SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-104376-98355

POROVNANIE SLAM ALGORITMOV S VYUŽITÍM ROS

DIPLOMOVÁ PRÁCA

Bratislava 2023 **Tomáš Nyiri**

SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-104376-98355

POROVNANIE SLAM ALGORITMOV S VYUŽITÍM ROS

DIPLOMOVÁ PRÁCA

Študijný program:	Robotika a kybernetika
Študijný odbor:	Kybernetika
Školiace pracovisko:	Ústav robotiky a kybernetiky
Vedúci záverečnej práce/školiteľ:	Ing. Martin Dekan, PhD.

Bratislava 2023 **Tomáš Nyiri**

STU FEI

ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Tomáš Nyiri**

ID študenta: 98355

Študijný program: robotika a kybernetika

Študijný odbor: kybernetika

Vedúci práce: Ing. Martin Dekan, PhD.

Miesto vypracovania: Ústav robotiky a kybernetiky

Názov práce: **Porovnanie SLAM algoritmov s využitím ROS**

Jazyk, v ktorom sa práca vypracuje : slovenský jazyk

Špecifikácia zadania:

Vývoj v oblasti mobilnej robotiky je značne zameraný na zlepšovanie fungovania algoritmov typu SLAM. Preto je potrebná pravidelná analýza a porovnanie novovzniknutých algoritmov, aby bolo možné vyhodnotiť, ktoré SLAM algoritmy sú vhodné na aký typ prostredia a robota. SLAM algoritmy môžu na svoje fungovanie využiť rôzne typy snímačov, táto diplomová práca je zameraná na tie algoritmy, ktoré sú schopné využiť 2D laserové diaľkomery. Diplomová práca bude vypracovaná v spolupráci s firmou Photoneo s.r.o.

Úlohy:

1. Naštudujte si fungovanie algoritmov typu SLAM v robotickom operačnom systéme.
2. Analyzujte aspoň tri SLAM algoritmy.
3. Navrhnite experimentálne overenie porovnávaných algoritmov.
4. Vyhodnoťte navrhnuté riešenie.
5. Vypracujte dokumentáciu.

Termín odovzdania diplomovej práce: 12. 05. 2023
Dátum schválenia zadania diplomovej práce: 20. 12. 2022
Zadanie diplomovej práce schválil: prof. Ing. Jarmila Pavlovičová, PhD.

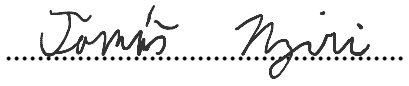
Tomáš Nyiri
študent

prof. Ing. Jarmila Pavlovičová, PhD.
vedúci pracoviska

doc. Ing. Eva Miklovičová, PhD.
garantka študijného programu

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú prácu s názvom Porovnanie SLAM algoritmov s využitím ROS vypracoval samostatne pod vedením Ing. Martina Dekana, PhD. a že som uviedol všetku použitú literatúru.



podpis autora

Poďakovanie

V prvom rade si dovoľujem poďakovať vedúcemu práce Ing. Martinovi Dekanovi, PhD za jeho odborné vedenie, cenné rady a neoceniteľnú pomoc, ktorú som zužitkoval počas príprav tejto diplomovej práce. Vaše múdre vedenie a pripomienky boli pre mňa inšpiráciou a privádzali ma k dosiahnutiu lepších výsledkov. Taktiež by som chcel poďakovať svojim rodičom, za ich podporu, ktorú mi preukazovali počas celého štúdia. Bez vašej podpory by som tento míľnik nedosiahol.

ABSTRAKT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Robotika a kybernetika
Autor:	Tomáš Nyiri
Diplomová práca:	Porovnanie SLAM algoritmov s využitím ROS
Vedúci záverečnej práce:	Ing. Martin Dekan, PhD.
Konzultant:	
Miesto a rok predloženia práce:	Bratislava 2023

Táto práca sa zaoberá porovnaním algoritmov SLAM s využitím systému ROS. SLAM sú dôležité súčasti robotiky, umožňujúce robotom vytvárať mapu prostredia a súčasne sa v nej lokalizovať. Cieľom tejto práce je analyzovať dostupné SLAM algoritmy a porovnať ich v testoch ako presnosť tvorby mapy, robustnosti voči chybám a veľkosti výpočtového výkonu. Experimenty budú vykonané v odlišných testovacích scenároch s rôznou dynamikou robota. Výsledky našej práce budú prezentované s dôrazom na výhody a nevýhody jednotlivých SLAM algoritmov, pričom na základe nadobudnutých výsledkov bude možné ďalej odporučiť výber vhodného SLAM algoritmu. Lepšie objasníme prácu so SLAM algoritmy v systéme ROS a dodáme prehľad o ich využiteľnosti pre rôzne aplikácie.

Kľúčové slová: ROS1, GMapping, Google Cartographer, SlamToolbox, HectorSlam, porovnanie

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND IFNORMATION TECHNOLOGY

Study program :	Robotics and cybernetics
Author:	Tomáš Nyiri
Thesis:	Comparison of SLAM Algorithms in ROS
Supervisor:	Ing. Martin Dekan, PhD.
Consultant:	
Place and year of submission of thesis:	Bratislava 2023

The Master thesis focuses on the issue of comparing SLAM algorithms using the ROS system. SLAM (Simultaneous Localization and Mapping) is an important aspect of robotics that allows robots to create a map of the environment while simultaneously localizing themselves within it. The aim of this work is to analyze available SLAM algorithms and compare them in tests such as map creation accuracy, robustness against errors, and computational performance. The experiments will be conducted in various testing scenarios with different robot dynamics. The results of our work will be presented, highlighting the advantages and disadvantages of each SLAM algorithm. Based on the acquired results, it will be possible to make recommendations for selecting a suitable SLAM algorithm. We will provide a better understanding of working with SLAM algorithms in the ROS system and provide an overview of their applicability for various applications.

Keywords: ROS1, GMapping, Google Cartographer, SlamToolbox, HectorSlam, compare

Obsah

Zoznam obrázkov a tabuliek

Zoznam použitých skratiek

ROS Robot Operating System

SLAM Simultaneous Localization And Mapping

GPS Globálny Lokalizačný systém

LiDAR Light Detection And Ranging

Úvod

Simultánna lokalizácia a mapovanie v skratke SLAM, je kľúčová technológia v oblasti robotiky a autonómnych systémov, ktorá umožňuje autonómnym vozidlám alebo robotom navigovať a mapovať neznáme okolie, v ktorom sa nachádzajú v reálnom čase. Robot Operating System v skratke ROS, je populárna open-source platforma, ktorá má široké využitie v oblasti robotiky na vývoj a prototypovanie nových robotických systémov. S narastajúcim dopytom po autonómnych vozidlách, dronoch a mobilných robotoch nasadených v rôznych aplikáciách sa zvyšuje potreba pre presnejšie algoritmy SLAM, ktoré je možné integrovať v systéme ROS.

Algoritmy SLAM sú navrhnuté pre mobilné vozidlá a roboty, ktoré vyžadujú znalosť odhadu svojej pozície (polohy a orientácie) a súčasne dokážu skonštruovať mapu prostredia. Aktuálne existuje mnoho typov SLAM algoritmov, pričom každý z nich má svoje slabé a silné stránky. Tieto algoritmy sa líšia v prvom rade v reprezentácii mapy, a teda či vytvárajú 2D alebo 3D mapu, následne ich môžeme odlíšiť napríklad pomocou extrakcií čŕt. Voľba toho správneho SLAM algoritmu je náročná a záleží v akom prostredí sa bude jeho aplikácia využívať, preto treba dôkladne zvážiť faktory akými sú presnosť algoritmu, výpočtová náročnosť, odolnosť voči šumu dostávaných zo senzorov a výkon algoritmu v reálnom čase.

V tejto práci sa budeme snažiť objektívne ohodnotiť rôzne typy 2D SLAM algoritmov s implementáciou v systéme ROS ako podkladovej platformy. Hlavným cieľom je poskytnúť komplexnú analýzu najmenej troch SLAM algoritmov v rôznych scenároch. Zameriame sa na SLAM algoritmy, ktoré je možné implementovať do systému ROS a sú používané v komunite robotiky, ako napríklad: Gmapping, Cartographer a HectorSLam.

Naša práca bude organizovaná do nasledujúcich šiestich bodov. V prvom bode si podrobnejšie rozdelíme a opíšeme jednotlivé SLAM algoritmy. Pokúsime sa zvýrazniť ich kľúčové vlastnosti a využitie v jednotlivých aplikáciách.

V druhom bode opíšeme systém ROS a jeho najdôležitejšie časti, ktoré je nevyhnutné poznať z dôvodu implementácie SLAM algoritmov do tohto prostredia. Popri tom si priblížime hardvér, ktorý bol použitý pri získavaní dát potrebných na tvorbu mapy. Takisto si vyzdvihneme jeho výhody a nevýhody, a priblížime technické špecifikácie jednotlivých komponentov.

Ďalšia časť bude zahŕňať detaily o experimentálnom zbere dát a hodnotiacich metrikách použitých na porovnanie výkonu rôznych SLAM algoritmov. Popíšeme rôzne scenáre, v ktorých sme SLAM algoritmy testovali, vrátane rôznych trajektórií pohybu a odlišných vnútorných miestností, ktoré sú zložité pre mapovanie.

Následne predstavíme výsledky každého testovacieho scenára, ktorým sme sa snažili overiť funkčnosť rôznych SLAM algoritmov. Predstavíme si kvantitatívne metriky akými sú presnosť odhadu pozície a presnosť mapovania, ako aj kvalitatívne metriky akými sú odolnosť voči šumu zo senzorov a schopnosť ich spracovávať, no taktiež zvládnutie mapovania dynamického prostredia. Rovnako budeme diskutovať o silných stránkach a obmedzeniach jednotlivých SLAM algoritmov vďaka dosiahnutým výsledkom.

Nakoniec prácu uzavrieme zhrnutím našich zistení a vynesieme kľúčové poznatky získané z hodnotenia algoritmov SLAM v systéme ROS, pričom pevne veríme, že naša štúdia prinesie cenné poznatky v oblasti robotiky a autonómnych systémov.

1 Charakteristika algoritmov SLAM

Simultánna lokalizácia a mapovanie je viacero spojených algoritmov umožňujúce dronom alebo mobilným robotom tvoriť mapu z neznámeho prostredia a následne sa vedieť lokalizovať na vytvorenej mape. Pomocou týchto algoritmov vieme vytvorené mapy využiť na vykonanie rôznorodých úloh, pre príklad si môžeme uviesť naplánovanie trajektórie pomedzi prekážky.

SLAM algoritmy sú jedným z hlavných prvkov v mobilnej robotike, pomocou ktorých sa dáva robotovi znalosť o jeho aktuálnej polohe a ďalšom kroku v akomkoľvek prostredí, bez akéhokoľvek externého zásahu, aby sa mobilný robot vedel svojpomocne navigovať v priestore. Cieľom je dosiahnuť taký stav, aby sa mobilné roboty vedeli bezpečne, bez kolízie orientovať a pohybovať v prostredí. SLAM je v konečnom dôsledku podriadený dátam, ktoré dostáva zo senzorov. Ak budeme zo senzorov dostávať príliš zašumené a nepresné dáta, výsledná mapa alebo lokalizácia nebudú dostatočne reálne opisovať prostredie a môže prísť k nežiadaným situáciám ako môže byť kolízia s prekážkou alebo nepresne zostrojená mapa.

Dnes sa SLAM využíva v rôznych aplikáciách a prostrediach interiérových alebo exteriérových. SLAM vie spracovávať údaje z 3D a 2D snímačov a tvoriť z nich príslušné mapy prostredia avšak, my v našej práci sa zameriame na SLAM-y zaoberajúce sa 2D mapovaním vo vnútornom priestore.

V interiérových aplikáciách sa mobilné zariadenia vedia lokalizovať za pomoci odometrie, externých kamier, WIFI alebo iBeacon, následne sa lokalizovať a presúvať v prostredí bez kolízie. Avšak, v externých aplikáciách sa mobilné zariadenia musia spoľahnúť na GPS signál, ktorý nemusí byť vždy presný na lokalizáciu, a preto vedia využiť taktiež vizuálne snímače pre reaktívnu navigáciu, pre dosiahnutie presne požadovaných súradníc. V našej práci využijeme pre lokalizovanie dáta z odometrie, prostredníctvom ktorých bude algoritmus vedieť lokalizovať robota.

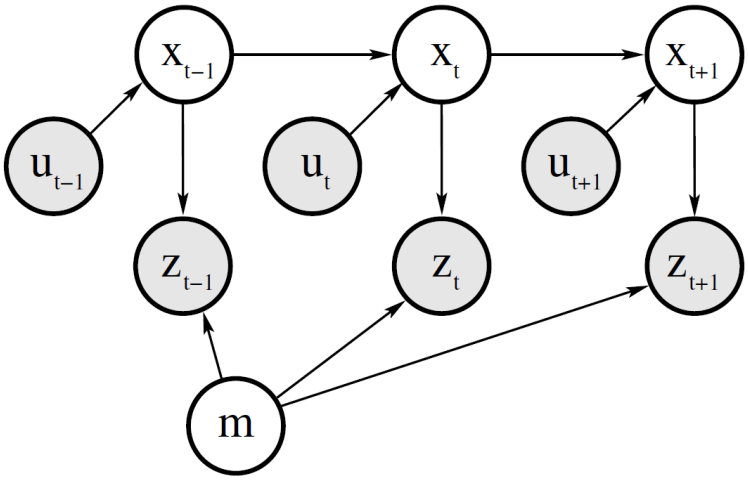
Pre tvorbu mapy za pomoci SLAM algoritmov sa využívajú 2D alebo 3D snímače nazývajúce sa Light Detection And Ranging v skratke LiDAR. Ich veľkou nevýhodou sú oblasti, ktoré zle odrážajú svetlo, kvôli čomu ich daný snímač nevie spracovať, pretože nezachytí prekážku. Typickým príkladom môže byť dlhá chodba bez vyznačených rohov so sklenenými prvkami, pričom LiDAR môže odignorovať

spracovanie týchto sklených prvkov. Ako sme už spomenuli, my v našej práci budeme využívať 2D LiDAR od značky HOKUYO.

V budúcnosti sa očakáva, že SLAM algoritmy budú zohrávať ešte väčšiu úlohu v nových oblastiach, ako sú inteligentné mestá, inteligentné domy a rôzne aplikácie Internetu vecí (IoT), pretože SLAM môže umožniť autonómnym vozidlám navigovať sa v zložitých mestských prostrediach, a pritom zabezpečiť služby založené na lokalite v aplikáciách IoT. Avšak, stále existuje niekoľko výziev v tejto oblasti, ako je presnosť, odolnosť voči chybám a výkon v reálnom čase. Tieto problémy je potrebné riešiť pre ďalší pokrok v tejto oblasti. SLAM je základnou technológiou, ktorá umožňuje robotom a autonómnym systémom vnímať a navigovať sa cez prostredie v reálnom čase a aplikácie, pričom využitie týchto algoritmov v aplikáciách sa neustále rozširuje a postupom času bude len narastať.

1.1 SLAM

Hlavná štruktúra SLAM-u zobrazená na Obr. 1 nám stvárňuje premenné, ktoré sú potrebné pre vytvorenie mapy. Na to aby vedel robot odhadnúť svoju pozíciu na mape a vytvoriť samotnú mapu z neznámeho prostredia, je potrebné iteratívnym spôsobom odhadovať premenné x_t a m.



Obr. 1 Grafická Štruktúra SLAM [28]

Následne si rozoberieme všetky premenné, ktoré sa nachádzajú na Obr. 1:

- x_t
- u_t
- z_t
- m

Premenná u_t reprezentuje relatívny pohyb v čase, pričom časové okno je zobrazené v rozmedzí t-1 až t+1, daný proces môžeme nazvať odometriou. Odometriu získavame zo snímačov, ktoré sú napojené priamo na kolesá mobilného robota, alebo na rotor motora, kde pomocou počtu impulzov obdržaných z optického rotačného kódera vieme zistiť, o aký úsek sa zariadenie pohlo a o aký uhol natočilo.

Premenná x_t reprezentuje odhad aktuálnych pozícií robota v čase, pričom index t nám udáva čas danej polohy robota. Jednotlivé polohy majú medzi sebou vzťahy, ktoré na seba nadväzujú, a teda v čase vytvárajú pole súradníc, po ktorých sa robot pohyboval.

Premenná z_t reprezentuje meranie bodov z vizuálnych snímačov, v našom prípade to bude z LiDAR-u. Dáta medzi z_t a x_t musia byť medzi sebou nadviazané, aby sme vedeli správne určiť meranie z LiDAR-u s aktuálnou polohou robota.

Samotná premenná m reprezentuje vytvorenú mapu z nameraných a poskytnutých bodov z premennej z_t, ktorá je následne naviazaná na polohu robota, a teda premennú x_t.[28]

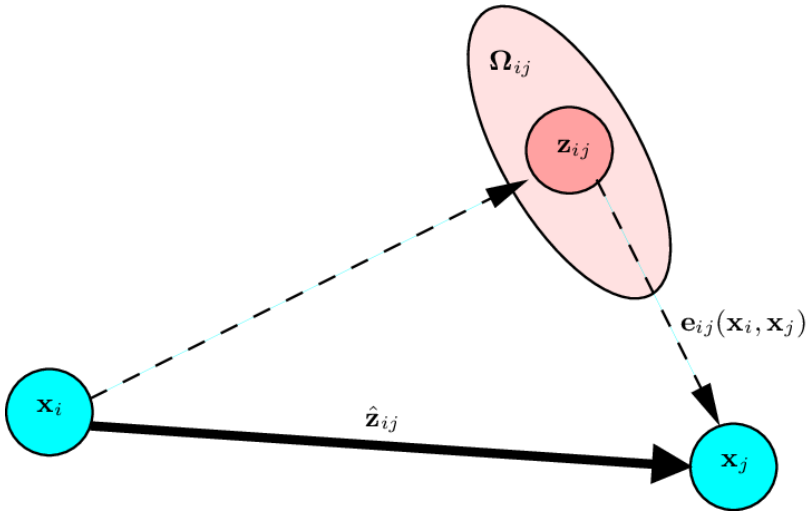
1.2 Rozdelenie SLAM metód

V tejto kapitole podrobnejšie opíšeme typy SLAM algoritmov, ktoré sú aktívne používané a zostavujú aktívnu oblasť výskumu, pričom niektoré z nich možno považovať aj ako kombináciu viacerých systémov zároveň.

1.2.1 Graf SLAM

Je SLAM metóda založená na grafe, pomocou ktorej sa tvorí zjednodušený problém odhadu abstrahovaním dát zo senzorov. Následne sú dáta zo senzorov nahradené uzlami na grafe, ktoré je možné predpokladať za „virtuálne merania."

Na obrázku Obr. 2 máme zobrazené dva uzly x_i a x_j spojené hranou z_{ij}. Z relatívnej polohy týchto dvoch uzlov je možné vypočítať očakávané meranie z_{ij}. Chyba, ktorú predstavuje e_{ij}, závisí od posunu medzi očakávaným a skutočným meraním, ktoré môžu skresľovať snímače a senzory. Výsledok merania plne charakterizuje svoju chybu e_{ij} a informačnú maticu Ω_{ij}, ktorá zodpovedá za jej presnosť.



Obr. 2 Graf SLAM

Pre vytvorenie kompletnej mapy na báze Graf SLAM-u, je potrebné nájsť a pospájať všetky konzistentné uzly. Uzly v sebe nesú dáta o polohe robota a dáta z LiDAR-U. Jednotlivé uzly sa medzi sebou spájajú tzv. hranami, ktoré popisujú vzájomnú polohu medzi uzlami v čase. Ak sa budú spájať iba tie uzly, ktoré na seba nadväzujú a dané uzly budú mať konzistentné merania, budeme minimalizovať tvorbu chýb na minimum čím sa vyhneme prípadným nedostatkom pri tvorení mapy. Práve z hľadiska presnosti Grafových SLAM-ov, patria aktuálne k najmodernejším algoritmom pre tvorbu mapy.

Obr. 3 Vizualizácia Graf SLAM-u

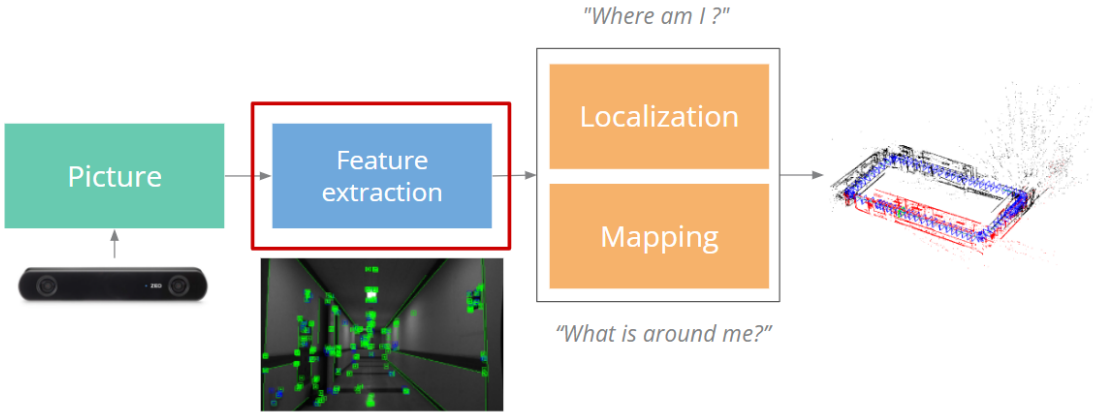
Na to aby sme dostali čo najpresnejší odhad mapy za pomoci odometrie a dát z diaľkového laseru, je potrebné optimalizovať rozloženie uzlov. Najskôr je potrebné si definovať funkciu $e_{ij}(x_i,x_j)$.

$$e_{ij}(x_i, x_j) = z_{ij} - \hat{z}_{ij}(x_i, x_j) \qquad 1$$

Funkcia vyobrazená na rovnici 1 nám vraví o celkovej chybe medzi z_{ij} a $\hat{z}_{ij}$ a jej vzájomné prepojenie si môžeme všimnúť na *Obr. 2*. Pravdivosť tvorenej mapy sa zväčšuje, pokiaľ štvorcová chyba medzi parametrami z_{ij} a $\hat{z}_{ij}$ klesá, a teda optimálne riešenie je možné nájsť pomocou aproximačnej metódy najmenších štvorcov. [24]

1.2.2 Vizuálny SLAM

Vizuálny SLAM pracuje s dátami z kamery alebo iných vizuálnych 3D senzorov. Pričom tieto dáta zo senzorov využíva na vykonávanie funkcií lokalizácie a mapovania. Väčšina vizuálnych systémov SLAM fungujú na rovnakom princípe. Sledujú body cez série snímok z kamery a pomocou triangulácie ich 3D polohy používajú tieto informácie na aproximáciu kamery. Na rozdiel od iných SLAM-ov, vizuálny SLAM vie pracovať iba s jednou 3D kamerou. Chyby a nezrovnalosti medzi skutočnými a zaznamenanými bodmi sú bežným problémom všetkých vizuálnych systémov SLAM. Tieto problémy sa dajú eliminovať na algoritmickej rovine, a to obmedzením šumu. Keďže SLAM musí pracovať v reálnom čase, údaje o polohe a mapové údaje sa často aktualizujú nezávisle od seba, aby sa zrýchlilo spracovanie pred konečným spojením.



Obr. 4 Vizuálny SLAM

Z komerčného hľadiska je vizuálny SLAM stále v ranom štádiu vývoja. Hoci má v mnohých bodoch veľký potenciál, tieto algoritmy majú pred sebou ešte dlhú cestu. Vzhľadom na to, budú hrať veľkú a významnú úlohu aplikácie rozšírenej reality, kde sa bude vizuálny SLAM využívať, pretože zatiaľ iba vizuálna technológia SLAM môže poskytnúť danú úroveň presnosti, pokiaľ ide o mapovanie fyzického okolia z kamery, ktoré je potrebné na presné premietanie virtuálnych obrazov do skutočného sveta, a možno aj opačne.

1.2.3 FastSLAM

FastSLAM algoritmy vieme rozdeliť na dve triedy. FastSLAM 1.0, ktorý je zastaralejší, a zároveň jednoduchší na implementáciu a použitie, a FastSLAM 2.0, ktorý je síce novší, ale jeho implementácia je náročnejšia oproti predošlej verzii.

Avšak, obe verzie FastSLAM tvoria dráhu pomocou filtra častíc, podobne ako to robia aj iné SLAM algoritmy v mobilnej robotike. Časticový filter má tú výhodu, že aj pri dlhej ceste mobilného robota, množstvo výpočtov potrebných pre každý jeden prírastok, zostáva konštantné. Keďže FastSLAM je nízko rozmerný, každý rozšírený Kalmanov filter odhaduje jeden orientačný uzol, čoho výsledkom je, že každý takýto uzol má jedinečný Kalmanov filter, pričom celkovo budú existovať jedinečné Kalmanove filtre pre každý objekt mapy a pre každý orientačný bod.

Ak by sme FastSLAM mali zapísať do rovnice, vyzerala by nasledovne:

$$S_t^m = \langle s^{t,m}, \mu_{1,t}^m, \sum_{1,t}^m, \dots, \mu_{N,t}^m, \sum_{N,t}^m \rangle \qquad 2$$

kde m označuje index jednotlivej vzorky, s je odhad dráhy a $\mu_{N,t}^m, \sum_{N,t}^m$ sú stredná hodnota a rozptyl Gaussovej hodnoty reprezentujúci n-tú polohu prvku, pričom všetky tieto premenné tvoria spolu časticu S, ktorých je v celkom m, v priestore mapovanom pomocou FastSLAM-u. FastSLAM v sebe zahŕňa trojkrokový filter pre odfiltrovanie zle mapovaných súradníc a optimálnejšie zostrojenie mapy. Kroky sú nasledujúce :

- Vzorkovanie . FastSLAM používa na nájdenie novej pozície predchádzajúcu pozíciu, pričom ich môžeme označiť ako s_t pre novú pozíciu a s_{t-1} pre predchádzajúcu pozíciu robota.

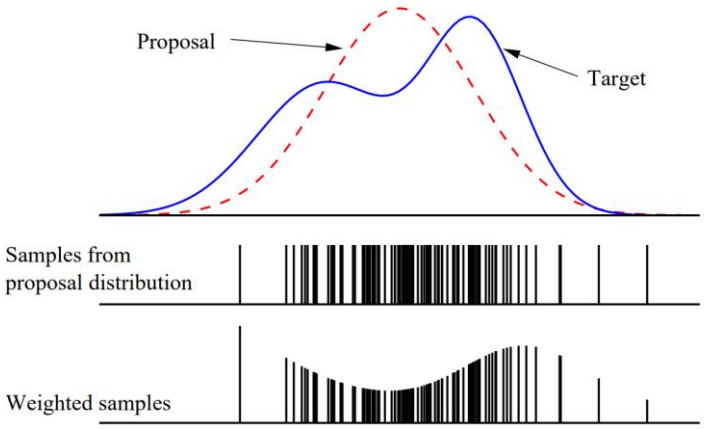
$$S_t^m \sim p(s_t | s_{t-1}^m, u_t) \qquad 3$$

Rovnica 3 popisuje odhad polohy robota v čase t-1, ktorý sa nachádza v m-tej mriežke mapy, pričom výsledok sa pridá k dočasnej sade dát, spolu s dráhou z predchádzajúcich polôh. Táto operácia si vyžaduje nezávislý čas od veľkosti mapy. Tento krok je graficky znázornený na obrázku nižšie, pričom ilustruje súpravu rôznych polôh z jednej počiatočnej polohy.

Obr. 5 Grafické znázornenie kroku je vo FastSLAM

- Váženie vzoriek. V tomto kroku je potrebné aktualizovať priestor voči orientačným odhadom z dát dostávaných zo senzorov. Taktiež, znova je potrebné vedieť strednú hodnotu a rozptyl Gaussovej hodnoty oproti orientačným bodom. Podľa daného rozptylu dostane každá častica svoju hodnotu, tkz. váhu. Rozdelenie váh je dané, ako je častica presná s pozíciou robota.

  Grafické preukázanie funkčnosti si zobrazíme na obrázku nižšie, kde si môžeme všimnúť tri paralelné grafy. Na prvom z nich vidíme porovnanie častíc medzi návrhom a cieľom, kam by sa mal mobilný robot posunúť. V druhom grafe vidíme jednotlivé vzorky z merania a v poslednom grafe si môžeme všimnúť ich váhu, ktorú sme vypočítali pomocou rozptylu Gaussovej hodnoty.



Obr. 6 FastSLAM odhad pozorovaného orientačného bodu

Kroky 1 a 2 sa opakujú m-krát, a ich rezultát je dočasný súbor M častíc obsahujúcich jednotlivé merania a polohy robota v čase.

- Prevzorkovanie. Počas prvých dvoch krokov sa môže udiať, že sa v dočasnom súbore objavia vzorky, ktoré nezodpovedajú presným údajom a obsahujú zašumené, alebo nie presné údaje o polohe robota. Následne je potrebné tieto chybové častice prevzorkovávať, aby nedošlo ku kolapsu algoritmu, a tým pádom k zle namapovanému okoliu. Pri prevzorkovaní dochádza k stavu, kedy systém vyberá z dočasných súborov N častíc, pomocou ktorých sa snaží vyskladať nový systém. Voľba nových častíc spočíva v ich váhe z druhého kroku, takže čím vyššia je váha častice, tým lepší by mal byť odhad presnosti robota. Dané údaje o odhadovanej polohe sa môžu vyťahovať opakovane pre znovu pretvorenie systému, aby sme mohli dostať lepší systém a nezostali v jednej množine bodov.[26] [27]

1.3 Kalmanov filter

Kalmanov filter je rekurzívny algoritmus, pomocou ktorého odhadujeme stav systému na základe viacerých meraní. Je značne používaný v rôznych oblastiach, akými môžu byť automatizácia, rôzne ekonomické a grafické aplikácie. Stav systému je reprezentovaný vektorom označujúcim x, ktorý obsahuje premenné opisujúce správanie stavov systému a merania sú reprezentované vektormi. Nasleduje stavová rovnica Kalmanovho filtra :

$$x_{k+1} = F_{k+1,k}x_k + q_k$$

4

kde q_k je procesný šum, x_k nám udáva stav v prechodovej matice označenej ako F_{+k1} a index k udáva postupnosť v čase. Následne si musíme definovať rovnicu s názvom rovnica pozorovaní:

$$y_k = H_k x_k + r_k$$

5

kde r_k taktiež udáva šum, avšak, je rozdielny ako v rovnici č. 4. H_k nám reprezentuje maticu pozorovaní a x_k stav prechodovej matice. Y_k je pozorovanie a index k udáva čas, v ktorom bolo pozorovanie vykonávané.

Avšak, základným problémom pri Kalmanovom filtri, je vyriešenie oboch rovníc súčasne, pričom treba započítať všetky pozorované údaje z vektorov y_k, aby sme dokázali odhadnúť stred minimálnej kvadratickej odchýlky pre každý stav v čase. Kalmanov filter pracuje v opakovateľnej slučke, vykonávajúcej dva body. Prvý bod vypočítava predikcie a kovariancie chyby a druhý bod sa nazýva aktualizácia pozorovania, pričom v danom kroku sa koriguje odhad systému získaný v prvom kroku, na základe nadobudnutých pozorovaní z vektoru y_k. Prvý bod vyzerá nasledovne :

$$x_k^- = F_{k,k-1}x_{k-1} \tag{6}$$

$$P_k^- = F_{k,k-1}P_{k-1}F_{k,k-1}^T Q_{k-1} \tag{7}$$

Po vypočítaní prvého bodu pokračujeme zadefinovaním rovníc pre korekciu odhadu:

$$K_k = P_k^- H_k^- [H_k P_k^- H_k^T + R_k]^{-1} \tag{8}$$

$$x_k = x_k^- + K_k(y_k - H_k x_k^-) \tag{9}$$

$$P_k = (I - K_k H_k)P_k^- \tag{10}$$

Ak algoritmus vypočíta aj druhý krok vracia sa naspäť do kroku, 1 s novými hodnotami, pričom časový vektor k už bude posunutý v čase o k+1.[15]

1.3.1 Rozšírený Kalmanov filter (EKF)

Hlavným prínosom rozšíreného Kalmanového filtra je počítanie nelineárnych systémov. Ak takýto systém chceme odhadovať, musíme použiť rozšírený Kalmanov filter, pretože nie je možné nelineárny systém odhadnúť klasickým Kalmanovým filtrom.

$$x_k^- = \mathrm{F}(x_{k-1}, u_k)R_k \tag{11}$$

$$x_k = x_k^- + K_k(y_k - h(x_k^-))$$

12

Vyniesli sme iba rovnice, v ktorých sa líšia KF a EKF. V týchto rovniciach K_k reprezentuje Kalmanov zisk, $h(x_k)$ reprezentuje nelineárnu funkciu opisujúcu vzťah medzi stavom a meraním systému. H_k je matica merania, R_k reprezentuje chybu modelu v čase.[2]

EKF, rovnako ako KF, opakuje kroky predikcie a aktualizácie pre každý časový krok. Celkovo rozšírený Kalmanov filter je robustný algoritmus pre odhad stavu v nelineárnych modeloch a systémoch s Gaussovým šumom. Avšak, taktiež má obmedzenia, ako napríklad, že EKF potrebuje znalosť odhadu počiatočného stavu systému.[15]

1.4 Lokalizácia

Mobilné roboty alebo mobilné vozidlá vieme rozdeliť podľa stupňa autonómie. Poznáme vozidlá, ktoré nedisponujú žiadnym stupňom autonómie, a teda sú priamo ovládané operátorom. Takéto vozidlá nepotrebujú mať znalosť lokalizácie. Ďalší druh sú semi-autonómne vozidlá, ktoré vie stále ovládať operátor, ale taktiež môžu prejsť do autonómneho módu. Vozidlá s najvyšším stupňom autonómie sa už samostatne rozhodujú o svojom nasledujúcom kroku a vyhýbaní sa prekážkam. Takéto vozidlá už musia mať znalosť svojej polohy v priestore. V nasledujúcich kapitolách si predstavíme niekoľko lokalizačných metód, medzi ktoré môžeme zaradiť lokalizáciu z odometrie, z GPS alebo vizuálnu lokalizáciu, kedy sa mobilný robot vie lokalizovať na základe kamery. Každá z daných metód má svoje výhody, ale aj nedostatky, ktoré treba dôsledne zvážiť pri ich výbere pre zvolenú aplikáciu.[1]

1.4.1 Relatívna lokalizácia

Relatívna lokalizácia je proces, kedy mobilný robot určuje svoju polohu, na základe svojej predchádzajúcej polohy. K jej výpočtu sa využívajú senzory, ako IMU alebo odometria. Čím presnejšie senzory budú, tým presnejšia bude aktuálna poloha robota. Relatívna lokalizácia je pomerne citlivá na chyby a nepresnosti, keďže jednotlivé polohy na seba priamo nadväzujú.[1]

1.4.2 Absolútna lokalizácia

Pri absolútnej lokalizácií sa proces lokalizovania robota určuje vzhľadom na určené pevné body v prostredí. Využívajú sa na to technológie, ako GPS alebo kamery. Absolútna lokalizácia je v priebehu času, ako relatívna lokalizácia, ale v praxi je najlepšie skombinovať obe tieto lokalizácie, a navýšiť tým ich presnosť a spoľahlivosť.[1] [12]

1.5 Rozdiel medzi online a offline SLAM-om

Online a offline SLAM sú dva rôzne prístupy k riešeniu problému SLAM. Je tu rozdielny prístup k lokalizácii robota, a zároveň aj k tvorbe mapy. Hlavnými rozdielmi sú aktualizácia mapy, spracovanie dát, pamäťová náročnosť týchto riešení, robustnosť voči chybám a implementácia.

Hlavným rozdielom je, že online SLAM spracováva všetky dáta v reálnom čase, zatiaľ čo offline SLAM číta už namerané dáta uložené z dátového súboru, ktoré následne spracováva a vytára z nich mapu.

Ďalším rozdielnym prístupom je aktualizácia mapy, v online SLAMe sa mapa aktualizuje kontinuálne s príchodom nových dát. Mapa je takmer okamžite aktualizovaná o nové informácie z rôznych senzorov, snímačov a následne prebieha aj lokalizácia robota. V offline SLAM-e sa mapa prostredia zobrazí až po ukončení celého pohybu robota a získaní všetkých dát zo senzorov. Keďže nie je potrebná okamžitá lokalizácie, tieto algoritmy pracujú trochu iným spôsobom.

Online SLAM je náročnejší na výpočtový výkon z dôvodu, že spracováva všetky dáta zo senzorov a aktualizuje mapu v reálnom čase. Naopak, pri offline SLAM-e obmedzenie na časovú náročnosť nemáme, keďže dáta sa načítavajú z dátového súboru. Z toho princípu vyplýva aj, že offline SLAM je náročnejší na väčšiu pamäťovú kapacitu, pretože množstvo dát zo senzorov sa ukladá počas celého behu robota. To môže vyžadovať značnú časť pamäte, hlavne pri väčších objektoch. Online SLAM, kde sa mapa aktualizuje postupne, nevyžaduje veľkú pamäťovú kapacitu.

Online SLAM môže byť náchylnejší voči chybám, keďže dáta sa spracovávajú v reálnom čase, ich odfiltrovanie nie je v niektorých prípadoch hneď možné a chybné dáta takto môžu ovplyvniť výslednú mapu. Pri offline SLAM-e vieme viac zasiahnuť voči chybám, keďže sa mapa spracováva ako celok zo všetkých nameraných dát. Chybné dáta sa vedia kvalitnejšie odfiltrovať a tým docieliť kvalitnejší opis reálneho

prostredia mapou s menším počtom chýb. Takže offline SLAM je robustnejší z pohľadu chybovosti.

Z hľadiska celkovej implementácie je online SLAM náročnejší, keďže, vyžaduje rýchle spracovanie dát a aktualizáciu mapy v reálnom čase. Získavanie dát zo senzorov občas vyžaduje ďalšie procesy, ktoré budú tieto dáta zbierať a odosielať ich do SLAMu. Avšak, pri offline SLAMe je tento proces o niečo jednoduchší z dôvodu, že dáta sú hromadne uložené a len sa preposielajú do algoritmu pod správnymi premennými na ich spracovanie.

Toto sú najzásadnejšie rozdiely medzi offline a online SLAMom. Každý z týchto algoritmov má svoje miesto vo vývoji mobilných aplikácií, je len potrebné zvážiť, pre ktorú aplikáciu je lepší offline SLAM a opačne. [23]

1.6 Problémy so simultánnou lokalizáciou a mapovaním

Pri 2D SLAM algoritmoch sa nachádzajú problémy, ktoré nie je vždy možné vyriešiť. Predstavíme si niektoré problémy, s ktorými sa SLAM-y stretávajú.

Ako prvý problém si spomenieme chýbajúcu informáciu o výške. Keďže 2D SLAM pracuje iba v dvoch rozmeroch, neeviduje informáciu o výške merania. Tento problém obmedzuje meranie vo viacúrovňových prostrediach a môže prísť k nesprávnym výsledkom lokalizácie a mapovania takýchto štruktúr.

Neschopnosť rozpoznať prekážky mimo roviny lasera. Keďže 2D SLAM pracuje v jednej rovine, nemusí zachytiť všetky objekty, ktoré sa javia ako prekážky pretože sa nachádzajú mimo roviny 2D lasera, ale zasahujú do trajektórie mobilného robota. Tieto prekážky vytvárajú chyby v mapovaní a môže prísť ku kolízii. Daný problém nie je chyba algoritmu avšak musíme počítať aj s takýmto problémom.

Ďalším problémom je veľká závislosť na senzoroch, keďže presnosť 2D SLAM-u priamo vychádza z presnosti senzorov. Ak sú senzory nedostatočne presné, dochádza k zbytočným chybám v mapovaní a lokalizácií robota v priestore. Okrem toho, senzory môžu pomýliť aj iné faktory ako osvetlenie, povrch (napríklad sklo ktoré LIDAR nezaregistruje), alebo rovnaký rovinný LIDAR na druhom mobilnom robotovi.

Na základe čoho môžeme prejsť ďalšiemu problému a to sú výzvy mapovania a lokalizácie v dynamicky sa meniacom prostredí. V prostrediach, kde sa menia objekty a pohybujú iné mobilné roboty, môže byť komplikované sa lokalizovať a mapovať prostredie, pretože všetky tieto objekty môžu zmeniť tvar a štruktúru mapy, čo môže viesť k nepresnosti pri jej tvorbe.

Ďalším nedostatkom môže byť veľká komplexnosť prostredia. 2D SLAMy majú problém pri mapovaní priestorov, pri ktorých je ťažké definovať príznaky a nadpojiť jednotlivé mapy. Rozprávame sa o dlhých chodbách alebo o veľkých otvorených miestnostiach. Ďalej sa v daných miestnostiach môžu nachádzať objekty, ktoré sa nezaznamenajú buď kvôli povrchu (sklo), alebo zle umiestnenému rovinnému senzoru.

2 ROS

Robot Operating System v skratke ROS, je open-source softvér určený pre vývoj a testovanie rôznych typov robotov alebo mobilných vozidiel a dronov. ROS poskytuje užívateľom rôzne balíčky knižnice a silné nástroje pre vývoj týchto zariadení. Tento softvér sa používa v akademických prostrediach, v priemysle a vo výskumných laboratóriách. ROS je založený na balíčkovej štruktúre, pričom jednotlivé balíčky vytvára samotná komunita a medzi sebou si ich pomocou rôznych platforiem, najčastejšie platformou GitHub, zdieľajú. Taktiež má ROS veľmi schopné nástroje na vizualizáciu akou je Rviz, alebo simulačný nástroj Gazebo. Dnes už evidujeme dve verzie ROS, a to ROS1 a ROS2 pričom naša práca sa bude ďalej zaoberať prvou verziou tohto systému.[19]

2.1 ROS Node

ROS Node (uzol) je samostatný proces, ktorý sa spúšťa v systéme ROS. Je základnou jednotkou tohto systému a zabezpečuje rôzne procesy, ako servisy, publikovanie a odoberanie tém z topicov iných uzlov. Jeho veľkou výhodou je, že môže byť implementovaný v jazykoch C++ a Python.

Uzly môžu zohrávať rôzne úlohy v robotike akými môžu byť zber a spracovanie dát zo senzorov alebo riadenia pohybu robota, tvorenie máp, komunikácia s inými systémami mimo ROSu a iné. Vďaka veľkej flexibilite systému ROS je možné tieto uzly rôzne kombinovať a implementovať, avšak, netreba zabudnúť na spustenie uzlu "Master", inak sa tieto uzly nebudú navzájom vidieť.[22]

2.2 ROS Topics

ROS topics je spôsob, akým systémové uzly komunikujú medzi sebou, zväčša asynchrónnym spôsobom. Pracujú s princípom "publisher-subscriber", čiže jednotlivé uzly vedia správy posielané pomocou ros topicov buď zdieľať (publisher), alebo prijímať (subscriber). Samozrejme, jeden uzol vie aj prijímať aj zdieľať správy, je to len na užívateľovi ako si jednotlivý uzol nastaví.

Topics sú jedným z hlavných spôsobov, aký využívajú ROS uzly na komunikáciu medzi sebou a sú súčasťou širšieho spektra komunikačných modelov,

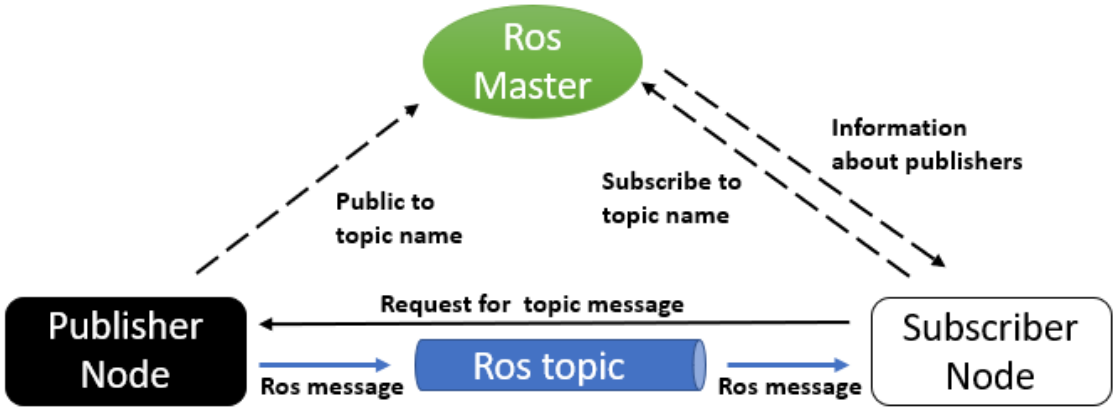
ktoré nám systém ROS poskytuje. Ďalšie komunikačné rozhrania môžu byť napríklad ROS služby alebo ROS akcie.[21]

2.3 Ros Messages

ROS Messages (správy) sú definície dátových štruktúr, ktoré používajú uzly v rámci komunikačného modelu ROS topics. Správy sú základným spôsobom, akým sa dáta zdieľajú medzi jednotlivými uzlami.

Podstatné je, že každý typ správy je dopredu definovaný. Definícia zahŕňa jeho štruktúru alebo triedu, ktoré obsahujú jednotlivé položky s rôznymi dátovými typmi, ako napríklad int, float, String alebo rôzne polia. Každá z ros správ má vlastný názov, čím sa vie odlíšiť ich implementácia a využitie v uzloch.

Definícia ROS správ je nezávislá od programovacieho jazyka v uzle, a tým pádom nám zabezpečuje rôznorodosť uzlov v systéme ros. Tieto správy sú definované v súboroch s príponou .msg. Ak by sme však potrebovali zadefinovať vlastnú správu, je možné ju definovať v týchto súboroch.



Obr. 7 ROS štruktúra systému [8]

Na Obr. 7 môžeme vidieť ilustračné zobrazenie ako medzi sebou pracujú ros uzly, ros správy a ros topicy, pričom tieto jednotlivé časti zastrešuje hlavný uzol s názvom Ros master.[20]

2.4 Transofrmačný balík „TF“

Pre prácu s pozíciou robota je potrebné si zadefinovať globálnu súradnicovú sústavu, v ROSe je označovaná ako „map.“ Následne musíme vytvoriť lokálnu

súradnicovú sústavu, ktorá bude predstavovať aktuálnu polohu robota v globálnej súradnicovej sústave avšak musí mať zadefinované tri súradnice:

$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Následne je potrebné vypočítať závislosť medzi jednotlivými súradnicovými systémami, najčastejšie sa používajú Eurelove uhly. Následne je možné pomocou rotačných a translačných matíc vypočítať posun robota v priestore.

Rotačné matice :

Rotácia okolo z-ovej osi:

$$R_z, \varphi = \begin{bmatrix} cos\varphi & -sin\varphi & 0 & 0 \\ sin\varphi & cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

13

Rotácia okolo z-ovej osi:

$$R_x, \varphi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\varphi & -sin\varphi & 0 \\ 0 & sin\varphi & cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

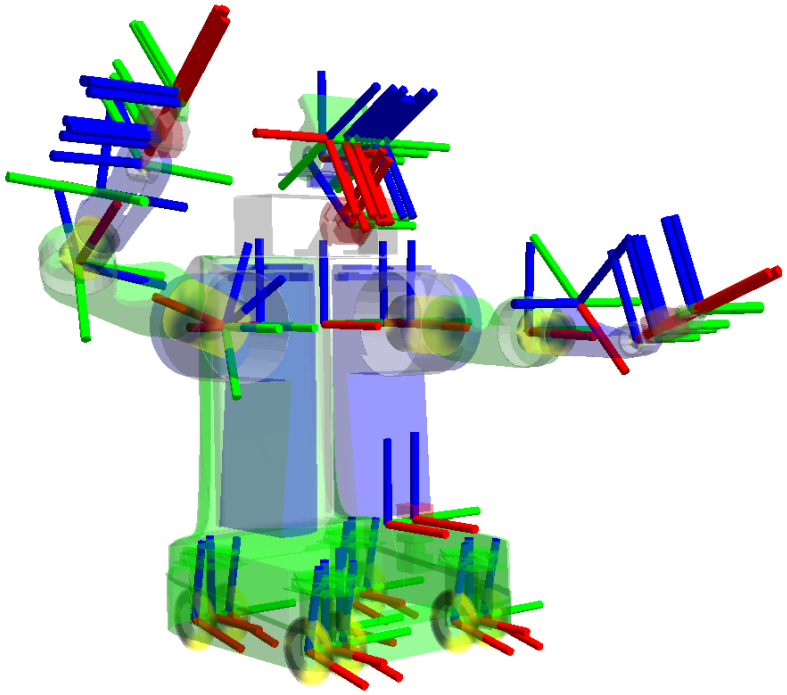
14

Rotácia okolo y-ovej osi:

$$R_y, \varphi = \begin{bmatrix} cos\varphi & 0 & sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -sin\varphi & 0 & cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

15

Avšak počítať natočenia pomocou Eulerových uhlov je výpočtovo veľmi náročné preto ROS prevažne používa kvaterniónový opis. Pričom tvoria 4 rozmernú normovanú algebru nad poľom reálnych čísel. O dané prepočty sa už stará transformačný balík „tf," ktorý potrebuje mať jasne zadefinované ako jednotlivé súradnicové sústavy na seba vplývajú. [17]



Obr. 8 Zobrazenie transformačného balíku[9]

2.5 Balík mapServer

Balíček mapServer je súčasťou systému ROS, bol navrhnutý tak, aby sa vedel jednoducho integrovať s inými balíčkami ROS. Jeho hlavnou úlohou je umožniť užívateľovi ukladať vytvorenú mapu alebo načítavať už uloženú mapu naspäť do prostredia ROS.

```
$ rosrun map_server map_saver -f <cesta k suboru>
```

Vyššie uvedený príkaz nám slúži pre uloženie 2D mapy z prostredia ROS. Po jeho zadaní nám mapServer uloží dva súbory, a to mapu vo formáte .pgm a jej bližšiu špecifikáciu vo formáte .yaml. Táto špecifikácia mapy vyzerá nasledovne :

```
image: testmap.png
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

Vieme z nej vyčítať názov obrázku mapy, ďalej rozlíšenie mapy ktoré nám udáva meter na pixel, takže v tomto prípade by nám jeden pixel na obrázku reprezentoval 10 cm v skutočnosti. *Origin* nám udáva polohu ľavého dolného pixelu na mape ako x,y, rotácia, rotácia je udaná v smere hodinových ručičiek, pričom číslo 0 znamená bez rotácie. *Occupied thresh* nám hovorí o pravdepodobnosti obsadenosti a *free_thresh* zase o pravdepodobnosti voľných pixelov.

Tento balíček bude pre nás nesmierne dôležitý keďže potrebujeme vytvorené mapy naďalej spracovávať pre ich vyhodnotenie a porovnanie. [6]

2.6 Balík laser_scan_matcher

Balík laser_scan_matcher zabezpečuje transformáciu medzi nehybnými súradnicovými systémami. Je potrebné použiť daný balíček z toho dôvodu, že naša práca nemá vytvorený udf model robota. Naďalej budeme využívať iba namerané dáta z odometrie a robota. Náš transformačný strom bude vyzerať nasledovne :

```
Map->odom->base_link->laser
```

My vieme, že transformáciu medzi `odom->base_link` budeme vypočítavať, ostatné transformácie nám zabezpečí tento uzol. Transformácia medzi mapou a odometriou je voliteľná. Niektoré SLAM-y vedia danú transformáciu zabezpečiť a niektoré nie.[7]

3 Analýza SLAM metód

V nasledujúcej kapitole si objasníme výber SLAM algoritmov, ktoré podrobíme testom a bližšie si každý z nich popíšeme. Zameriame sa na ich algoritmické vlastnosti, ako sú API jednotlivých balíčkov a ich základné parametre.

3.1 Analýza aktuálnych 2D SLAM metód

Pri analýze dostupných SLAM algoritmovsme sa odrazili od základných požiadaviek, ktoré by mal spĺňať každý z nich. Tieto požiadavky boli nasledujúce: SLAM musí podporovať rovinný LiDAR, teda 2D LiDAR, ktorý by mal byť spustiteľný v v systéme ROS1 a konkrétne na verzií Noetic. Nasledujúce SLAM algoritmy spĺňali naše požiadavky podľa dostupných charakteristík:

Tabuľka 1 Dostupné SLAM algoritmy

Názov SLAM algoritmu	Dátum vydania (orientačný)	Podpora systému ROS
Google Cartographer	2016	Noetic
SlamToolbox	2019	Noetic
Gmapping	2006	Noetic
HectorSlam	2013	Noetic
SSA	2011	Noetic
FalkoLib	2016	-
FlirtLib	2015	-

Každý zo SLAM algoritmov uvedených v tabuľke sme sa pokúsili implementovať do systému ROS, avšak, pri posledných troch spomínaných nás zastavila ich nedostatočná dokumentácia, a teda výslednú implementáciu sme museli prerušiť. Naďalej budeme v práci sa budeme zaoberať nasledujúcimi algoritmam: Google Cartographer, SlamToolbox, HectorSLam a ako najznámejší SLAM algoritmus sme zaradili aj Gmapping.[10]

3.2 GMapping

GMapping je jeden z najznámejších 2D SLAM algoritmov v systéme ROS 1. Jeho implementácia do systému je užívateľsky dobre opísaná. Pre správne zostrojenie mapy potrebuje dáta z dvoch senzorov, a to z laserového 2D skenera a dáta z odometrie.

Celý proces začína odhadom prvej pozície robota. Robot pošle dáta z odometrie a laserového diaľkomeru a algoritmus ich spracuje do jednotlivých meraní, ktoré následne využije pre aktualizáciu mapy. Mapa sa zobrazuje ako mriežka buniek (pixelov), kde každý pixel zodpovedá malej oblasti prostredia. Záleží od rozlíšenia mapy, aké si zvolíme. Pixely môžu nadobudnúť 3 rôzne hodnoty: obsadený, voľný a neznámy priestor.

Algoritmus GMapping využíva techniku filtrovania častíc, aby si udržal prehľad o pozícii robota. Ako sme spomínali už pri podkapitole 1.2.3 FastSlam-e, algoritmus má viacero možných polôh robota, ktoré vyplývajú z dát. S plynúcim časom tvorby mapy sa dostáva do algoritmu viac dát a mapa sa stáva presnejšou, rovnako aj odhad pozície sa stáva presnejším. To umožňuje robotovi sa účinnejšie prechádzať prostredím a dosiahnuť tak svoj cieľ.

Odhad stavov robota ktoré využíva GMapping, vieme vyjadriť nasledovne :

$$p(x_t|z_{1:t-1}, u_{1:t}, m) = \int p(x_t|z_{t-1}, u_t, m)\, p(x_{t-1}|z_{1:t-1}, u_{1:t-1}, m) dx_{t-1}$$

16

kde x reprezentuje stav(pravdepodobnosť mriežky), z reprezentuje senzorové merania a u je pohyb robota. M reprezentuje mapu prostredia a t je index času. Ďalším krokom je aktualizácia, kedy sa pravdepodobnosť stavu aktualizuje na základe senzorových meraní nasledovne:

$$p(x_t|z_{1:t}, u_{1:t}, m) = \frac{p(z_t|x_t, m)p(x_t|z_{1:t-1}, u_{1:t}, m)}{p(z_t|z_{1:t-1}, u_{1:t}, m)}$$

17

kde prvý člen v čitateli je pravdepodobnosť zistenia senzorových dát z aktuálnej pozície a druhý člen nám udáva predikciu stavu na základe minulých stavov. Menovateľ je konštanta zabezpečujúca, že pravdepodobnostné hodnoty stavov sú normalizované.

Pri výpočte pravdepodobnostných stavov sa dané kroky opakujú pre každé meranie.

Okrem množstva parametrov, ktoré je potrebné pred spustením algoritmu skontrolovať, sa pozrieme na API ktoré nám tento balíček ponúka. GMapping nám odoberá transformačný topic v ktorom sa nachádzajú transformácie jednotlivých súradnicových systémov robota a topic scan, ktorý reprezentuje dáta z 2D LiDARu.[14]

3.3 Google Cartographer

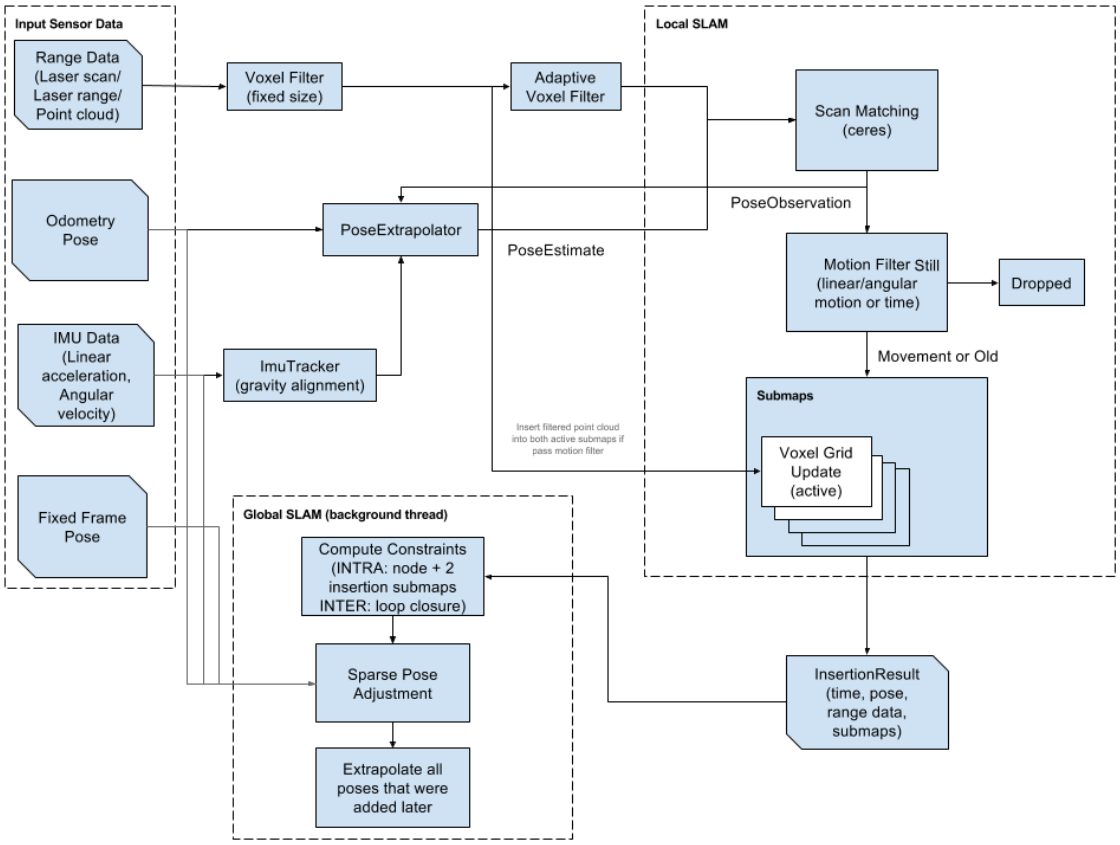
Google Cartographer je univerzálny SLAM algoritmus z dôvodu, že vie spracovávať mapu ako v 2D, tak aj v 3D, záleží od konfiguračných nastavení. Cartographer odoberá 3 topicky pre dáta z vizuálnych senzorov, a to *scan, echoes* a *points2*. V našej práci budeme využívať topick scan v 2D. Topick *echoes* sa využíva ako v 2D tak aj v 3D a zužitkovávajú ho rotačné lasery. Points2 je topick pre pointcloud snímaný napríklad z 3D kamery.

Senzory z robota vieme reprezentovať taktiež v dvoch rôznych topikoch, a to imu (inerciálna meracia jednotka) alebo odom (odometria). Oba tieto odoberané topicky sa musia vopred nastaviť, či sa budú publikovať v 2D, alebo v 3D prostredí.

Google Cartographer využíva techniku scan matching, pomocou ktorej odhaduje aktuálnu polohu a orientáciu robota. Scan matching funguje na princípe najlepšieho nadpojenia aktuálneho scanu na už vytvorenú mapu pri extrahovaných príznakoch z aktuálne skenu a mapy. Nakoniec odhadne polohu robota.

2D mapa sa zobrazuje rovnako ako pri GMappingu kde pixel nadobúda tri hodnoty, obsadený prekážkou, voľný a neznámy.

Vytvorená mapa je aj naďalej optimalizovaná, aby sa minimalizovali chyby a zlepšila presnosť mapy. Táto minimalizácia prebieha technikou optimalizácie na grafoch.



Obr. 9 Schéma závislostí GoogleCartographeru

Na obrázku vyššie, môžeme vidieť podrobnejšie rozobraný celý proces algoritmu Google Cartographer, ak sa zameriame na ľavú časť, môžeme vidieť vstupné témy, ktoré Cartographer odoberá. My použijeme všetky okrem *IMU Data,* pretože využijeme odometriu robota. [11]

3.4 HectorSlam

HectorSlam je ďalší SLAM algoritmus ktorý sa dá stiahnuť ako balíček do systému ROS. Skladá sa z troch uzlov hector_mapping, hector_geotiff a hector_trajetcory_server.

Posledný zo spomínaných ma na starosti lokalizovanie robota v čase a zostrojovanie trajektórie, ktorú robot prešiel za pomoci odometrie.

Hector_mapping, pomocou odhadnutej pozície a senzorových dát tvorí mapu prostredia. Mapa je reprezentovaná v 2D ako grafová mapa, kde každý pixel nadobúda tri hodnoty, ako voľný, obsadený alebo neznámy.

Hector_geotiff je uzol, ktorý uschováva údaje o vytvorenej mape a trajektórií ktorú robot prešiel. Dáta sa ukladajú podľa pravidiel RoboCup pričom vyhovujú súborom geotiffov s georeferenčnými súradnicami. [16]

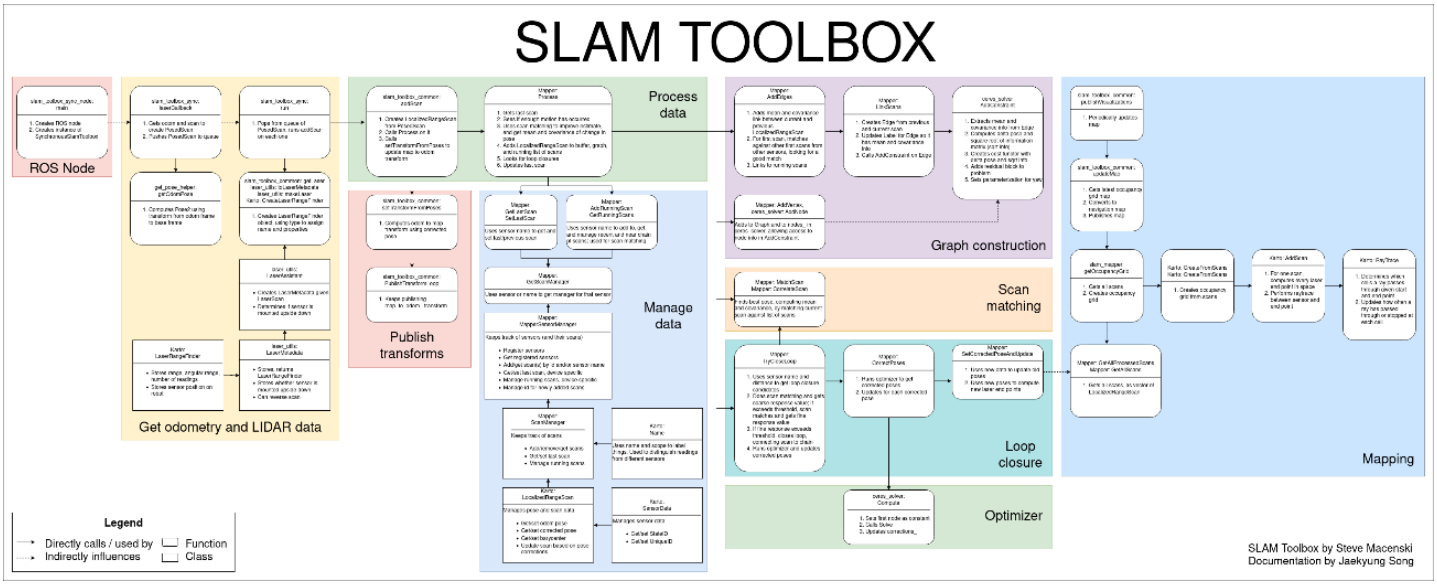
3.5 SlamToolbox

SlamToolbox je posledný SLAM algoritmus, ktorý si popíšeme. Je to taktiež voľne dostupný balíček ktorý sa dá doinštalovať do systému ROS. V balíčku sa nachádza aj rozšírenie do nástroja RVIZ na prácu s mapou.

SlamToolbox podporuje tvorbu 2D mapy z dát rovinného lasera, a z dát odometrie alebo IMU. Tento balíček obsahuje iba jeden uzol, ktorý spracováva všetky dáta pomocou ktorých odhaduje pozíciu robota a vykresľuje 2D mapu.

Tak ako aj predchádzajúce SLAM algoritmy aj SlamTtoolbox disponuje pomerne veľa vstupným parametrom ktoré je možné nastaviť podľa potreby užívateľa a vylepšiť tak jeho použitie podľa aplikácie na ktorú sa bude využívať. Všetky parametre sú spísané v dokumentácií na portály GitHub.

SlamToolbox vykresľuje mapu do 2D mriežkovej mapy pričom sa na nej môžu objaviť iba tri farby pixelov reprezentujúce, obsadenosť pixelu, prázdny pixel a neznámu oblasť.



Obr. 10 Štruktúra SlamToolboxu

Na Obr. 10 môžeme vidieť základnú štruktúru SlamToolboxu ako spracováva údaje a až po vykreslenie mapy.

4 Použitý hardvér

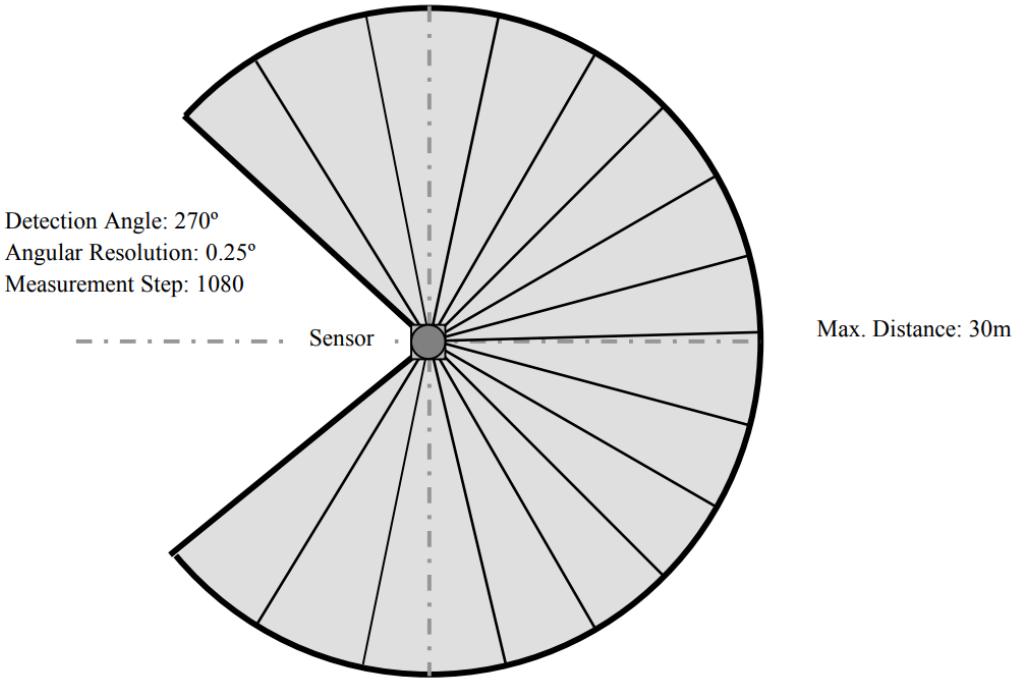
V nasledujúcej kapitole si priblížime hardvér, ktorý bol využitý na zber a tvorbu dát potrebných pre jednotlivé SLAM-y. Hardvér si bližšie špecifikujeme a vynesieme jeho základné vlastnosti.

4.1 Hokuyo LiDAR

Hokuyo UTM-30LX je 2D infračervený laserový diaľkomer, ktorý skenuje plochy v 2D rovine.

Obr. 11 Hokuyo UTM-30LX

Jeho vlnová dĺžka je 785nm, (bezpečnostná kategória 1 – neškodné pre človeka). S týmto laserom môžeme pracovať dlhodobo a sledovať ho priamo aj s optickými pomôckami. Jeho dosah činí 0,1 m až 30 m, pričom zachytí plochu v uhle 270° s krokom 0,25°, kde vykoná dohromady 1081 meraní.



Obr. 12 Rozsah Hokuyo UTM-30LX lasera. [3]

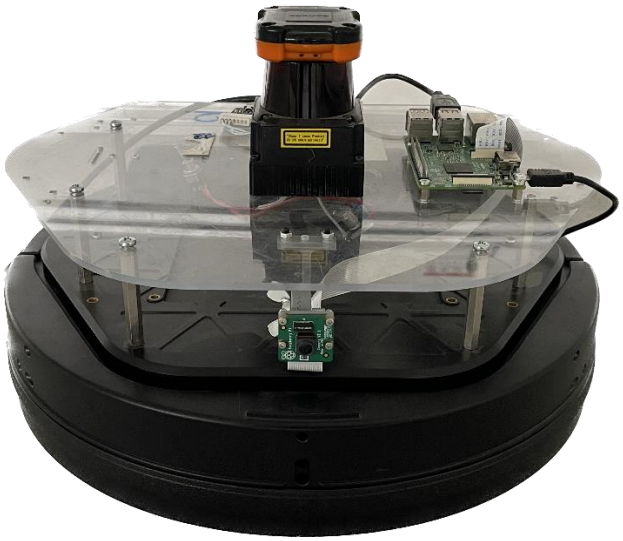
Výrobca uvádza odchýlku LiDARu na ± 30mm v rozsahu od 0,1m do 10m a ±50mm v rozsahu od 10m do 30m.[3]

Ak Hokuyo LiDAR chceme použiť v systéme ROS, je potrebné využitie balíčku Hokuyo_node, ktorý zabezpečuje komunikáciu medzi diaľkomerom a systémom ROS.

4.2 Mobilný robot Kobuki

Kobuki je mobilný robot navrhnutý spoločnosťou Yujin Robot, ktorý sa využíva v priemyselnej automatizácii, pri výskume a výučbe robotiky. Kobuki je malý robot s diferenciálnym podvozkom. Jeho podvozok je prispôsobený rôznym povrchom, ako interiérovým tak aj exteriérovým, pričom je navrhnutý tak, aby sa mobilný robot vedel efektívne vyhýbať rôznym prekážkam.

Hmotnosť robota je udaná na 4.5 kg a jeho rozmery sú 350mmx 380mm x 100mm, čo značí že robot je naozaj malý a vieme ho prirovnať k novodobým automatickým vysávačom kruhového tvaru. Napájanie robota zabezpečuje 19V DC, 4.75 Ah Li-ion batéria, ktorá zaručuje 3-hodinovú výdrž pri klasikom používaní. Pre nás je výhodou, že robot ma na jeho zadnej strane aj 12V výstup, z ktorého vieme napájať LiDAR Hokuyo a vieme využiť batériu robota pričom nie sme limitovaný napájaním zo siete.



Obr. 13 Mobilný robot Kobuki s Hokuyo LiDARom

Komunikácia s mobilným robotom Kobuki vie prebiehať cez niekoľko rozhraní, ako napríklad cez USB, Ethernet alebo WiFi. V našom prípade využijeme komunikáciu cez WiFi.

Aj keď tento robot disponuje rôznymi senzormi, ako napríklad odrazovou optikou, snímačmi infračerveného diaľkového ovládania a aj senzormi tlaku, my ich nevyužijeme, preto je potrebné pripevniť na robota Hokuyo LiDAR, ako je zobrazené aj na Obr. 13 pripevniť na robota Hokuyo LiDAR.

Tabuľka 2 Špecifikácia Kobuki Robota

	Názov	Hodnota
Parametre robota	Rázvor	230 mm
	Polomer kolesa	35 mm
	Šírka kolesa	21 mm
Konverzie	Tick na meter	8. 5292090497737556558*10^{-5} m/tick

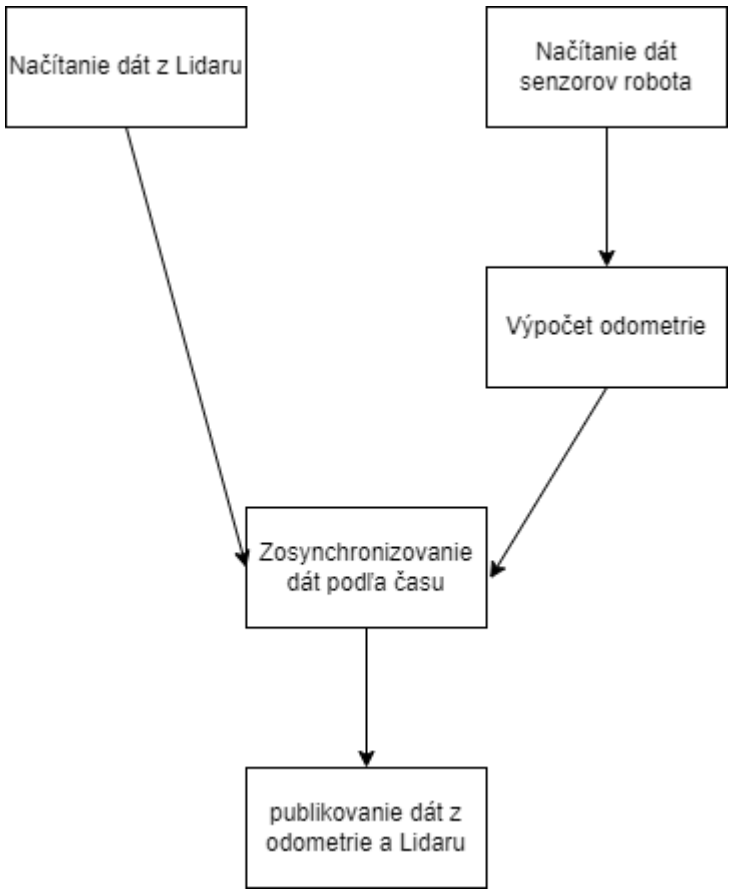
V Tabuľka 2 môžeme vidieť niekoľko podstatných špecifikácií, ako je rázvor robota alebo tick na meter. Tieto hodnoty sú pre nás podstatné. Na dáta sa budeme odvolávať pri výpočte odometrie pre tohto robota, pretože raw dáta nám udajú iba jednotlivé pohyby z enkodérov kolies robota a nie súradnice v priestore, tie musíme dopočítať.[4] [5]

5 Aplikovanie SLAM metód v robotickom operačnom systéme

V nasledujúcej kapitole popíšeme ako možno implementovať jednotlivé metódy SLAM a ich následnú konfiguráciu pre naše potreby. Taktiež si popíšeme kompletnú tvorbu ROS publishera, vytvoreného pre spracovanie dát z robota a LIDARu a taktiež ich následné publikovanie do SLAM algoritmov.

5.1 Vytvorenie vlastného uzla v systéme ROS

Pre lepšie vysvetlenie celého procesu spracovania dát a publikovania do SLAM-u sme si pomohli stavovým diagramom.



Obr. 14 Stavový diagram postupnosti vykonávania ROS uzla

Každý stav zo stavového diagramu si bližšie opíšeme v nasledujúcich podkapitolách. Náš uzol sa nachádza v nami vytvorenom balíčku slúžiacemu na spracovanie dát nameraných na robote Kobuki s LiDARom Hokuyo. Dáta sme ukladali do separátnych textových súborov, tj. lidar.txt a robot.txt. Všetky základné a pomocné funkcie v našom balíčku sú písané v jazyku Python, pretože je z hľadiska spracovávania dát lepšou voľbou , ako jazyk C++. Práve z tohto dôvodu sme si ho zvolili.

5.1.1 Spracovanie prijatých dát

Na to, aby sme mohli ďalej pracovať s dátami, je potrebné ich najskôr spracovať. Spracovávame ich v dvoch rôznych funkciách, pretože z hľadiska ich veľkej rozdielnosti nie je možné ich spracovať iba jednou funkciou.

Dáta z odometrie majú formát štyroch stĺpcov s nasledovnou postupnosťou: čas merania, ľavý enkodér, pravý enkodér robota a gyroskop, avšak, ten sme nebrali do úvahy. Jeden z mnoho riadkov môže vyzerať nasledovne :

2664722; 60179; 55523; -801

Po načítaní sme dáta jednoducho roztriedili do 2D poľa pre ďalšiu prácu. Avšak, dáta z diaľkového laseru boli o niečo zložitejšie na spracovanie. Ich postupnosť pre jedno meranie začalo časovou konštantou, pokračovalo konštantou, ktorá udávala počet nameraných vzoriek a tých bolo v každom meraní 1081. Ďalej nasledovali uhly v rozmedzí <-135;135> s krokom 0.25 a samotné hodnoty pre každý uhol, pomocou ktorých sme vedeli vykresliť jednotlivé obrazy, ktoré LIDAR nasnímal. Po načítaní sme dáta taktiež rozdelili zvlášť do polí pre efektívnejšiu prácu v ďalších krokoch.

5.1.2 Algoritmus pre výpočet odometrie robota

Po načítaní dát z textového súboru je potrebné hodnoty enkodérov prepočítať na súradnice x,y. Keďže náš robot mapuje a chodí iba v 2D priestore, premennej z nastavíme hodnotu 0. Prvú polohu sme nastavili na súradnice 0,0,0 (x,y,z).

Kobuki má diferenciálny podvozok, preto výpočet odometrie budeme počítať dvoma rovnicami. Prvé rovnice využijeme, ak robot pôjde priamočiarym pohybom :

$$x_{k+1} = x_k + l_k \cos \varphi_k \quad (18)$$

$$y_{k+1} = y_k + l_k \sin \varphi_k \quad (19)$$

$$\varphi_{k+1} = \varphi_k + \Delta\alpha$$

20

Kde *x* a *y* udávajú hodnoty v priestore, l je vypočítaná hodnota z enkodérov, φ nám udáva uhol natočenia robota a index k udáva čas merania. Následne si priblížime výpočet hodnôt l a $\Delta\alpha$.

$$l_r = \text{tickToMeter} * (enc_{rk+1} - enc_{rk})$$
$$l_l = \text{tickToMeter} * (enc_{lk+1} - enc_{lk})$$

21

$$l = \frac{l_r - l_l}{2}$$

22

$$\Delta\alpha = \frac{l_{kr} - l_{kl}}{d}$$

23

Hodnotu *tickToMeter* sme uviedli v Tabuľka 2 Špecifikácia Kobuki Robota a *enc* udáva hodnotu enkodéru, pričom ich musíme rozlíšiť podľa indexu r a l, teda pravý a ľavý. Premenná d reprezentuje rázvor, ktorý je taktiež udaný v Tabuľka 2. Následne môžeme prejsť na výpočet po kružnici, ktorý nám definujú nasledujúce rovnice:

$$x_{k+1} = x_k + \frac{d(l_r + l_l)}{2(l_r - l_l)}(\sin\varphi_{k+1} - \sin\varphi_k)$$

24

$$y_{k+1} = y_k - \frac{d(l_r + l_l)}{2(l_r - l_l)}(\cos\varphi_{k+1} - \cos\varphi_k)$$

25

$$\varphi_{k+1} = \varphi_k + \Delta\alpha$$

26

Všetky premenné sme už vysvetlili vyššie. Ešte je potrebné si objasniť kedy budeme používať rovnice pre priamočiary, a kedy pre kruhový pohyb. Ak odčítame l_r od l_l a hodnota je nulová, vieme, že ideme priamočiaro. Ak je hodnota rôzna nule vieme že robot opisuje pohyb po kružnici.

5.1.3 Nadviazanie komunikácie

Pre správnu komunikáciu medzi našim uzlom a SLAM algoritmami potrebujeme publikovať dve správy. Správu pre dáta z lasera a správu pre dáta s odometriou. Najskôr popíšeme tvorbu správy pre dáta z diaľkového 2D lasera.

Každá správa začína hlavičkou, v ktorej sa zaznamenávajú tri hodnoty, ktorými sú poradie správy, čas, kedy boli dáta namerané a názov súradnicového systému, pod ktorým laser vystupuje. V našom prípade je názov súradnicového systému *„laser.“*

Do premenných *angle_min, angle_max* zapíšeme maximálne a minimálne rozmedzie v akých laser meria, v našom prípade sú to hodnoty <-135;135>. Hodnota angle_increment definuje prírastok v každom meraní, čo je v našom prípade 0.25, avšak, dané hodnoty je nutné previesť na radiány.

Premenné *range_min a range_max* udávjú rozmedzie v akých vie LIDAR pracovať. Ako sme už uviedli, jeho pracovný rozsah je od 0.1m do 30m.

Ďalej máme premennú, ktorá očakáva na vstup pole nameraných hodnôt v [m] a jej názov je *ranges.* Následne už vieme zostrojiť správu pre dáta z diaľkového lasera. Dodatočne si musíme opísať tvorbu správy aj pre odometriu. Tá je zložitejšia na spracovanie pretože ROS vyžaduje dáta vždy v karteziánskej sústave a v kvaterniónoch , avšak, v balíčku TF existuje na prepočet funkcia s názvom, *quaternion_from_euler,* kde na vstupe očakáva hodnoty [x,y, φ] a výstupom z nej je prepočítaná karteziánska sústava na kvaternióny.

Následne môžeme zadefinovať správu pre odometriu, ktorá taktiež začína s hlavičkou, v ktorej sa zaznamenávajú tri hodnoty ako sú poradie správy, čas, kedy boli dáta namerané a názov súradnicového systému, pod ktorým laser vystupuje. V našom prípade je názov súradnicového systému „*odom*.“ Taktiež je potrebné doplniť aj *child_frame_id* čo je v našom prípade „*base_link*.“ potom musíme zadefinovať premenné pre *PoseWithCovariance*, ktorá sa z skladá už zo spomínanej karteziánskej sústavy a kvaterniónovej sústavy.

Na záver je dôležité vyplnenie dvoch vektorov obsahujúcich dáta o lineárnom a angulárnom pohybe robota, pričom oba vektory sa skladajú z troch premenných [x,y, *z*]. Do lineárnej vpíšeme rýchlosti iba pre x a y, a do angulárneho vpíšeme uhlovú rýchlosť ktorú robot nadobúda v z-ovej ose. Rýchlosť medzi aktuálnym a predchádzajúcim bodom sme vypočítali nasledovne:

$$velX = \frac{x_{k-1} - x}{t_{k-1} - t}$$

27

Kde velX určuje rýchlosť medzi bodom x a prechádzajúcou hodnotou x_{k-1}, k je časový index merania a t určuje čas, kedy boli namerané hodnoty x_{k-1} a x. Následne vieme vypočítať rýchlosť pre x. Pre velY je vzorec rovnaký, len je potrebné vymeniť konštanty x na y.

5.2 Implementácia Gmappingu do systému ROS

SLAM Gmapping nie je súčasťou systému ROS, aby sme s balíkom vedeli aj naďalej pracovať je nutné si balíček dotiahnuť do systému ROS a môžeme to spraviť dvoma spôsobmi. Prvý spôsob je nainštalovanie balíka pomocou ROS príkazu, kedy sa nám GMapping dotiahne pre všetky „workspace.“ Druhý spôsob je kedy si balíček stiahneme z portálu Github ako zložku a vložíme ju do nášho aktuálneho workspac-u.

Po implementovaní balíka do systému ROS je potrebné správne navolenie parametrov. Väčšine SLAM algoritmov je možné udať vstupné parametre pre mapovanie prosredia napríklad ako rozlíšenie mapy, maximálna veľkosť a podobne. Všetky potrebné parametre vieme dohľadať v dokumentácií balíčku. Avšak my budeme mapy medzi sebou porovnávať preto je potrebné zadať rovnaké rozlíšenie mapy a to zmenením nasledujúcich parametrov :

<param name="map_udpate_interval" value="1.0"/>

<param name="xmin" value="-0.5"/>

<param name="ymin" value="-0.5"/>

<param name="xmax" value="0.5"/>

<param name="ymax" value="0.5"/>

<param name="delta" value="0.05"/>

Kde xmin, xmax, ymin a ymax definujú veľkosť inicializačnej mapy, a parameter delta definuje rozlíšenie mapy. Taktiež sme museli skontrolovať vstupné parametre pre všetky súradnicové systémy či sa zhodujú aby ich vedel algoritmus správne vyčítať.

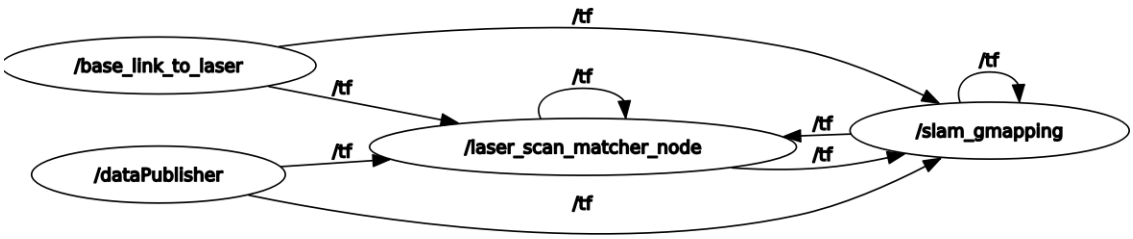
<param name="base_frame" value="base_link"/>

<param name="map_frame" value="map"/>

<param name="odom_frame" value="odom"/>

Avšak tieto hodnoty sme nemuseli meniť z dôvodu že takto sú už predvolené avšak ak by sa náš súradnicový systém nazýval inak je nutné ho zmeniť aj v parametroch SLAM-u.

Po nastavení parametrov môžeme spustiť balíček s SLAM na spustenie sme si vytvorili osobitný spúšťací súbor gmapping.launch. Spúšťajú sa nám všetky potrebné balíčky pre namapovanie prostredia ako sú statické transformátory, náš uzol a laser_scan_matcher. Úspešné prepojenie všetkých uzlov sme si skontrolovali pomocou rqt grafu.



Obr. 15 Rqt graf pre SLAM GMapping

5.3 Implementácia HectorSlam do systému ROS

Aj keď HectorSlam sa skladá z viacerých uzlov, všetky uzly sú v jednom balíčku čiže pri inštalácií máme rovnaké možnosti ako pri prvom SLAM algoritme. A to inštaláciu priamo do systému alebo stiahnutie balíčku z portálu GitHub. Po implementovaní balíčku do nášho workspacu je potrebné taktiež nastaviť parametre pre mapu, skontrolovať či sú správne zadefinované súradnicové systémy a parametre pre LiDAR.

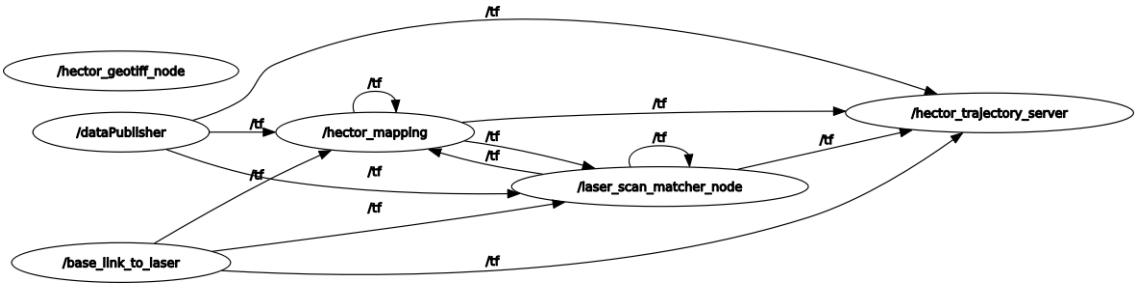
<arg name="pub_map_odom_transform" default="true"/>

<param name="map_frame" value="map" />

<param name="map_resolution" value="0.05"/>

Kde sme taktiež museli zadať rovnakú veľkosť rozlíšenia mapy aby sme zachovali ich veľkosť pri porovnaní. Následne sme museli zadefinovať súradnicový systém na „*map,“* pretože prednastavený názov je „*map_link*.“Taktiež tento algoritmus ponúkal možnosť vytvorenia statickej transformácie medzi súradnicovými systémami map a odom čo sme využili, keďže táto transformácia je statická a v čase nemení súradnice.

Po nastavení parametrov sme taktiež vytvorili dva spúšťacie súbory nazvané „hector.launch a mapping_hector.launch.“ V prvom spomínanom máme zadefinované spustenie všetkých potrebných uzlov ako je náš publisher, statické transformácie a samotný HectorSLAM, pričom v druhom súbore máme pre prehľadnosť vypísané všetky parametre, ktoré je možné nastaviť v aktuálnom algoritme. Následne po spustení si taktiež skontrolujeme správne prepojenia pomocou rqt grafu.



Obr. 16 Rqt graf pre HectorSlam

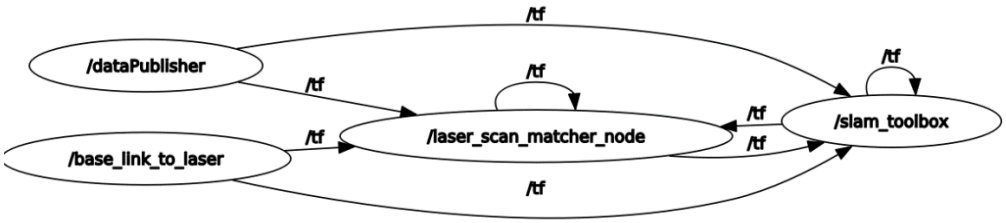
5.4 Implementácia SlamToolboxu do systému ROS

SlamToolbox sme si stiahli ako balíček s github serveru avšak aj tento balíček sa dá doinštalovať aj klasickým konzolovým príkazom. Po implementovaní do workspacu sme znova museli prekontrolovať všetky parametre. Dokumentácia k SlamToolboxu sa však nenachádza na oficiálne stránke ros.org, ale na portály github.

Po prekontrolovaní sme zadefinovali nasledujúce parametre :

<param name="odom_frame" value = "odom"/>
<param name="base_frame" value = "base_link"/>
<param name="resolution" value="0.05"/>

Kde sme taktiež pre konzistentnosť máp zadali rovnaké rozlíšenie mapy a zadefinovali sme jednotlivé súradnicové systémy. Po zadefinovaní sme vytvorili spúšťací súbor s názvom „slamToolbox.launch,“ kde sme zaradili všetky potrebné uzly pre vykonávanie SLAMu. Po spustení sme všetky jednotlivé väzby medzi uzlami skontrolovali príkazom rqt_graph.



Obr. 17 Rqt graf pre SlamToolbox

5.5 Implementácia Google Cartographeru do systému ROS

Implementovať googleCartographer je zložitejšie ako predošlé balíčky. Jeho inštalácia sa skladá z viacerých krokov ktorých postupnosť je nutné dodržať inak sa balíček nestiahne správne. Podrobný návod je spísaný v dokumentácií.[13] Avšak pri inštalácií sa využíva novší konzolový príkaz pre skompilovanie workspacu a to :

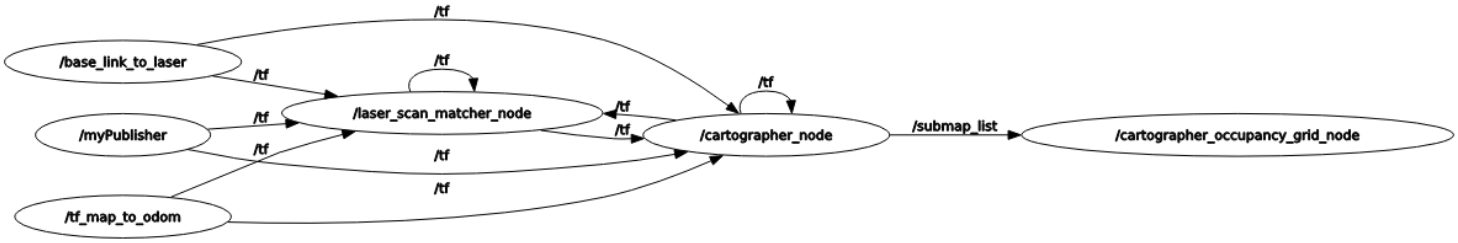
catkin_make_isolated

Pričom sa neodporúča použiť starý príkaz, mohlo by to viesť k zlému prepojeniu jednotlivých balíčkov. Po stiahnutí sa nám vo workspace zobrazia dva balíčky s názvami cartographer a cartographer_ros.

GoogleCartographer nemá možnosť upravovať vstupné parametre cez spúšťací súbor. Ale v adresári konfigurácií sa nachádzajú súbory s príponou .lua, kde sa nachádzajú príslušné vstupné argumenty. Keďže googleCartographer je možné použiť rôzne ako sme už spomínali pri jeho opisovaní je potrebné dôkladné zadefinovanie všetkých parametrov nasledovne pričom zvýrazníme len tie najpodstatnejšie:

map_frame = "map",
tracking_frame = "base_link",
published_frame = "odom",
odom_frame = "odom",
provide_odom_frame = true,
publish_frame_projected_to_2d = true,
use_pose_extrapolator = false,
use_odometry = true,
num_laser_scans = 1,

kde bolo potrebné zadefinovať všetky súradnicové systémy, provide_odom_frame nám znova zabezpečuje statickú transformáciu medzi súradnicovým systémom mapy a odometrie. *publish_frame_projected_to_2d* udáva že mapa bude iba 2D. *use_odometry* zavedieme že budeme využívať odometriu namiesto jednotky IMU. *Num_laser_scans* zadefinujeme počet použitých laserov. Pro zadefinovaní všetkých potrebných parametrov vytvoríme spúšťací súbor, kde zahrnieme všetky potrebné balíčky ako cartographer, nami vytvorený publisher, a ďalšie potrebné



uzly. Po spustení si skontrolujeme správne prepojenie uzlov za pomoci rqt grafu.

Obr. 18 Rqt graf pre GoogleCartographer

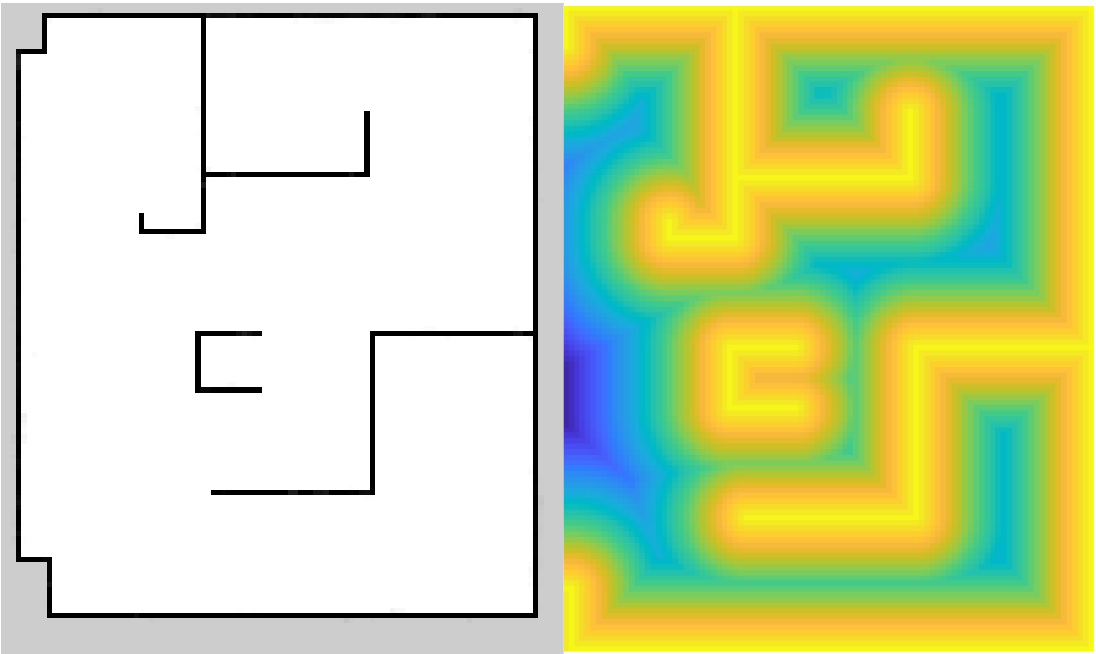
Na obrázku vyššie môžeme vidieť že všetky uzly sú správne prepojene pričom SLAM algoritmus využíva na vykreslenie máp uzol s názvom */cartographer_occupancy_grid_node.*

6 Experimentálne porovnanie SLAM metód

V nasledujúcej kapitole vystavíme SLAM algoritmy rôznym experimentálnym testom, ktoré následne aj vyhodnotíme. Taktiež sa pozrieme na výpočtovú záťaž procesoru pri spracovávaní mapy jednotlivých SLAM algoritmov. Všetky experimenty sme vykonali v priestoroch NCR na Fakulte elektrotechniky a informatiky STU v Bratislave.

6.1 Vyhodnotenie jednotlivých metód pri experimente 1

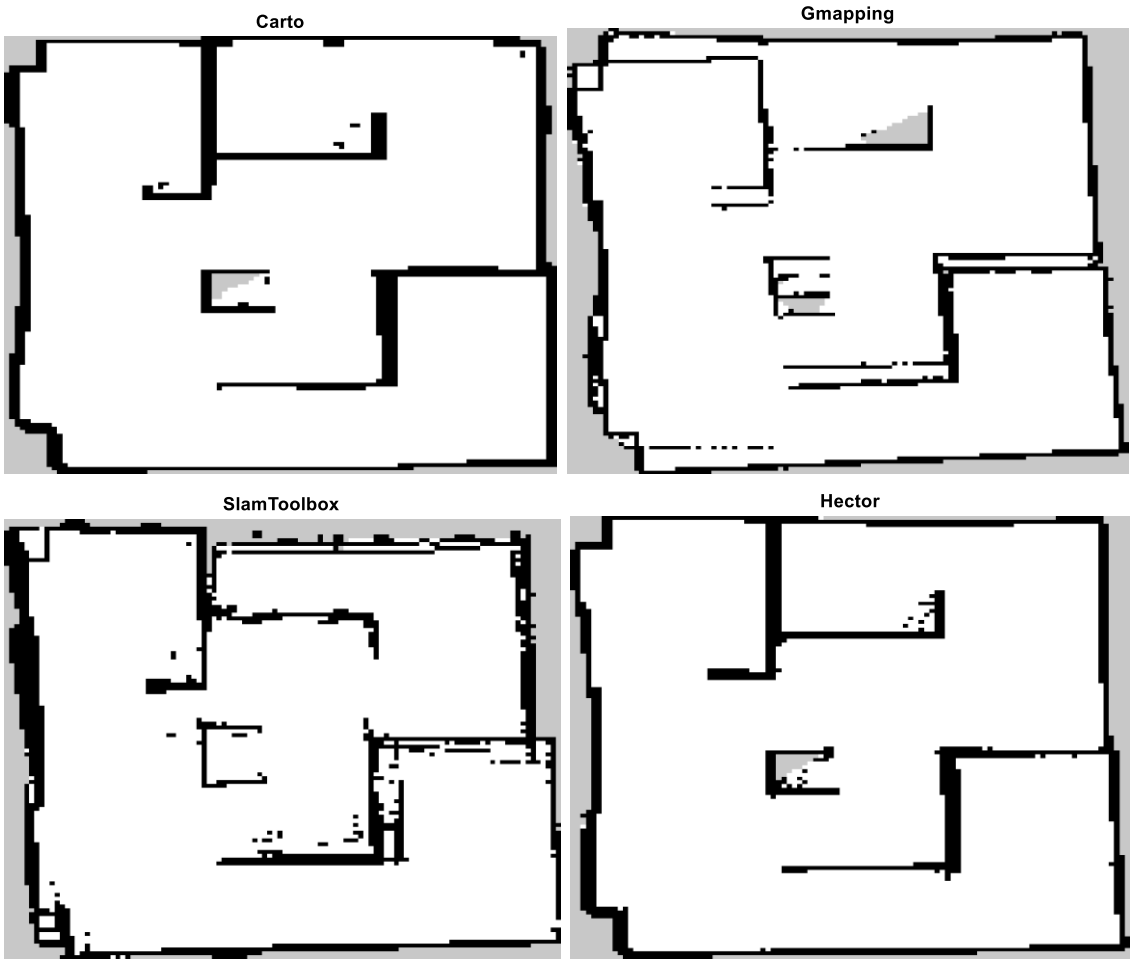
Prvý experiment sme vykonali v bludisku v NCR, avšak, aby sme mapy vytvorené pomocou SLAM algoritmov vedeli vyhodnotiť, bolo potrebné vytvoriť ideálnu mapu prostredia, ktorú môžeme vidieť nižšie. Následne sme z nej vytvorili dištančnú mapu. Dištančná mapa je mapa, kde každý voľný pixel je ohodnotený hodnotou takou, aká zodpovedá najbližšej stene k tomuto pixelu. Zobrazenie dištančnej mapy môžeme vidieť taktiež na obrázku nižšie.



Obr. 19 Ideálna mapa a dištančná mapa

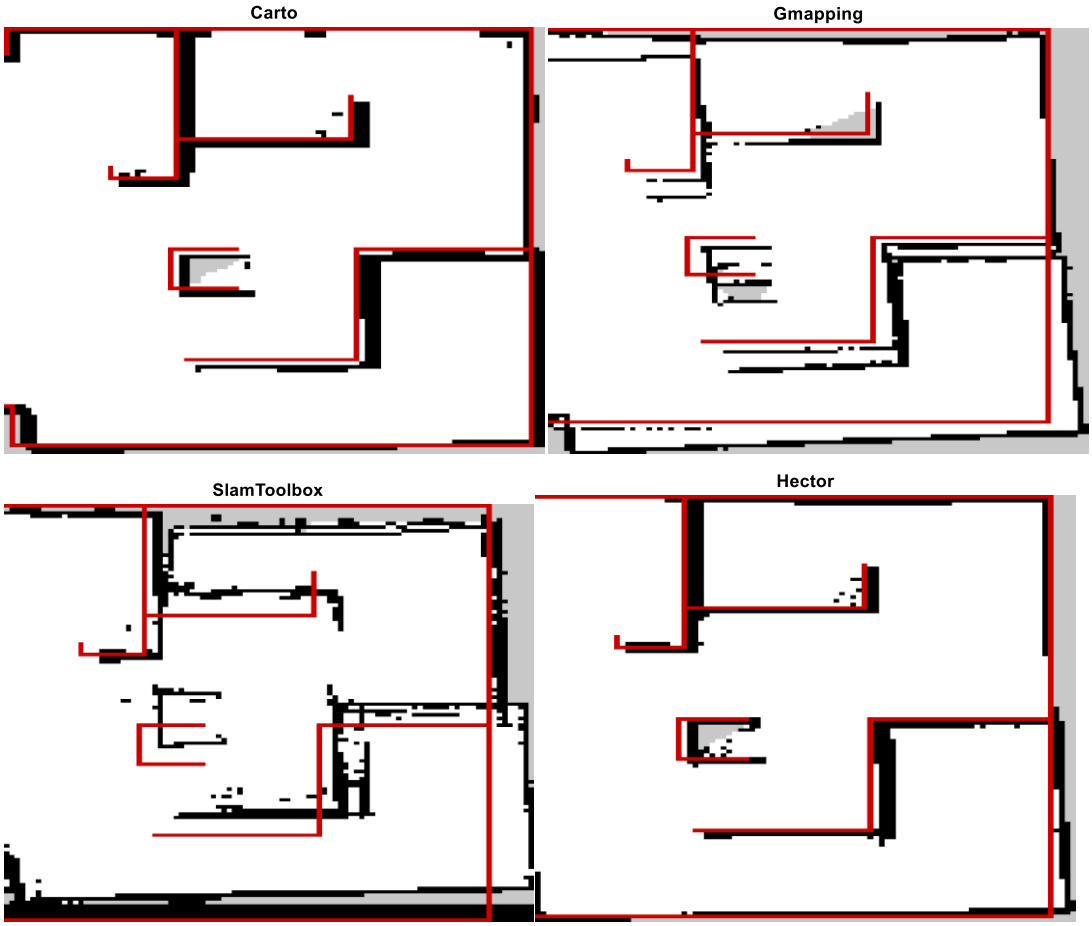
Ľavá stena sa na dištančnej mape nezobrazuje, nakoľko z dôvodu nepresnosti ju nebolo možné správne prepísať do ideálnej mapy. Z tohto dôvodu sme sa rozhodli, že

ľavú stenu pri vyhodnocovaní nezoberieme do úvahy, preto môžeme vidieť že sa ľavá stena na dištančnej mape nezobrazuje.



Obr. 20 Mapa bludiska Cartographer, Gmapping, SlamToolbox, HectorSlam

Pomocou každého SLAM algoritmu sme následne vytvorili z datasetu mapy, ktoré môžeme vidieť na obrázku vyššie. Zoradené sú zľava hore Google Cartographer, Gmapping, vľavo dole SlamToolbox a HectoSlam. Už len z vizuálneho hľadiska si môžeme všimnúť, že najväčšie problémy s mapovaním mal SlamToolbox a pri Gmappingu môžeme vidieť jemné natočenie stien.



Obr. 21 Prekrytie máp ideálnou mapou

Avšak, aby sme vypočítali zhodu medzi ideálnou mapou a jednotlivými mapami zo SLAM algoritmov, je potrebné dané mapy prekryť, ako sme spravili na Obr. 21 pre lepšiu vizualizáciu, a následne vypočítať chybu.

Tabuľka 3 Vyhodnotenie chýb SLAM algoritmov

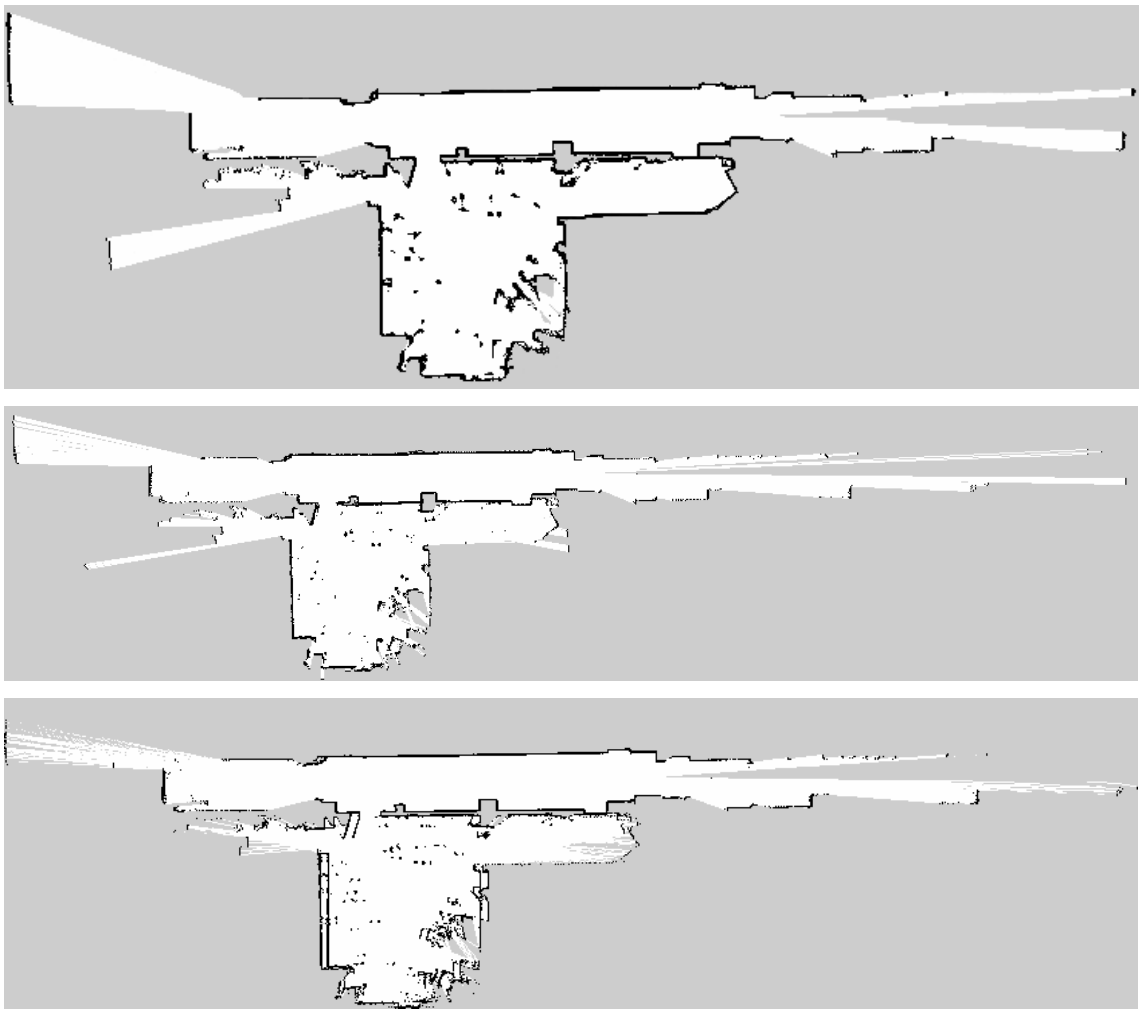
	Priemerná chyba na pixel [m]	Počet obsadených pixelov
GoogleCartographer	0.077	857
GMapping	0.177	700
SlamToolbox	0.252	1019
Hector	0.062	933

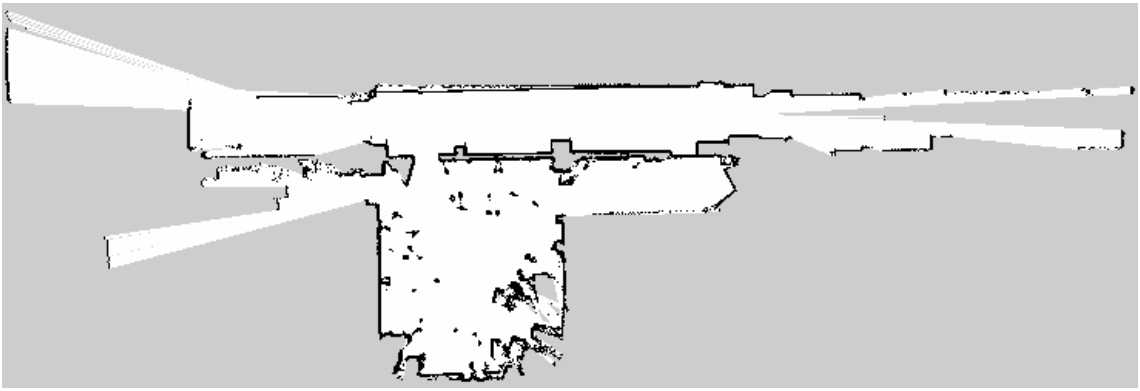
V Tabuľka 3 sme vyčíslili priemerné chyby na obsadený pixel a celkový počet obsadených pixelov. Z aktuálneho porovnávacieho experimentu je zjavné, že bludisko

dokázali najvierohodnejšie opísať algoritmus HectorSlam s priemernou chybou na pixel 6.2 cm a GoogleCartographer s priemernou chybou na pixel 7.7 cm. Najhoršie bludisko opísal SlamToolbox, čo bolo zjavné už z vizuálneho hľadiska s chybou na pixel 25.2 cm a Gmapping mal chybu na pixel 17.7 cm.

6.2 Vyhodnotenie jednotlivých metód pri experimente 2

SLAM algoritmy majú vo všeobecnosti väčší problém s mapovaním veľkých prázdnych plôch a dlhých rovných chodieb. V takýchto priestoroch vzniká najviac chýb a nepresností v tvorenej mape. Nasledujúci experiment sa zameriava na prepojenie chodby s väčšou miestnosťou. Na záver experimentu prekryjeme mapovanú chodbu so statickým snímkom chodby vytvoreným z lasera.



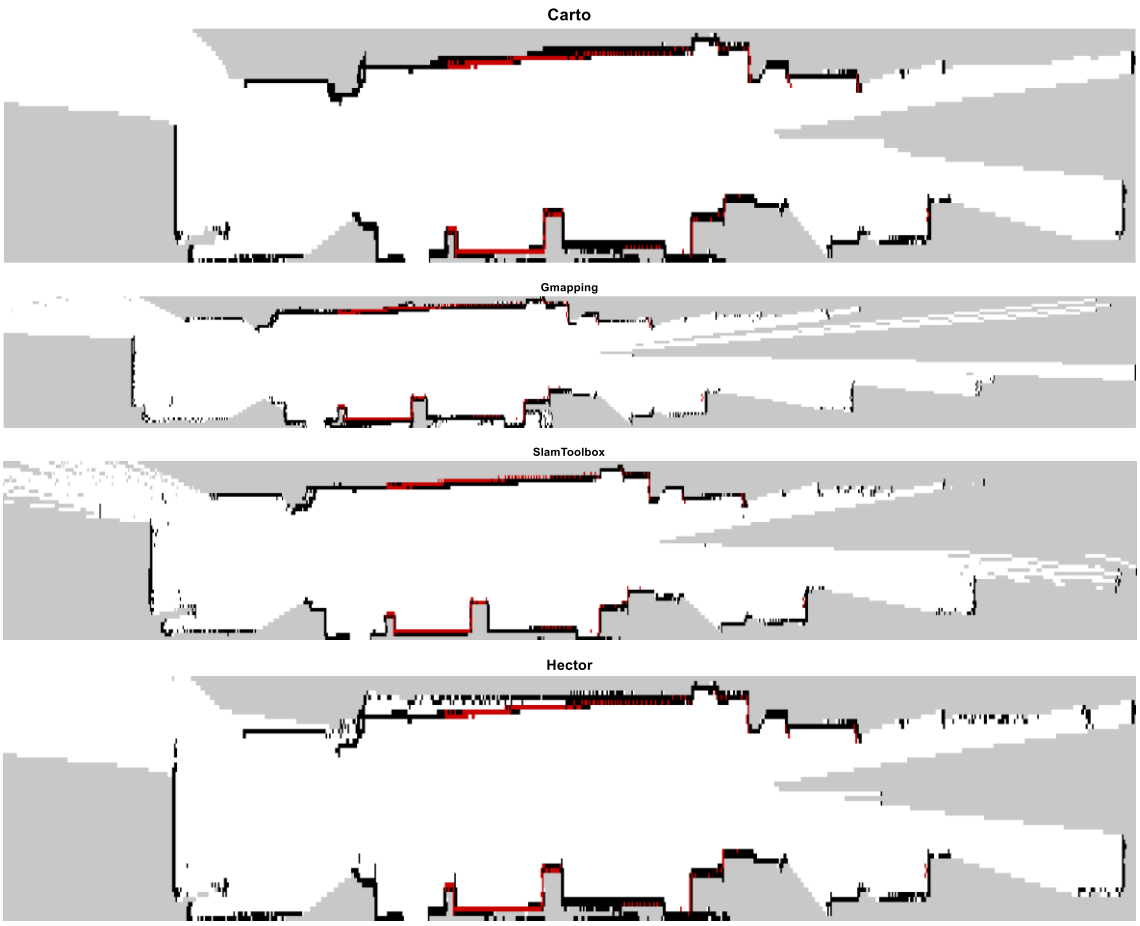


Obr. 22 Z hora GoogleCartographer, Gmapping, SlamToolbox, HectorSlam

Po namapovaní priestorov a ich jednotlivých zobrazeniach za pomoci rôznych SLAM algoritmov môžeme pozorovať rozdiely. GoogleCartographer zostrojil mapu bez zdvojení alebo nežiadúcich posunov. GMapping zostrojil taktiež rovnako opisujúcu mapu, ale môžeme si všimnúť jemné chyby, kde mapa pokračuje aj za stenou, avšak, dané chyby sú malé a pri opätovnom použití mapy by ich reaktívna lokalizácia minimalizovala. Pri mape vytvorenou algoritmom SlamToolbox vidíme zdvojenie stien v miestnosti. Dané zdvojenie mohlo nastať z dôvodu, že algoritmus odchytil bod, ktorý sa v čase mohol posunúť, pretože v miestnosti sa počas mapovania nachádzali ľudia, ktorí obmedzili svoj pohyb, hoci pripúšťame aj možnosť, že mohol nastať jemný posun. HectorSlam v mapovaní miestnosti na vizuálny pohľad rovnako nepochybil. Avšak, pri mapovaní chodby si môžeme všimnuť na vrchnej stene jemné posunutie, kedy nevedel presne vyniesť správnosť steny do mapy.

Obr. 23 Statický záznam z laseru

Na obrázku vyššie môžeme vidieť statický záznam z LiDARU Hokuyo v chodbe. Následne budeme porovnávať nepresnosti vyskladania chodby za pomoci algoritmu a statického záznamu.



Obr. 24 Prekrytie máp statickým záznamom z LiDARU

Na obrázku vyššie, kedy sme prekryli vytvorené mapy statickým záberom z LiDARU, si môžeme všimnúť, že algoritmy GMapping a SlamToolbox sa takmer zhodujú so statickým záznamom z kamery. Mapa, ktorú vytvoril algoritmus Google Cartographer, sa dostatočne zhoduje, čo môžeme vidieť v porovnaní medzi stenou na mape a stenou zo statického snímku. Pri mape, ktorú vytvoril HectorSlam, môžeme vidieť dvojitú stenu, pričom jedna z nich sa zhoduje so statickým záznamom, avšak, daná chyba so zdvojením steny sa na mape nachádza a je potrebné ju vziať do úvahy ako chybu.

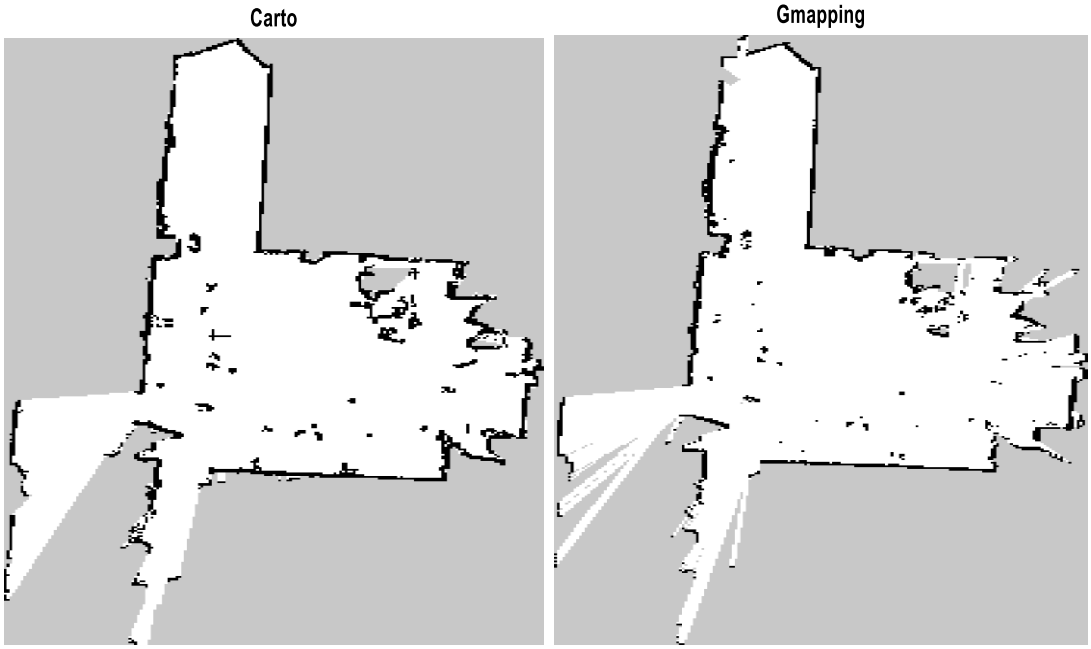
Tabuľka 4 Vyčíslená percentuálna zhoda

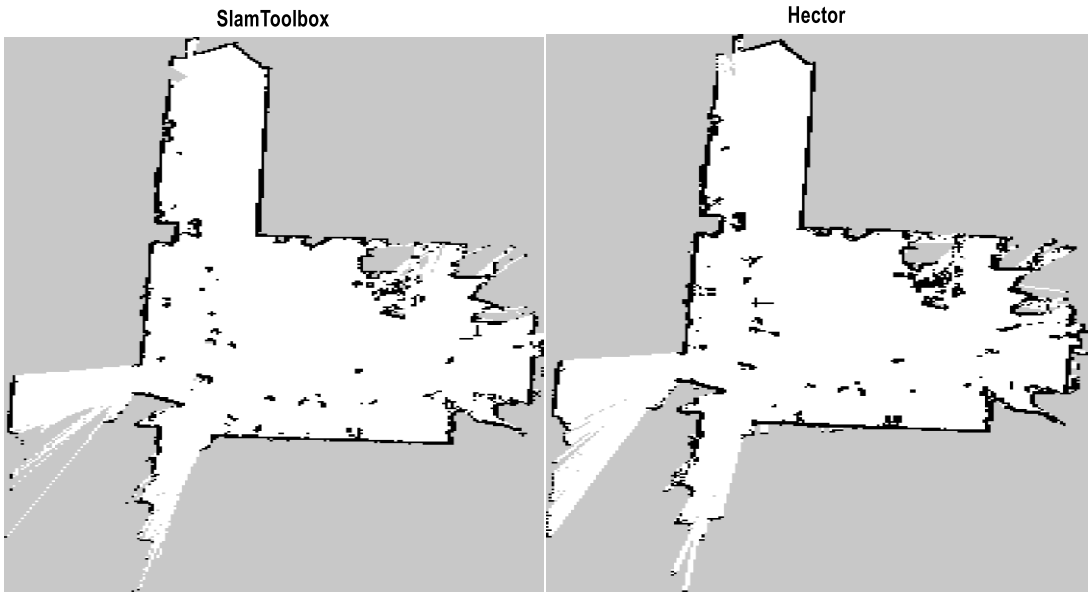
	Počet zhodných pixelov
GoogleCartographer	91.33 %
GMapping	66.91 %
SlamToolbox	69.51 %
Hector	87.51 %

Po presnejšom vyčíslení zhody, kde sme vyčíslili, kedy sa pixely steny na mape zhodujú s pixelom steny v statickej chodbe, môžeme vidieť, že SlamToolbox a GMapping dlhú chodbu opisujú vierohodne s porovnaním so záznamom statickej chodby. GoogleCartographer dosahuje najvyššie percento zhody, pričom vytvoril mapu najpresnejšiu k statickému snímku. HectorSlam aj pri chybe so zdvojenou stenou preukázal vysokú zhodu so statickým záznamom z Hokuyo LiDARU. Počet obsadených pixelov v zázname z Obr. 23 je 269.

6.3 Vyhodnotenie jednotlivých metód pri experimente 3

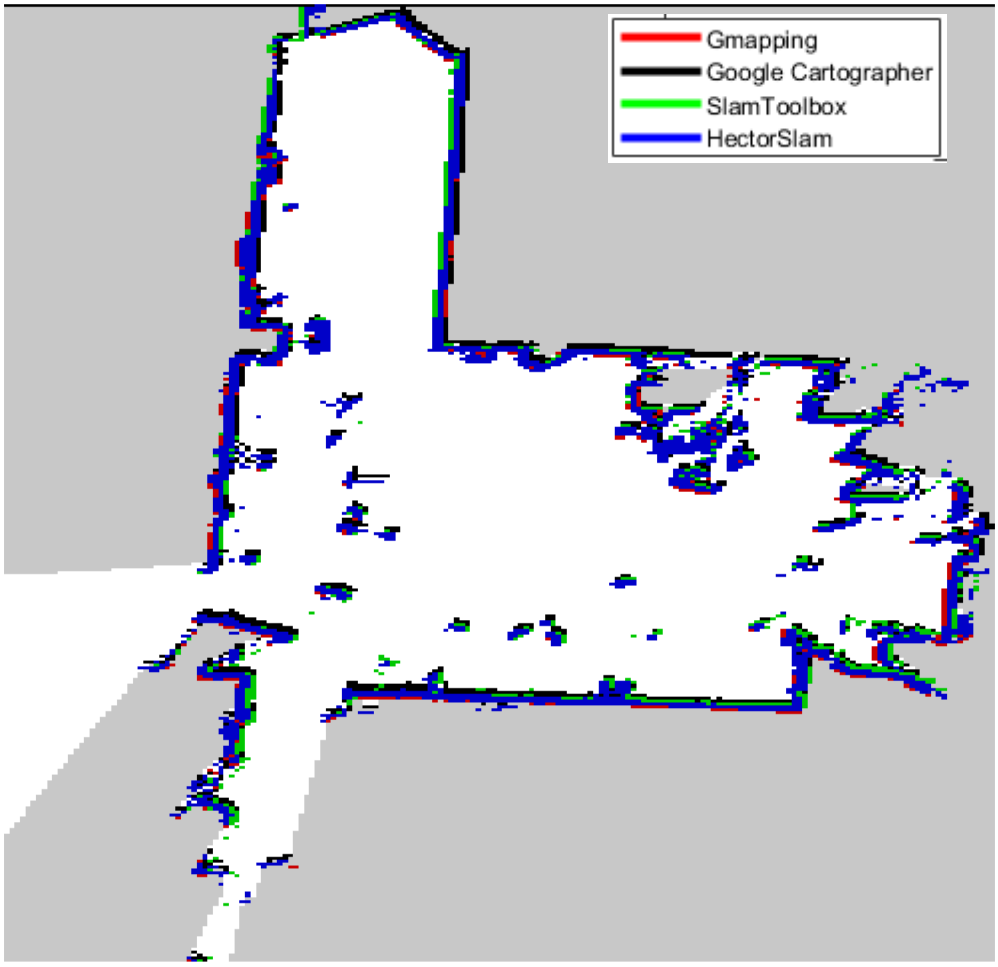
V aktuálnom experimente vytvoríme dataset z miestnosti, v ktorej sa nachádza viacero kancelárskych prekážok, ako sú stoličky a stolíky. V miestnosti sa taktiež pri mapovaní nachádzali ludia, ktorí mali stálu polohu. V experimente sa pokúsime preukázať, ako algoritmy namapujú dané priestory. Avšak, z dôvodu nevedomosti rozmerov miestnosti a presnosti prekážok, nie je možné porovnať dané vyhotovenia máp s reálnou mapou, ale vieme porovnať mapy medzi.





Obr. 25 Vytvorené mapy pomocou SLAM algoritmov

Rozdiely vo vyobrazených mapách sú nepatrné, pre ich lepšie zvýraznenie sme jednotlivé mapy prekryli.



Obr. 26 Prekyrie máp všetkých SLAM algoritmov

Na obrázku vyššie, kde sú všetky štyri mapy z Obr. 25 môžeme vidieť, že sú konzistentné. Mapy medzi sebou nevykazujú skoro žiadne zásadné rozdiely, a teda si dovolíme tvrdiť, že všetky algoritmy mapovali danú miestnosť rovnako. Napriek tomu, že existujú jemné odlišnosti medzi pixelmi, tak vieme, že jeden pixel činí 5cm, preto si dovolíme tvrdiť, že dané rozdiely, ktoré sme spozorovali, sú len minimálne.

6.4 Vyhodnotenie jednotlivých metód pri experimente 4

V tomto experimente nás zaujíma spracovanie mapy jednotlivými algoritmami, keď počas mapovania pozmeníme polohu jednej prekážky v strede miestnosti. Dataset sme vytvorili v laboratóriu firmy Photoneo s.r.o. na Fakulte elektrotechniky a informatiky. V experimente sme najskôr prešli z ľavého okraja miestnosti do pravého okraja miestnosti, pričom keď sa robot vracal naspäť na štartovaciu pozíciu, sme odstránili prekážku zo stredu miestnosti.

Obr. 27 Laboratórium Photoneo

Na Obr. 27 môžeme vidieť zobrazenie mapy, v polovici behu, kedy robot prešiel do pravého rohu, kde zastavil. Ako si môžeme všimnúť, mapy sú veľmi podobné a nevykazujú značné známky odlišnosti. Následne sme odstránili prekážku zo stredu a s robotom sme sa vrátili na štartovaciu pozíciu.

Obr. 28 Laboratórium Photoneo 2

Výsledné mapy môžeme vidieť na Obr. 28. Experimentom sme sa snažili preukázať správanie jednotlivých SLAM algoritmov na zmenu prostredia pri mapovaní. Ako si môžeme všimnúť, Gmapping a SlamToolbox zmenu prostredia nespracovali korektne a začali dotvárať novú mapu, ktorá už nezodpovedala reálnej, ak predpokladáme, že reálna mapa je zobrazená na Obr. 27. Pre lepšie porovnanie sme sa rozhodli prekryť jednotlivé mapy.

Na Obr. 29 si môžeme všimnúť prekrytie jednotlivých máp z Obr. 27 a Obr. 28. Prekrytím obrázkov sme sa snažili preukázať odlišnosť máp po mapovaní prostredia, ak sa prostredie v čase zmenilo. K danému výsledku môže prispieť fakt, že sme mapovali veľkú prázdnu miestnosť, a teda jednotlivé algoritmy už nevedeli správne napasovať

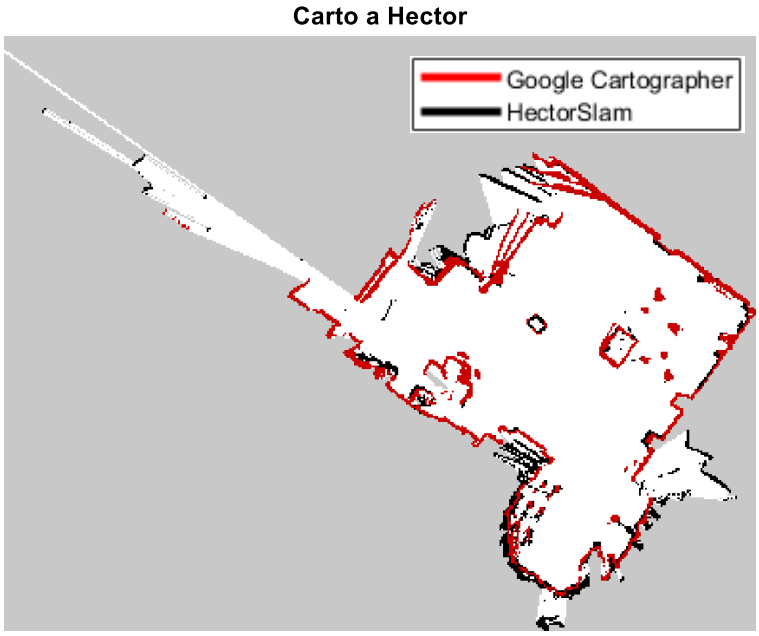
nové dáta z algoritmov na staré a vyhodnotili to ako pokračovanie miestnosti, čo má za následok nesprávne premapovaný priestor. Algoritmy GoogleCartographer a HectorSlam sa so zmenou prostredia vysporiadali o niečo lepšie, avšak, tiež sa tam nájdu chyby, ktoré na Obr. 27 neevidujeme.

Obr. 29 Prekyrtie máp jednotlivých SLAM algoritmov

Vo výsledku môžeme vidieť, že SLAM algoritmy ako SlamToolbox a Gmapping neuspeli v mapovaní priestoru, a pretože je mapa nečitateľná a nezobrazuje reálne miesto. Avšak, Cartographer a HectorSlam sa medzi sebou líšia len v maličkostiach, kvôli čomu žiaľ nevieme presne vyhodnotiť, ktorý zo SLAM algoritmov bol presnejší, ale vieme povedať, že robot by sa vedel podľa daných máp orientovať a navigovať v priestore.

Pre lepšie porovnanie týchto dvoch výsledkov sme sa ich navzájom prekryť, aby sme videli aké rozdiely medzi sebou tieto algoritmy majú. Na Obr. 30 HectorSlam prekrytý GoogleCartographerom môžeme vidieť mapu vytvorenú pomocou HectorSlam

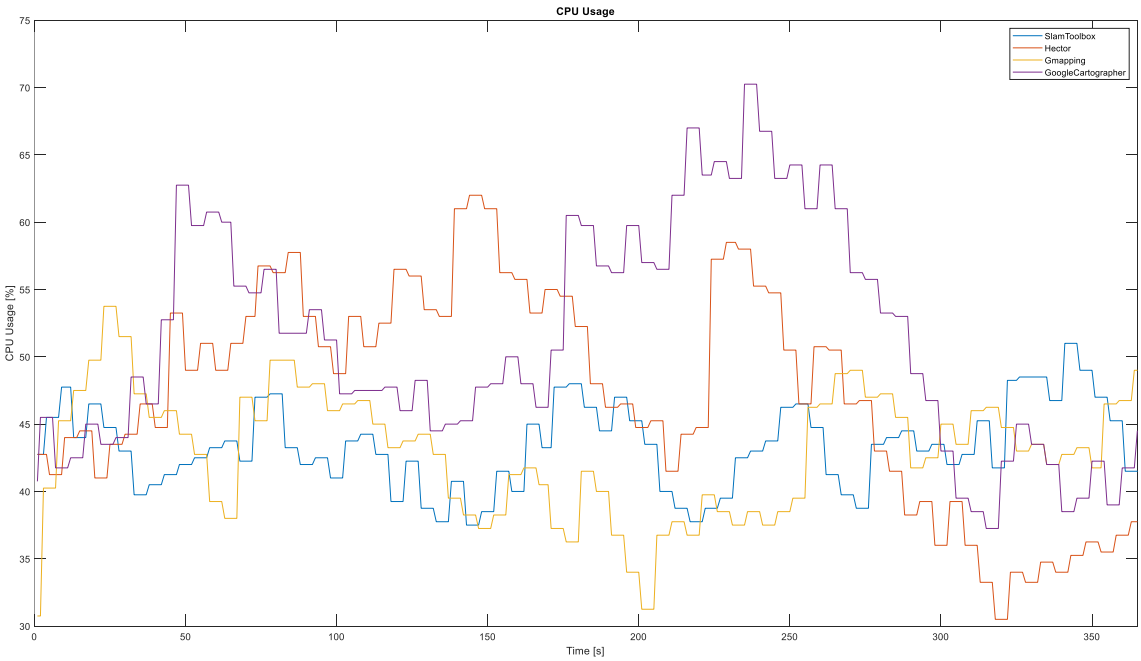
(čierne pixely) a GoogleCartographer (červené pixely). Vytvorené mapy sú takmer identické až na pár detailov.



Obr. 30 HectorSlam prekrytý GoogleCartographerom

Tým pádom vieme, že v našom prípade HectorSlam a Google Cartographer najlepšie reagovali na zmenu prostredia počas mapovania, dokázali túto zmenu najlepšie vyhodnotiť a vykresliť mapu takmer zhodnú s prvým mapovaním.

6.4.1 Náročnosť SLAM algoritmov na procesor



Obr. 31 Vyťaženie procesora pri jednotlivých SLAM algoritmoch

Pri vytváraní mapy pomocou všetkých SLAM algoritmov sme zaznamenávali ich vyťaženie na procesore, pričom sme sa snažili minimalizovať všetky ostatné procesy bežiace na našom systéme. Pri meraní bol spustený len operačný systém Ubuntu 20.04 a samotný ROS s SLAM algoritmom. Meranie bolo vykonané na 4 jadrovom procesore Intel Core i5. Zaznamenávali sme zaťaženie od začiatku tvorby mapy až po jej ukončenie.

Tabuľka 5 Zaťaženie CPU pri jednotlivých SLAM algoritmoch

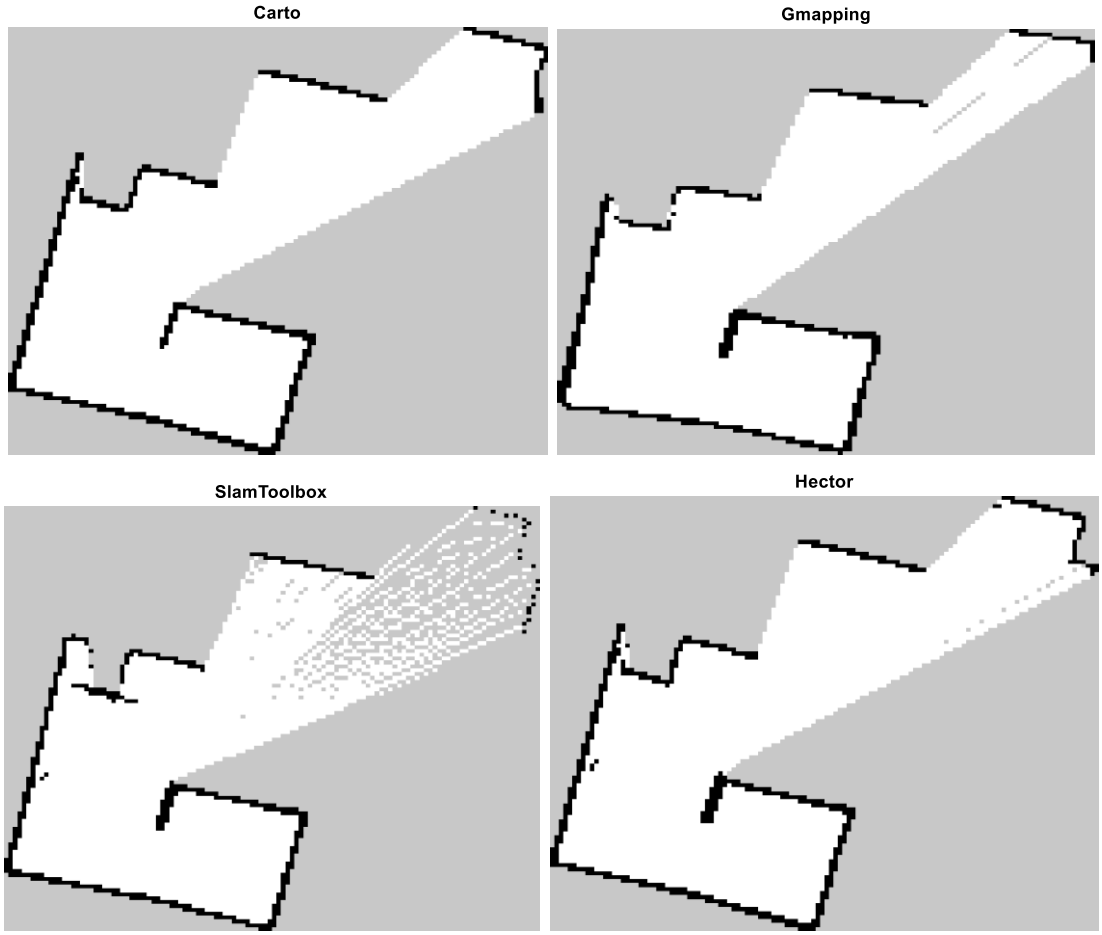
	Priemerné zaťaženie	Minimálne zaťaženie	Maximálne zaťaženie
GMapping	43.03 %	30.75 %	53.75 %
HectorSLAM	47.2 %	30.5 %	62 %
SlamToolbox	43.3 %	37.5 %	51 %
Google Cartographer	51.33 %	37.25 %	70.25 %

Z vyššie uvedenej tabuľky vyplýva, že maximálny zaťaženie dosiahol Google Cartographer a to 70,25% a najnižšie zaťaženie evidujeme pri algoritme HectorSlam. Hoci, priemerné zaťaženie mali všetky SLAM algoritmy podobné v rozpätí 40 až 55 %, ale výpočtovo náročnejší je algoritmus GoogleCartographer. SlamToolbox alebo GMapping spotrebovali priemerne zhruba o 10% výkonu menej počas celého mapovania.

6.5 Vyhodnotenie jednotlivých metód pri experimente 5

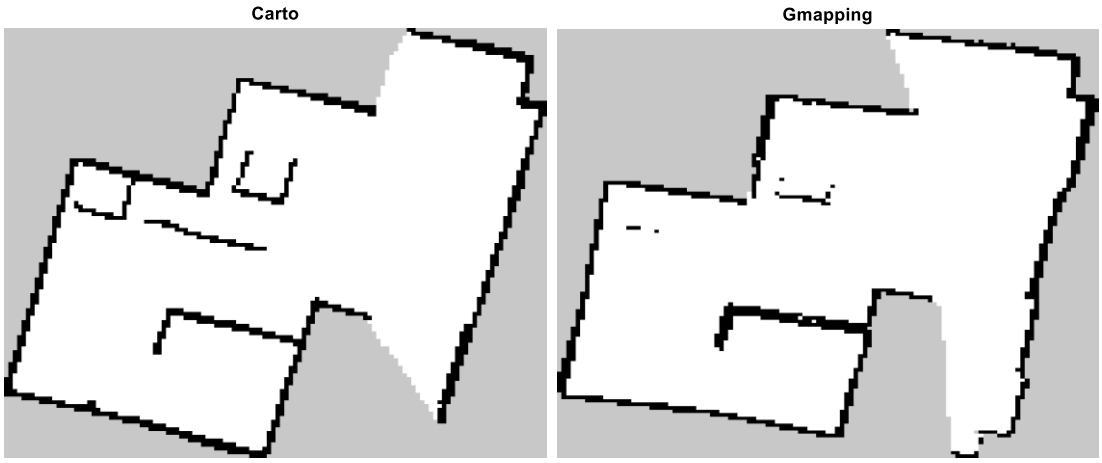
Pri tomto experimente sme znova využili priestory bludiska, ako pri experimente v kapitole 6.1. Avšak, tentokrát sa počas celého mapovania sa v priestore s robotom nachádzala aj dynamická prekážka. Táto prekážka menila svoju polohu zároveň s robotom. Pri danom experimente nás zaujímalo ako jednotlivé SLAM-y budú reagovať a spracovávať takúto prekážku a ako to ovplyvní tvorbu mapy.

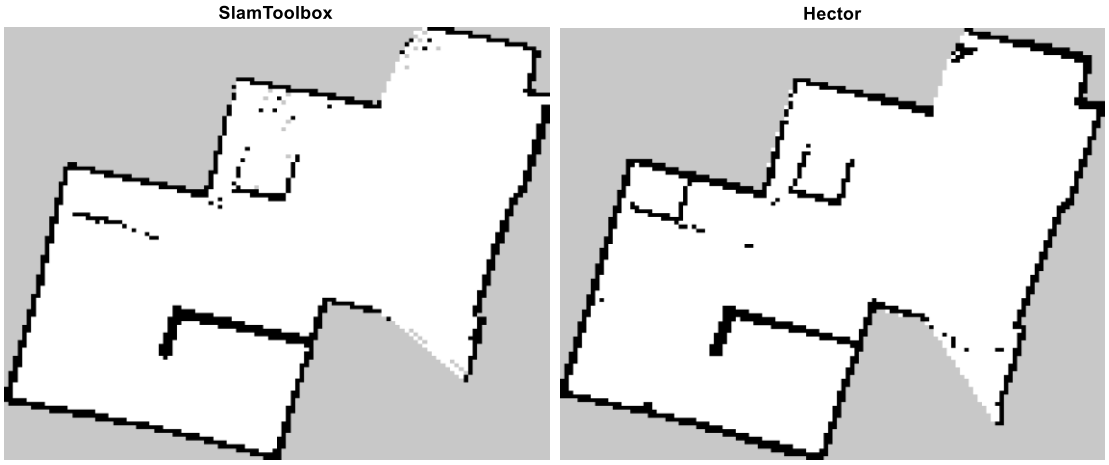
Na Obr. 32 vidíme znova štyri mapy zo štyroch SLAM algoritmov. Všetky štyri mapy sme sa pokúsili zachytiť pri rovnakej vzorke.



Obr. 32 Prvý krok experimentu

V prvom zachytenom kroku nevidíme medzi mapami žiadny rozdiel. Všetky algoritmy dynamickú prekážku zachytili a miesto za ňou vykreslili ako neznáme.

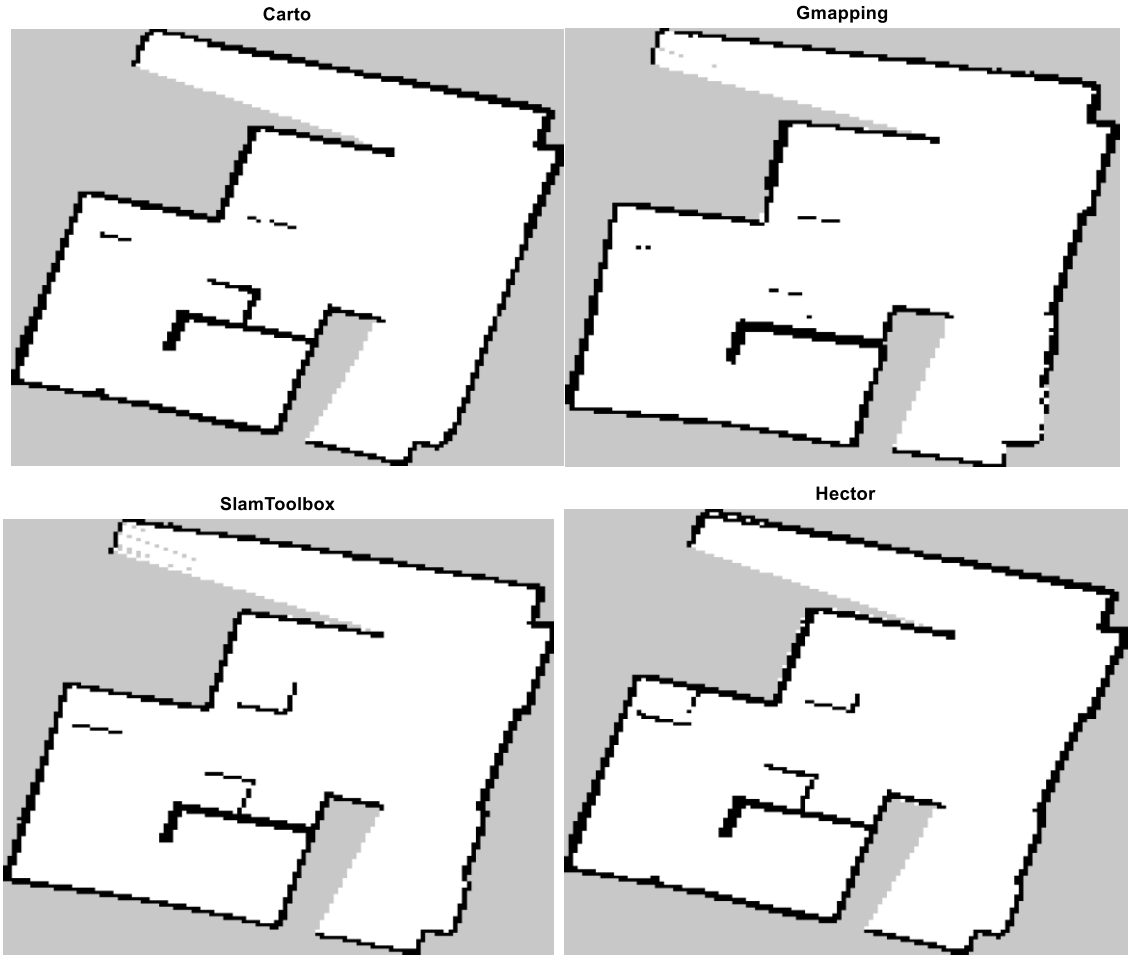




Obr. 33 Druhý krok experimentu

V druhom kroku na Obr. 32 a Obr. 33 Druhý krok si môžeme všimnúť pohyb dynamickej prekážky. Gmapping mal problém zobraziť jej pohyb a zväčša ju odfiltroval a zobrazil na mape len malé útržky. Cartographer nám vyobrazil celú stenu a aj jej poslednú polohu pred zachytením mapy v druhom kroku. Na mape algoritmu HectorSlam môžeme vidieť začiatočný a koncový bod dynamickej prekážky pred zachytením danej mapy, ale pohyb zobrazený na mape, bol odfiltrovaný. Pri SlamToolboxe vidíme koncový pod dynamickej prekážky pri zachytení mapy.

Na Obr. 34 vidíme finálnu mapu s dynamickou prekážkou. GoogleCartographer eliminoval predchádzajúce polohy dynamickej prekážky a zachoval iba poslednú polohu, ktorú aj vyobrazil. Gmapping odfiltroval skoro všetky polohy dynamickej prekážky a je ťažko rozpoznateľné, kde sa daná prekážka mohla nachádzať, pretože pixely reprezentujúce prekážku mohli predstavovať šum alebo nepresnosť LiDARu. SlamToolbox a Hector majú finálnu mapu veľmi podobnú, kedy nechali zachované tri polohy, kde dynamická prekážka nejaký čas pretrvala.



Obr. 34 Tretí krok experimentu

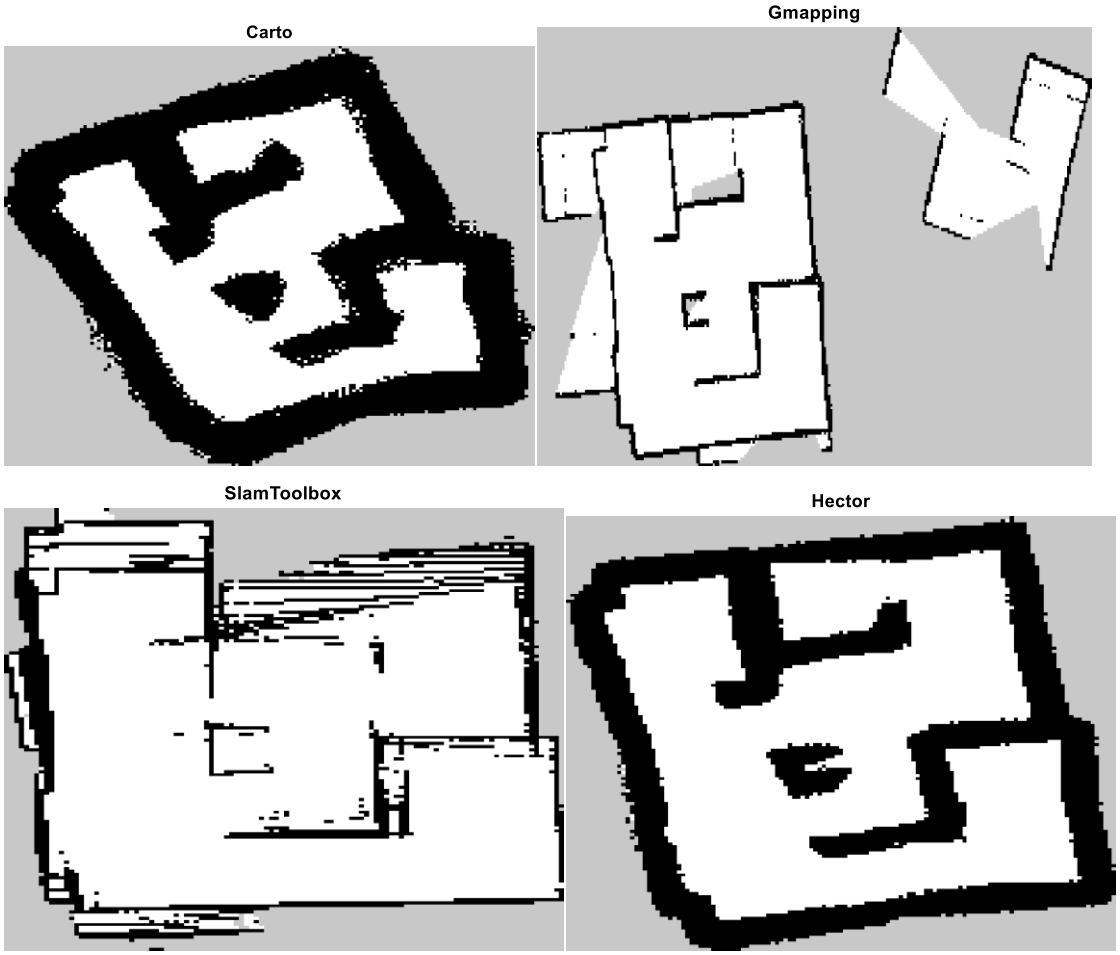
6.6 Vyhodnotenie metód pri experimente so zašumeným LiDAR-om

V nasledujúcom experimente sa pokúsime zistiť hraničné zašumenie dát z LiDARU, pri ktorých vedia algoritmy zostrojiť mapu. Dataset sme použili rovnaký ako v podkapitole 6.1 a to dataset z bludiska.

Tabuľka 6 Veľkosť aplikovaného šumu

	Veľkosť šumu [m]
GoogleCartographer	-0.35m;0.35m
GMapping	-0.08m;0.08m
SlamToolbox	-0.065;0.065
Hector	-0.2;0.2

Počas experimentov sme zistili maximálne možné šumy, s ktorými vedia algoritmy pracovať a ich veľkosť sme zaznačili do Tabuľka 6.



Obr. 35 Výsledné mapy s pridaným šumom na dátach z LiDAR-u

Google Cartographer zvládol šum v rozmedzí <-0.35m;0.35m>, pri ktorom ešte vedel zostrojiť čitateľnú mapu. Aj keď je zjavné, že mapa prostredia už nebude presná, nakoľko je viditeľný jej značný posun, je pozoruhodné, že pri takomto veľkom zašumení daný algoritmus vedel zostrojiť mapu.

Algoritmus GMapping už pri šume v rozmedzí <-0.08m;0.08m> začal vykazovať známky zle rozpoznaných príznakov mapy a nevedel jednotlivé snímky dostatočne spájať, čo má za následok tvorbu druhej mapy, hneď vedľa mapy bludiska.

SlamToolbox vytvoril mapu pri šume v rozmedzí <-0.065m;0.065m>, kde však už môžeme pozorovať veľký posun mapy. Ak sme šum ešte zväčšili, mapa sa následne stala nečitateľnou.

Na algoritmus HectorSlam sme aplikovali šum v rozmedzí <-0.2m;0.2m>, pričom môžeme vidieť podobné správanie, aké mal algoritmus Cartographer, kde sa

nám pri väčšom šume sa nám zväčšovali steny, ale mapa si zachovala svoj tvar. Pokiaľ sme však daný rozsah zväčšili, mapa sa už stala nečitateľnou.

Záver

Pred začatím práce sme sa oboznámili s úlohami potrebnými pre porovnanie SLAM algoritmov v prostredí ROS. Mimoriadne dôležitou časťou bolo bližšie sa zoznámiť aj so systémom ROS a jeho súčasťami a nástrojmi. Nevyhnutnou súčasťou bolo dôkladné naštudovanie problematiku SLAM algoritmov a zanalyzovať aspoň tri SLAM algoritmy, implementovať ich do systému ROS a experimentálne otestovať ich funkčnosť.

Vykonali sme rozsiahlu analýzu dostupných 2D SLAM algoritmov, ktoré vyhovovali našim podmienkam akými boli implementačný systém, verzia systému ROS a vstupné parametre pre tieto algoritmy. Vybrali sme 7 kľúčových algoritmov, avšak, kvôli zlej dokumentácií sme 3 algoritmy nedokázali implementovať. Pri výbere sme taktiež zohľadnili aj rok vydania daného balíčku, pričom sme sa snažili vybrať tie najnovšie a porovnať ich s najznámejším SLAM algoritmom v systéme ROS1, ktorým je GMapping. Ďalej bolo potrebné si dôkladne prejsť dokumentáciu ku každému z vybraných algoritmov, kvôli nastaviteľným parametrom každého z nich, ich následnej úprave a správnom nastavení pre naše použitie.

Museli sme sa oboznámiť so systémom ROS, s jeho funkciami a komunikáciou medzi jednotlivými balíčkami kvôli tvorbe vlastného uzla, ktorý spracováva namerané dáta z odometrie robota a LiDAR-u. Následne tieto dáta spracovávame a publikujeme pre SLAM algoritmy. Avšak, pre správne výpočty odometrie a nastavenia LiDAR-u sme sa museli zoznámiť s hardvérom.

Po naštudovaní všetkých potrebných parametrov a vytvorení nášho uzla v systéme ROS sme prešli k implementácií jednotlivých SLAM-ov. Google Cartographer je veľmi kompatibilný balíček, avšak, jeho implementácia do systému ROS je náročnejšia, pretože je potrebné dôkladné nastavenie všetkých parametrov a kvôli jeho podpore pre 2D a 3D lasery je to omnoho zložitejšie, ale po správnej konfigurácií sa nám podarilo daný balíček správne implementovať. SlamToolbox je nový SLAM algoritmus, ktorého implementácia bola jednoduchšia a v náročnosti by sme ho mohli porovnať s implementáciou algoritmu GMapping, kde je nevyhnutné okrem stiahnutia balíčka nastaviť potrebné parametre pre našu aplikáciu. Oba tieto balíčky sme implementovali do systému ROS. HectorSlam sme implementovali ako posledný algoritmus. Ako jediný je rozdelený do viacerých uzlov, ktoré medzi sebou

komunikujú. Po nastavení správnych parametrov sa nám podarilo aj HectorSlam správne implementovať do systému ROS.

V poslednej kapitole sme jednotlivé SLAM algoritmy podrobili niekoľkým experimentálnym testom, kde sme preverovali ich presnosť pri zostrojovaní mapy, spracovanie mapy pri dynamických prekážkach a v miestach, ktoré sú zložité na mapovanie, ako sú dlhé chodby a väčšie prázdne miestnosti. Google Cartographer dosiahol dobré výsledky vo viacerých testoch. Dokázal spracovať mapu pri obrovskom zašumení LiDAR-u a taktiež sa vie vysporiadať s dynamickými prekážkami pri mapovaní, čo dokazujú predošlé experimenty. Avšak, v presnosti mapovania ho predbehol algoritmus HectorSlam, ktorý bol o 1,5 cm na pixel presnejší, čo dokazuje vyhodnotenie v Tabuľka 3 Vyhodnotenie chýb SLAM algoritmov. HectorSlam však nedokázal spracovať tak veľké zašumenie dát z LiDARU a taktiež mal problémy pri mapovaní dynamickej prekážky, kde zanechal aj jej predošlé polohy. Avšak, v záverečnom porovnaní sú tieto algoritmy veľmi schopné a porovnateľné. Algoritmus GMapping sme zaradili, aby sme ho dokázali porovnať s novšími SLAM algoritmami. Pri teste na zaťaženie procesoru mal výborné výsledky, kde dosahoval hranicu 50%, avšak, v podkapitole 6.4, kde sme sa zameriavali na vytvorenie mapy po odstránení prekážky, sme videli jasné zlyhanie algoritmu, kedy nedokázal správne spojiť tvorenú mapu. Taktiež mu robil problém zachytiť dynamickú prekážku v podkapitole 6.5, kedy ju úplne odignoroval a zobrazoval to len ako šum, , náležitosť tohto správania závisí od aplikácie použitia. SlamToolbox je najnovší balíček z už vyššie spomínaných, ktorého vývoj stále prebieha. V danom balíčku vidíme potenciál v rozšíreniach, ktoré ponúka do nástroja Rviz, ktoré môžu byť užitočné pri vývoji aplikácie. Avšak, balíček pri našich testoch preukázal zlé výsledky. V prvom teste, kde sme overovali presnosť mapy, v podkapitole 6.1 mal chybu na pixel 0.252m. Taktiež nezvládol mapovanie pri teste v podkapitole 6.4 z čoho môžeme usúdiť, že algoritmu mu robí problém dynamicky sa meniace sa prostredie.

V našej práci sme dokázali implementovať štyri SLAM , ktoré sme preverili vo viacerých testoch. Zamerali sme ich na nepriaznivé situácie, v ktorých SLAM algoritmy zlyhávajú aby sme zistili ich negatívne a pozitívne vlastnosti.

Do budúcna vidíme posun v pridaní ďalších algoritmov, ktoré by sme vedeli implementovať z Tabuľka 1 Dostupné SLAM algoritmy a porovnať ich v rovnakých testoch, ako súčasne implementované algoritmy.

7 Zdroje

[1]Duchoň, František. Lokalizácia a navigácia mobilných robotov do vnútorného Nakladateľstvo STU v Bratislave, 2012. ISBN 978-80-227-3646-6. [Citované 15.1.2023]

[2]Peter Trebatický. Kalmanov filter Dostupné na internete: http://acm.vsb.cz/is/2004/finale/prispevky/trebaticky.pdf

[3]Autor neuvedený. Dokumentácia Hokuyo UTM-30LX .Dostupné na internete: http://wiki.ros.org/hokuyo_node?action=AttachFile&do=get&target=UTM-30LX_Specification.pdf [Citované 10.10.2022]

[4]Daniel Stonier, Younghun Ju, Jorge Santos Simon, Marcus Špecifikácia kobuki robota. Dostupné na internete : http://wiki.ros.org/kobuki. [Citované 10.10.2022]

[5]Autor neuvedený. Špecifikácia kobuki robota. Dostupné na internete :https://kobuki.readthedocs.io/en/devel/conversions.html [Citované 10.10.2022]

[6]Brian Gerkey, Tony Pratkanis, MapServer , Dostupné na internete : http://wiki.ros.org/map_server [Citované 18.12.2022]

[7] Ivan Dryanovski, William Morris, Andrea Censi, Opis balíčku laser scan matcher Dostupné na internete : http://wiki.ros.org/laser_scan_matcher [Citované 18.12.2022]

[8]Autor neuvedený. Obrázok systému ROS . [Citované 5.1.2023]Dostupné na internete: https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system%28ros%29/

[9]Tully Foote, Eitan Marder-Eppstein, Wim Meeussen. Špecifikáciebalíku tf . Dostupné na internete: http://wiki.ros.org/tf

[10] Autor neuvedený. SLAM algortimy. Dostupné na internete : https://openslam-org.github.io

[11] Google Cartographer . Dokumentácia k Google Cartographer [Citované 9.2.2023] Dostupné na internete: https://google-cartographer.readthedocs.io/en/latest/

[12] Marek Skalka. Porovnanie lokalizačných techník [Citované 18.12.2022] Dostupné na internete : http://marek.sk.sweb.cz/lokalizace/index.html.

[13] Autor neznámy. Inštalácia google cartographeru. Dostupné na internete :https://google-cartographer-ros.readthedocs.io/en/latest/compilation.html

[14] Brian Gerkey, Gmapping [Citované 20.2.2023] Dostupné na internete : http://wiki.ros.org/gmapping

[15] Autor neznámy. Informácie o kalmanovom filtre a EKF. [Citované 20.2.2023] Dostupné na internete :http://www.senzorika.leteckafakulta.sk/?q=node/269

[16] Stefan Kohlbrecher, Johannes Meyer. Špecifikácia HectorSlam. [Citované 20.1.2023] Dostupné na internete : http://wiki.ros.org/hector_slam

[17] Foote, Marder-Eppstein, Meeussen. Detaily o balíčku tf.[Citované 10.2.2023] Dostupné na internete : http://wiki.ros.org/tf

[18] Cyrill Stachniss. Informácie o FastSLAM metode. [Citované 10.9.2022] Dostupné na internete : http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam10-fastslam-4.pdf

[19] Autor neuvedený. Detaily o systéme ROS [Citované 5.1.2023] Dostupné na internete : https://www.ros.org/core-components/

[20] Autor neuvedený. Detaily o ROS msg [Citované 5.1.2023] Dostupné na internete : http://wiki.ros.org/msg

[21] Autor neuvedený. Detaily o ROS Topics [Citované 5.1.2023] Dostupné na internete : http://wiki.ros.org/Topics

[22] Autor neuvedený. Detaily o ROS Node [Citované 5.1.2023] Dostupné na internete : http://wiki.ros.org/Nodes

[23] Autor neuvedneý. Informácie o offline a online SLAM-e. .[Citované 10.2.2023] Dostupné na internete : https://www.exyn.com/news/differences-between-online-and-offline-slam

[24] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, Wolfram Burgard, Tutorial on Graph-Based [Citované 29.1.2023] Dostupné na internete : https://ieeexplore.ieee.org/abstract/document/5681215

[25] Awabot VisualSlam [Citované 29.1.2023] Dostupne na internete: https://awabot.com/en/visual-slam/

[26] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit FastSLAM Dostupné na internete: http://robots.stanford.edu/papers/montemerlo.fastslam-tr.pdf

[27] Sebastian Thrun1 , Michael Montemerlo1 , Daphne Koller1 , Ben Wegbreit1 Juan Nieto2 , and Eduardo Nebot2 FastSLAM Dostupné na internete: http://robots.stanford.edu/papers/Thrun03g.pdf

[28] Jefferies, Yeap. Informácie o SLAM-e Robotics and Cognitive Approaches to Spatial Mapping

8 Prílohy

Elektronická príloha :

1. ROS balíčky a súčasti
    a. Uzol pre spracovanie a publikovanie dát
    b. carto.launch
    c. gmapping.launch
    d. hector.launch
    e. mapping_hector.launch
    f. slamGmapping.launch
    g. slamToolbox.launch
    h. myConfiguration.lua (konfiguarcny subor pre GoogleCartographer)

Dokumentačná príloha :

8.1 Hlavný cyklus v nami vytvorenom uzle

```
if __name__ == '__main__':
    pocetVzoriek = 1080
    scann = LaserScan()
    rospy.init_node('dataPublisher')

    uhly,
vzdialenost,size,startcount,endcount,sampleTime_laser,arrSec_laser,arrNano_las
er=read_txt_laser()

    sampleTime_odom,LeftEncoder,RightEncoder,arrSec_odom,arrNano_odom=read_txt
_odom()

    x,y,z,angle,velX,velY,velA=oc.calculateOdom(LeftEncoder,RightEncoder,sampl
eTime_odom)

    #vzdialenost= gs.genSumu(vzdialenost,30)

    #x,y=gs.genSumuOdom(x,y,0.1)

    fullArray,indexArr=TS.createSyncArr(sampleTime_odom,sampleTime_laser)
    r = rospy.Rate(4) # 1hz
    o=0
    for i in range(len(indexArr)-1):
        current_time=rospy.Time.now()

        if indexArr[i]==0:
            publishOdomMy(x[o],y[o],z[o],angle[o],velX[o],velY[o],velA[o],curr
ent_time)
            o+=1
            print("odom "+str(i)+" z "+str(len(indexArr))+"
uhol"+str(angle[o]))
        else:
            laserPublisher(uhly,
vzdialenost,size,startcount,endcount,current_time)
            startcount=pocetVzoriek+startcount
            endcount=pocetVzoriek+endcount
            print("laser "+str(i)+" z "+str(len(indexArr))+"
uhol"+str(angle[o]))
        time.sleep(0.1)
```

8.2 Výpočet odometrie

```
import math

x=[]
y=[]
z=[]
posX=0
posY=0
posAngle=0
Angle = []
angleLast =0
encoderRightLast=0
encoderLeftLast =0
SampletimeLast =0

tickToMeter = 0.00008529209049773755655 8; # [m/tick]
b = 0.23; # wheelbase distance in meters, from kobuki manual
https://yujinrobot.github.io/kobuki/doxygen/enAppendixProtocolSpecificati
on.html

wheelBase = b

def initializePos(EncoderRightNew,EncoderLeftNew,SampletimeNew):
    global encoderRightLast,encoderLeftLast,SampletimeLast
    encoderRightLast =EncoderRightNew
    encoderLeftLast = EncoderLeftNew
    SampletimeLast = SampletimeNew


def overflowTest(encoder,encoderLast):
    if(encoder-encoderLast <= 65536/2 and encoder-encoderLast >= -
65536/2):
        return encoder-encoderLast

    elif(encoder-encoderLast < -65536/2):
        return 65535-encoderLast+encoder

    elif(encoder-encoderLast > 65536/2):
        return encoder-encoderLast-65535

    else:
        return 0.0

def calculatePos(EncoderRight, EncoderLeft):
    global encoderRightLast,encoderLeftLast,posAngle,angleLast,posX,posY

    wheelDisRight = tickToMeter * (overflowTest(EncoderRight,
encoderRightLast))
```

```
    wheelDisLeft = tickToMeter * (overflowTest(EncoderLeft,
encoderLeftLast))
    distance = (wheelDisRight + wheelDisLeft) / 2
    posAngle = posAngle + ((wheelDisRight-wheelDisLeft) / wheelBase)
    if(posAngle*180/math.pi >= 180.5):
        posAngle = posAngle-360.5/180*math.pi
    if(posAngle*180/math.pi <= -180.5):
        posAngle = posAngle+360.5/180*math.pi

    if(wheelDisRight-wheelDisLeft != 0):
        posX = posX +
wheelBase*(wheelDisRight+wheelDisLeft)/(2*(wheelDisRight-
wheelDisLeft))*(math.sin(posAngle)-math.sin(angleLast))
        posY = posY -
wheelBase*(wheelDisRight+wheelDisLeft)/(2*(wheelDisRight-
wheelDisLeft))*(math.cos(posAngle)-math.cos(angleLast))
    else:
        posX = posX + distance*math.cos(posAngle)
        posY = posY + distance*math.sin(posAngle)

    encoderRightLast = EncoderRight
    encoderLeftLast = EncoderLeft
    angleLast = posAngle

    return posX,posY,posAngle


def velCalculate(pos,posOld,time,timeOld):

      time = time /1000000
      timeOld = timeOld /1000000

      velocity = (posOld-pos)/(timeOld-time)

      return velocity


def CalculateVelocity(posX,posY,Angle,sampleTime):

    velX=[]
    velY=[]
    velA=[]

    for i in range(len(posX)):
        if i ==1:
            velX.append(0)
            velY.append(0)
            velA.append(0)
        else:
```

```
            velX.append(velCalculate(posX[i],posX[i-
1],sampleTime[i],sampleTime[i-1]))
            velY.append(velCalculate(posY[i],posY[i-
1],sampleTime[i],sampleTime[i-1]))
            velA.append(velCalculate(Angle[i],Angle[i-
1],sampleTime[i],sampleTime[i-1]))




    return velX,velY,velA

def calculateOdom(arrLeftEncoder,arrRightEncoder,arrsampleTime):
    i=0

    initializePos(arrRightEncoder[0],arrLeftEncoder[0],arrsampleTime[0])

    #print(arrLeftEncoder)


    for i in range(len(arrLeftEncoder)):
        tempx,tempy,tempA
=calculatePos(arrRightEncoder[i],arrLeftEncoder[i])

        x.append(tempx)
        y.append(tempy)
        z.append(0)
        Angle.append(tempA)

    velX,velY,velA=CalculateVelocity(x,y,Angle,arrsampleTime)
    print(len(velX), len(velY), len(velA))
    #print(x)
    #print(y)
    #print(Angle)

    return x,y,z,Angle,velX,velY,velA
```

8.3 Tvorba správy pre dáta z lasera

```
def laserPublisher(uhly,
vzdialenost,size,startcount,endcount,current_time):
    global seq
    global pocetVzoriek
    pub = rospy.Publisher('scan', LaserScan, queue_size=10)

    scann.header.stamp = current_time
    scann.header.frame_id = 'laser'

    seq=seq+1
```

```
    scann.angle_min = -2.3561944902
    scann.angle_max = 2.3561944902
    scann.angle_increment = 0.00436332313
    scann.time_increment = 0.00025

    scann.range_min = 0.0001
    scann.range_max = 30.0

    scann.ranges = []
    scann.intensities = []


    for j in range(startcount, endcount):
        scann.ranges.append((vzdialenost[j]/1000))
        if j == endcount-1 and pocetVzoriek == 1080:
            scann.ranges.append((vzdialenost[j]/1000))
        if j==size-1:
            i=0
            rospy.loginfo("Idem znova")
            startcount=0
            endcount=pocetVzoriek
            break

    print(len(scann.ranges))
    pub.publish(scann)
```

8.4 Tvorba správy pre dáta z odometrie

```
def publishOdomMy(x,y,z,angle,velX,velY,velA,current_time):
    odom_pub = rospy.Publisher("odom", Odometry, queue_size=50)
    odom_broadcaster = tf.TransformBroadcaster()



    # since all odometry is 6DOF we'll need a quaternion created from yaw
    odom_quat = tf.transformations.quaternion_from_euler(0, 0, angle)

    # first, we'll publish the transform over tf
    odom_broadcaster.sendTransform(
        (x, y, 0.),
        odom_quat,
        current_time,
        "base_link",
        "odom"
    )

    # next, we'll publish the odometry message over ROS
    odom = Odometry()
```

```
    odom.header.stamp = current_time
    odom.header.frame_id = "odom"

    # set the position
    odom.pose.pose = Pose(Point(x, y, 0.), Quaternion(*odom_quat))

    # set the velocity
    odom.child_frame_id = "base_link"
    odom.twist.twist = Twist(Vector3(velX, velY, 0), Vector3(0, 0, velA))

    # publish the message
    odom_pub.publish(odom)

    #print(odom)

    last_time = current_time
```

8.5 Spúšťací súbor pre Google Cartographer

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

  <node name="cartographer_node" pkg="cartographer_ros"
      type="cartographer_node" args="
          -configuration_directory
              $(find cartographer_ros)/configuration_files
          -configuration_basename myConfig.lua"
      output="screen">
    <remap from="scan" to="/base_scan" />
  </node>

  <node name="cartographer_occupancy_grid_node" pkg="cartographer_ros"
      type="cartographer_occupancy_grid_node" args="-resolution 0.05" />

    <node name="rviz" pkg="rviz" type="rviz" required="true"
      args="-d $(find cartographer_ros)/configuration_files/demo_2d.rviz"
/>

    <node name="tf_map_to_odom" pkg="publish_data"
type="tf_map_to_odom.py" >
    </node>

    <node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser" args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /scan
10">
    </node>

    <node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
name="laser_scan_matcher_node" output="screen">
```

```
        <param name="fixed_frame" value = "odom"/>
        <param name="base_frame" value = "base_link"/>
        <param name="use_odom" value="true"/>
        <param name="publish_odom" value = "true"/>
        <param name="use_alpha_beta" value="true"/>
        <param name="max_iterations" value="10"/>
    </node>

    <!--node name="dataPublisher" pkg="publish_data"
type="dataPublisher.py" output="screen"/-->

</launch>
```

8.6 Spúštací súbor pre GMapping

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

<launch>
    <node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser" args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /laser
10">
    </node>
    <!--<node pkg="tf" type="static_transform_publisher"
name="odom_to_base_link" args="0.0 0.0 0.0 0.0 0.0 0.0 /odom /base_link
10">
    </node>
    <node pkg="tf" type="static_transform_publisher" name="map_to_odom"
args="0.0 0.0 0.0 0.0 0.0 0.0 /map /odom 10 ">
    </node>-->
    <node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
name="laser_scan_matcher_node" output="screen">
        <param name="fixed_frame" value = "odom"/>
        <param name="use_odom" value="true"/>
        <param name="publish_odom" value = "true"/>
        <param name="use_alpha_beta" value="true"/>
        <param name="max_iterations" value="10"/>
    </node>
    <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
        <param name="map_udpate_interval" value="1.0"/>
        <param name="xmin" value="-0.5"/>
        <param name="ymin" value="-0.5"/>
        <param name="xmax" value="0.5"/>
        <param name="ymax" value="0.5"/>
        <param name="delta" value="0.05"/>
    </node>
```

```
     <!-- launch rviz with my cnofiguration  -->
  <node type="rviz" name="rviz" pkg="rviz" args="-d $(find
publish_data)/rviz/rviz_config.rviz" />
  <node name="dataPublisher" pkg="publish_data" type="dataPublisher.py"
output="screen"/>
</launch>
```

8.7 Spúštací súbor pre HectorSlam

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

<launch>

    <node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser" args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /laser
10">
    </node>
    <!--<node pkg="tf" type="static_transform_publisher"
name="odom_to_base_link" args="0.0 0.0 0.0 0.0 0.0 0.0 /odom /base_link
10">
    </node>
    <node pkg="tf" type="static_transform_publisher" name="map_to_odom"
args="0.0 0.0 0.0 0.0 0.0 0.0 /map /odom 10 ">
    </node>-->
    <node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
name="laser_scan_matcher_node" output="screen">
        <param name="fixed_frame" value = "odom"/>
        <param name="use_odom" value="true"/>
        <param name="publish_odom" value = "true"/>
        <param name="use_alpha_beta" value="true"/>
        <param name="max_iterations" value="10"/>
    </node>

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



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

    <include file="$(find publish_data)/launch/mapping_hector.launch"/>

    <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>


    <node name="dataPublisher" pkg="publish_data" type="dataPublisher.py"
output="screen"/>

</launch>
```

8.8 Spúšťací súbor pre SlamToolbox

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

<launch>

    <node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser" args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /laser
10">
    </node>

    <node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
name="laser_scan_matcher_node" output="screen">
        <param name="fixed_frame" value = "odom"/>
        <param name="use_odom" value="true"/>
        <param name="publish_odom" value = "true"/>
        <param name="use_alpha_beta" value="true"/>
        <param name="max_iterations" value="10"/>
    </node>

    <node pkg="slam_toolbox" type="async_slam_toolbox_node"
name="slam_toolbox" output="screen">
        <rosparam command="load" file="$(find
slam_toolbox)/config/mapper_params_online_async.yaml" />
        <param name="odom_frame" value = "odom"/>
        <param name="base_frame" value = "base_link"/>
        <param name="resolution" value="0.05"/>
    </node>

     <!-- launch rviz with my cnofiguration  -->
  <node type="rviz" name="rviz" pkg="rviz" args="-d $(find
publish_data)/rviz/rviz_config.rviz" />

  <node name="dataPublisher" pkg="publish_data" type="dataPublisher.py"
output="screen"/>

</launch>
```