Sjøkrigsskolen

Bacheloroppgave

GPS-fri posisjonering ved bruk av kartkorrelasjon

– Metoder for å beregne posisjon uten bruk av GPS –

av

Robin Moen og Bjørn Olav Rebni

Levert som en del av kravet til graden:

BACHELOR I MILITÆRE STUDIER MED FORDYPNING I LEDELSE OG MARINEINGENIØR VÅPEN, ELEKTRONIKK OG DATA

Antall ord: 13.666

Innlevert: des 2023

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.

Jeg (Vi) gir herved Sjøkrigsskolen rett til å gjøre denne oppgaven tilgjengelig elektronisk, gratis og uten kostnader	☒ Ja	☐ Nei
Finnes det en avtale om forsinket eller kun intern publisering? (Utfyllende opplysninger må fylles ut)	☒ Ja	☐ 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– 2023

Robin Moen
Kadett navn

Robin Moen

Bjørn Olav Rebni
Kadett navn

Bjørn Olav Rebni

Forord

Denne oppgaven har blitt skrevet av Robin Moen, og Bjørn Olav Rebni ved FHS Sjøkrigsskolen i perioden fra september til desember 2023.

Arbeidet med dette prosjektet har bydd mange utfordringer. Å benytte en plattform med tilhørendetilhørende programvare fra 2019 har ført til mange timer med feilsøking og frustrasjon. Til tross for utfordringene har vi gjennom dette prosjektet fått en dypere forståelse for kartleggingsalgoritmen SLAM, ulike bildeanalysealgoritmer og ikke minst konseptet kartkorrelasjon. Prosjektet har undersøkt kartkorrelasjons som metode for GPS-fri posisjonering. Det er kun blitt sett på to metoder for kartkorrelasjon som har blitt testet i kontrollerte rammer. Mulighetene for videre utvikling av dette konseptet er store. Det blir spennende å se hvilke metoder og sensorer som blir brukt ved eventuell videre forskning på dette konseptet i fremtiden.

En stor takk rettes til hovedveileder og førsteamanuensis Alexander Sauter og førsteamanuensis Christophe Massacand for god veiledning gjennom prosjektperioden. I tillegg er det ønskelig å takke Marcus Omholt Hovelsen for hjelp med 3D-printing av komponenter brukt i prosjektet.

Bergen, Sjøkrigsskolen, 03-12-2023

(Signatur)
Robin Moen

(Signatur)
Bjørn Olav Rebni

Sammendrag

GPS og andre satellittnavigasjonssystemer er sårbare for sabotasje, i form av blant annet jamming og andre forstyrrelser. Autonome systemer, som mange andre systemer, er avhengige av GPS for posisjonering. Når det det er lett å forstyrre GPS, er det viktig for autonome systemer å ha redundante posisjoneringsverktøy. Hensikten med denne oppgaven er å undersøke metoder for å estimere global posisjon ved hjelp av å sammenligne sensorgenererte kart av miljøet rundt det autonome systemet, opp mot kart over et større område. Denne metoden for å finne posisjon blir kalt kartkorrelasjon, da det er i praksis to kart av forskjellig kvalitet som skal plasseres i forhold til hverandre. For at kartkorrelasjonsmetoder som dette skal være pålitelig som redundans for GPS, har det blitt definert enkelte mål som omhandler estimeringens presisjon og allsidighet. For å undersøke metoder for å estimere global posisjon har en landbasert testplattform med en lidarsensor blitt benyttet. Videre har det blitt implementert to forskjellige bildeanalysealgoritmer som sammenligner kart. Det ble gjennomført flere tester for å finne ut hvorvidt metodene oppnår målsetningene som ble satt. Forskjellige områder ble kartlagt, og sammenlignet med flere større kart med en ulik mengde detaljer. Resultatene fra testene er varierende. Ved enkelte forsøk presterer metodene bra, og gir et akseptabelt avvik på mindre enn to meter mellom estimert og faktisk posisjon. Likevel er det flere resultater som fører til et stort avvik. Totalt sett viser testresultatene ingen klar sammenheng mellom presisjon til metodene, og hvilke kart som ble sammenlignet. Dette kan skyldes både kvaliteten til kartleggingen, men også hvordan metodene blir brukt. Metodene for kartkorrelasjon kan likevel vise et potensiale til å fungere da det blir sammenlignet med et annet sensorgenerert kart. Mulighetene til posisjonering med ved bruk av kartkorrelasjon må undersøkes videre for å kunne bli et anvendelig konsept i fremtiden.

Innholdsfortegnelse

Figurer

Tabeller

Nomenklatur

BRIEF – Binary Robust Independent Elementary Features

FAST – Features from Accelerated Segment Test

ORB – Oriented FAST and Rotated BRIEF

SLAM – Simultaneous Localization and Mapping

GNSS - Global Navigation Satellite System

GPS – Global Positioning System

LIDAR – Light Detection and Ranging

PGO – Pose Graph Optimization

RADAR –Radio Detection and Ranging

ROS – Robot Operating System

RVIZ – Visualiseringsverktøy

Template – En del eller et utsnitt av et bilde

1 Innledning

I innledningen blir bakgrunnen for prosjektet, konkrete mål og viktige begrensinger presentert.

1.1 Bakgrunn

Autonomi betyr selvstendig eller selvstyre (Det norske akademis ordbok, u.d.). Autonome systemer er et fagområde med stort fokus og utvikling. Et viktig element for autonome systemer er robust navigasjon. For å oppnå den nødvendige robustheten er det viktig å ha redundans i styrings- og sensorsystem (Hareide, et al., 2018, ss. 129-130). En betingelse for å kunne navigere er evnen til å bestemme egen posisjon i forhold til verden, altså en global posisjon. Dette er også viktig for effektiv samhandling med andre enheter. I tillegg kan endringer i det lokale terrenget skje fort, noe som fordrer at man også evner å orientere og posisjonere seg relativt til omgivelsene rundt, en lokal posisjon. Bestemmelse av lokal og global posisjon kan oppnås med en kombinasjon av GPS og sensorer som kan kartlegge omgivelsene.

Problemet er at GPS og andre satellittnavigasjonssystemer er sårbare for sabotasje, i form av jamming og andre forstyrrelser. De siste årene har forstyrrende signaler i GPS-båndet vært økende. I Øst-Finnmark ble antall dager med bortfall av GPS-signaler seksdoblet fra 20 dager i 2020 til 120 dager i 2022 (Hykkerud, 2023). Det er derfor viktig å se på andre metoder for å finne global posisjon og med det forebygge avhengighet til satellittnavigasjonssystemer.

I Sjøforsvaret arbeides det med å implementere flere systemer med autonome egenskaper som blant annet skal overta for de nåværende minemottiltakskapabilitetene (Forsvarsdepartementet, 2023). Dette innebærer en overgang der minerydderoperasjoner skal utføres av ubemannende farkoster, med støtte fra bemannende fartøyer utenfor minefarlig område. Forsvarets Forskningsinstitutt (FFI) har pågående forskningsprosjekter på autonome systemer, og har sett på hvordan man skal beregne posisjon og retning dersom man ikke har GPS. De har utviklet GPS-uavhengige teknikker for navigasjon, som har resultert i HUGIN; en undervannsdrone med høy grad av autonomi (FFI, u.d.).

Behovet for autonome enheter på havet er tydelig, særlig med hensyn til minemottiltak, men utvikling og testing av autonome plattformer på havet kan være vanskelig, særlig av praktiske hensyn. Til forskjell fra land, bringer sjøen utfordringer som bølger, strømninger og skiftende vær. Faktorer som bølger og strømninger trengs derimot ikke å kontrolleres på land. På Sjøkrigsskolen har det derfor gjennom to bachelorprosjekt blitt utviklet en landbasert testplattform for autonome systemer. Det første prosjektet hadde som mål å utvikle en plattform som kan brukes til demonstrasjon og undervisning av konsepter rundt autonomi. Resultatet var en rover med fire hjul hvor nye systemer og teknologi relativt enkelt kan integreres for videre forskning (Flaaten & Tande, 2018). Bacheloroppgaven fra 2019 videreutviklet den allerede etablerte plattformen, med fokus på autonom navigering uten bruk av GPS (Hassel, Moen, & Ohma, 2019). Det ble implementert en optisk sensor av typen RPLiDAR A1. Datastrømmen fra denne sensoren ble videre brukt til å generere et lokalt kart ved hjelp av Simultaneous Localization And Mapping (SLAM), som er en metode for å lokalisere en enhet i et kart generert ved hjelp av sine sensorer, og derav finne enhetens lokale posisjon. Dette kartet ble brukt sammen med en pathfinding-algoritme for å gjøre plattformen i stand til å kjøre fra en startposisjon til en sluttposisjon i trange miljøer. Plattformen er altså i stand til å generere et lokalt kart og predikere lokal posisjon basert på sine omgivelser, men evner ikke å sette dette opp mot sin globale posisjon. Med dette menes at plattformen ikke vet hvor den befinner seg i forhold til områder som ikke er innenfor ens sensorrekkevidde, og den har dermed heller ikke noe forhold til kartene en eventuell bruker forholder seg til.

I 2022 skrev Algrøy og Opdahl en oppgave som omhandlet GPS uavhengig posisjonering ved hjelp av passive, optiske sensorer. Bakgrunnen for denne oppgaven var å utforske hvorvidt et system kunne bruke sensorer for å estimere sin egen posisjon, og dermed unngå bruk av GPS som kan bli utsatt for jamming eller andre former for sabotasje (Algrøy & Opdahl, 2022). Her ble det også brukt SLAM som metode for å beregne lokal posisjon. Det ble heller ikke her sett på metoder for å estimere global posisjon.

Robust navigasjon innebefatter at enheten må være i stand til å bestemme egen posisjon, både globalt og lokalt. Det er videre interessant å se på metoder for posisjonering ved bortfall av GPS. På Sjøkrigsskolen har to tidligere oppgaver tatt for seg henholdsvis autonom navigering og lokal posisjonering, begge uten bruk av GPS. Fellesnevneren for disse prosjektene er at de ikke har utforsket mulighetene for å estimere global posisjon uten satellittnavigasjonssystemer. Hvordan kan man bruke informasjon om lokal posisjon og det lokale kartet til å beregne sin globale posisjon? En mulig løsning kan være å sammenligne det lokale kartet med globale kart for å estimere posisjon.

1.2 Mål

Det er viktig for autonome systemer å kunne bestemme egen posisjon uten bruk av GPS, både lokalt og globalt. Hovedmålet med denne oppgaven er å undersøke to metoder for å sammenligne lokale kart fra sensorer med globale kart, og videre bruke dette til å estimere global posisjon.

Metoder for å sammenligne kart, vil videre bli kalt for kartkorrelasjonsmetoder. I tillegg vil utgangspunktet for oppgaven være den landbaserte testplattformen fra 2018/2019 med den tilhørende sensoren RPLidar A1 (Hassel, Moen, & Ohma, 2019). De globale kartene som benyttes i denne oppgaven representerer samme område. Det er totalt fire globale kart, der et er i likt format og detaljnivå som de lokale kartene. De tre resterende kartene i ulikt format og har varierende grad av detaljnivå.

I henhold til den offisielle amerikanske nettsiden for GPS, har mobiltelefoner med innebygd GPS en nøyaktighet på ≤ 4.9 meter i åpne områder (National Coordination Office for Space-Based Positioning, Navigation, and Timing, 2022). Denne oppgaven tester metodene på et mindre område, noe som setter et høyere krav til presis posisjonering. Kravet til metodene for avvik blir dermed satt $\text{til} \leq 2 \text{ meter}$.

For å kunne betrakte en metode som funksjonell, settes det dermed følgende krav:

- Metodene må gi stabile resultater med avvik ≤ 2 meter, for å kunne være et pålitelig alternativ for GPS og tilføye systemet redundans.
- Metodene må kunne håndtere varierende grad av detaljer på de globale kartene.

1.3 Avgrensninger

Oppgaven avgrenses til at enheten som skal benytte seg av kartkorrelasjon til å estimere posisjon, har nylig mistet sitt GPS-signal. Den vil dermed vite hvilket område den befinner seg i. Derfor trenger enheten kun å sammenligne det lokale kartet med det aktuelle området, og har ikke behov for å sammenligne med alle kart i verden.

Oppgaven skal kun se på metoder for å estimere posisjon, og vil dermed ikke videreutvikle andre konkrete aspekter med den autonome navigeringen. I tillegg skal det kun benyttes sensorer som plattformen allerede besitter

1.4 Struktur

Denne oppgaven består videre av fem kapitler. Det første kapittelet tar for seg relevant teori som har blitt benyttet til forståelse og anvendelse i dette prosjektet. Første del tar for seg sensoren og relevant programvare, deretter blir teorigrunnlaget for metodene lagt fram. Deretter følger implementeringen som beskriver programvare og maskinvare brukt i oppgaven, samt tar for seg utviklingen av kildekode. Resultatkapittelet deles to deler, innledende testing av systemet og deretter testing av kartkorrelasjonsmetodene. Den påfølgende drøftingen diskuter resultatene opp mot målene, diverse feilkilder og kartkorrelasjon som et konsept. Til slutt vil det bli presentert en konklusjon av oppgaven med videre anbefalinger.

2 Teori

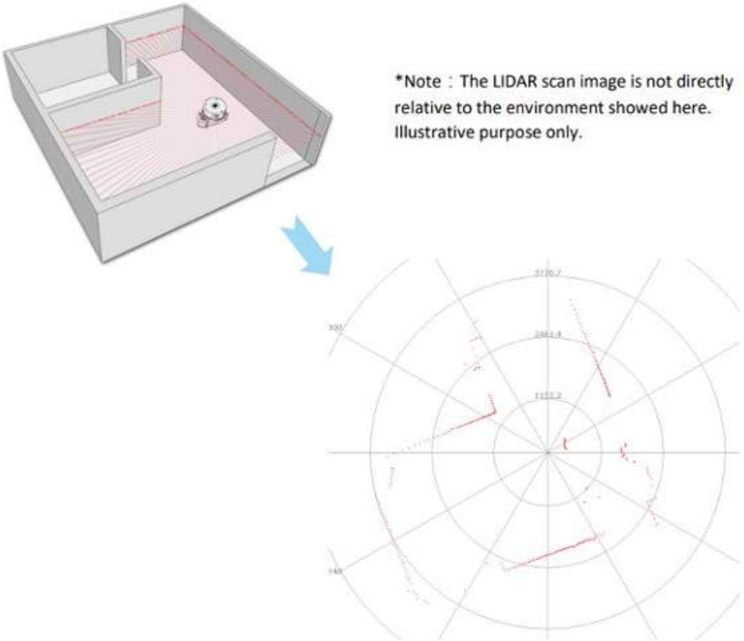
Dette kapittelet vil omhandle teori som skal gi en grunnleggende forståelse for sensoren og programvaren som ligger til grunn for å frembringe et lokalt kart. Videre vil kapittelet ta for seg nødvendig teori for å forstå virkemåten til de to metodene brukt til kartkorrelasjon i dette prosjektet, Oriented FAST and Rotated BRIEF (ORB) og Template Matching.

2.1 Sensorer for posisjonering

Det finnes mange forskjellige sensorer som kan bli brukt til posisjonering. Et eksempel er sensorer som samhandler med satellittnavigasjonssystemer. Mottakere regner ut avstanden til satellitter ved å sammenligne tiden fra når et signal ble kringkastet og mottatt. Med denne informasjonen fra minst 4 ulike satellitter kan man beregne vertikal og horisontal posisjon med en nøyaktighet på noen få meter. (Norsk Romsenter, 2016)

Man kan også benytte seg av treghetsnavigasjonssystemer for å beregne posisjon. Slike systemer er ikke avhengig av ytre signaler fordi de baserer seg på eksempelvis akselerometre og gyroskop, som måler bevegelse over tid. Man kan derfor beregne egen posisjon dersom man har kjennskap til utgangsposisjonen. Ulempen med treghetsnavigasjonssystemer er at feilene i hver måling akkumulerer over tid. Små feil i beregninger vil derfor resultere i økende posisjoneringsfeil over tid. Sagem sigma-40 er et eksempel på en relativ høy-kvalitets treghetssensor som er brukt av mange militære fartøyer. Dette systemet har en usikkerhet på 1 nautisk mil per døgn. (Kjerstad, 2022)

Det er også mulig å bruke aktive sensorer til posisjonering. Aktive sensorer sender ut signaler og registrerer endringer i det returnerte signalet. Radar (Radio Detection And Ranging) og Lidar (Light Detection And Ranging) er eksempler på slike sensorer (NASA, 2021). Lidar er en fjernmålingsteknikk som benytter seg av lys, og sammenligner differansen mellom utsendt og reflektert lys for å beregne avstand og bevegelser til objekter i måleområdet (Holtet, 2022). En lidarsensor som roterer rundt, vil derfor kunne lage et «bilde» av området og vite sin lokale posisjon. Figur 2-1 illustrerer dette.



*Note：The LIDAR scan image is not directly relative to the environment showed here. Illustrative purpose only.

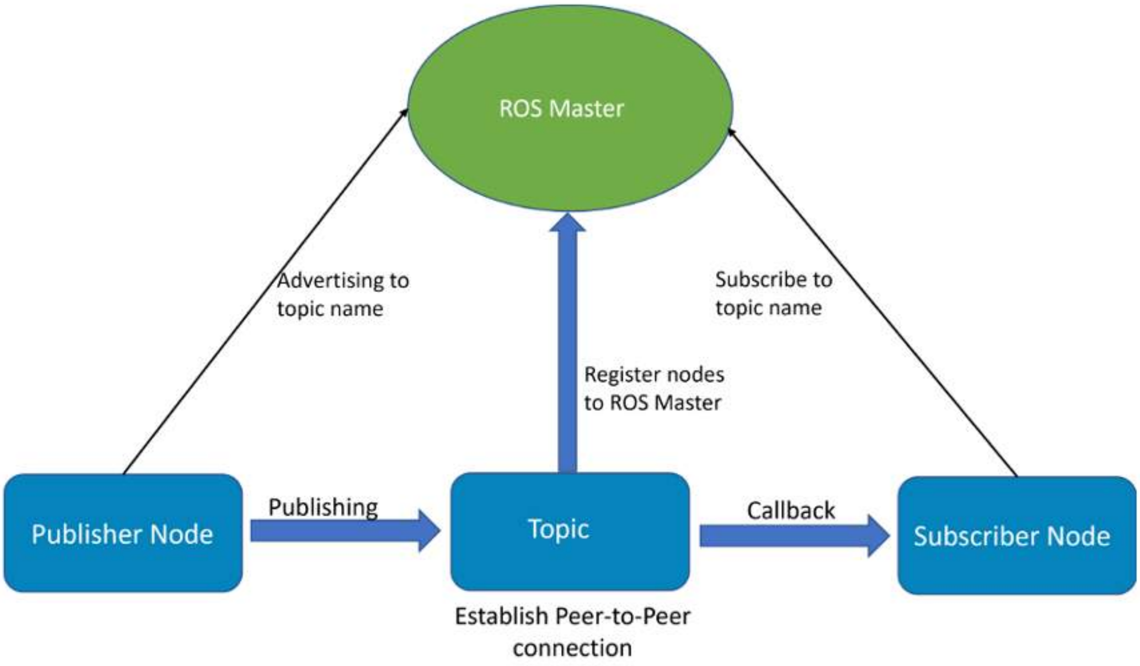
Figur 2-1: Bildet viser en lidarsensor som befinner seg i et rom. (Shanghai Slamtec Co., Ltd, 2016) Blå pil peker mot en illustrasjon som viser hva sensoren oppfatter rundt seg.

Muligheten til å danne seg et bilde av omgivelsene rundt gjør at lidarsensorer er hensikts-messig å bruke til å generere lokale kart. Ettersom lidarsensoren ikke beholder målingene av seg selv trengs det programvare som behandler dataen over tid for å kunne bygge et kart.

2.2 Programvare for å lage lokale kart

Robot Operating System (ROS) er et sett med programvarebibliotek og verktøy som man bruker til å bygge applikasjoner for å operere roboter. (Open Robotics, 2021) Bachelor-oppgaven til Hassel et al fra 2019 benyttet ROS til å innhente og behandle data fra Lidar sensoren. Denne informasjonen ble videre brukt av en pathfindingalgoritme som til slutt sendte styringskommandoer til plattformens motorer. ROS har også en relevans i denne oppgaven. De neste avsnittene vil derfor presentere den mest vesentlige teorien bak ROS.

Programvare i ROS er organisert i pakker. Disse pakkene inneholder diverse filer som til sammen fungerer som en brukbar modul. Intensjonen med denne organiseringen er at det skal være enkelt å gjenbruke programvaren og funksjonaliteten den har (Open Robotics, 2019). ROS Melodic, som er en distribusjon av ROS, er et eksempel på ett sett med diverse pakker.



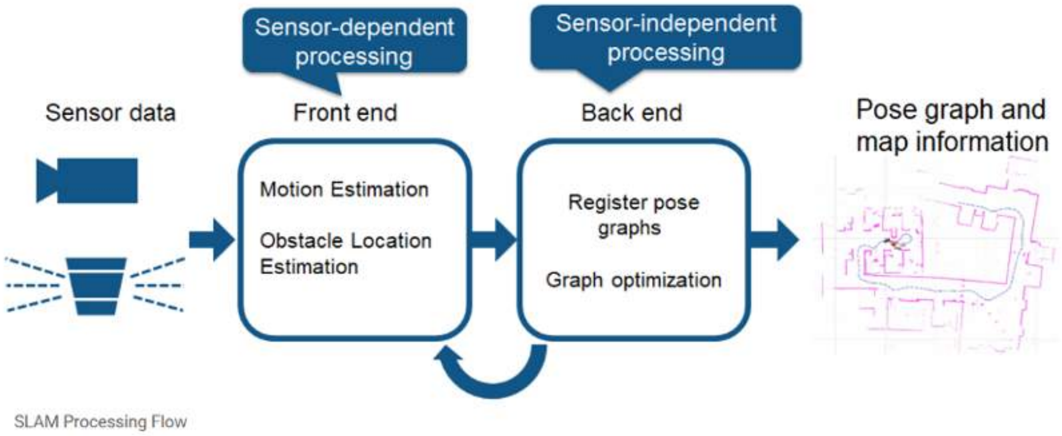
Figur 2-2: Viser til hvordan noder kommuniserer med hverandre i ROS. (Studica, 2021)

ROS har en del konsepter for å beskrive hvordan et nettverk av prosesser samhandler. **Noder** er prosesser som gjennomfører beregninger. Et ROS system består ofte av flere noder med forskjellige arbeidsoppgaver. Eksempelvis kan en node styre motoren på en robot, en annen produsere lokaliseringsdata, mens en siste sørger for at alt blir vist i et visualiseringsprogram. Noder sender meldinger (**messages**) for å kommunisere med hverandre. Dette er illustrert med blå piler i Figur 2-2. Meldingene er samlet i korte datastrukturer som for eksempel matriser, som kan bestå av diverse datatyper. (Open Robotics, 2022)

Meldinger blir sendt og mottatt via **topics**. Konseptet baserer seg på at produsenter av informasjon og mottakere av informasjonen holdes adskilt og ikke vet hvem den andre er. Flere noder kan både publisere informasjon under et gitt topic, og motta informasjon ved å abonnere på det. Topics er navngitt på måten «/navnpåtopic» og alle kan «koble seg til» den og dele informasjon, så lenge dataene er av riktig format og type. (Open Robotics, 2022) For at nodene skal finne hverandre og ha anledning til å kommunisere trenger man en **ROS Master**. Dens oppgave er å føre register på hvilke noder som abonnerer og publiserer til hvilke topics og services. Når nodene har lokalisert hverandre ved hjelp av ROS masteren, kommuniserer de på egenhånd. (Open Robotics, 2018)

ROS er altså et verktøy man kan bruke for å operere roboter. Det kan også brukes til å hente inn og behandle data fra en lidarsensor. For å systematisere denne dataen og faktisk generere lokale kart av området rundt, kan man installere pakker som bruker prinsippene til Simultaneous Localization And Mapping (SLAM).

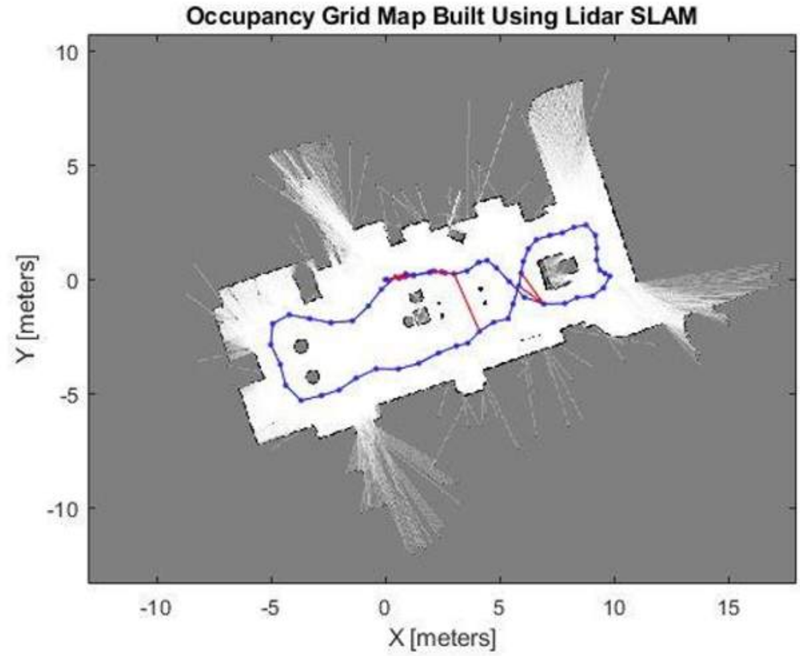
SLAM er en metode for å bygge kart ved hjelp av sensorer, og videre posisjonere et kjøretøy i det gitte kartet på en og samme tid. Generelt kan SLAM deles opp i to prosesser. Den første er den initiale prosesseringen av sensordata. (Front end i Figur 2-3) Kort fortalt blir utdataen fra sensoren tolket og systematisert i den hensikt å estimere bevegelse og kartlegge objekter. Den andre prosessen kalles pose-graph optimization (PGO). (Back end i Figur 2-3)



Figur 2-3: Prosessoversikt for SLAM. (The MathWorks, Inc., 2023)

SLAM estimerer bevegelse over tid, og PGO er teknikker for å minimere eventuelle feil i beregningene. Dersom dette ikke skjer, kan feil akkumulere, noe som resulterer i at det SLAM genererer ikke representerer virkeligheten. (The MathWorks, Inc., 2023) Dette kan eksempelvis skje Pose er et begrep innenfor robotikk som brukes når man snakker om posisjonen sammen med orienteringen til en robot. Pose graph er visualiseringen av flere poses fra forskjellige tidspunkt, og sammenhengen mellom dem (simsangcheol, 2023). Se Figur 2-4 for et eksempel på pose graph.

LiDAR SLAM baserer seg rundt lasersensorer som bruker lidar teknologi. Fordelen med lasere er at de er presise. Dette gjør at de ofte benyttes til droner og autonome kjøretøy som skal bevege seg fort. Lasere genererer som oftest point cloud data, som er et sett med datapunkter som til sammen representerer hva den aktuelle sensoren oppfatter rundt seg. For å lokalisere en enhet som bruker LiDAR SLAM, sammenlignes point cloud data fortløpende for å kartlegge bevegelse over tid. Denne sammenligningen kalles for point cloud matching.



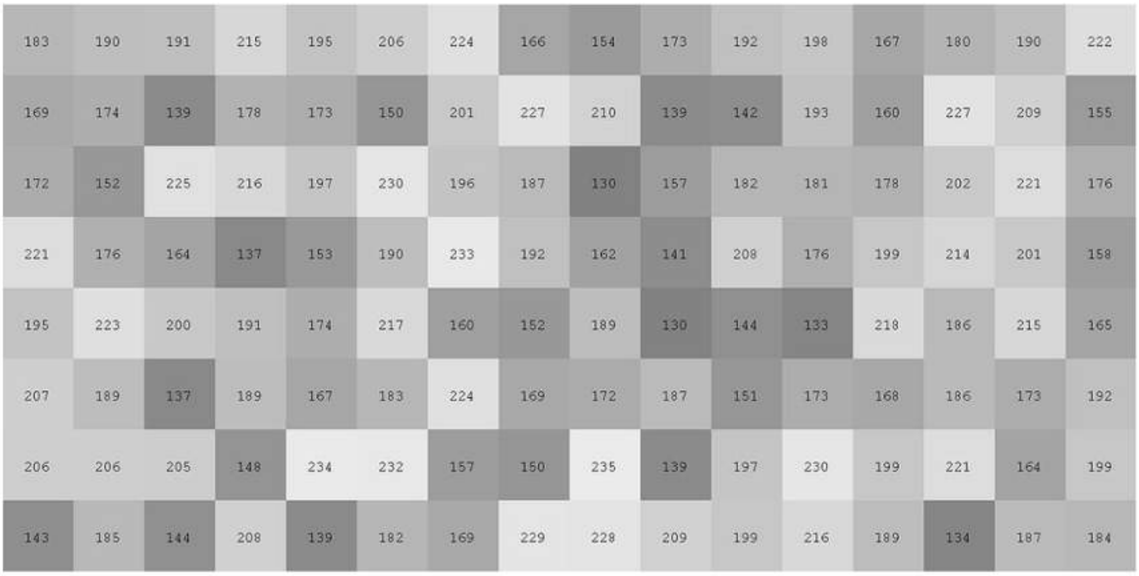
Figur 2-4: Kart generert ved bruk av Lidar SLAM. (The MathWorks, Inc., 2023) Blå linje viser hvor sensoren har beveget seg og er et eksempel på pose graph. Grått representerer usett område. Hvitt representerer sett område uten hindringer. De sorte linjene representerer sett område med hindring. Disse elementene resulterer i et kart over området som ble kartlagt.

Det kan av og til skje at lidarsensoren ikke klarer å registrere nok distinkte datapunkter til å gjennomføre point cloud matching på en god måte. Dette kan føre til lokaliseringsfeil som vil si at posisjonen en enhet har i sitt generte kart ikke samsvarer med virkeligheten. Lokaliseringsfeil kan for eksempel forekomme dersom det befinner seg få objekter i nærheten av sensoren, eller at man beveger seg i en lang og monoton gang. I tillegg krever point cloud matching en del prosessorkraft, noe som kan føre til at man må senke hastigheten til et kjøretøy for å oppnå tilfredsstillende resultater. En løsning for å minimere feil fra disse utfordringene er å faktorisere inn data fra odometry eller GNSS i lokaliseringsprosessen. (The MathWorks, Inc., 2023) Odometry er bruken av data fra bevegelsessensorer for å beregne endring av posisjon over tid. Et eksempel på odometry er bruken av enkodere på hjul til å bestemme avstand som er kartlagt (Singh, 2022).

2.3 Kartkorrelasjonsalgoritmer

2.3.1 Kart i form av bilder

I denne oppgaven er kartene som sammenlignes representert som bilder, og det er i praksis bilder som sammenlignes av algoritmene benyttet i denne oppgaven. På datamaskiner representeres bilder som matriser av binære tall. Hver piksel i et bilde tilsvarer et element i matrisen og er representert med en binær verdi.



183	190	191	215	195	206	224	166	154	173	192	198	167	180	190	222
169	174	139	178	173	150	201	227	210	139	142	193	160	227	209	155
172	152	225	216	197	230	196	187	130	157	182	181	178	202	221	176
221	176	164	137	153	190	233	192	162	141	208	176	199	214	201	158
195	223	200	191	174	217	160	152	189	130	144	133	218	186	215	165
207	189	137	189	167	183	224	169	172	187	151	173	168	186	173	192
206	206	205	148	234	232	157	150	235	139	197	230	199	221	164	199
143	185	144	208	139	182	169	229	228	209	199	216	189	134	187	184

Figur 2-5: Viser hvordan et bilde er bygget opp av piksler med forskjellig grad av intensitet. Bildet er i greyscale. Hver piksel er representert med en verdi som viser intensiteten fra 0 (sort) til 255 (hvitt). **(Nogare, 2019)**

Den binære verdien representerer fargen og intensiteten den har som vist i figur 2-5. Graden av dybde og intensitet farger kan ha, bestemmes av hvor mange bits hver piksel blir representert av. Har bildet bare to farger, sort og hvitt, kan det representeres som en bit der 0 representerer sort, og 1 representerer hvitt. Et bilde kan ha 8 bit per piksel, som gir 256 farger, i tillegg kan piksler representeres med 16 og 24 bits. I et 8-bit greyscale-bilde, hvor fargene varierer mellom sort og hvitt, vil sort som er fargen med mest intensitet representeres som 0, og hvitt som er mest intens representeres som 255. (Ragipally, 2020)

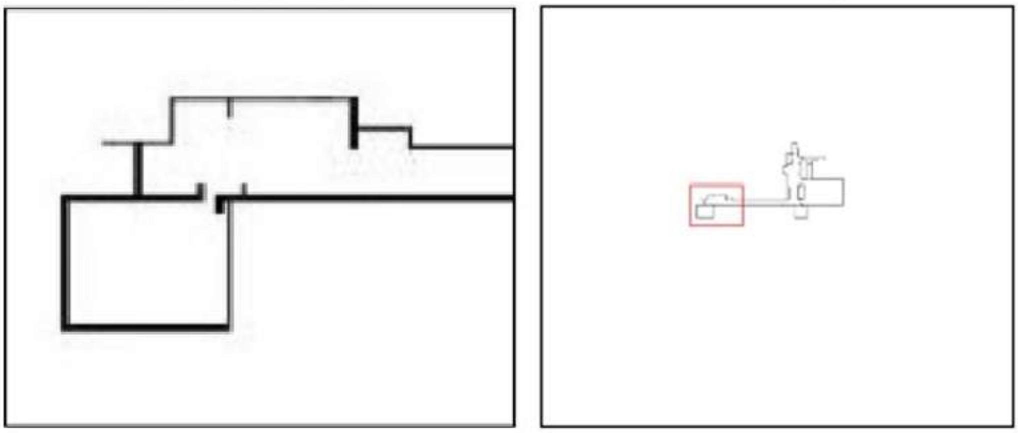
2.3.2 Deteksjon av kjennetegn

Bildeanalysealgoritmer som sammenligner bilder, gjør det med å finne unike deler av hvert bilde og deretter se om disse delene ligner på hverandre. De unike delene av et bilde blir kalt en feature eller et kjennetegn. I et bilde kan et kjennetegn ta form som et punkt, et objekt, eller et hjørne. (Tyagi, Introduction To Feature Detection And Matching, 2019)

Deteksjon av kjennetegn er en prosess som identifiserer kjennetegn i bilder og representerer det som et numerisk objekt. (MathWorks, u.d.). Disse kjennetegnene blir ofte beskrevet basert på hvordan området rundt ser ut. Denne prosessen gjør det enklere å sammenligne kjennetegn i bilder og reduserer dermed tiden det tar for algoritmer å kjøre. (DeepAI, u.d.) Ulike algoritmer kan ha forskjellige måter å beskrive et kjennetegn. Metoden brukt av ORB er beskrevet i kapittel 2.3.1.

2.3.3 Template

Template Matching-algoritmen fungerer ved å sammenligne et «Template» med et bilde. I kontekst av Template Matching kan Template oversettes til begrepet «utsnitt». Template Matching blir brukt til å finne posisjonen til en mindre del av et bilde, eller et utsnitt, i et større bilde (se Figur 2-6). Dette blir gjort ved å sammenligne utsnittet med hver del av bildet (OpenCV, u.d.). Resultatet er basert hvilket område i bildet som hadde høyest korrelasjonsverdi sammenlignet med utsnittet. Hvordan korrelasjonsverdien blir regnet ut beskrives i kapittel 2.3.2. For kartkorrelasjon som er en sammenligning av to kart, et mindre kart og et større kart, vil utsnittet i være det lokale og mindre kartet. Det lokale kartet er et utsnitt av et kart over et større område.



Figur 2-6: Viser et utsnitt (til venstre) av et kart (til høyre). Plasseringen til utsnittet er markert i kartet til høyre.

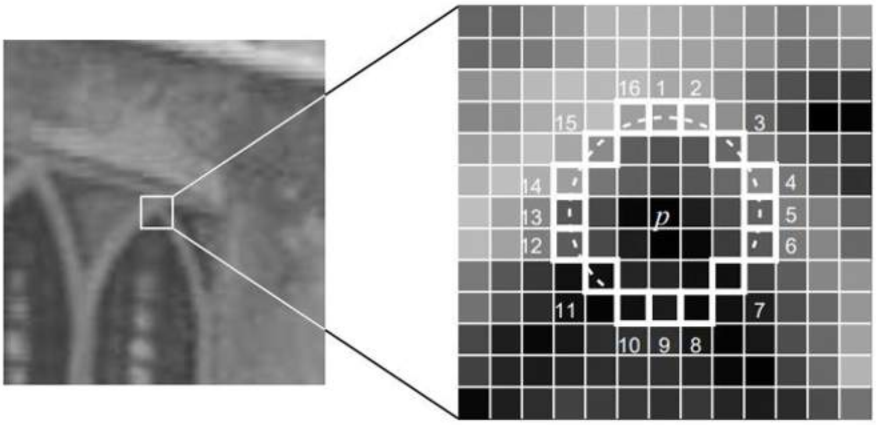
2.3.4 ORB

ORB er en «feature matching» algoritme som detekterer og beskriver kjennetegn i to forskjellige bilder, og sammenligner disse basert på beskrivelsene. ORB setter sammen og bygger videre på FAST-algoritmen som skal detektere kjennetegn, samt BRIEF algoritmen som er utviklet for å gi beskrivelser av kjennetegn funnet av FAST. Det er også lagt til flere elementer som orientering og skalering for å utbedre resultatene fra FAST og BRIEF, samt samspillet mellom de i ORB-algoritmen (Rublee, Rabaud, Konolige, & Bradski, 2011). ORB ble utviklet som et alternativ til lignende algoritmer som SIFT (Scale Invariant Feature Transform) og SURF (Speeded Up Robust Features) som ikke er open-source (OpenCV, u.d.). Formålet til utviklerne av ORB var å skape en open-source algoritme som krevde mindre prosesseringskraft, men samtidig kunne konkurrere med de eksisterende algoritmene på resultater (Rublee, Rabaud, Konolige, & Bradski, 2011, s. 1).

FAST

ORB benytter seg av FAST som er en «feature detection» algoritme, en algoritme som finner kjennetegn i form av hjørner. Et hjørne kan defineres et punkt i bildet hvor det er betydelig endring i intensitet i alle retninger (Tyagi, Introduction to Harris Corner Detector, 2019). FAST detekterer kjennetegn i et bilde med en segmenttest. Segmenttesten har til hensikt å sjekke om piksler oppfyller kriteriet satt til å være et kjennetegn. Kriteriet for at en piksel skal defineres som et kjennetegn er at et satt antall sammenhengende piksler i en sirkel rundt pikselen, enten er lysere eller mørkere enn pikselen i fokus.

Segmenttesten blir gjort på alle piksler i bildet. Den fungerer ved å ta en piksel, definert som p, for så å definere en sirkel bestående av 16 piksler rundt p (Se Figur 2-7). Videre ser algoritmen på intensiteten til punktet p og punktene i sirkelen. Intensiteten i hver piksel er representert av et tall. I et sort/hvitt-bilde vil intensiteten variere fra 0 til 255, hvor hvitt er 255 og sort er 0. Intensiteten til p defineres som Ip, i tillegg blir det definert en terskelverdi t som har til hensikt å motvirke effekten av støy i bildet, og dermed redusere mengden «false-positives». (Rosten, Porter, & Drummond, 2010)



Figur 2-7: Viser sirkelen bestående av 16 piksler definert rundt det mulige hjørnet, pikselen p (Rosten, Porter, & Drummond, 2010, s. 13).

Kriteriet for at p skal defineres som et hjørne er at 12 sammenhengende piksler i sirkelen er lysere enn p, og dermed har en høyere verdi enn (Formel 2-8)

$$Ip + t$$

Formel 2-8 : Øvre terskelgrense for segmenttesten. Terskelgrensen er satt av intensiteten til en piksel Ip pluss den satte terskelverdien t.

eller har en lavere verdi enn (Formel 2-9),

$$Ip - t$$

Formel 2-9: Nedre terskelgrense for segmenttesten. Terskelgrensen er satt av intensiteten til en piksel Ip minus den satte terskelverdien t.

og dermed er mørkere enn p. For å redusere tidsbruk blir det gjort en innledende test for å se om pikslene har potensial til å kunne oppfylle kriteriet til et hjørne. Den reduserer tidsbruk gjennom å se på to piksler om gangen istedenfor å sjekke alle 16 piksler i området rundt p. Dette gjør at algoritmen raskt kan eliminere piksler som ikke har potensiale til å være et hjørne, og sparer dermed prosessorkraft. FAST algoritmen ser først på piksel 1 og 9 (se Figur 2-7), dersom begge disse har en lavere verdi enn nedre del av terskelgrensen (Formel 2-9), eller en høyere verdi enn øvre del av terskelgrensen (Formel 2-8), vil piksel p fortsatt være et mulig hjørne. Dersom begge pikslene ikke ligger utenfor terskelgrensen vil p forkastes som et mulig hjørne. Det neste algoritmen ser på er piksel nr. 5 og 13. Dersom tre av de fire pikslene 1, 5, 9 og 13 er lysere enn $Ip + t$ eller mørkere enn $Ip - t$ vil det kjøres en full test på p. Algoritmen sjekker om det er 12 sammenhengende piksler i den definerte sirkelen som alle har en høyere eller lavere verdi enn terskelgrensene og bestemmer om p er et kjennetegn. (Rosten, Porter, & Drummond, 2010)

Orientering og skalering i ORB

FAST har ingen evne til å ta hensyn til hverken orientering eller skalering. ORB modifiserer FAST for å løse dette. Metoden brukt for å orientere kjennetegn er kalt «Intensity centroid». Det er en enkel metode som finner orienteringen til et kjennetegn ved å anta at intensiteten har utgangspunkt fra midten av kjennetegnet. Ved å finne retningen på vektoren ut fra midten av kjennetegnet finner man den tilhørende orienteringen. (Rublee, Rabaud, Konolige, & Bradski, 2011, ss. 2-3).

Metoden definerer «moments» (Formel 2-10) til et utsnitt av bildet. Et moment er en matematisk størrelse, som i sammenheng med bildeanalyse er et gjennomsnitt av intensiteten til pikslene i et utsnitt av bildet. (Academic Accelerator, 2023)

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y)$$

Formel 2-10: Formel for moment til et utsnitt av et bilde. $Moment$et m_{pq} er summen av produktet til pikselintensiteten til et punkt $I(x,y)$ og koordinatverdiene x, y opphøyd i p, q.

Neste steg i orienteringsprosessen er å finne massesenteret til utsnittet av bildet. Formel 2-11 finner massesenteret i form av et (x, y) koordinat. Koordinaten er basert på moment i x og y retning. (Rublee, Rabaud, Konolige, & Bradski, 2011, s. 2)

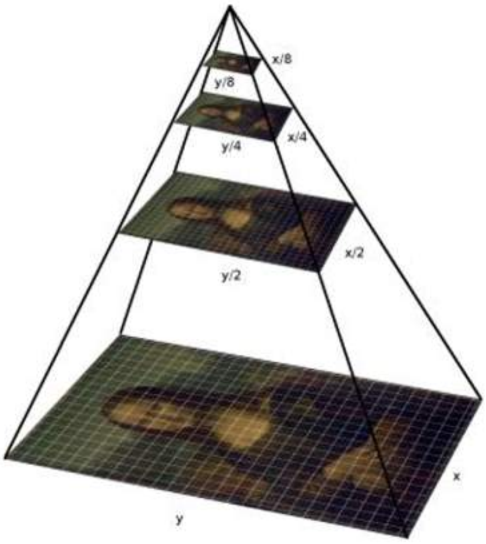
$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$$

Formel 2-11: Formel for å finne massesenteret til et utsnitt av et bilde. Koordinaten til massesentret C er gitt av moment i x retning m_{10}, og moment i y retning m_{01} delt på totalt moment i utsnittet m_{00}.

Koordinaten som kommer fra Formel 2-11 gjør det mulig å konstruere en vektor som går fra massesenteret C til senter av hjørnet. Denne vektoren kan videre benyttes til å finne orienteringen på kjennetegnet med invers tangens (Formel 2-12). (Rublee, Rabaud, Konolige, & Bradski, 2011, s. 2)

$$\theta = atan2(m_{01}, m_{10})$$

Figur 2-12: Formel for å finne orientering til kjennetegnet. Vinkelen θ blir funnet med invers tangens $atan2$ av koordinaten til moment i x retning m_{10} og y retning m_{01}.



Figur 2-13: Viser prinsippet med pyramideskalering. Bildet blir skalert ned med en satt faktor et brukerdefinert antall ganger. **(Tyagi, Introduction to ORB (Oriented FAST and Rotated BRIEF), 2019)**

For å løse problemstillingen med ulik skalering på bilder. benytter skaperne av ORB seg en skaleringspyramide som illustrert i Figur 2-13. En skaleringspyramide fungerer ved å ta det originale bildet, og skalere dette med en faktor. Dette blir gjort flere ganger, der antall ganger er brukerbestemt, slik at bildet blir representert med flere ulike oppløsninger hvor FAST kan finne kjennetegn i hvert av de skalerte bildene. (Rublee, Rabaud, Konolige, & Bradski, 2011, s. 2)

Pyramideskalering reduserer i hvor stor grad ORB blir påvirket av forskjell i skalering mellom bilder, og algoritmen kan oppdage kjennetegn delvis uavhengig av skaleringen til bildene. (Tyagi, Introduction to ORB (Oriented FAST and Rotated BRIEF), 2019)

BRIEF

Når kjennetegnene er funnet i to kart som skal sammenlignes må kjennetegnene kunne identifiseres på tvers av kartene. Dette kan bli gjort ved å gi hvert kjennetegn en beskrivelse. Disse beskrivelsene har til hensikt å muliggjøre sammenligning av kjennetegn i to forskjellige bilder. Beskrivelsene kommer i form av «Binary strings», som bør være så unik som mulig for hvert individuelle kjennetegn. BRIEF er en algoritme laget for å gi slike beskrivelser av kjennetegn, funnet av en algoritme, som for eksempel FAST.

Bakgrunnen til utviklingen av BRIEF var et behov for å redusere tiden brukt til å sammenligne kjennetegn. Dette gjør den bedre egnet til bruk i SLAM-algoritmer som er avhengig av å kunne behandle og prosessere mange kjennetegn. Samtidig krever bruk av slik teknologi på mindre enheter som mobiler og på systemer som behandler store mengder data, at prosessen bruker mindre prosesseringskraft, og har et redusert bruk av minne. (Calonder, Lepetit, Strecha, & Fua, 2010, ss. 1-2)

BRIEF er en algoritme som ser på en piksel om gangen. Det fører til at algoritmen er lett påvirket av støy i bildet. Derfor gjør BRIEF en «smoothing» av en mindre del av bildet rundt pikselen den ser på, før den blir prosessert. «Smoothing»'en er av en type som kalles «Gaussian smoothing» som gir bildet en normalfordelt (Gaussisk) uskarphet. Ved å gjøre dette før man begynner med utregning vil stabiliteten til beskrivelsene av kjennetegn øke, samtidig som sensitiviteten for støy reduseres. (Calonder, Lepetit, Strecha, & Fua, 2010, s. 4)

Etter «smoothing»'en er gjort blir det laget en bitstreng på følgende måte:

$$\tau(p; x, y) := \begin{cases} 1 & if\ p(x) < p(y) \\ 0 & otherwise \end{cases}$$

Formel 2-14: Viser definisjonen av testen τ på et utsnitt av bildet p av dimensjon $S\ x\ S$. $p(x)$ representer pikselintensiteten i et gitt punkt. Deretter blir det valgt to testpiksler (x_i, y_i) innenfor området p. Disse pikslene settes inn i testen og sammenlignes

Formel 2-14 viser definisjonen av testen τ på et utsnitt av bildet p av dimensjon $S\ x\ S$. $p(x)$ representer pikselintensiteten i et gitt punkt. Deretter blir det valgt to testpiksler (x_i, y_i) innenfor området p. Disse pikslene settes inn i testen i Formel 2-14 og sammenlignes basert på intensitet. Resultatet går inn i Formel 2-15, hvor BRIEF beskrivelsen blir laget i form av en bitstreng. n_d representerer antall lokasjoner som blir testet og bestemmer dermed hvor lang strengen blir. Det er tre alternativer for n_d, 128, 256 og 512. I ORB algoritmen er n_d satt til 256. (Rublee, Rabaud, Konolige, & Bradski, 2011, s. 4)

$$f_{n_d}(p) \coloneqq \sum_{1 \le i \le n_d} 2^{i-1}\,\tau(p; x_i, y_i)$$

Formel 2-15: For å lage beskrivelse av et kjennetegn. Beskrivelsen eller bitstrengen f_{n_d} er gitt av summen av produktet mellom 2^{i-1} og testen τ for antall lokasjoner n_d som testes

Valg av lokasjoner eller piksler (x_i, y_i) skjer tilfeldig innenfor området p. Hvert pikselpar er valgt uavhengig av hverandre. Metoden som blir brukt er en normalfordeling av piksler ut fra senter av p. Det legges inn en varians på $\frac{1}{25}S^2$ for å spre lokasjonene for best mulig grunnlag for sammenligning. **(Calonder, Lepetit, Strecha, & Fua, 2010, ss. 5-6)**

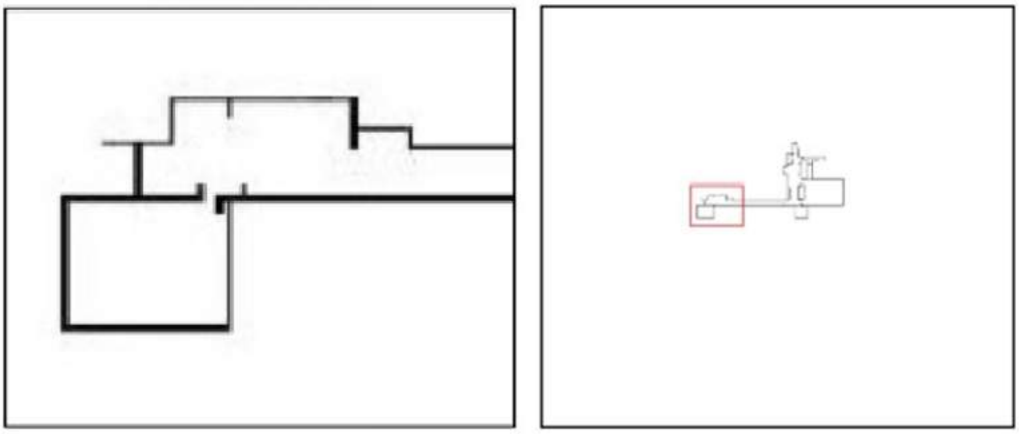
Valg av lokasjoner eller piksler (x_i, y_i) skjer tilfeldig innenfor området p. Hvert pikselpar er valgt uavhengig av hverandre. Metoden som blir brukt er en normalfordeling av piksler ut fra senter av p. Det legges inn en varians på $\frac{1}{25}S^2$ for å spre lokasjonene for best mulig grunnlag for sammenligning. **(Calonder, Lepetit, Strecha, & Fua, 2010, ss. 5-6)** BRIEF er ikke utviklet til å ta hensyn til rotasjon i bildet. Det kan føre til at resultatene blir svakere når kjennetegn skal sammenlignes i to bilder med ulik grad av rotasjon. For å redusere sensitiviteten mot forskjeller i rotasjon benytter ORB-algoritmen orienteringen funnet for kjennetegn i FAST og gir dermed BRIEF et bedre utgangspunkt for å sammenligne dem. **(Rublee, Rabaud, Konolige, & Bradski, 2011, s. 3)**

Metoden ORB bruker for å sammenligne kjennetegn er en «Brute force matcher». Som navnet tilsvarer, tar denne metoden og sammenligner alle kjennetegn i et bilde med alle kjennetegn i det andre bildet. De blir sammenlignet basert på beskrivelsen laget av BRIEF. (GeeksforGeeks, 2023) Resultatene av «Brute force»-sammenligningen blir målt i «Hamming Distance». (Rublee, Rabaud, Konolige, & Bradski, 2011) Distansen er et mål på hvor forskjellig to bitstrenger er, og i kontekst av ORB vil det være et mål av hvor forskjellig beskrivelsen til kjennetegn er. Hver posisjon i beskrivelsen til kjennetegnene som sammenlignes blir sjekket, og for alle posisjoner som er forskjellig, øker distansen. Dermed vil de beste «matches» være de med kortest distanse. (Pradhan, 2023)

For å oppsummere er ORB en algoritme som sammenligner kjennetegn i to bilder. For å oppnå dette implementerer den to algoritmer, FAST og BRIEF, samt legger til funksjoner for skalering og orientering for å forbedre samhandlingen mellom disse. For å kunne sammenligne kjennetegn finner FAST kjennetegn i begge bildene, BRIEF lager en beskrivelse av disse basert på området rundt kjennetegnet. ORB bruker deretter en «Brute Force matching» for å sammenligne kjennetegn.

2.3.5 Template matching

Template matching er en metode for å se om et utsnitt av et bilde og dens karakteristikker korrelerer med et tilsvarende område på et større bilde. Algoritmen sammenligner utsnittet med alle lokasjoner på bildet og finner det området på bildet som er mest likt (se Figur 2-16) (OpenCV, u.d.).



Figur 2-16: Samme figur som 2-6. Figuren viser et utsnitt (til venstre) og et kart (til høyre). Template Matching har funnet riktig området på kartet markert med en rød firkant.

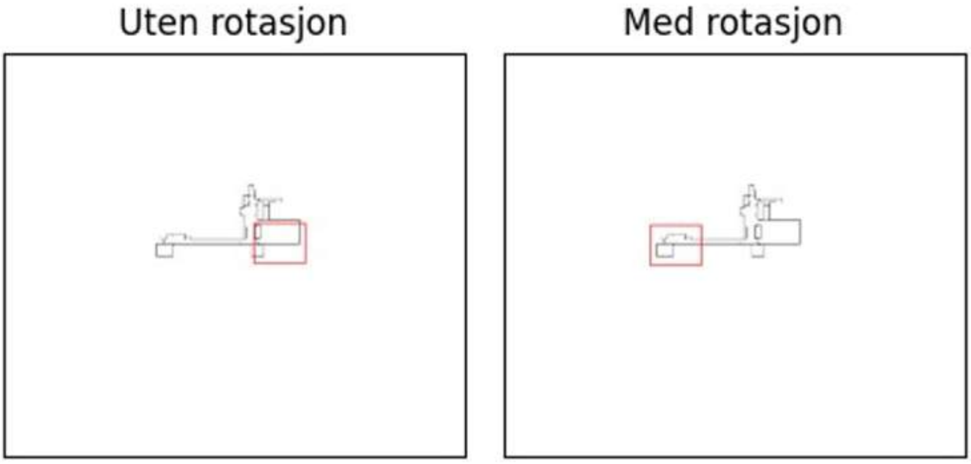
Template Matching algoritmen fungerer ved å flytte utsnittet en piksel av gangen, helt til utsnittet har blitt sammenlignet med hvert område i bildet (OpenCV, u.d.). For hvert område i bildet som blir sammenlignet med utsnittet blir det gitt en korrelasjonsverdi, en verdi som beskriver hvor god match området er med utsnittet. Denne korrelasjonsverdien blir gitt ut ifra Formel 2-17 (OpenCV, u.d.).

$$R(x,y) = \sum_{x',y'} \left(T'(x',y') * I'(x+x',y+y')\right)$$

Formel 2-17: Formel for å regne ut korrelasjonsverdi mellom et utsnitt og et område av et bilde. Korrelasjonsverdien $R(x,y)$ er av summen av alle produkter mellom intensiteten til koordinatene $T'(x',y')$ i utsnittet og intensiteten til de tilsvarende koordinatene i bildet $I'(x+x',y+y')$.

I Formel 2-17 representerer $R(x,y)$ korrelasjonsverdien til området rundt piksel (x,y). T og I representerer utsnittet og bildet. Piksel (x,y) er posisjonen i bildet til øverste venstre hjørne av utsnittet, altså posisjon (0, 0) i utsnittet. Piksel (x',y') referer til posisjon i utsnittet. For å finne korrelasjonsverdier er første steg i formelen å finne produktet av pikselintensiteten til alle korresponderende piksler i utsnittet og bildet. Korrelasjonsverdien blir så gitt som summen av alle produktene. Det området med høyest korrelasjonsverdi vil være den beste «match»'en.

Etter alle punktene i bildet har fått en korrelasjonsverdi blir det framstilt som et «greyscale»-bilde. Alle piksler i «greyscale»-bildet har en farge mellom sort og hvitt, hvor intensiteten er basert på korrelasjonsverdien. Pikselen med høyest intensitet i «greyscale»-bildet, med tilhørende område på bildet, har den høyeste korrelasjonsverdien. Det er dette området som passer best sammenlignet med utsnittet.

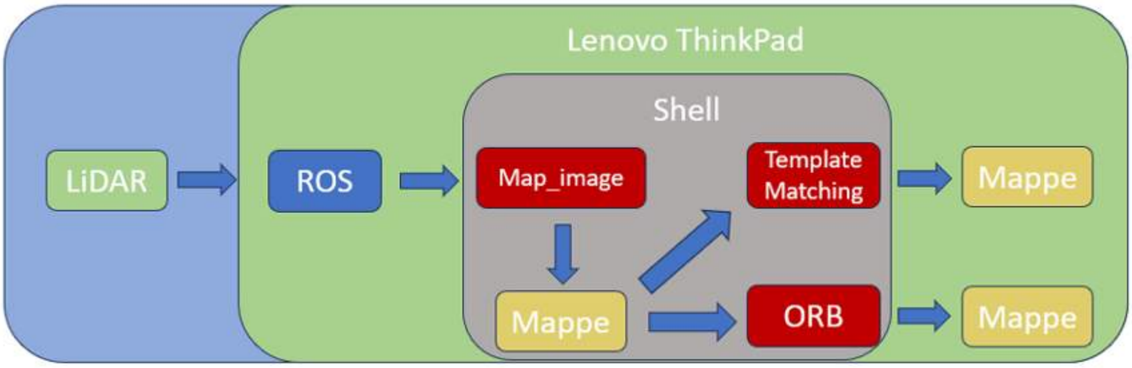


Figur 2-18: Viser to eksempler av resultater fra Template Matching med et rotert lokalt kart. På venstre side er mislykket forsøk på å sammenligne uten å rotere det interne kartet. Høyre side viser et vellykket forsøk etter å ha lagt in en rotasjonsfunksjon som tester ved ulike vinkler. Det interne kartet eller utsnittet er det samme som i figur 2-16

Template Matching er en metode som sammenligner et utsnitt direkte med et bilde. Selv om det kan være en god måte å finne området i bildet som tilsvarer utsnittet, er det noen svakheter med denne metoden. Metoden gjør allerede mange utregninger, og med et større bilder hvor det må gjøres enda flere utregninger, vil algoritmen bruke mye tid. Den er også avhengig av at utsnittet har lik orientering som bildet for å kunne få gode resultater. I tillegg vil større forskjeller i skala på utsnittet og bildet kunne føre til problemer med sammenligningen. Forskjeller i rotasjon og/eller skala mellom utsnittet og bildet kan føre til falske positiver som vist i Figur 2-16. (Upadhyay, 2023)

3 Implementering

Forrige kapittel tok for seg teorien som ligger bak fremstillingen av det lokale kartet, samt teorien som ligger bak algoritmene brukt til kartkorrelasjon, ORB og Template Matching. I dette kapittelet blir fremgangsmåten i prosjektet presentert. Kapittelet omhandler valg av plattform, programvare og operativsystem, samt utviklingen av kildekodene for kartkorrelasjon og de globale kartene.



Figur 3-1: Systemoversikt

Figur 3-1 viser systemoversikten i denne oppgaven. Hvert element blir forklart nærmere i dette kapittelet. Den grønne boksen representer LiDAR sensoren som kartlegger området rundt plattformen. De neste prosessene utføres på en PC (Lenovo ThinkPad). Utdataen fra sensoren blir prosessert av en SLAM-programvare. Samhandlingen mellom sensor og SLAM muliggjøres ved hjelp av ROS. Kartet blir hentet ut og lagret i mappestrukturen av Map_image-scriptet. Kartkorrelasjonsmetodene Template Matching og ORB, henter ut kartene lagret i mappestrukturen, sammenligner kartene og lagrer resultatene i mappestrukturen. Kildekoden, markert i rødt, blir styrt av et Shell-script.

3.1 Maskin- og programvare

3.1.1 Valg av plattform

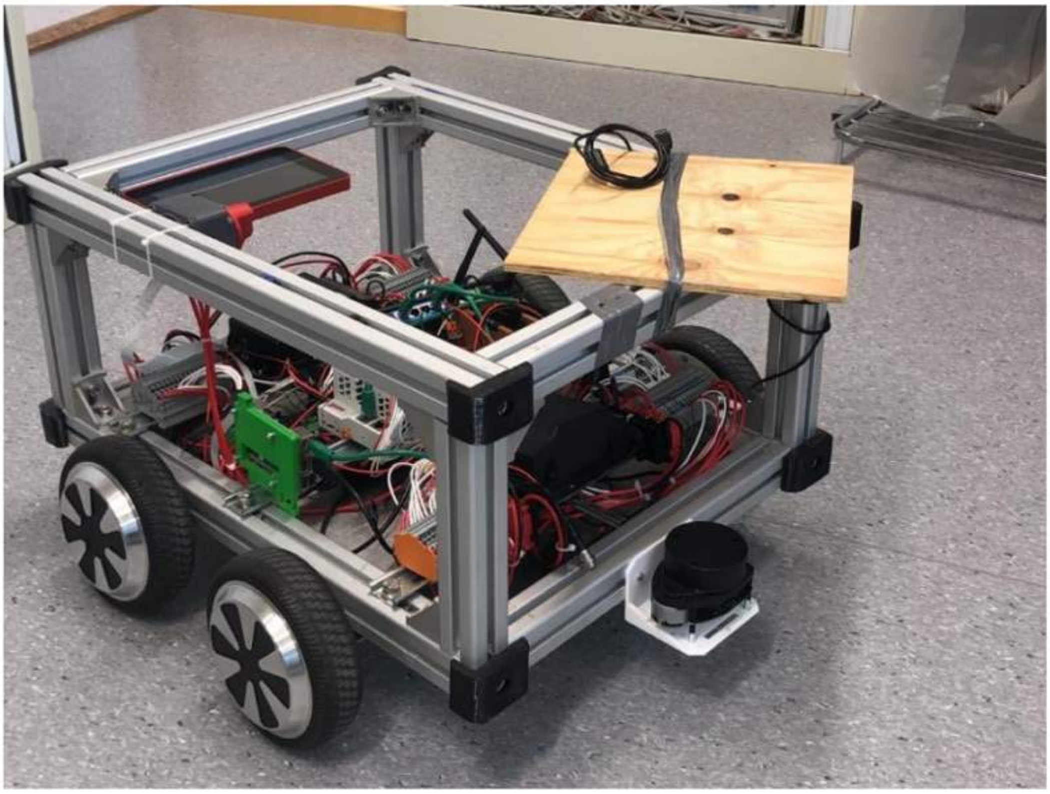
Det ble valgt å bruke den tidligere utviklede plattformen fra 2018 og 2019 som grunnlag for dette prosjektet. Denne plattformen var allerede utstyrt med en LiDAR sensor og hadde testet en metode for SLAM. Dette var fordelaktig for dette prosjektet, fordi man unngikk å bruke en vesentlig del av tiden til å implementere dette på et annet system. Ved å innføre metoder for kartkorrelasjon til denne plattformen, legger man grunnlaget for å utbedre den autonome navigeringen.

Figur 3-2: Bilder av plattformen (Hassel, Moen, & Ohma, 2019)

3.1.2 RPLidar A1 og dens plassering

For å kunne drive med kartkorrelasjon trenger plattformen en sensor for å se omgivelsene rundt seg og danne seg et bilde. RPLidar A1 er en optisk sensor som bruker prinsippet om fjernmålingsteknikken Lidar, som tidligere ble beskrevet i teoridelen. Sensoren roterer 360 grader rundt og sender ut infrarødt lys. Den evner å detektere objekter i et todimensjonalt plan mellom 0,15- og 12 meter, og har en samplingsfrekvens på opptil 8000 samplinger per sekund. (Slamtec Global network, 2023)

Figur 3-3: RPLidar A1 (Slamtec Global network, 2023)



Figur 3-4: Plattform med ny plassering av RPLidar A1

Det var ønskelig å gjøre plattformen i stand til å kartlegge områder både innendørs og utendørs. På bakgrunn av dette ble kartleggingsevnen testet i forskjellige høyder utenfor skolebygget. Resultatene viste til at dersom man plasserer Lidaren lavere, evner den å registrere flere punkter rundt seg, som igjen fører til bedre kartlegging av området. Grunnen til dette var hovedsakelig at sensoren fikk med seg veikantene ved lavere høyder. Det ble dermed vurdert å plassere Lidaren 13,5 cm over bakken i forkant av plattformen. En ulempe med denne endringen er at det reduserer sensordekningen fra 360 grader til cirka 180 grader. Dette medfører at kartleggingen ikke blir like effektiv, da man må rotere plattformen rundt for å full dekning. Likevel endte man opp med å gjennomføre endringen. Det ble 3D-printet et feste til lidar som vist på Figur 3-4.

Datamaskinen som ble benyttet til prosjektet er av typen Lenovo Thinkpad E470. Dette er maskinen som inneholder alt av programvare som ble lastet ned og benyttet i prosjektet.

Tabell 3-1: Spesifikasjoner for Lenovo E470. Hentet fra systemoversikt inne på datamaskinen.

Datamaskin	Lenovo Thinkpad E470
Prosessor	Intel® Core™ i7-7500U CPU @ 2.7GHz x 4
Grafikkort	Intel® HD Graphics 620 (KBL GT2)
RAM	8 Gb
Lagringsplass	250.9 GB

3.1.3 Valg av programvare

For å opprettholde kontinuitet med forrige bachelor var det ønskelig å benytte seg av samme distribusjoner av programvare. Nyere distribusjoner kan inneholde vesentlige strukturelle forandringer, og dette kunne eventuelt føre til at det måtte gjøres mye endringer i kildekoden som ble utviklet av Hassel et al i 2019. Man anså derfor at å installere samme distribusjoner antageligvis ville resultere i færrest problemer under installering, og testing av spesielt LiDAR-sensoren med tilhørende programvare. Til tross for dette ble det vurdert å heller installere de nyeste distribusjonene for å få alt oppdatert. Den endelige avgjørelsen man endte på var å bruke de nyeste versjonene av de opprinnelige distribusjonene. Det betyr at programvaren ble oppdatert så mye som det kunne uten å bytte til nye distribusjoner.

Operativsystem

Ubuntu 18.04.06 ble lastet ned fra Ubuntu sin offisielle nettside og deretter installert på datamaskinen (Ubuntu, u.d.). Ubuntu er et Linux basert operativsystem. Fordelen med dette operativsystemet er at man enkelt kan kjøre scripts og ha oversikt over programvare.

ROS, RPLiDAR_ROS og HECTOR_SLAM

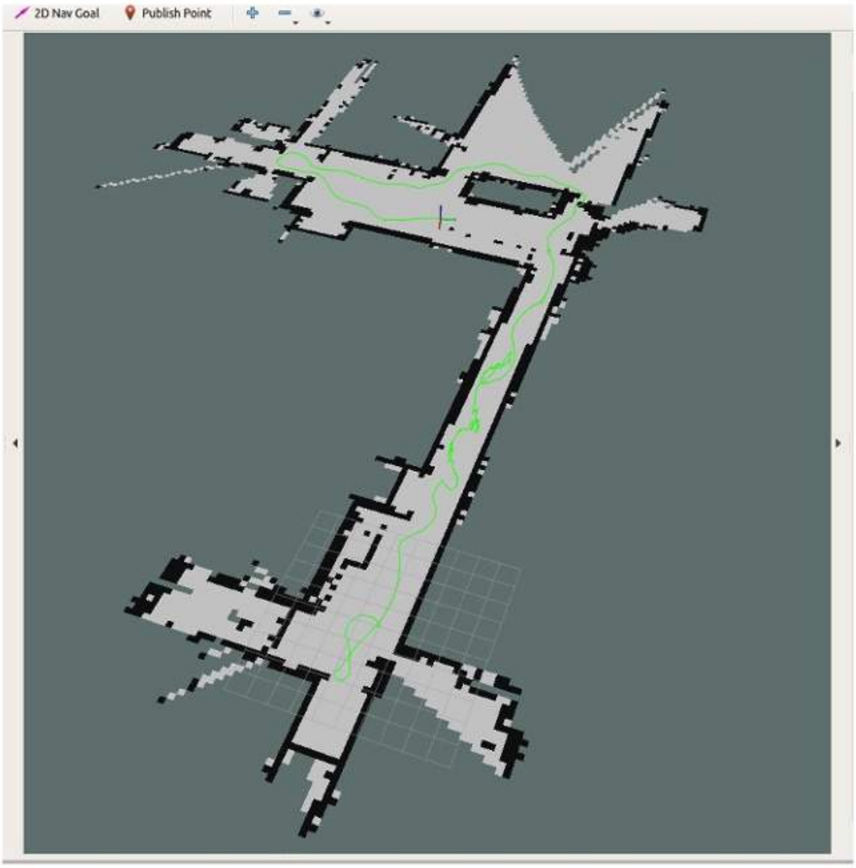
Tabell 3-2: Oversikt over er programvare. Se vedlegg B for oppskrift på innstallering av programvare.

Programvare/pakke	Versjon/ distribusjon	Hentet fra
ROS	MELODIC	http://wiki.ros.org/melodic/Installation/Ubuntu
RPLiDAR_ROS	1.9.0	https://github.com/robopeak/rplidar_ros
HECTOR_SLAM	0.5.2	https://github.com/tu-darmstadt-ros-pkg/hector_slam

ROS MELODIC ble installert på datamaskinen (Open Robotics, 2020). Deretter ble RPLiDAR_ROS installert (RoboPeak Public Repos, 2018). Denne pakken er nødvendig for at sensordataene kan samhandle med SLAM pakken. HECTOR_SLAM ble installert og lagt inn i et ROS miljø (Technische Universität Darmstadt ROS Packages, 2022). Denne pakken er designet til å utføre SLAM med data fra sensoren. I tillegg ble enkelte scripts fra kildekoden til forrige bachelorprosjekt lastet ned, og lagt inn i mappestrukturen på datamaskinen (thedanielhassel, 2019). Dette gjaldt modifiserte oppstartfiler som endrer hvordan det SLAM-genererte kartet blir fremvist i visualiseringsverktøyet. Verktøyet heter Rviz, og inngår som en del av ROS MELODIC. Se vedlegg B for ytterligere informasjon angående installering av programvare.

Kartlegging av områder ved hjelp av SLAM

For å kartlegge et område må man først aktivere LiDAR sensoren. Dette gjøres ved å kjøre oppstartfilen til pakken RPLiDAR_ROS i ROS miljøet. Deretter starter man opp HECTOR_SLAM ved å kjøre ønsket oppstartfil. I dette prosjektet ble oppstartfilen til forrige bacheloroppgave brukt, med noen modifiseringer. Oppløsningen på kartet som vises i Rviz ble økt fra 100x100 til 400x400 piksler. Deretter kan man bevege plattformen rundt og kartlegge området.

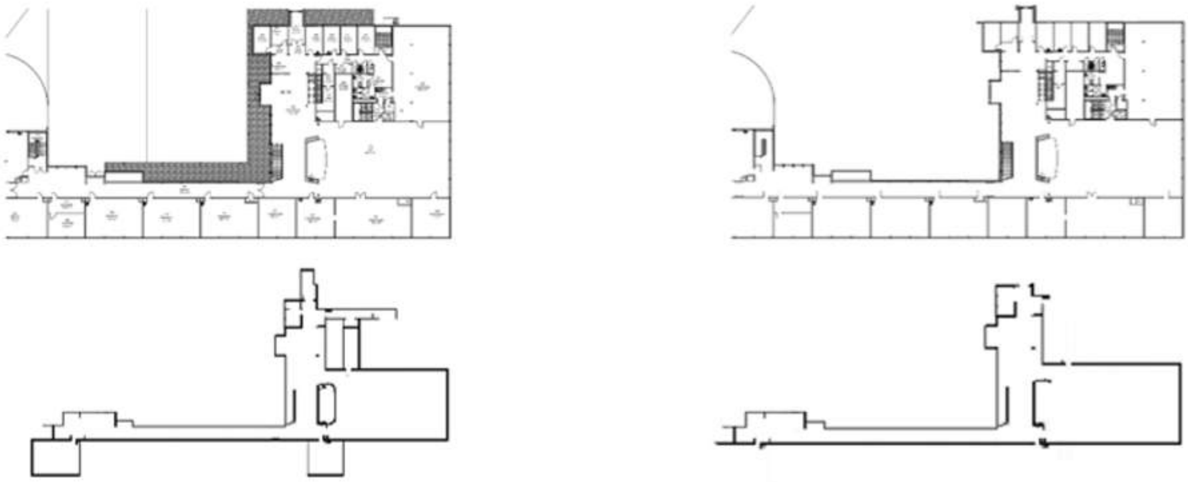


Figur 3-5: Skjermdump fra visualiseringsverktøyet Rviz. Grønn linje viser hvor sensoren har beveget seg. Mørkegrått representerer usett område. Lysegrått representerer observert område uten hindringer. De sorte linjene representerer observert område med hindring. Til sammen danner dette et kart over området som plattformen har beveget seg i.

3.2 Kartkorrelasjon

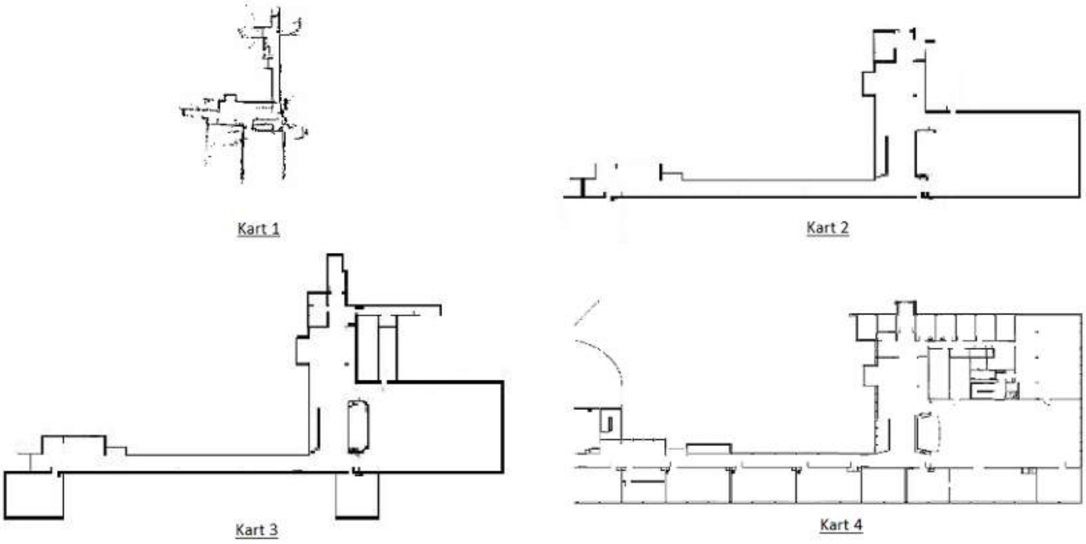
3.2.1 Utvikling av kart til kartkorrelasjon

For å kunne oppnå gode resultater i kartkorrelasjon var det viktig at de globale kartene er nøyaktige. Med dette menes at kartene så godt som mulig representerer virkeligheten ved at avstander mellom eksempelvis vegger samsvarer proporsjonalt sett. Det ble først vurdert å tegne kart egne kart, men dette ble gått bort i fra da dette potensielt ikke ville bli nøyaktig nok. Innendørs kart ble i stedet hentet fra Forsvarsbygg (FB) sine skisser over skolebygget. Disse skissene ble videre redigert i Microsoft Paint. Se vedlegg F for original kart.



Figur 3-6: Forskjellige versjoner av kart fra skolebygget. Bildet øverst til venstre er det originale bildet og har mest detaljer. Resterende bilder har blitt redigert i Paint hvor detaljer har blitt fjernet i varierende grad.

Først ble alt unødvendig fjernet fra kartene. Deretter ble det laget forskjellige versjoner som varierte i detaljnivå. Grunnen til dette var at det var ønskelig å teste om de forskjellige kartkorrelasjonsmetodenes prestasjon avhenger av hvor lite eller hvor mye detaljer de forskjellige kartene besatt.



Kart 1

Kart 2

Kart 3

Kart 4

Figur 3-7: Oversikt over globale kart brukt til testing. Kart 1 er et sensorgenerert kart i likt format som de lokale kartene. Kart 2-4 er basert på forsvarsbygg sitt kart som vist i Figur 3-6.

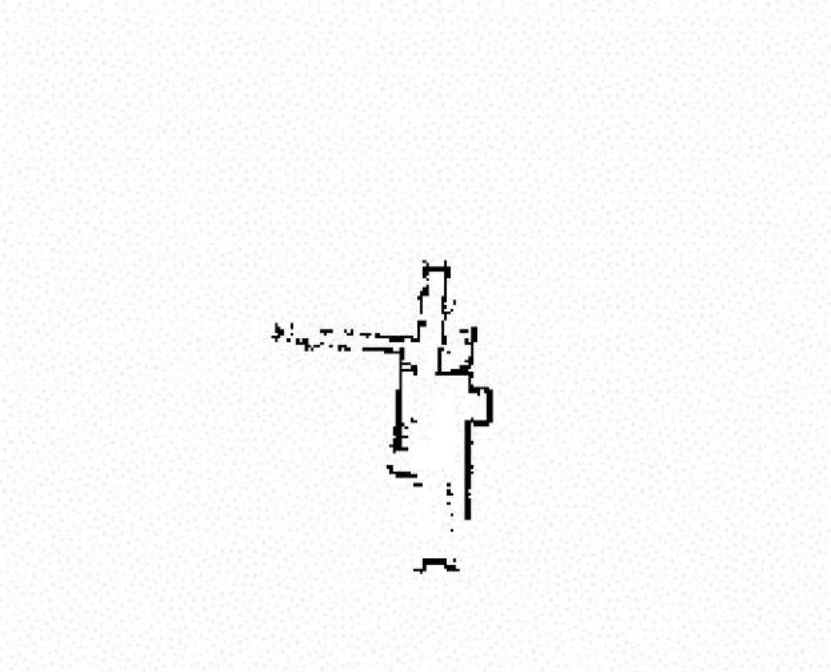
Figur 3-7 viser kartene som ble benyttet til testing av kartkorrelasjonsmetodene. Kart 1 er et bilde av et kart som ble generert av HECTOR_SLAM. Kart 2,3, og 4 er kart over skolebygget med forskjellig detaljnivå.

3.2.2 Utvikling av Kildekode

Metodene som brukes for kartkorrelasjon i dette prosjektet er hentet fra OpenCV (Open Source Computer Vision Library). OpenCV er et stort open-source bibliotek for «computer vision»- og maskinlæringsprogramvare u.d.). OpenCV gir en felles infrastruktur for mange forskjellige bildeanalysealgoritmer, deriblant algoritmene som benyttes i denne oppgaven. Det benyttes to forskjellige bildeanalysealgoritmer til kartkorrelasjon. Template Matching som sammenligner et utsnitt direkte med alle områder på et bilde og ORB som sammenligner kjennetegn mellom utsnittet og bildet. Se vedlegg D for kildekodene.

For å gjennomføre kartkorrelasjonen ble det laget et shell-script som styrer prosessen. Et shell-script er en fil som inneholder flere kommandoer slik at de kan kjøres automatisk, i stedet for at hver enkelt kommando må skrives i terminalvinduet separat (GeeksforGeeks, 2023). Dette shell-scriptet styrer kun prosessen med å hente ut det lokale kartet og kjøre kartkorrelasjonsalgoritmene. Prosessene som aktiverer lidarsensoren og SLAM er pågående under hele prosessen og startes derfor i forkant. Bakgrunnen for shell-script løsningen er at distribusjonen av ROS som brukes i denne oppgaven er tilpasset Python 2, men kartkorrelasjonskodene kan kun kjøres på Python 3. Dette gjør at kartkorrelasjonskodene ikke kan samhandle direkte med ROS. Konsekvensen av dette var at det ble laget et shell-script istedenfor for en løsning med et master Python-program som kunne styrt hele prosessen.

Det lokale kartet blir laget når HECTOR_SLAM kjører. Det ble laget et script for å hente ut det lokale kartet. Chat-GPT ble brukt som et verktøy under utvikling av dette scriptet. Se vedlegg C for ytterligere informasjon om hvordan det ble brukt. Scriptet abonnerer på topic'et «/map» og bruker dataen, som kommer i form av matriser, til å lage og lagre det lokale kartet i form av en PNG fil. Scriptet er satt til å kun kjøre en gang, og vil dermed bare produsere et kart for hver gang den kjøres. Et eksempel på et slikt kart er vist i Figur 3-8. Det lokale kartet blir lagret i en egen mappe markert med tidspunkt.



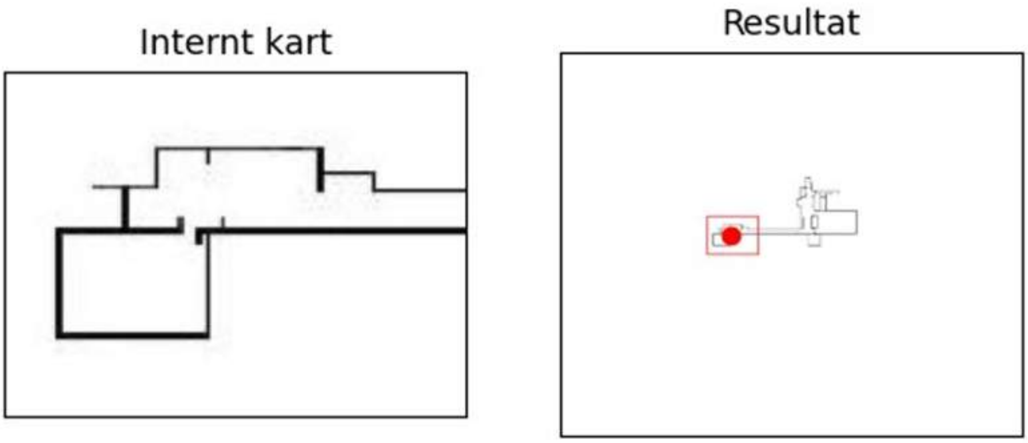
Figur 3-8: Viser et sensorgenerert lokalt kart.

Både Template matching og ORB scriptene henter inn informasjonen de trenger på samme måte. Informasjonsbehovet er et globalt kart og det lokale kartet som skal sammenlignes mot hverandre. Det er fire kart som er laget. Kartet som brukes spesifiseres i selve scriptet til metodene. For å hente inn det lokale kartet er det en litt lengre prosess. Som med det globale kartet er det ikke mulig å spesifisere hvilket av de lokale kartene som skal brukes, ettersom det blir laget like før Template Matching og ORB scriptene kjøres. Scriptene må derfor inn i en overordnet mappe som inneholder alle mappene hvor de lokale kartene lagres, og hente ut det lokale kartet fra den nyeste mappen. Dette blir gjort ved å sortere den overordnede mappen slik at den nyste mappen kommer først, for å hente ut filen som ender med «.png» fra mappen. Når dette er gjort har begge metodene det de trenger for å sammenligne kartene og gi resultater.

Posisjonene funnet i dette prosjektet vil være startposisjonen til Plattformen ved hver test istedenfor posisjonen den er i da testen blir gjort. Dette bunner først og fremst ut i at det lokale kartet som blir produsert alltid har senter som midten av det lokale kartet. Dersom Template matching fungerer vil metoden finne startposisjonen til plattformen. For ORB er ikke dette noe som påvirker resultatene den kommer med. Den andre utfordringen var unøyaktigheten av posisjonsdata fra ROS-topic/SLAM med manglende treghetsnavigering, samt orienteringen til plattformen. Dersom man skulle funnet sluttposisjon måtte man benyttet seg av den unøyaktige posisjonsdataen og dermed fått et resultat som ikke ville vært nøyaktig. I tillegg vet ikke plattformen egen orientering, og selv om posisjonsdata hadde vært nøyaktig ville ikke systemet kunne regne ut hvilken retning den hadde forflyttet seg i det globale kartet. Denne oppgaven ser på kartkorrelasjon som metode og krever derfor at man har nøyaktig posisjon til plattformen for å få reproduserbare resultater. Ved å benytte seg av startposisjon vil man ha en nøyaktig posisjon, samtidig som orienteringen på det lokale kartet kan kontrolleres.

Template Matching

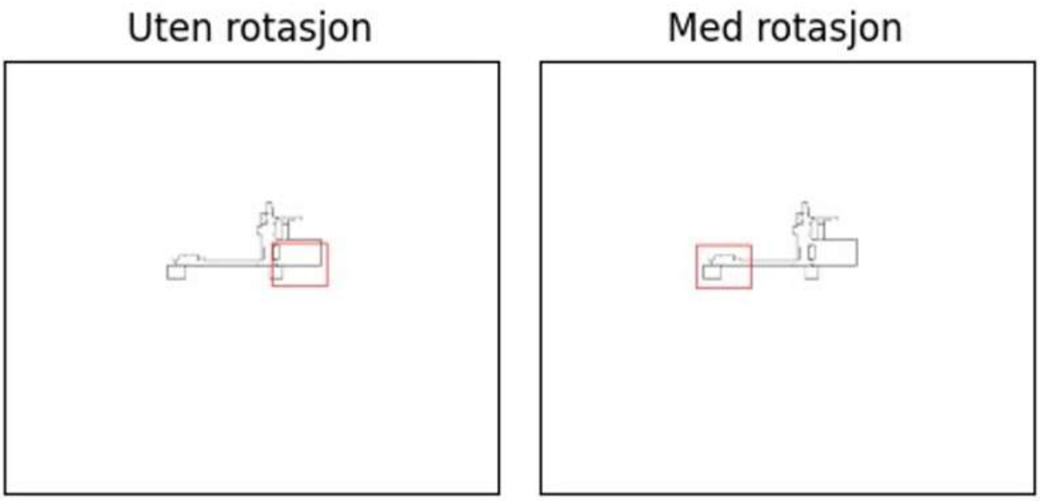
Template Matching er en relativ enkel metode som sammenligner et bilde med et mindre utsnitt. Koden i denne oppgaven er utviklet basert på OpenCV sin eksempelkode og tilpasset til bruk i denne oppgaven (OpenCV, u.d.). Metoden sjekker hver piksel i et bilde og et område rundt, tilsvarende utsnittet bildet sammenlignes mot. For å finne pikselen med høyest korrelasjonskoeffisient, blir bildematrisen som blir gitt av Template Matching algoritmen sortert i synkende rekkefølge av koden. Deretter henter koden ut punktet som representerer øverste venstre hjørne av et område som er lik størrelsen på det lokale kartet. Pikselen med best resultat befinner seg i midten. Startposisjonen til plattformen er i senter av det lokale kartet. Den tilsvarende posisjonen i det globale kartet blir dermed midten av det området som passer best på det globale kartet.



Figur 3-9: Viser resultat fra innledende testing av Template Matching. Her er det lokale kartet et utsnitt av det globale kartet til høyre, under resultat. Resultatet til høyre viser hvor det lokale kartet passer og plattformens posisjon.

Rotasjon og speiling

Template matching er ikke laget for å ta hensyn til om det er en forskjell på orientering fra det lokale kartet og det globale kartet. Ved forskjell i rotasjon vil ikke algoritmen finne en god match og resultatet kan bli feil, som vist i Figur 3-1 nedenfor. Det lokale kartet er orientert ut fra startretning til plattformen og vil derfor variere for hver gjennomføring. Det var derfor behov for å legge til en funksjon som kan rotere bildet og dermed tillate algoritmen å sjekke for en mulig match ved ulike vinkler. Rotasjonsfunksjonen er open-source og er hentet fra GitHub (prayat-py, 2021). Rotasjonen av det lokale kartet skjer ved hjelp av en rotasjonsmatrise. Rotasjonskoden ligger i filsystemet som et eget Python-script. Template Matching koden importerer rotasjonsfunksjonen fra den nevnte rotasjon-koden.



Figur 3-10: Viser eksempel av Template Matching uten og med rotasjonsfunksjon. Det interne kartet brukt er det samme som i figur 3-10, men har blitt rotert. Til venstre er et resultat uten funksjon for rotasjon hvor resultatet er feil. Til høyre er et resultat med funksjon for rotasjon, som finner rett posisjon.

Ved å sjekke etter mulige matcher ved flere forskjellig vinkler vil antall ganger Template matching funksjonen kalles øke. Som beskrevet over vil dette føre til at man kan detektere matcher selv med varierende orientering. Samtidig vil det føre til at prosesseringstiden øker betraktelig. Sjekker man for matcher ved hver 10. grad vil man øke antall iterasjoner av Template matching funksjonen til 36 ganger, og sjekker man hver grad vil den kjøre 360 ganger. Ved å teste utførelsestiden av koden man se at den passerer ett sekund ved ca. 10 iterasjoner av funksjonen. Dersom man skal sjekke for hver grad, vil koden bruke rundt 37 sekunder. Koden ble satt til å teste for hver 10. grad slik at den ikke brukte for lang tid å kjøre, men samtidig fikk testet for flere ulike vinkler.

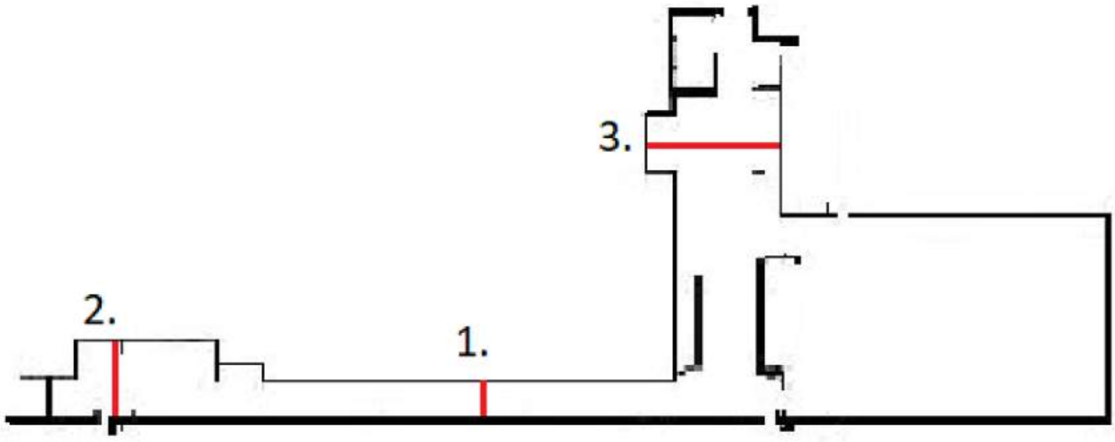
Det lokale kartet blir speilvendt i forhold til de tre siste globale kartene. Det kan føre til utfordringer for Template Matching-algoritmen. Det er derfor lagt inn en funksjon som speilvender et bilde. Med denne funksjonen er slam-bildet i rett format når Template matching skal gjennomføres.

Skalering

Til forskjell fra ORB-algoritmen tar ikke Template Matching hensyn til forskjeller i skalering mellom de to bildene som sammenlignes. Dersom det er forskjell i størrelsen på objekter/områder i det lokale kartet og de globale kartene vil det oppstå feilmålinger, med mindre de globale kartene har blitt skalert riktig. Dette er på grunn av virkemåten til

«Template matching». I template matching blir kartene sammenlignet ved å plassere lokale kart på de globale kartene. Dersom ikke objekter (vegger, rom, osv.) er lik i størrelse vil algoritmen ha problemer med å kunne finne et område som er tilnærmet likt i begge kartene.

For å løse dette problemet ble det lagt til en skalering i Python-scriptet som skalerer ned de globale kartene. Skaleringsfaktoren ble bestemt på grunnlag av en oppmåling av tre ulike steder i området som de globale kartene representerer, som vist på Figur 3-11. Basert på disse målingene er det mulig å finne ut hvor stor avstand hver piksel representerer, og deretter finne skaleringsfaktoren med å sammenligne det med parameterne til det lokale kartet.



Figur 3-11: Viser de tre målepunktene brukt i skaleringsprosessen på kart 2 (se Figur 3-7). De samme målepunktene er brukt i alle kart.

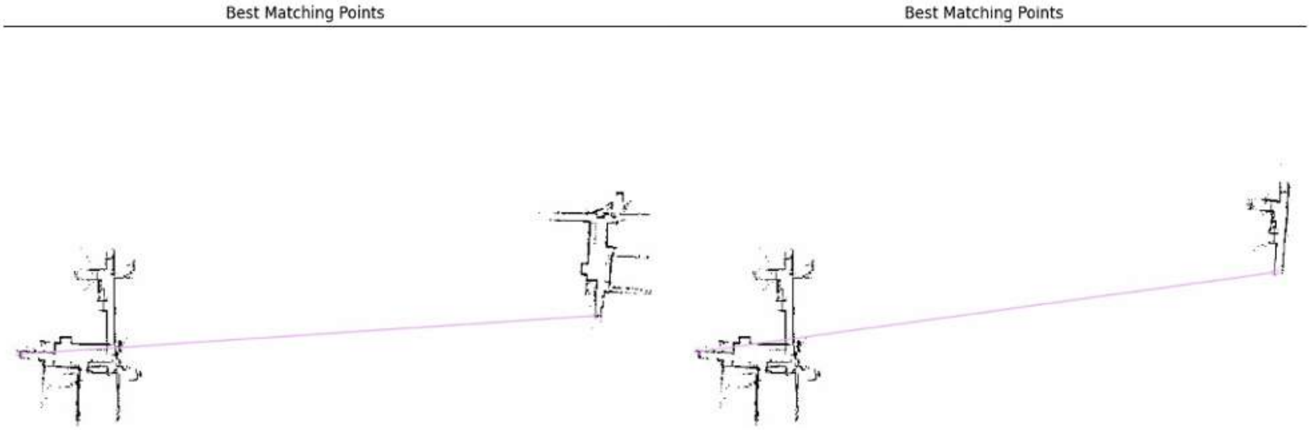
Målingene ble gjort med lasermåler for høy presisjon, og gjennomført på markerte steder i Figur 3-11. Deretter ble antall piksler på samme lokasjoner telt. Ved å dele avstand på antall piksler finner vi lengde per piksel. Etter denne prosessen er gjort for hver lokasjon, ble gjennomsnittet av avstanden per piksel regnet ut. Dette ble gjort for å få redusere effekten av eventuelle små feil i målingene. I tillegg til eventuelle avvikelser i forskjellige områder i kartene, fra prosessen med å forenkle kartene beskrevet i delkapittel 3.2.1. For å finne skaleringsfaktoren ble gjennomsnittet av avstand per piksel satt opp mot avstand per piksel i de lokale kartene. Utregningene av skaleringsfaktoren til de ulike kartene vises i kapittel 4.2.

Én piksel i de SLAM-generert kartene er satt til å representere et område med lengde og bredde på 30 cm, altså 30x30 cm^2. Det er derfor dette som benyttes i skaleringsutregningene. Lidarsensoren og HECTOR-SLAM er ikke feilfritt, og det kan derfor være mindre avvik fra dette på de lokale kartene. Det kan være enkelte områder hvor pikslene representer litt under 30x30 cm^2, og andre hvor de kan representere litt over 30x30 cm^2. Tidspunkt, sted og grad av avvik kan være vanskelig å forutse og dermed benyttes 30x30 cm^2 som beste alternativ for beregning av skaleringskonstanten til de ulike kartene.

ORB

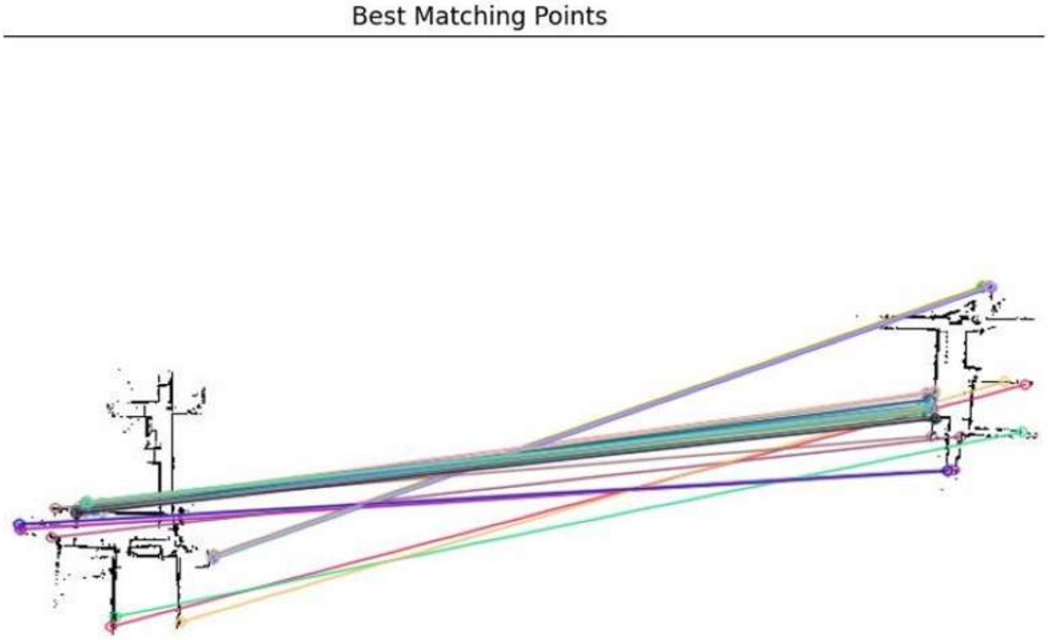
ORB algoritmen fungerer ved å finne kjennetegn i bildene som skal sammenlignes og matche disse opp mot hverandre. ORB kommer som en rekke funksjoner i OpenCV biblioteket. Som beskrevet i kapittel 2 har ORB en måte å håndtere forskjeller i skalering mellom bildene. De bruker en metode kalt pyramideskalering, som beskrevet i kapittel 2.3.1. Pyramideskaleringen kan justeres etter behov og er blitt stilt til å ha 10 nivåer som endres med skaleringsfaktoren 1,1. Det gjør at det er mange forskjellige skaleringer som blir tatt hensyn til uten at de endrer seg mye mellom hvert nivå i pyramiden.

Resultatene fra matchingen i ORB blir lagret i en tabell og sortert i stigende rekkefølge etter distanse. I Template Matchingen er det sammenligningen med høyest korrelasjonsverdi som forteller posisjonen til plattformen. For ORB viser det seg at dette ikke er tilfellet.



Figur 3-12: Figuren viser et globalt kart sammenlignet med to forskjellige interne kart. Det er dratt en linje mellom det som er tilsvarende piksler i kartene ifølge ORB. Venstre side viser en match mellom to piksler som tilsvarer hverandre. Høyre side viser en match mellom to piksler som ikke tilsvarer hverandre

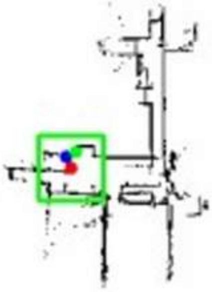
Matchen med kortest distanse gir ikke nødvendigvis rett resultat. Figur 3-12 viser to eksempler på den beste matchen fra testing av ORB-algoritmen, med to ulike lokale kart. Venstre del av figuren viser et eksempel der den beste matchen har et minimalt avvik, og dermed ga et godt resultat. Høyre del av figuren viser en match hvor avviket er stort. Erfaringen fra innledende testing av ORB-algoritmen var at den beste matchen ved de fleste tilfeller ga feil resultat, og kunne dermed ikke brukes for å gi posisjon.



Figur 3-13: Viser de 40 beste matchene mellom det som skal være tilsvarende piksler på det lokale (til høyre) og det globale (til venstre) kartet. Det interne kartet er orientert 90 grader mot klokken i forhold til det globale kartet. De fleste matchene ser ut til å være rett, men med enkelte unntak.

Ettersom den matchen med kortest distanse ikke nødvendigvis gir en høy grad av korrelasjon, må man finne resultatet på en annen måte. Som vist i Figur 3-13 er det enkelte matcher hvor kjennetegn ikke samsvarer i de to kartene, men de fleste matchene er mellom samsvarende kjennetegn. Istedenfor å se på en enkelt match kan man se på en større samling av kjennetegn fra de matchene med kortest distanse. For å finne en samling av matcher blir de 40 beste matchene valgt, deretter sjekkes et område på 40x40 piksler, og antallet kjennetegn fra matchene i området blir telt. I likhet med virkemåten til Template matching blir området på 40x40 piksler dratt over det globale kartet, her med en steglengde på 10 piksler, slik at hele kartet blir sjekket.

I området med flest matcher blir gjennomsnittet av koordinatene til kjennetegnene regnet ut på det globale kartet. Det tilsvarende koordinaten på det lokale kartet settes som gjennomsnittskoordinaten til de tilsvarende kjennetegnene i det lokale kartet.



Figur 3-14: Viser resultat fra ORB matching. Den grønne firkanten representer området med flest matcher. Den røde prikken er senter av området og den grønne prikken representer gjennomsnittskoordinaten av kjennetegn i området.

Koden finner altså koordinatene til tilsvarende punkter i det globale kartet og i det lokale kartet, men finner ikke startposisjonen til plattformen. Ettersom det er startposisjonen som skal være endelig resultat må koordinaten til matchene gjøres om til startposisjon. For å finne startposisjonen regner man ut avstand og retning til midtpunkt i det lokale karte. Deretter tar man det globale kartet, orienter det i samme retning som det lokale kartet. Startposisjonen i det globale kartet vil da være i samme retning og avstand som er funnet i det lokale kartet, fra det tilsvarende kjennetegn i det globale kartet.

Oppsummert består dette prosjektet av en lidarsensor som er montert på den landbaserte testplattformen. Sensoren sender informasjon til en datamaskin som genererer et lokalt kart ved hjelp av HECTOR_SLAM. Dette kartet blir omformet til et bilde som blir sammenlignet med et utvalg av globale kart. Resultatene fra sammenligningen blir til slutt brukt for å estimere posisjon.

4 Resultater

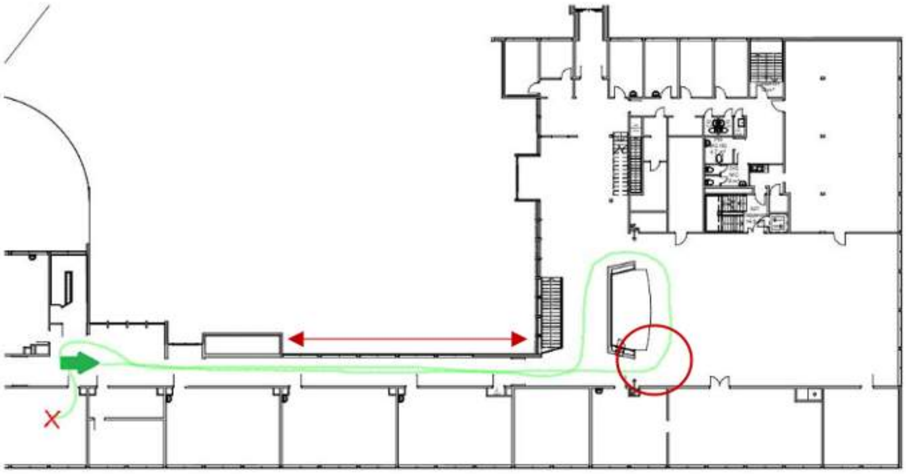
Forrige kapittel har beskrevet plattformen, tilhørende programvare, utvikling av kart, samt utviklingen av kildekoden for kartkorrelasjonen. Dette kapittelet skal presentere resultater fra testene av kartkorrelasjonsalgoritmene, samt innledende testing og kalibrering. Rekkefølgen er kronologisk og vil starte med å vise til innledende testing av SLAM algoritmen kartleggingsevne. Deretter vil det bli presentert utregninger for skaleringsfaktorer. Siste delen av kapittelet vil gjennomgå testing av kartkorrelasjonsmetodenes evne til å estimere posisjon.

4.1 Generering av lokalt kart

Det ble gjort innledende tester av kartlegging med HECTOR_SLAM for å få en oversikt over hva som påvirker hvor god kartleggingsprosessen blir. I tillegg var det ønskelig å lære om hvilke hensyn man kan ta for å forebygge typiske SLAM problemer. Fremgangsmåten for testingen startet med å kjøre oppstartfilene til Lidaren og HECTOR_SLAM. Deretter ble plattformen fysisk trillet tilfeldig rundt. Plattformen ble også enkelte ganger styrt med en Ps4 kontroller. Underveis I kartleggingen ble kartet i Rviz studert.

Kartleggingskvaliteten var sensitiv for høye hastigheter og krappe svingninger. Dersom man trillet plattformen for fort, eller endret retning hurtig ble kartene ofte ufullstendig og inneholdt flere feil enn kart som ble produsert med lavere hastigheter. Et eksempel på en slik kartleggingsfeil vises i Figur 4-2. Dette skyldes mest sannsynlig at lokaliseringsprosessen ikke klarer å holde følge med store endringer på kort tid. Det ble erfart at når lokaliseringen feilet, risikerte enkelte deler av kartet å bli forvrengt i forhold til virkeligheten. I slike tilfeller, ville det være vanskeligere for kartkorrelasjonsmetodene å sammenligne det lokale og det globale kartet. I tillegg ble det gått vekk ifra å benytte ps4 kontroller til å styre plattformen da man ikke klarte å regulere hastigheten godt nok til å unngå store feil i genererte kart.

HECTOR_SLAM slet med å produsere kart som representerte virkeligheten på en god måte i områder uten distinkte objekter som skilte seg ut. Figurene nedenfor viser et eksempel på et av forsøkene. Ruten som ble tatt er skissert i kartet i Figur 4-1, og kartet som ble visualisert i Rviz i Figur 4-2.



Figur 4-1: Kart over skolebygg skissert med rute. Start er illustrert med grønn pil, og slutt med rødt kryss. Rød strek viser til avlang gang, og sirkel viser til nordlige hovedinngang til Aula

Figur 4-2: Visualisering hentet ut fra Rviz. Startposisjonen til plattformen er illustrert med grønn pil, og slutt med rødt kryss. De hvite strekene og sirkelen representerer avlang gang og nordlig hovedinngang til aula.

Lokaliseringen feilet da plattformen ble trillet rett frem i den avlange gangen. Det så ut til at HECTOR_SLAM ikke klarte å identifisere bevegelse i dette partiet, noe som gjorde at kartleggingsprosessen stoppet midlertidig opp frem til man nådde enden av gangen.

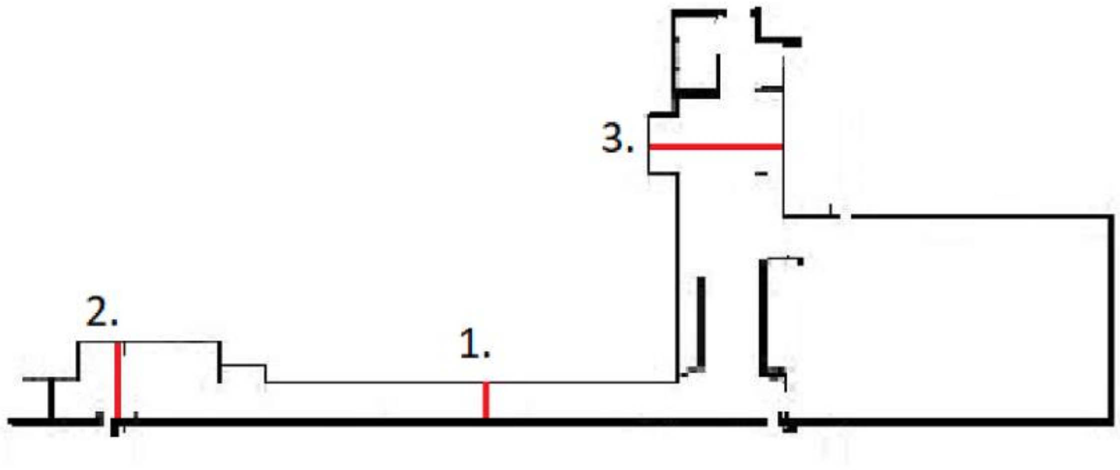
Dette kan man se i Figur 4-2: Visualisering hentet ut fra Rviz. **Startposisjonen til plattformen er illustrert med grønn pil, og slutt med rødt kryss. De hvite strekene og sirkelen representerer avlang gang og nordlig hovedinngang til aula.**, der gangen ved 1. passering ikke er proporsjonal med resten av kartet. På vei tilbake ble det gjort et forsøk på å hjelpe algoritmen ved å trille plattformen gjennom gangen på en måte som kan minnes om en sinuskurve. Dette kan man se i Figur 4-2, i 2. passering av gang. Kartleggingen av gangen ble litt nærmere et faktisk kart, men lokaliseringsprosessen slet likevel med å registrere bevegelse til tider. Man kan se at gangen i 2. passering ble lengre, men er fortsatt ikke lang nok til å samsvare med gangen i Figur 4-1.

Et annet problem som oppsto, fant sted ved inngangen til Aulaen, som er markert med sirkel i Figur 4-1 og 4-2 ovenfor. Planen var å trille plattformen rundt scenen i aulaen. Da plattformen ble rotert for å svinge, ser det ut til at lokaliseringsprosessen ikke klarte å registrere endringen i retning. Dette ga et tydelig utslag i kartet, som man kan se ved at all kartlegging etter dette punktet hadde forskjøvet seg med omtrentlig 45 grader sammenlignet med før dette punktet. Denne lokaliseringsfeilen har antageligvis grunnlag i at endring i retning skjedde for brått, samtidig som kartleggingsprosessen hadde for få holdepunkter til å matche point cloud data.

4.2 Kalibrering av Template Matching

Utregning av skaleringsfaktor er tidligere beskrevet i delkapittel 3.2.3. Template Matching er en algoritme som sammenligner et utsnitt direkte med et område på det globale kartet. Algoritmen er derfor sensitiv for forskjell i skalering mellom de to kartene som sammenlignes. Derfor må det beregnes en skaleringsfaktor for det globale kartet slik at begge kartene har lik skala. Beregningene av skaleringsfaktoren er basert på tre målinger i ulike deler av kartet som vist i Figur 4-3.

Se Figur 4-4 for oversikt over de globale kartene (kart 1-4). Kart 2 og 3 har like dimensjoner og vil dermed ha lik skaleringsverdi. Skaleringsverdi for kart 4 må regnes ut separat. Kart 1 er et SLAM-generert kart over hele området vi tester i og har ikke behov for skalering, ettersom det har likt format og skala.



Figur 4-3: Samme figur som i delkapittel 3.2.2 (Figur 3-11). Viser de tre målepunktene brukt i skaleringsprosessen på kart 2. De samme målepunktene er brukt i alle kart som må skaleres (kart 2-4) som vist i Figur 3-7 og 4-4.

Regner ut skalering for kart 2 og 3:

$$\text{Punkt 1: } \frac{avstand}{antall\ piksler} = \frac{228{,}3\ cm}{10\ piksler} = 22{,}83\,\frac{cm}{piksel}$$

$$\text{Punkt 2: } \frac{avstand}{antall\ piksler} = \frac{500\ cm}{22\ piksler} = 22{,}73\,\frac{cm}{piksel}$$

$$\text{Punkt 3: } \frac{avstand}{antall\ piksler} = \frac{886{,}15\ cm}{42\ piksler} = 21{,}1\,\frac{cm}{piksel}$$

$$\text{Gjennomsnitt: } \frac{Punkt\ 1 + Punkt\ 2 + Puntk\ 3}{3} = \frac{22{,}83 + 22{,}73 + 21{,}1}{3} = 22{,}22\,\frac{cm}{piksel}$$

$$\text{Skaleringsfaktor: } \frac{\frac{cm}{piksel}\ \ i\ globalt\ kart}{\frac{cm}{piksel}\ i\ SLAM{-}kart\ kart} = \frac{22{,}22\,\frac{cm}{piksel}}{30\,\frac{cm}{piksel}} = 0{,}74$$

Samme prosess blir gjentatt for kart 4:

$$\text{Punkt 1: } \frac{228{,}3\ cm}{35\ piksler} = 6{,}52\,\frac{cm}{piksel}$$

$$\text{Punkt 2: } \frac{500\ cm}{77\ piksler} = 6{,}49\,\frac{cm}{piksel}$$

$$\text{Punkt 3: } \frac{886{,}15\ cm}{138\ piksler} = 6{,}42\,\frac{cm}{piksel}$$

$$\text{Gjennomsnitt: } \frac{6{,}52 + 6{,}49 + 6{,}42}{3} = 6{,}48\,\frac{cm}{piksel}$$

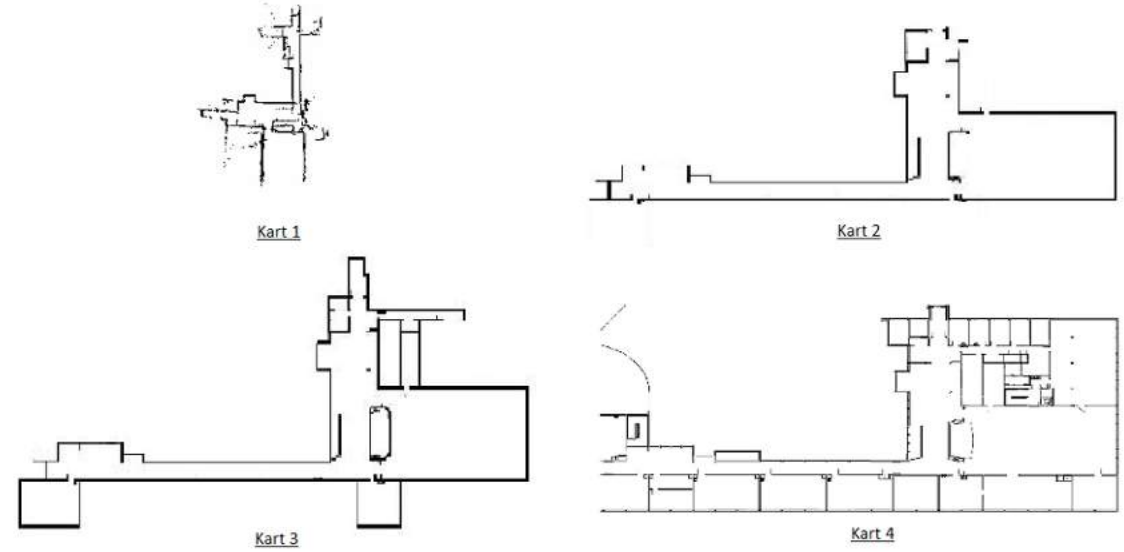
$$\text{Skaleringsfaktor: } \frac{6{,}48\,\frac{cm}{piksel}}{30\,\frac{cm}{piksel}} = 0{,}22$$

I kart 2, 3 og 4 (Figur 4-4) er området representert skalert forskjellig fra de lokale kartene. Dette delkapittelet har tatt for seg utregningene av skaleringsfaktorene som skal brukes når de lokale kartene sammenlignes med de globale kartene 2, 3 og 4 ved bruk av Template Matching.

4.3 Posisjonering

Når lokale kart kan genereres og skaleringsfaktoren mellom det globale og lokale kartet er kjent, skal nå metodene for kartkorrelasjon bli sammenlignet med hensyn til deres evne til å finne rett kartutsnitt og dermed rett posisjon til plattformen i det globale kartet. Når det lokale SLAM-kartet prøves å matches med det globale kartet vil startposisjonen bli benyttet som faktisk posisjon til plattformen. Som nevnt i delkapittel 3.2 er det av praktiske grunner lettere å bestemme posisjonsavviket basert på startposisjonen.

Denne posisjonen har blitt fysisk målt opp ved bruk av håndholdt lasermåler for å finne koordinaten til startposisjon i alle globale kart. Når metodene har plassert det lokale kartet i det globale kartet, kan ett estimat av startposisjonen regnes ut. Avstanden mellom estimatet av startposisjonen og den faktiske posisjonen er avviket som blir brukt til å sammenligne metodene opp mot målene som er satt.


Kart 1

Kart 2

Kart 3

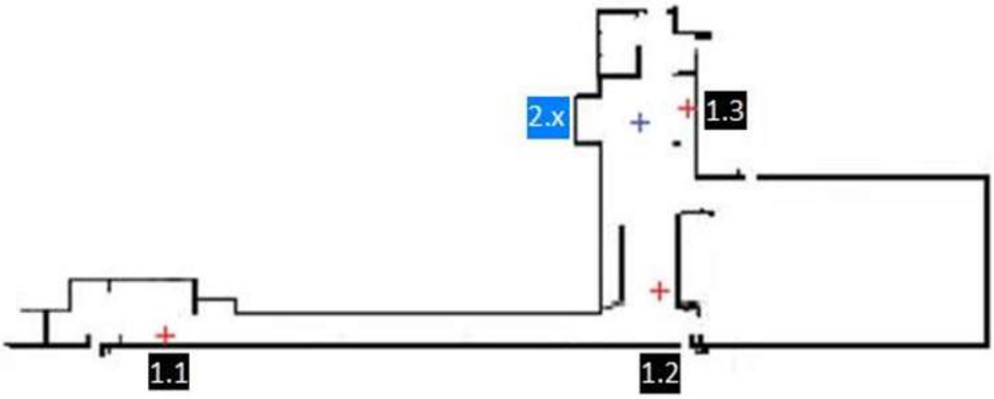
Kart 4

Figur 4-4: Oversikt over globale kart med navn. Kart 1 er et SLAM generert kart. Kart 2,3,4 er forenklede versjoner av FB skisser over skolebygget. Figuren er den samme som Figur 3-6 i kapittel 3.2.1

For hver enkelt test, ble det generert ett lokalt kart. Dette kartet ble først sammenlignet med de fire globale kartene ved bruk av Template matching. Deretter ble det samme gjort ved bruk av ORB. Det ble gjennomført to runder med testing av kartkorrelasjonsalgoritmene, der hver av rundene inneholdt tre tester.

I alle testene ble «rover_test.launch» benyttet som launch-fil for HECTOR_SLAM. Denne launch-filen er identisk med den tidligere bacheloroppgavens launch-fil, med unntak av oppløsningen til kartet i Rviz som ble justert opp fra 100x100 til 400x400 piksler for å få bedre posisjonsoppløsning. Plattformen ble plassert parallelt og med kjent avstand til nærmeste vegg ved oppstart av HECTOR_SLAM. I tillegg ble den trillet rundt i tilstrekkelig lav fart til at SLAM prosessen skulle kartlegge området med minimale feil. Sensoren RPLidar A1 var plassert i forkant av plattformen, i en høyde på 13,5 cm over bakken. I tillegg ble det sørget for at ingen personer eller andre bevegelige objekter var til stede. For hver av de seks testene som ble gjennomført ble det innhentet et lokalt kart ved å kjøre «map_image.py». Hensikten med å teste metodene opp mot flere globale kart var å se om ulik mengde detaljer vil påvirke posisjonsestimatene for hver enkelt test.

Posisjonene funnet i resultatene representerer et område på varierende størrelse. I kart 1 representerer en piksel et område på $30x30\ cm^2$, i kart 2 og 3 er området $22{,}2x22{,}2\ cm^2$ og i kart 4 et område på $6{,}48x6{,}48\ cm^2$(basert på målinger i kapittel 4.2). Ettersom startposisjon er satt til en spesifikk piksel vil de nevnte områdene være begrensingen på hvor presise resultatene kan være. Avviket oppgitt senere i kapittelet er avstanden mellom funnet posisjon og posisjonen plattformen faktisk startet i. Dette avviket kan ikke bli mer presist enn størrelsen til pikslene de forskjellige kartene representer.



Figur 4-5: Startposisjoner til de forskjellige testene. Rødt kryss representerer startposisjonene til testene i første runde. Blått kryss representerer startposisjonen til alle testene i andre runde

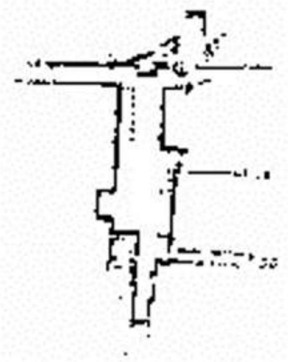
4.3.1 Første runde med testing

Hensikten med den første runden av testing var å få et generelt inntrykk av ytelsen til kartkorrelasjonsmetodene. Plattformen ble derfor plassert på tre forskjellige steder i skolebygget, og området rundt ble i varierende grad kartlagt. Deretter ble kartkorrelasjonsprogrammene kjørt. Bildet av visualiseringen til Rviz ble sammenlignet med 4 forskjellige globale kart. Se vedlegg E for alle bildene av de lokale kartene. Figur 4-5 viser startposisjonen for hver enkelt test.

Test 1.1: Under trapp ved resepsjon

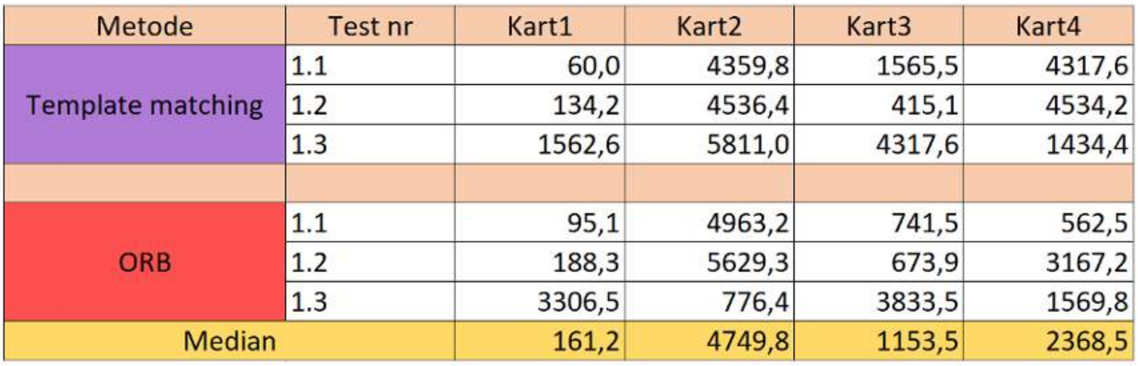
Test 1.2: Ved flagg

Test 1.3: Ved datalab



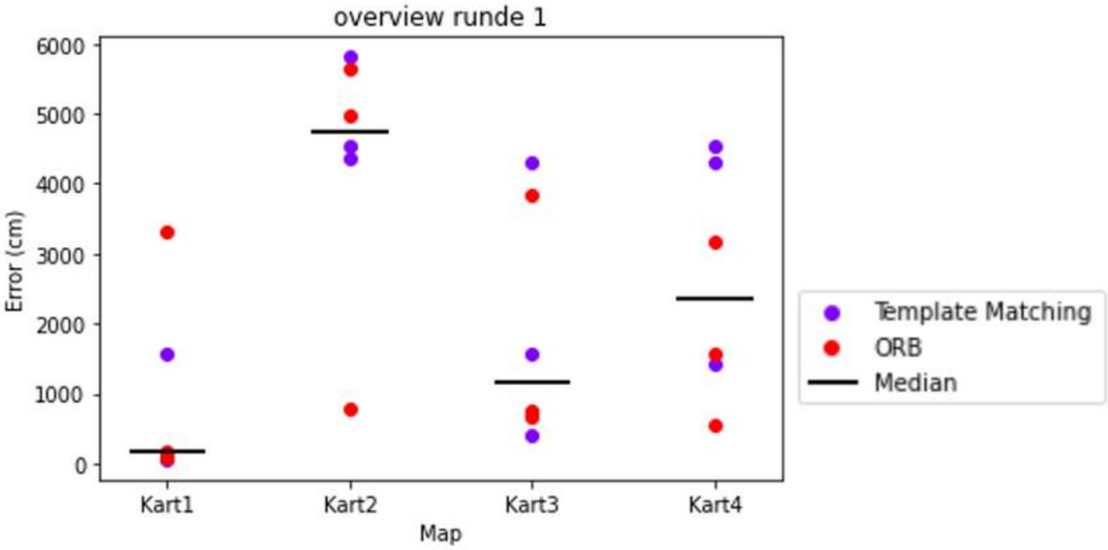
Figur 4-6: Viser det lokale kartet generert i test 1.2 som ble hentet ut av Map_image-scriptet.

Tabell 4-1: Viser avvik mellom estimert og faktisk posisjon i centimeter(cm)



Metode	Test nr	Kart1	Kart2	Kart3	Kart4
Template matching	1.1	60,0	4359,8	1565,5	4317,6
	1.2	134,2	4536,4	415,1	4534,2
	1.3	1562,6	5811,0	4317,6	1434,4
ORB	1.1	95,1	4963,2	741,5	562,5
	1.2	188,3	5629,3	673,9	3167,2
	1.3	3306,5	776,4	3833,5	1569,8
Median		161,2	4749,8	1153,5	2368,5

Tabell 4-2: Avvik fra testrunde 1 med de forskjellige referansekartene. Hver farge representerer en metode. Sort strek viser medianverdiene.

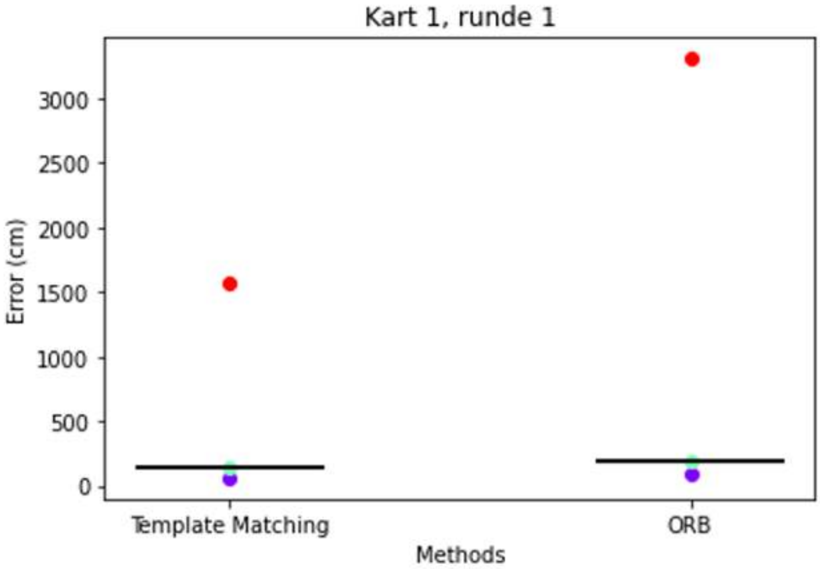


I Tabell 4-1 og 4-2 kan man se at kartkorrelasjon med kart 2,3,4 gir generelt store avvik. Medianverdiene tilsvarer over 10 meter feil mellom faktisk posisjon og utregnet posisjon. Kart 1 har et lavere medianavvik enn resten, men det finnes likevel en høy verdi for begge metodene her også. De høye verdiene hører til test 1.3.

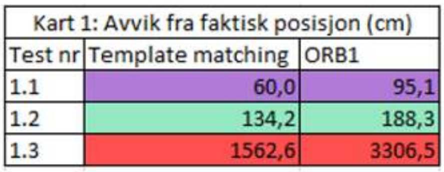
Template matching har store avstander mellom avvik, med den laveste verdien på 60cm og den høyeste på 5811 cm. Dersom man skiller mellom enkelte kart, er samlingene noe bedre, men det er likevel store variasjoner i de forskjellige testene. Template matching presterer best ved bruk av kart 1.

ORB har også store forskjeller i avvik, men presterer bedre enn Template matching ved bruk av kart 2,3 og 4. ORB gjør det også best ved bruk av kart 1, men har likevel større avvik enn Template matching.

Tabell 4-3: Avvik fra testrunde 1 med globalt kart 1. Hver farge representerer en enkelt test. Sort strek viser medianverdi.

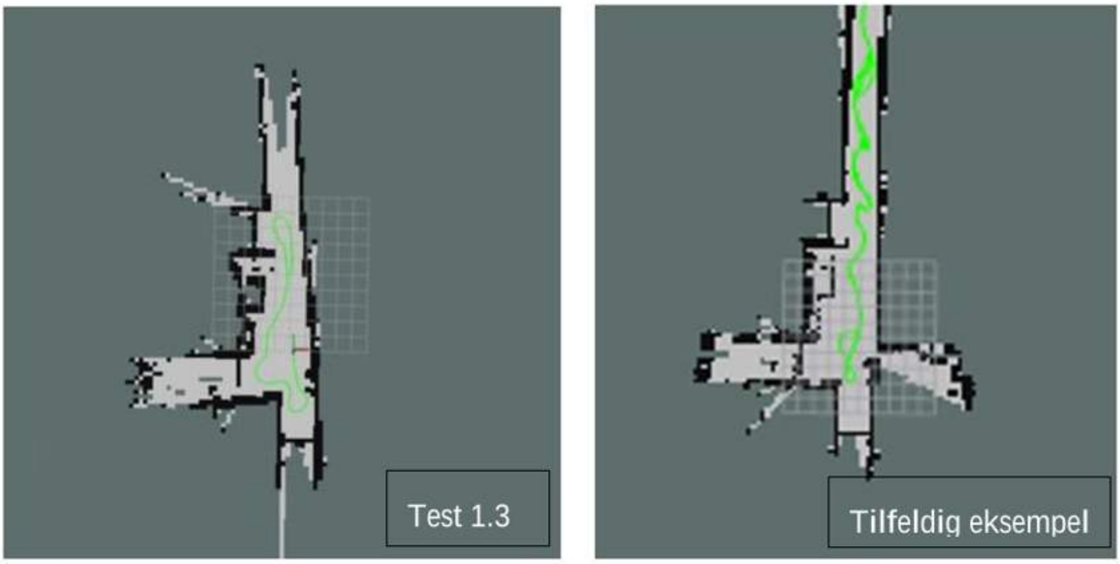


Tabell 4-4. Tabell viser avvik i centimeter. (cm) fargekodene samsvarer med fargene på datapunktene i Tabell 4-3



Kart 1: Avvik fra faktisk posisjon (cm)		
Test nr	Template matching	ORB1
1.1	60,0	95,1
1.2	134,2	188,3
1.3	1562,6	3306,5

Som Tabell 4-3 og 4-4 viser, kan man se at begge metodene gir tilnærmet likt avvik for test 1.1. Det samme gjelder test 1.2. Begge metodene presterer dårlig i test 1.3. Dette kan skyldes at kartet generert av HECTOR_SLAM under denne testen ble skjevt i visualiseringen.



Figur 4-7: Viser to skjermbilder fra Rviz. Bildet til venstre viser test 1.3. Bildet til høyre viser et eksempel fra innledende testing der veggene ikke ble skjeve.

I Figur 4-7 kan man se at veggene i kartet til test 1.3 er mer ujevne enn i kartet til høyre.

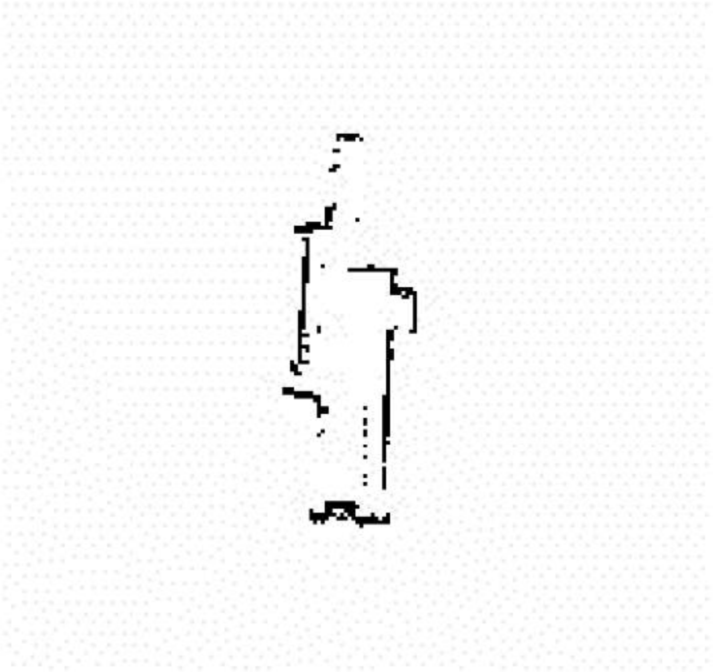
4.3.2 Andre runde med testing

I runde 2 ble det gjennomført totalt tre tester. Alle testene hadde samme startpunkt, men kartleggingen varierte. Det ble benyttet håndholdt lasermåler for å måle opp avstander. Startposisjon for testene vises i Figur 4-55.

Test 2.1: Plattformen ble flyttet 10 meter frem

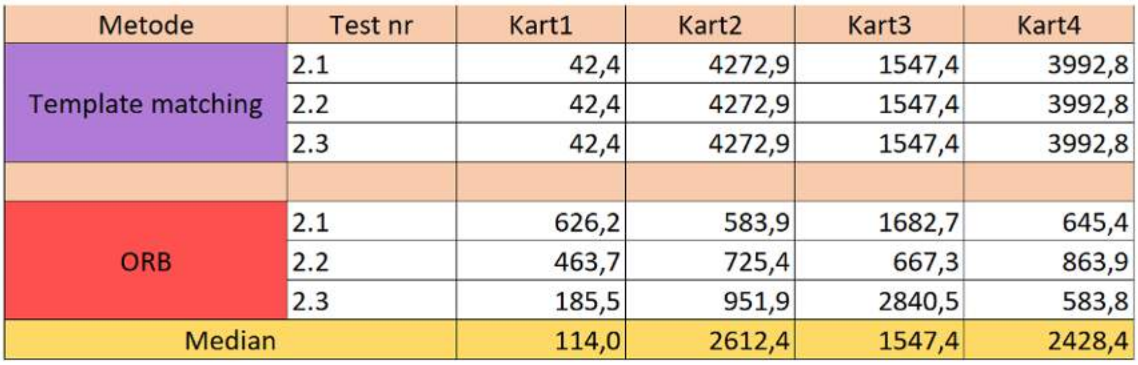
Test 2.2: Plattformen ble flyttet 2 meter frem, og deretter rotert 90 grader mot høyre. Dette ble gjentatt frem til plattformen befant seg på samme sted som den startet.

Test 2.3: Plattformen roterte rundt egen akse



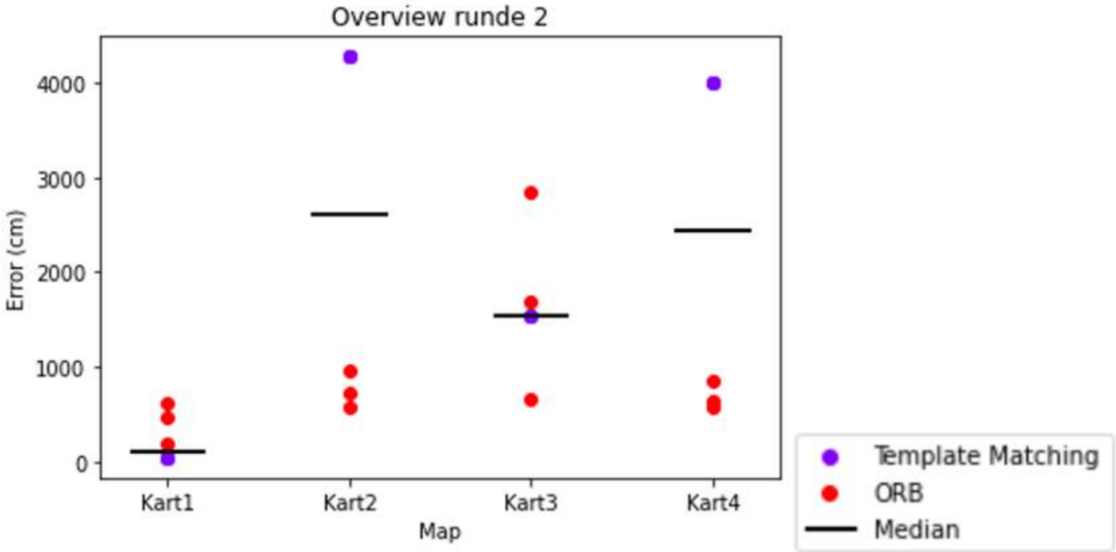
Figur 4-88: Viser det lokale kartet generert i test 2.2 som ble hentet ut av Map_image-scriptet.

Tabell 4-5: Viser avvik mellom estimert og faktisk posisjon i centimeter (cm).



Metode	Test nr	Kart1	Kart2	Kart3	Kart4
Template matching	2.1	42,4	4272,9	1547,4	3992,8
	2.2	42,4	4272,9	1547,4	3992,8
	2.3	42,4	4272,9	1547,4	3992,8
ORB	2.1	626,2	583,9	1682,7	645,4
	2.2	463,7	725,4	667,3	863,9
	2.3	185,5	951,9	2840,5	583,8
Median		114,0	2612,4	1547,4	2428,4

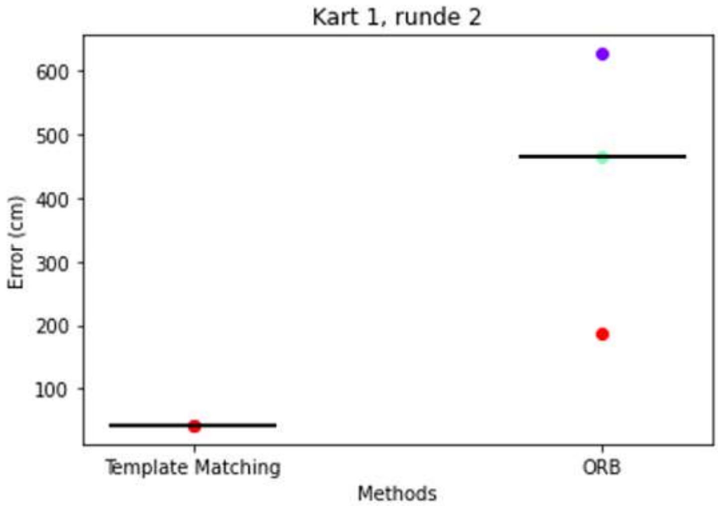
Tabell 4-6: Avvik fra testrunde to med de forskjellige globale kartene. Hver farge representerer en metode



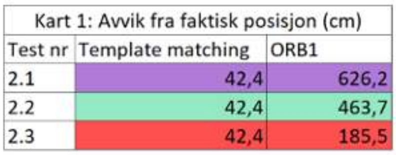
For Template matching viser resultatene fra sammenligningen til kart 2, 3 og 4 at metoden ikke fungerer like godt når det lokale kartet sammenlignes med globale kart som ikke er generert av SLAM. I tillegg har kartene har en økende grad av detaljnivå. Avvikene varierer fra ca. 10 til 40 meter og viser at Template matching ikke klarer å finne rett posisjon, og heller ikke område på kartene hvor startposisjonen er.

ORB-algoritmen viser varierende resultater i sammenligningen mellom det lokale kartet og kart 2, 3 og 4. ORB har på kart 2 og 4, avvik som varierer mellom fem og ti meter.

Tabell 4-7: Avvik fra testrunde 1 med kart 1



Tabell 4-8: Tabell viser avvik i centimeter (cm)



Kart 1: Avvik fra faktisk posisjon (cm)		
Test nr	Template matching	ORB1
2.1	42,4	626,2
2.2	42,4	463,7
2.3	42,4	185,5

Det er i rett område av kartet, men er et stykke unna startposisjon. I kart 3 er det større avvik, hvor to av tre tester setter posisjon over 15 meter feil. De varierende resultatene ser ikke ut til å ha noen sammenheng med varierende grad av kartlegging under testene.

Som Tabell 4-7 og 4-8 viser, har testene på kart 1 lavere avvik for både Template Matching og ORB-algoritmen. Alle avvik var under 7 meter og medianen for begge testene på kart 1 ligger på litt over en meter. Template Matching metoden fant samme posisjon i alle testene på kart 1, hvor avviket var ca. 42 cm, selv med varierende grad av kartlegging. Avviket er også lite sett i lys av at hver piksel representerer et område på 30x30 cm.

For ORB som metode resulterer sammenligningen med varierende avvik. Avvikene viser til at ORB-algoritmen klarer å finne riktig område av kart 1, men er ikke konsistent nok til å fastslå nøyaktig posisjon i hver test. I en av testene er avviket lite, med en avstand på ca. 185 cm fra startposisjon. De resterende resultatene er mindre presise, hvor avvikene varierer fra 4-7 meter (se Tabell 4-1 og 4-5).

Oppsummert, for begge rundene med testing viser Template Matching stabile resultater på alle testene for kart 1, med unntak av én test hvor resultatet er vidt forskjellig fra resten. Når detaljnivå på kartene øker evner ikke Template Matching å finne riktig posisjon i kartet og avvikene blir derfor store på nesten samtlige tester på kart 2, 3 og 4.

For ORB er også resultatene fra testingen på kart 1 stabile, men metoden har høyere avvik på de gjeldende testene. I tillegg er resultatet fra 1.3 ved bruk av kart1 veldig forskjellig fra resten. Det er samme test som også ga et avvikende resultat i Template Matching. På kart 2,3 og 4 klarer ikke ORB å levere stabile resultater. De variere i stor grad, hvor det i enkelte tester finner rett område uten å være presis nok og andre tester hvor det er veldig store avvik. Fullstendige resultater ligger i vedlegg E.

5 Drøfting

5.1 Resultater

I denne oppgaven har det blitt implementert to forskjellige metoder for å korrelere et lokalt kart og et globalt kart. Videre ble det gjennomført tester for å sjekke i hvor stor grad estimert posisjon samsvarer med faktisk posisjon. Kravene som ble satt til metodene var følgende:

- Metodene må gi stabile resultater med avvik ≤ 2 meter, for å kunne være et pålitelig alternativ for GPS og tilføye systemet redundans.
- Metodene må kunne håndtere varierende grad av detaljer på de globale kartene.

I testrunde 1 ga Template matching varierende resultater mellom ca. en halv til 15 meter. Dette kan tyde på at metoden ikke kan fungere som et pålitelig posisjoneringsverktøy skulle GPS-signalet til et autonomt system falle bort. Først og fremst ser det ut til at avviket fra testene ved bruk av de forskjellige kartene ikke har noen tydelig sammenheng. For en enkelt test kan man se at avviket varierer med flere titalls meter. I tillegg er avvikene generelt veldig store. Dette tyder også på at metoden ikke er pålitelig nok til å være et funksjonelt alternativ til GPS ennå.

Ved bruk av kart 1 presterer Template matching likevel noe bedre. Dette kan indikere på at metoden evner å gi gode estimater under visse omstendigheter. Test 1.1 og 1.2 oppnår avvik på 60cm og 134,2cm ved bruk av kart 1. Dette kan sees på som tilstrekkelig presisjon dersom metodene kun skal fungere ved midlertidig bortfall av GPS. Man kan derimot argumentere for at de gode resultatene betyr lite, sett i lys av test 1.3. Denne testen hadde et avvik på 15,63 meter som er helt klart utilstrekkelig presisjon. Som vist i delkapittel 4.3.1 var i tillegg det generte SLAM-kartet i denne testen skjevt. Dette kan vise til at Template Matching ikke er robust nok til å håndtere selv små ulikheter mellom kartene som sammenlignes. Dette argumentet blir også styrket av at kart 1 er det eneste globale kartet som gir lave avvik for Template matching. Kart 1 er generert av SLAM og deler derfor mest likheter med de lokale kartene. Sett under ett er det rimelig å anta at Template matching ikke er allsidig nok til å håndtere varierende grad av detaljer.

I runde 2 ga Template matching stabile resultater, noe som kan tyde på at dette er en god metode for å beregne posisjon. Som vist i resultatdelen er avvikene for Template matching helt like for alle testene ved bruk av de forskjellige kartene. I tillegg er startposisjonen for alle testene i runde 2 lik. Dette kan tyde på at metoden er konsistent. Likevel er feilen stor ved bruk av kart 2, 3 og 4. Ved bruk av kart 2 er feilen mellom faktisk og utregnet posisjon 42,73 meter, noe som tilsier at metoden ikke er god nok allikevel. Videre kan man fra resultatene se at metoden gir et avvik på 42,4 cm ved bruk av kart 1. Dette kan regnes som et relativt godt estimat.

ORB gir resultater med varierende grad av avvik i begge rundene med tester. Dette kan tyde på at metoden ikke er presis nok til å være et alternativ til GPS. Resultatene fra sammenligningen med kart 1 viser avvik som varier fra ca. en meter til over seks meter, med ett unntak. På test 1.3 var avviket ca. 33 meter. Selv om metoden ser ut til å finne rett område i kartet, klarer den ikke å finne nøyaktig posisjon til plattformen i alle testene. Runde 1 har to tester der avviket er under to meter, som er presist. Runde 2 har mer varierende resultater. Dette kan skyldes måten resultatene blir hentet ut fra ORB-algoritmen. Å bruke gjennomsnittskoordinaten til de beste matchene i området med flest gode matcher, er ikke nødvendigvis den beste måten til å finne rett posisjon. Som vist i Figur 3-12 er ikke matchen med kortest distanse alltid riktig. Dersom en eller flere av matchene som gjennomsnittet blir regnet ut ifra ikke er mellom tilsvarende koordinater i de to kartene, kan resultatet bli forskjøvet i feil retning. Avviket vil i dette tilfellet kunne bli større. Dersom bruk av gjennomsnittskoordinaten fører til større avvik, kan det være at ORB-algoritmen er kapabel til mer presis posisjonering ved å finne en bedre måte å hente inn resultater. Oppsummert gir resultatene i denne oppgaven inntrykk av at ORB ikke er presis nok. Samtidig er det mulig at presisjonen kan forbedres ved å hente ut resultatene på en alternativ måte.

Grunnen til at noen av matchene er mellom koordinater som ikke samsvarer, kan skyldes kartenes enkle oppbygging. Kartene er sort/hvitt-bilder hvor kun vegger blir vist. Ettersom kartene har få detaljer kan det være enkelte piksler som ikke tilsvarer hverandre, men som likevel blir en match ettersom området rundt ligner. Et eksempel på dette kan man se på høyre side av Figur 3-12 som viser en match hvor pikslene ikke samsvarer. Begge disse pikslene er helt i enden av sine respektive kart og området rundt pikselen kan minne om hverandre.

Tendensen til varierende avvik fortsetter i testene på kart 2, 3 og 4. Dette kan tyde på at ORB-algoritmen ikke klarer å håndtere kart med et annet format, og med flere detaljer. Enkelte resultater kan tyde på at metoden klarer å finne rett område i kartet. Samtidig er avvikene hvor metoden finner et område nært startposisjonen, generelt større enn resultatene fra sammenligningen med kart 1. Det er også flere store avvik hvor algoritmen ikke finner rett område, der avvikene er større enn 10 meter. I testing med kart 2, 3 og 4 ser store avvik ut til å være mer fremtredende, i motsetning til testene med kart 1. Dette støtter påstanden om at ORB ikke klarer å håndtere kart av ulikt format, samt varierende grad av detaljnivå. Grunnen til dette kan, som nevnt over, være at gjennomsnittskoordinaten gir et større avvik. Denne effekten kan ha blitt forsterket av at kartene har et annet format og flere detaljer. Med flere mulige kjennetegn kan det også være en økt sannsynlighet for at matcher ikke er mellom tilsvarende koordinater i de to kartene. Dette kan flytte området med flest matcher til feil sted på kartet og dermed forårsake større avvik.

I testene på kart 1 viser resultatene større avvik i runde 2, noe som kan tyde på at når en mindre del av området er kartlagt blir resultatene dårligere. På kart 2, 3 og 4 viser derimot ingen tendenser til at det påvirker resultatene i særlig stor grad. Der varierer resultatene ca. like mye i runde 1 og 2, men det heller mer mot at runde 2 gir bedre resultater. Dette kan tilsi at ORB ikke blir påvirket av testene i runde 2 hvor området bli kartlagt i varierende grad, og klarer å finne nok kjennetegn.

Kartene fra Forsvarsbygg (FB) sine plansjer ble redigert i Microsoft Paint, og har derfor en viss grad av inkonsistente trekk sett opp mot hverandre. Dette kan ha påvirket hvor god kartkorrelasjonen har blitt. Å få kartene til å holde samme standard, eksempelvis at veggene blir representert med like mange piksler, er vanskelig å få til ved bruk av datamus. På den andre siden kan inkonsistente kart gi svar på hvor robuste metodene er. Men som resultatene viser, sliter begge metodene med å gi nøyaktig posisjon ved bruk av disse kartene.

I tillegg kan oppløsningen til de sensorgenerte kartene bidratt til å begrense prestasjonen til kartkorrelasjonsmetodene for FB-kartene. Dette er fordi pikslene i de lokale kartene som ble generert av HECTOR_SLAM representerer et område på 30x30cm. Et objekt trenger nødvendigvis ikke å være så stort for å bli registrert av lidarsensoren. Eksempler på slike objekter er inventar som flagg og miniatyrmodeller av skip befinner seg i gangene på skolen. Det er derfor mulig at mindre objekter som ble detektert av lidarsensoren representerer et større område i selve kartet. Dersom man hadde justert ned størrelsen hver piksel representer, og dermed økt oppløsningen, kan man tenke seg at presisjonen hadde økt. Det er likevel ikke sikkert at dette hadde hatt noen særlig effekt, ettersom disse mindre objektene ikke er til stede i kartene fra FB. Uavhengig av oppløsningen, vil disse objektene fortsatt være uoverensstemmelser for kartkorrelasjonsmetodene. Oppsummert kan det se ut til at metodene er for sårbare mot variasjoner som ikke ligger i begge kartene. En korrelasjonsmetode bør derimot være robust med hensyn til støy og variasjoner, også hvis man tenker på f.eks. mennesker og andre dynamiske elementer som kan dukke opp i praksis.

Skaleringsfaktoren brukt for Template Matching-algoritmen kan være en feilkilde. For å finne skaleringsfaktoren ble avstander målt opp på flere steder. Ettersom kartene er redigert for å gjøre de mer like formatet til det SLAM-genererte lokale kartet, vil det kunne oppstått avvik ved enkelte lokasjoner, slik at avstandene i ulike deler av kartene ikke samsvarer helt. Det ble derfor vurdert at et gjennomsnitt for avstand per piksel var det beste alternativet for å sikre en skalering som passet best for hele kartet. Template Matching er en metode hvor det er viktig at skaleringsfaktoren er riktig ettersom den plasserer det lokale kartet direkte på det globale kartet. Dersom det er feil i skaleringen kan det føre til at man ikke får gode matcher, og resultatet blir et «tilfeldig» sted. Dette kan være én grunn til de store avvikene på testene med kart 2, 3 og 4.

Det har blitt gjort få tester, noe som kan bety at resultatenes validitet ikke er god nok til å trekke avsluttende konklusjoner. «Validitet, eller gyldighet, betyr i hvilken grad man ut fra resultatene av et forsøk eller en studie kan trekke gyldige slutninger om det man har satt seg som formål å undersøke.» (Dahlum, 2021). For estimering av posisjon ble det gjennomført to runder med testing, der hver av rundene inneholdt tre tester. I tillegg ble hver metode testet opp mot fire forskjellige globale kart. Dette tilsvarer totalt 24 svar for hver av metodene, noe som kan anses som en del data. Samtidig er diversiteten i testene store og sett i lys av dette er mengden data begrenset. Videre er det mange faktorer som kan ha påvirket resultatene. Det er mulig at dersom det hadde blitt gjort flere tester, kunne man enklere sett trender som i større grad avdekket disse eventuelle faktorene. Dessuten finnes det kjente faktorer som kan ha påvirket resultatene i større grad enn det som kommer frem i resultatene. Som tidligere beskrevet i både teori, og erfart i innledende testing, har SLAM en del utfordringer. Med flere resultater å vise til, kunne man i større grad hatt en formening om hvor mye dette har påvirket dataene.

Selv om det ikke ble gjort mange tester av posisjoneringsevnen, er det rimelig å anta at innledende forsøk med SLAM økte validiteten til resultatene for estimering av posisjonering. Som beskrevet i resultatdelen ble det gjennomført en del tester for å avdekke hva som påvirket SLAM prosessen. Faktorer som påvirket prosessen negativt, ble definert og hensyntatt i videre testing. Det er mulig at bevisstheten rundt disse faktorene gjorde at resultatene i større grad belyser metodenes evne til å beregne korrekt posisjon. Altså, resultatene ble mer relevante for det oppgaven faktisk ønsker å undersøke.

Oppsummert, resultatene viser generelt store variasjoner og høye avvik, noe som tyder på at metodene ikke fungerer per nå. Det har blitt diskutert forskjellige faktorer og feilkilder som kan ha påvirket dette i begge retninger. Enkelte resultater viser likevel til at kartkorrelasjon som konsept har potensiale. Dette skal nå diskuteres videre

5.2 Konseptuelt

Resultatene fra testene med kart 1 kan tyde på at kartkorrelasjon bør testes videre og utvikles som konsept. Kartkorrelasjon som konsept kan være gunstig å implementere som et posisjoneringsverktøy i det maritime domenet i den hensikt å redusere avhengigheten til GPS. Dette trenger heller ikke å være avgrenset kun til ubemannede fartøy, men kan være et alternativ også for bemannede fartøy. Dersom GPS ikke skulle fungere på grunn av f.eks. jamming, kan en mulighet være å benytte seg av radar og sammenligne radarbildet med eksisterende sjøkart. Det finnes allerede sjøkart med «overlay» av radarbilde, men de justerer seg ikke til hverandre. Å sammenligne sjøkart med radarbilde kan mulig gi et mer nøyaktig posisjonsestimat i kystnære områder enn treghetsnavigasjon hvor avviket i posisjon kan bygge seg opp over tid. Selv om kartkorrelasjon kan ha potensiale som posisjoneringsverktøy, finnes det allerede lignende systemer som er uavhengig av GPS. Ubåter som ikke har GPS-signal ved navigering under vann bruker treghetsnavigasjon, og korrelerer dybdeinformasjon fra sensorer med dybdekonturen på kart av havbunnen for å korrigere posisjon (Homeier, 2021). Denne prosessen kan ligne på metoden brukt i denne oppgaven. For bemannede fartøy kan personellet om bord bruke andre metoder som f.eks. krysspeiling til objekter for å estimere posisjon. Likevel vil innføring av konseptet øke redundansen for posisjoneringssystemene til enhetene som implementer det. I tillegg kan konseptet være økonomisk ettersom de fleste fartøy allerede har sensorer som benyttes, som for eksempel radar. Det vil også potensielt kunne frigjøre personell til andre oppgaver på bemannede fartøy.

For at et konsept som kartkorrelasjon skal utvikles videre, automatiseres og implementeres på enheter må det undersøkes hvilke type sensorer, og eventuelt andre metoder som kan egne seg til dette formålet. I denne oppgaven testes to metoder i kontrollerte forhold på land. Sensoren som brukes på plattformen har liten rekkevidde og vil ikke egne seg til bruk i det maritime domenet. Resultatene i sammenligningen med kart 1, som er det sensorsgenerte kartet, kan tilsi at metodene har potensiale til å fungere på et kart som er laget i helt likt format av samme type sensor. I et bachelorprosjekt fra 2022 diskuteres en mulighet til å bygge et globalt kart, ved å sette sammen de lokale kartene fra forskjellige områder, som alle er generert basert på sensordata (Algrøy & Opdahl, 2022). Om dette var tilfellet, vil man kunne sammenligne de lokale kartene med et globalt kart som er laget på samme måte og med samme type sensordata. Med metodene brukt i denne oppgaven kan dette gi bedre posisjonering enn å sammenligne de lokale kartene med for eksempel sjøkart. Det er allikevel foreløpig uklart hvor vidt dynamiske elementer ville forstyrret denne prosessen, til tross for bedre utgangspunkt for matching av kartene.

6 Konklusjon med anbefaling

Denne oppgaven har tatt for seg kartkorrelasjon som et alternativ til GPS-fri posisjonering. Kartkorrelasjon er en metode for å sammenligne kart, i den hensikt å estimere posisjon i et globalt kart basert på lokal posisjonering. To forskjellige kartkorrelasjonsmetoder har blitt testet på en landbasert plattform. Metodene, Template Matching og ORB, sammenligner et sensorbasert autogenerert kart av omgivelsene rundt plattformen med et globalt kart av området den befinner seg i.

Testresultatene indikerer at kartkorrelasjonsmetodene, i den form det ble testet her, ikke er stabile og presise nok til å være et alternativ for GPS-fri posisjonering. Sammenlignet med det sensorgenererte kartet, kart 1, oppnår Template Matching kravet til presisjon i fem av seks tester. I de resterende testene med andre kart er avvikene store. ORB algoritmen har varierende resultater i alle kartene, og oppfyller ikke kravet til presisjon ved unntak av noen få tester. I likhet med Template Matching, oppnår ORB-metoden de mest presise resultatene ved bruk av det sensorgenererte kartet.

I denne oppgaven oppfyller ikke metodene kravene som ble satt, til å kunne være et alternativ for GPS-fri posisjonering. Konseptet bak kartkorrelasjon kan allikevel ansees som en lovende metode for en alternativ posisjonering i utsatte perioder. Matching av f.eks. radarkart med navigasjonskart bør undersøkes videre som en mulig posisjoneringsmetode.

Gjennom arbeidet med dette oppgaven har det blitt identifisert noen forbedringer og muligheter for videreutvikling av konseptet kartkorrelasjon. Skal prosjektet videreutvikles anbefales det å forbedre kartleggingen. Dette kan blant annet bli gjort ved å justere oppløsningen til de lokale kartene som genereres. Det kan også være en mulighet å implementere flere sensorer som kan bidra inn til kartleggingen. Siden metodene fikk de beste resultatene i sammenligningen med kart 1 kan det være også være interessant å se på muligheten rundt å bygge globale kart ved å sette sammen de lokale kartene som diskutert i delkapittel 5.2 og i et tidligere bachelorprosjekt fra 2022 (Algrøy & Opdahl, 2022).

En annen anbefaling til videre forskning er å undersøke hvordan dette konseptet kan overføres til det maritime domenet. Dette innebærer blant annet å finne ut av hvilke sensorer og kart som skal benyttes. I tillegg til å se på andre alternative metoder for kartkorrelasjon. Her kan kartkorrelasjon ved bruk av maskinlæring være et spennende alternativ å undersøke.

Referanser

Academic Accelerator. (2023, November 16). Image Moment. Hentet november 15, 2023 fra Academic Accelerator: https://academic-accelerator.com/encyclopedia/image-moment

Algrøy, S., & Opdahl, M. (2022). GPS uavhengig posisjonering. Utforsking av datamaskinens synsevne i bildeanalyse. Bergen: FHS, Sjøkrigsskolen. Hentet oktober 10, 2023

Calonder, M., Lepetit, V., Strecha, C., & Fua, P. (2010). BRIEF: Binary Robust Independent Elementary Features. Lausanne: European Conference on Computer Vision (ECCV). Hentet november 13, 2023 fra https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Features

Dahlum, S. (2021, mars 9). validitet. Hentet desember 2, 2023 fra Store norske leksikon: https://snl.no/validitet?gclid=Cj0KCQiAyKurBhD5ARIsALamXaEjrREKI-IJbnRhsxndsP3Hey6C98-fPctKINPde7KI8nM_DCqSaocaAshNEALw_wcB

DeepAI. (u.d.). Feature Extraction. Hentet november 27, 2023 fra DeepAI: https://deepai.org/machine-learning-glossary-and-terms/feature-extraction

FFI. (u.d.). Navigasjon: Hvordan kan vi beregne nøyaktig posisjon og retning hvis vi ikke har GPS? Hentet November 22, 2023 fra FFI: https://www.ffi.no/forskning/prosjekter/navigasjon

Flaaten, S., & Tande, M. (2018). Testplattform for autonome systemer. Bergen: Sjøkrigsskolen. Hentet August 15, 2023 fra https://fhs.brage.unit.no/fhs-xmlui/handle/11250/2574767

Forsvarsdepartementet. (2023, Oktober 6). Statsbudsjettet 2024 | Moderne førerløse mineryddere til Sjøforsvaret. Hentet Oktober 22, 2023 fra regjeringen.no: https://www.regjeringen.no/no/aktuelt/moderne-forerlose-mineryddere-til-sjoforsvaret/id2998910/

GeeksforGeeks. (2023, Februar 20). Feature Matching using Brute Force in OpenCV. Hentet november 16, 2023 fra GeeksforGeeks: https://www.geeksforgeeks.org/feature-matching-using-brute-force-in-opencv/

GeeksforGeeks. (2023, mai 2). Introduction to Linux Shell and Shell Scripting. Hentet November 20, 2023 fra GeeksforGeeks: https://www.geeksforgeeks.org/introduction-linux-shell-shell-scripting/

Hassel, D., Moen, S., & Ohma, K. (2019). Utvikling av autonomt system på landbasert plattform - GPS-uavhengig navigering i trange miljøer. Bergen: FHS, Sjøkrigsskolen. Hentet August 15, 2023 fra https://fhs.brage.unit.no/fhs-xmlui/handle/11250/2673205

Holtet, J. A. (2022, April 1). Lidar. Hentet november 2, 2023 fra snl.no: https://snl.no/lidar

Homeier, M. G. (2021, oktober). Navigate by the Stars—From Beneath the Waves. Hentet november 30, 2023 fra U.S Naval Institute: https://www.usni.org/magazines/proceedings/2021/october/navigate-stars-beneath-waves

Hykkerud, E. (2023, september 19). Mangedobling av GPS-jamming mot Norge. Hentet november 22, 2023 fra NRK: https://www.nrk.no/tromsogfinnmark/mangedobling-av-gps-jamming-mot-norge-1.16563000

Kjerstad, N. (2022, november 22). treghetsnavigasjon. Hentet november 30, 2023 fra Store norske leksikon: https://snl.no/treghetsnavigasjon

MathWorks. (u.d.). What Is Feature Extraction? Hentet november 27, 2023 fra MathWorks: https://se.mathworks.com/discovery/feature-extraction.html

NASA. (2021, januar 29). Active sensors. Hentet november 30, 2023 fra NASA EARTHDATA: https://www.earthdata.nasa.gov/learn/backgrounders/active-sensors

National Coordination Office for Space-Based Positioning, Navigation, and Timing. (2022, mars 3). GPS Accuracy. Hentet november 29, 2023 fra GPS: https://www.gps.gov/systems/gps/performance/accuracy/

Nogare, D. D. (2019, juli 1). An Introduction To Biological Image Processing In ImageJ, Part 1: What Is An Image? Hentet desember 1, 2023 fra Medium: https://medium.com/@damiandn/an-introduction-to-biological-image-processing-in-imagej-part-1-what-is-an-image-54fc31f3d02d

Norsk Romsenter. (2016). HVA DU BØR VITE OM SATELLITTNAVIGASJON. (M. Moen, Redaktør) Hentet november 30, 2023 fra romsenter: https://www.romsenter.no/no/Fagomraader/Satellittnavigasjon/Hva-du-boer-vite-om-satellittnavigasjon2

Open Robotics. (2018, januar 15). Master. Hentet november 13, 2023 fra Ros.org: https://wiki.ros.org/Master

Open Robotics. (2019, april 14). Packages. Hentet november 13, 2023 fra Ros.org: https://wiki.ros.org/Packages

Open Robotics. (2020, mars 25). Ubuntu install of ROS Melodic. Hentet september 25, 2023 fra ROS.org: http://wiki.ros.org/melodic/Installation/Ubuntu

Open Robotics. (2021). ROS - Robot Operating System. Hentet november 13, 2023 fra ROS: https://www.ros.org/

Open Robotics. (2022, september 20). ROS/ Consepts. Hentet november 13, 2023 fra ROS.org: https://wiki.ros.org/ROS/Concepts

OpenCV. (u.d.). Object Detection. Hentet november 14, 2023 fra OpenCV: https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be

OpenCV. (u.d.). ORB (Oriented FAST and Rotated BRIEF. Hentet november 13, 2023 fra OpenCV: https://docs.opencv.org/4.x/d1/d89/tutorial_py_orb.html

OpenCV. (u.d.). Template Matching. Hentet oktober 27, 2023 fra OpenCV: https://docs.opencv.org/3.4/d4/dc6/tutorial_py_template_matching.html

Pradhan, S. (2023, September 13). Hamming Distance between two strings. Hentet november 16, 2023 fra GeeksforGeeks: https://www.geeksforgeeks.org/hamming-distance-two-strings/

prayat-py. (2021, april 2021). Computer_vision. Hentet oktober 26, 2023 fra GitHub: https://github.com/prayat-pu/Computer_vision/blob/master/tranform_image/rotation_img.py

Ragipally, A. (2020, april 22). How images are stored on a computer? Hentet november 26, 2023 fra Medium: https://alekya3.medium.com/how-images-are-stored-in-a-computer-f364d11b4e93

RoboPeak Public Repos. (2018, august 24). rplidar_ros. Hentet september 27, 2023 fra Github: https://github.com/robopeak/rplidar_ros

Rosten, E., Porter, R., & Drummond, T. (2010). Faster and better: a machine learning approach. IEEE (Institute of Electrical and Electronics Engineers). Hentet november 13, 2023 fra https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Features

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011). ORB: an efficient alternative to SIFT or SURF. California: Proceedings of the IEEE International Conference on Computer Vision. Hentet november 13, 2023 fra https://www.researchgate.net/publication/221111151_ORB_an_efficient_alternative_to_SIFT_or_SURF

Shanghai Slamtec Co., Ltd. (2016). RPLIDAR A1. Hentet desember 2, 2023 fra generationrobots: https://www.generationrobots.com/media/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf

simsangcheol. (2023, mai 2). Pose Graph Optimization. Hentet november 26, 2023 fra Medium: https://medium.com/@sim30217/pose-graph-optimization-30ce29e4d65f

Singh, J. (2022, oktober 28). What is odometry? (T. Point, Redaktør) Hentet november 26, 2023 fra tutorialspoint: https://www.tutorialspoint.com/what-is-odometry

Slamtec Global network. (2023). RPLidar A1. Hentet november 2, 2023 fra slamtec.com: https://www.slamtec.com/en/Lidar/A1/

Studica. (2021). The ROS package. Hentet november 14, 2023 fra Studica Robotics: https://docs.wsr.studica.com/en/latest/docs/ROS/UsingROS/intro-to-ros.html

Technische Universität Darmstadt ROS Packages. (2022, juli 13). hector_slam. Hentet oktober 8, 2023 fra Github: https://github.com/tu-darmstadt-ros-pkg/hector_slam

The MathWorks, Inc. (2023). What is SLAM? Hentet november 2, 2023 fra MathWorks: https://se.mathworks.com/discovery/slam.html

thedanielhassel. (2019, desember 2). Project-LIBERTY. Hentet september 9, 2023 fra Github: https://github.com/thedanielhassel/Project-LIBERTY

Tyagi, D. (2019, januar 3). Introduction To Feature Detection And Matching. Hentet November 27, 2023 fra Medium: https://medium.com/@deepanshut041/introduction-to-feature-detection-and-matching-65e27179885d

Tyagi, D. (2019, mars 2019). Introduction to Harris Corner Detector. Hentet november 30, 2023 fra Medium: https://medium.com/@deepanshut041/introduction-to-harris-corner-detector-32a88850b3f6

Tyagi, D. (2019, Januar 1). Introduction to ORB (Oriented FAST and Rotated BRIEF). Hentet november 13, 2023 fra Medium: https://medium.com/@deepanshut041/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

Upadhyay, P. (2023, januar 4). Template matching using OpenCV in Python. Hentet oktober 24, 2023 fra GeeksforGeeks: https://www.geeksforgeeks.org/template-matching-using-opencv-in-python/

Vedlegg A: Prosedyre for å kartlegge med SLAM og utførelse av kartkorrelasjon

Dette vedlegget beskriver hvilke kommandoer man benytter i terminalvindu for å gjenskape det prosjektet. Det første man sørger for et at RPLidar A1 er koblet til en datamaskin som har lastet ned riktig programvare. Se vedlegg B for installasjon av programvare.

Åpne et terminalvindu og skriv kommandoene:

```
sudo chmod 666 /dev/ttyUSB0
```

og

```
roslaunch rplidar_ros rplidar.launch
```

Dette starter opp lidarsensoren. Åpne deretter et nytt terminalvindu og skriv inn kommandoen:

```
roslaunch hector_slam_launch rover_test.launch
```

Dette starter selve kartleggingen. I tillegg vil visualiseringsverktøyet Rviz åpnes. Her vil man se hva lidarsensoren registrerer og hva som blir kartlagt. Etter man har beveget plattformen rundt og er fornøyd med kartleggingen, kjører man shell scriptet ved å skrive kommandoen:

```
sh ~/”path”/shell_script
```

- Erstatt “path” med mappen/mappene du har valgt å legge «shell_script» i.

«shell_script» gjør tre ting. Det første som skjer, er at Python scriptet «map_image.py» blir kjørt. Dette scriptet henter ut informasjon fra topic’et «/map» og gjør det om til en png-fil. Denne png-filen blir lagt til i mappestrukturen. Deretter blir «orb.py» og «template_matching.py» kjørt. Disse scriptene er selve kartkorrelasjonsmetodene. De sammenligner kartene og legger resultatene inn i mappestrukturen. For å bestemme hvor i mappestrukturen disse tre Python-scriptene lagrer produktene sine må man gå inn i koden og endre dette etter behov.

Vedlegg B: Installering av programvare

Denne instruksen er hentet direkte fra det forrige bachelorprosjektet som omhandlet den landbaserte testplattformen (Hassel, Moen, & Ohma, 2019). Instruksen ble brukt i dette prosjektet for å komme i gang med HECTOR_SLAM. Under installering møtte man på en den problemer. Instruksen har derfor blitt oppdatert. I tillegg har det blitt lagt til enkelte elementer som er relevant for denne oppgaven. Instruksen forklarer man installerer ROS og HECTOR_SLAM, som brukes til å behandle dataene som produseres av RPLidar. (Hassel, Moen, & Ohma, 2019)

Den forrige bacheloroppgaven benyttet seg av operativsystemet ubuntu 18, med ROS melodic. Dette var fordi denne ROS versjonen var kompetitiv med alle pakkene som skulle brukes. Siden den tid har det blitt lansert nyere versjoner av både ubuntu og ROS. Det ble likevel besluttet å benytte seg av de seneste versjonene av de samme distribusjonene, i den hensikt å unngå kompatibilitetsproblemer. Det ble altså benyttet siste versjon av UBUNTU 18.04, ROS MELODIC, HECTOR_SLAM, og RPLIDAR_ROS. Likevel oppsto det noen problemer, som hovedsakelig skyldtes at HECTOR_SLAM versjonen som ble lastet ned hadde blitt tilpasset en nyere distribusjon av ROS. Dette kunne blitt unngått dersom man hadde vært mer bevisst på dette under installering. Problemene og løsningene blir presentert i instruksen.

Dersom den som leser dette har til hensikt å gjenskape og/eller videreutvikle dette prosjektet, anbefales det å veie opp fordeler og ulemper med å bruke ny/eldre programvare.

Det første man begynner med er å gå til applikasjonen «Ubuntu Software» og deretter huke av alle alternativene som ligger under «downloadable from the internet». Videre må man legge til kilden for ROS i ubuntu sin kildeliste «sources.list». Dette gjøres ved hjelp av å kjøre kommandoen:

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

Det neste man må gjøre er å legge til nøklene for nedlastning fra denne serveren. Dette gjøres ved å først installere curl:

```
sudo apt install curl
```

Videre benytter man seg av kommandoen:

```
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
```

Deretter sjekker man i programmet «Software & Updates» under authentication at det har blitt lagt til en nøkkel:

F42ED6FBAB17C654 2019-05-30

Open Robotics info@osrfoundation.org

Til slutt bruker man kommandoen:

```
sudo apt update
```

Denne kommandoen sjekker at DEBIAN pakkene som er installert er de nyeste. Dersom man får noen som helst feilmeldinger på dette stadiet, anbefales det å finne årsaken før man går videre. Nå starter selve installasjonen. Ettersom det ble valgt å kjøre ROS fra en egen pc, ble den fulle versjonen av ROS MELODIC installert. Det gjør man ved å kjøre kommandoen:

```
sudo apt install ros-melodic-desktop-full
```

I dette prosjektet ble den fulle versjonen installert på datamaskinen den 27.09.23. For å slippe å henvise til den lokale ROS installasjonen hver gang man åpner et nytt terminalvindu, kan man legge dette inn i oppstartfilen for et terminalvindu (.bashrc). Dette gjøres ved å bruke «echo» kommandoen, som her skriver gjeldende input til «.bashrc»:

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

Deretter åpnes et nytt terminalvindu. Man kan også skrive følgende for å oppdatere gjeldende vindu:

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

```
sudo rosdep init
```

Og

```
rosdep update
```

ROSDEP skal bli lastet ned dersom man laster ned full desktop versjon. Hvis ikke dette skjer kan man laste det ned ved:

```
sudo apt install python-rosdep
```

ROSDEP hjelper oss å installere systemavhengigheter for de ulike kildene man bruker i prosjektet sitt. ROSDEP er også nødvendig for å kjøre enkelte kjerneelementer i ROS. Det siste som trenger å gjøres er å installere avhengighetene ROS trenger for å bygge pakker:

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

ROS er nå installert, og det må videre installeres et miljø for ROS. Det er i dette miljøet alt videre arbeid vil foregå. Først benytter «mkdir» kommandoen og oppretter en mappe «catkin_ws»:

```
mkdir -p ~/catkin_ws/src
```

Deretter bruker man kommandoen «catkin_make» for å lage selve miljøet. NB: det er viktig at denne kommandoen bare utføres når man befinner seg i mappen «/~catkin_ws», og øverst i mappestrukturen. Det anbefales derfor å sørge for dette ved å skrive kommandoen:

```
cd ~/catkin_ws
```

For så videre å bruke kommandoen:

```
catkin_make
```

Etter miljøet har blitt laget, legger man på samme måte som etter ROS installasjonen dette inn «.bashrc»:

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

Om man skal jobbe videre med samme terminalvindu, må man også skrive:

```
source devel/setup.bash
```

eller

```
source ~/catkin_ws/devel/setup.bash
```

Dette avhenger av hvor man befinner seg i mappestrukturen. ROS MELODIC er nå installert. Det neste man gjør er å laste ned pakker for RPlidar og HECTOR SLAM.

RPLIDAR ROS Pakke

Denne pakken er laget av Robopeak og er åpen kildekode (RoboPeak Public Repos, 2018) Det første man gjør for å installere pakken er å flytte seg til kildemappen i «catkin_ws»:

```
cd catkin_ws/src
```

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

```
git clone https://github.com/robopeak/rplidar_ros
```

Dette ble utført 27.09.23. Når denne mappen er klonet til kildemappen vår må miljøet lages igjen. Dette gjøres som tidligere nevnt ved hjelp av «catkin_make». Det er fortsatt viktig at denne kommandoen bare kjøres i hovedmappen «~/catkin_ws»:

```
catkin_make
```

For å kjøre den nye pakken må man åpne et nytt terminalvindu, eller skrive kommandoen:

```
source devel/setup.bash
```

eller

```
source ~/catkin_ws/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
```

Denne kommandoen gir pc'en anledning til å skrive til USB utgangen, noe den må for å sende kommandoer til RPLidar. Dette må gjøres hver gang lidarsensoren kobles til datamaskinen via USB. For å sjekke om lidarsensoren fungerer kan man kjøre kommandoen:

```
roslaunch rplidar_ros view_rplidar.launch
```

Dette vil åpne visualiseringsverktøyet RVIZ og vise hva Lidar «ser».

Dersom man bruker kommandoen:

```
rostopic list
```

Vil man få opp alle aktive topics. Når det bare er lidarsensoren som er aktiv, vil man få opp 3 topics, der den ene heter /scan. Dette topic'et publiserer utregningen av hvor punktene som lidar oppfatter er i forhold til sensoren. /scan blir brukt videre i SLAM.

HECTOR_SLAM ROS Pakke

HECTOR_SLAM er en SLAM algoritme som ligger åpent på nett og genererer kart ut ifra RPlidar sine observasjoner. (Technische Universität Darmstadt ROS Packages, 2022) For å installere, må man bevege seg til sourcemappen i catkin workspace:

```
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
```

I dette prosjektet ble dette gjort 08.10.23. Til slutt bruker man "catkin_make» for å lage miljøet. Husk på å kjøre denne kommandoen fra ~/catkin_ws.

```
cd ~/catkin_ws
```

så

```
catkin_make
```

Åpne et nytt terminalvindu eller skriv kommandoen:

```
source ~/catkin_ws/devel/setup.bash
```

Nå har HECTOR_SLAM noden i ROS blitt installert.

HECTOR_SLAM er nå installert. Dersom man skal benytte seg av pathfinding algoritmen til den forrige bacheloroppgaven, må man legge til en del filer fra filer fra PROJECT LIBERTY. (thedanielhassel, 2019) I dette prosjektet har det bare blitt brukt enkelte filer som angår HECTOR_SLAM og visualiseringen. Den første er oppstartsfilen til den modifiserte HECTOR_SLAM noden ved navn «rover.launch». Den legges inn i:

```
catkin_ws/src/hector_slam/hector_slam_launch/launch
```

I tillegg må man legge til «rviz_rover.rviz». Denne inneholder informasjon om hvordan det genererte kartet skal fremstilles. Filen legges inn i:

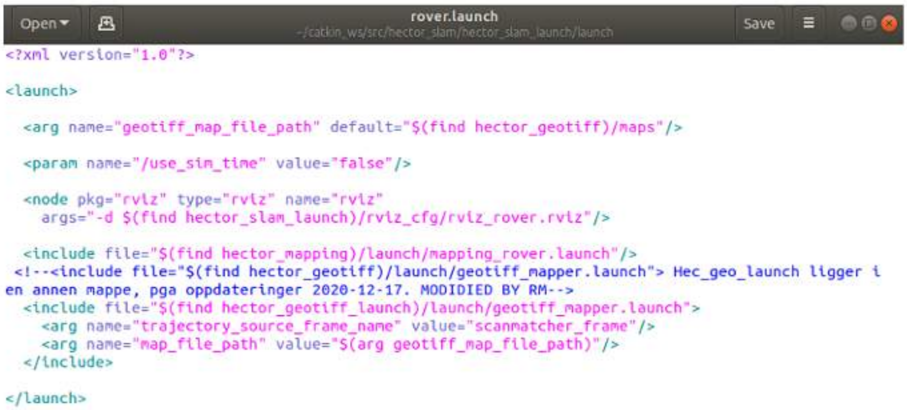
`catkin_ws/src/hector_slam/hector_slam_launch/rviz_cfg`

Til slutt må man legge inn filen «mapping_rover.launch», i mappen:

`catkin_ws/src/hector_slam/hector_mapping/launch`

For å komme seg videre, må man løse opp i noen kompatibilitetsproblemer. Siden 2019 har både ROS MELODIC, HECTOR_SLAM, og RPLIDAR_ROS gjennomgått oppdateringer. Dette medfører til enkelte problemer som man må fikse, dersom man skal benytte seg av kildekoden til Hassel et al.

Skjermbilde av scriptet



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

<launch>

  <arg name="geotiff_map_file_path" default="$(find hector_geotiff)/maps"/>

  <param name="/use_sim_time" value="false"/>

  <node pkg="rviz" type="rviz" name="rviz"
    args="-d $(find hector_slam_launch)/rviz_cfg/rviz_rover.rviz"/>

  <include file="$(find hector_mapping)/launch/mapping_rover.launch"/>
 <!--<include file="$(find hector_geotiff)/launch/geotiff_mapper.launch"> Hec_geo_launch ligger i
en annen mappe, pga oppdateringer 2020-12-17. MODIDIED BY RM-->
  <include file="$(find hector_geotiff_launch)/launch/geotiff_mapper.launch">
    <arg name="trajectory_source_frame_name" value="scanmatcher_frame"/>
    <arg name="map_file_path" value="$(arg geotiff_map_file_path)"/>
  </include>

</launch>
```

I bildet ovenfor kan man se at en linje i «rover.launch» blitt endret på. Den originale har blitt kommentert ut, og en ny linje under har blitt etablert. HECTOR_SLAM fikk en oppdatering, som flyttet undermappen /launch og dens innhold i mappen i /hector_geotiff, til en egen mappe kalt /hector_geotiff_launch. Dette førte til at «rover.launch» ikke fant filen «geotiff_mapper.launch». Det eneste som måtte gjøres var derfor å få «rover.launch» til å lete etter den nevnte filen på korrekt sted.

Disse filene (rover.launch/mapping_rover.launch/rviz_rover.rviz) har videre bitt modifisert i dette prosjektet, og kan hentes fra github:

https://github.com/borebni/Kartkorrelasjon

For å bruke de, legger man de inn i samme mapper som originalfilene. De heter:

«rover_test.launch», «mapping_rover_test.launch», «rviz_rover_test.rviz»

Man har nå anledning til å starte HECTOR_SLAM med forrige bacheloroppgaves innstillinger ved å bruke kommandoene:

```
roslaunch rplidar_ros rplidar.launch
```

og

```
roslaunch hector_slam_launch rover.launch
```

Disse må kjøres i separate terminalvinduer.

Den første kommandoen starter RPLidar. Den andre starter HECTOR_SLAM og åpner et RVIZ vindu med visualisering.

Vedlegg C: Bruk av ChatGPT som verktøy

Det finnes ingen tydelige retningslinjer for hvordan man skal kildeføre bruken ChatGPT som et verktøy. Dette vedlegget har til hensikt å redegjøre denne oppgavens bruk av verktøyet. ChatGPT 3.5 har generert et script som videre har blitt modifisert og endret for å kunne benyttes i oppgaven. For kode brukt i oppgaven se vedlegg D.

map_image.py

Chat-GPT ble brukt som et verktøy under utviklingen av scriptet map_image.py. Dette vedlegget viser relevant inndata og utdata fra verktøyet.

- Første spørring:

create a script in python that can extract the visualization in rviz and save it as a image file in a folder. I am using a RPlidar A1 and hector slam to create the certain. Explain the code thoroughly

Svar:

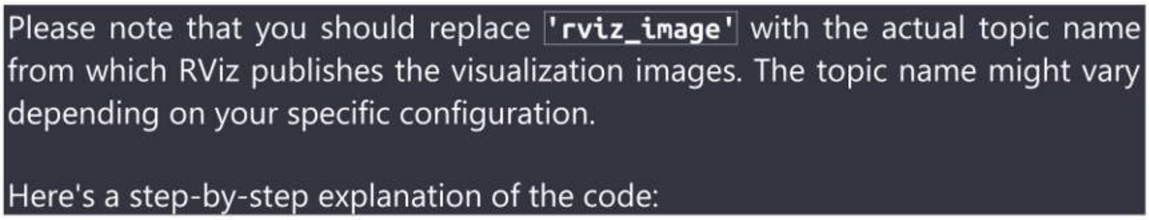
To capture visualizations from RViz and save them as image files, you can use a Python script in conjunction with the ROS (Robot Operating System) ecosystem. In this example, I'll provide you with a Python script that uses the `roslib` and `rospy` libraries to capture images from RViz and save them to a folder. This script assumes you have already set up ROS, RViz, RPLidar A1, and Hector SLAM in your environment.

Here's the Python script with thorough explanations:

```
#!/usr/bin/env python


# Import necessary libraries

import rospy

from sensor_msgs.msg import Image

from cv_bridge import CvBridge
```

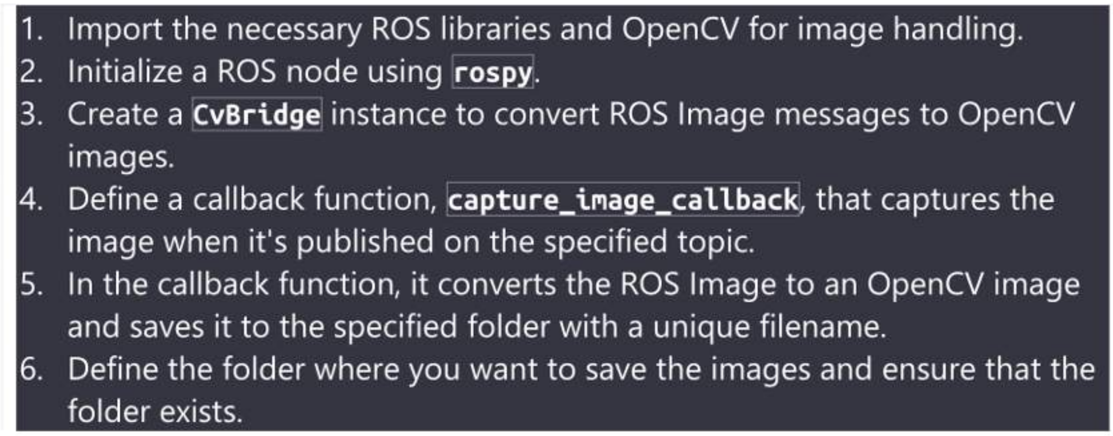
```
import cv2
import os

# Initialize the ROS node
rospy.init_node('rviz_screenshot_node')

# Initialize a CvBridge to convert ROS Image messages to
OpenCV images
bridge = CvBridge()

# Define a callback function to capture images when a new
visualization is published
def capture_image_callback(msg):
    try:
        # Convert the ROS Image message to an OpenCV image
        cv_image = bridge.imgmsg_to_cv2(msg, "bgr8")

        # Define the folder where you want to save the images
        save_folder = "/path/to/your/image/folder/"

        # Ensure the folder exists, create it if not
        if not os.path.exists(save_folder):
            os.makedirs(save_folder)

        # Generate a unique filename for each image (you can
customize this part)
        image_filename = os.path.join(save_folder,
f"rviz_screenshot_{rospy.Time.now()}.png")
```

```
        # Save the image to the specified folder
        cv2.imwrite(image_filename, cv_image)
        rospy.loginfo(f"Saved screenshot as {image_file-
name}")
    except Exception as e:
        rospy.logerr(f"Error capturing and saving image:
{e}")


# Subscribe to the RViz visualization topic, replace
'rviz_image' with the actual topic
rospy.Subscriber('rviz_image', Image, capture_im-
age_callback)


# Spin the ROS node to keep it running
rospy.spin()
```



Please note that you should replace `'rviz_image'` with the actual topic name from which RViz publishes the visualization images. The topic name might vary depending on your specific configuration.

Here's a step-by-step explanation of the code:



1. Import the necessary ROS libraries and OpenCV for image handling.
2. Initialize a ROS node using `rospy`.
3. Create a `CvBridge` instance to convert ROS Image messages to OpenCV images.
4. Define a callback function, `capture_image_callback`, that captures the image when it's published on the specified topic.
5. In the callback function, it converts the ROS Image to an OpenCV image and saves it to the specified folder with a unique filename.
6. Define the folder where you want to save the images and ensure that the folder exists.

7. Subscribe to the RViz visualization topic using `rospy.Subscriber`. Make sure to replace `'rviz_image'` with the actual topic name.
8. Finally, call `rospy.spin()` to keep the ROS node running.

Make sure your RViz configuration is set to publish images on the specified topic, and that topic name matches the one you're subscribing to in the script.

- Den neste spørringen omhandler hvordan man konverterer data fra topic'et «/map» til en bildefil:

Spørring:

i am not sure, but i think /map produces only numbers which represents if an area is occupied or not. How am i suppoused to end up with an image file (either png og jpeg) by using /map in the code?

Svar:

If the `/map` topic represents occupancy data and not a directly visualized image, you'll need to convert this occupancy grid map into an image format (e.g., PNG or JPEG) to save it as an image file. Here's how you can modify the script to achieve this:

```
#!/usr/bin/env python


# Import necessary libraries
import rospy
from nav_msgs.msg import OccupancyGrid
import numpy as np
from PIL import Image
import os
```

```
# Define a callback function to convert the occupancy grid to an image
def occupancy_grid_callback(msg):
    # Convert the occupancy grid data to a numpy array
    occupancy_data = np.array(msg.data).reshape((msg.info.height, msg.info.width))

    # Convert the occupancy data to an image
    occupancy_image = (255 * (1 - (occupancy_data / 100))).astype(np.uint8)
    occupancy_image = Image.fromarray(occupancy_image)

    # Define the folder where you want to save the images
    save_folder = "/path/to/your/image/folder/"

    # Ensure the folder exists, create it if not
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    # Generate a unique filename for each image (you can customize this part)
    image_filename = os.path.join(save_folder, f"map_screenshot_{rospy.Time.now()}.png")

    # Save the image to the specified folder
    occupancy_image.save(image_filename)
    rospy.loginfo(f"Saved map screenshot as {image_filename}")
```

```
if __name__ == '__main__':

    # Initialize the ROS node

    rospy.init_node('map_screenshot_node')


    # Subscribe to the /map topic

    rospy.Subscriber('/map', OccupancyGrid, occupancy_grid_callback)


    # Spin the ROS node to keep it running

    rospy.spin()
```

In this modified script:

1. We use the `/map` topic to subscribe to an `OccupancyGrid` message, which represents occupancy data.
2. In the callback function, we convert the occupancy grid data into an image using numpy and the Python Imaging Library (PIL). The occupancy grid values are transformed into grayscale pixel values.
3. We save the occupancy image as a PNG file in the specified folder.

Please make sure that the `/map` topic actually publishes occupancy grid data, and adjust the data transformation as needed based on the specific occupancy grid format in your environment.

- Den neste spørringen omhandler en feilmelding som skyldtes at deler av scriptet som ChatGPT generte inneholdt syntax som ble først introdusert i Python 3.69. Ettersom ROS MELODIC ikke støtter Python versjon 3 og oppover, måtte scriptet bruke eldre syntax for å få det til å fungere. Denne feilen som ChatGPT viser en løsning til nedenfor ble også fikset i en annen linje. Dette er den første spørringen som omhandler dette spesifikke problemet:

Spørring:

```
bo@bo:~/rviz_screen_test/script$ python screen_test.py File "screen_test.py", line 27 image_filename = os.path.join(save_folder, f"map_screenshot_{rospy.Time.now()}.png")
```

i tried to run and got this error. What could be wrong?

Svar:

The error you're encountering is due to the usage of f-strings, which were introduced in Python 3.6. It seems you are using a version of Python earlier than 3.6. To resolve this, you can either upgrade your Python version to 3.6 or later or modify the line to use the older string formatting method.

Here's how you can modify the line to use older string formatting:

Replace this line:

```
image_filename = os.path.join(save_folder, f"map_screenshot_{rospy.Time.now()}.png")
```

With this line using the `.format()` method for string formatting:

```
image_filename = os.path.join(save_folder, "map_screenshot_{}.png".format(rospy.Time.now()))
```

This modification should work with older versions of Python (Python 2.x or Python 3.x prior to 3.6) and allow you to create the image filename using the `rospy.Time.now()` timestamp.

Vedlegg D: Kildekode

Dette vedlegget inneholder kildekoden til oppgaven. For alle kodene må «path» og filnavn legges inn etter brukerens behov.

Kildekoden kan lastes ned fra GitHub:

GitHub - borebni/Kartkorrelasjon: Kodene fra bacheloroppgave om kartkorrelasjon

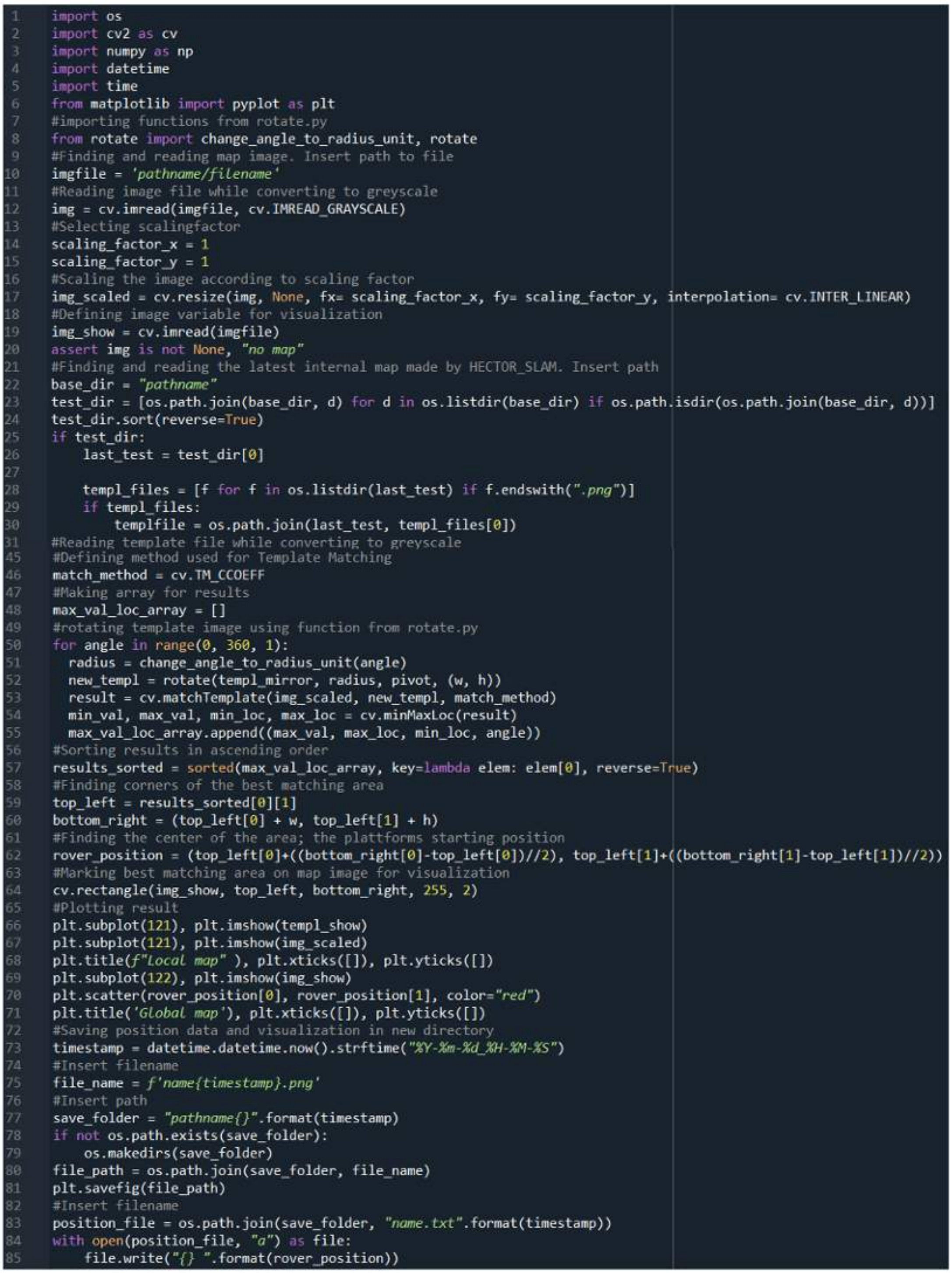
Shell-script:

```
python2.7 /path/map_image.py
sleep 3
python3 orb_matching.py
python3 template_matching.py
```

Rotasjonsfunksjon: rotate.py

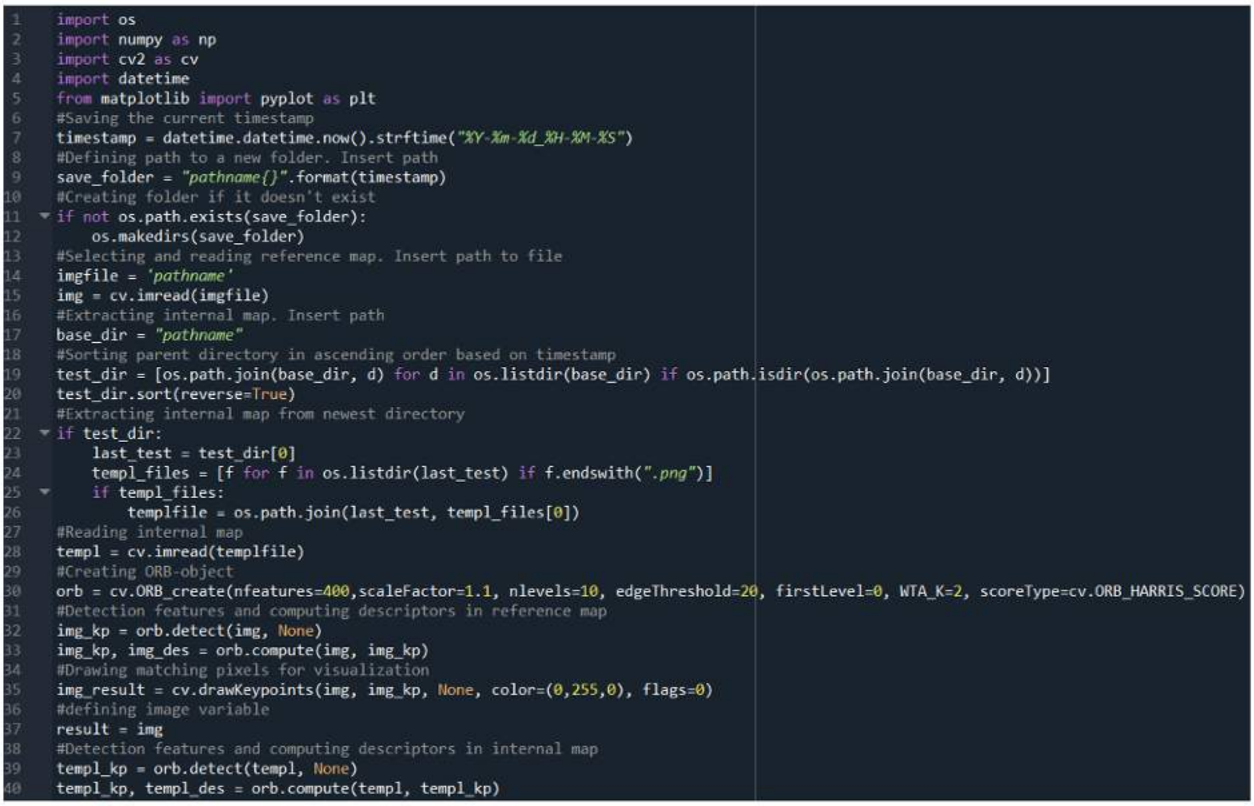
```python
import numpy as np
#function for converting degrees to radius
def change_angle_to_radius_unit(angle):
    angle_radius = angle * (np.pi/180)
    return angle_radius
def rotate(src_img,angle_of_rotation,pivot_point,shape_img):
    #creating rotation matrix
    rotation_mat = np.transpose(np.array([[np.cos(angle_of_rotation),-np.sin(angle_of_rotation)],
                                          [np.sin(angle_of_rotation), np.cos(angle_of_rotation)]]))
    w,h = shape_img
    pivot_point_x =  pivot_point[0]
    pivot_point_y = pivot_point[1]
    new_img = np.zeros(src_img.shape,dtype='u1')
    #rotating image
    for height in range(h): #h = number of row
        for width in range(w): #w = number of col
            xy_mat = np.array([[width-pivot_point_x],[height-pivot_point_y]])
            rotate_mat = np.dot(rotation_mat,xy_mat)
            new_x = pivot_point_x + int(rotate_mat[0])
            new_y = pivot_point_y + int(rotate_mat[1])
            if (0<=new_x<=w-1) and (0<=new_y<=h-1):
                new_img[new_y,new_x] = src_img[height,width]
    return new_img
```

Template Matching: template_matching.py

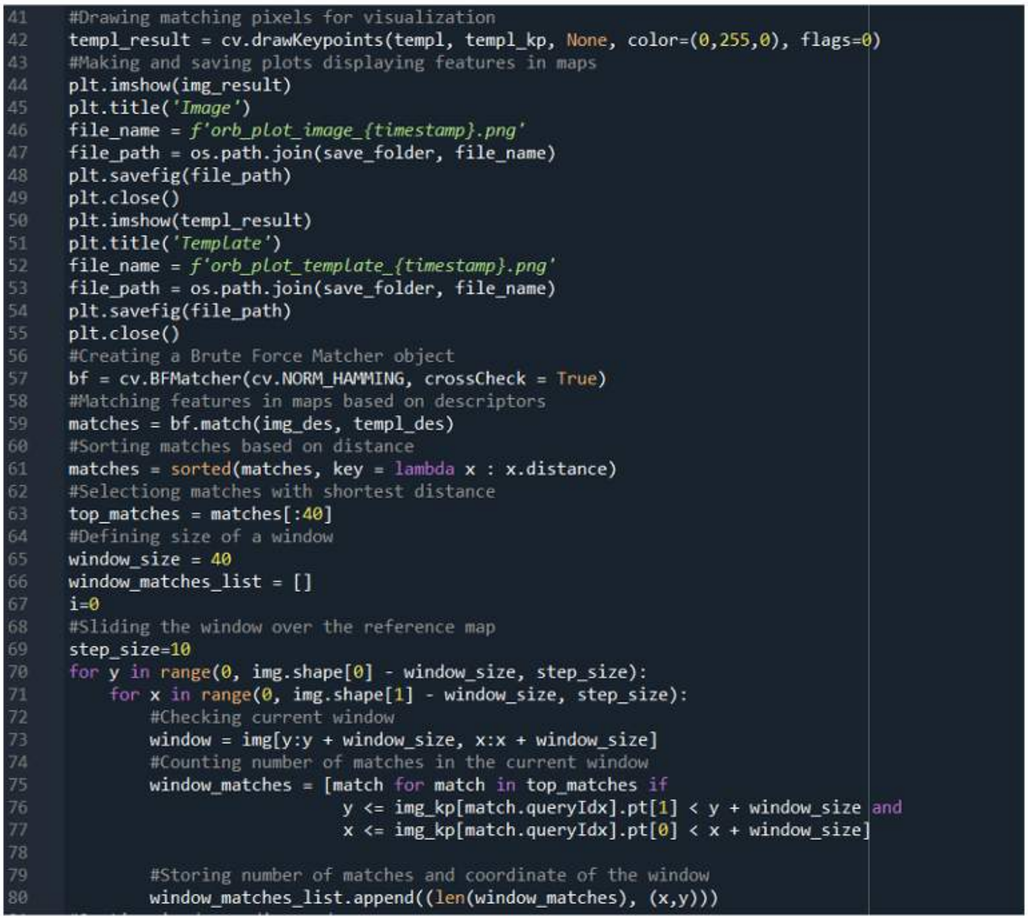


```python
import os
import cv2 as cv
import numpy as np
import datetime
import time
from matplotlib import pyplot as plt
#importing functions from rotate.py
from rotate import change_angle_to_radius_unit, rotate
#Finding and reading map image. Insert path to file
imgfile = 'pathname/filename'
#Reading image file while converting to greyscale
img = cv.imread(imgfile, cv.IMREAD_GRAYSCALE)
#Selecting scalingfactor
scaling_factor_x = 1
scaling_factor_y = 1
#Scaling the image according to scaling factor
img_scaled = cv.resize(img, None, fx= scaling_factor_x, fy= scaling_factor_y, interpolation= cv.INTER_LINEAR)
#Defining image variable for visualization
img_show = cv.imread(imgfile)
assert img is not None, "no map"
#Finding and reading the latest internal map made by HECTOR_SLAM. Insert path
base_dir = "pathname"
test_dir = [os.path.join(base_dir, d) for d in os.listdir(base_dir) if os.path.isdir(os.path.join(base_dir, d))]
test_dir.sort(reverse=True)
if test_dir:
    last_test = test_dir[0]

    templ_files = [f for f in os.listdir(last_test) if f.endswith(".png")]
    if templ_files:
        templfile = os.path.join(last_test, templ_files[0])
#Reading template file while converting to greyscale
#Defining method used for Template Matching
match_method = cv.TM_CCOEFF
#Making array for results
max_val_loc_array = []
#rotating template image using function from rotate.py
for angle in range(0, 360, 1):
  radius = change_angle_to_radius_unit(angle)
  new_templ = rotate(templ_mirror, radius, pivot, (w, h))
  result = cv.matchTemplate(img_scaled, new_templ, match_method)
  min_val, max_val, min_loc, max_loc = cv.minMaxLoc(result)
  max_val_loc_array.append((max_val, max_loc, min_loc, angle))
#Sorting results in ascending order
results_sorted = sorted(max_val_loc_array, key=lambda elem: elem[0], reverse=True)
#Finding corners of the best matching area
top_left = results_sorted[0][1]
bottom_right = (top_left[0] + w, top_left[1] + h)
#Finding the center of the area; the plattforms starting position
rover_position = (top_left[0]+((bottom_right[0]-top_left[0])//2), top_left[1]+((bottom_right[1]-top_left[1])//2))
#Marking best matching area on map image for visualization
cv.rectangle(img_show, top_left, bottom_right, 255, 2)
#Plotting result
plt.subplot(121), plt.imshow(templ_show)
plt.subplot(121), plt.imshow(img_scaled)
plt.title(f"Local map" ), plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.imshow(img_show)
plt.scatter(rover_position[0], rover_position[1], color="red")
plt.title('Global map'), plt.xticks([]), plt.yticks([])
#Saving position data and visualization in new directory
timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
#Insert filename
file_name = f'name{timestamp}.png'
#Insert path
save_folder = "pathname{}".format(timestamp)
if not os.path.exists(save_folder):
    os.makedirs(save_folder)
file_path = os.path.join(save_folder, file_name)
plt.savefig(file_path)
#Insert filename
position_file = os.path.join(save_folder, "name.txt".format(timestamp))
with open(position_file, "a") as file:
    file.write("{} ".format(rover_position))
```

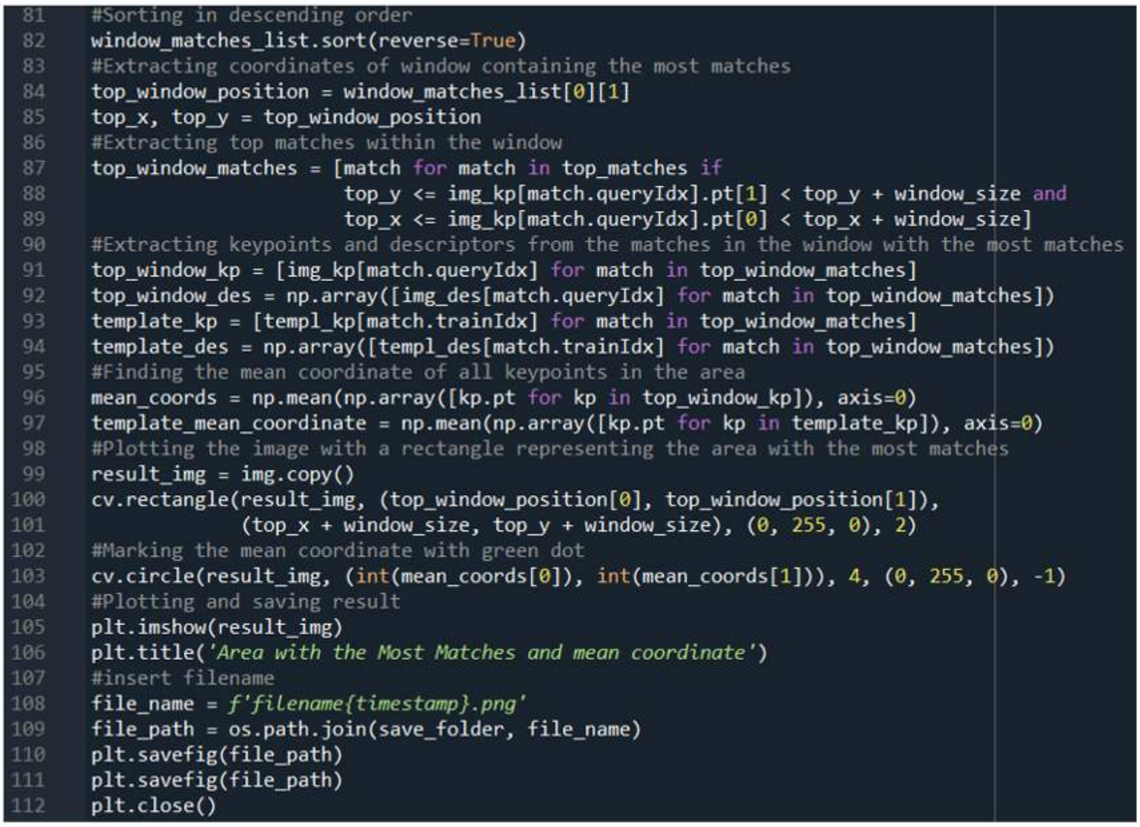
ORB: orb_matching.py



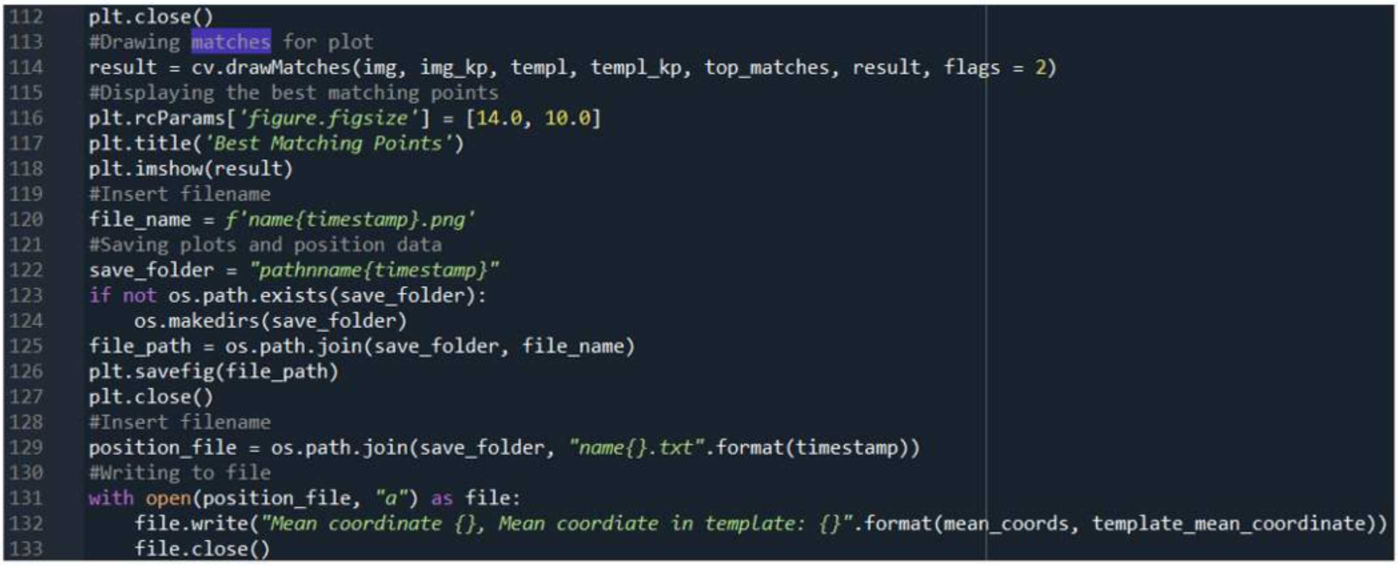
```python
import os
import numpy as np
import cv2 as cv
import datetime
from matplotlib import pyplot as plt
#Saving the current timestamp
timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
#Defining path to a new folder. Insert path
save_folder = "pathname{}".format(timestamp)
#Creating folder if it doesn't exist
if not os.path.exists(save_folder):
    os.makedirs(save_folder)
#Selecting and reading reference map. Insert path to file
imgfile = 'pathname'
img = cv.imread(imgfile)
#Extracting internal map. Insert path
base_dir = "pathname"
#Sorting parent directory in ascending order based on timestamp
test_dir = [os.path.join(base_dir, d) for d in os.listdir(base_dir) if os.path.isdir(os.path.join(base_dir, d))]
test_dir.sort(reverse=True)
#Extracting internal map from newest directory
if test_dir:
    last_test = test_dir[0]
    templ_files = [f for f in os.listdir(last_test) if f.endswith(".png")]
    if templ_files:
        templfile = os.path.join(last_test, templ_files[0])
#Reading internal map
templ = cv.imread(templfile)
#Creating ORB-object
orb = cv.ORB_create(nfeatures=400,scaleFactor=1.1, nlevels=10, edgeThreshold=20, firstLevel=0, WTA_K=2, scoreType=cv.ORB_HARRIS_SCORE)
#Detection features and computing descriptors in reference map
img_kp = orb.detect(img, None)
img_kp, img_des = orb.compute(img, img_kp)
#Drawing matching pixels for visualization
img_result = cv.drawKeypoints(img, img_kp, None, color=(0,255,0), flags=0)
#defining image variable
result = img
#Detection features and computing descriptors in internal map
templ_kp = orb.detect(templ, None)
templ_kp, templ_des = orb.compute(templ, templ_kp)
```



```python
#Drawing matching pixels for visualization
templ_result = cv.drawKeypoints(templ, templ_kp, None, color=(0,255,0), flags=0)
#Making and saving plots displaying features in maps
plt.imshow(img_result)
plt.title('Image')
file_name = f'orb_plot_image_{timestamp}.png'
file_path = os.path.join(save_folder, file_name)
plt.savefig(file_path)
plt.close()
plt.imshow(templ_result)
plt.title('Template')
file_name = f'orb_plot_template_{timestamp}.png'
file_path = os.path.join(save_folder, file_name)
plt.savefig(file_path)
plt.close()
#Creating a Brute Force Matcher object
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck = True)
#Matching features in maps based on descriptors
matches = bf.match(img_des, templ_des)
#Sorting matches based on distance
matches = sorted(matches, key = lambda x : x.distance)
#Selectiong matches with shortest distance
top_matches = matches[:40]
#Defining size of a window
window_size = 40
window_matches_list = []
i=0
#Sliding the window over the reference map
step_size=10
for y in range(0, img.shape[0] - window_size, step_size):
    for x in range(0, img.shape[1] - window_size, step_size):
        #Checking current window
        window = img[y:y + window_size, x:x + window_size]
        #Counting number of matches in the current window
        window_matches = [match for match in top_matches if
                           y <= img_kp[match.queryIdx].pt[1] < y + window_size and
                           x <= img_kp[match.queryIdx].pt[0] < x + window_size]

        #Storing number of matches and coordinate of the window
        window_matches_list.append((len(window_matches), (x,y)))
```

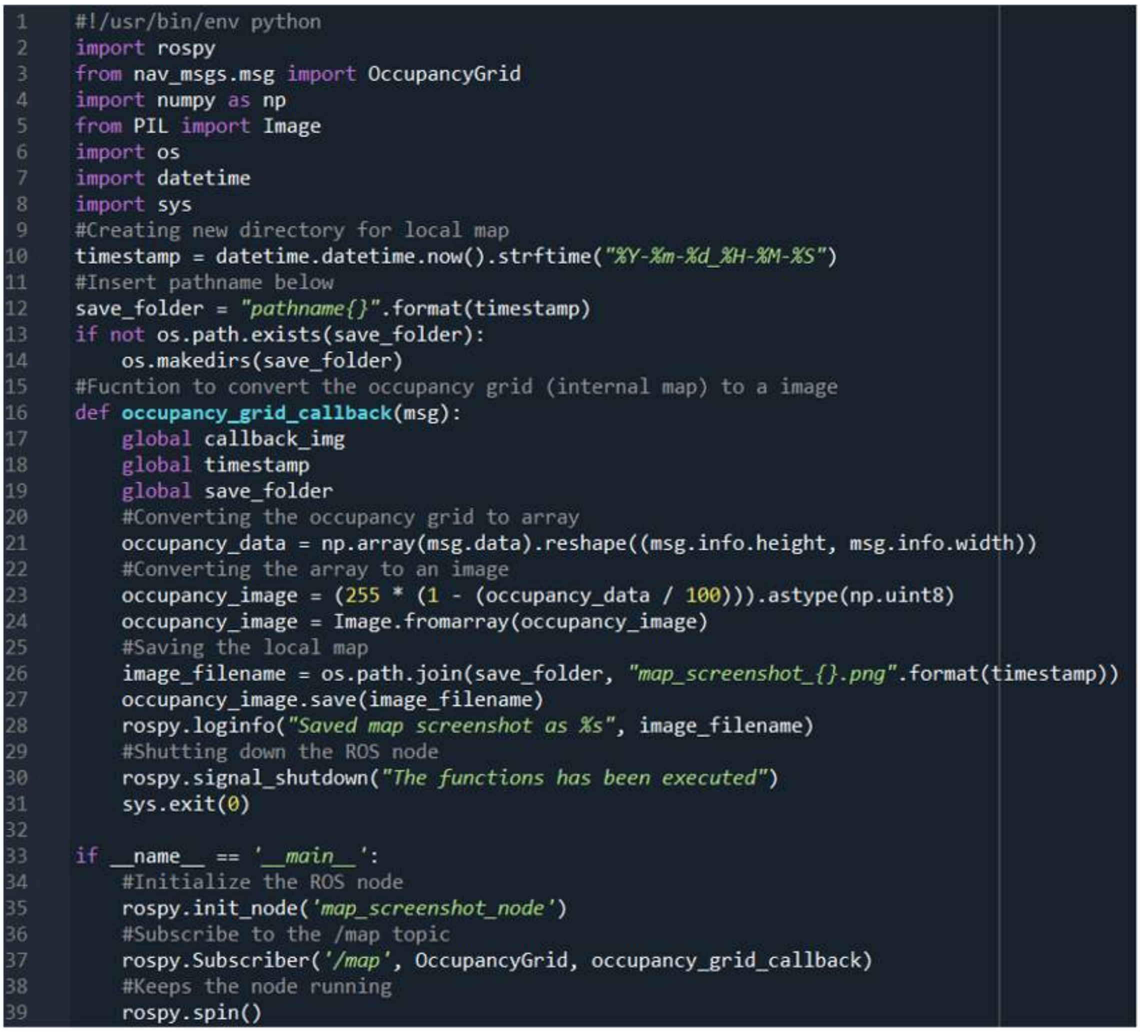


```
#Sorting in descending order
window_matches_list.sort(reverse=True)
#Extracting coordinates of window containing the most matches
top_window_position = window_matches_list[0][1]
top_x, top_y = top_window_position
#Extracting top matches within the window
top_window_matches = [match for match in top_matches if
                      top_y <= img_kp[match.queryIdx].pt[1] < top_y + window_size and
                      top_x <= img_kp[match.queryIdx].pt[0] < top_x + window_size]
#Extracting keypoints and descriptors from the matches in the window with the most matches
top_window_kp = [img_kp[match.queryIdx] for match in top_window_matches]
top_window_des = np.array([img_des[match.queryIdx] for match in top_window_matches])
template_kp = [templ_kp[match.trainIdx] for match in top_window_matches]
template_des = np.array([templ_des[match.trainIdx] for match in top_window_matches])
#Finding the mean coordinate of all keypoints in the area
mean_coords = np.mean(np.array([kp.pt for kp in top_window_kp]), axis=0)
template_mean_coordinate = np.mean(np.array([kp.pt for kp in template_kp]), axis=0)
#Plotting the image with a rectangle representing the area with the most matches
result_img = img.copy()
cv.rectangle(result_img, (top_window_position[0], top_window_position[1]),
             (top_x + window_size, top_y + window_size), (0, 255, 0), 2)
#Marking the mean coordinate with green dot
cv.circle(result_img, (int(mean_coords[0]), int(mean_coords[1])), 4, (0, 255, 0), -1)
#Plotting and saving result
plt.imshow(result_img)
plt.title('Area with the Most Matches and mean coordinate')
#insert filename
file_name = f'filename{timestamp}.png'
file_path = os.path.join(save_folder, file_name)
plt.savefig(file_path)
plt.savefig(file_path)
plt.close()
```



```
plt.close()
#Drawing matches for plot
result = cv.drawMatches(img, img_kp, templ, templ_kp, top_matches, result, flags = 2)
#Displaying the best matching points
plt.rcParams['figure.figsize'] = [14.0, 10.0]
plt.title('Best Matching Points')
plt.imshow(result)
#Insert filename
file_name = f'name{timestamp}.png'
#Saving plots and position data
save_folder = "pathnname{timestamp}"
if not os.path.exists(save_folder):
    os.makedirs(save_folder)
file_path = os.path.join(save_folder, file_name)
plt.savefig(file_path)
plt.close()
#Insert filename
position_file = os.path.join(save_folder, "name{}.txt".format(timestamp))
#Writing to file
with open(position_file, "a") as file:
    file.write("Mean coordinate {}, Mean coordiate in template: {}".format(mean_coords, template_mean_coordinate))
    file.close()
```

Kode for å hente lokalt kart: map_image.py



```python
#!/usr/bin/env python
import rospy
from nav_msgs.msg import OccupancyGrid
import numpy as np
from PIL import Image
import os
import datetime
import sys
#Creating new directory for local map
timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
#Insert pathname below
save_folder = "pathname{}".format(timestamp)
if not os.path.exists(save_folder):
    os.makedirs(save_folder)
#Fucntion to convert the occupancy grid (internal map) to a image
def occupancy_grid_callback(msg):
    global callback_img
    global timestamp
    global save_folder
    #Converting the occupancy grid to array
    occupancy_data = np.array(msg.data).reshape((msg.info.height, msg.info.width))
    #Converting the array to an image
    occupancy_image = (255 * (1 - (occupancy_data / 100))).astype(np.uint8)
    occupancy_image = Image.fromarray(occupancy_image)
    #Saving the local map
    image_filename = os.path.join(save_folder, "map_screenshot_{}.png".format(timestamp))
    occupancy_image.save(image_filename)
    rospy.loginfo("Saved map screenshot as %s", image_filename)
    #Shutting down the ROS node
    rospy.signal_shutdown("The functions has been executed")
    sys.exit(0)

if __name__ == '__main__':
    #Initialize the ROS node
    rospy.init_node('map_screenshot_node')
    #Subscribe to the /map topic
    rospy.Subscriber('/map', OccupancyGrid, occupancy_grid_callback)
    #Keeps the node running
    rospy.spin()
```

Vedlegg E: Resultater

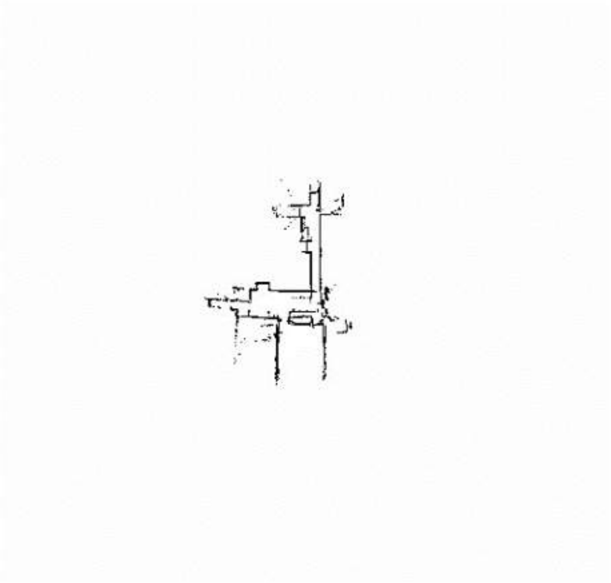
Dette vedlegget inneholder resultatene fra testingen av kartkorrelasjonsmetodene, samt de lokale og globale kartene brukt i testingen.

Resultater:

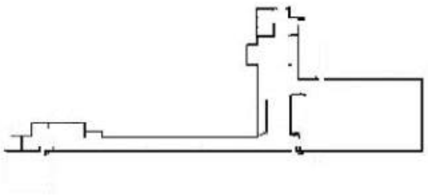
	Startposisjon		Estimert posisjon						Estimert posisjon	
Kart 1	x	y	x	y	Avvik	Gjennomsnittskoordinat i globalt kart	Gjennomsnittskoordinat i lokalt kart:	Avvik	x	y
1.1	257	332	255	332	60	(254.177, 311.162)	(221.208, 198.592)	95,1	254	332
1.2	294	327	298	329	134,1640786	(253.571, 310.329)	(181.876, 244.303)	188,3	296	333
1.3	304	263	301	315	1562,593997	(217.697, 321.487)	(190.972, 193.434)	3306,5	219	333
2.1	260	324	259	323	42,42640687	(255.375, 310.203)	(189.411, 225.127)	626,2	242	334
2.2	260	324	259	323	42,42640687	(260.6, 310.525)	(202.848, 198.697)	463,7	263	309
2.3	260	324	259	323	42,42640687	(260.846, 312.285)	(205.678, 208.485)	185,5	255	321
			Gjennomsnitt:		314,0062161	Gjennomsnitt:		810,8833333		
			Median		42,42640687	Median		188,3		
Kart 2										
1.1	453	390	350	223	4359,765584	(228.544, 467.519)	(200.807, 189.586)	4963,2	242	463
1.2	443	455	327	287	4536,364151	(228.39, 469. 478)	(203.661, 234.132)	5629,3	189,9	443,6
1.3	267	471	350	223	5810,986703	(228.842, 468.928)	(190.77, 197.48)	776,4	233,7	481,4
2.1	436	395	350	223	4272,94702	(437.8, 363.771)	(197.954, 230.828)	583,9	409,7	394,6
2.2	436	395	350	223	4272,94702	(437.52, 366.237)	(195.617, 205.663)	725,4	447	364
2.3	436	395	350	223	4272,94702	(432.008, 365.062)	(201.912, 209.516)	951,9	434	352
			Gjennomsnitt:		4587,659583	Gjennomsnitt:		2271,683333		
			Median		4316,356302	Median		864,15		
Kart 3										
1.1	623	391	631	461	1565,524761	(587.898, 393.649)	(192.302, 199.61)	741,5	592	403
1.2	613	456	631	461	415,1038564	(589.716, 387.748)	(195.146, 231.395)	673,9	598	430
1.3	437	472	631	461	4317,603872	(590.125, 384.648)	(196.139, 197.435)	3833,5	589	391
2.1	606	396	631	461	1547,443938	(590.218, 387.228)	(188.661, 244.868)	1682,7	576	326
2.2	606	396	631	461	1547,443938	(589.594, 387.717)	(193.514, 193.723)	667,3	582	379
2.3	606	396	631	461	1547,443938	(727.8, 473.443)	(213.322, 211.738)	2840,5	722	450
			Gjennomsnitt:		1823,427384	Gjennomsnitt:		1739,9		
			Median		1547,443938	Median		1212,1		
Kart 4										
1.1	915	322	249	342	4317,625507	(906.242, 361.587)	(200.237, 211.499)	562,5	935	407
1.2	880	530	206	342	4534,240802	(903.588, 278.714)	(194.191, 285.409)	3167,2	1300	280
1.3	319	585	249	375	1434,409147	(219.882, 450.077)	(187.179, 190.143)	1569,8	173	392
2.1	864	337	249	375	3992,800168	(842.24, 459.91)	(198.805, 237.895)	645,4	772	299
2.2	864	337	249	375	3992,800168	(842.154, 459.582)	(197.093, 178.976)	863,9	755	414
2.3	864	337	249	375	3992,800168	(802.358, 360.364)	(204.424, 205.311)	583,8	791	390
			Gjennomsnitt:		3710,779326	Gjennomsnitt:		1232,1		
			Median		3992,800168	Median		754,65		

De globale kartene:

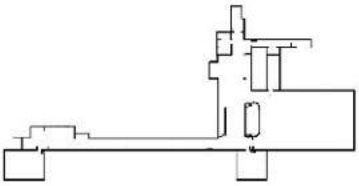
Kart 1:



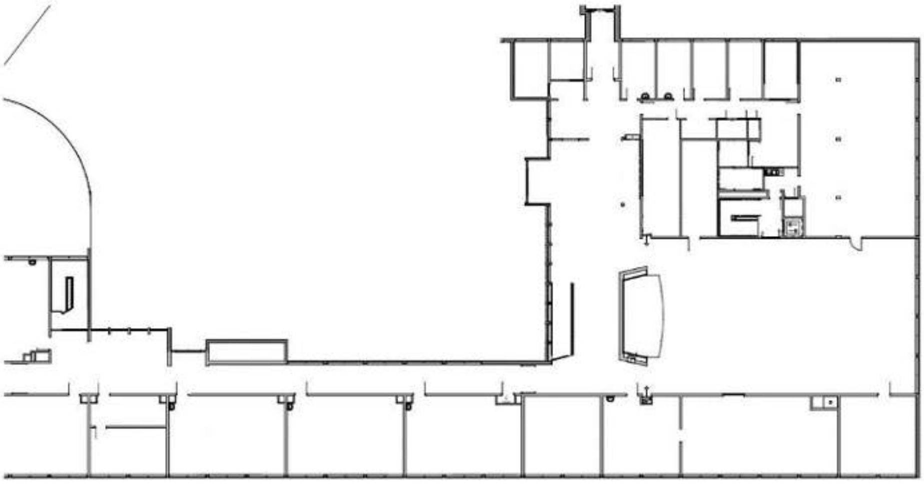
Kart 2:



Kart 3:

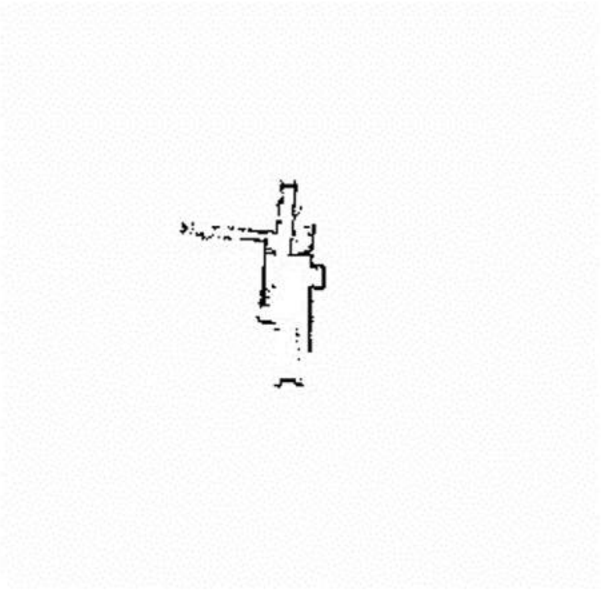


Kart 4:

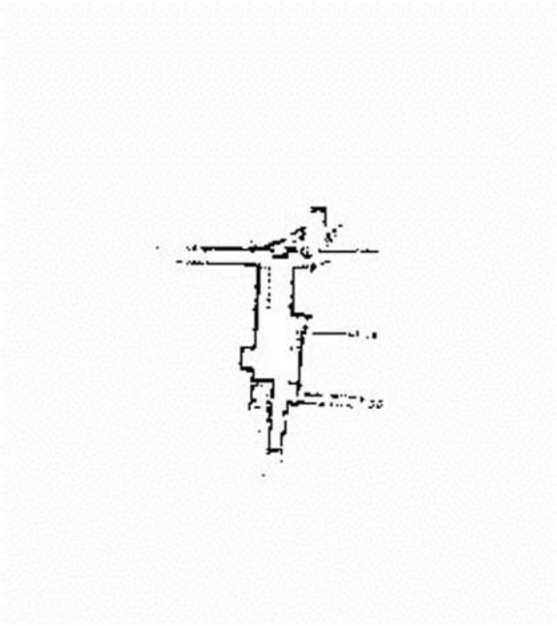


De lokale kartene:

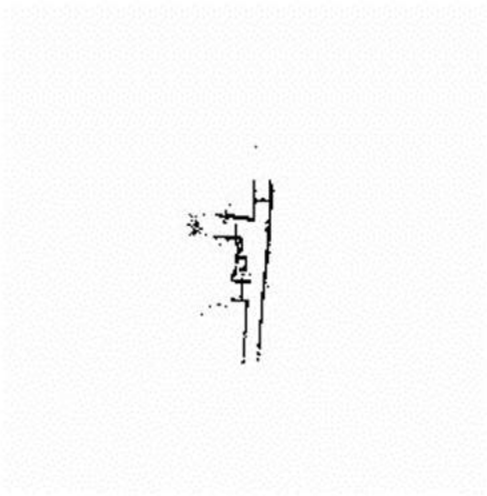
Test 1.1: Under trapp ved resepsjon



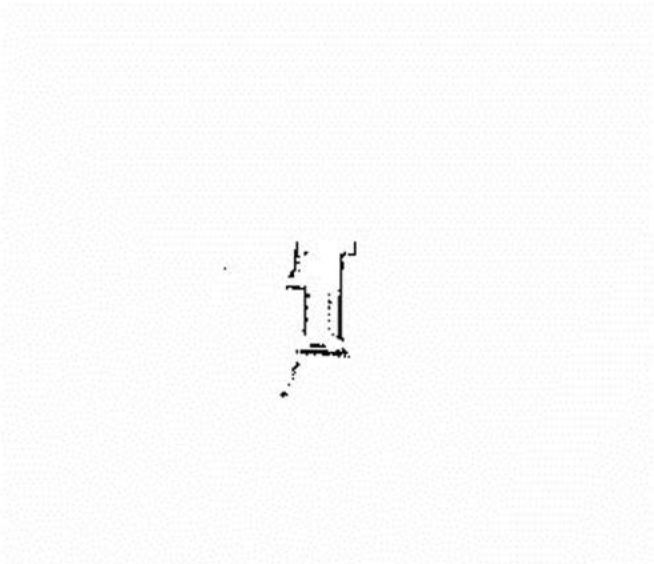
Test 1.2: Ved flagg



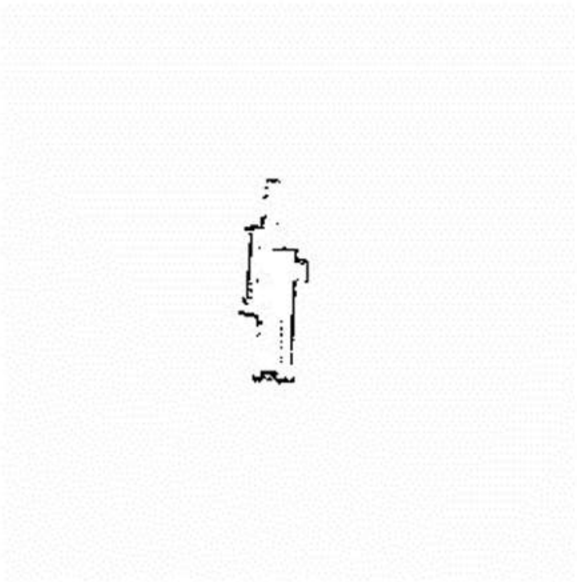
Test 1.3: Ved datalab



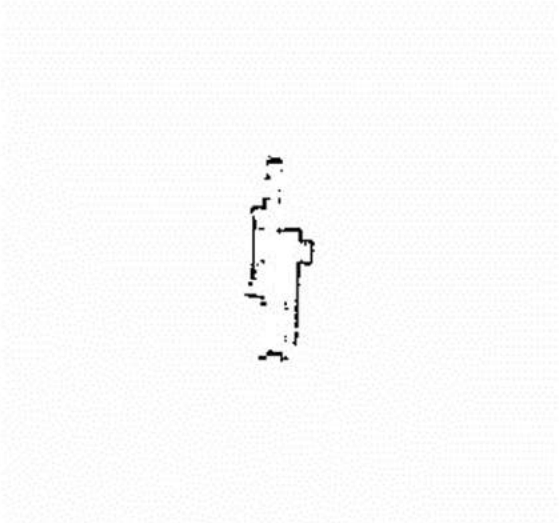
Test 2.1: Plattformen ble flyttet 10 meter frem



Test 2.2: Plattformen ble flyttet 2 meter frem, og deretter rotert 90 grader mot høyre. Dette ble gjentatt frem til plattformen befant seg på samme sted som den startet.



Test 2.3: Plattformen roterte rundt egen akse



Vedlegg F: Skisse over skolebygget

Dette er en ugradert tegning over en del av skolebygget. Tegningen ble hentet fra Forsvarsbygg på Sjøkrigsskolen.

Det ble gitt tillatelse av driftskoordinator Vestland, eiendomsforvaltning region vest, Forsvarsbygg til å bruke tegningen i dette prosjektet. Bruk utover dette skal <u>ikke</u> forekomme uten videre avtale med Forsvarsbygg.