonome systemer kan brukes i baseforsvar, siden maskiner ikke sovner på vakt, samt finner løsninger på problemer raskere enn mennesker. Til slutt peker FFI på at droneovervåkning kan bli benyttet i fremtiden, ettersom mange maskiner som samarbeider kan gjøre jobben billigere enn mennesker (FFI, 2019). Det kan tenkes at plattformen kan brukes til å bistå operatører til å frakte utstyr, personell eller fungere som en observasjonsenhet.

Med bakgrunn i dette skal det i denne oppgaven utvikles en autonom plattform med den hensikt å få et større innblikk i hvordan autonomi fungerer i praksis. Dette fordi det er et fremtidsrettet emne som potensielt kommer til å ha stor påvirkning på forsvarssektoren

og enkeltpersoner. Det kan derfor argumenteres for at Sjøkrigsskolen bør legge til rette for at utdanningen i større grad skal fokusere på autonome systemer.

1.2 Mål

Hovedmålet med denne oppgaven er å lage en autonom plattform som kan være med å danne en større forståelse for hvordan autonome systemer fungerer. Fokuset skal rettes mot å videreføre modulariteten i plattformen. Med modularitet menes tanken om at plattformen er konstruert for at nye systemer enkelt kan integreres. Oppgaven tar utgangspunkt i en kravspesifikasjon, hvor hensikten er å gi bruker innblikk i hva plattformen kan gjøre.

1.3 Begrensninger

Plattformen er begrenset til å arbeide ut i fra noen krav som vist i kravspesifikasjonen. Dette ble gjort for å ha spesifikke punkter å arbeide ut fra for lettere å definere hvordan bruksområde til plattformen skal være.

Bacheloroppgaven er avgrenset med en budsjettgrense på 30 000 kr. Dette har gjort at oppgaven i større grad kan implementere produkter av kvalitet, fremfor produkter som begrenser oppgaven. Oppgaven har vært opptatt av å ikke bruke mer enn den er nødt til, hvor det har blitt forsøkt å holde bruk av budsjettet til et minimum. Følgende krav ble satt til utviklingen i løpet av denne oppgaven.

1. Plattformen skal selv navigere fra en startposisjon til en sluttposisjon
2. Plattformen skal detektere objekter i veibanen og kunne unngå disse.
3. Farten på plattformen må kunne reguleres.
4. Krav til overvåkning av systemet.
5. Krav til ulike brukergrensesnitt for å gi brukerne flere valgmuligheter.
6. Plattformen skal få et oppgradert brukergrensesnitt.

1.4 Metode

I planleggingsfasen var det mye arbeid med å finne komponenter som kunne brukes for å gjøre plattformen autonom. Enkelte av komponentene som var på plattformen fra før, måtte byttes ut for å løse målene til oppgaven. Videre begynte integrasjonen av komponentene. Da de ulike komponentene ble valgt og integrert i den bestående plattformen, kunne software utvikles. For å videreføre modulariteten i plattformen ble det valgt å bruke flere programmeringsspråk og styringssystemer. Testing ble gjennomført for å verifisere måloppnåelse og kapasiteten til plattformen.

1.5 Struktur

Oppgaven tar for seg en teoridel hvor ulike konsepter og komponenter brukt i prosjektet blir beskrevet. En konseptutvikling følger etter teoridelen, som er ment for å gi leser en forståelse av hvorfor oppgaven ble løst med disse komponentene. Videre fremstilles programmeringen og integreringen av komponenter som er blitt gjort for å gjøre plattformen autonom. Avslutningsvis drøftes det hvorvidt testene ble vellykket og hvilke lærdommer oppgaven har belyst.

2 Teori

Dette kapittelet tar for seg nødvendig teori for oppgaven som består av begrepsavklaringer og informasjon om komponenter og konsepter. Dette er gjort for å gi leseren en forståelse av teorien bak utviklingen av den autonome plattformen.

2.1 Autonomi

«Det finnes ikke noen felles definisjon for hva autonome systemer er» (Hareide, 2018). Ofte blandes det mellom automatiske, fjernstyrte og autonome systemer. Forskjellen mellom de ulike systemene er at et autonomt system kan tilpasse seg situasjoner som oppstår. I et fjernstyrt system vil operatøren være fysisk skilt fra systemet, men vil fremdeles kunne styre og ta beslutningene (Hareide, 2018). Denne oppgaven betegner autonomi som «ett system som kan vurdere, samt ta avgjørelser selv» (FFI, 2018). Dette betyr at man kan gi et autonomt system en oppgave hvor den tolker, planlegger og gjennomfører arbeidsoppgaven.

I tabell 2.1 har Sheridan og Verplank kommet med en rangering som tar utgangspunkt i hvor autonomt et system er, hvor nivå 1 er beskrevet som at systemet ikke er autonomt. Nivå 10 tar utgangspunkt i at datamaskinen bestemmer alt, uten innblanding fra mennesket.

Nivå	Forklaring
1	Mennesket tar alle beslutninger og systemet utfører basert på dette.
2	Datamaskinen tilbyr en fullstendig oversikt over beslutningsalternativene.
3	Datamaskinen gir forslag til beslutningsalternativer.
4	Datamaskinen gir forslag til et beslutningsalternativ. Mennesket bestemmer selv om dette skal utføres.
5	Dersom alternativ er godkjent av mennesket, vil datamaskinen utføre foreslått beslutningsalternativ.
6	Datamaskinen tillater mennesket veto i en begrenset tid før utførelse.
7	Datamaskinen utfører automatisk, for deretter å informere mennesket.
8	Datamaskinen utfører automatisk, og informerer mennesket ved forespørsel
9	Datamaskinen utfører automatisk, og informerer mennesket bare ved forhåndsprogrammert.
10	Datamaskinen bestemmer alt, uten menneskelig innblanding.

Tabell 2.1 - Sheridan og Verplanks nivå av automasjon (Hareide, 2018).

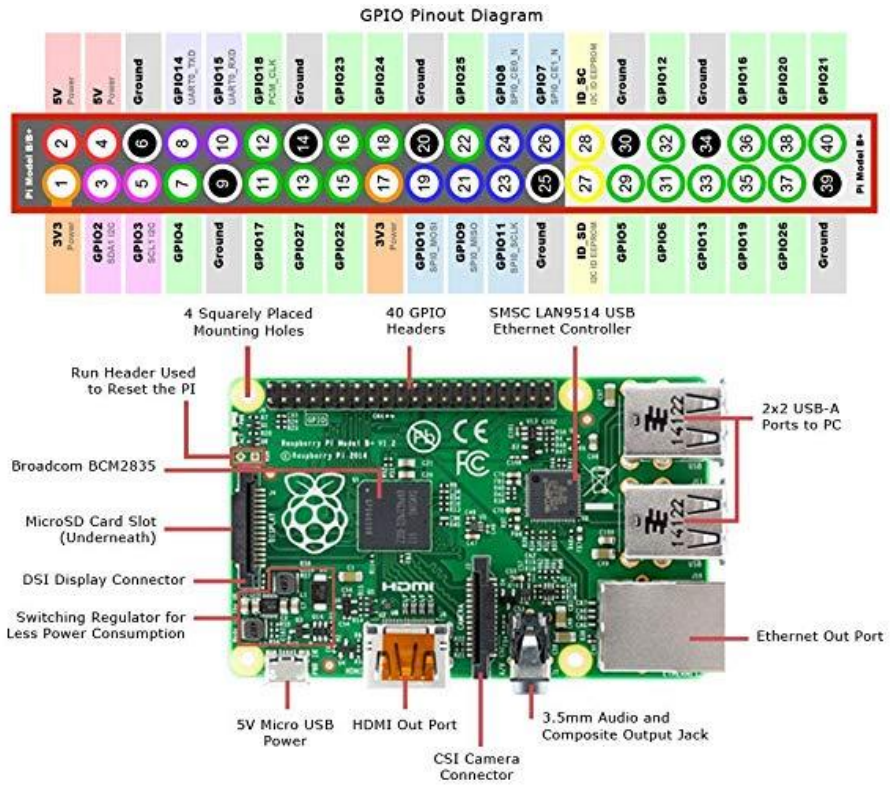
I en militær kontekst kan det tenkes at en selvkjørende bil enten kan frakte utstyr eller rekognosere et usikret område. Dette kan gjøres fjernstyrt, hvor bilen selv rekognoserer, som ved observasjon rapporterer tilbake til basen. En operatør kan overta styringen av bilen hvor det kan tenkes at datamaskinen gir beslutningsalternativ. Det kan argumenteres for at nivå fire vil være passende i en slik situasjon. Graden av autonomi avhenger av situasjon og kompleksitet (Hofoss, 2019). Dersom soldatene er avhengig av at teknologien skal virke, må graden av autonomi tilpasses slik at liv i verste konsekvens ikke går tapt.

2.2 Raspberry Pi

En Raspberry Pi er en liten datamaskin, som er basert på operativsystemet Linux. Raspberry Pi 3 model B+ er den nest nyeste utgaven på markedet. Den fungerer som hjernen og hovedbasen i plattformen vår, som er koblingsleddet mellom Node-Red og ROS, altså mellom styringsgrensesnittet og den automatiske beslutningstakeren.

En Raspberry Pi passer perfekt for våre arbeidsoppgaver i denne bacheloren. Den har åpen kildekode, som gjør at man kan utvikle egne programmer, for eksempel ROS pakker eller Navio2 systemer.

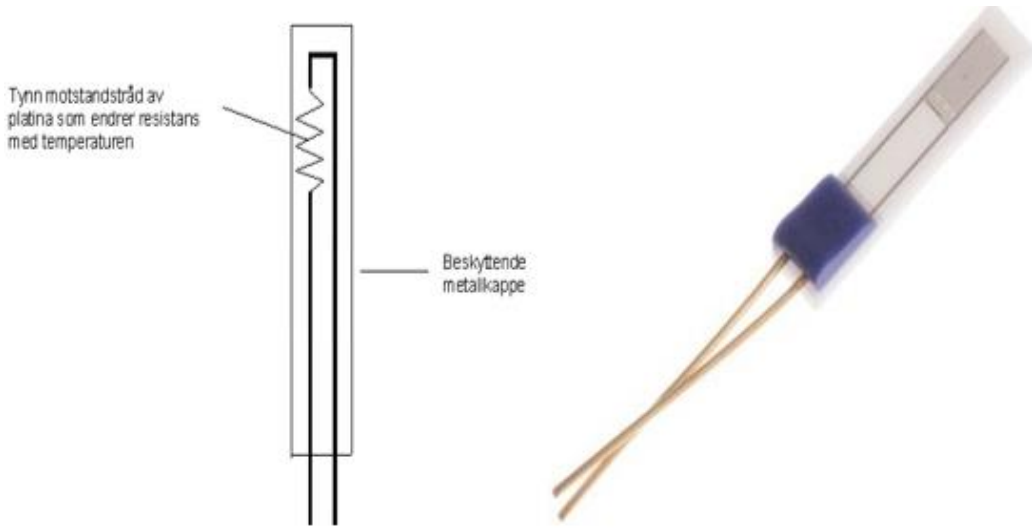
Hardware pakken til Raspberry Pi blir vist i figur 2.1. Figuren har en oversikt over hvilke komponenter en Raspberry Pi består av og funksjonen til de forskjellige GPIO pinsene.



Figur 2.1 - Oversikt over Raspberry Pi. Det finnes 40 tilkoblingspunkter som stort sett kan konfigureres fleksibelt (Carnino, 2016).

2.3 Temperatursensor

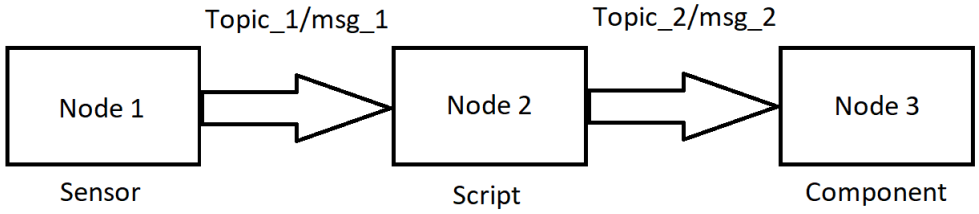
Siden oppgaven arbeider med elektriske elementer som genererer varme brukes temperatursensorer til å holde øye med de viktigste komponentene, nemlig motorkontrollene og hoverboard motorene. Temperatursensoren som brukes er av typen PT100-element. Denne består av en platina-basert motstand som endrer resistans med temperaturen. Ved måling 0 ºC er resistansen 100 Ω, derav navnet PT100. Fra 0 ºC har motstand og temperatur et tilnærmet lineært stigningstall på 0,385 Ω pr. ºC (Refvik, 2013). Sensoren har et temperaturområde mellom –70 ºC til 600 ºC. Disse temperaturmålerne ligger inne i hoverboardmotorene og inntil motorkontrollerene. De er så koblet opp mot PLSen, som leser av og tolker motstandsverdien som temperatur. Figur 2.2 illustrerer en PT100 sensor.



Figur 2.2 - Temperatursensoren PT100 (elektrofag.info, 2019), (RS-online, 2019).

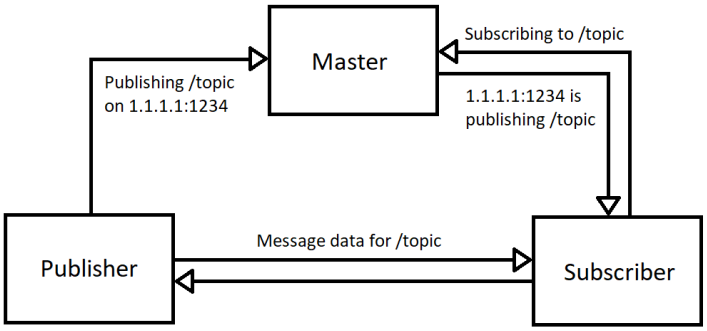
2.4 Robot Operating System (ROS)

Det ble besluttet å bruke programvaren Robot Operation System (ROS) i videreutviklingen av plattformen. ROS er et program som tillater «noder» å kommunisere med hverandre, man kan si at ROS er en infrastruktur til delprogrammer. En «node» kan være en sensor, en mekanisk komponent eller et script. Nodene kommuniserer med hverandre ved hjelp av «topics». Meldingen som sendes ut under et topic kan leses av andre noder, som igjen kan sende ut ny informasjon på samme eller andre topic.



Figur 2.3 - Noder kommuniserer ved hjelp av topics. Her kommuniserer Node_1 med Node_2 via Topic_1 og Node_2 med Node_3 via Topic_2 (Huang, 2016)

Figur 2.3 viser hvordan nodene 1-3 kommuniserer. Node 1 sender ut en melding under «Topic_1». Node 2 abonnerer på topic «Topic_1», som betyr at Node 2 ser alle meldingene som blir publisert under topic'et. Node 2 bruker informasjonen fra Node 1 og sender ut en melding under «Topic_2». Node 3 abonnerer på topic med navn «Topic_3», som betyr at Node 3 kan se meldingene som legges ut under dette topic'et.



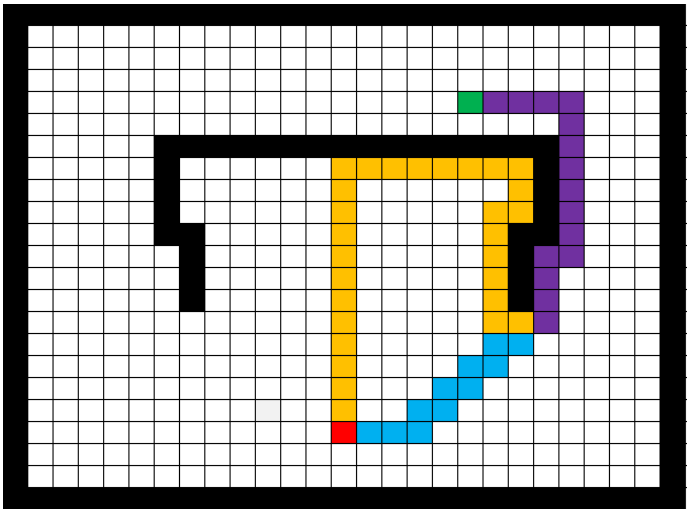
Figur 2.4 - Viser hvordan Master-noden fungerer (Huang, 2016)

Det er «Master»-noden som formidler informasjon mellom alle de andre nodene. Når en node starter å publisere til et topic sier denne noden, ofte kalt «publisher», til «master»-noden hvilket topic den publiserer, samt sin adresse. Når en annen node, ofte kalt «subscriber» vil abonnere på et topic, gir den beskjed til «master». Dersom noen allerede publiserer på topic'et vil «master»-noden gi denne adressen til «subscriber»-noden. «Subscriber»-noden oppretter en direkte kommunikasjonskanal med «publisher»-noden og mottar da samtlige meldinger som blir sendt under det gjeldene topic'et fra «publisher»-noden.

Flere noder kan publisere og abonnere på et topic, hvor hver enkelt «subscriber»-node oppretter en kommunikasjonskanal med hver enkelt «publisher»-node. «Master»-noden vil fortløpende gi informasjon til «subscriber»-nodene dersom nye noder begynner å publisere under et topic. Slik sørger «master»-noden for at alle «subscriber»-nodene får den informasjonen de trenger.

2.5 Pathfinding

«Pathfinding» defineres som den plottingen en datamaskin finner for den raskeste ruten mellom to punkter (Wikipedia 2019). Figur 2.5 viser en situasjon der datamaskinen må finne den raskeste veien mellom start og mål. Start vil i alle figurene i delkapittel 2.5 være representert som rød og mål vil være representert som grønn.

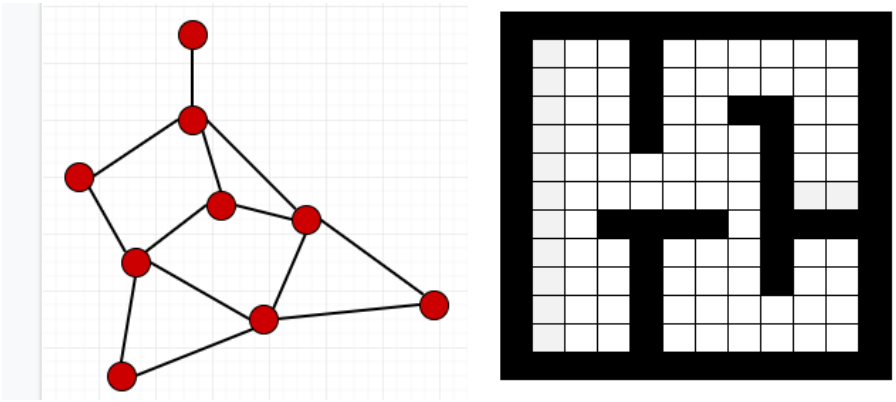


Figur 2.5 - Illustrasjon av hvordan pathfinding algortimer fungerer (Patel, 2019)

Når denne oppgaven omtaler ordet "punkt" i sammenheng med pathfinding, menes det et punkt eller en posisjon på et kart man kan bevege seg til. Disse kartene kan ha forskjellig oppbygning, som illustrert i figur 2.6.

Hvordan datamaskinen kalkulerer den rette veien kommer an på hvilken algoritme som blir brukt. Den rette veien kan være den korteste veien, den raskeste veien eller noe helt annet. Det kommer an på parameterne som blir satt i koden. Det finnes uendelige mange algoritmer som kan brukes for å kalkulere disse veiene. Problemet illustrert i figur 2.5 viser et objekt som skal bevege seg fra start til mål (Patel, 2019). Objektet skanner i dette tilfellet kun et lite område rundt seg, men vet hvor det er i forhold til målet. Et oppstående problem dersom man kun bruker retningsbasert bevegelse og ikke en pathfinder kan være at objektet tar den oransje veien og ikke den blå. Man kan se at den blå veien er raskere enn den oransje. Uten en pathfinder vil man bevege seg mot målet til man møter en hindring. Deretter vil denne hindringen passeres før man fortsetter mot målet. En pathfinder kan hjelpe deg med dette, ved å se hindringene før man kommer til de, slik at man kan velge den veien som møter minst hindringer i første omgang. Dette er fordi en pathfinder vurderer hele det kjente miljøet rundt seg og ikke kun deler av det når den skal avgjøre hva som er den beste veien.

Så og si alle algoritmene har enkelte fellestrekk. De er bygget opp av punkter i et kart. Disse punktene er «steder» man kan bevege seg mellom. Videre har de også to lister. Disse er henholdsvis den åpne - og den lukkede listen. I den åpne listen ligger alle punktene som er naboer til de punktene som allerede har blitt analysert av algoritmen. I den lukkede listen ligger alle punktene som har blitt analysert. Dette gjør at algoritmen enkelt holder oversikt over hvilke punkter som har blitt analysert, samt gjør det lettere å velge hvilke punkter den skal se på i neste analyse. Når en algoritme skal velge hvilket punkt den skal se på, velger den et punkt fra den åpne listen. Algoritmen undersøker punktet, legger det til i den lukkede listen og fjerner det fra den åpne listen. Videre legger algoritmen alle de punktene rett ved siden av det punktet som nettopp har blitt sett på, inn i den åpne listen. Ved å gjøre dette vet algoritmen hvilke punkt som er gjennomsøkt og hvilke punkter den skal fortsette med. Videre er det opp til hver enkelt algoritme og implementeringen av den, hvilket punkt fra den åpne listen man skal se på som neste punkt.

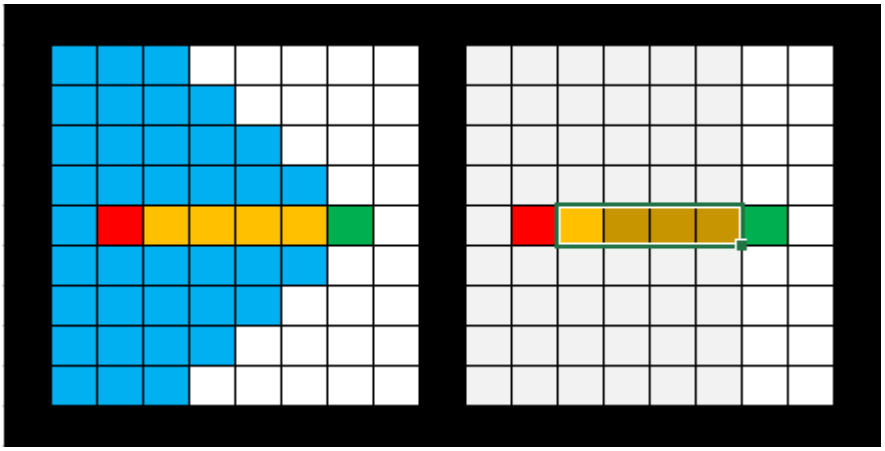


Figur 2.6 - Ulike typer kart

Figur 2.6 illustrerer to forskjellige typer kart. Teksten vil referere til "nodekartet" til venstre som figur 2.6.A og rutenettet som figur 2.6.B. Figur 2.6.A er et kart med punkter som er koblet sammen på en litt tilfeldig måte og figur 2.6.B er et rutenett. Når algoritmen ser på et punkt og skal kalkulere hvilke punkter som skal legges til i den åpne listen ser den på to ting. Den første er hvor mange, og hvilke punkter som er rett ved siden av det punktet den for øyeblikket ser på. Den andre er hvorvidt man kan bevege seg på dette punktet. Figur 2.6.B viser et rutenett der alle punktene, med unntak av sidene har 4 punkter rett ved siden av seg. Disse fire er henholdsvis over, under, til høyre og til venstre for punktet. Dersom en rute i rutenettet er hvitt kan man bevege seg på det og hvis det er svart, illustrerer dette en vegg, altså et punkt man ikke kan bevege seg på. Figur 2.6.A viser et nettverk av punkter. I dette nettverket varierer det hvor mange "naboer" hvert punkt har. I disse to eksemplene er det mellom ett til fire punkter som må legges til i den åpne listen før man gjør seg ferdig med et punkt.

2.5.1 Dijkstras algoritme og Best-First-Search

Dijkstras algoritme garanterer å finne den beste veien, det kan være flere, til målet (Patel, 2019). Hvilke kriterier man skal vurdere de ulike veiene opp mot kommer an på bruksområdet til systemet. Algoritmen tar utgangspunkt i objektets startposisjon. Deretter ser den på om punktet som befinner seg nærmest start kan bli krysset av objektet og om det ikke har blitt søkt igjennom enda. Algoritmen søker hele tiden gjennom det punktet som er nærmest startpunktet, helt til det finner målet eller har søkt igjennom alle punktene det er mulig å gå til. Dette er en algoritme som passer utmerket dersom man må finne den absolutt beste veien, uten å ta hensyn til hvor mye datakraft man trenger. Det faktum at algoritmen må sjekke samtlige punkt som er nærmere startposisjonen enn målet gjør naturligvis at man søker gjennom en god del unødvendige punkter. Dette er illustrert av figur 2.7.

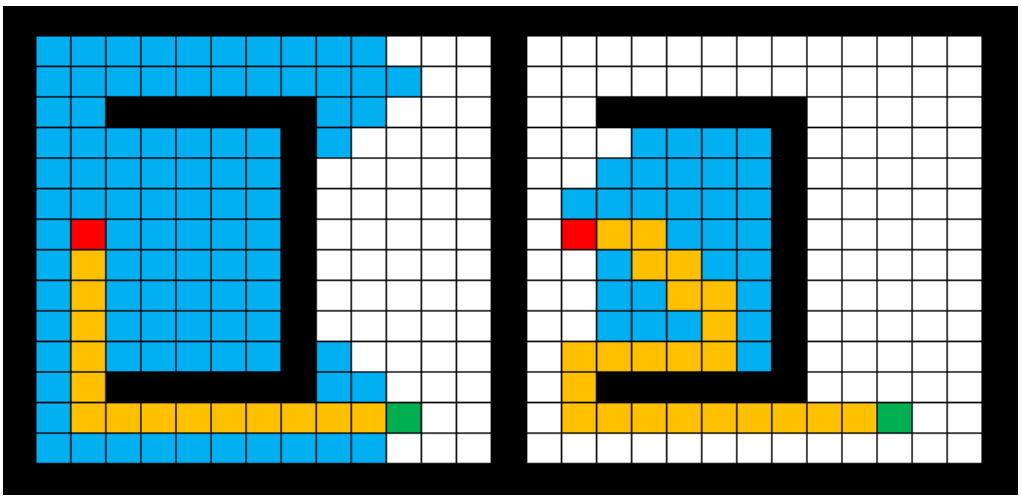


Figur 2.7 - Illustrasjon av svakhetene til Dijkstra (Patel, 2019)

Figur 2.7 viser at algoritmen, til venstre, søker igjennom cirka 10 ganger flere punkter enn en vektet algoritme som for eksempel A* ville trengt. A* vil bli forklart i kapittel 2.5.2. Det tar nødvendigvis tid å kalkulere og dersom dette er kritisk for din anvendelse av pathfinding er kanskje ikke Dijkstras algoritme det du trenger.

«Best-First algoritmen fungerer på en lignende måte som Dijkstra, men tar utgangspunkt i målet og ikke i starten» (Patel, 2019). Dette betyr at den starter fra startpuntet og ser på det punktet som har en allerede kartlagt vei til start, er nærmest mål, kan bli krysset av objektet og som ikke har blitt søkt enda. En fordel med dette er at den finner veien mye raskere siden den ikke trenger å søke gjennom like mange punkter. Svakheten til denne

algoritmen er at den ikke nødvendigvis finner den raskeste veien til målet dersom det er noen hinder i veien. Dette er illustrert i figur 2.8.

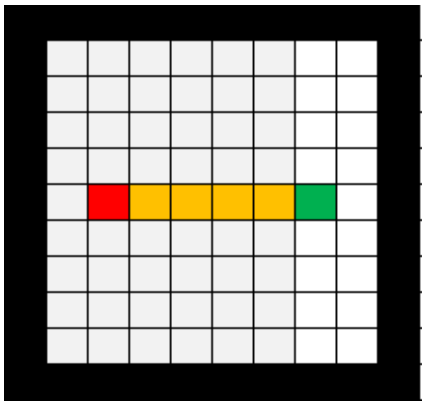


Figur 2.8 - Forskjellen på Dijkstra (venstre) og Best-First (Høyre) (Patel, 2019)

Figur 2.8 viser at Best-First algoritmen er grådig. Algoritmen tar kun høyde for hvor langt det er til målet og ikke hvor langt den må gå for å komme seg dit. Dijkstras algoritme tenker helt motsatt, den tenker kun på hvor langt den har beveget seg. Når Dijkstra har funnet målet jobber den seg tilbake til start via den raskeste veien. Ved å kombinere disse to egenskapene kan man konstruere en algoritme som både tar høyde for hvor langt den må bevege seg og hvor målet er. Dermed slipper man å søke gjennom unødvendige punkter. A* er en algoritme som gjør nettopp dette.

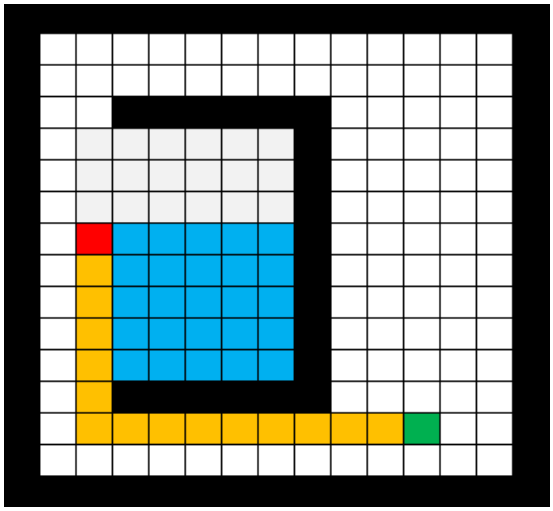
2.5.2 Algoritme A*

A* algoritmen er like rask og like bra som Best-First på de enkle oppgavene, grunnet bevegelser mot målet når den søker. Dette er illustrert i figur 2.9.



Figur 2.9 - A* på enkle oppgaver (Patel, 2019)

Videre tar algoritmen også hensyn til hvor langt den har gått for å komme til et gitt punkt. Dette er likheten den har med Dijkstras algoritme. Resultatet av A* på en oppgave hvor det er et hinder mellom start og mål, er illustrert i figur 2.10.



Figur 2.10 - A* på mer komplekse oppgaver (Patel, 2019)

En sammenligning av figur 2.10 og figur 2.8 viser at A* er raskere enn både Dijkstras algoritme og Best-First. Videre vises det at A* finner den beste veien. Dette er noe Dijkstras algoritme også klarer, mens Best-First ikke klarer, dersom oppgaven er kompleks. A* algoritmen klarer dette fordi den ser på både den eksakte kostnaden for å komme seg til

et punkt, representert som g(n). Samt en estimert kostnad for å komme seg til mål, representert som h(n). Algoritmen ser på summen av allerede brukt kostnad og et estimert videre bruk. Dette følger ligningen:

$$f(n) = g(n) + h(n)$$

Når algoritmen skal velge hvilket punkt den skal se på velger den det punktet med den laveste verdien for f(n) og som ligger ved siden av et punkt den allerede har sett på. På denne måten slipper algoritmen å se igjennom like mange punkt som enkelte andre algoritmer. Når den har funnet målet jobber den seg tilbake fra målet og mot starten. Algoritmen ser nå kun på g(n) og velger alltid det punktet med lavest g(n) som neste punkt. Algoritmen fortsetter med dette til den har kommet tilbake til start. Den veien algoritmen nå har funnet vil alltid være den beste veien mellom de to punktene.

2.6 Kvaternioner

Kvaternioner er en måte å illustrere vinkler på. Dette er nevnt i teoridelen fordi ROS noder ofte bruker kvaternioner som standardenhet for vinkler siden roboter ofte opererer i flere enn to dimensjoner. På samme måte som de komplekse tallene kan betraktes som en utvidelse av de reelle tallene, kan også kvaternioner betraktes slik. Hos kvaternionene har man tilføyet elementene i, j og k. Disse elementene oppfyller ligningene:

$$i^2 = j^2 = k^2 = ijk = -1$$

Multiplikasjonen av elementene er assosiativ, noe som medfører at man får følgende relasjoner:

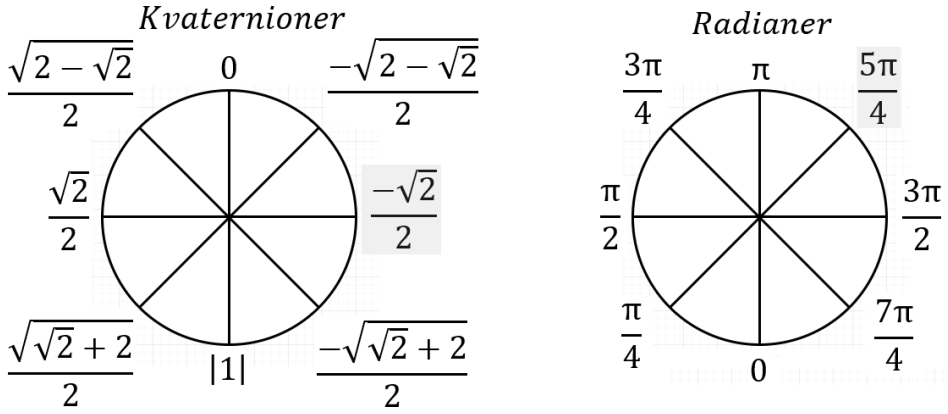
$$ij = k, \qquad ji = -k$$
$$jk = i, \qquad kj = -i$$
$$ki = j, \qquad ik = -j$$

Kvaternioner blir deretter regnet over til radianer. Omregning mellom kvaternioner (K) og radianer (R) følger formlene:

$$\cos\left(\frac{R}{2}\right) = K$$

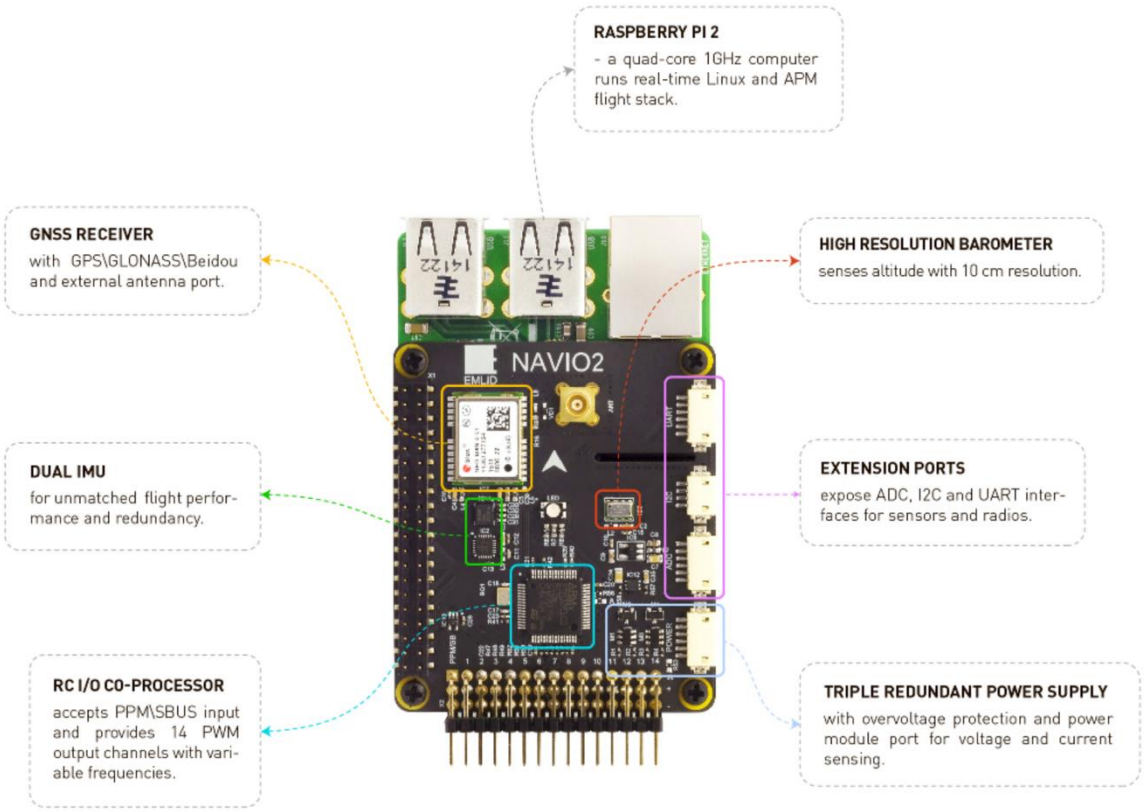
$$cos^{-1}(K) \times 2 = R$$

Dette fører til at verdiene for de respektive «enhetssirklene» blir:



Figur 2.11 - Enhetssirkler for Kvaternioner og Radianer sett med Nord som opp

2.7 Navio2



Figur 2.12 - Oversiktsbilde over Navio2 på en Raspberry Pi (Emlid, 2015).

Figur 2.12 illustrerer en Navio2, som er en HAT for Raspberry Pi. Det vil si at det er en sensor pakke som er kompatibel med bruk av styringspakker fra Ardupilot, som er et åpen kildekode autopilot program. Navio2 undersøkes som mulighet til styring av plattformen ved bruk av GPS-signaler og Mission planner. Denne programpakken kan styre plattformen fra A til B utendørs, ved bruk av en planlagt rute. Den kan komme seg forbi hindre ved bruk av bevegelsessensorer på sidene, som var inkludert i den opprinnelige konstruksjonen til plattformen. Navio2 bruker Linux som operativsystem, som gjør det enkelt å modifisere.

Sensorpakkene ved Navio2 består av en GPS-sensor for posisjonering, samt en bevegelsessensor, akselerometer og magnetometer. I tillegg finnes det en bevegelsesprosessor og et barometer for høydemåling. Sensorpakken er mest brukt til å fly droner og er dermed mer designet for flyging enn kjøring med rover.

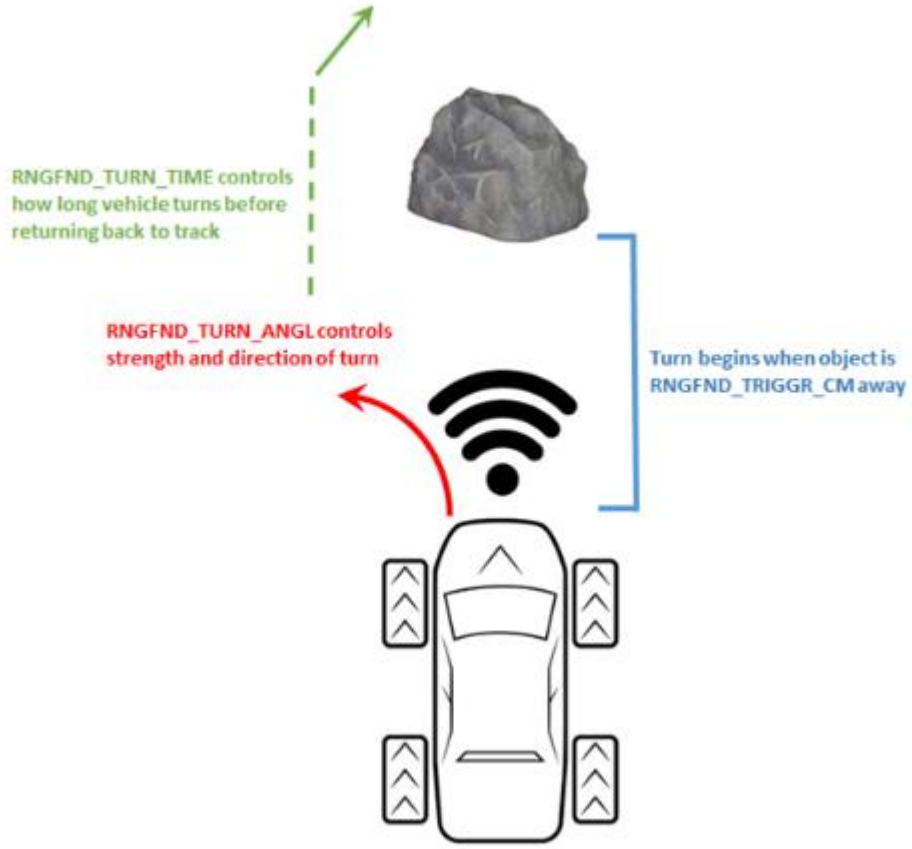
2.8 Ardupilot (Mission Planner)

Ardupilot er en programmeringsplattform for utvikling av autonome ubemannede kjøretøy. Koden er åpen for alle og under konstant oppdatering fra forskjellige brukere rundt om i verden. Systemet bygges opp med en sensorpakke som hardware, eksempelvis Navio2 eller Pixhawk og software som Ardupilot og mission planner. Sistnevnte er et brukergrensesnitt som kan kjøres fra datamaskin eller andre mobile enheter som støtter Ardupilot. Mission planner er illustrert i figur 2.13, hvor det øverste bilde viser GPS-bildet du får ved tilkobling av enhet og det nederste viser et eksempel på en planlagt rute.

Figur 2.13 – Bilde øverst viser GPS posisjonering, mens under vises GPS mapping.

Her kan man planlegge og følge plattformen på et kart ved oppdatering av GPS-signalene. Videre kan man legge inn en rute på hvor plattformen skal kjøre. Den vil da følge oppsatt rute for videre å stoppe ved sluttpunktet som illustrert nederst i figur 2.13. Ultralyd

sensorer kan brukes for å oppdage objekter i veibanen til plattformen og til å endre retning. Figur 2.14 illustrerer et eksempel ved bruk av denne metoden.



Figur 2.13 - Tiltenkt styring ved implementering av Ardupilot (Ardupilot Dev Team, 2018)

3 Konseptutvikling

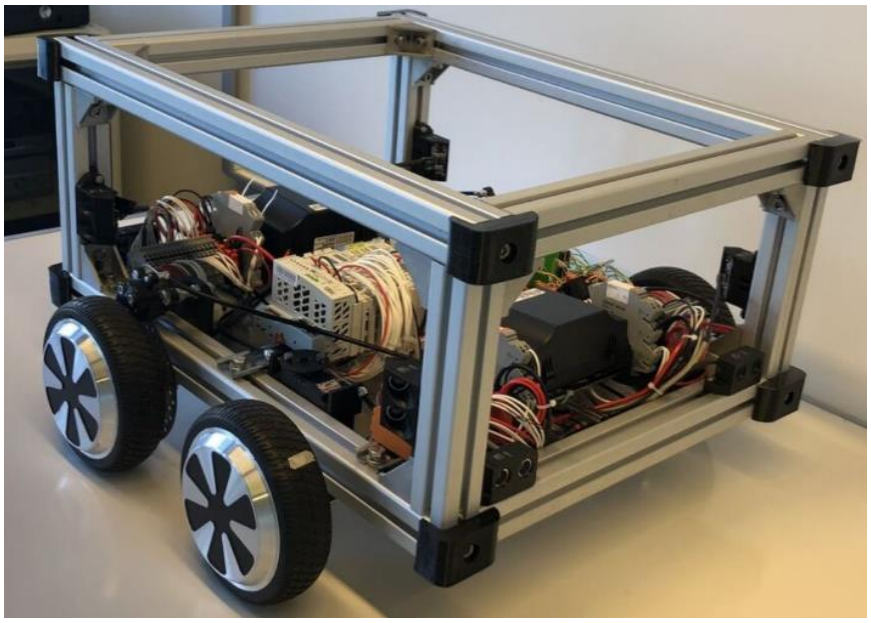
I startfasen av et hvilket som helst prosjekt er det viktig å utvikle et konsept. Under utvikling av oppgavens konsept var det flere viktige valg som måtte tas. Hvilket utstyr skulle brukes og hva er hensikten med disse komponentene. Oppgaven skal nå gå nærmere inn på de viktigste valgene.

3.1 Plattform

Da det kom til valg av plattform handlet det om hva som skulle gjøres og hvordan det kunne bli gjort på enklest mulig måte, i den hensikt å kunne gjøre plattformen autonom.

Det ble vurdert å lage en undervannsdrone eller en rover. En faktor som spilte inn var hvilken type sensor valget falt på. Her falt valget på Lidaren, som egner seg bedre for en rover. Lidar fungerer i et plan og er framtidsrettet når det kommer til selvkjørende biler. Bruksområde for plattformen kan for eksempel være å redusere tiden det tar for ansatte å flytte på materiale. Inspirasjon ble hentet fra lager-roverne til Amazon. De blir brukt i et rutenettsystem for å frakte rundt på forskjellige varer som blir sendt ut fra hovedlageret deres daglig (Simon, 2019).

Valget falt dermed på rover og etter anbefaling fra Alexander Sauter, vår faglærer i transmisjonsteknikk og ansatt på skolen i teknologiavdelingen, ble det ansett som fordelaktig å kunne bygge på en tidligere bacheloroppgave. I figur 3.1 er plattformen avbildet.



Figur 3.1 - Opprinnelige plattform (Flaatten og Tande, 2018).

Basert på førsteinntrykket virket plattformen god nok som utgangspunkt for denne oppgaven. Plattformen hadde noen komponenter som trengte å erstattes eller oppgraderes for å tilfredsstille kravene som er stilt i denne oppgaven.

Siden plattformen var bygget som en tidligere bachelor var det enklere å tilpasse hardware i den. I motsetning til å kjøpe en ferdigbygd plattform, som det hadde vært vanskeligere å bytte deler i. Den opprinnelige oppgaven hadde også dokumentert produksjonsfasen godt, så det var lett å sette seg inn i strukturen på hardwaren til plattformen.

3.2 Fremdriftssystem

For å kunne styre motorkontrollene brukes et programmeringsverktøy som binder de ulike enhetene sammen, slik at motoren kan få ulike verdier som endrer posisjonen til hoverboard motorene. De ulike komponentene er valgt basert på undervisning på Sjøkrigsskolen. Det naturlige valget ble Node-Red, da dette programmeringsverktøyet er kompatibelt med enhetene i plattformen. Neste steg er å velge hvordan informasjonen skal sendes fra Node-Red til hoverboard motorene. Valget falt på en Arduino siden det er en robust måte å sende informasjonen på.

I bacheloroppgaven "Testplattform for autonome systemer" (Flaatten og Tande, 2018) står det beskrevet i kapittel 5, konklusjon med anbefaling, at den største begrensningen i plattformen er motorkontrollene. "De har god evne til å styre de motorene som er i bruk, men akselerasjonskurvene er ikke justerbare" (Flaatten og Tande, 2018). Det viste seg at de opprinnelige motorkontrollene BLDC Motorkontroller, JYQD_V7.3E2, ikke hadde en reaksjonsevne hurtig nok til å bremse for å tilfredsstille behovene. På grunnlag av dette ble det ansett som nødvendig å oppgradere motorkontrollene til en bedre type.

Under valg av motorkontroller, ble det satt noen kriterier. Siden plattformen skal kunne kjøre autonomt, brukes rolig gangfart. Oppgaven krever at reaksjonstiden til motorkontrolleren er hurtig, grunnet det autonome aspektet. I tillegg kreves en kontroller som er kraftig nok til å håndtere motorene. En motorkontroller som støtter blant annet PBM og general purpose digitale og analoge innganger var også ønsket.

Valget falt på Odrive sin motorkontroll, da disse er relativt enkle å benytte seg av, og gir oss tilgang til ulike nåtids variabler, eksempelvis posisjon og hastighet. Det finnes to forskjellige versjoner med ulikt input spenningsområde på henholdsvis 12-24 volt og 12-56 volt. Siden batteriene i plattformen er på 36 volt måtte det kjøpes 12-56 volt.

Softwarepakken til Odrive er åpen kildekode som gjør at den hele tiden er i utvikling. I tillegg kan kontrolleren være i ulike moduser som eksempelvis posisjon og fartskontroll. Fartskontroll modusen ble brukt i vår oppgave. Dette kan endres på, dersom det heller er ønskelig å bruke posisjonskontroll.

En annen fordel med motorkontrollen er at man kun trenger to stykk for å kontrollere alle fire motorene, gjennom M1 og M0 inngangene. Disse fungerer uavhengig som gjør at man kan styre hjulene slik man måtte ønske. Motorkontrollene er dyrere enn de opprinnelige, men man får en mer robust kontroller som gjør at blant annet sikkerheten blir bedre.

Under valg av motor ble to forskjellige typer vurdert, nemlig DC - og børsteløse DC motorer. Fordelen med DC-motorer er at de er relativt billige og enkle å håndtere. Denne likestrøms motoren har et roterende anker med viklinger. For at strømmen skal komme frem brukes "børster". Problemet med disse børstene er at de slites. På grunn av dette dannes det en økende motstand, som gjør at de må vedlikeholdes (Motion control online marketing team, 2017).

Den børsteløse motoren har mindre friksjon da det ikke er noen børster som må slepes over viklingene i statoren. Det gjør at det er minimalt med vedlikehold på de børsteløse motorene.

Siden det allerede var montert fire hoverboard motorer på plattformen, ble det bestemt å bruke disse. Se kapittel 4.3 fremdriftssystem for detaljert informasjon om motorene.

3.3 Sensorer

Under valg av hvilke sensorer plattformen skulle bruke, ble det sett på hvilke sensorer som passet til bruk i autonome prosjekter og hvilken omtale disse hadde fått i ulike media. Det ble vurdert tre ulike systemer som kunne integreres i plattformen.

Figur 3.2 - Ultralyd sensor HC-HC-SR04

Mulighetene til å beholde ultralyd sensorene ble også vurdert, som opprinnelig bestod av åtte sensorer av typen HC-HC-SR04 som sender ut ultralyd pulser. Dersom denne pulsen treffer et objekt innenfor sin rekkevidde og pulsen sendes tilbake til sensoren, vil den kunne måle avstanden til objektet ved bruk av formelen:

$$strekning = \frac{fart * tid}{2}$$

Denne kan derimot være utilregnelig, ettersom en uheldig vinkel kan gjøre at den aldri vil returnere tilbake til sensoren. Dette vil gjøre at avstanden til objektet blir unøyaktig. På bakgrunn av dette, og i kombinasjon med at ultralyd sensorene ikke oppfylte kravene våre til sensorpakken, ble det besluttet å erstatte disse. Løsningen ble å ha en sensor som dekket 360º rundt plattformen.

Etter å ha utelukket ultralyd sensorene fantes det to mulige systemer, Lidar og Radar. Radaren fungerer på samme måte som ultralyd sensorene. Sensorene definerer hvor langt unna et objekt er ved hjelp av refleksjon av radiobølgene. Ulempen med å bruke radar på denne måten er at den har en lang bølgelengde som gir dårlig oppløsning på bilde av objektet.

I bacheloroppgaven "Testplattform for autonome systemer" står det i drøftingsdelen av oppgaven at det anbefales å implementere en Lidar i større prosjekter som 3D-mapping og beslutningssystemer (Flaatten og Tande, 2018). Den fungerer på samme måte som ultralyd sensorene men istedenfor å bruke lydbølger brukes lysbølger. Lidaren har kortere

bølgelengde som gjør at den kan detektere mindre objekter. Det ble besluttet å undersøke Lidaren nærmere som resulterte i fire nøkkelfordeler. Den genererer store mengder målinger og kan være nøyaktig ned til cirka en centimeter. I tillegg kan Lidar dataen brukes til å generere 3D-kart for å tolke miljøet rundt. Den klarer seg også bra under lave lysforhold. Sett i en militær kontekst kan dette være gunstig dersom man ønsker å sende plattformen inn for å søke et rom hvor det er lav belysning. På bakgrunn av dette falt valget på Lidaren.

Sensoren er av typen RPLidar A1M8, som er en 360° laser skanner med en rekkevidde på 12 meter, og har en modul som spinner på toppen via en DC motor. (TechMesh, 2018).

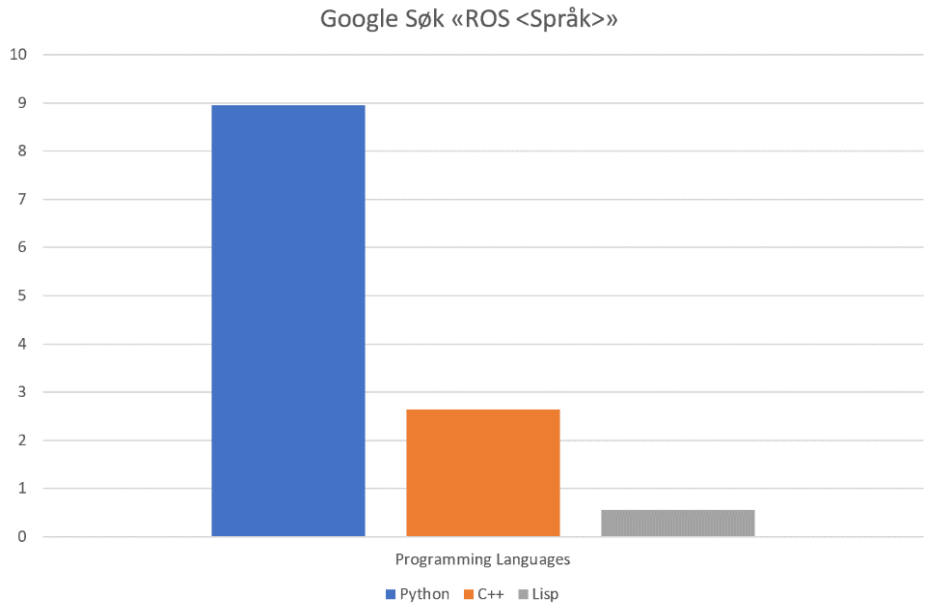
Lidaren kan gjøre 8000 målinger per sekund, i tillegg til at man kan kontrollere hvor fort motoren kan spinne modulen på toppen. Dette gjør at man kan regulere hvordan oppløsningen på bilde blir.

3.4 System for utvikling av autonomt konsept

Tidlig i oppgaveprosessen var det naturlig å finne ut hvilken programvare som skulle bli benyttet. Når søket etter programvare for roboter startet var det egentlig bare et bra alternativ. Dette alternativet var Robot Operating System (ROS), se kapittel 2.4. ROS har et enormt samfunn som støtter opp under utviklingen av programvaren. ROS er åpen kildekode og tilrettelegger for implementering av egen kode. ROS har et robust kommunikasjonssystem som gjør det enkelt for programmer å snakke sammen. Dette gjelder uavhengig i om det er egen eller andre sin kildekode.

3.5 Programmeringsspråk

ROS brukes, som forklart i kapittel 2.4, for å opprette kommunikasjonskanaler mellom de ulike delene eller «nodene» innad i et system. For å lage disse nodene må man lage scripts eller programmer for å utføre diverse handlinger. «ROS støtter Python, C++ og Lisp for utvikling av nodene» (ROS wiki, 2018).



Tabell 3.1 - Antall søkeresultater for de respektive programmeringsspråkene

Det ble valgt å ikke gå for Lisp. Dette er fordi ingen av oss har noen forkunnskap om språket. Det er det desidert minst brukte programmeringsspråket i ROS, samt at verken Tellez eller Lin nevner dette når de skriver om hvilket språk som er best i hvilken sammenheng. Da gjenstår to muligheter, Python og C++. Tabellene 3.2 og 3.3 viser en liste med fordeler og ulemper ved begge språkene og utvikling i ROS.

Python

Fordeler	Ulemper
Tar kort tid å lage en prototype.	Det ferdige programmet kjører sakte.
Tar kort tid å finne feil.	Større sannsynlighet for at programmet krasjer under testing.
Tar kort tid å lære	
Små programmer for komplekse noder.	
Lett å forstå/tolke den ferdige koden.	

Tabell 3.2 - Fordeler/Ulemper ved Python (Tellez, 2019)

C++

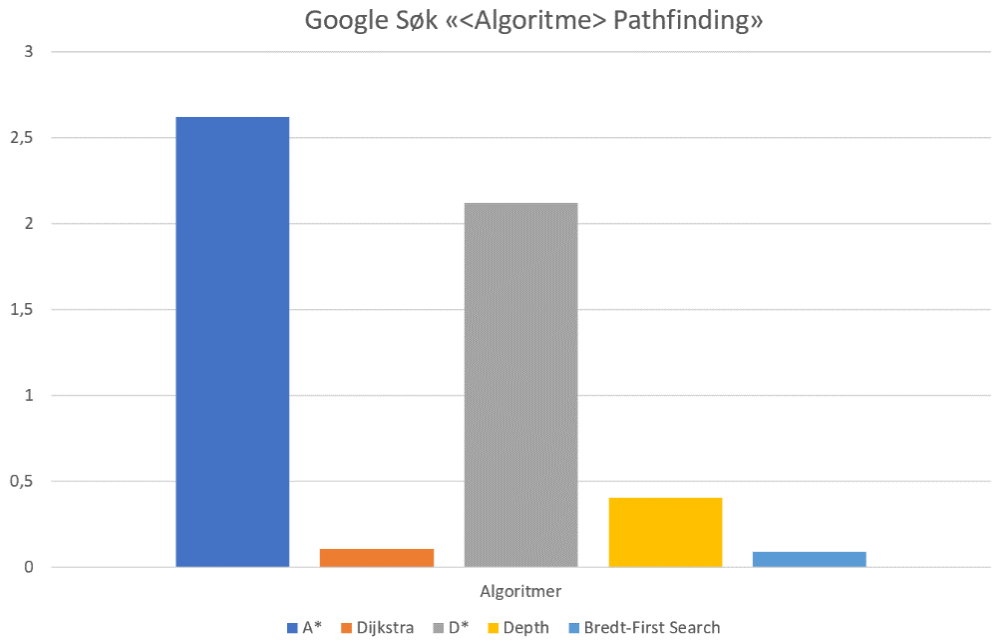
Fordeler	Ulemper
Det ferdige programmet kjører raskt	Vanskeligere kode å lære seg
Liten til null sannsynlighet for at programmet krasjer under testing	Lange programmer å skrive, selv for små noder
Uendelig mengde biblioteker for import	Vanskelig å forstå/tolke den ferdige koden
	Kan ta lang tid å feilsøke.

Tabell 3.3 - Fordeler/Ulemper ved C++ (Tellez, 2019)

Lin er for det meste enig med Tellez. «C++ er det beste, samt industristandarden for robotutvikling og ROS» (Lin, 2018). Lin får støtte av Tellez i dette utsagnet. De er begge enige i at man burde starte med Python, siden dette er mye enklere å lære seg, samt lage noder med. «Dersom målet ikke er å oppnå industristandardene, men å bruke ROS for akademiske grunner vil Python være bedre» (Lin, 2018). Valget falt derfor på å bruke Python for utviklingen i ROS. Dersom det skulle være nødvendig med et raskere script i en node vil det, grunnet kommunikasjonsstrukturen i ROS lage denne noden i C++.

3.6 Pathfinding algoritme

I denne oppgaven blir det sett på noen av de mest populære algoritmene tilgengelig for allmennheten i den hensikt å finne den som passer best for vårt formål. Det ble gjennomført et kjapt google søk, som bestod av å søke på de ulike algoritmene med ordet «pathfinding» bak. Dette hjelper med å velge en algoritme som tidligere er blitt skrevet mye om. Når noe er skrevet mye om finnes det naturlig vis flere svar på spørsmål som kan dukke opp på et senere tidspunkt. Resultatet av søket vises i tabell 3.4.



Tabell 3.4 - Antall søkeresultater for de respektive algoritmene

Algoritmen Depth er mest brukt i søk av tre eller graf-lignende strukturer. Dersom dette skulle blitt implementert ville et vektorbasert kartsystem være nødvendig. Dette er siden Depth algoritmen velger en "gren" og søker gjennom den helt til den finner målet eller alle undergrenene er gjennomsøkt. Denne algoritmen er heller ikke mye brukt i pathfinding i en åpen verden.

D* algoritmen jobber på en lignende måte som A*. Den største forskjellen er at A* hele tiden rekalkulerer hele veien fra start til mål. D* legger til nye objekter den observerer i kartet den allerede har. Dersom noen av de nye objektene er der den tidligere veien var, rekalkulerer D* denne delen av veien. Og dersom objekter forsvinner fra et område D* anser som raskere enn en tidligere vei, vil den også rekalkulere denne delen av veien. D* er altså raskere enn A* dersom miljøet blir større og mer komplisert.

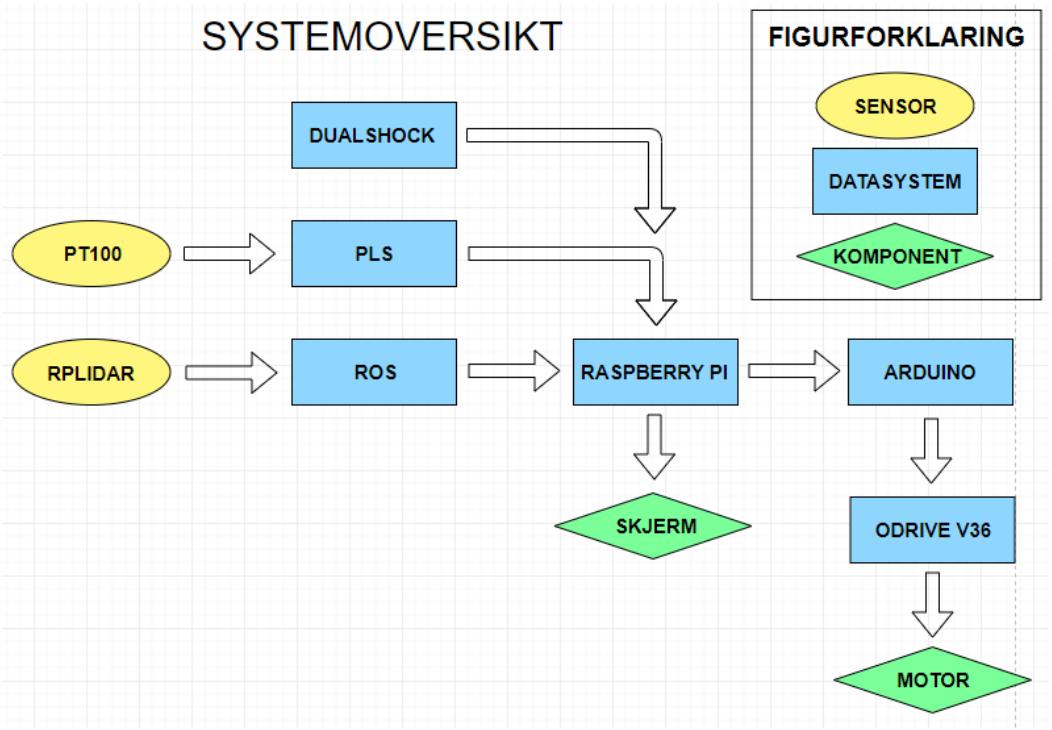
Som man ser ut fra resultatet er det A* og D* som kommer best ut av dette søket. Dersom man har lest litt om pathfinding på nettet, eventuelt sett en informativ video på for eksempel Youtube har man nok mest sannsynlig også hørt om A* eller Dijkstra. Med litt mer innsikt i hver av algoritmene, samt hvor lett man kan finne informasjon om de ulike algoritmene falt beslutningen på å benytte algoritmen A* i oppgaven.

4 Implementering

I dette kapittelet vil informasjon om implementeringen av de ulike komponentene og undersystemene bli presentert.

4.1 Systemoversikt

Plattformen kan styres av tre forskjellige styringssystemer. Det første systemet er en Sony Playstation 4 kontroller. Det andre styringssystemet er via PLS, hvor man kan styre plattformen ved hjelp av knapper på en nettside. Man kan koble seg på nettverket "BOAT" og bruke dette styringssystemet fra for eksempel en pc eller en mobil. Det tredje styringssystemet er ved bruk av ROS. Det er et delvis autonomt system som kan føre plattformen fra punkt A til punkt B uten videre menneskelig involvering. Hvilket styringssystem man ønsker å bruke kan velges fra styresiden til PLS nettsiden, men plattformen er programmert til å starte med PS4 som oppstartstyringssystem.



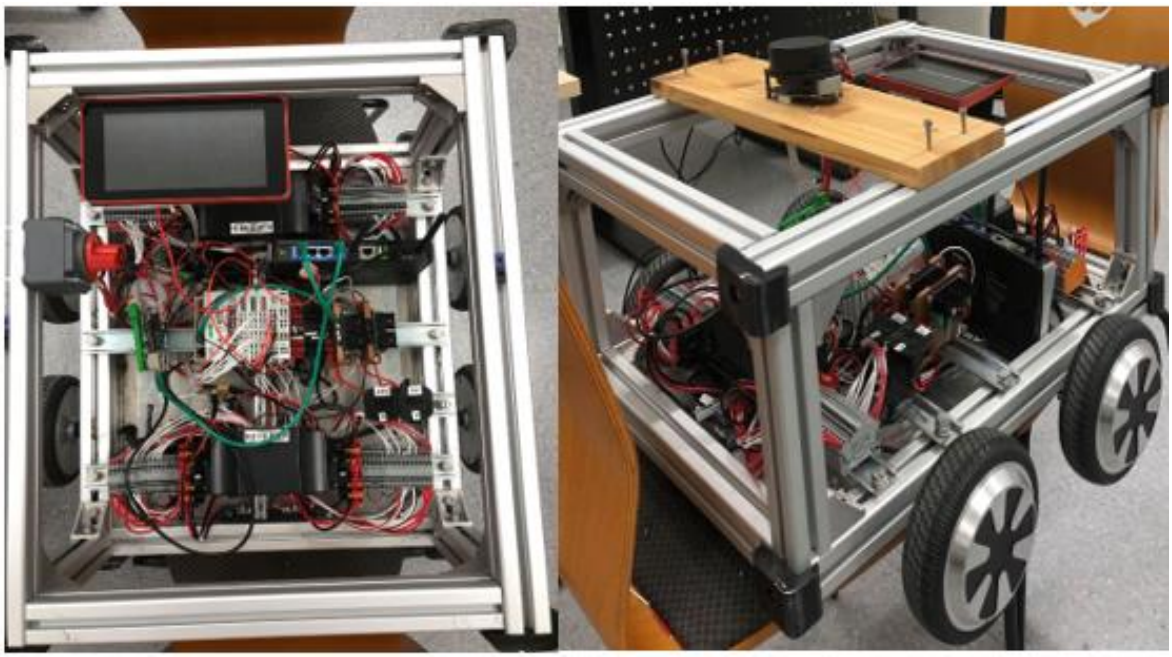
Figur 4.1 - Systemoversikt

Som illustrert i figur 4.1 har systemet i hovedsak to sensortyper. Den ene er PT100 sensorer som er temperatursensorer som overvåker temperaturen i motorkontrollene. Den andre er Lidaren som ROS bruker for å danne seg et bilde av miljøet.

Uavhengig av hvilket styringssystem man velger å benytte seg av sendes signalene til Node-Red på Raspberry Pi. Touchskjermen kan så brukes til å se overvåkingen til systemet eller velge kilden for styringen. Når valget er satt av brukeren sendes denne informasjonen til Node-Red som håndterer resten. Fra Node-Red sendes signaler om hvordan motorene skal bevege seg til Arduinoen, som oversetter disse til pulsvidde-modulerte signaler. Disse sendes videre til motorkontrollerne og motorene styrer tilsvarende.

4.2 Karosseri

Det har ikke blitt endret noe på selve karosseriet, men det har blitt fjernet ulike komponenter som ikke ble brukt. Til venstre i figur 4.2 vises et oversiktsbilde av plattformen, til høyre vises et bilde av hele plattformen med Lidaren.



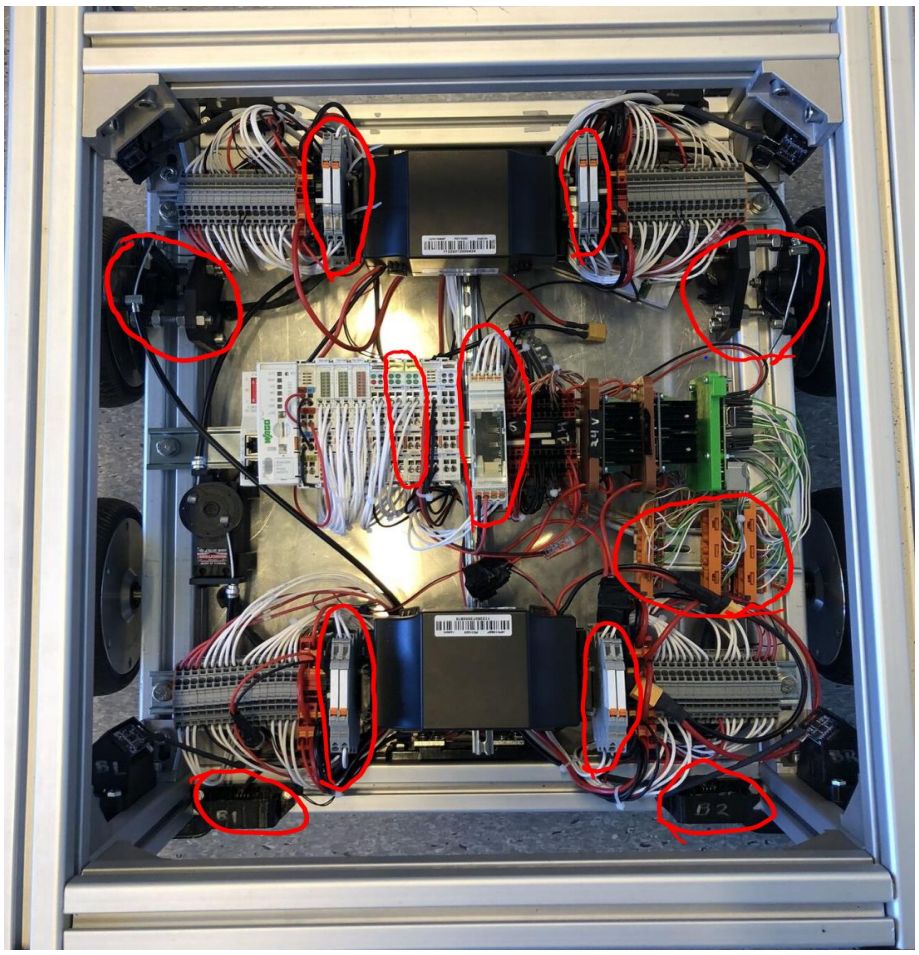
Figur 4.2 - Bilde av plattformen

4.2.1 Endringer i karosseri

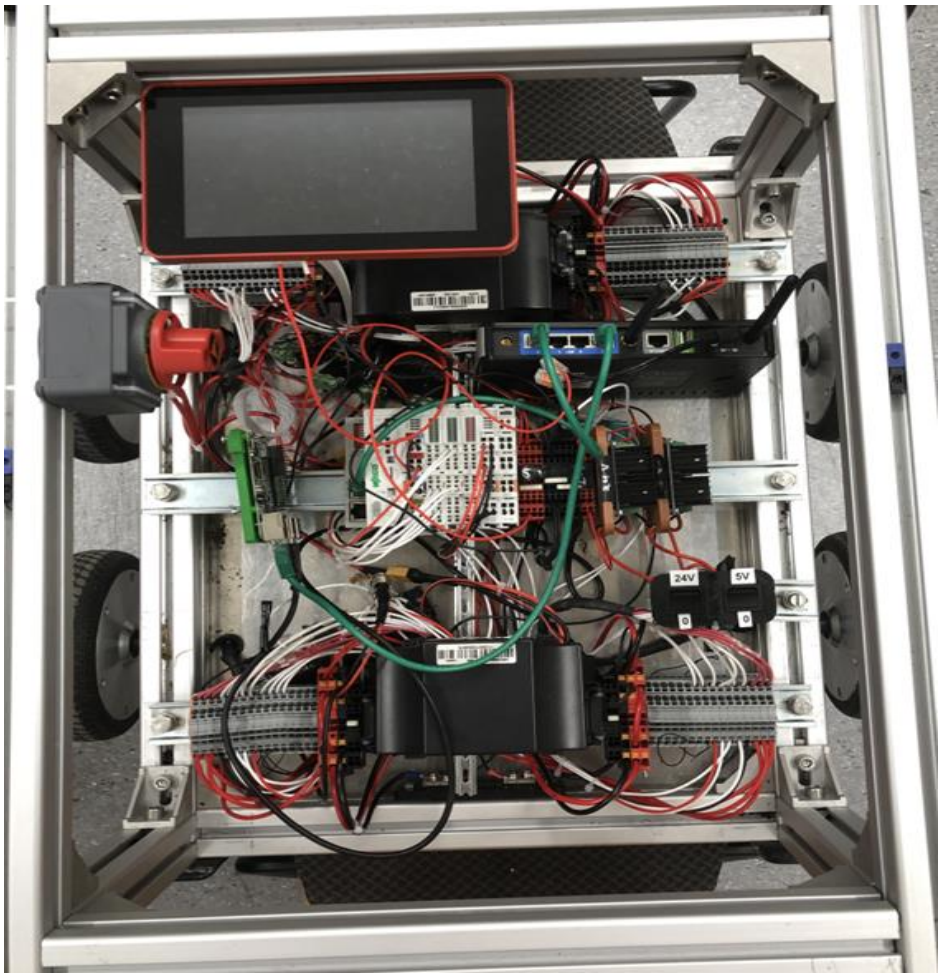
Hensikten med endringene var å fjerne overflødige funksjoner, samt oppgraderinger av utstyr som motorkontrollere.

Figurene 4.3 og 4.4 viser forskjellen mellom plattformen før og etter endringene, hvor det har blitt satt inn en touchskjerm og fjernet enkelte moduler, tidligere tilkoblet PLSen. Ledningssystemet til ultralyd sensorene samt selve sensorene er fjernet. Det mekaniske

bremsesystemet ble også fjernet. Endringene fra tidligere plattform er markert med røde sirkler.



Figur 4.3 - Plattform før endringer



Figur 4.4 – Plattform etter endringer

Plattformen hadde i utgangspunktet 8 ultralyd sensorer. Disse var plassert slik at man fikk 360º dekning, og ble brukt i et bremsesystem hvor plattformen skulle stoppe dersom den kom for nære vegger og objekter. De ble også brukt for testing av et enkelt autonomt system. Sensorene har blitt erstattet av en Lidar sensor som utfører de samme funksjonene. (Flaatten og Tande, 2018).

Planen var å bruke ultralyd sensorene sammen med Lidaren for ekstra dekke, men oppgaven kom ikke så langt før innlevering av bachelor. Dette førte til at sensorene ble fjernet grunnet frigjøring av plass. En anbefaling til videre arbeid blir å sette disse på for å optimere det autonome systemet over flere plan. Figur 4.5 viser et bilde av ultralyd sensor til venstre og RPLIDAR A1 til høyre.

Figur 4.5 - Ultralyd sensor og Lidar

Som følge av implementeringen av nye motorkontrollere, var det ikke behov for bremseskivene og det tilhørende mekaniske bremsesystemet, da motorkontrollene sitt elektroniske bremsesystem var godt nok. Bremseskivene ga også en risiko for at hoverboard motorene kunne låse seg i uønsket tilstand.

4.2.2 Konstruksjon

På innsiden av plattformen er det satt opp tre DIN-skinner. Øverste og nederste skinne består av de samme komponentene. Dette er batteri og motorkontroller, som er koblet opp identisk. Batteriet er plassert i midten og på hver side har det blitt plassert rekkeklemmer for inputs og outputs fra motor. Motorkontrollene er plassert under batteriene. På den midterste skinnen er styringsenhetene, PLS og Raspberry Pi plassert. Du finner også to DC-DC omsettere, 36-24V og 36-5V. (Tande og Flaatten,2018)

Rammen til plattformen er designet etter Tande og Flaatten sine kravspesifikasjoner. Det ene kravet var at konstruksjonen skulle tåle en totalvekt på 100kg. Dette gjorde at valget endte på 4x4cm aluminiumprofiler som konstruksjonsmaterialet til rammen. Disse profilene er lette, robuste og rustfrie. Det ble også vurdert stål og rustfritt stål, men grunnet vekt og mulighetene for å kutte og skjære i materiale, ble dette ikke implementert (Tande og Flaatten, 2018). I Figur 4.6 vises et utsnitt av den øvre rammen til konstruksjonen.

Figur 4.6 - Ramme (Flaatten og Tande, 2018)

Videre består konstruksjonen av 3D-printede deler. Disse er festet til forskjellige sensorer og komponenter, som for eksempel touchskjerm og hoverboard motorene.

3D-printing er brukt i oppgaven grunnet enkelheten og muligheten for å tilpasse ønskede deler, samt for feste av ulike komponenter. Det fantes også en enkel tilgang til 3D-printing grunnet egen printer på Sjøkrigsskolen. (Tande og Flaatten,2018)

3D-printingen er hovedsakelig blitt brukt for å lage braketter til ulike komponenter. Den ble også brukt til å feste hoverboard motorene til rammen og for å lage feste til touch-skjermen til Raspberry Pi. Det er også laget fester til de elektriske komponentene DC-DC omformer og batterier.

Alle design for 3D-modellene er laget av Flaatten og Tande i programmet Autodesk 123D-design. De hadde lagret tegningene som ble brukt for å erstatte ødelagte deler i en mappe. Figur 4.7 viser to eksempler på 3D-tegningene som ble produsert.

Figur 4.7 - Tegning av 3D-deler

Strømmen blir kontrollert av en hovedbryter, vist i figur 4.8. Denne er laget for å bryte hovedstrømmen mellom batteriene og hovedfordelingen. Bryteren kan bli brukt til å skille mellom batteriene og koble hvert enkelt batteri til hovedfordelingen, slik at bare batteri en eller batteri to forsyner plattformen med strøm.

Figur 4.8 - Hovedstrømbryter

I tillegg til hovedbryter har plattformen to DC-DC omsettere, 36-24V og 36-5V, siden PLSen trenger 24V og Raspberry Pi trenger 5V. Disse er videre koblet opp mot hver sin bryter slik at man kan ha ingen, en eller begge aktive. Figur 4.9 viser omsetterne til venstre og bryterne til høyre.

Figur 4.9 - DC-DC omsettere og brytere for 24V og 5V

Strømmen og spenningen blir målt av PLS-modulen 750-494/000-005. Batteriene ligger i parallell, noe som gjør at det kun trengs en måling for spenning. Strømmen blir målt ved bruk av en shunt motstand. Det er en lav motstand, 0,2 Ω, som tåler høy strøm. Den tåler

en belastning på 50A, og den har et spenningsfall på 75mV ved maks belastning. Videre kobles den i serie med resten av lasten og er det siste som kommer inn til batteriet på den negative siden. På denne måten forsikres plattformen mot at shunten blir bypasset av noe. Shunt motstand finners på hvert sitt batteri. Spenningsfallet over shuntmotstanden blir målt av PLS-modulen, som beregner hvor mye strøm som går gjennom motstanden (Flaatten og Tande, 2018).

Det ble besluttet å oppgradere brukergrensesnittet til plattformen og legge til en touch-screen skjerm. Denne får strøm gjennom pin tilkoblingen fra Raspberry Pi. Hensikten med touchskjermen er å vise grensesnitt siden til PLS ved oppstart. For å gjøre dette må det gjøres noen konfigurasjoner i Raspberry Pi.

Det første som blir gjort er å skrive følgende kommando i terminal vinduet:

sudo leafpad /etc/xdg/lxsession/LXDE-pi/autostart

Dette vil åpne konfigurasjonsfilen for oppstarten av Raspberry Pien. Denne filen inneholder tre linjer fra før av. Videre legges en fjerde linje til som gjør at PLSens webvisu side åpnes automatisk under oppstart av systemet. Denne filen skal nå inneholde følgende linjer:

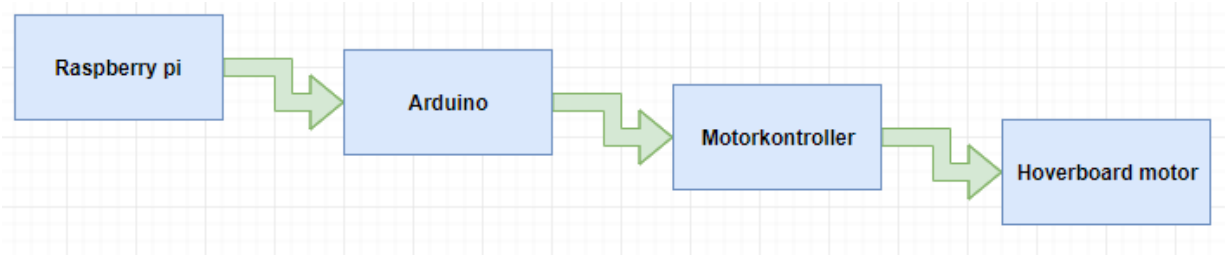
@lxpanel – profile LXDE -pi

@pcmanfm – desktop – profile LXDE-pi

@xscreensaver -no-splash

Chromium-browser –kiosk 192.168.84.182

4.3 Fremdriftssystem

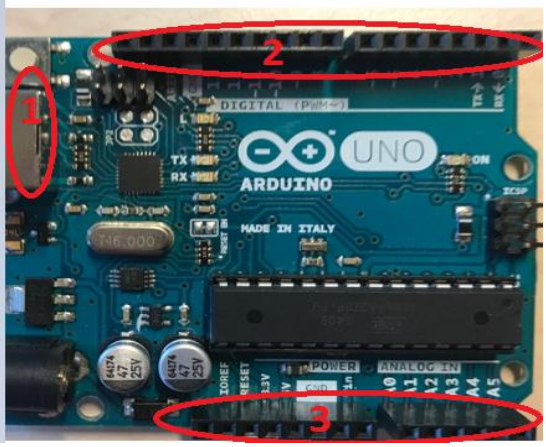


Figur 4.10 - Oversikt over fremdriftssystemet til plattformen

Figur 4.10 viser hvordan fremdriftssystemet til plattformen fungerer. Det er i korte trekk bygget opp av en Raspberry Pi som primært benyttes gjennom Node-Red. Kommunikasjonen mellom Raspberry Pi og Arduino går gjennom en USB-kabel. Gjennom pulsbreddemodulasjon sendes informasjon til motorkontrolleren, som får inn verdier som gir kommando til hoverboard motorene om hvilken hastighet og retning de skal bevege seg i. I delkapitlene under vil det beskrives i større detalj hvordan enhetene fungerer og sammensetningen av disse.

4.3.1 Arduino

En Arduino kan beskrives som en plattform som blir brukt til å bygge elektroniske prosjekter. Den inneholder både et fysisk programmerbart brett, samt software på datamaskinen som kan brukes til å skrive og laste opp kode til det fysiske brettet. En av grunnene til at Arduinoen på kort tid er blitt så populær er fordi det ikke trengs tilleggsutstyr for å få lastet ny kode til brettet, men kun en USB-kabel.



Figur 4.11 - Oversiktsbilde over Arduino

Nummer	**Type**
1	USB port
2	Digitale pins
3	Analoge pins

Tabell 4.1 - Nummereringen i Figur 4.11 vises.

USB porten på Arduinoen kobles direkte inn i Raspberry Pi, som er strømkilden til enheten. Slik Figur 4.11 viser, består den av ulike pins som kan tilkobles.

Pinsene i område 3 er analoge. Dette kan for eksempel brukes til å lese signal fra en temperatur sensor, og konvertere avlesningen til et digitalt signal. Dette området har ikke blitt brukt i systemet vårt.

I område 2 på Arduinoen finnes den digitale siden hvor pinsene går fra 0 til 13. Noen av pinsene kan også brukes gjennom puls bredde modulasjon (PBM). Dette blir ofte brukt til å styre retningen på en motor og er relativt enkelt å benytte seg av. Digitale signaler har to posisjoner. En av og en på som ofte blir tolket som 1 og 0. Analoge signaler derimot kan ha utrolig mange ulike posisjoner. Ofte i signalbehandling er disse to formene nødt til å samarbeide. PBM er en måte å kontrollere analoge enheter med en digital utgang. Det baserer seg på å endre pulsbredden i den hensikt å kontrollere utgangsspenningen, samt bestemme utgangsfrekvensen ved å endre syklusen (MotekPower, 2018). I forbindelse med dette snakkes det ofte om duty cycle. Denne blir gitt som en prosent av tiden den er på. Dersom man eksempelvis har 60% duty cycle, medfører det at signalet er skrudd på 60 % og av 40 % av tiden. Enkelt forklart er duty cycle forholdet mellom når den er aktiv og varigheten av perioden.

Den ene motorkontrollen bruker pin 3 og 4, mens pins 5 og 6 blir benyttet for den andre. Disse er koblet opp med felles jord mellom motorkontrollene og Arduinoen. Tabell 4.2 viser hvilke pins som er brukt for å at tilkoblingen mellom Arduino og motorkontroller skal forekomme.

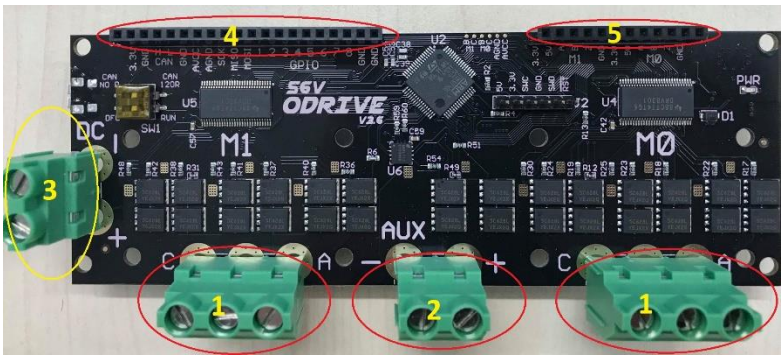
Arduino	Motorkontroll 1 (framhjul)	Motorkontroll 2 (bakhjul)
3	3	
4	4	
5		3
6		4
GND	GND	GND

Tabell 4.2 - Pins fra Arduino til motorkontroll 1 & 2

Siden systemet er basert på Node-Red, trengs en installasjon av en firmware på Arduinoen. Denne kalles “Firmata” og gir oss mulighet til kommunisere mellom Arduino og datamaskin. Firmata gir oss direkte tilgang til input/output pinsene. Videre installeres en palette på Node-Red som heter Node-Red-node-Arduino. Dette gjør at Node-Red får tilgang til Arduino sine inn- og utganger. Disse utgangene støtter primært tre forskjellige måter å operere på. Dette gjelder digitalt, enten 0 eller 1, analogt mellom 0 og 255, samt servostyrt fra 0 til 180.

4.3.2 Odrive Motorkontroller

I plattformen er det brukt trefase hoverboard motorer, som er indikert med gul, blå og grønne ledninger. Disse er koblet til de grønne skruterminalene på motorkontrollene indikert med M1 og M0, illustrert i figur 4.12. Det er også fem hall sensor feedback ledninger som er koblet inn på J4.



Figur 4.12 - Oversiktsbilde over motorkontrolleren. Nummereringen forklares i tabell 4.3

Hallsensor ledninger	J4 SIGNAL
Rød	5V
Gul	A
Blå	B
Grønn	Z
Svart	GND

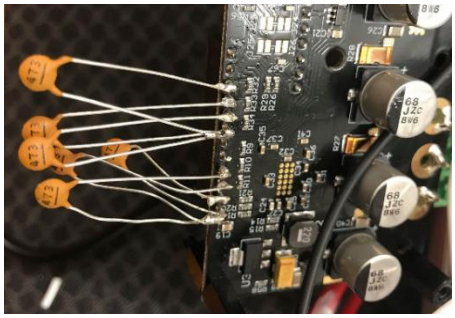
Nummer	Type
1	Skruterminal
2	Power resistor
3	Skruterminal
4	General Input/Output
5	J4, hall sensor innganger

Tabell 4.3 - Tabellen til venstre viser koblingene mellom hallsensorene og motorkontrolleren. Tabellen til høyre forklarer nummereringen på Figur 4.12

For å være kompatibel med ledningene fra hoverboard motorene til motorkontrollen, må det installeres noen filterkondensatorer på pinnene hvor hallsensorene kobles til. Hensikten med hall sensorene er at de endrer utgangsspenningen til magnetfeltet det blir påvirket av. Ved å ikke ha disse filterkondensatorene oppstod problemer med å få begge hoverboard motorene til å kjøre samtidig. Det ble loddet på tre filterkondensatorer for hver akse, illustrert i tabell 4.4 og figur 4.13 viser hvordan 47nF filterkondensatorene er loddet på.

Pin 1	Pin 2
Hallsensor A	GND
Hallsensor B	GND
Hallsensor Z	GND

Tabell 4.4 - Oppkobling filterkondensatorer



Figur 4.13 - Viser hvordan filterkondensatorene er loddet på

For å styre motorkontrolleren på ønsket måte, må det gjøres noen forhåndskalibreringer i den medfølgende programvaren odrivetool. Hovedhensikten med denne, er å gi bruker en mulighet til å styre kontrolleren manuelt, samt støttende funksjoner som eksempelvis firmwareoppdateringer.

Python 3 er kompatibel med motorkontrollen. For å kunne kalibrere kontrolleren, må Raspberry Pi og motorkontrollen kommunisere med hverandre. Dette gjøres ved å bruke den medfølgende mikro USB-kabelen. Udev konfigurasjoner må legges til for å få tilgang til odrivetool. Hensikten med Udev er at den oppretter eller fjerner nodefiler i device katalogen ved oppstart. Etter å ha skrevet inn linjene under i Linux terminalen, restartes Raspberry Pi slik at dette ikke må gjøres ved hver oppstart.

echo 'SUBSYSTEM=="usb", ATTR{idVendor}=="1209", ATTR{idProduct}=="0d[0-9][0-9]", MODE="0666"' | sudo tee /etc/udev/rules.d/50-odrive.rules

sudo udevadm control --reload-rules

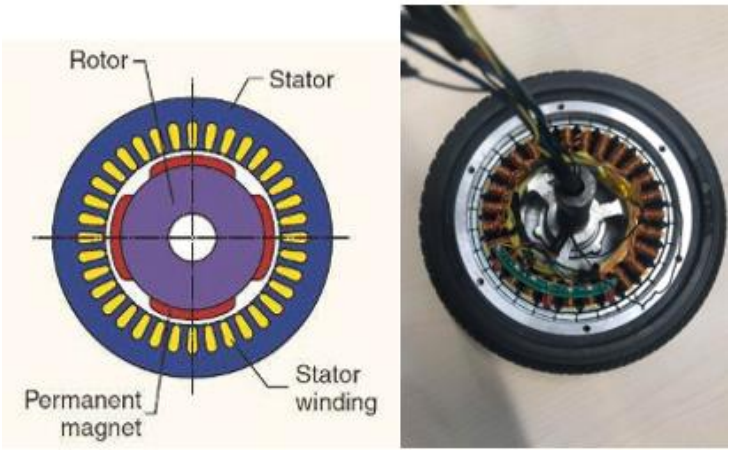
sudo udevadm trigger

```
pi@MotorPi:~ $ odrivetool
ODrive control utility v0.4.11
Please connect your ODrive.
You can also type help() or quit().

Connected to ODrive 205E3592524B as odrv0
In [1]:
```

Figur 4.14 - Odrv0 refererer til serienummeret på motorkontrolleren.

Figur 4.14 illustrerer at motorkontrolleren har kontakt med Raspberry Pi. Når disse er tilkoblet kommer serienummeret til den spesifikke Odrive motorkontrolleren opp i terminalvinduet og alle videre konfigurasjoner som blir gjort gjøres gjennom odrv0. Når enda en motorkontroller kobles til brukes odrv1 som referanse til sitt serienummer. På denne måten slipper man å skrive hele serienummeret hver gang man skal gjøre en kalibrering.

Når motorkontrollen er koblet til Raspberry Pi kan kalibreringen av kontrolleren begynne. Det første som må gjøres er å bestemme antall pol par i hoverboard motoren. Magnetene er konstruert slik at de har annenhver nord- og sydpol inn mot statoren. Enkelt forklart er en pol et magnetisk kraftfelt med retning nord eller sør som genereres av en permanent magnet eller strøm gjennom en spole.



Figur 4.15 - Innsiden av hoverboard motor (Jaszczolt, 2017)

Figur 4.15 viser hvordan innsiden av motoren ser ut. I statoren er det montert elektriske viklinger. Disse ligger på innsiden i motoren som spoler viklet rundt jernkjerner. De permanente magnetene er plassert på rotoren, som har til hensikt å skape et magnetfelt. På utsiden av statoren er det bygget 30 permanente magneter. Dette gjør at hoverboard motorene har 15 pol par tilgjengelig.

Permanent magnet motorer har vanligvis flere poler enn hva en induksjons motor har. På grunn av dette kreves det en høyere frekvens for å oppnå samme rotasjonsfart. Bakgrunnen for hvorfor man ønsker å benytte seg av et høyere antall poler er for å redusere tannhjulsmomentet, eller «cogging torque». Konseptet går ut på at det trengs et dreiemoment som er stort nok til å overvinne det motsatte dreiemomentet som er skapt av den magnetiske kraften mellom magnetene på rotoren og statorens jerntenner (Collins, 2018). Selv om det ikke kommer noe spenning eller strøm gjennom viklingene, er det fortsatt en magnetisk forbindelse mellom rotoren og statoren. Som følge av dette kommer et lite utslag i hoverboard motorene som gjør at det oppstår små rykninger. Dette fikk konsekvenser i startfasen under testingen av plattformen, hvor det bygget seg opp en spenning i hoverboard motorene, mens den ventet på kommandoer fra ROS.

Siden hoverboard motorene har en høyere motstand enn hva vanlige hobby motorer har, må det brukes en høyere spenning for kalibreringen av motorene. I tillegg har disse også en høy grad av induktivitet. Som følge av dette senkes båndbredden til motoren for å holde den stabil. Den samme konfigurasjonen under, ble gjort på de fire ulike aksene, hvor forskjellene ligger i hvilken motorkontroller og akse som arbeides med. Under vises sekvensen for hvordan odrv0.axis0, høyre bakhjul, blir kalibrert (Weigl, 2019).

Odrv0.axis0.motor.config.resistance_calib_max_voltage = 4

Odrv0.axis0.motor.config.requested_current_range = 25

Odrv0.axis0.motor.config.current_control_bandwidth = 100

Neste steg handler om å kalibrere hall sensorene. Disse har seks tilstander for hvert pol par i motoren. For å finne ut hvor mange counts per revolution (CPR) motoren har multipliseres antall pol par med antall tilstander i hver pol, som resulterer i 15 * 6 = 90.

Odrv0.axis0.encoder.config.mode = ENCODER_MODE_HALL

Odrv0.axis0.encoder.config.cpr = 90

Siden hall sensoren kun har 90 CPR, reduseres hastighetsbåndbredden for å få noen finere hastighetsestimater.

Odrv0.axis0.encoder.config.bandwidth = 100

Odrv0.axis0.controller.config.pos_gain = 1

Odrv0.axis0.controller.config.vel_gain = 0.02

Odrv0.axis0.controller.config.vel_limit =1000

Odrv0.axis0.conroller.config.control_mode = CTRL_MODE_VELOCITY_CONTROL

Etter å ha lagret konfigurasjonene, restartes Raspberry Pi, og de nye verdiene for motoren sjekkes. Dette gjøres ved å skrive inn *Odrv0.axis0.motor* i odrivetool. Eventuelle feilmeldinger kommer nå opp. Dersom dette ikke er tilfellet lagres kalibreringene. Neste steg er å sjekke forbindelsen mellom hall sensorene og motorkontrolleren. På samme måte som ved motoren, sjekkes også hall sensorene for eventuelle feilmeldinger. Om alt virker som det skal lagres kalibreringene.

Gjennom bachelorarbeidet fikk systemet feilmeldinger på de bakre hoverboard motorene. Dersom man skriver inn *dump_errors(odrv0)* får man vite i hvilken enhet feilmeldingen ligger. Dette kan eksempelvis være i motorkontrolleren eller i hoverboard motoren. Erfaringsmessig oppstod det feilmeldinger på hall sensorene. Disse fungerer på den måten at dersom de påvirkes av en magnet vil hull og elektroner dyttes til hver sin side som gjør at den ene siden blir positiv og den andre negativ. Et hull fungerer i denne sammenhengen som en positiv ladningsbærer. Den kan også betegnes som en ledig plass, hvor det er en mangel av et elektron. Med bakgrunn i dette prinsippet kan ikke alle hall sensorene enten være 1 eller 0. Dersom dette er tilfellet, vil en feilmelding på hall sensorene vises, som løses ved å bruke filterkondensatorene.

Noen av GPIO pinnene støtter bruk av pulsbredde modulasjon som kobles til Arduino, se tabell 4.2 for oversikt over hvilke pins som er brukt. I konfigurasjonen under brukes pin 3 som eksempel. De to første linjene bestemmer min og maks område som hoverboard motorene kan operere i. Den siste linjen forteller hvilken modus som brukes. Det samme gjøres for den andre aksen på motorkontrolleren. For å aktivere PBM inngangene må vi lagre konfigurasjonene og restarte odrivetool.

Odrv0.config.gpio3_pwm_mapping.min = -200

Odrv0.config.gpio3_pwm_mapping.max = 200

Odrv0.config.gpio3_pwm_mapping.endpoint=odrv0.axis0.controller._remote attributes['vel_setpoint']

Det som nå gjenstår er å gjøre oppstartsprosedyrene til Odrive automatiske. Hensikten med dette er å slippe å gå manuelt inn i odrivetool å sette aksene til True, hver gang plattformen skal brukes.

Odrv0.axis0.config.startup_closed_loop_control = True.
Odrv0.axis1.config.startup_closed_loop_control = True.

4.4 Styringsenheter

Opprinnelig bestod brukergrensesnittet for plattformen av en «visualisering fra e!Cockpit og en Playstation 4 kontroll» (Flaatten og Tande, 2018). Brukergrensesnittet har blitt oppgradert ved å installere en touchskjerm slik at man har mulighet til å se overvåkninger av blant annet batterikapasitet og temperatur på hoverboard motorene. Det er også mulig å regulere hvilket styringssystem man ønsker å benytte seg av på skjermen.

4.4.1 Programmerbar Logisk Styring

Programmerbar Logisk Styring (PLS) er en programmerbar prosessorenhet, som en datamaskin, og blir brukt for å styre blant annet industrielle prosesser. PLS er en robust, sikker modulær, og godt dokumentert system som er brukt i industrien for styring, regulering og måling. Modulene kan sees på som hvert sitt grensesnitt mellom interne variabler og eksterne elektriske signaler. Wago PLS programmeres i e!Cockpit og språket baserer seg på den industrielle standarden IEC 61131-3, basert på codesys. Tabell 4.5 består av navn og forklaring på bruken av de ulike modulene til PLSen.

Hovedmodul 750-8101	Hovedmodul – er en ethernet programmerbar feltbus kontroller som har to ethernet tilkoblingspunkter. Den har også tilkobling for 24VDC som føres ut på koblingsskinner til neste modul.
RTD 750-450	Dette er en Resistance Temperature Detector (RTD)-modul. Denne modulen blir brukt til å måle temperaturen til motor og motorkontroller. Systemet tar i bruk to moduler av denne typen hvor den ene tar for seg motor og den andre motorkontroller.
DO 750-1504	16 kanalers Digital Output-modul for 0/24VDC. Modulen er ikke i bruk, men blir stående siden det er lettere å legge til digitale output komponenter.
Power Measurement Module 750-494/000-005	Effektmålermodul. Den består av to spenningsinnganger og strømmålingsinnganger for DC. Strømmåling går via Shunt motstand (0,2 Ohm). Modulen kan måle opp til 277VDC og 20 kA DC via denne eksterne shuntmotstanden. Modulen brukes til å overvåke strøm fra hvert batteri og spenningen på batteriene.
Endemodul 750-600	Endemodul til PLS. Denne modulen er kun til bruk som endemodul som da er nødvendig for å fullføre den interne data kretsen og sikrer at dataflyt skal gå riktig.

Tabell 4.5 - WAGO modulene som er brukt med tilhørende forklaring

4.4.1.1 Kildekode

GVL består av globale variable lister til PLSen. Det er to forskjellige lister som består av fysiske måleinstrumenter som for eksempel temperaturmåling. Den andre inneholder interne variabler som blir brukt til programmering av systemet.

Den eksterne GVL listen inneholder variabler som er direkte koblet opp mot PLS boksen og har fysiske ledninger som er koblet til. Det er disse som direkte tar imot målinger som PLS boksen registrerer. Figur 4.16 viser listen av variablene som er i bruk.

```
1  VAR_GLOBAL
2
3      //RTD Variables
4      Motor3_temp          AT %IW3 : INT;
5      Motor4_temp          AT %IW4 : INT;
6      ESC1_temp            AT %IW5 : INT;
7      ESC3_temp            AT %IW7 : INT;
8  END_VAR
```

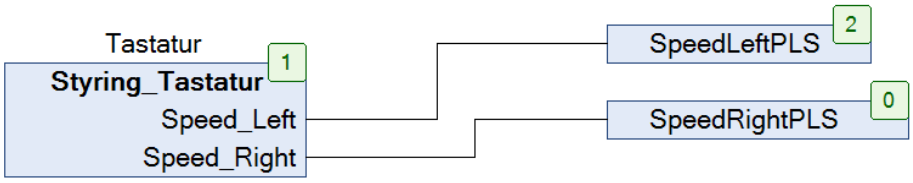
Figur 4.16 - Global variabel liste

Den interne GVL listen inneholder variabler brukt i konstruering av blokk diagrammer og programmeringen til de ulike ønskede funksjonene. Variablene vist i figur 4.17, er ikke fysisk koblet til PLS boksen, men er tilleggs variabler som kan bli brukt til sitt ønskede formål. I dette tilfellet er det switchen over til Node-Red, styring av plattform ved bruk av PLS og variabler for måling av temperatur i hoverboard motorene og motorkontroller.

```
VAR_GLOBAL
    //Power
    Power_Intern :  BOOL;


    //Switch mode
    SwitchDualshock : BOOL;
    SwitchPLS : BOOL;
    SwitchROS : BOOL;

    MODE : INT := 0;     //Mode = 0 is Dualshock -> Default

    // Controls for PLS steering
    SpeedLeftPLS : REAL;
    SpeedRightPLS : REAL;

    //Visualization
    Menuindex :      INT := 0;
    Showmenu :       BOOL;

    //Measuring variables
    M3_temp :        REAL;
    M4_temp :        REAL;
    MC1_temp :       REAL;
    MC3_temp :       REAL;
    END_VAR
```

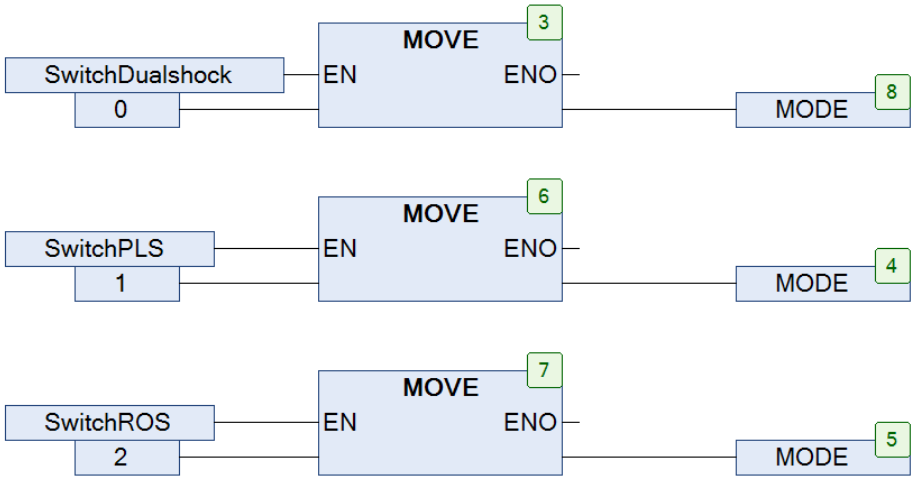
Figur 4.17 - Interne variabler i PLS

Tastaturstyringen består av en tastatur blokk med tilhørende kommandoer som vises i figur 4.18. Denne strukturen viser hvilke kommandoer som blir gitt for å svinge til høyre, venstre og for å kjøre framover eller bakover. Blokken er koblet opp mot to globale variabler som vist i figur 4.18. Variablene blir videre hentet til Node-Red.



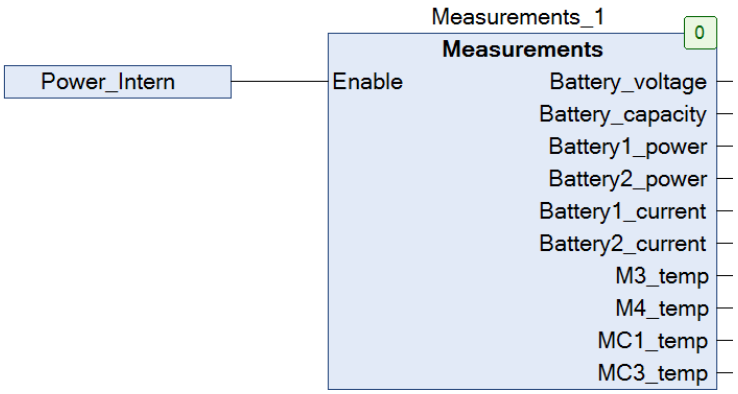
Figur 4.18 - Tastaturblokk

Blokksystemet til switchen mellom styringsformene blir illustrert i figur 4.19. Switchen består av en MOVE blokk. Denne aktiverer ønsket styringsform ved at variabelen mode sender tallet 0, 1 eller 2. Tall 0 tilsvarer PS4 -, 1 tilsvarer PLS - og 2 tilsvarer ROS styring.



Figur 4.19 - Switchblokk system

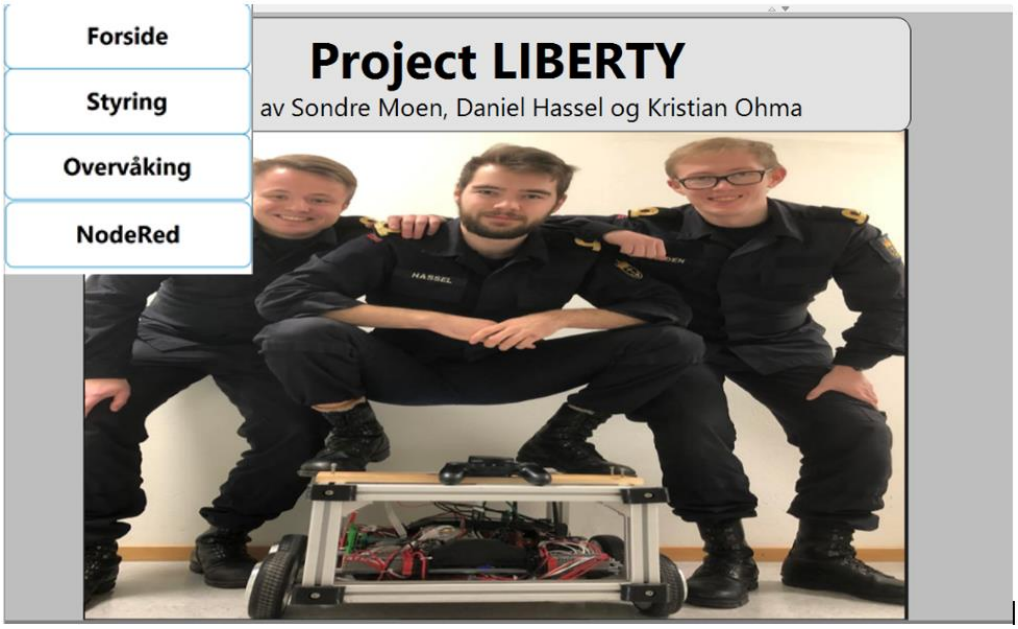
Figur 4.20 er den visuelle boksen av utregningene fra vedlegg B.13. Her vises power variabelen koblet opp mot hele måle boksen.



Figur 4.20 - Målingsblokk

4.4.1.2 Webvisu

Oppkoblingsprosedyren, vedlegg A.4, sender deg til PLS sin startside. Dette er en nettside designet for brukergrensesnitt.



Figur 4.21 - Forside

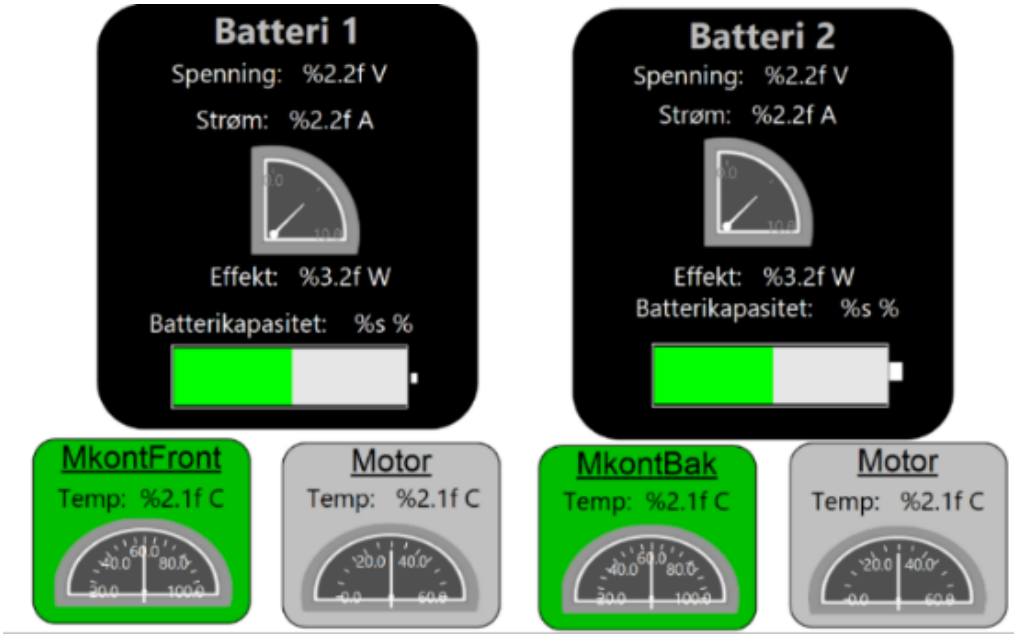
Forsiden er designet med navnet på prosjektet og et bilde av oss med plattformen som vist i figur 4.21. Oppe til venstre, finnes valgmenyen. Den består av forside, styring og overvåking. Menyen er interaktiv og sender deg til ønsket side ved et klikk på navnet.

Figur 4.22 - PLS styreside

Styringen består av flere interaktive elementer, som har ulike funksjoner. Figur 4.22 viser styrings-siden. Øverst til venstre er power-knappen for hele systemet vårt. Denne er laget slik at alle hoverboard motorene vil stoppe ved av-modus og starte ved på-modus. Den styrer strømforsyningen ut til motorene. Under finner du knappene for hvilket styrings-system som skal være aktivt. Ved starttilstand er det PS4 systemet som er aktivt. Knap-pene har en tilhørende lyslampe som aktiveres ved på-modus.

Øverst i midten finner du batterikapasiteten til plattformen, som aktiveres når power er aktivert. Under finner du styringssystemet for PLS styring, hvor knappene lyser når de er aktive ved en liten hvit ring rundt knappen. Bruker må deaktivere aktiv knapp for å bytte kjøreretning. Dette gjelder framover eller bakover, svinge kan brukes uansett.

Til høyre er styringen av pådraget til PLS-styringen, som går fra -1 til 1. Ved positive verdier vil den kjøre framover i framover-modus og bakover ved negative verdier. Den vil fungerer motsatt ved bakover-modus.



Figur 4.23 - Overvåkinger fra PLS

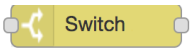
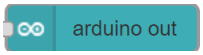
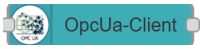
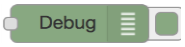
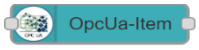
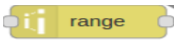
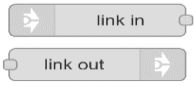
Figur 4.23 viser overvåkingen av PLSen. Siden består av målinger fra begge batteriene og temperaturmåling av viktige komponenter. Batterimålingen i de svarte boksene, blir først aktivert når power-knappen er aktivert. I hoverboard motorene blir spenning, strøm og effekt målt. Strøm og effekt som blir hentet ut fra hvert batteri vil være den samme, ettersom de står i parallell.

Nederst i figur 4.23 finner du målinger fra henholdsvis motorkontrollene i grønne bokser og måling av hoverboard motorene i de grå boksene. Det som blir målt er temperaturene til de forskjellige enhetene ved hjelp av en PT100 sensor.

4.4.2 Node-Red

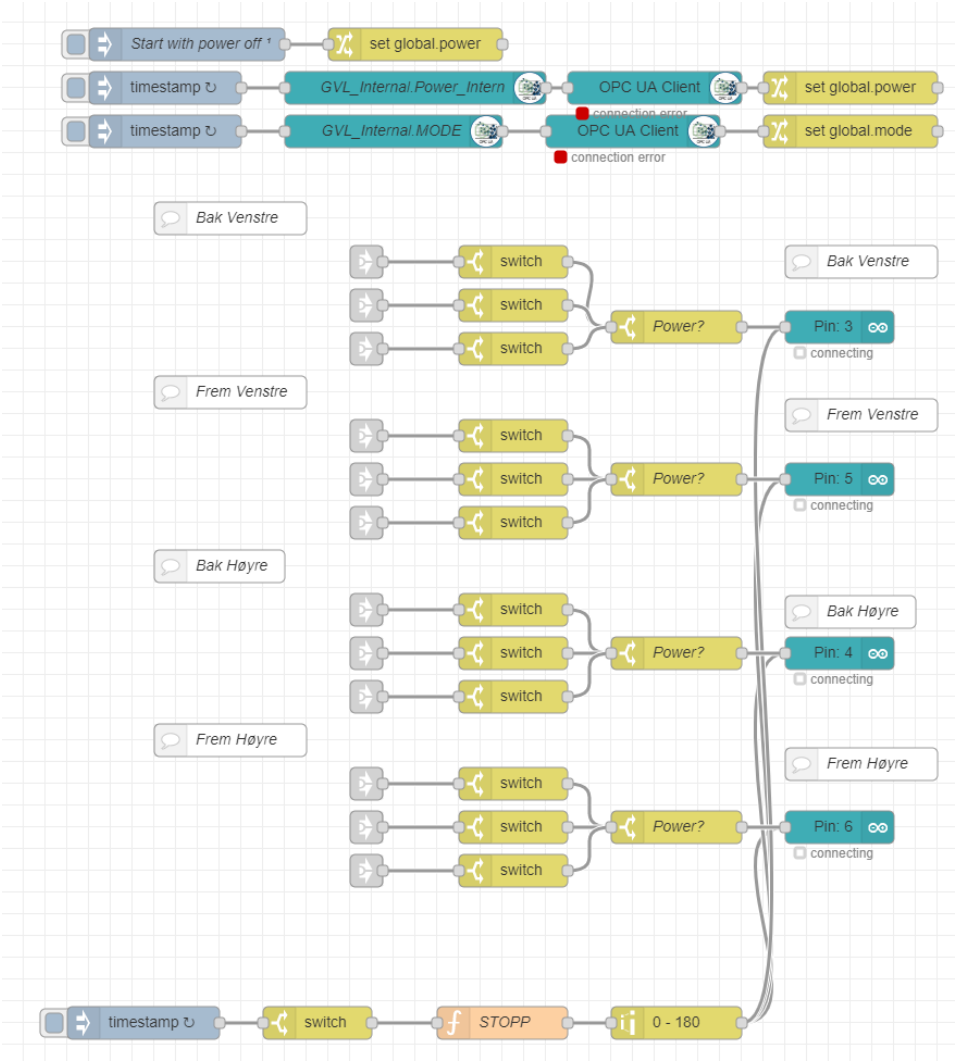
Node-Red er et programmeringsverktøy som gjerne brukes til å knytte sammen ulike hardware deler. Denne type programmering fremstår som enkel og oversiktlig. Programmeringsspråket bygger på Node(.js), som har forløp fra JavaScript kode.

For å bruke Node-Red på ønsket måte må det installeres noen paletter. En palette oversikt består av alle ønskede noder som er installert og kan brukes. Siden det trengs andre paletter enn de standardiserte brukes palette manageren til å installere nye noder, som legges inn i paletten. Tabell 4.6 viser hvilke noder som er brukt i Node-Red.

Type	**Bilde**	**Beskrivelse**	**Palette**
Switch		Dirigerer data til ulike grener.	Node-Red
UDP		Sender informasjon fra vertsmaskin til mottaker	Node-Red
Arduino		Oppretter forbindelse med Arduino og skriver til pin.	Node-Red-node-Arduino
OPC UA Client		Kobler Node-Red og PLS sammen	Node-Red-contrib-opcua
Funksjon		Tillater JavaScript kode	Node-Red
Debug		Oversikt over meldingen som blir sendt	Node-Red
Inject		Brukes til å manuelt trigge en flow	Node-Red
OPC UA item		Inneholder informasjon om variabler	Node-Red-contrib-opcua
range		Skalere input signaler.	Node-Red
Link in Link out		Link in og Link out brukes til å lage virtuelle ledninger mellom ulike flows.	Node-Red
Pythonshell in		Gjør at det kan kjøres et Python script.	Node-Red-contrib-pythonshell

Tabell 4.6 - Oversikt over nodene som er brukt i Node-Red

4.4.2.1 Switch



Figur 4.24 - Viser hvordan switch systemet fungerer

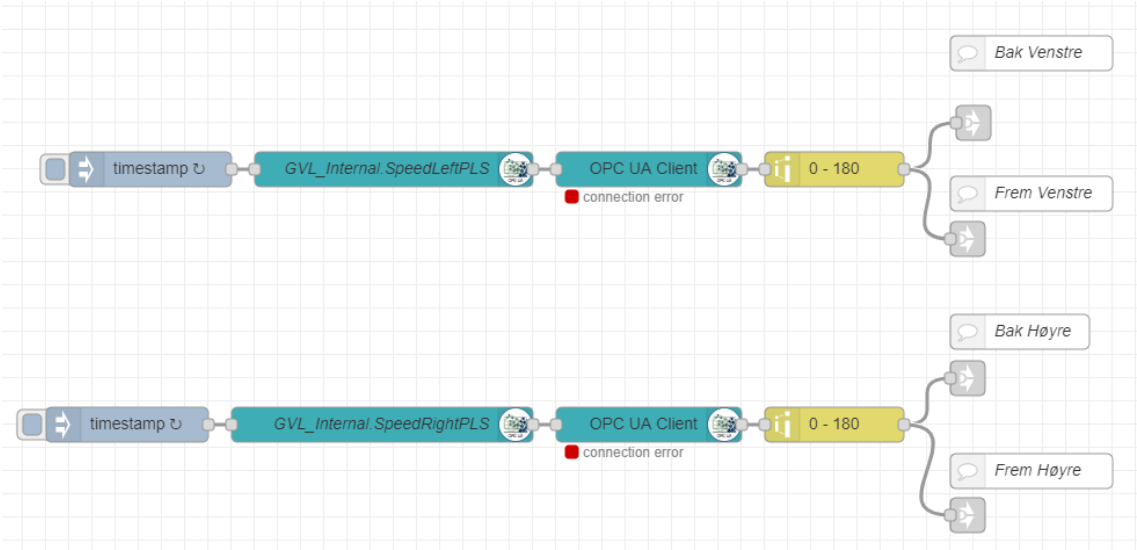
Figur 4.24 viser flowen for switchen i styringssystemet vårt. Denne blir brukt til å bytte mellom PS4kontroller, PLS og ROS. De to førstnevnte er manuelle styringssystemer hvor man styrer hva plattformen skal gjøre. ROS er derimot et autonomt program som selv finner den beste ruten for å komme fram til målet.

På toppen av flowen finner du en oppstartlinje. Systemet er programmert slik at det starter med power av på webvisu. Dette gjør at systemet vil starte i en stillestående posisjon istedenfor å laste opp sist brukte modus, som reduserer risikoen for at plattformen plutselig kjører framover.

Linje nummer to og tre er koblingen mellom Node-Red og PLS, hvor den øverste er for å hente ut data fra PLS om power-knappen og linjen under er koblet opp mot Mode i PLS. Mode styrer hvilket styringssystem som er i bruk.

I neste del av flowen følger selve switchen til plattformen. Link-out noden har fått beskjed om hvilken modus som skal være aktiv fra Mode systemet. Dersom power ikke er på, vil prosessen bli blokkert. Stopp noden fungerer som en brems i systemet, da ingenting går gjennom når den er aktiv.

4.4.2.2 Styresystem mellom Node-Red og PLS



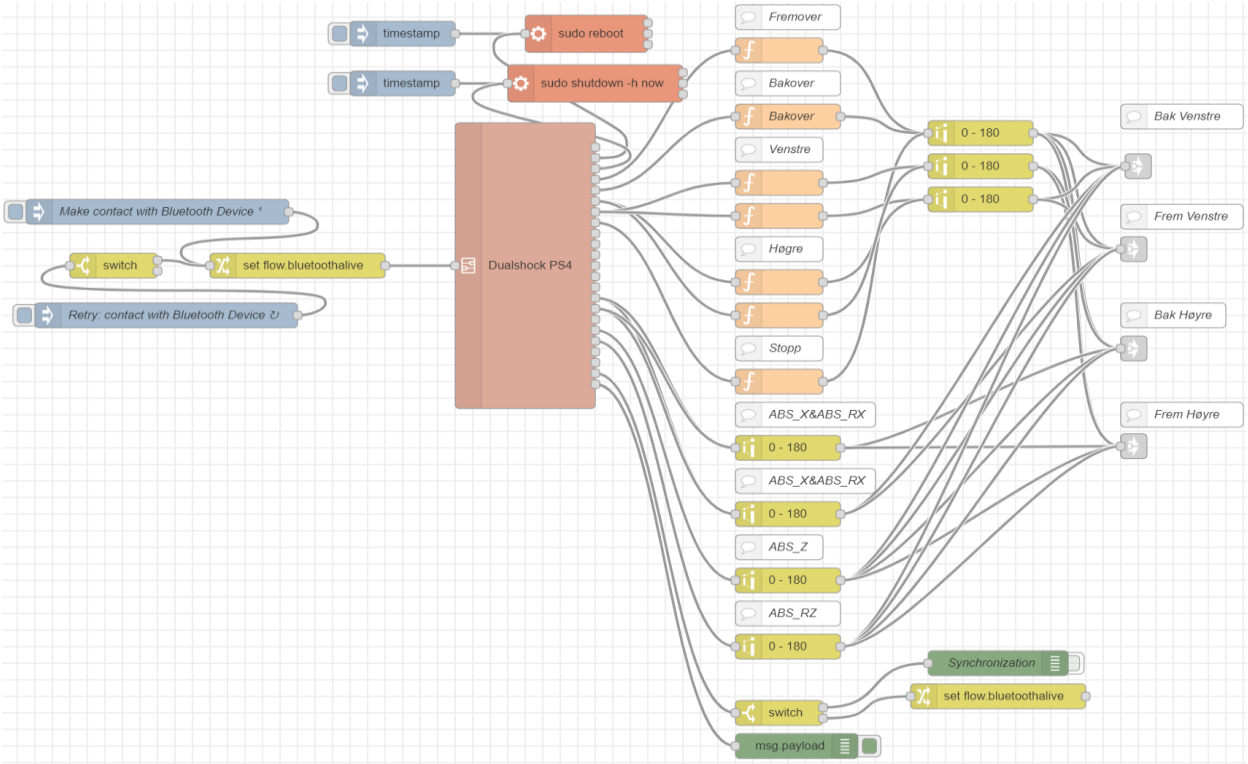
Figur 4.25 - Tilkobling mellom Node-Red og PLS

Figur 4.25 viser kommunikasjonen mellom Node-Red og PLS som går via OPCUA protokollen. Den har egne noder til å knytte sammen variabler i Node-Red med variabler i PLS. «*GVL_Internal.SpeedLeftPLS*» noden inneholder informasjonen som trengs for en variabel. Videre er det en Opcua Client node som gjør at Node-Red får forbindelse med PLS. Denne noden gir informasjon om PLS og Node-Red er koblet sammen. Dette går inn i en range node, som gjør at input signalet kan skaleres. I denne noden får man et input signal fra 0-255 som skal omgjøres til 0-180. Dette fordi det skal være samsvar mellom input til Arduino out noden og hvilket resultatområde som settes på range noden. Videre går signalet inn i en node som lager virtuelle koblinger mellom ulike flows. Dette blir gjort da det er ønskelig at systemet skal ha tre ulike måter å styres på.

4.4.2.3 Playstation 4 kontroller

Plattformen kunne opprinnelig styres fra en playstation 4 kontroller. Node-Red ble brukt som webserver for scriptet som håndterte kommunikasjonen med fjernkontrolleren. For å hente tilbake dataen fra HTML siden ble det brukt en Websocket node. Denne setter opp en adresse og viderefører all data sendt dit (Flaatten og Tande, 2018). Det var også koblet opp blokker i PLS for å styre farten på motorene.

Denne oppgaven begrenser PS4 styresystemet til Node-Red. Med bakgrunn i dette gikk systemet bort fra den opprinnelige måten å styre på, da det ikke var hensiktsmessig å gå gjennom en HTML side for å hente verdiene som behøvdes for å kjøre plattformen. I figur 4.26 illustreres det hvordan styresystemet til PS4 kontrolleren fungerer. Dualshock 4 boksen inneholder en subflow, se figur A.3, hvor verdiene fra PS4 kontrolleren hentes ut.



Figur 4.26 - Bildet viser flowen til PS4 styresystemet og hvordan det er satt sammen.

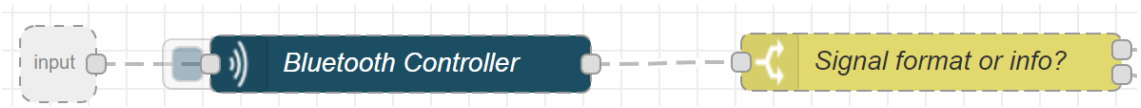
Før programmet kan kjøres må noen forhåndsinnstillinger gjøres. Dette innebærer å laste ned noen tilleggspakker fra Python til Raspberry Pi.

Sudo apt-get install python-dev

Sudo apt-get install python-pip

Sudo pip install evdev

Etter dette installeres en palette som gjør at man har tilgang til de ulike Python pakkene i Node-Red. Denne hentes ved å installere Node-Red-contrib-pythonshell. Videre må en fil ved navn Bluetoothminimal.py legges til i /home/pi mappen. Denne filen lagrer og printer ut data fra enheten på starten av flowen.



Figur 4.27 - Bluetooth noden sender verdier fra PS4 kontrolleren

Ved å klikke på den blå pythonshell in noden, illustrert i figur 4.27, vises informasjonen om plasseringen til Python filen. Evdev importeres og det hentes ut nødvendige biblioteker. Videre lages en variabel ved navn gamepad som lagrer data fra PS4 kontrolleren. Denne variabelen skal være lik inputdevice (/dev/input/event3). Deretter skrives informasjonen fra gamepad variabelen ut. Videre lages en loop som printer ut verdier fra PS4 kontrolleren som sendes inn i en switch node.

Denne inneholder to ulike betingelser. Den en sjekker om msg.payload inneholder riktig format ved å sette en string som skal inneholde formatet som er likt innholdet i kategori formatet. Hvis dette ikke stemmer og den inneholder en annen informasjon blir dette sendt videre til status noden.

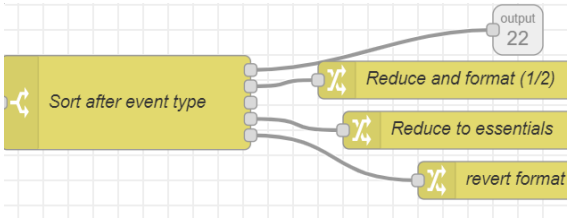
Dersom formatet stemmer overens sendes meldingen inn i en split node. Hensikten er at den splitter meldingen til ulike sekvenser av meldingen ved å bruke /n som gir linjeskift. Disse sekvensene tas videre inn i en change node som bestemmer hvordan meldingen settes opp.

I mappen /dev/input er det tre ulike event filer, henholdsvis 1, 2 og 3. Disse gir mulighet til å styre kontrolleren på ulike måter. I tabell 4.7 er det en oversikt over hvilken fil som gir hvilken måte å styre på.

Filnavn	**Styremåte**
Evdev1	Touchpad
Evdev2	Bevegelse av PS4 kontrolleren
Evdev3	Knapper

Tabell 4.7 - Styremåter for PS4

Kontrolleren er satt opp til å kunne styre med knappene, da dette ble ansett som mest hensiktsmessig til plattformen.

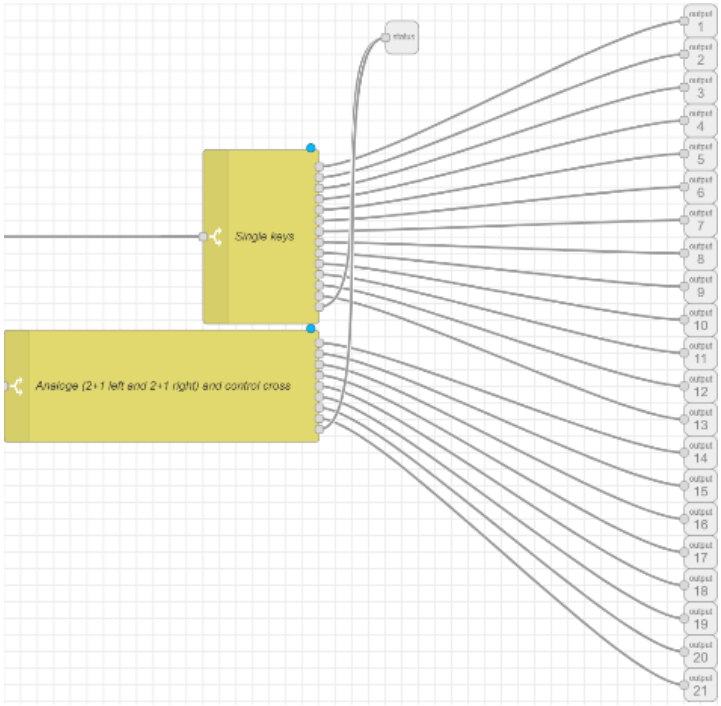


Figur 4.28 - Viser utgangene til switch noden

Etter at meldingen er konvertert sendes den videre til tre change noder og til output 22. Dette er et virtuelt signal som går ut av subflowen og til flowen. Den går så videre til synkoriseringsnoden hvor bluetoothalive noden settes til true.

Utgang nummer to fra switch noden "sort after event type" går til change noden "reduce and format (½) som går videre til "reduce and format (2/2)" noden. Hensikten med disse nodene er å redusere og formatere msg.payload fra switch noden. Utgang nummer to fungerer på tilsvarende måte som reduce and format noden.

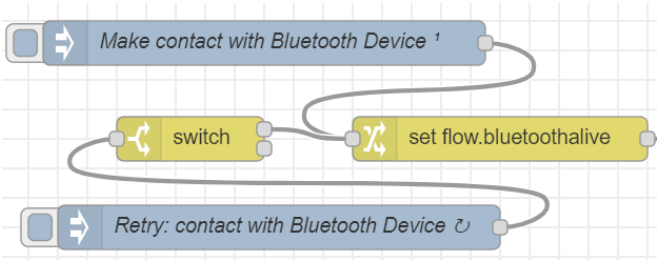
I figur 4.28 går utgang nummer fem fra switch noden inn i en change node som igjen går til utgang nummer 23. En debug node settes på utgangen for å sjekke hvilke meldinger Node-Red får fra PS4 kontrolleren. Disse meldingene vil ikke være lik formatet til de tre andre utgangene.



Figur 4.29 - To switch noder som separerer analoge og boolske knapper

Figur 4.29 består av to ulike switch noder som separerer analoge og boolske knapper. Med boolske knapper menes de knappene som tilsvarer 1 når den er på og 0 når den er av. Med analoge signaler menes de knappene hvor man kan ha knappen i ulik posisjon avhengig av hvor mye man presser den inn. Det analoge signalet går fra 0 til 255. I switch noden "Analoge (2+1 and 2+1 right) and control cross" er knappene som har et signal fra –1 til 1 også tatt med. De to utgangene som går til den virtuelle status noden er satt til otherwise. Meldingene blir sendt til denne noden dersom input signalet ikke tilsvarer noen av betingelsene satt i switchen. Videre går hvert enkelt output signal inn i flowen hvor det er mulig å programmere funksjonen til hver knapp.

I figur 4.30 sendes en puls etter 3 sekunder fra en inject node. Denne går videre inn i en change node, hvor verdien av bluetoothalive settes til false. Noden henger sammen med synchronization noden som settes til true dersom den får kontakt. Dersom ikke, vil "retry contact with Bluetooth device" noden forsøke å koble seg til PS4 kontrolleren. Denne sender en puls hvert 3 sekund inn i en switch node, som på lik linje med change noden er satt til false.



Figur 4.30 - Sjekker om Node-Red har fått kontakt med bluetooth

Playstation kontrolleren er satt opp med to moduser, hvor den ene modusen kan regulere hastigheten, mens den andre har forhåndsbestemt hastighet. Bakgrunnen for dette valget er fordi det er ønskelig å gi brukere mulighet til selv å bestemme hvordan man ønsker å kjøre. Det kan tenkes at brukere ønsker å kjøre plattformen utendørs og har et større behov for å regulere farten enn hva som muligens ville vært tilfellet innendørs. I dokumentasjon tabell A.3 fremlegges en tabell med oversikt over knapper med tilhørende modus.

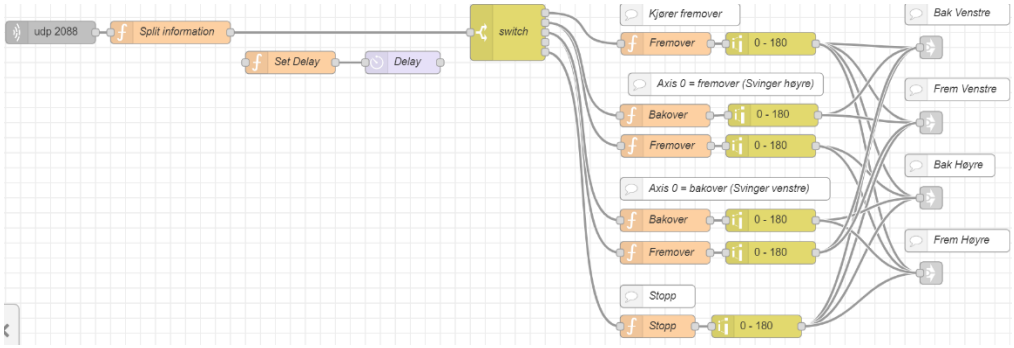
I den ikke regulerbare modusen får det enkelte output signalet en funksjon som forteller hva den skal gjøre. Trekant knappen på PS4 kontrolleren gjør at plattformen kjører frem-over. For at dette skal skje, sendes msg.payload inn i en funksjon som gir meldingen en gitt verdi.

Denne verdien er satt på grunn av nullpunktet til motorene. For at plattformen skal kjøre bakover brukes en lavere verdi enn nullpunktet. For å svinge til høyre eller venstre brukes to funksjoner, hvor den ene siden har lavere verdi enn nullpunktet, og den andre siden høyere.

I regulerbar fartsmodus er funksjonene blitt utelatt, hvor msg.payload sendes direkte til en range node. Her endres input signalet til et område mellom 0 og 180. Dette gjør at bruker kan endre hastigheten avhengig av hvor hardt knappene trykkes inn. De to stickene brukes i denne modusen til å svinge. Disse er programmert slik at to av hoverboard mo-torene går i lik retning. Dette er løst på en enkel måte ved at input område endres fra negativt til positivt for den ene siden, og omvendt for den andre. På denne måten kan brukeren styre motorene slik man vil.

Etter at msg.payload er sendt gjennom hver sin range node går den videre til en virtuell node, se avsnitt 4.4.2.1 for switch systemet i Node-Red.

4.4.2.4 Kommunikasjon med ROS



Figur 4.31 - Kommunikasjonen mellom Node-Red og ROS

Som vist i figur 4.31 brukes User Datagram Protocol, UDP, for at kommunikasjonen mellom ROS delen og Node-Red skal skje. Fordelen med UDP er at den sender informasjon direkte til destinasjonen uten å sjekke om systemet er klar til å motta dataene eller ikke. Dette gjør at hastigheten mellom vertsdatamaskinen og mottaker er rask. UDP blir betegnet som en upålitelig protokoll, da den ikke kan korrigere feil eller sende tapte pakker på nytt (Rouse, 2019). Til vårt formål fungerer UDP på en god måte, da det sendes korte meldinger om retningen plattformen skal ta. Dette går inn i en switch, hvor meldingen som kommer fra UDP, bestemmer hvilken node den skal sende ut til. Dersom switchen får en melding om "D", vil den gå til noden "fremover", som er en funksjon, hvor farten er forhåndsbestemt. Dette er fordi det ikke er ønskelig å ha ujevn fart, men heller en konstant og sakte fart på plattformen.

Informasjonen går videre inn i en range node hvor input området endres til en range som er tilsvarende området på Arduino out noden. Dette blir gjort for at det skal være samsvar mellom output området fra range til Arduino out noden.

4.5 Utvikling av kildekode

Oppgaven kildekoden må løse er å hente ut informasjon fra HECTOR_SLAM og RPLIDAR ROS nodene. Neste steg er kalkuleringen av veien til mål. Avslutningsvis må informasjonen sendes til både Node-Red for styring av plattformen og til RVIZ for visualisering av data.

Filen som inneholder dette heter «main.py». Scriptet starter med å abonnere på topics «map» og «slam_out_pose». Dette er topic'ene som inneholder kart- og posisjonsdata. Dette gjøres ved hjelp av ROS modulen «message_filters». Når programmet har mottatt begge disse dataene med mindre enn 0.1 sekunder avstand, registreres en callback-funksjon. Til slutt genereres en loop, som er enkelt med spin-funksjonen til rospy.

Callback-funksjonen bruker først klassene definert i scriptene «myMap.py» og «position.py» til å lagre dataene for videre bruk. Videre regnes det ut hvilken node plattformen befinner seg i. Målet blir deretter satt som posisjon i koden og regnet over til nodeform.

Neste steg er å finne veien til målet. Dette løses ved hjelp av funksjonen «aStarSolver» fra scriptet «aStar.py». Funksjonen returnerer veien som brukes til å sende informasjon til RVIZ for visualisering til brukeren. Det er funksjonen «line_draw» fra scriptet «drawLine.py» som står for dette.

Videre brukes funksjonen «determine_action» fra scriptet «action.py» for å beregne hva plattformen skal gjøre. Denne funksjonen regner ut om plattformen skal kjøre, svinge eller stå stille. Denne informasjonen sendes til Node-Red som utfører disse kommandoene.

Til slutt publiseres ønsket retning og faktisk retning til ROS. Denne informasjonen brukes for å lage rosbags, som brukes til å analysere data man får under testing. Programmets tidsforbruk publiseres også slik at programmets hastighet kan overvåkes.

4.5.1 Hvordan fremstilles kartet

Programvaren ser kartet det får fra topic'et «map» som en liste med tall. Tallene representerer innholdet i kartet, der 1 betyr okkupert område, 0 betyr ikke okkupert område og –1 betyr ingen informasjon om området. Kartets størrelse er definert i filen «mapping_rover.launch» som kartets størrelse multiplisert med oppløsning. Dette vil produsere et kvadratisk kart, som gjør utregninger innad i kartet enkelt. Gjennom oppgaven er posisjoner fremstilt på tre ulike måter.

Den første måten å fremstille posisjoner på er slik de er fremstilt i ROS. Denne måten er et helt vanlig X,Y koordinatsystem angitt i meter. Y aksen er definert som plattformens startretning og X aksen er definert som høyre for plattformens startretning.

Den andre måten er å fremstille posisjonene som noder i et X,Y koordinatsystem. X og Y aksene er definert i samme retninger som tidligere, men nå telles avstander i antall noder, eller punkter og ikke i meter.

Den siste måten å fremstille kartet er ved hjelp av node nummer. Dette er et punkts posisjon i listen av noder. Node nummer 0 er helt i nord-vest og node nummer øker med 1 i østlig retning og med kartets bredde i sørlig retning. I denne tredje fremstillingen vil nord være definert som robotens startretning.

4.5.2 Implementering av A*

A* er implementert i scriptet «aStar.py». Den åpne listen lages ved å bruke prioriterte lister. Dette gjøres med filen «queue.py» som er en wrapper for enkelte funksjonaliteter fra heapq biblioteket i Python. Ved å bruke en prioritert kø er det enkelt for algoritmen å velge neste punkt. Startperimeterne blir satt før man går inn i selve utregningen av veien.

Algoritmen velger det første objektet fra den åpne listen og finner naboene dens. Deretter regner scriptet ut kostnaden for å komme seg til hvert av punktene. Disse legges inn i den åpne listen.

Dersom algoritmen har funnet frem til målet brytes loopen. Ved hjelp av funksjonen «makePath» lager scriptet veien til målet og den returneres. Dersom det ikke finnes en vei til målet, søker algoritmen gjennom alle punkter den kan komme seg til før den returnerer nullverdier for veien til målet.

4.5.3 Implementering av valgfunksjonen

Valgfunksjonen «determine_action» finnes i scriptet «action.py». Denne funksjonen sjekker først hvor langt plattformen må svinge for å kjøre rett vei. Dette gjør den ved hjelp av funksjonen «getAngles». Denne funksjonen regner ut både faktisk og ønsket retning i radianer. Faktisk retning er gitt i kvaternioner fra ROS og ønsket retning er gitt i kvaternioner i scriptet «directionDatabase.py». For å tolke dette er radianer mer hensiktsmessig og derfor gjøres denne omregningen.

Når funksjonen har funnet ut hvor langt unna ønsket retning plattformen er, velger den hva plattformen skal gjøre. Dersom den er mindre enn nøyaktighetskonstanten, satt til 0.2 radianer unna ønsket retning kjører den fremover. Dersom verdien er større enn dette vil den svinge. Dersom ønsket retningsendring er negativ svinger den mot høyre. Dette gjør at positive retningsendringer er mot venstre. Når funksjonen har bestemt seg for hva som skal gjøres sendes informasjonen til Node-Red. Dette gjøres ved hjelp av funksjonen «sendinstructions» fra scriptet «sendMessage.py».

4.5.4 Systemets virkemåte

Systemet henter inn dataene RPLIDAR ROS noden har publisert til topic “scan”. Disse dataene blir brukt i HECTOR_SLAM noden til å generere blant annet dataene publisert til topics “map” og “slam_out_pose”. Disse topicene inneholder henholdsvis kartet, som er generert i HECTOR_SLAM av dataene fra Lidaren og faktisk posisjonsdata som også er generert i HECTOR_SLAM noden.

Videre henter Python scriptene inn dataen fra de to sistnevnte topics og bruker disse til å generere veien til målet ved hjelp av A*. Det er disse scriptene som genererer plattformens ønskede vinkel. Dette gjøres ved å se på vinkelen mellom noden plattformen befinner seg i og den neste noden plattformen skal bevege seg til.

Informasjonen om “current” og “wanted” retning blir publisert til topics “current_out_pose” og “wanted_out_pose”. Det er hovedsakelig disse verdiene som er blitt brukt under analyseringen av funksjonstestene til systemet, se kapittel 5 tester.

4.5.5 Utvikling av script for plotting

Scriptet «plotter.py» inneholder to funksjoner som er «make_graph» og «make_graph1». Begge funksjonene er brukt for å plotte informasjon fra systemtester til en graf. Disse leser fra en gitt «.bag»-fil som blir generert under testene ved hjelp av kommandoen «rosbag record» i ROS.

Funksjonen «make_graph» plotter den faktiske og den ønskede retningen til plattformen inn i samme graf. «make_graph1» plotter differansen mellom ønsket og faktisk retning, plattformen presterer bedre desto nærmere null denne grafen er. Disse funksjonene kan brukes for å analysere hvor raskt plattformen responderer på kommandoer fra programmet.

5 Tester

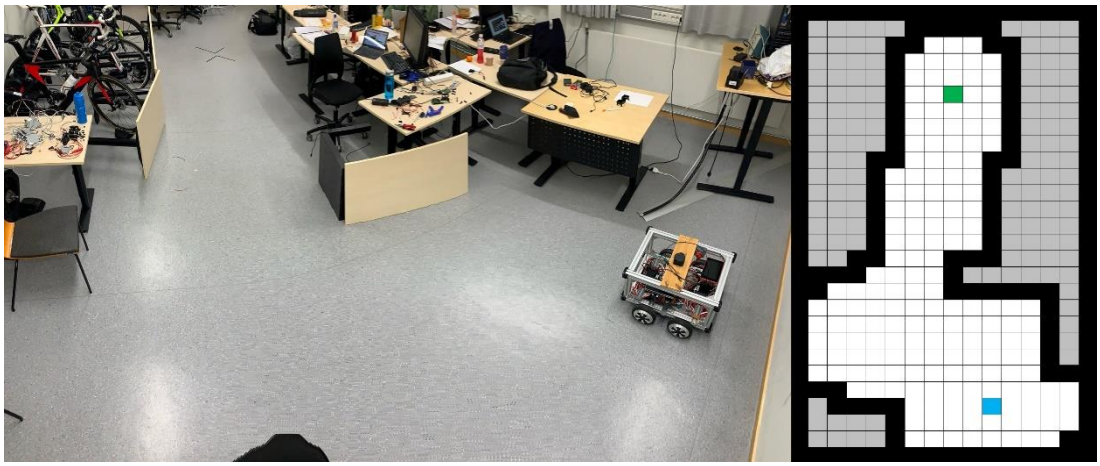
5.1 Tuning av nøyaktighetskonstant

Formål

Teste hvilken nøyaktighetsgrad plattformen sitt styresystem skal bruke for å optimalisere tidsbruk og nøyaktighet.

Metode

Testen gikk ut på at plattformen skulle manøvrere seg autonomt gjennom et klasserom. En del pulter var flyttet slik at plattformen hadde en fri vei fra A til B. Som illustrert i figur 5.1 må plattformen først bevege seg mot sin venstre for deretter å svinge til høyre rundt ett hjørne. Videre måtte den rette seg opp for å kjøre mot målet.

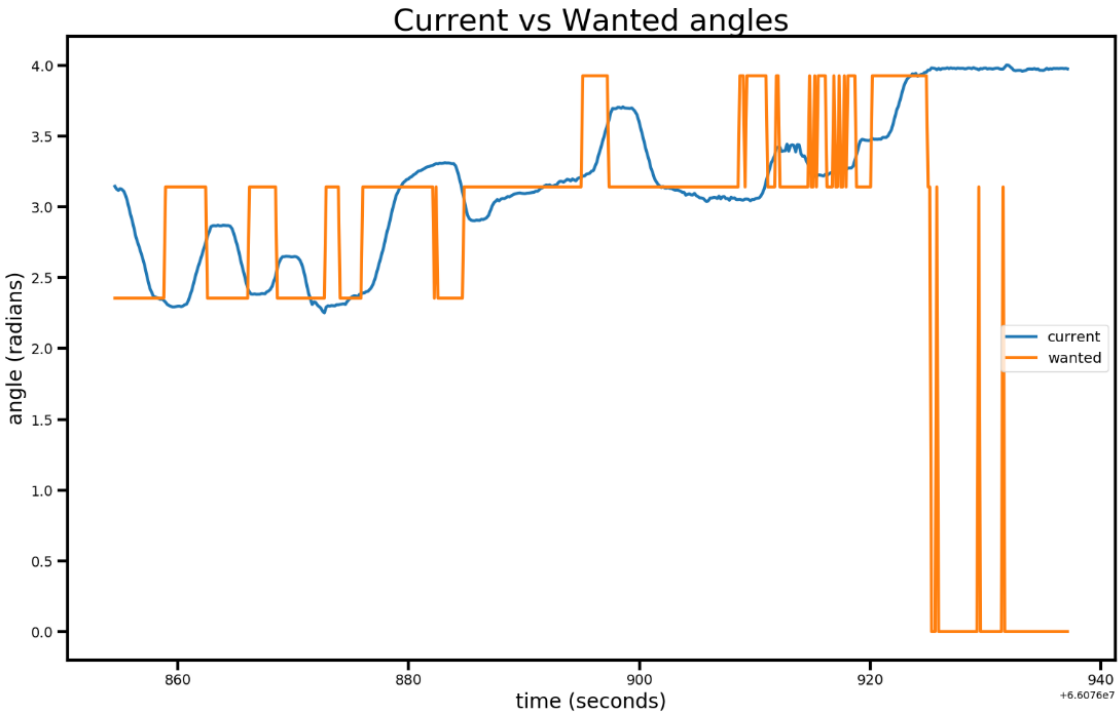


Figur 5.1 - Testløype for testen bilde og punktfremstilling.

Programmet som inneholder denne nøyaktighetsgraden er scriptet action.py. Nøyaktighetsgraden sier noe om hvor nærme ønsket retning plattformen må være før den begynner å kjøre fremover. Dersom denne er for høy vil man ikke kjøre i riktig retning og i verste fall kan en kollisjon oppstå. Dersom den er for lav vil man bruke lenger tid og i verste fall vil programmet aldri bli fornøyd. Dette kan gjøre at plattformen ikke kommer seg til målet.

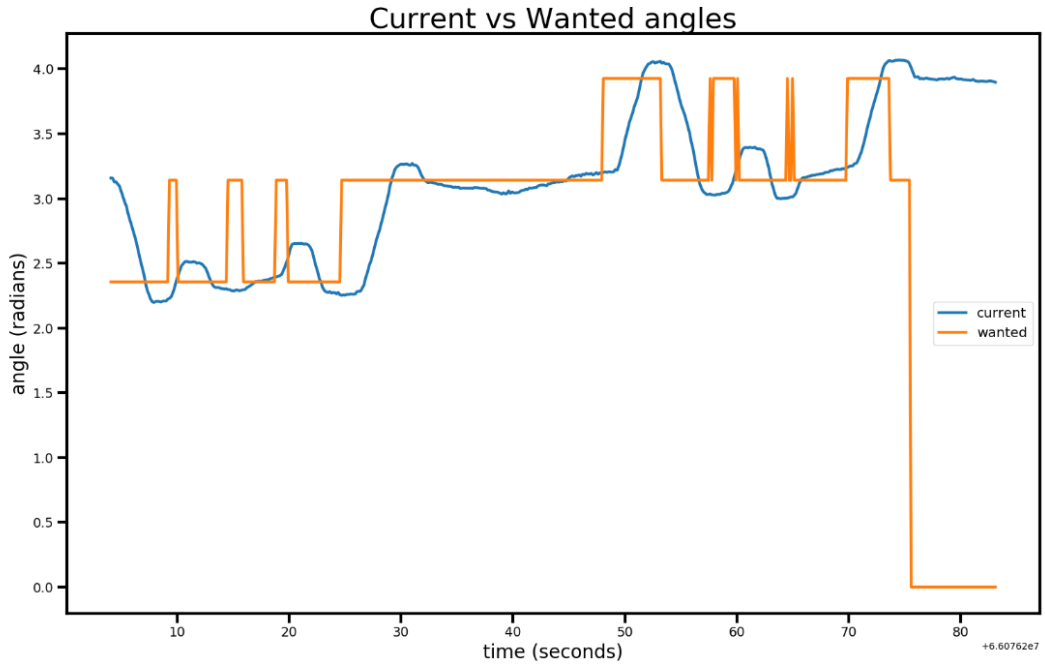
Resultat

Det ble testet tre ulike verdier for nøyaktighetsgraden. Disse tre verdiene er 0.2, 0.1 og 0.05 radianer. Under testene var «drive» hastigheten satt til 54. Dersom denne øker eller minker vil plattformen sin hastighet fremover endres.



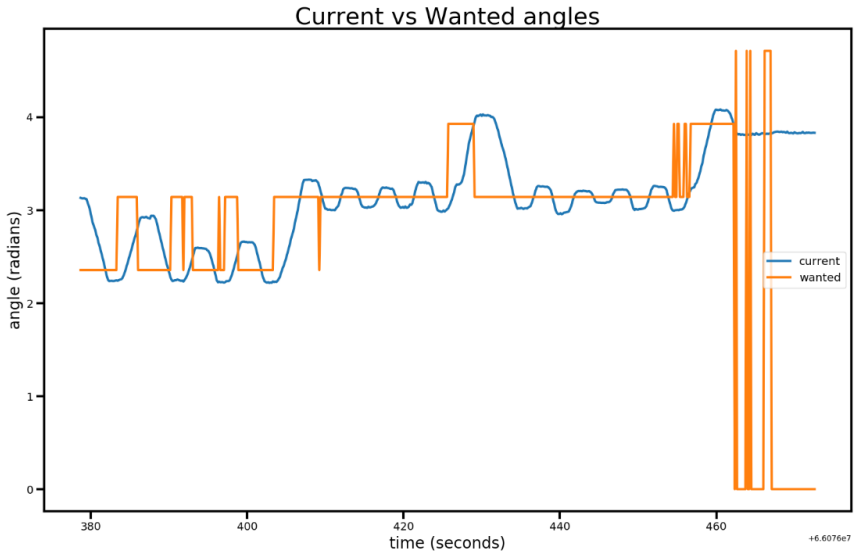
Tabell 5.1 - Nøyaktighetsgrad 0.2 radianer

Med en nøyaktighetsgrad på 0.2 radianer viser grafen fra tabell 5.1 at testen tar ca. 70 sekunder. Plattformen kommer seg til ønsket vinkel på ca. 3 sekunder, og retningsforandringer kommer på ønsket tidspunkt. Det trengs ingen etterjusteringer for svingningene.



Tabell 5.2 - Nøyaktighetsgrad 0.1 radianer

Med en nøyaktighetsgrad på 0.1 radianer viser grafen i tabell 5.2 at testen tar ca. 70 sekunder. Plattformen kommer seg til ønsket vinkel på ca. 3 sekunder. Retningsforandringer kommer på ønsket tidspunkt. Det trengs enkelte etterjusteringer til svingningene.



Tabell 5.3 - Nøyaktighetsgrad 0.05 radianer

Med en nøyaktighetsgrad på 0.05 radianer viser grafen i tabell 5.3 at testen tar ca. 85 sekunder. Plattformen kommer seg til ønsket vinkel på ca. 4 sekunder. Retningsforandringene kommer på ønsket tidspunkt. Det trengs konstante etterjusteringer til svingningene.

Drøfting

Testen viser at en nøyaktighetskonstant på 0.05 radianer er for nøyaktig for vår plattform. Denne høye nøyaktigheten gjør at plattformen bruker lenger tid på å komme til rett vinkel, samt at det kreves konstante etterjusteringer. Dette fører til et tidstap på 15 sekunder eller 21% i forhold til de mer unøyaktige verdiene.

Nøyaktighetskonstantene 0.2 og 0.1 radianer presterer veldig likt. Tidsbruket er tilnærmet identisk, både for hele testen og for svingningen. Det som skiller de to er hvor nærme ønsket retning de befinner seg til enhver tid. Dette kommer verdien 0.1 bedre ut av, som vil hjelpe dersom det er lite plass hvor plattformen skal bevege seg. Videre viser grafen at verdien 0.1 trenger 15 svinger for å komme frem, mens verdien 0.2 trenger 13. Dersom «drive» hastigheten øker vil dette gjøre at verdien 0.2 vil prestere bedre enn 0.1. Konklusjonen av testen blir derfor at nøyaktighetskonstant 0.1 er best for generelt bruk ved lave hastigheter og 0.2 ved høyere hastigheter.

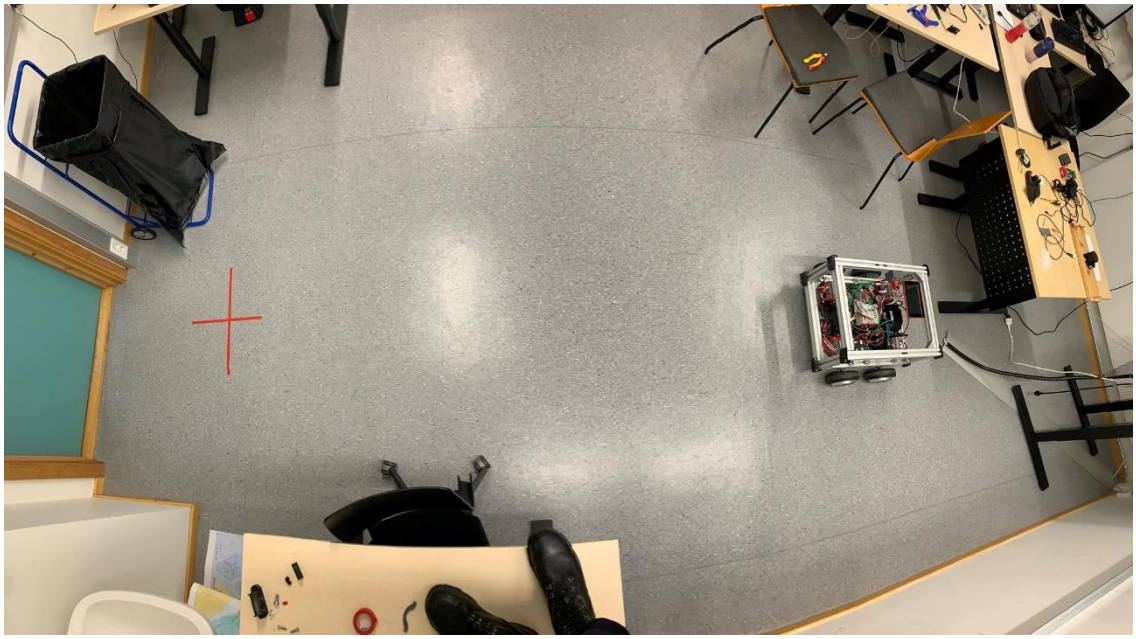
5.2 Navigasjonsevne i dynamiske omgivelser

Formål

Et av kravene for autonomi er evnen til å navigere i et stadig endrende miljø. Denne testen vil kartlegge systemets egenskaper innenfor dette.

Metode

Formålet ble testet ved å sette en løype på 3m som plattformen måtte kjøre. Da den startet å kjøre gikk en person inn i veien for plattformen. Systemet måtte nå registrere endringen i miljøet, og rekalkulere veien mot målet dersom dette fremdeles var mulig. Videre måtte den handle ut i fra denne informasjonen.



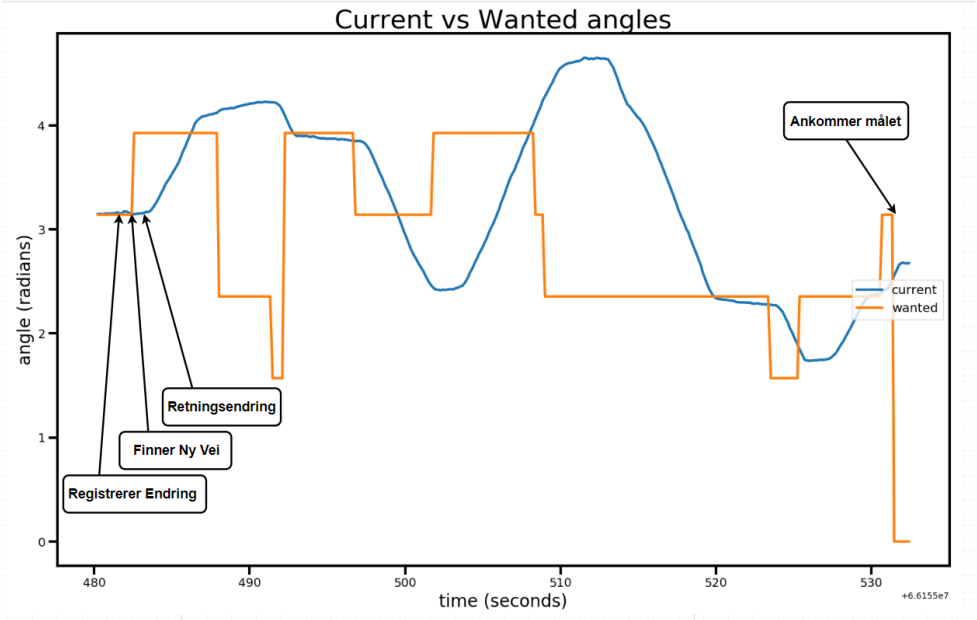
Figur 5.1 - Testoppsett dynamisk miljø

Som vist i figur 5.1 trengte plattformen kun å kjøre rett frem mot målet.

Resultater

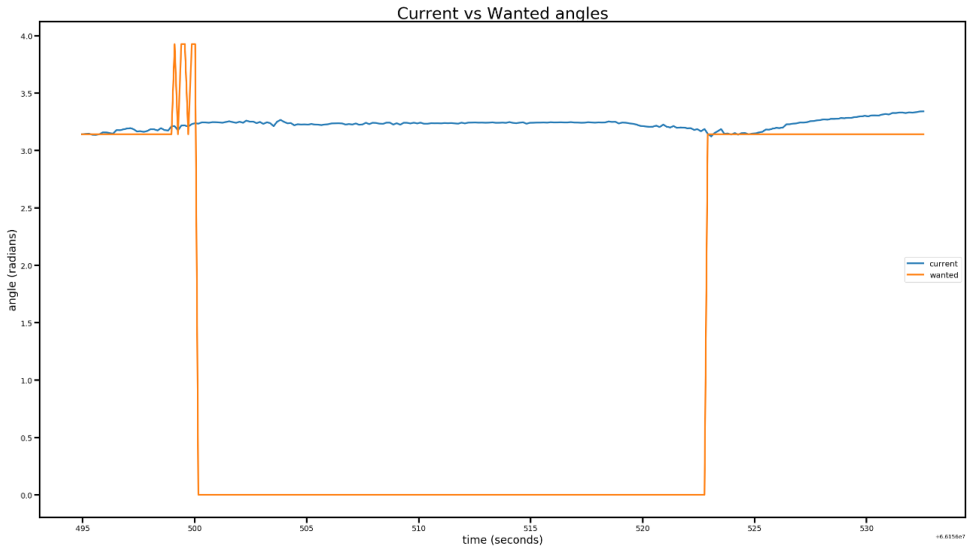
Under den første gjennomføringen skulle en person gå inn ca. en meter foran plattformen. Det skulle være tilstrekkelig plass for plattformen å kjøre rundt, og personen skulle ikke

bevege seg etter å ha stilt seg i veibanen til plattformen.



Tabell 5.4 - Dynamisk endring av vei, hinder

Grafen fra tabell 5.4 viser at programmet reagerer nesten umiddelbart. Programmet finner også en omvei rundt hinderet, og kommer seg til målet. Videre ble det testet hvordan systemet taklet en situasjon der veien til målet ble totalt blokkert for deretter å åpne seg igjen.



Tabell 5.5 - Dynamisk endring av vei, total blokade

Grafen fra tabell 5.5 viser at i likhet med den første testen registrerer systemet endringen i miljøet raskt. Plattformen stopper da det ikke finnes noen vei. Videre flyttes hindrene og det går rundt 15 sekunder før systemet oppfatter dette, og kjører videre mot målet.

Drøfting

Resultatene viser at systemets egenskaper til å unngå dynamiske hindre er gode. Videre viser resultatene at systemet klarer å tilpasse seg en midlertidig blokkert vei. Systemet klarer dette, men tiden som kreves fra veien er klar igjen til systemet oppfatter dette er høy.

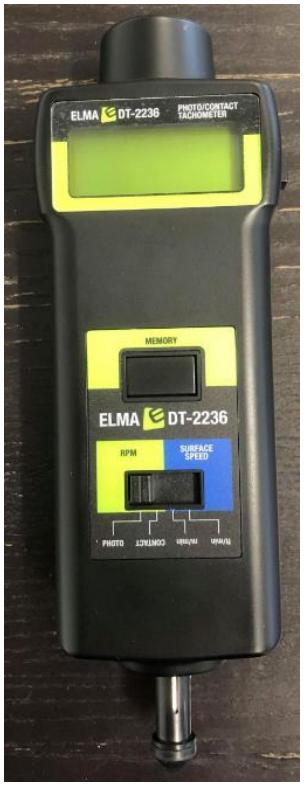
5.3 Test av turtall og fart på motor

Formål

Testen er ment for å teste maks fart av motor med innstillingene på motorkontrolleren.

Metode

Plattformen ble satt på to stoler slik at hjulene kunne spinne fritt uten noe motstand. Farten ble testet med maks pådrag fra PLS- og PS4 styresystemet. Motorkontrollene ble kjørt med 4V, men har mulighet til å dra 36V. Figur 5.4 viser en ELMA DT-2236 turtallsmåler som ble brukt til måling av turtall i RPM.



Figur 5.2 - Turtallsmåler

Resultat

	PLS styringssystem	PS4 styringssystem
Turtall [rpm]	133,3	133,3
Fart [km/t]	4,394	4,394

Tabell 5.6 - Resultatene viser at PLS og PS4 er kalibrert på samme maks hastighet

Drøfting

Testen ble gjennomført uten friksjon fra bakken. Maks hastigheten er satt lav, grunnet oppdateringene til ROS programmet. Med denne hastigheten vil plattformen ikke kunne kjøre lenger enn 20 cm før ny kommando blir gitt. Dette er hensiktsmessig for oss siden det da er lettere å unngå uhell og skader på plattformen. Det anses heller ikke som nødvendig med høyere hastighet for plattformen sitt bruksområde.

5.4 Test av effektforbruk

Formål

Å teste effektforbruk av komponenter og batterikapasitet.

Metode

Det ble brukt et multimeter av typen mprobe 38XR-A med strøm sonde Tektronix A622, som vist i figur 5.5. Multimeteret ble brukt til å måle strømtrekket til komponentene. Testen ble gjennomført med vår satte maks hastighet på 4,39 km/t for realistisk bruk av plattformen.

Figur 5.3 - Multimeter av typen Amprobe 38XR-A med strøm sonde Tektronix A622

Resultat

Et tidsestimat på en time ble brukt for å sjekke hvor lenge komponentene trekker strøm.

Last	Antall	Effekt [W]	Spenning [V]	Strøm [mA]	Tid i bruk [T]	Effekt*tid*antall [Wh]
Motorkontroller	2	10,728	36	298	1	21,456
Ruter	1	3,775	5	755	1	3,775
PLS	1	3,696	24	154	1	3,696
Raspberry Pi	1	5,89	5	1178	1	5,89
Totalt						34,817

Tabell 5.7 - Effektforbuk til komponentene.

Drøfting

Testen viser at for en timers kjøring i 4,394 km/t, vil estimert effekttrekk være på 34,817Wh.

Med en batteripakke på 288Wh, vil plattformen kunne kjøre i 8 timer og 16 minutter. Det er da forutsatt at all kapasitet i batteriene blir brukt til disse komponentene. Siden batteriene kutter spenningstilførselen før de er totalt utladet, vil ikke 8 timer og 16 minutter være tilfellet. Ved normal driftsmønster med Lidar vil ikke motorene trekke strøm kontinuerlig som testen gjorde, noe som vil påvirke effektforbruket. Testen viser med sikkerhet at plattformen kan kjøre kontinuerlig mellom 7-8 timer med en fart på 4,394 km/t og ingen helning.

6 Drøfting

I denne oppgaven har det blitt bygget videre på en allerede eksisterende plattform. Produktet har blitt utviklet til et autonomt system, med fokus på å opprettholde modulariteten til fremkomstmiddelet. Gjennom arbeidet har det blitt gjort noen endringer på plattformen, blant annet ved å erstatte motorkontrollene til en mer robust type. Dette ble gjort etter anbefalinger fra plattformens originale utviklere.

Eksisterende ultralyd sensorer har blitt erstattet med en Lidar. Denne blir brukt til å kartlegge omgivelsene rundt plattformen i den hensikt å vite hvor den skal bevege seg for å komme til angitt mål, uten å kjøre på hindringer i veibanen. Som følge av integreringen av Lidaren har modulene tilhørende de opprinnelige ultralyd sensorene blitt fjernet. Det er enkelt å montere ultralyd sensorene på plattformen igjen, dersom det er ønskelig for videreutvikling av samarbeidet mellom Lidar og sensorene. Det har vært viktig å opprettholde modulariteten i oppgaven til videreutvikling av prosjektet, grunnet et ønske om at plattformen skal kunne brukes til videre testing og muligens i undervisning. Dette henger sammen med at vi bevisst har valgt å bruke lærdommer fra tidligere undervisning i oppgaven.

I bacheloroppgaven "testplattform for autonome systemer" (Flaatten og Tande, 2018) står det i drøftingsdelen at dersom man ønsker en bedre manøvreringsevne anbefales det å gi muligheter til at hoverboard motorene kan svinge individuelt. Med tanke på det utviklede bruksområdet til plattformen var det ikke hensiktsmessig å endre på strukturen til hoverboard motorene. Videre har ingen av styringsformene behov for å endre manøvreringsegenskapene.

Kommunikasjonen mellom de ulike komponentene er ivaretatt av en ruter. Denne gjør at en bruker kan styre Raspberry Pi og PLS via trådløs tilkobling. Ruteren er essensiell for tilkobling av ROS, da programvaren kjøres fra en ekstern PC. Ruteren lager et lokalt nettverk for plattformen som gjør det mulig for alle enheter å koble seg til det trådløse nettverket for å styre plattformen via PLS styringen. Bakdelen med bruk av ruter som ikke er koblet opp mot FHS nettet er at man ikke kan styre plattformen utenfor rekkevidden til ruteren. Dette er noe som kan jobbes videre med dersom ønskelig.

Da det kom til styringsform startet prosessen med å lage et ROS program for autonom styring. Programmet skulle bruke en Lidar sensor for å unngå gjenstander og for å nå ønsket mål. Samtidig ble det arbeidet med en Navio2 som skulle bruke et ferdig

programmert styringssystem for å gjøre samme oppgave. Planen var å bruke Ardupilot i den hensikt å endre koden slik at plattformen kunne unngå objekter ved bruk av ultralyd sensorer. Valget falt hovedsakelig på autonom ferdsel innendørs. Det ble bestemt at ROS programmet ville bli mer nyttig for autonom styring inne, da den kan se vegger og mer av omgivelsene. Dette fordi Lidaren kan registrere gjenstander i samme plan som sensoren. Navioen ville brukt ultralyd sensorer som er billigere enn Lidaren, og som kan bli plassert i forskjellige vinkler på plattformen slik at den kan unngå gjenstander i flere plan. Den bruker også GPS posisjon for å vise hvor den befinner seg. Dette fungerer ikke like bra innendørs hvor signalene blir forstyrret av isolasjon fra vegger og tak. Systemet har også mindre kontroll på hva som er rundt seg selv innendørs med GPS signal, siden det ikke vet hvordan ruten ser ut inne i bygget i forhold til hva som er utenfor. Dette gjør at plattformen vår ikke vil prestere 100% utendørs hvor man arbeider borte fra bebyggelse. Plattformen vil slite med å vite sine begrensninger om hva som er vei og hvor det går opp og ned. Den vil også sannsynligvis velge en rute som inneholder hinder, som den egentlig ikke kan kjøre over og dermed ikke kjøre forbi.

Til videre arbeid er det mulig å integrere ultralyd sensorene for flere plan av styring innendørs med Lidar. En utfordring vil i dette tilfelle være å få ROS programmet til å kommunisere med disse sensorene. For oppgradering til autonomi utendørs vil du måtte integrere en styringsenhet med GPS, hvor man kan ta i bruk Navio2 og Ardupilot. Navio2 ville åpnet for bruk av ardupilot sine programmer, eksempelvis mission planner. Dette ville gitt en annen type ruteplanlegging som enten kan settes sammen med ROS programmet, eller la de jobbe hver for seg slik at bruker kan velge mellom ute - eller innemodus. Navio og ROS er kompatible og kan derfor brukes direkte med hverandre.

Den enkleste av disse tre metodene ville nok vært å bruke de hver for seg som inne- og utemodus. Metoden hvor Navio kobles direkte med ROS er et godt andre alternativ siden man kunne jobbet med GPS punkter sammen med bildet som Lidar bygget med sensoren. Å sette ROS programmet sammen med mission planner ville nok vært den vanskeligste oppgaven siden man må forholde seg til to ulike ruteplanleggingssystemer i samme pakke. Det kan derfor antas at de ville forstyrret hverandre mer enn å arbeide sammen.

Under utviklingen av programvaren i ROS ble det gjort et bevisst valg om å bruke et nodebasert kart. Dette førte til at plattformen kun kan bevege seg mellom nodene og derfor kun rette seg inn etter åtte forskjellige hovedretninger. Et alternativ kan være å se på muligheten for å bruke et åpent kart og et vektorbasert styresystem. I stedet for at dette

systemet velger hvilken av de nærmeste nodene plattformen skal bevege seg etter, vil det generere en vektor mellom to steder i kartet. Et slikt system vil kunne bidra til økt effektivitet og lavere tidsbruk for plattformen.

Målet som blir satt for det autonome systemet i plattformen vår må skrives inn i koden. Dette er ikke spesielt effektivt. En mulighet for videreutvikling innenfor dette temaet kan være å lage et program som gir målet som input fra terminalvinduet. Dette vil gjøre det enklere for en bruker å gi kommandoer til systemet. En muligens enda bedre løsning vil være en mulighet til å velge et grafisk punkt fra kartet som vises i RVIZ. Det finnes allerede funksjoner i RVIZ som publiserer punkter fra kartet, men å integrere dette med resten av systemet har ikke blitt gjort.

Vi støtte på mange utfordringer underveis i oppgaven. Den første utfordringen var ventetiden på deler. Bachelorperioden startet i Mai, men delene kom ikke før i juni, noe som førte til at alt praktisk arbeid måtte utsettes. Dette gjorde at arbeidet med å erstatte motorkontrollerne ble forsinket. En konsekvens av dette er at plattformen ikke ble brukt praktisk før i September. Dette gjorde derimot at integreringen av komponentene i plattformen, samt utviklingen av kildekode ble satt i fokus. Siden Lidaren ikke kom før i juni fikk vi heller ikke satt i gang med ROS programmeringen før August.

En annen utfordring var at vi jobbet med et karosseri som allerede var konstruert. Spesielt siden vi ikke hadde brukt slike komponenter før i utdanningen, og spesielt det elektriske nettverket og koblingen mellom motorkontroller og hoverboard motorene. Dette gjorde at feilsøkingen tok lenger tid enn nødvendig. Siden denne prosessen tok såpass lang tid, lærte vi oss systemet skikkelig fra start. Det har gjort at vi nå raskt og enkelt kan finne feilene som oppstår.

Det var også utfordringer med tid etter hvert som prosessen ble satt i gang. Dette medførte at Navio2 systemet ble droppet. Vi fikk heller ikke tid til å bestille aluminiumsplate for montering av Lidar. På bakgrunn av dette bruker vi nå en treplanke som Lidarens modul. Som følge av dette har tiden brukt til testing av ROS programmet økt betraktelig.

En utfordring som oppstod under manøvreringen av plattformen, var å ha presise nok svingeegenskaper. Siden plattformen er ganske stor og tung er det ofte en restspenning i motorene etter at den har endret retning. Dette resulterer i en forskjell i ønsket - og faktisk grader når plattformen skal begynne å kjøre fremover. Dette er hovedgrunnen til at nøyaktighetskonstanten i det autonome systemet ikke er satt lavere.

For å betrakte hvor autonomt systemet vårt er, tar vi utgangspunkt i Sheridan og Verplank sin tabell, vist i tabell 2.1. Autonomi på nivå syv i tabellen forklares ved at en datamaskin utfører handlinger automatisk, for deretter å informere mennesket. I plattformen setter brukeren selv start – og sluttpunkt, men det er datamaskinen som bestemmer ruten, og utfører denne. Samtidig som den informerer bruker hva den gjør. Det kan derfor argumenteres for at systemet vårt er nummer syv på Sheridan og Verplank sin tabell.

7 Konklusjon med anbefaling

Denne oppgaven har tatt for seg hvordan autonome systemer er bygget opp, hvor fokuset har vært å opprettholde modulariteten i plattformen. Det har også vært viktig for oss å integrere ulike enheter vi har jobbet med i prosjekter på skolen. Bacheloroppgaven har gitt oss en forståelse av hvordan autonome systemer fungerer. Dette er viktig, grunnet det økende fokuset på denne type systemer.

Vi mener at plattformen kan brukes i undervisning og i fremtidige bacheloroppgaver, da det er tilrettelagt for å kunne legge til ulike moduler på PLS, eller utvide bruksområdet til eksempelvis å kjøre utendørs. Det er også meget relevant opp mot prosjekter på skolen, da det gir en god innsikt i hvordan et konkret autonomt system er bygget opp.

Gjennom bacheloroppgaven har modulariteten i plattformen blitt testet, moduler har blitt fjernet og motorkontroller har blitt erstattet. Dette viser til at det er enkelt og tidseffektivt å bytte ut ulike komponenter, som gjør at endringer i plattformen ikke nødvendigvis trenger å være den mest krevende oppgaven.

Den største begrensningen i plattformen i skrivende stund er at bruker ikke har mulighet til å kjøre plattformen autonomt utendørs. Mulighetene for dette har blitt undersøkt og det konkluderes med at det kan integreres en Navio2 på plattformen. Dette gir plattformen muligheter til å skifte mellom å kjøre autonomt ute og inne, som øker bruksområdet betraktelig. På bakgrunn av dette anbefaler vi å gjøre plattformen i stand til å kjøre autonomt utendørs, med bruk av GPS posisjon, samt Lidar.

Det var utfordrende for det autonome systemet å detektere eksempelvis trapper som går nedover, siden Lidaren kun detekterer objekter i ett plan. For å løse denne utfordringen ble det vurdert å bruke ultralyd sensorer. Vi kom derimot ikke så langt i prosessen og mener dette er noe som kan gjøre den autonome plattformen enda bedre. Testene viser at det autonome systemet bruker tid på å se objekter som flytter seg.

Videre mener vi at systemet som en helhet har løst de kravene som ble satt innledningsvis i kravspesifikasjonen til plattformen:

- Plattformen klarer å kjøre fra en startposisjon til en sluttposisjon ved hjelp av ROS programmet og Lidar som sensor.
- Plattformen kan unngå objekter i veibanen og kjøre rundt dem, men har et forbedringspotensialet ved at den kan fryse eller slite med å lage en ny rute når veibanen er helt stengt.

- Farten på plattformen er regulerbar ved å justere på spenningen til motorkontrolleren og konfigurasjonene til hoverboard motorene. Det ble valgt en hastighet på 4 km/t for å bruke ROS programmet mest optimalt. En løsning til forbedring kan være en egen justeringsknapp på PLS styringspanelet for motorkontroller slik at du kan justere tempoet ut ifra hva du ønsker å oppnå med plattformen.
- Plattformen har en egen overvåkingsside på PLS som tar for seg overvåkingen av systemet.
- Plattformen har tre forskjellige styringsformer. Den kan styres gjennom en PS4 kontroller, gjennom PLS og via et ROS program. Hovedfokuset var på ROS, som er det autonome styringssystemet. De to andre er for manuell styring.
- Tidligere bestod brukergrensesnittet til plattformen av en PS4 kontroller og PLS. Disse trengte derimot endringer og oppgraderinger, da deres fokus var mer praktisk rettet. Dette gjelder spesielt PS4 kontroller som nå blir styrt direkte av Node-Red istedenfor å gå via PLS.

På bakgrunn av oppgavens tema og de resultatene som ble oppnådd, mener vi at modulariteten i plattformen er vedlikeholdt, samt at det er laget et konsept som er godt egnet i undervisning for å få et større innblikk i hvordan fremtidens systemer virker.

8 Bibliografi

ArduPilot Dev Team

2018. Archived: Object Avoidance. Hentet 02.12.19 fra:

http://ardupilot.org/rover/docs/rover-object-avoidance.html?fbclid=IwAR3CMB3LLtsAkhB04zb_xnAEpQd4f8gb78v6Fp34qFbAJV-4EYF4g2Q972c

Carnino, Claudio

2016. Control the Raspberry Pi 2/3 GPIO pins with Swift 3.0 on Ubuntu 16.04. Hentet 02.12.19 fra:

https://medium.com/@ccarnino/control-the-raspberry-pi-2-3-gpio-pins-with-swift-3-0-on-ubuntu-16-04-c66ada06efe

Collins, Danielle

2018. What's the difference between cogging torque and torque ripple? Hentet 18.10.19 fra:

https://www.motioncontroltips.com/whats-the-difference-between-cogging-torque-and-torque-ripple/

Components 101

2018. Arduino Uno. Hentet 14.10.19 fra:

https://components101.com/microcontrollers/arduino-uno

Dattalo, A.

2018. ROS/Introduction. Hentet 25.05.19 fra:

http://wiki.ros.org/ROS/Introduction

Elektrofag.info

2019. PT 100 element. Hentet 02.12.19 fra:

http://w3.elektrofag.info/instrumentering/pt100?fbclid=IwAR0lRUxwWp9yzdfmM4ZkBrxJwHX_ADUm9-0UnpUnUhN-TikGE4oluRouC5b0

Emlid

2015. Bilde Navio2. Hentet 23.06.19 fra:

http://emlid.com/wp-content/uploads/2015/12/Navio2-features.jpg

Flaatten, Sondre og Tande, Martin

2018. Testplattform for autonome systemer. Hentet 20.05.19 fra:

https://www.getsky.no/p/get-60197696/_9f0f0499e2384a58a790b878990e1c2e

Fradj, J.

2011. Introduction to A* pathfinding. Hentet 19.11.19 fra:

https://www.raywenderlich.com/3016-introduction-to-a-pathfinding

Hareide, Odd Sveinung.

2018. Fremtidens autonome ubemannede kapasiteter i Sjøforsvaret, *Necesse* (Vol. 3, 2.ut., 123-148). Hentet 27.05.19 fra:

https://pdfs.semanticscholar.org/7729/cb266d3879248a97933a2802b90b644c8fa9.pdf?_ga=2.102546630.613649696.1575015892-1461460232.1575015892

Hassel, Daniel

2019. Project LIBERTY. Hentet 02.12.19 fra:

https://github.com/thedanielhassel/Project-LIBERTY

Hofoss, Espen.

2019. Autonomi vil gjøre Forsvaret bedre. Hentet 27.10.19 fra:

https://forskning.no/forsvarets-forskningsinstitutt-partner-roboter/autonomi-vil-gjore-forsvaret-bedre/1564270

Hollingworth, G.

2015. The Eagerly Awaited Raspberry Pi Display. Hentet 12.06.19 fra:

https://www.raspberrypi.org/blog/the-eagerly-awaited-raspberry-pi-display/

Huang, J.

2016. ROS tutorial #2: Publishers and subscribers [videofil]. Hentet 25.05.19 fra:

https://www.youtube.com/watch?v=bJB9tv4ThV4&t=

Jaszczolt, Christopher

2017. Understanding permanent magnet motors. Hentet 02.12.19 fra:

https://www.controleng.com/articles/understanding-permanent-magnet-motors/

Leddartech

2019. Why Lidar. Hentet 25.11.19 fra:

https://leddartech.com/why-lidar/

Lin, Y.

2018. ROS Python vs C++. [bildefil]. Hentet 25.08.19 fra:

http://www.theconstructsim.com/infographic-start-ros-python-ros-cpp/

Motion control online marketing team

2017. Brushed DC motors Vs. Brushless DC motors. Hentet 27.05.19 fra:

https://www.motioncontrolonline.org/blog-article.cfm/Brushed-DC-Motors-Vs-Brushless-DC-Motors/24

Motekpower

2018. Prinsippet om PWM pulsbredde-modulering. Hentet 23.11.19 fra:

http://no.sinosolarcharger.com/info/the-principle-of-pwm-pulse-width-modulation-26684072.html

Patel, A.

2019. Introduction to A*. Hentet 19.11.19 fra:

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Pettersen, Andre

2017. Framtiden er ubemannet – Forskningsleder Andre Pettersen forteller, *Forsvarets Forskningsinstitutt*. Hentet 11.10.19 fra:

https://vimeo.com/212736735

Refvik, Rita

2013. Temperaturmåling – slik virker PT100 elementet. Hentet 02.12.19 fra:

https://www.tu.no/artikler/temperaturmaling-slik-virker-pt100-elementet/218877?fbclid=IwAR3cXR-P9yg8V_wX66ouTnd7U6HE8HKrGD_B3K3bm3ymd2H2EG61Mxgk21k

Robopeak

2019. ROS pakke – rplidar_ros. Hentet 11.09.19 fra:

https://github.com/robopeak/rplidar_ros

ROS wiki

2019. Hovedside. Hentet 18.11.19 fra:

http://wiki.ros.org/

ROS wiki

2019. Ubuntu install of ROS Melodic. Hentet 11.09.19 fra:

http://wiki.ros.org/melodic/Installation/Ubuntu

ROS wiki

2019. Installing and configuring your ROS environment. Hentet 11.09.19 fra:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

Rouse, Margaret

2019. UDP (User Datagram Protocol). Hentet 15.10.19 fra:

https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol

RS-online

2019. RS PRO 2 wire PT100 Sensor, - 50ºC + 500 ºC max, 10 mm Probe Length

https://no.rs-online.com/web/p/platinum-resistance-temperature-sensors/3629799/?fbclid=IwAR3LMxMyaD3TmMxe3cJJU36I_PAYcAiv3-TEIMPnVCcWRp20iVtSnEH5slE

Simon, Matt

2019. Inside the Amazon Warehouse Where Humans and Machines Become One. Hentet 20.10.19 fra:

https://www.wired.com/story/amazon-warehouse-robots/

Technische Universität Darmstadt

2019. ROS pakke – hector_slam. Hentet 11.09.19 fra:

https://github.com/tu-darmstadt-ros-pkg/hector_slam

Tellez, R.

2019. Should I learn ROS with Python or with C++?. Hentet 25.08.19 fra:

http://www.theconstructsim.com/learn-ros-python-or-cpp/

Weigl, Oscar

2019. Hoverboard guide and remote control setup guide. Hentet 11.09.19 fra:

https://docs.odriverobotics.com/hoverboard

Weisstein, Eric W.

2019. Quaternion. Hentet 18.11.19 fra:

http://mathworld.wolfram.com/Quaternion.html

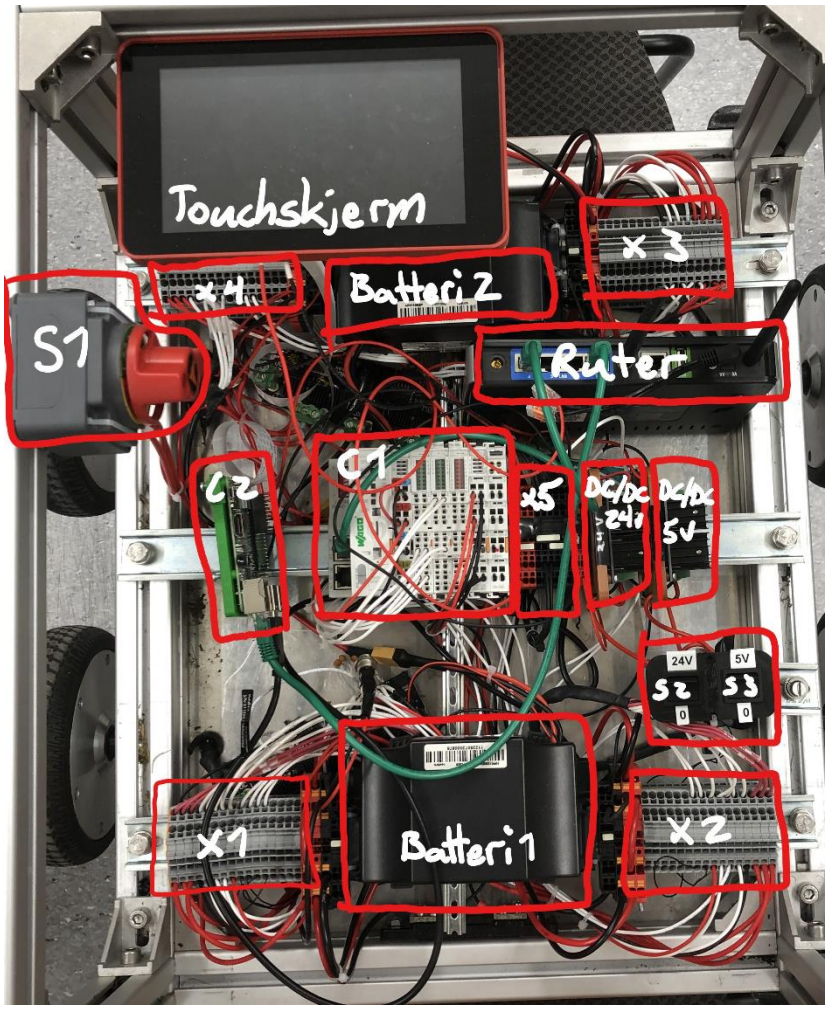
Wikipedia

2019. Pathfinding. Hentet 30.11.19 fra:

https://en.wikipedia.org/wiki/Pathfinding

A. Dokumentasjon

A.1 Arrangementstegning



Figur A.1 - Oversiktsbilde med forklaring

S1: Hovedstrømbryter

S2: Bryter 24V

S3: Bryter 5V

X1-X5: Rekkeklemmer

C1: PLS

C2: Raspberry Pi

Figur A.2 - RPLIDAR modul

A.2 Rekkeklemmetabeller

X1		
M1: C	1	ODRV1: C
M1: B	2	ODRV1: B
M1: A	3	ODRV1: A
M1: Hall +	4	ODRV1: Hall +
M1: Hall C	5	ODRV1: Hall C
M1: Hall B	6	ODRV1: Hall B
M1: Hall A	7	ODRV1: Hall A
M1: Hall -	8	ODRV1: Hall -
	9	
	10	
ODRV1: PT	11	PLS: RTD 2.1
ODRV1: PT	12	PLS: RTD 2.9
	13	
	14	
	15	
	16	
36 V	17	
0 V	18	
0 V	19	
0 V	20	

X2		
M2: C	1	ODRV1: C
M2: B	2	ODRV1: B
M2: A	3	ODRV1: A
M2: Hall +	4	ODRV1: Hall +
M2: Hall C	5	ODRV1: Hall C
M2: Hall B	6	ODRV1: Hall B
M2: Hall A	7	ODRV1: Hall A
M2: Hall -	8	ODRV1: Hall -
	9	
	10	
	11	
	12	
	13	
	14	
	15	
	16	
36 V	17	
0 V	18	
0 V	19	
0 V	20	

X3		
M3: C	1	ODRV2: C
M3: B	2	ODRV2: B
M3: A	3	ODRV2: A
M3: Hall +	4	ODRV2: Hall +
M3: Hall C	5	ODRV2: Hall C
M3: Hall B	6	ODRV2: Hall B
M3: Hall A	7	ODRV2: Hall A
M3: Hall -	8	ODRV2: Hall -
M3: PT	9	PLS: RTD 1.5
M3: PT	10	PLS: RTD 1.13
ESC 3: PT	11	PLS: RTD 2.5
ESC 3: PT	12	PLS: RTD 2.13
	13	
	14	
	15	
	16	
36 V	17	
36 V	18	
0 V	19	
0 V	20	
0 V	21	

X4		
M4: C	1	ODRV2: C
M4: B	2	ODRV2: B
M4: A	3	ODRV2: A
M4: Hall +	4	ODRV2: Hall +
M4: Hall C	5	ODRV2: Hall C
M4: Hall B	6	ODRV2: Hall B
M4: Hall A	7	ODRV2: Hall A
M4: Hall -	8	ODRV2: Hall -
M4: PT	9	PLS: RTD 1.7
M4: PT	10	PLS: RTD 1.15
	11	
	12	
	13	
	14	
	15	
	16	
36 V	17	
36 V	18	
0 V	19	
0 V	20	
0 V	21	

Tabell A.1 – Rekkeklemmetabeller

A.3 Playstation 4 kontroller

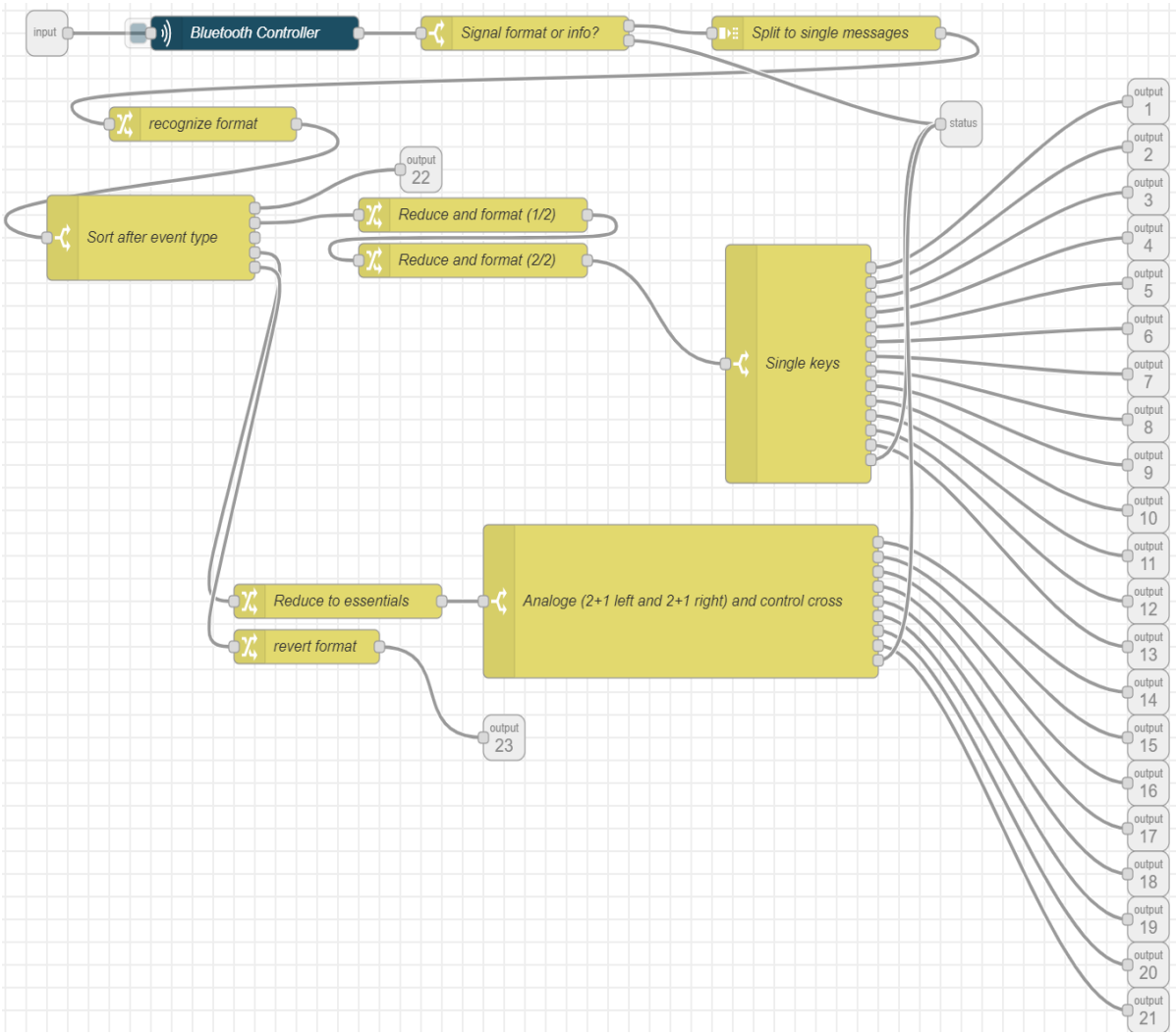
Figur A.3 - Oversiktsbilde over knapper og navn

PS4 kontroller	**Funksjon**
UREGULERBAR	**FART**
BTN_NORTH	Fremover
BTN_WEST	Venstre
BTN_EAST	Høyre
BTN_SOUTH	Bakover
BTN_TR	STOPP
REGULERBAR	**FART**
BTN_THUMBR	Venstre
BTN_THUMBL	Høyre
BTN_TR2	Fremover
BTN_TL2	Bakover

Tabell A.1 - Knapper brukt på PS4 kontrolleren

OUTPUT	**PS4 KONTROLLER KNAPP**	**Visuelt bilde**
1	BTN_MODE	
2	BTN_SELECT	
3	BTN_START	
4	BTN_NORTH	
5	BTN_SOUTH	
6	BTN_WEST	
7	BTN_EAST	
8	BTN_TR	
9	BTN_TR2	
10	BTN_TL	
11	BTN_TL2	
12	BTN_THUMBL	
13	BTN_THUMBR	
14	ABS_X	
15	ABS_Y	
16	ABS_RX	
17	ABS_RY	
18	ABS_Z	
19	ABS_RZ	
20	ABS_HAT0X	
21	ABS_HAT0Y	

Tabell A.2 - Oversiktsbilde over output og tilhørende navn



Figur A.4 - Subflow Dualshock PS4 kontroller

```
#import evdev
from evdev import InputDevice, categorize, ecodes

#import sys

#creates object 'gamepad' to store the data
#you can call it whatever you like
#deviceselect = sys.argv
gamepad = InputDevice('/dev/input/event3')

#prints out device info at start
print(gamepad)

#evdev takes care of polling the controller in a loop
for event in gamepad.read_loop():
    print("category: " + str(categorize(event)) + ", event.type: " +
    str(event.type) + ", event.value: " + str(event.value) +
    ", event.code: " + str(event.code))
```

Figur A.5 - Viser innholdet i bluetoothminimal.py

A.4 Oppstartsprosedyre

For å kunne starte plattformen må man først skru på hovedstrøms bryteren. For så å skru på bryter for 24V og 5V DC-DC omformere. Neste steg er å skru på PS4 kontrolleren slik at den ikke blinker men lyser blått. Du må nå vente på at touchscreen skjermen har fått opp webvisu/kontrollpanelet. For styring av plattformen må du gå inn på styring på touch-skjermen. Her kan du nå velge mellom PS4-, PLS- eller ROS-styring.

PS4-styring og PLS-styring er helt klar til styring, mens ROS trenger noen flere trinn.

For oppstart av ROS må man ta i bruk PCen med ROS pakken installert, installasjonspro-sedyrer finner du under ROS i rapporten. Koble PCen til nettverket "BOAT", uten dette kan ikke PCen snakke med plattformen. Deretter kobler man til Lidaren til PCen med en USB kabel. Det er viktig at man her bruker en Lidar av typen RPLIDAR A1. Dersom man bruker en annen Lidar må man selv installere pakker for dette. I et terminalvindu skriv kommandoene:

sudo chmod 666 /dev/ttyUSB0

roslaunch rplidar_ros rplidar.launch

Dette programmet må kjøre i bakgrunnen hele tiden. Deretter åpne et nytt terminalvindu og skriv inn en av to kommandoer:

roslaunch hector_slam_launch rover.launch

roslaunch hector_mapping mapping_rover.launch

Den første kommandoen vil kjøre programmet og illustrasjonsverktøyet RVIZ slik at man kan monitorere det som skjer. Dersom man ikke ønsker dette eller PCen ikke er kraftig nok kan man benytte seg av den andre kommandoen. Denne vil kun kjøre pro-grammet uten illustrasjoner for brukeren. Dette programmet må kjøre i bakgrunnen hele tiden. Til slutt må man i et tredje terminalvindu skrive inn kommandoen:

rosrun test_code main.py

Dette vil starte selve hjernen til det autonome systemet og plattformen vil øyeblikkelig begynne å navigere mot målet som er satt. Dersom man vil endre målet må man inn i kildekoden til scriptet main.py og endre x og y verdiene. Dersom man stopper program-met og plattformen ikke stopper kan man benytte kommandoen:

rosrun test_code stop.py

Denne kommandoen sender et stopp signal til ROS delen i Node-Red.

Hvis du nå ønsker å styre plattformen fra andre enheter enn touchskjermen og PS4-kontrolleren kan du koble deg opp på BOAT-nettet og styre enheten fra for eksempel en PC eller en mobil. Her er ikke kontrollpanelet skalert for disse enhetene, men du har fortsatt mulighet til å styre. IP-adressen du må skrive i din nettleser er 192.168.84.182/webvisu/webvisu.htm.

Hvis du ønsker å gjøre endringer i Node-Red finner du det på IP-adressen 192.168.84.254:1880.

A.5 Installering av ROS

For å kunne bruke Lidar og HECTOR_SLAM ble det enkleste å installere ROS MELODIC på en PC med UBUNTU 18 som operativsystem. Dette er fordi denne versjonen av ROS er kompetitiv med alle ROS pakkene som skulle brukes, samt med UBUNTU 18. Oppstartsprosedyre finnes under kapittel A.4 og dekkes derfor ikke i dette kapittelet. Installasjonen av ROS MELODIC starter med å konfigurere mappestrukturen til UBUNTU. Dette gjøres ved å gå inn under «Ubuntu Software» innstillingene, for deretter å huke av samtlige alternativer under «Downloadable from the internet». Videre må man legge til kilden for ROS i ubuntu sin kildeliste «sources.list». Dette gjøres ved hjelp av kommandoen:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

Det neste man må gjøre er å legge til nøklene for nedlastning fra denne serveren. Dette gjøres ved hjelp av kommandoen:

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

Deretter må man sjekke at DEBIAN pakkene som er installert er de nyeste. Dette gjøres ved hjelp av kommandoen:

sudo apt update

Når dette er gjort kan man starte selve installasjonen. Det første man da må gjøre er å velge hvilken pakke man vil installere. Dersom man skal kjøre systemet fra en pc og ikke en Raspberry Pi er nok det enkleste å installere den fulle pc versjonen av ROS MELODIC. Siden ROS skulle kjøres fra en pc var dette valget vi tok, noe som var enkelt med kommandoen:

sudo apt install ros-melodic-desktop-full

Dette installerer blant annet pakken til RVIZ som ble brukt for å visualisere hvordan det autonome systemet fungerer. Dersom man ikke ønsker å bruke like mye plass kan man installere den vanlige desktop versjonen av ROS. Dette kan gjøres med å bruke kommandoen:

sudo apt install ros-melodic-desktop

Når man har installert ROS på sin maskin må man initialisere ROSDEP. Dette gjøres ved hjelp av kommandoene:

sudo rosdep init

rosdep update

ROSDEP hjelper oss å installere system avhengigheter for de ulike kildene man bruker i prosjektet sitt. ROSDEP er også nødvendig for å kjøre enkelte kjerneelementer i ROS. Videre var det for vår del ønskelig å slippe å henvise til den lokale ROS installasjonen hver gang man åpnet et nytt terminalvindu. Dette løses enkelt ved å legge dette inn som en betingelse i filen «.bashrc» som er oppstartsfilen for et terminalvindu i UBUNTU. Dette gjøres ved å bruke kommandoen:

echo «source /opt/ros/melodic/setup.bash» >> ~/.bashrc

Deretter åpnes et nytt terminalvindu eller man skriver inn kommandoen:

Source ~/.bashrc

Dette er fordi terminalvinduet bare leser oppstartsfilen når det starter og denne kommandoen får det til å lese den igjen. Dette fører til at terminalvinduet får de egenskapene som har blitt installert. Det siste som trenger å gjøres er å installere avhengighetene ROS trenger for å bygge pakker. Dette gjøres ved å bruke kommandoen:

sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential

Når dette er gjort er ROS installert. Videre må et miljø for ROS installeres på vår pc. Man starter med å lage en mappe med navn «catkin_ws». Dette gjøres ved hjelp av mkdir kommandoen:

mkdir -p ~/catkin_ws/src

cd ~/catkin_ws

Nå har en mappe som skal inneholde arbeidsmiljøet til ROS blitt laget. Deretter har man byttet mappe slik at man befinner seg inne i denne mappen. Nå må miljøet lages ved å bruke kommandoen:

catkin_make

Nå har miljøet blitt laget. Det neste man må gjøre for at ROS skal fungere er å legge dette til i kildene til terminalvinduet. Dette gjøres på en lignende måte som etter initialiseringen av ROSDEP med kommandoen:

echo «source devel/setup.bash» >> ~/.bashrc

Dette må også gjøres for det gjeldene terminalvinduet:

source devel/setup.bash

Det finnes nå en fungerende installasjon av ROS MELODIC på vår UBUNTU 18.4 data-maskin. Dette betyr at grunnpilaren i vårt autonome system er installert. Det neste steget for implementeringen av systemet vårt, er pakker for Lidaren og HECTOR_SLAM. Disse pakkene finnes allerede ferdig lagde for vår versjon av ROS. Disse ble derfor benyttet.

A.5.1 RPLIDAR ROS Pakke

Denne pakken er laget av Robopeak og er åpen kildekode (Robopeak, 2019). For å installere denne pakken må man først bevege seg til kildemappen inne i sitt ROS miljø. Dette gjøres ved å bruke kommandoen:

cd catkin_ws/src

Deretter må rplidar_ros mappen fra github klones til denne kildemappen. Dette gjøres enkelt ved hjelp av kommandoen:

git clone github.com/robopeak/rplidar_ros

Når denne mappen er klonet til kildemappen vår må miljøet lages igjen. Dette gjøres som tidligere nevnt ved hjelp av kommandoen:

catkin_make

Dette er en kommando som må kjøres hver gang man endrer pakkestrukturen i sitt ROS miljø. Eksempler på dette er når man legger til eller fjerner pakker. Man må også kjøre denne kommandoen hver gang man endrer avhengighetene til en ROS pakke. Dersom man skal kunne kjøre den nye pakken må man huske å starte et nytt terminalvindu eller kjøre kommandoen:

source devel/setup.bash

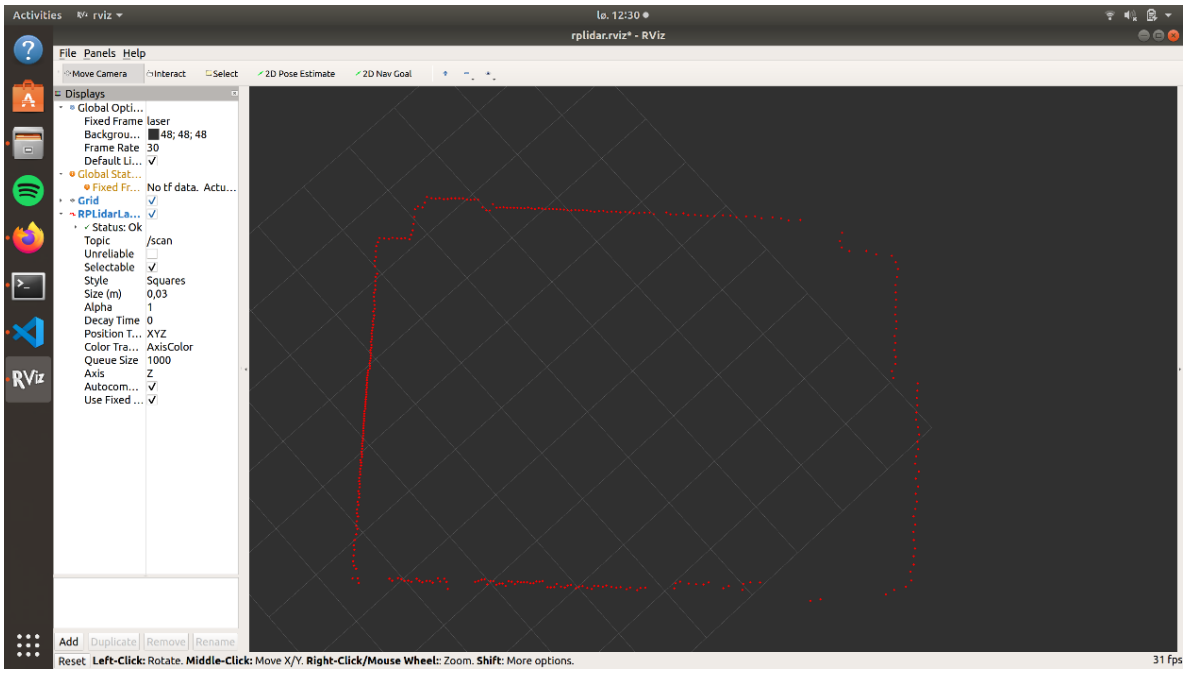
Man kan nå kjøre de forskjellige innebygde programmene for RPLIDAR A1. Når man kobler Lidaren til en pc må man kjøre kommandoen:

sudo chmod 666 /dev/ttyUSB0

Dette er for å gi pcen tilgang til å skrive til USB utgangen, noe den må for å sende kommandoer til Lidar. Denne kommandoen må kjører hver gang Lidaren kobles til eller fra pcen. Dersom man vil sjekke at Lidaren fungerer kan følgende kommando kjøres:

roslaunch rplidar_ros view_rplidar.launch

Dette vil åpne RVIZ og visualisere hva Lidar «ser».



Figur A.6 - Lidar laserscan visualisert i RVIZ

Som figur A.7 viser fungerer vår Lidar. Punktene Lidaren oppfatter med sin laserskanning er visualisert i rødt. Figuren viser at Lidaren befinner seg inne i et klasserom med fire vegger. Lidar noden når den er aktiv har bare to funksjoner. Den ene er en utregning av hvor disse punktene laseren har registrert er i forhold til dens egen posisjon. Den andre funksjonen publiserer denne informasjonen til et ROS topic. Dette topic'et heter «/scan» og det er dette topic'et som er RPLIDAR nodens viktigste funksjon. Dette topic'et skal nemlig brukes videre.

A.5.2 HECTOR_SLAM ROS Pakke

Til nå har systemet bare en rekke punkter fra Lidaren, for å generere et dynamisk kart ut fra denne dataen trengs noe mer. Det ble valgt å bruke HECTOR_SLAM. Dette er en algoritme for «simultaneous localization and mapping». Denne ROS pakken er åpen kildekode (Technische Universität Darmstadt, 2019). Installasjonen til denne pakken følger den samme starten som for RPLIDAR pakken. Endre mappe i terminalvinduet til kildemappen:

cd catkin_ws/src

Deretter må github mappen for HECTOR_SLAM klones inn i kildemappen:

git clone https://github.com/tu-darmstadt-ros-pkg/hector_slam

Miljøet blir laget ved hjelp av:

catkin_make

Deretter startes terminalvinduet på nytt eller man kjører kommandoen:

Source /devel/setup.bash

Nå har HECTOR_SLAM noden i ROS blitt installert, men for at denne skal fungere med de programmene som skal lages i Python må det legges til noen filer. Den første filen som må legges inn er filen «rover.launch». Denne filen ligger som et vedlegg til oppgaven og skal legges inn i mappen:

catkin_ws/src/hector_slam/hector_slam_launch/launch

Dette er oppstartsfilen til vår modifiserte HECTOR_SLAM node. Videre må det også legges til to modifiserte filer for vår HECTOR_SLAM node. Den første filen som må legges til er filen «rviz_rover.rviz». Denne filen må legges inn i mappen:

catkin_ws/src/hector_slam/hector_slam_launch/rviz_cfg

Denne filen inneholder informasjonen RVIZ trenger når programmet starter for å vise de rette vinduene til brukeren. Filen inneholder også informasjon om hvilke topic RVIZ skal abonnere på og hvordan denne informasjonen skal fremstilles. Den siste filen som trenger å legges til er filen «mapping_rover.launch», som må legges inn i mappen:

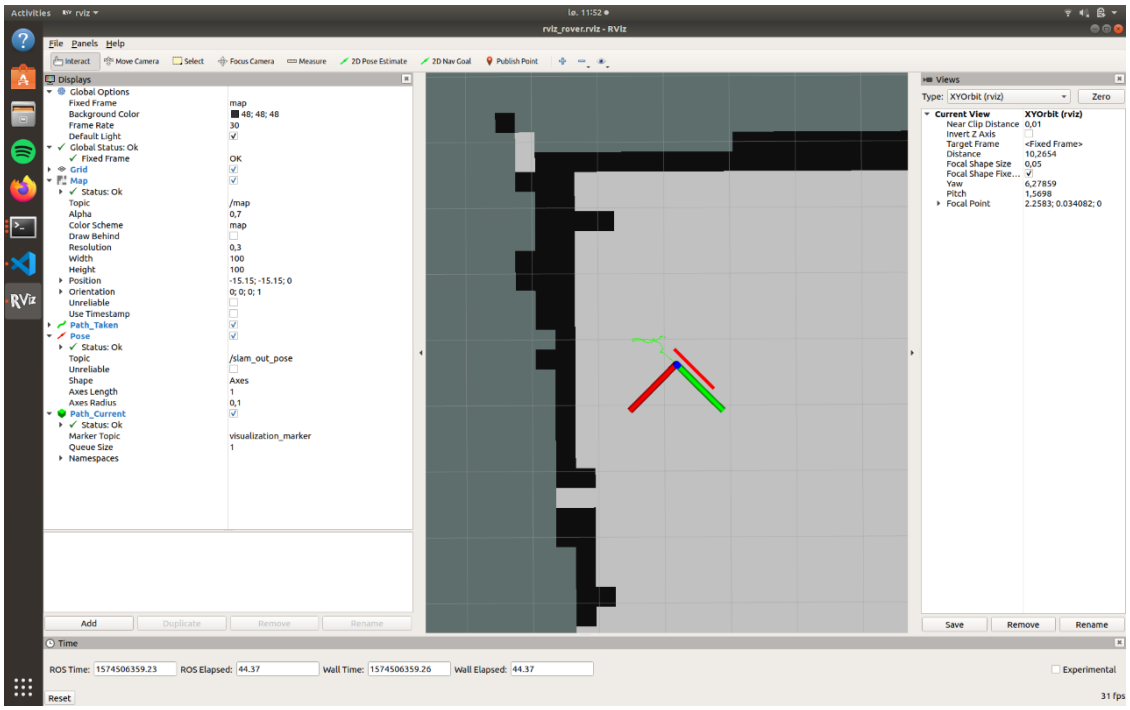
catkin_ws/src/hector_slam/hector_mapping/launch

Dette er oppstartsfilen for selve kartleggingen av verdenen roboten ser. Det er med andre ord denne filen som inneholder alle parameterne algoritmen trenger for å regne ut hvor plattformen befinner seg og hvordan miljøet faktisk er.

Nå har alle pakkene som trengs for implementeringen av vårt autonome system blitt installert. Dersom man kjører både RPLIDAR noden og HECTOR_SLAM noden vil systemet generere et kart som kan leses av i et Python program. Dette kartet kan visualiseres ved å kjøre kommandoen:

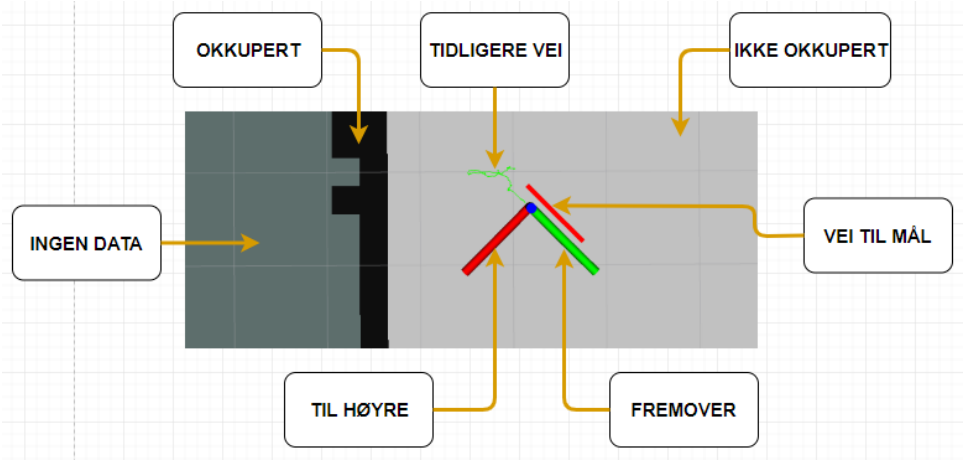
roslaunch hector_slam_launch rover.launch

Dette vil åpne RVIZ med vår redigerte konfigurasjon.



Figur A.7 - HECTOR_SLAM kartet visualisert i RVIZ

Til høyre i vinduet vist i figur A.8 kan man lese om alle visualiseringene og litt generell info om disse. Denne infoen inneholder blant annet hvilket topic hver enkelt illustrasjon får informasjon fra. Mest interessant i RVIZ er visualiseringen som oppdaterer seg konstant. Kartet oppdaterer seg så snart det kommer et nytt kart under topic «/map». Hvor ofte dette skjer kan endres i filen «mapping_rover.launch». Kartet inneholder i all hovedsak tre elementer.



Figur A.8 - Forklaring av RVITZ kart

Som vist i figur A.9 er de svarte områdene steder algoritmen mener er okkuperte. Med dette menes steder man ikke kan bevege seg grunnet at det står et objekt der. De grå områdene er steder som er blitt scannet og som ikke inneholder objekter. Altså kan disse stedene bli brukt av vår plattform for å komme seg mellom to punkter. Den tredje delen er der man ikke har noen data, dette er illustrert ved den mørkegrå fargen i kartet. Dette kan være områder bak eksempelvis vegger eller på andre siden av objekter.

Videre illustreres veien plattformen allerede har tatt ved hjelp av den grønne veien. Veien den trenger å ta for å komme seg til målet illustreres ved den røde veien. Det siste som blir illustrert i RVIZ for brukeren er orienteringen til plattformen. Dette er illustrert ved den grønne og røde 90 graderen. Grønn stolpe representerer fremover og rød stolpe representerer høyresiden av plattformen.

Illustrasjonen av veien plattformen må ta for å komme seg til mål er det eneste illustrert i RVIZ som ikke er blitt forklart hittil. Det er nettopp dette Python programvaren vår gjør. Den gjør dette ved hjelp av pathfinding algoritmen A*.

Videre informasjon om installasjonsprosessen kan finnes på ROS wiki. Denne nettsiden er linket til i bibliografien.

A.5.3 Egen ROS pakke

Vår kildekode består av flere scripts eller filer som er skrevet i programmerings språket Python. For å kunne lage hjemmelagde ROS filer, uten å blande disse med allerede eksisterende åpen kildekode, ble det valgt å lage en ny ROS pakke. Endre først din posisjon i terminalvinduet til «catkin_ws/src» mappen. Deretter kjøres kommandoen:

catkin_create_pkg test_code std_msgs rospy roscpp

Dette vil lage en ny ROS pakke med navn «test_code» og avhengighetene «std_msgs», «rospy» og «roscpp». Når dette er gjort må man huske å lage miljøet igjen ved hjelp av «catkin_make». Det er viktig at man befinner seg i «catkin_ws» mappen vår når man lager miljøet. Nå har det blitt generert en egen ROS pakke, som for øyeblikket er uten innhold av betydning. Derfor lages en undermappe for kildekoden vår. Dette gjøres ved å kjøre kommandoen:

mkdir catkin_ws/src/test_code/scripts

Dette vil generere en undermappe kalt «scripts» i vår ROS pakke «test_codes». All kildekode som blir presentert i denne oppgaven er lagt inn i denne mappen. Nå kan man enkelt kjøre hvilket som helst av scriptene fra denne mappen, men først må rettighetene til scriptet endres slik at det kan kjøres. Dette gjøres ved å bruke kommandoen:

sudo chmod +x script.py

Dette vil endre rettighetene til eksempelfilen «script.py» slik at man kan kjøre den fra terminalvinduet. Rettighetene til alle filene med kildekode som trenger å kjøre direkte fra terminalvinduet vil bli endret på denne måten.

A.6 Raspberry Pi GPIO tilkoblinger

GPIO	Funksjon
2	Power touchscreen
4	Power Raspberry Pi
6	GND Raspberry Pi
14	GND touchscreen

Tabell A.3 - Viser GPIO punktene som er i bruk.

A.7 Budsjett

Kostnadsoversikt for Bachelor					
Artikkel	Antall	Stykkpris (kr)		Total kostnad (kr)	
		Beregnet	Faktisk	Beregnet	Faktisk
Fremdrift					
Odrive v3.6 motorkontroller	4	1500	1380	6000	5520
Styring					
Navio2 Autopilot Kit for Raspberry Pi 2/3	1	2500	1849	2500	1849
RPLidar A1M8	1	3000	859	3000	859
Raspberry Pi 4 Model B	1	800	729	800	729
Totalsum				12300	8957

Tabell A.4 – Budsjettoversikt

B Kildekode

All kildekode for ROS delen av prosjektet kan hentes fra github (Hassel, 2019)

B.1 Action.py

```
1.  #! /usr/bin/env python
2.
3.  #script containing the decision making prosess
4.
5.  #importing the necessary libraries
6.  import math
7.
8.  #importing the scripts sendMessage.py and directionDatabase.py
9.  import directionDatabase as ddModule
10. import sendMessage as sendMessageModule
11.
12. #importing the string object from std_msgs.msg
13. from std_msgs.msg import String
14.
15. #function used to find the angle between two directions in radians
16. def getAngles(current, needed):
17.     pi = math.pi
18.
19.     #calculate from quaternions to radians
20.     curr = math.acos(current) * 2
21.     need = math.acos(needed) * 2
22.
23.     #get the difference
24.     temp = curr - need
25.
26.     #make sure the difference is always the shortest way (always less than
27.      pi radians)
28.     if -pi < temp < pi:
29.         diff = temp
30.     else:
31.         if temp > 0:
32.             diff = temp - (2 * pi)
33.         if temp < 0:
34.             diff = temp + (2 * pi)
35.     return diff, need
36.
37. #function used to determine what action to take
38. def determine_action(path, direction, myDirection, goal, pos):
39.
40.     #set IP and PORT for issuing commands
41.     IP = '192.168.84.254'
42.     PORT = 2088
43.
44.     #if we are not currently at our goal
45.     if goal != pos:
46.
47.         #if there is a path
48.         if path != 0:
49.
50.             #genereate a direction database object
51.             move_dir = ddModule.check_direction()
52.             next_move = move_dir.getDir(direction)
53.
54.             #check what direction is needed
55.             turn_needed, needed_angle = getAngles(myDirection, next_move)
56.
```

```
57.             #if it is within the accuraccy constant
58.             if -0.2 < turn_needed < 0.2:
59.
60.                 #send instructions to drive forward
61.                 msg = 'D'
62.                 sendMessageModule.sendinstructions(IP, PORT, msg)
63.                 print('driving')
64.
65.             #if it isn't make it turn
66.             else:
67.
68.                 #for negative turn amounts we turn right
69.                 if turn_needed < 0:
70.
71.                     #make the value needed a positive number
72.                     turn_abs = abs(turn_needed)
73.                     msg = 'R' + str(turn_abs)
74.
75.                     #send the instructions to turn right
76.                     sendMessageModule.sendinstructions(IP, PORT, msg)
77.                     print(msg)
78.
79.                 #for positive turn amounts we turn left
80.                 else:
81.                     msg = 'L' + str(turn_needed)
82.
83.                     #send the instructions to rutn left
84.                     sendMessageModule.sendinstructions(IP, PORT, msg)
85.                     print(msg)
86.             return needed_angle
87.
88.         #there is no path, wait for new orders or available path
89.         else:
90.
91.             #send instructions to stop to avoid collision
92.             sendMessageModule.sendinstructions(IP, PORT, 'S')
93.             print('Searching')
94.             return 0.0
95.
96.
97.     #if we are at the goal
98.     else:
99.
100.            #send instrutions to stop, we have arrived at the goal
101.            sendMessageModule.sendinstructions(IP, PORT, 'S')
102.            print('arrived')
103.            return 0.0
```

B.2 aStar.py

```
1.  #! /usr/bin/env python
2.
3.  #script containing the implementation of A*
4.
5.  #importing the necessary libraries
6.  import math
7.
8.  #importing the scripts needed
9.  import myMap as mapModule
10. import queue as queueModule
11. import directionDatabase as ddModule
12.
13. #function used to make a path from start to finish
14. def makePath(cameFrom, start, finish):
```

```
15.
16.     #set the finish and setup path list
17.     current = finish
18.     path = []
19.
20.     #while we are not at the start
21.     while current != start:
22.
23.         #put the curent node in the path
24.         path.append(current)
25.
26.         #retrive its parent and set it as the current node
27.         current = cameFrom[current]
28.
29.     #place the start node in the path
30.     path.append(start)
31.
32.     #reverses it so that it can be used from the rovers perspective
33.     path.reverse()
34.     return path
35.
36. #function used to set the starting nodes parents
37. def setCameFromStart(dirr, start, theMap):
38.
39.     #generate direction database object and calculate the rovers current di-
    rection
40.     move_dir = ddModule.check_direction()
41.     d = math.acos(dirr) * 2
42.     step = math.pi / 8
43.     currentPlus = d + step
44.     currentMinus = d - step
45.     temp = 0
46.
47.     #find the major direction that corresponds to the rovers back
48.     #set said direction as its parent
49.     if currentMinus < math.acos(move_dir.getDir('N')) * 2 < currentPlus:
50.         temp = start + theMap.get_width()
51.         print('Facing: N')
52.     if currentMinus < math.acos(move_dir.getDir('NE')) * 2 < currentPlus:
53.         temp = start + (theMap.get_width() - 1)
54.         print('Facing: NE')
55.     if currentMinus < math.acos(move_dir.getDir('E')) * 2 < currentPlus:
56.         temp = start - 1
57.         print('Facing: E')
58.     if currentMinus < math.acos(move_dir.getDir('SE')) * 2 < currentPlus:
59.         temp = start - (theMap.get_width() + 1)
60.         print('Facing: SE')
61.     if (currentMinus < math.acos(move_dir.getDir('S')) * 2 < current-
    Plus) or (0 < math.acos(move_dir.getDir('S')) * 2 < step):
62.         temp = start - theMap.get_width()
63.         print('Facing: S')
64.     if currentMinus < math.acos(move_dir.getDir('SW')) * 2 < currentPlus:
65.         temp = start - (theMap.get_width() - 1)
66.         print('Facing: SW')
67.     if currentMinus < math.acos(move_dir.getDir('W')) * 2 < currentPlus:
68.         temp = start + 1
69.         print('Facing: W')
70.     if currentMinus < math.acos(move_dir.getDir('NW')) * 2 < currentPlus:
71.         temp = start + (theMap.get_width() + 1)
72.         print('Facing: NW')
73.     return temp
74.
75. #function used to solve the actual pathfinding
76. def aStarSolver(theMap, start, finish, currentDir):
77.
78.     #set the goal and add the start to the open list
79.     posFinish = theMap.node_to_gridpos(finish)
```

```
80.     openList.put(start, 0)
81.
82.     #generate lists for parents and costs
83.     cameFrom = {}
84.     currentCost = {}
85.
86.     #set parents to none and cost to inf for the beginning
87.     for i in range(0, int(theMap.get_number_of_nodes() - 1)):
88.         cameFrom[i] = None
89.         currentCost[i] = float("inf")
90.
91.     #set the cost of starting to 0
92.     currentCost[start] = 0
93.
94.     #get the starts parent using setCameFromStart fucntion
95.     cameFrom[start] = setCameFromStart(currentDir, start, theMap)
96.
97.     pathfound = False
98.
99.     #while there are objects in the open list
100.        while not openList.empty():
101.
102.            #get the object with the lowest cost
103.            current = openList.get()
104.
105.            #check if it is the goal
106.            if current == finish:
107.                print("Found path")
108.
109.                #break the loop if it is, the goal has been reached
110.                pathfound = True
111.                break
112.
113.            #check all the current nodes neighbours for their cost and availa-
    bility
114.            for next in theMap.nextTo(current):
115.
116.                #if the node is to the currents NW, NE, SW or SE
117.                if (next == current + theMap.get_width() + 1) or (next == cur-
    rent + theMap.get_width() - 1) or (next == cur-
    rent - theMap.get_width() + 1) or (next == cur-
    rent - theMap.get_width() - 1 ):
118.
119.                    #if node infront of rover
120.                    if (current - cameFrom[current]) == (next - current):
121.
122.                        #set the cost to current + 2
123.                        newCost = currentCost[current] + 2
124.
125.                    #if it is not infront
126.                    else:
127.
128.                        #set the cost to current + 3
129.                        newCost = currentCost[current] + 3
130.
131.                #if the node is to the currents N, S, E og W
132.                else:
133.
134.                    #if the node is infront of the rover
135.                    if (current - cameFrom[current]) == (next - current):
136.
137.                        #set the cost to current  + 1
138.                        newCost = currentCost[current] + 1
139.
140.                    #if it is not infront
141.                    else:
142.
```

```
143.                         #set the cost to current + 3
144.                         newCost = currentCost[current] + 3
145.
146.                 #if the next node is not the currentCost list of the new-
    Cost is smaller
147.                 #than the pervious cost
148.                 if next not in currentCost or newCost < currentCost[next]:
149.
150.                     #we ser the new cost of the node
151.                     currentCost[next] = newCost
152.
153.                     #calculates its priority
154.                     priority = newCost + posFinish.dist(theMap.node_to_grid-
    pos(current))
155.
156.                     #places it in the openList
157.                     openList.put(next, priority)
158.
159.                     #selects next node
160.                     cameFrom[next] = current
161.
162.         #if the path is found
163.         if pathfound == True:
164.
165.             #we make the path using the makePath() function
166.             path = makePath(cameFrom, start, finish)
167.             path_drawn = []
168.
169.             #we draw the path in node form
170.             for next in path:
171.                 temp = theMap.node_to_pos(next)
172.                 path_drawn.append(temp)
173.
174.             #finally we return the path
175.             return path, path_drawn
176.
177.         #there is no path to the goal
178.         else:
179.
180.             #we return the none-values for the path
181.             return 0, 0
```

B.3 directionDatabase.py

```
1.  #! /usr/bin/env python
2.
3.  #script containing a database of the major directions used
4.
5.  #import the math librarie
6.  import math
7.
8.  #make the calss for checking directions
9.  class check_direction:
10.
11.     #initialize the values
12.     def __init__(self):
13.         self.N = 0.0
14.         self.S = 1.0 #could be -1 or 1 (0 deg = 1, 360 deg = -1)
15.         self.E = -math.sqrt(2)/2
16.         self.W = math.sqrt(2)/2
17.         self.NE = -math.sqrt(2 - math.sqrt(2))/2
```

```
18.         self.NW = math.sqrt(2 - math.sqrt(2))/2
19.         self.SE = -math.sqrt(math.sqrt(2) + 2)/2
20.         self.SW = math.sqrt(math.sqrt(2) + 2)/2
21.
22.
23.     #function for returning the value corresponding with a major direction
24.     def getDir(self, dir):
25.         if dir == 'N':
26.             return self.N
27.         if dir == 'S':
28.             return self.S
29.         if dir == 'E':
30.             return self.E
31.         if dir == 'W':
32.             return self.W
33.         if dir == 'NE':
34.             return self.NE
35.         if dir == 'NW':
36.             return self.NW
37.         if dir == 'SE':
38.             return self.SE
39.         if dir == 'SW':
40.             return self.SW
41.         else:
```

B.4 drawLine.py

```
1.  #! /usr/bin/env python
2.
3.  #script used to setup and publish the marker used to visualize
4.  #the path in RVIZ
5.
6.  #importing rospy and position.py
7.  import rospy
8.  import position as posModule
9.
10. #importing message types
11. from visualization_msgs.msg import Marker
12. from geometry_msgs.msg import Point
13.
14. #function used to draw the line
15. def line_draw(nodes, mypos):
16.
17.     #setup the publisher
18.     pub_path = rospy.Publisher('visualization_marker', Marker, queue_size=1)
19.
20.     #setup the marker
21.     marker = Marker()
22.     marker.header.frame_id = "/map"
23.     marker.type = marker.LINE_STRIP
24.     marker.action = marker.ADD
25.
26.     #set the markers scale
27.     marker.scale.x = 0.05
28.     marker.scale.y = 0.05
29.     marker.scale.z = 0.05
30.
31.     #set the markers color
32.     marker.color.a = 1.0
33.     marker.color.r = 1.0
34.     marker.color.g = 0.0
```

```
35.     marker.color.b = 0.0
36.
37.     #set the markers orientation
38.     marker.pose.orientation.x = 0.0
39.     marker.pose.orientation.y = 0.0
40.     marker.pose.orientation.z = 0.0
41.     marker.pose.orientation.w = 1.0
42.
43.     #set the markers start position
44.     marker.pose.position.x = 0.0
45.     marker.pose.position.z = 0.0
46.
47.     #generate a list for the points within the marker
48.     marker.points = []
49.
50.     #add points to the list
51.     if nodes != 0:
52.         for next in nodes:
53.             temp = Point()
54.             temp.x = next.getX()
55.             temp.y = next.getY()
56.             temp.z = 0.0
57.             marker.points.append(temp)
58.
59.     #publish the marker to specified topic
60.     pub_path.publish(marker)
```

B.5 main.py

```
1.  #! /usr/bin/env python
2.
3.  #script used for running the brain for our autonomous system
4.
5.  #importing the necessary libraries
6.  import rospy
7.  import message_filters
8.  import time
9.  import math
10.
11. #importing necessary message types
12. from nav_msgs.msg import OccupancyGrid
13. from geometry_msgs.msg import PoseStamped
14.
15. #importing neccesary scripts
16. import position as posModule
17. import myMap as mapModule
18. import aStar as aStarModule
19. import drawLine as drawLineModule
20. import action as actionModule
21. import sendMessage as sendMessageModule
22.
23. #callback function used for running the enitre process
24. def callback(map_data, pos_data):
25.
26.     #ser the start time
27.     start_time = time.time()
28.
29.     #place map and position data into classes created in myMap.py and posi-
    tion.py
```

```
30.     theMap = mapModule.my-
    Map(map_data.data, map_data.info.height, map_data.info.width, map_data.info.re
    solution)
31.     myPos = posModule.position(pos_data.pose.position.x, pos_data.pose.posi-
    tion.y)
32.
33.     #calculate the node of the current position
34.     myPosNode = theMap.pos_to_node(myPos)
35.
36.     #send current position via UDP to specified IP, PORT
37.     sendCurrentPos = str(myPos.getX()) + ',' + str(myPos.getY())
38.     sendMessageModule.sendinstructions('192.168.84.254', 2089, sendCurrent-
    Pos)
39.
40.     #set the goal of the operation in meters and in node
41.     myGoal = posModule.position(-0.25, 0.5)
42.     myGoalNode = theMap.pos_to_node(myGoal)
43.
44.     #get current direction
45.     myDirection = pos_data.pose.orientation.z
46.
47.     #get the path from the aStarSolver from aStar.py
48.     path, path_drawn = aStarModule.aStarSolver(theMap, myPosNode, myGoal-
    Node, myDirection)
49.
50.     #draw the RVIZ line from drawLine.py
51.     drawLineModule.line_draw(path_drawn, myPos)
52.
53.     #check if there is a path to the goal and set the direction of next move
54.     if (path != 0) and (myPosNode != myGoalNode):
55.         direction = theMap.next_move(path[0], path[1])
56.     else:
57.         direction = False
58.
59.     #determine the action needed to get to the next position and send the in-
    structions
60.     g = actionModule.determine_action(path, direction, myDirection, myGoal-
    Node, myPosNode)
61.
62.     #create publishers for later analysis using rosbags
63.     #publishes current and wanted directions to two different topics
64.     pub = rospy.Publisher('goal_out_pose', PoseStamped, queue_size=3)
65.     goalPose = PoseStamped()
66.     goalPose.pose.orientation.z = g
67.     pub.publish(goalPose)
68.     pub2 = pub = rospy.Publisher('current_out_pose', PoseStam-
    ped, queue_size=3)
69.     currentPose = PoseStamped()
70.     c = pos_data.pose.orientation.z
71.     currentPose.pose.orientation.z = math.acos(c) * 2
72.     pub2.publish(currentPose)
73.
74.     #calculates and prints the elapsed time in order to moni-
    tor the time usage of the prosess
75.     print('elapsed time: ' + str((time.time() - start_time)*1000) + 'ms\n')
76.
77. #function used to register messages and call the callback function
78. def registerMessages():
79.
80.     #initialize ROS node
81.     rospy.init_node('registerMessages', anonymous=True)
82.
83.     #create message filter subscribers for specified topics
84.     map_sub = message_filters.Subscriber('map', OccupancyGrid)
85.     pos_sub = message_filters.Subscriber('slam_out_pose', PoseStamped)
86.
87.     #waits untill both subscribers recieve messages within the slop time
```

```
88.     ts = message_filters.ApproximateTimeSynchroni-
    zer([map_sub, pos_sub], queue_size = 3, slop = 0.1)
89.
90.     #calls the callback function
91.     ts.registerCallback(callback)
92.
93.     #spins the runction, creating a loop
94.     rospy.spin()
95.
96. #runs the script
97. if __name__ == '__main__':
98.
99.     #calls function to registerMessages()
100.        registerMessages()
```

B.6 myMap.py

```
1.  #! /usr/bin/env python
2.
3.  #script containing the class that handles the map generated from hector_slam
4.
5.  #import the position.py script
6.  import position as posModule
7.
8.  #create class
9.  class myMap:
10.
11.     #initialization
12.     def __init__(self, theMap, height, width, res):
13.         self.mapData = theMap
14.         self.height = height
15.         self.width = width
16.         self.resolution = res
17.
18.     #fucntion for returning the map data
19.     def get_mapData(self):
20.         return self.mapData
21.
22.     #function setting a specified node value
23.     def set_mapData(self, node, val):
24.         self.mapData[node] = val
25.
26.     #function returning the map height
27.     def get_height(self):
28.         return self.height
29.
30.     #function returning the map width
31.     def get_width(self):
32.         return self.width
33.
34.     #function returning the map resolution
35.     def get_resolution(self):
36.         return self.resolution
37.
38.     #function returning the number of nodes in the map
39.     def get_number_of_nodes(self):
40.         return (self.height * self.width)
41.
42.     #function returning the map size in square meters
43.     def get_map_size(self):
```

```
44.         return ((self.height * self.resolution) * (self.width * self.resolu-
    tion))
45.
46.     #function that translates a specified node into its posi-
    tion in the map (meters)
47.     def node_to_pos(self, node):
48.         x_pos = ((((node % self.width)) - (self.width / 2)) * (self.resolu-
    tion) + (self.resolution / 2))
49.         y_pos = (((((node - (node % self.width)) / self.height) - (self.height
     / 2)) * self.resolution) + (self.resolution / 2))
50.         return posModule.position(x_pos, y_pos)
51.
52.     #function that translates a specified node into its gridposi-
    tion in the map (nodes)
53.     def node_to_gridpos(self, node):
54.         x_pos = (node % self.width)
55.         y_pos = ((node - x_pos) / self.height)
56.         return posModule.position(x_pos, y_pos)
57.
58.     #function that translates a given position (meters) to its correspon-
    ding node
59.     def pos_to_node(self, pos):
60.         x = (int(((pos.getX() - (self.resolution / 2)) / self.resolu-
    tion)) + int((self.width / 2)))
61.         y = (int((((pos.getY() - (self.resolution / 2)) / self.resolu-
    tion) + (self.height / 2))) * int(self.height))
62.         return (x + y)
63.
64.     #function that translates a given position (meters) to its correspon-
    ding position (nodes)
65.     def pos_to_gridpos(self, pos):
66.         return self.node_to_gridpos(self.pos_to_node(pos))
67.
68.     #funcrtion that checksif a node is withing the boundaries of the map
69.     def inMap(self, node):
70.         temp = self.node_to_gridpos(node)
71.         x = temp.getX()
72.         y = temp.getY()
73.         if x != 0 and x != self.width - 1:
74.             return ((0 <= x < self.width) and (0 <= y < self.height))
75.         else:
76.             return False
77.
78.     #function that checks if a node is occupied
79.     def noWall(self, node):
80.         return self.mapData[node] != 100
81.
82.     #function that checks if all nodes surrounding a speci-
    fied node are not occupied
83.     def space(self,node):
84.         if node >= (self.width + 1):
85.             a1 = self.mapData[node - (self.width + 1)] != 100
86.         else:
87.             a1 = False
88.         if node >= (self.width):
89.             a2 = self.mapData[node - (self.width)] != 100
90.         else:
91.             a2 = False
92.         if node >= (self.width - 1):
93.             a3 = self.mapData[node - (self.width -1)] != 100
94.         else:
95.             a3 = False
96.         if node >= 1:
97.             a4 = self.mapData[node - 1] != 100
98.         else:
99.             a4 = False
100.            if node < ((self.height * self.width) - 1):
```

```
101.                a5 = self.mapData[node + 1] != 100
102.            else:
103.                a5 = False
104.            if node < ((self.height * self.width) - (self.width - 1)):
105.                a6 = self.mapData[node + (self.width - 1)] != 100
106.            else:
107.                a6 = False
108.            if node < ((self.height * self.width) - (self.width)):
109.                a7 = self.mapData[node + (self.width)] != 100
110.            else:
111.                a7 = False
112.            if node < ((self.height * self.width) - (self.width + 1)):
113.                a8 = self.mapData[node + (self.width + 1)] != 100
114.            else:
115.                a8 = False
116.            if (a1 == True) and (a2 == True) and (a3 == True) and (a4 == True)
    and
117.               (a5 == True) and (a6 == True) and (a7 == True ) and (a8 == True)
    :
118.                return True
119.            else:
120.                return False
121.
122.        #function that filters and returns the available neighbours of a node
123.        def nextTo(self, node):
124.
125.            #specifies what a neighbour is
126.            n = [node - (self.width + 1), node - (self.width), node - (self.wid
    th - 1),
127.                 node - 1, node + 1, node + (self.width - 1), node + (self.widt
    h),
128.                 node + (self.width + 1)]
129.
130.            #uses filter functions to filter out "bad" nodes
131.            n = filter(self.inMap, n)
132.            n = filter(self.noWall, n)
133.            n = filter(self.space, n)
134.            return n
135.
136.        #function defining which direction the ro-
    ver should take between two nodes
137.        def next_move(self, current_node, next_node):
138.            move = current_node - next_node
139.            if move == 1:
140.                return 'W'
141.            if move == - 1:
142.                return 'E'
143.            if move == self.width:
144.                return 'S'
145.            if move == -self.width:
146.                return 'N'
147.            if move == self.width + 1:
148.                return 'SW'
149.            if move == self.width - 1:
150.                return 'SE'
151.            if move == -(self.width + 1):
152.                return 'NE'
153.            if move == -(self.width - 1):
154.                return 'NW'
```

B.7 plotter.py

```
1.  #! /usr/bin/env python
2.
3.  #script used for plotting data from tests gathered in rosbags
4.
5.  #importing libraries used in the script
6.  import rospy
7.  import rosbag
8.  import matplotlib.pyplot as plt
9.  import numpy as np
10.
11. #function for plotting two values in one graph
12. def make_graph(filename):
13.
14.     #make a rosbag object from the file
15.     bag = rosbag.Bag(filename)
16.
17.     #generate lists for storing datapoints for value 1
18.     time1 = list()
19.     data1 = list()
20.     i = 0
21.
22.     #read the rosbag and add data from the specified topic
23.     for topic, msg, t in bag.read_messages(topics='current_out_pose'):
24.         sec = t.to_nsec()
25.         time1.append((sec-1508618888979416609)*1e-9)
26.         data1.append(msg.pose.orientation.z)
27.         i = i + 1
28.
29.     #generate lists for storing datapoints for value 2
30.     time2 = list()
31.     data2 = list()
32.     i = 0
33.
34.     #read the rosbag and add data from the specified topic
35.     for topic, msg, t in bag.read_messages(topics='goal_out_pose'):
36.         sec = t.to_nsec()
37.         time2.append((sec-1508618888979416609)*1e-9)
38.         data2.append(msg.pose.orientation.z)
39.         i = i + 1
40.
41.     #close the rosbag
42.     bag.close()
43.
44.     #create a subplot
45.     ax = plt.subplot()
46.
47.     #plot the data from data1 and data2 in the subplot
48.     ax.plot(time1, data1, label='current', linewidth=3)
49.     ax.plot(time2, data2, label='wanted', linewidth=3)
50.
51.     #create a legend
52.     legend = ax.legend(loc='center right', fontsize='x-large')
53.
54.     #specify label names and font sizes for the graph
55.     plt.xlabel('time (seconds)', fontsize=20)
56.     plt.ylabel('angle (radians)', fontsize=20)
57.     plt.title('Current vs Wanted angles', fontsize=30)
58.
59.     #set the tick parameters
```

```
60.     ax.tick_params(length=10, width=3)
61.     axi = plt.gca()
62.     for axis in ['top','bottom','left','right']:
63.         axi.spines[axis].set_linewidth(3)
64.     for tick in ax.xaxis.get_major_ticks():
65.         tick.label.set_fontsize(14)
66.         tick.label.set_rotation(1)
67.     for tick in ax.yaxis.get_major_ticks():
68.         tick.label.set_fontsize(14)
69.         tick.label.set_rotation(1)
70.
71.     #show the graph to the user
72.     plt.show()
73.
74. #function for plotting the difference between two values in a graph
75. def make_graph1(filename):
76.
77.     #make a rosbag object from the file
78.     bag = rosbag.Bag(filename)
79.
80.     #generate lists for storing datapoints for value 1
81.     time1 = list()
82.     data1 = list()
83.     i = 0
84.
85.     #read the rosbag and add data from the specified topic
86.     for topic, msg, t in bag.read_messages(topics='current_out_pose'):
87.         sec = t.to_nsec()
88.         time1.append((sec-1508618888979416609)*1e-9)
89.         data1.append(msg.pose.orientation.z)
90.         i = i + 1
91.
92.     #generate lists for storing datapoints for value 2
93.     time2 = list()
94.     data2 = list()
95.     i = 0
96.
97.     #read the rosbag and add data from the specified topic
98.     for topic, msg, t in bag.read_messages(topics='goal_out_pose'):
99.         sec = t.to_nsec()
100.            time2.append((sec-1508618888979416609)*1e-9)
101.            data2.append(msg.pose.orientation.z)
102.            i = i + 1
103.
104.        #close the rosbag
105.        bag.close()
106.
107.        #generate temporary list for storing the difference values
108.        temp = list()
109.        ii = 0
110.
111.        #calculate the difference and add it to the temp list
112.        for next in data1:
113.            temp.append(next - data2[ii])
114.            ii = ii + 1
115.
116.        #plot the data in the graph
117.        plt.plot(time1, temp, label='difference')
118.        plt.title('Angular difference (rad)', fontsize=30)
119.
120.        #show the graph to the user
121.        plt.show()
122.
123.    #runs the script
124.    if __name__ == '__main__':
125.
126.        #set the filename of the rosbag
```

```
127.        filename = '/home/daniel/rosbags/avoidance_tests/unknown.bag'
128.
129.        #plot the data
130.        make_graph(filename)
```

B.8 position.py

```
1.  #! /usr/bin/env python
2.
3.  #script containing our position class
4.
5.  #generates the class
6.  class position:
7.
8.      #initializes it
9.      def __init__(self, x, y):
10.         self.x = x
11.         self.y = y
12.
13.     #function for setting X value
14.     def setX(self, x):
15.         self.x = x
16.
17.     #function for getting X value
18.     def getX(self):
19.         return self.x
20.
21.     #function for setting Y value
22.     def setY(self, y):
23.         self.y = y
24.
25.     #function for getting Y value
26.     def getY(self):
27.         return self.y
28.
29.     #function for getting distance between two points on the map
30.     def dist(self, pos):
31.         return abs(self.x - pos.getX()) + abs(self.y - pos.getY())
```

B.9 queue.py

```
1.  #! /usr/bin/env python
2.
3.  #scripts that functions as a wrapper for the heapq library
4.
5.  #import the heapq library
6.  import heapq
7.
8.  #define a class for using it
9.  class priorityQueue:
10.
11.     #initialization
12.     def __init__(self):
13.         self.elements = []
14.
```

```
15.     #function to empty the queue
16.     def empty(self):
17.         return len(self.elements) == 0
18.
19.     #function to put an item in the queue
20.     def put(self, item, priority):
21.         heapq.heappush(self.elements, (priority, item))
22.
23.     #function to extract next item from the queue
24.     def get(self):
25.         return heapq.heappop(self.elements)[1]
```

B.10 sendMessage.py

```
1.  #! /usr/bin/env python
2.
3.  #script used to send UDP messages
4.
5.  #import the socket library
6.  import socket
7.
8.  #function used for sending messages
9.  def sendinstructions(TARGET_IP, TARGET_PORT, MESSAGE):
10.     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
11.     sock.sendto(MESSAGE,(TARGET_IP, TARGET_PORT))
```

B.11 stop.py

```
1.  #! /usr/bin/env python
2.
3.  #script used to send stop instructions to NODE-RED
4.  #Used only if main.py fails to stop the rover or if
5.  #something goes wrong
6.
7.  #imports necessary libraries and scripts
8.  import socket
9.  import sendMessage as sm
10.
11. #reuns the script at start
12. if __name__ == '__main__':
13.
14.     #specify IP, PORT and MESSAGE
15.     IP = '192.168.84.254'
16.     PORT = 2088
17.     MESSAGE = 'S0.232'
18.
19.     #uses function from sendMessage.py to send messages
20.     #to NODE-RED
21.     sm.sendinstructions(IP, PORT, MESSAGE)
```

B.12 Tastaturstyring

```
FUNCTION_BLOCK Styring_Tastatur
VAR_INPUT
END_VAR
VAR_OUTPUT
    Speed_Left : REAL;
    Speed_Right : REAL;
END_VAR
VAR
    Forward : BOOL;
    Reverse : BOOL;
    Right : BOOL;
    Left : BOOL;
    Speed : REAL;
END_VAR
```

```
//Forward and Reverse
IF Forward AND NOT Reverse THEN
    Speed_Left := Speed;
    Speed_Right := Speed;
END_IF

IF Reverse AND NOT Forward THEN
    Speed_Left := -Speed;
    Speed_Right := -Speed;
END_IF

//Left
IF Left AND NOT Right THEN
    Speed_Left := -Speed;
    Speed_Right := Speed;
END_IF

//Right
IF Right AND NOT Left THEN
    Speed_Left := Speed;
    Speed_Right := -Speed;
END_IF

IF NOT Forward AND NOT Reverse AND NOT Left AND NOT Right THEN
    Speed_Left := 0;
    Speed_Right := 0;
END_IF
```

Figur 0.1 – Tastaturstyring i PLS

B.13 Kode for måling

```
//FB for measuring DC Power, Voltage and Current in Battery 1 and 2
Battery_Pow_Measure(xEnable:=Enable, I_Port:=IoConfig_Globals._3_PHASE_POM_277VAC_DC_EXT);

Battery_Pow_Measure.eDC_Value1 := WagoAppPowerMeasurement.e494DC_Values.DcVoltageL1_N;              //inputs for Power FB
Battery_Pow_Measure.eDC_Value2 := WagoAppPowerMeasurement.e494DC_Values.DcPowerL1;
Battery_Pow_Measure.eDC_Value3 := WagoAppPowerMeasurement.e494DC_Values.DirectCurrentL1;
Battery_Pow_Measure.eDC_Value4 := WagoAppPowerMeasurement.e494DC_Values.DirectCurrentL2;

Battery_Voltage := Battery_Pow_Measure.rMeasuredValue1;                                            //outputs for Power FB
Battery1_Power := Battery_Pow_Measure.rMeasuredValue2;
Battery1_current := Battery_Pow_Measure.rMeasuredValue3;
Battery2_current := Battery_Pow_Measure.rMeasuredValue4;
Battery2_Power := Battery_Voltage * Battery2_current;

//measuring capacity of batteries

IF Battery_Voltage >= 42 THEN
        Battery_capacity := 100;
    ELSIF Battery_voltage >= 41.5 THEN
        Battery_capacity := 95;
    ELSIF Battery_voltage >= 41.0 THEN
            Battery_capacity := 90;
    ELSIF Battery_voltage >= 40.5 THEN
            Battery_capacity := 85;
    ELSIF Battery_voltage >= 40.0 THEN
            Battery_capacity := 75;
    ELSIF Battery_voltage >= 39.0 THEN
            Battery_capacity := 65;
    ELSIF Battery_voltage >= 38.0 THEN
            Battery_capacity := 40;
    ELSIF Battery_voltage >= 37.0 THEN
            Battery_capacity := 30;
    ELSIF Battery_voltage >= 36.0 THEN
            Battery_capacity := 20;
END_IF

//measuring temperature in motors and ESC's
M3_temp := Motor3_temp/10.0;
M4_temp := Motor4_temp/10.0;
MC1_temp := ESC1_temp/10.0;
MC3_temp := ESC3_temp/10.0;
```

Figur 0.2 - Kode for målinger i PLS

B.14 mapping_rover.launch

```
<?xml version="1.0"?>

<launch>
  <arg name="tf_map_scanmatch_transform_frame_name" default="scan-matcher_frame"/>
  <arg name="base_frame" default="base_link"/>
  <arg name="odom_frame" default="base_link"/>
  <arg name="pub_map_odom_transform" default="true"/>
  <arg name="scan_subscriber_queue_size" default="1"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="100"/>

  <node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen">

    <!-- Frame names -->
    <param name="map_frame" value="map" />
    <param name="base_frame" value="$(arg base_frame)" />
    <param name="odom_frame" value="$(arg odom_frame)" />

    <!-- Tf use -->
    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="false"/>
    <param name="pub_map_odom_transform" value="$(arg pub_map_odom_transform)"/>

```

```
24.     <!-- Map size / start point -->
25.     <param name="map_resolution" value="0.30"/>
26.     <param name="map_size" value="$(arg map_size)"/>
27.     <param name="map_start_x" value="0.5"/>
28.     <param name="map_start_y" value="0.5" />
29.     <param name="map_multi_res_levels" value="2" />
30.
31.     <!-- Map update parameters -->
32.     <param name="update_factor_free" value="0.4"/>
33.     <param name="update_factor_occupied" value="0.9" />
34.     <param name="map_update_distance_thresh" value="0.25"/>
35.     <param name="map_update_angle_thresh" value="0.06" />
36.     <param name="laser_z_min_value" value = "-1.0" />
37.     <param name="laser_z_max_value" value = "1.0" />
38.     <param name="map_pub_period" value = "0.5" />
39.
40.     <!-- Advertising config -->
41.     <param name="advertise_map_service" value="true"/>
42.
43.    <!-- <param name="scan_subscriber_queue_size" value="$(arg scan_subscri-
    ber_queue_size)"/>-->
44.     <param name="scan_subscriber_queue_size" value="1"/>
45.
46.     <param name="scan_topic" value="$(arg scan_topic)"/>
47.
48.     <!-- Debug parameters -->
49.     <!--
50.       <param name="output_timing" value="false"/>
51.       <param name="pub_drawings" value="true"/>
52.       <param name="pub_debug_output" value="true"/>
53.     -->
54.     <param name="tf_map_scanmatch_transform_frame_name" va-
    lue="$(arg tf_map_scanmatch_transform_frame_name)" />
55.   </node>
56.
57.   <node pkg="tf" type="static_transform_publisher" name="base_to_laser_broad-
    caster" args="0 0 0 0 0 0 base_link laser 100"/>
58. </launch>
```

B.15 rover.launch

```
1.  <?xml version="1.0"?>
2.
3.  <launch>
4.
5.    <arg name="geotiff_map_file_path" default="$(find hector_geotiff)/maps"/>
6.
7.    <param name="/use_sim_time" value="false"/>
8.
9.    <node pkg="rviz" type="rviz" name="rviz"
10.     args="-d $(find hector_slam_launch)/rviz_cfg/rviz_rover.rviz"/>
11.
12.   <include file="$(find hector_mapping)/launch/mapping_rover.launch"/>
13.
14.   <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch">
15.     <arg name="trajectory_source_frame_name" value="scanmatcher_frame"/>
16.     <arg name="map_file_path" value="$(arg geotiff_map_file_path)"/>
17.   </include>
18.
19. </launch>
```

B.16 rviz_rover.rviz

```
Panels:
  - Class: rviz/Displays
    Help Height: 138
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /Status1
        - /Map1
        - /Pose1
        - /Path_Current1
      Splitter Ratio: 0.5
    Tree Height: 1183
  - Class: rviz/Selection
    Name: Selection
  - Class: rviz/Tool Properties
    Expanded:
      - /2D Pose Estimate1
      - /2D Nav Goal1
      - /Publish Point1
    Name: Tool Properties
    Splitter Ratio: 0.5886790156364441
  - Class: rviz/Views
    Expanded:
      - /Current View1
    Name: Views
    Splitter Ratio: 0.5
  - Class: rviz/Time
    Experimental: false
    Name: Time
    SyncMode: 0
    SyncSource: ""
Preferences:
  PromptSaveOnExit: true
Toolbars:
  toolButtonStyle: 2
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz/Grid
      Color: 160; 160; 164
      Enabled: true
      Line Style:
        Line Width: 0.029999999329447746
        Value: Lines
      Name: Grid
      Normal Cell Count: 0
      Offset:
        X: 0
        Y: 0
        Z: 0
      Plane: XY
      Plane Cell Count: 10
      Reference Frame: <Fixed Frame>
```

```
      Value: true
    - Alpha: 0.699999988079071
      Class: rviz/Map
      Color Scheme: map
      Draw Behind: false
      Enabled: true
      Name: Map
      Topic: /map
      Unreliable: false
      Use Timestamp: false
      Value: true
    - Alpha: 1
      Buffer Length: 1
      Class: rviz/Path
      Color: 25; 255; 0
      Enabled: true
      Head Diameter: 0.30000001192092896
      Head Length: 0.20000000298023224
      Length: 0.30000001192092896
      Line Style: Lines
      Line Width: 0.029999999329447746
      Name: Path_Taken
      Offset:
        X: 0
        Y: 0
        Z: 0
      Pose Color: 255; 85; 255
      Pose Style: None
      Radius: 0.029999999329447746
      Shaft Diameter: 0.10000000149011612
      Shaft Length: 0.10000000149011612
      Topic: /trajectory
      Unreliable: false
      Value: true
    - Alpha: 1
      Axes Length: 1
      Axes Radius: 0.10000000149011612
      Class: rviz/Pose
      Color: 255; 25; 0
      Enabled: true
      Head Length: 0.30000001192092896
      Head Radius: 0.10000000149011612
      Name: Pose
          Shaft Length: 1
          Shaft Radius: 0.05000000074505806
          Shape: Axes
          Topic: /slam_out_pose
          Unreliable: false
          Value: true
        - Class: rviz/Marker
          Enabled: true
          Marker Topic: visualization_marker
          Name: Path_Current
          Namespaces:
            {}
          Queue Size: 1
          Value: true
    Enabled: true
    Global Options:
      Background Color: 48; 48; 48
      Default Light: true
      Fixed Frame: map
      Frame Rate: 30
    Name: root
```

```
    Tools:
      - Class: rviz/Interact
        Hide Inactive Objects: true
      - Class: rviz/MoveCamera
      - Class: rviz/Select
      - Class: rviz/FocusCamera
      - Class: rviz/Measure
      - Class: rviz/SetInitialPose
        Theta std deviation: 0.2617993950843811
        Topic: /initialpose
        X std deviation: 0.5
        Y std deviation: 0.5
      - Class: rviz/SetGoal
        Topic: /move_base_simple/goal
      - Class: rviz/PublishPoint
        Single click: true
        Topic: /clicked_point
    Value: true
    Views:
      Current:
        Class: rviz/XYOrbit
        Distance: 10.265443801879883
        Enable Stereo Rendering:
          Stereo Eye Separation: 0.05999999865889549
          Stereo Focal Distance: 1
          Swap Stereo Eyes: false
          Value: false
        Focal Point:
          X: 2.2582809925079346
          Y: 0.03408151865005493
          Z: 0
        Focal Shape Fixed Size: true
        Focal Shape Size: 0.05000000074505806
        Invert Z Axis: false
        Name: Current View
        Near Clip Distance: 0.009999999776482582
        Pitch: 1.5697963237762451
        Target Frame: <Fixed Frame>
        Value: XYOrbit (rviz)
        Yaw: 6.278593063354492
      Saved: ~
  Window Geometry:
    Displays:
      collapsed: false
    Height: 1689
    Hide Left Dock: false
    Hide Right Dock: false
    QMainWin-
dow State: 000000ff00000000fd0000000400000000000002790000059 7fc0200000008fb000
00012005300650 06c006500630074006900 6f006e00000001e10000009b000000b000fffffffb0
000001e0054006f006f006c002000500072006f007000650072007400690065007302000001ed0
00001df00000185000000a3fb0000001200560069006500770073002000540 06f006f02000001d
f000002110000018500000122fb0000002000540 06f006f006c002000500072006f00700065007
2007400690065007300320300000288000001 1d000002210000017afb0000001000440 06900730
070006c006100790073010000006e000005970000018200ffffffb0000002000730065006c006
5006300740069006f006e002000620075006600660065007202000001380000 00aa0000023a000
00294fb00000014005700690064006500530074006500720065006f02000000e6000000d200000
3ee0000030bfb0000000c004b0069006e006500630074002000001860000010600 00030c0000026
1000000010000015f00000597fc0200000003fb0000001e0054006f006f006c002000500072006
f007000650072007400690065007301000000410000007800000000000000000fb0000000a00560
069006500770073010000006e000005970000013200ffffffb0000001200530065006c0065006
300740069006f006e010000025a000000b2000000000000000000000000200000490000000a9fc0
100000001fb0000000a00560069006500770073030000004e00000080000002e10000019700000
00300000bfa0000005afc0100000002fb0000000800540069006d006501000000000000bfa000
```

```
    0057100fffffffb0000000800540069006d00650100000000000004500000000000000000000000
    80a000005970000000400000004000000080000000 8fc00000010000000200000001 0000000a0
    054006f006f006c00730100000000ffffffff0000000000000000
169.      Selection:
170.        collapsed: false
171.      Time:
172.        collapsed: false
173.      Tool Properties:
174.        collapsed: false
175.      Views:
176.        collapsed: false
177.      Width: 3066
178.      X: 134
179.      Y: 55
```