ROB2 rapport - gruppe 15

Forår 2019

https://www.turtlebot.com/assets/images/turtlebot_2_lg.png

Oscar Pradel - 201406711
Mikkel Kastrup - 201509809
Simon Hermansen - 201500156

Indholdsfortegnelse

Systembeskrivelse

Den benyttede TurtleBot 2 består af en Kobuki base med et 2200/4400 mAh batteri. Denne Kobuki base er med relevant hardware til at håndtere mange opgaver. Af hardware kan nævnes følgende:

- Fabrikskalibreret gyroskop
- Odometri med 25718 ticks pr. rev
- Tre bumpere (venstre, midt, højre) til håndtering af påkørsel
- Fald-sensorer til at registrere højdeforskelle og derved undgå at køre ud over kanter
- ”Wheel drop”-sensorer der registrerer hvornår der ingen vægt er på hjulene

På denne base er der monteret to plateauer til placering af hhv. en bærbar computer til håndtering af kommunikationen mellem TurtleBot og egen bærbar, og ét til selve kameraet. Kameraet er et Asus Xtion Pro kamera med IR lys, farve- og dybdebillede. Kameraet er derfor i stand til at tage RGB billeder, Point cloud og laser scan.

TurtleBots software håndteres af en HP bærbar computer med Ubuntu 12.04 LTS styresystem. Der er ydermere specielle Kobuki ROS drivere, non-ROS drivere til programmering i C++ samt Kobuki og TurtleBot (Gazebo) simulatorer.

Hvad er ROS?

ROS (Robot Operating System) er et open source software framework der tillader brugeren at skrive softwareapplikationer til at styre en robot - i vores tilfælde en TurtleBot 2. Denne platform stiller desuden nogle værktøjer og biblioteker til rådighed samt faciliterer hardware-abstraktioner. ROS bygger grundlæggende på to principper:

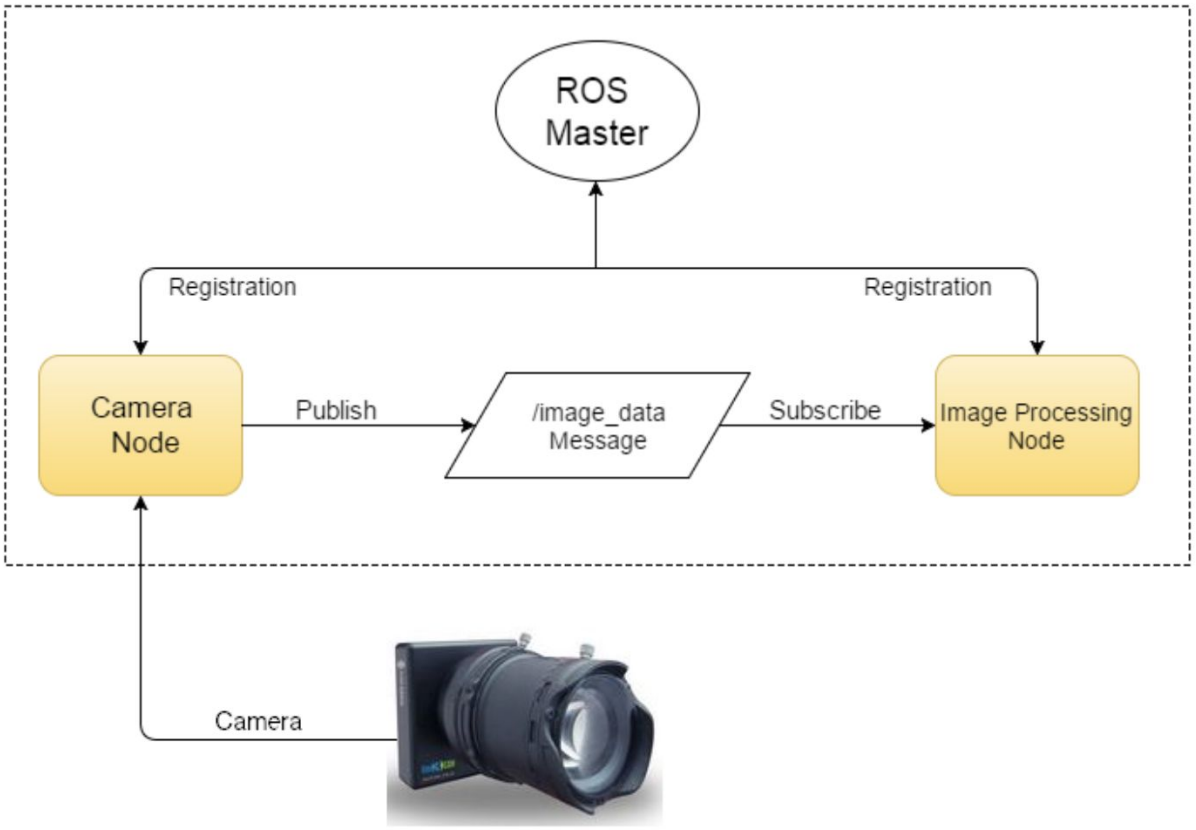
1. Kode genbrugelighed
2. Modulære designkomponenter der kan implementeres individuelt og let sammenkobles ved runtime

Kode genbrugelighed skal forstås således, at man ved at benytte ROS package management system, nemt kan opbygge ens eget system ved at benytte flere ROS pakker. Man skal derfor ikke selv ud og skabe modulerne fra ny, men kan benytte de forskellige præfabrikerede moduler der bliver kompilet og eksekveret individuelt.
Det andet grundprincip er særligt interessant, idet det tillader os at opbygge et samlet program i separate moduler. Vi har blandt andet i denne opgave benyttet os meget af dette, og separeret bl.a. navigation fra billedgenkendelses-delen for til sidst at sammenkoble dem.

Dette var den helt grundlæggende beskrivelse. Mere konkret er ROS systemet opbygget af en række uafhængige noder, som alle kommunikerer med hinanden gennem en publish/subscribe model. Disse noder kunne f.eks. være sensor driver, actuator driver, mapper, planner osv.

ROS starter med en ROS Master. Denne Master tillader alle andre ROS noder at finde hinanden og kommunikere. Masteren kan derfor ses som en tabel der viser aller noderne hvor de skal sende beskederne hen.

Nedenstående eksempel kunne være fra vores TurtleBot med kameraet. Vi har her en kamera-node der håndterer kommunikationen med kameraet, en billedprocesserings-node der håndterer dataet, og til sidst en display-node, der endeligt viser billeder på computerens skærm.
I et sådan tilfælde, vil alle noder være registreret med Masteren. Ved at registrere med Masteren fortæller kamera-noden, at den vil publicere et *topic* kaldet **/image_data**. Idet både vores billedprocesserings-node og display-node abonnerer på dette topic, vil begge noder automatisk modtage **/image_data** beskeden når kamera-noden modtager data fra kameraet.



Figur 1, ROS Publish/Subscribe[1]

Ved at implementere services kan man gøre det omvendte og anmode (requeste) om data fra kameraet på et givet tidspunkt. En node kan derfor registrere en specifik service med ROS masteren, præcis som den ville registrerer en besked.

[1] https://www.e-consystems.com/Articles/Camera/see3cam-10cug-c-with-ros-robot-operating-system.asp

Algoritmer

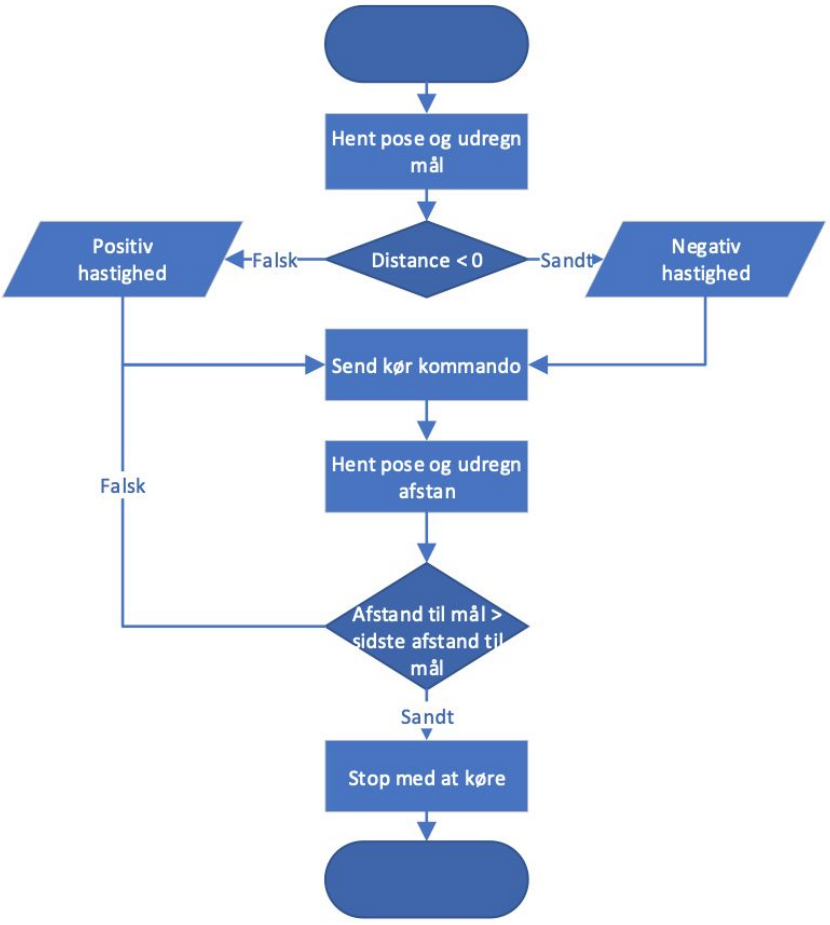
Dette afsnit præsenterer de forskellige algoritmer, som er udarbejdet i udviklingen af projektet.

getRobotPose

Denne funktion bruges til at trække robottens odometri-data ud og derefter udregne robottens position. Positionen udregnes på baggrund af robottens startposition, så funktionen returnerer robottens reelle position på mappet - se mere om live-mapping i afsnittet om utility-funktioner. Funktionens bruges igennem hele projektet i flere funktioner til at få robottens nuværende position.

driveDistance

Denne funktion får robotten til at køre en given afstand med en given hastighed ud fra dens odometri. Et flowdiagram for denne algoritme kan ses i figuren nedenfor. Funktionen bruges når der skal køres til en bestemt afstand fra væggen ved billeddetektion. Parametrene til funktionen er informationer om robotten (herunder dens odometri), samt den ønskede distance og hastighed.



Figur 3, Flowdiagram for driveDistance algoritme

Funktionen starter med at få robottens position med funktionen *getRobotPose*, og derefter udregne x- og y-koordinaterne for målet. Nu kører robotten (enten frem eller tilbage afhængig af målet) indtil den når sit mål.

rotateToAngle

Roterer robotten til en given vinkel i forhold til robottens startposition. Denne funktion bruges når robotten har nået sit første mål og skal i gang med billeddetektion. Her kendes robottens position og funktionen kan derfor benyttes til at rotere robotten ind mod væggen med billeder, da robottens front ikke peger mod væggen. Her bruges funktionen *getRobotPose* til at få robottens position.

rotate

Roterer robotten i en given retning, i et givet stykke tid, med en given hastighed. I forhold til *rotateToAngle* er denne funktion mere dum, i den forstand at den ikke indeholder så megen logik. Her kan der blot sendes tid, hastighed og retning, hvor robotten så vil rotere enten med eller mod uret i en mængde tid, med en hastighed. *rotate*-funktionen bruges under billedgenkendelse, når der skal køres mellem forskellige objekter. Her bruges funktionen til

at rotere 90 grader den ene vej, for herefter at rotere 90 grader den anden vej.

getRangeAndAngle

Denne funktion bruger robottens *laserscanner*. Scanneren giver en 2D-visning af robottens omgivelser, som kan bruges til at hjælpe robotten med lokalisering - f.eks. ved billedgenkendelse for at være den korrekte afstand og vinkel fra væggen. I funktionen tjekkes der først for et validt scan. Hvis der ikke er et validt scan efter 5 forsøg returneres -1 for både længde og retning. Hvis ikke konverteres scannet til et kartesisk plot, som der kan læses fra. Herefter laves der nogle tjek for at få den korrekte vinkel, som returneres i radianer, sammen med den korteste længde i centimeter.

lookStraightAtWall

Denne funktion får robotten til stå vinkelret i forhold til væggen. Her bruges *getRangeAndAngle*-funktionen til at få den initiale retning, hvorefter *rotate*-funktionen bruges for at få robotten til at stå vinkelret mod væggen. Der regnes med en margin på 0,05 grader.

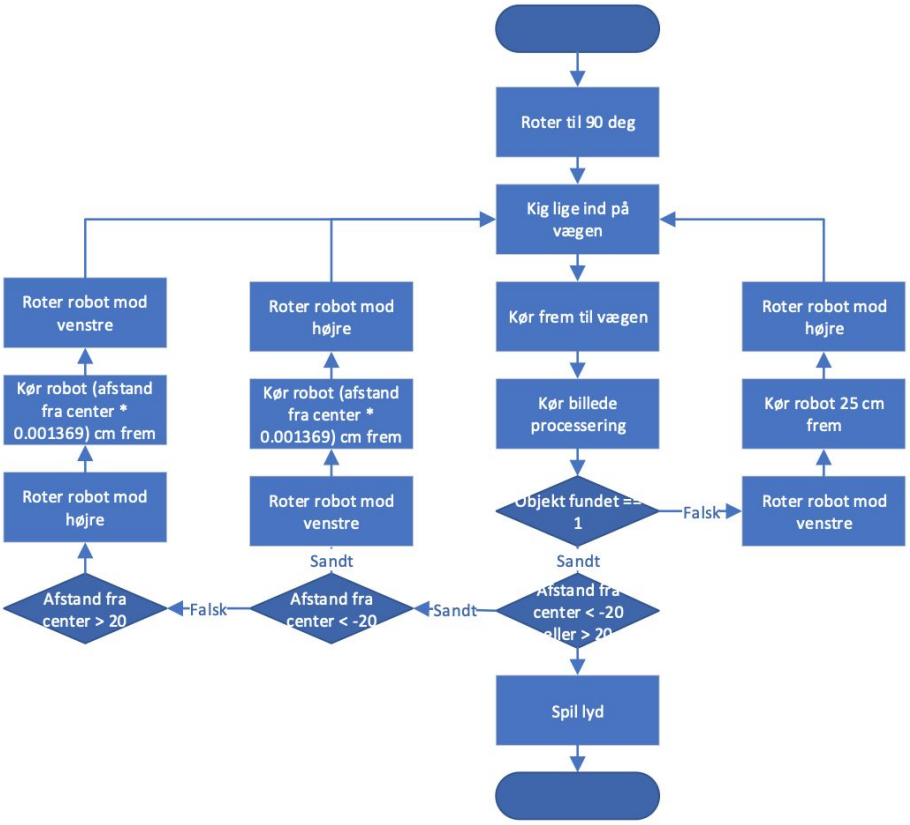
objectRecognition

Denne funktion udfører billedgenkendelsen, som bygger på det filter der blev præsenteret i uge 5. Denne står for at filtrere billedet så det kun er de udleverede objekter der er i fokus. Herefter benyttes en algoritme[2] fra *MathWorks* som skal identificere runde objekter og give en score på baggrund af hvor rundt et objekt er. Hvis det fundne objekt både er grønt og rundt returneres objectFound = 1. Og hvis der findes andre objekter, returneres der 2 samt afstanden til objektets center i pixels.

findGreenDot

Denne funktion får robotten til at scanne væggen ved punkt B, for at finde den grønne cirkel. Først roterer robotten så den kigger ind mod væggen med funktionen *rotateToAngle*. For at få robotten til at stå vinkelret mod væggen (for at få det reneste billede) benyttes funktionen *lookStraightAtWall*. Nu kører robotten til en afstand af 40 cm fra væggen og bruger billedprocessering for at finde objekter. Der returneres om et objekt er fundet og dets afstand til væggen. Hvis objektet (den grønne cirkel) ikke er fundet scanner robotten videre, ved at lave sekvensen set til højre på flowdiagrammet nedenfor.

[2] Metode til at finde runde objekter:
https://se.mathworks.com/help/images/identifying-round-objects.html



Figur 4, Flowdiagram for findGreenDot algoritme

Som det kan ses til højre i flowdiagrammet roterer robotten først 90 grader mod venstre, kører 25 cm frem og drejer 90 grader mod højre igen. Sekvensen kan nu starte forfra og gentages til den grønne cirkel er fundet. Herefter udregner vi afstanden fra robottens center til den grønne cirkel ved at gange pixelværdien med en faktor. Faktoren er fundet ved at tage et billede af et målebånd med en kendt afstand til væggen - se afsnittet om resultater for mere information. Når afstanden er fundet drejer robotten alt efter hvilket side af robottens center den grønne cirkel befinder sig på. Nu køres den fundne afstand, så robotten kommer til at stå lige ud for cirklen, og drejer ind mod væggen igen. Så kan der køres frem til væggen og hvis den grønne cirkel er lige ud for robotten (20+/- pixels) så spilles der en lyd.

driveToCircleAndGoBack

Følgende afsnit præsenterer *driveToCircleAndGoBack*, der bl.a. indeholder *driveFromX2Y* funktionen. Herunder vil PRM, Pure Pursuit og VFH algoritmerne blive forklaret.

PRM

Probabilistic Road Map er en velkendt motion-planning algoritme. PRM fungerer ved at indlæse et kort, og tage tilfældige samples i de frie område omkring robotten. Algoritmen

prøver dernæst at forbinde de forskellige samples og derved opnå en samlet forbindelse mellem punkter. Man kan derefter benytte en path finding algoritme til at finde den korteste vej fra start til slut.
PRM kræver at man i forvejen har oprettet et *robotics.BinaryOccupancyGrid*[3] objekt på kortet. Dette *BinaryOccupancyGrid* bliver brugt til at visualisere robottens workspace samt mulige forhindringer. Hver celle i et *OccupancyGrid* får en værdi der repræsenterer cellens status. Er cellen optaget/er det en forhindring, bliver cellen tildelt værdien 1 (true) og er den ledig, bliver den tildelt værdien 0 (false). Deraf binary delen i navnet.

I vores tilfælde indlæser vi et billede, konverterer det til greyscale og til sidst til et B/W-billede baseret på et bestemt threshold. Dernæst benyttes dette B/W image sammen med vores *k-konstant* som parametre i vores *robotics.OccupancyGrid* objekt.

Der kan nu laves et roadmap ved at benytte *robotics.PRM*[4] med vores map. *Prm.NumNodes* sættes til 1000 (50 er default), da et større antal noder øger antallet af forbindelser. *Prm.ConnectionDistance*, der bestemmer den maksimale afstand mellem forbundne noder, sættes til 10.
En forbindelse kan nu blive oprettet mellem to noder, så længe der ingen forhindringer er imellem dem og så længe de er inden for den angivne *ConnectionDistance* på 10.
I *driveToCircleAndGoBack* kan vi nu benytte *findpath* metoden der som parametre tager PRM, et start-koordinat og et slut-koordinat, og genererer en path.

Pure Pursuit

Da vi nu har styr på *BinaryOccupancyGrid* og PRM, kan vi nu implementere en path tracking algoritme. Pure Pursuit algoritmen fungerer ved at oprette et controller-objekt der får robotten til at følge en bestemt path. Denne controller kan tilpasses ved at definere en række parametre, som vil blive nærmere gennemgået herunder.

Der oprettes i første omgang et *robotics.PurePursuit*[5] objekt og dens parametre sættes. Her er især *controller.LookaheadDistance* interessant. En relativ høj look-ahead værdi vil resultere i ret flydende ruter, der dog kan risikere at tage svingene meget bredt. En lavere look-ahead værdi vil derimod følge PRM ruten mere præcist, og tage svingene skarpt. Grundet de smalle gange i Shannon bygningen, valgte vi derfor efter test med forskellige parametre, at halvere look-ahead distance værdien fra 1 til 0.5.
Controllerens waypoints bliver nu sat til at være vores tidligere definerede PRM path og robottens CurrentPose samt dens goal bliver sat.
Vi kan nu beregne afstande til målet ved at tage *norm* af robottens nuværende placering fratrukket det endelige mål.

Vi kører nu robotten ved at opsætte et while loop der kører så længe distanceToGoal er større end vores prædefinerede goalRadius (0.1 meter).

---

[3] BinaryOccupancyGrid: https://se.mathworks.com/help/robotics/ref/robotics.binaryoccupancygrid-class.html
[4] PRM: https://se.mathworks.com/help/robotics/ref/robotics.prm-class.html
[5] Pure Pursuit: https://se.mathworks.com/help/robotics/ref/robotics.purepursuit-system-object.html

While loopet sørger for, at håndtere alt fra computering af controllerens output (inputs til robotten), extraction af robottens nuværende placering vha. *robot.getRobotPose* samt at genberegne den nye afstand til målet.

VFH

Blandt de stillede krav til denne opgave, skulle vores robot være i stand til at undgå forhindringer på dens vej. Til at løse dette krav, valgte vi at benytte *robotics.VectorFieldHistogram* biblioteket (VFH[6]). VFH benytter vores TurtleBots laser scan range data, samt en given destinationsretning til at beregne en vej udenom en eventuel forhindring.

Til VFH følger en del parametre. *vfh.DistanceLimits* bestemmer et min/max af afstandsmålingen. Som default er *min* sat til 0.05 m og *max* til 2 m. Vi har dog i løbet af de forskellige ROB journaler kunne se, at en for lav *min* til tider resulterede i falske positiver fra sensoren og at en for høj *max* kunne medtage forhindringer der var for langt fremme.

Udover de indlysende parametre[7] såsom *RobotRadius*, *SafetyDistance* og *MinTurningRadius*, er yderligere tre parametre specielt interessante. *TargetDirectionWeight*, *CurrentDirectionWeight* samt *PreviousDirectionWeight* bliver alle tre benyttet i en beregning af omkostningerne af forskellige mulige retninger, således at der returneres den bedst mulige retning, med den mindst mulige omkostning.

Der oprettes først et *robotics.VectorFieldHistogram* objekt hvortil de relevante parametre sættes. Dernæst et while loop der kontinuert samler data, udregner retningen og driver robotten vha. controller outputs. Hvis while loopet vurderer, at distanceToGoal er større end *goalRadius* (0.1 m) indlæses laser scan dataet via en ROS subscriber, og VFH objektet kaldes med range, angles og omega parametrene.
Da der opstod problemer med robotten, da den ville tilbage på dens oprindelige path inden den havde passeret forhindringer, oprettes der en if-sætning der tjekker om den er i færd med at undgå et forhindring. Er dette tilfældet, inkrementeres der til 20, hvorefter *isAvoidingObs* sættes til false og gøres dermed klar til næste gang. Der tjekkes desuden med en *elseif* om hvorvidt *distanceToGoal* < 2. Er dette tilfældet, benytter vi ikke VFH.

Er *isAvoidingObs* false og er *steerDirIsOk* samt at *steerDir* er forskellige fra omega, sætter vi *isAvoidingObs* til true, så den ikke afbryder sin rute omkring forhindringen for tidligt, for i stedet at vende tilbage til sin oprindelige path. I det modsatte tilfælde, hvor *steerDir* er ens med omega, fortsætter robotten blot.
Den sidste else, er til hvis steering direction er invalid. I så fald stopper den op og leder efter mulige retninger.
Derefter sendes en kør besked til robotten ved at benytte vores controller outputs. Vi extracter dernæst location til benyttelse i vores live plot, genberegner *distanceToGoal* og loopet fortsætter indtil vi er indenfor 0.1 m af vores mål.

[6] VFH: https://se.mathworks.com/help/robotics/ref/robotics.vectorfieldhistogram-system-object.html
[7] Simple parametre: https://se.mathworks.com/help/robotics/ug/vector-field-histograms.html

Utilities

I dette afsnit præsenteres en række funktionalitet udviklet i dette projekt, der benyttes som hjælpeværktøjer.

Live mapping

Da der i projektet ikke er opnået at få localization til at virke, var det svært for gruppen af finde ud af om robotten vidste hvor den var. Derfor blev der brugt robottens odometri-data til at trække den position ud med *getRobotPose*. Dette bliver brugt af robotten til at beregne hvor den er og hvor den skal hen. Det samme kunne bruges til at visualisere robottens teoretiske position på et kort, så der kunne fås et overblik over robottens placering. Dette blev implementeret ved at plotte robottens position ved hver cyklus. Cyklussen er hvor robotten udregner dens position samt sender kommandoer afsted for at køre. Dette gøres hele tiden, og man får derfor live mapping.

Bumper collision

Som en udvidelse til *obstacleAvoidance* blev der implementeret funktionalitet ved events fra robottens bumpers. Først blev der abonneret på bumper-events med en funktion, som først afspiller en lyd (se næste afsnit), og derefter sætter to globale variable: *bumperHit* og *bumberNumber*. *bumperHit* er boolean, som der tjekkes på i hver cyklus, for at se om robotten er kørt ind i noget. *bumperNumber* repræsenterer den bumper på robotten som er blevet aktiveret. Dette kan være højre, midten eller venstre. Når der kommer et event og boolean'en sættes til true, bliver der i den kommende cyklus kørt en funktion, som skal få robotten til at bakke og dreje modsat vej end den bumper, som blev ramt.

Resultater

Funktionen *driveDistance* er testet med Matlab scriptet *testDriveDistance* der er sat til at køre en halv meter frem og tilbage. Den kørte afstand blev målt med målebånd til at være 50 cm.

rotateToAngle er testet med scriptet *testRotateToAngle* den starter med at dreje robotten til 0 grader i forhold til dens startposition, så 180 grader og så tilbage til 0 grader, hvilket visuelt så rigtigt ud.

Funktionen *rotate* testes med *testRotate* som skiftevis drejer mod venstre og højre i henholdsvis 2,5 sek og 2,6 sek for at komme så tæt på at dreje 90 grader som muligt, men *rotate* viste sig at have et slør på omkring 5 grader. Ligesom det ses at der skal drejes mod højre i længere tid end mod venstre for at ramme 0 grader.

Funktionen *lookStrightAtWall* er testet visuelt og har et lille men observerbart offset, hvilket koden også giver udtryk for - det vurderes dog at være tilfredsstillende.

ObjectRecognition er testet med de udleverede objekter, men det ROS topic som vi modtager data fra viser sig at være ustabilt og nogle gange returnerer et billede hvor farverne er forkerte, hvorfor det er umuligt at detektere den grønne cirkel. Men vi fandt ud af at det kunne afhjælpes ved at der bliver taget et billede tidligere i programmet for at få billedet tilbage i de rigtig farver.

findGreenDot kræver at robotten er placeret ud for væggen når funktionen køres, ellers kan robotten ikke se væggen når den prøver at køre 40 cm foran væggen. Derudover er funktionen testet mange gange og fejler på samme måde som beskrevet i *objectRecognition*, hvis farverne er forvrængede.

Vores obstacle avoidance funktion blev testet gentagne gange ved at placere en stor papkasse midt i robottens path. Vi oplevede tidligt i processen problemer med VFH funktionen, idet robotten til tider ville tilbage på dens oprindelige path inden den havde passeret forhindringen. Efter at have mindsket *vfh.DistanceLimits* parameteren en anelse, blev funktionen mere stabil, men en definitiv løsning på problemet blev først implementeret da *isAvoidingObs* med 20 steps, samt bumper events blev taget i brug.

Vi fandt desuden for VFH, at *TargetDirectionWeight* skulle være højere end summen af *CurrentDirectionWeight* og *PreviousDirectionWeight*, for at robotten fortsat ville følge vores ønskede retning. I samme ombæring fandt vi, at en højere *PreviousDirectionWeight* hjalp til at gøre den undvigende path mere flydende. Det var derfor en afvejning af en højere *PreviousDirectionWeight* og en lidt lavere *CurrentDirectionWeight,* men hvor summen af disse stadig var lavere end *TargetDirectionWeight*.

Diskussion

Der burde laves en funktion der detekterer om farverne på billedet er rigtige eller forvrængede for derefter at håndtere dette. Men denne fejl afhjalp vi ved bare at tage et billede i starten af programmet og så var farverne gode igen. Vi har prøvet flere forskellige ROS topics såsom ”/camera/rgb/image_raw” og flere andre, med samme resultat.

Vi endte ud med en fungerende obstacle avoidance function, men denne skulle ændres og tilpasses en hel del, fra den givne VFH dokumentation. Dette skyldtes at vi i ROS dokumentationen for Pure Pursuit[8] opdagede, at man kunne generere et *TargetDir* output fra Pure Pursuit der kunne benyttes direkte som input i *TargetDir* porten for VFH blokken. Dette viste sig desværre ikke at fungere, hvorfor en anden måde blev implementeret.

Grundet en mangel på en lokaliseringsalgoritme var robottens performance lettere ustabil. Robotten blev placeret på en given lokation, fik hardcoded start og slut koordinater og ved hjælp af PRM til path planning, forsøgte den at køre ruten.

Ved konstant at trække robottens position ud, kunne dens opfattede position plottes live. Vi kunne derfor til tider visuelt se, hvordan robotten opfattede dens path som korrekt, men hele tiden f.eks. søgte ind i væggen for at finde tilbage på dens “rette” placering. Eventuelle forhindringer og sammenstød hvor f.eks. robottens hjul fortsatte med at køre i få sekunder efter impact, bidrog til en ændret opfattelse af dens placering. Vi var derfor stærkt afhængige

[8] Pure Pursuit dokumentation: https://se.mathworks.com/help/robotics/ref/purepursuit.html

af bumper events og obstacle avoidance, for at få robotten på rette kurs, omend det langtfra var nok.

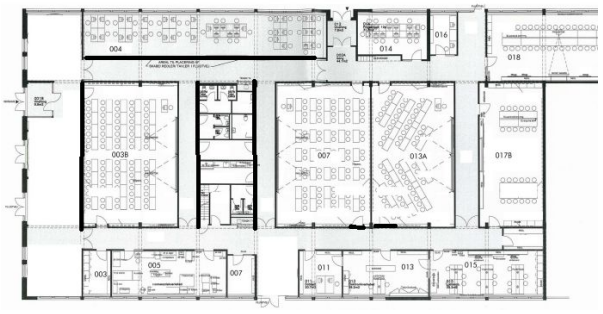
Et particle filter blev forsøgt implementeret efter eksemplet givet i uge 9, men grundet tidspres og mangel på fremskridt med denne del, blev featuren nedprioriteret.

At køre robotten uden en fungerende lokaliseringsalgoritme betyder derfor også, at nøjagtigheden af vores startplacering af robottens i forhold til dens start-koordinater har været utrolig vigtigt. Vi havde et gennemløb (vedhæftet som video), hvor robotten gennemførte 90% af ruten uden problemer. Det er derfor muligt at køre robotten uden en lokaliseringsalgoritme, men udefrakommende faktorer såsom personer i bevægelse, objekter samt præcisionen af vores placering af robotten, vil spille ind, idet robotten ikke kan korrigere for dette da den mangler en forståelse for dens placering i rummet.

PRM motion-planning algoritmen viste sig at indeholde nogle parametre der skulle tweakes lidt, idet antallet af noder samt connection distance spiller en relativt stor rolle i performance.

I de tidlige iterationer benyttede vi et relativt højt antal noder. Vi opdagede hurtigt, at dette øgede antallet af mulige forbindelser og påvirkede kompleksiteten. Desværre påvirkede det også i høj grad den tid det tog at beregne PRM.

Det blev derfor besluttet, at ændre i kortet, således at det overflødige blev fjernet: Alle irrelevante lokaler, døre, gange og lignende. Dette bevirkede, at der i samspil med vores Pure Pursuit kun blev genereret relevante ruter, og at vi kunne nedsætte antallet af noder uden at påvirke resultatet, mens det samtidigt boostede vores computation time.



Figur 3, Shannon før

Figur 6, Shannon efter

Konklusion

Det er lykkedes os at få robotten til at køre fra punkt A til B, udføre billedgenkendelse og finde den grønne cirkel og næsten køre til punkt C. Dette fungerer desværre uden Kalman eller particle filter, da vi ikke fik dette til at virke. Ruten er fundet ved hjælp af PRM og på ruten udfører robotten obstacles avoidance med VFH og reagerer på bumber events. Dermed er det altså lykkedes os at opfylde de fleste funktionelle krav til projektet, samt mange af de generelle krav. Arbejdet med ROS og robotten generelt har givet gruppen et godt indblik i hvordan autonome robotter fungerer. Gruppen har i projektet selv mærket

hvordan de fysiske omgivelser kan give problemer for gennemførelsen af en korrekt gennemkørsel. Derudover har gruppen set hvordan robottens forskellige hardware-dele kan være svære at arbejde med. Dette er kommet til syne under arbejdet med robottens RGB kamera, som gav ustabile billeder. Ydermere har gruppen lidt af manglende dokumentation til ROS, når der skulle søges information på internettet. Dette har været problematisk i udviklingen af problematisk. Samlet set har projektet været positivt for gruppen, selvom der ikke blev implementeret en lokaliseringsalgoritme.