SLAM na TurtleBot3 platformi

Uranjek, Filip

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, School of Applied Mathematics and Informatics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet primijenjene matematike i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:772968

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-09-27**

Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika

SLAM na TurtleBot3 platformi

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Komentor:

Jurica Maltar

Kandidat:

Filip Uranjek

Osijek, 2024

Sadržaj

1 | Uvod

1.1 Uvod u rad i motivacija

Autonomni roboti prisutniji su sve više i više među nama. Mogu se pronaći u logistici, zdravstvu, poljoprivredi i kućanstvu. Kako vrijeme prolazi robotima se zadaju sve složeniji zadaci poput izbjegavanja prepreka, stvaranja mapa i navigacija u nepoznatom prostoru. Ti zadaci zahtijevaju sustave koji omogućuju preciznu kontrolu i percepciju okoline.

Jedan od ključnih alata u pronalasku rješenja za navedene probleme je Robot Operating System (ROS), fleksibilna softverska platforma, čija je svrha razvoj i upravljanje robotskim aplikacijama. Bogata biblioteka i velik broj gotovih alata omogućuje jednostavan razvoj, korištenje i integriranje različitih komponenti. Zbog svoje modularnosti i otvorenog koda, ROS je postao standard u istraživanju i industriji.

U okviru ovog rada promatrat će se korištenje ROS-a sa TurtleBot platformom, konkretno TurtleBot3 robotom. TurtleBot dizajniran je za testiranje i razvoj algoritama za robotiku, a uz to on nudi vrlo lagan i jednostavan uvod u svijet robotike i algoritama koji se koriste. Jedan od tih algoritama je istovremena lokalizacija i mapiranje (eng. simultaneous localization and mapping, abbr. SLAM) koji omogućava robotu da mapira prostor oko sebe i određuje svoju poziciju unutar tog prostora korištenjem raznih senzora i algoritama.

Motivacija za pisanje ovog rada dolazi iz želje za učenjem robotike i algoritama koji se koriste. Korištenjem stvarnog robota dobiva se vizualna povratna informacija o trenutnom radu koja dodatno potiče interes i želju za učenjem.

Rad je podijeljen u četiri cjeline.
U prvoj cjelini uvodimo Robot Operating System, ili ROS kako ćemo navoditi u nastavku rada. Razmatramo arhitekturu i osnovne komponente sustava, njegovu primjenu i njegove glavne prednosti.
U drugoj cjelini predstavljamo TurtleBot platformu. Prikazujemo specifikacije i mogućnosti TurtleBot3 robota i njegovu integraciju sa ROS-om.
Treća cjelina definira simultaneous localization and mapping (SLAM) problem koja je praktički baza za autonomnu kretnju robota. Promatramo rješenja, ali isto tako i izazove pri rješavanju i realizaciji SLAM-a.
Na kraju, u četvrtoj cjelini opisujemo postupak postavljanja i integracije ROS-a sa TurtleBot-om. Uz to promatramo kako SLAM funkcionira sa robotom koristeći već gotove module, ali i vlastito izrađene module.

2 | Robot Operating System

2.1 Uvod u ROS

Iz imena Robot Operating System dalo bi se zaključiti da je ROS operativni sustav za robote, kao što su Windows ili Linux operativni sustavi, no to nije u potpunosti točno. ROS je zapravo Meta-Operating System. Meta-Operating System možemo definirati kao apstraktni sloj između aplikacije i računalnog sklopovlja. To znači da ROS mora biti instaliran na neki operativni sustav, najpoželjnije na neku Ubuntu distribuciju Linuxa.

Slika 2.1: Vizualni prikaz meta-operativnog sustava.

ROS nudi biblioteke koje podržavaju različite programske jezike, komunikacijske sustave za protok podataka, hardversko sučelje za upravljanje hardver djelom, Framework za robotske aplikacije, gotove robotske aplikacije s mogućnosti nadogradnje, alate za simulaciju robota u virtualnom prostoru i alate za razvoj softvera. Dodatno glavne karakteristike koje se vežu uz ROS su:

1. **Jednostavnost.** Programiran je u obliku neovisnih manjih jedinica, tzv. čvorova(eng. nodes), koji sustavno razmjenjuju podatke.
2. **Modularnost.** Procesi i jedinice koje imaju sličnu svrhu kombiniraju se u pakete, čime se olakšava korištenje i daljnji razvoj.
3. **Otvoreni kod.** Svi paketi su javno dostupni u njihovom izvornom kodu.
4. **API.** Prilikom razvoja programa koji koristi ROS, jednostavno se poziva API i lako integrira u postojeći kod.

5. **Mogućnost korištenja raznih programskih jezika.** ROS nudi razne biblioteke koje podržavaju razne programske jezike poput popularnih jezika u robotici Python i C++ i malo manje popularnih kao Lua i Ruby.

Sve navedeno je korisniku dano u svrhu razvoja novih robotskih aplikacija, ali isto tako i daljnjem unaprjeđenju šire robotske zajednice. Korištenjem ROS-a, programeri mogu dijeliti, modificirati i ponovno koristiti kod, čime se potiče suradnja i napredak, a samim time i stvaranje novih ideja i rješenja na području robotike.

2.2 ROS verzije

Prva je verzija ROS-a izašla 2007. godine, a koristi se i danas. Postoji i nova verzija nazvana ROS2 koja je izašla 2017. godine i koja je povećala skalabilnost projekta i podršku za naprednije tehnologije. Uz to, pojednostavila je arhitekturu i rukovanje sa ROS-om i osigurala je bolju komunikaciju u stvarnom vremenu i veću sigurnost. ROS1 i ROS2 trenutno koegzistiraju, pri čemu se ROS2 više koristi u novim projektima zbog naprednijih mogućnosti i lakšeg rukovanja. To ne znači da je ROS1 izgubio svoju svrhu, jer neki projekti i dalje rade sa starijom verzijom zbog bolje dokumentiranosti i više materijala koji robotska zajednica nudi. ROS2 je ipak novija verzija napravljena sa svrhom da zamjeni ROS1 u budućnosti i zato je i u ovom radu korišten ROS2.

Dodatno, ROS1 i ROS2 imaju i svoje distribucije. ROS distribucija je skup ROS paketa, slično kao kod Linux distribucije (npr. Ubuntu 20.04 i Ubuntu 22. 04). Svrha ROS distribucije je omogućiti programerima rad na stabilnoj bazi dok ne budu spremni za nadogradnju na noviju verziju. Odabir ROS distribucije ovisi o operativnom sustavu koji se koristi. Najpopularniji operativni sustav koji se koristi za ROS je Ubuntu distribucija Linuxa. Naime, svaka ROS distribucija odgovara jednoj Ubuntu distribuciji. Tako ROS2 "Foxy Fitzroy" odgovara Ubuntu 20.04 distribuciji, dok ROS2 "Humble Hawksbill" odgovara Ubuntu 22.04 distribuciji. Svaka ROS distribucija ima odgovarajuću kornjaču kao prepoznatljiv simbol ROS-a, a dodatno ima i "End-of-life" datum, gdje neke distribucije imaju dugoročnu podršku. U ovom radu korišten je ROS2 "Humble Hawksbill" na Ubuntu 22.04 ("Jammy Jellyfish") budući da Humble distribucija ima dugoročnu podršku.

Distro	Release date	Logo	EOL date
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018
beta3	September 13th, 2017		December 2017
beta2	July 5th, 2017		September 2017
beta1	December 19th, 2016		Jul 2017
alpha1 - alpha8	August 31th, 2015		December 2016

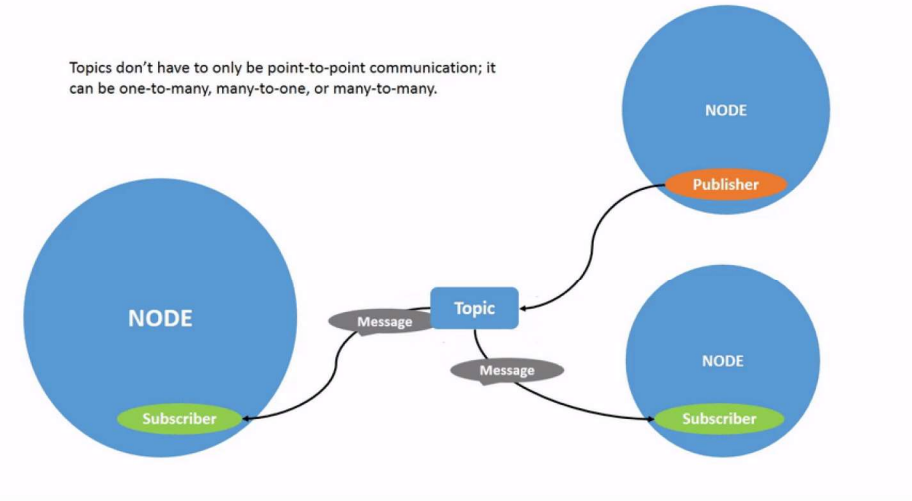
Slika 2.2: ROS2 distribucije. Slika preuzeta sa [7]

2.3 ROS komponente i koncepti

U središtu svakog ROS2 sustava nalazi se ROS graf koji označava mrežu čvorova unutar sustava i veze između njih koje predstavljaju njihovu komunikaciju. Komunikacija se bazira na konceptu izdavača (eng. publisher) i pretplatnika (eng. subscriber), gdje pojedine jedinice objavljuju poruke koje drugi pretplatnici onda osluškuju.

2.3.1 Čvorovi

Čvor (eng. node) je temeljna jedinica ROS procesa koja obavlja jedan ili više zadataka. Čvor bi trebao imati samo jednu svrhu koju bi trebao obavljati. Time se omogućuje jednostavna modularnost i mogućnost ponovne upotrebe. Čvor može imati više funkcionalnosti. Može biti izdavač koji objavljuje informacije, odnosno tzv. teme (eng. topics) drugim čvorovima. Može biti pretplatnik koji prima poslane podatke od drugog čvora. Također može biti klijent neke usluge (eng. service) od koje dobiva odgovarajuće podatke ili i on sam može bit poslužitelj usluge koji će onda pružati funkcionalnosti drugim čvorovima. Kada pogledamo jedan kompletan ROS projekt možemo vidjeti složeni graf objavljivač , pretplatnika, poslužitelja usluga i klijenta usluga sve u isto vrijeme.



Slika 2.3: Komunikacija između čvorova. Slika preuzeta sa [7]

2.3.2 Paketi

Paket (eng. package) je skup čvorova slične funkcionalnosti, dodatnih ROS biblioteka i konfiguracijskih datoteka čija je svrha pravilno izvođenje određenog zadatka. Primjer takvog paketa je *turtlesim* koji uz ostalo ima dva čvora, *turtlesim* čvor koji je zadužen za pravilno funkcioniranje kornjače u simulaciji i *turtle teleop key* čvor koji omogućuje kretanje kornjače uz tipkovnicu.

2.3.3 Teme

Izdavači i pretplatnici koriste teme kako bi međusobno uspostavili komunikaciju. Kada kreiramo izdavača i pretplatnika dodjeljujemo naziv teme. Tema u sebi sadrži informaciju o tipu podatka koji se prenosi. Ona može biti jednostavna poput običnog broja, no može biti i kompleksnija poput objekta koji sadrži različite tipove podataka. U radu se koristi light detection and ranging (abbr. LIDAR) senzor i tema koja se objavljuje je u suštini objekt koji se sastoji od dva 3-dimenzionalna vektora.

2.3.4 Usluge

Usluga (eng. service) je sinkrona dvosmjerna komunikacija između dva čvora. Čvor koji zahtjeva uslugu naziva se klijent usluge, a čvor koji pruža uslugu naziva se poslužitelj usluge. Usluga funkcionira na način da klijent zatraži uslugu od poslužitelja i prima odgovor od njega, dok poslužitelj prima zahtjev, provodi odgovarajući postupak i šalje odgovor klijentu. Usluge su najčešće izračuni, a komunikacija bi između čvorova tokom usluge trebala biti što kraća, jer klijent usluge čeka povratnu informaciju poslužitelja usluge.

2.3.5 Akcije

Akcija (eng. action) je asinkrona dvosmjerna komunikacija između dva čvora. Isto kao i kod servisa, čvor koji zahtjeva proceduru naziva se klijent akcije, a čvor koji pruža proceduru naziva se poslužitelj akcije. Akcija se razlikuje od usluge u vremenu izvršavanja. Naime, akcija je dugotrajna procedura gdje se u međuvremenu šalju povratne informacije o napretku klijentu i isto tako ima i mogućnost otkazivanja.

2.3.6 Poruke

Teme, usluge i akcije koriste različite poruke pri komunikaciji. Poruke (eng. messages) mogu biti jednostavne poput cijelih brojeva, decimala, boolean vrijednosti, listi, ali češće se pojavljuju složenije strukture poruka. Jedan primjer složene poruke je poruka nazvana *Twist*. Kontroliranjem robota uz pomoć tipkovnice možemo uočiti da čvor koji je dužan za mapiranje tipki kretnje zapravo objavljuje temu čija je poruka obilka *Twist*. *Twist* poruke su zapravo dva trodimenzionalna vektora, gdje jedan vektor predstavlja linearnu brzinu, a drugi angularnu brzinu kojom želimo se robot kreće.

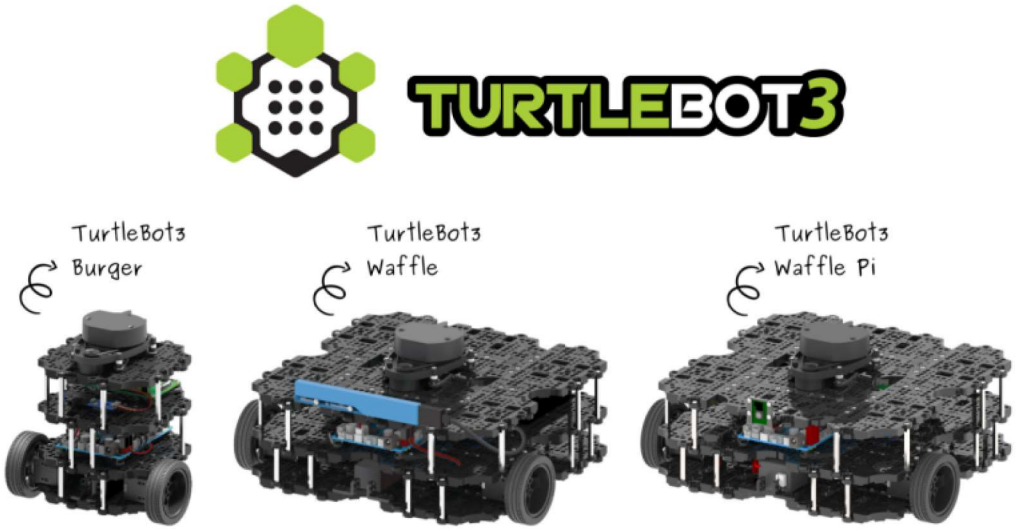
2.3.7 Launch

ROS2 sustav često se sastoji od velikog broja čvorova koji obavljaju različite zadatke. Kako bi se olakšalo pokretanje cijelog sustava pišu se tzv. launch datoteke koje automatiziraju cijeli proces. Launch datoteke mogu biti pisane u Pythonu, XML-u ili YAML-u i u njima se navodi koji čvorovi trebaju biti pokrenuti i sa kojim argumentima. Također su odgovorni za praćenje stanja procesa i za izvještavanje i reagiranje na promjene. Time se cijeli postupak pokretanja svodi na pokretanje jedne launch datoteke.

3 | TurtleBot3 platforma

3.1 Uvod u TurtleBot3

Turtlebot3 je pristupačni programabilni robot baziran na ROS-u, namijenjen za upotrebu u istraživanju, obrazovanju i prototipiranju proizvoda. Cilj TurtleBot-a je učiniti svijet robotike pristupačnim i jednostavnim za ulazak svima koji žele učiti o robotici. Glavne funkcionalnosti robota su pokretanje SLAM-a, navigacija u prostoru i manipulacija objektima. Samo neki od primjera korištenja robota su autonomno kretanje, mapiranje prostora, manipulacija objektima, kretanje po liniji. Zbog velike prilagodljivosti TurtleBot-a kao što su rekonstrukcija mehaničkih dijelova i dodavanje opcionalnih senzora, njegove mogućnosti se značajno povećavaju.
Postoje 3 verzije TurtleBot3 robota, TurtleBot3 Burger, TurtleBot3 Waffle i TurtleBot3 Waffle Pi.
U ovom radu je korišten TurtleBot3 burger, pa će se njega opširnije opisati u sljedećem poglavlju.



Slika 3.1: TurtleBot3 verzije. Slika preuzeta sa [8]

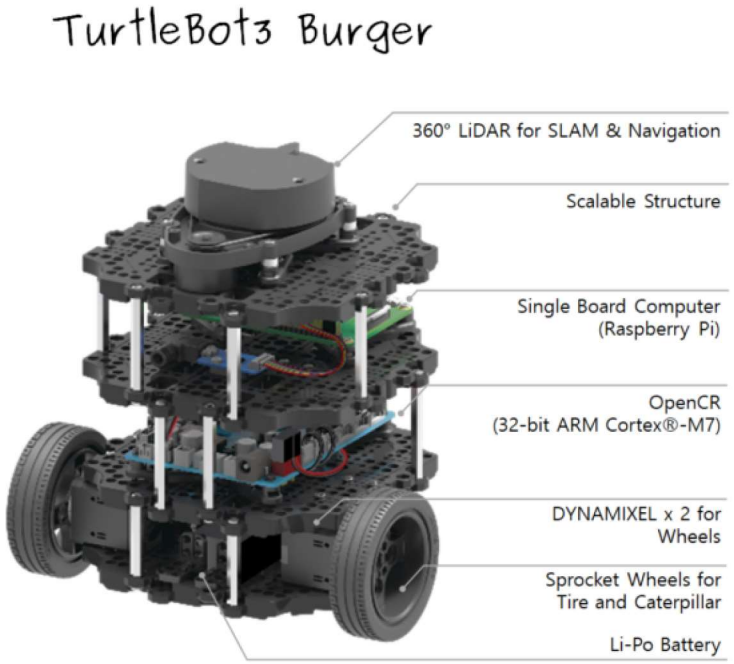
3.2 TurtleBot3 Burger

TurtleBot3 Burger je najmanji i najpristupačniji model iz serije TurtleBot3 robota, idealan za ulazak u svijet robotike.

U središtu robota nalazi se Raspberry Pi 3, koji služi kao glavno računalo i omogućuje izvršavanje algoritama, obradu senzorskih podataka, spajanje na internet i komunikaciju robota sa računalom.
OpenCR ploča integrira sve hardverske komponente robota, upravlja motornim pogonima i napajanjem i osigurava komunikaciju između senzora i računala.

Na vrhu robota se nalazi LIDAR senzor koji omogućuje mapiranje prostora i detekciju prepreka u stvarnom vremenu. LIDAR emitira pulsne svjetlosne valove iz lasera u okolinu. Ti pulsevi odbijaju se od okolinu i vraćaju se natrag do senzora. Senzor u međuvremenu mjeri vrijeme potrebno da se svaki puls vrati do senzora i iz tog izračunava udaljenost koju je prešao. Tim procesom računalo može dobiti dojam o prostoru u kojem se nalazi i po želji može prikazati taj prostor.

IMU (Inertial Measurement Unit) pruža informacije o orijentaciji i ubrzanju robota, što pomaže u održavanju stabilnosti tijekom kretanja. IMU koji se nalazi na robotu ima akcelerometar iz kojeg dobivamo podatke o ubrzavanju i ima žiroskop iz kojeg dobijemo podatke o kutnoj brzini. Kombinacija ovih podataka omogućuje precizan nadzor svog kretanja i stabilnost tijekom rada.
Robot se kreće pomoću dva kotača koja su povezana s DYNAMIXEL servo motorima koji omogućuju pravilno i precizno kretanje robota.



Slika 3.2: Komponente TurtleBot3 Burger-a . Slika preuzeta sa [8]

3.3 TurtleBot3 i ROS

3.3.1 Postavljanje ROS-a i TurtleBot-a

Prije nego što možemo koristiti ROS na TurtleBot-u, prvo moramo postaviti i konfigurirati naše okruženje.

Prvo i osnovno, računalo koje koristimo trebalo bi imati instaliran Ubuntu operativni sustav koji je najpoželjniji za rad sa ROS-om. Nakon toga, odgovarajuća ROS distribucija trebala bi biti instalirana. Instalacija se sastoji od postavljanja izvora, instaliranja odgovarajućih paketa i učitavanja setup skripte.

Nakon toga prelazimo na postavljanje Raspberry Pi-a na našem robotu. Prvo što trebamo imati je microSD karticu sa odgovarajućim operativnim sustavom, u kontekstu ovog rada Raspberry Pi ima instaliran Ubuntu Server. Idući je korak konfiguracija Raspberry Pi-a. To uključuje spajanje na internet, instalaciju ROS-a i izgradnju ROS paketa.

Zadnji korak je konfiguriranje OpenCR ploče. Na naš Raspberry preuzimamo odgovarajući OpenCR softver i prenosimo ga na OpenCR ploču.

Svi ovi koraci se detaljno mogu pronaći na [8].

3.3.2 Korištenje ROS-a i TurtleBot-a

Nakon pravilne instalacije robot je spreman za korištenje.
Na robotu moramo pokrenut launch datoteku *turtlebot3-bringup robot.launch.py* koja pokreće odgovarajuće čvorove kako bi robot pravilno funkcionirao. Neki od tih čvorova inicijaliziraju upravljanje servo motorm, neki komunikaciju s IMU-om, neki komuniciraju s računalom, itd. Nakon toga na računalu možemo pokretati gotove primjere i pakete za funkcionalnosti robota. Neki od paketa su SLAM paket koji mapira prostor i iscrtava mapu na računalu, Navigation paket koji zadaje robotu putanju u zadanom prostoru. Isto tako, postoji i Gazeboo simulator koji simulira robota i njegove kretnje, što je vrlo korisno ukoliko nam robot nije pri ruci. Također imamo pakete za kretnje robota, gdje robota možemo pokretati tipkovnicom ili kontrolerom poput PS3 kontrolera.

Uz sve navedene funkcionalnosti, korisniku su dani i alati za izgradnju vlastitih aplikacija koji omogućuju prilagodbu robota prema specifičnim potrebama.

4 | Simultaneous Localization and Mapping(SLAM)

4.1 Uvod u SLAM

SLAM je problem konstruiranja ili ažuriranja mape nepoznatog terena i istovremena procjena poze robota unutar istog terena. Poza robota se odnosi na njegov položaj i orijentaciju u prostoru. Pozu matematički možemo definirati kao kombinaciju translacije (pomicanje robota u prostoru) i rotacije (smjera u kojem je robot usmjeren). Poza robota može se predstaviti na nekoliko načina:

- Rotacijska matrica i vektor pozicije: rotacijska matrica opisuje orijentaciju robota, a vektor pozicije opisuje njegov položaj.
- Transformacijska matrica: kombinira rotaciju i translaciju u jednu matricu koja opisuje pozu robota.
- Kvaternion i vektor pozicije: kvaternion predstavlja rotaciju, a koristi se zbog numeričke stabilnosti i učinkovitosti, dok vektor pozicije specifira translaciju.

SLAM je aktualna tema u svijetu robotike koja se konstantno pokušava unaprijediti i optimizirati raznim istraživačkim metodama.

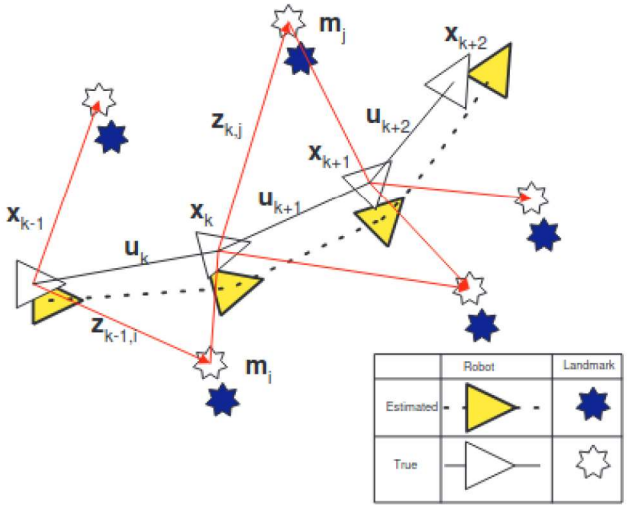
Enkoderi i IMU jedinice služe za odometriju, odnosno procjenu položaja robota koristeći podatke od odgovarajućih senzora. U našem slučaju enkoder izračunava približni položaj robota prateći rotacije kotača. Kako bi procjena položaja bila što točnija, podaci sa IMU-a se koriste za dodatnu korekciju i poboljšanje preciznosti ove procjene.
Unatoč svemu tome, proces odometrije je i dalje podložan pogreškama i smetnjama, pa se koriste podaci dobiveni od senzora udaljenosti ili kamere koji služe za izradu karte. Popularne metode za procjenu poze su Kalmanov filter, Particle filter, Markovljeva lokalizacija, Graph SLAM i slično.

Za konstrukciju karte koriste se senzori udaljenosti kao što su infracrveni, laserski i ultrazvučni senzori. Osim senzora udaljenosti često se koristi i mono ili stereo kamera, a SLAM koji koristi takve podatke naziva se još i vizualni SLAM. Naš robot koristi LIDAR senzor koji je zapravo detektor svijetla.

4.2 Formalni zapis SLAM problema

Promotrimo robota koji se kreće i koristi senzor smješten na robotu za uzimanje relativnih promatranja nekoliko orijentira (eng. landmarks), kao što je prikazanon na slici 4.1. U trenutku k definirane su sljedeće veličine:

- x_k: Vektor koji opisuje lokaciju i orijentaciju, odn. pozu vozila
- u_k: Vektor upravljanja koji dovodi stanje x_{k-1} u x_k
- m_i: Vektor koji opisuje lokaciju i-tog orijentira čija se stvarna lokacija pretpostavlja nepromjenjivom tijekom vremena.
- z_{ik}: Promatranje uzeto s robota koje pokazuje i-ti orijentir u trenutku k.



Slika 4.1: Vizualni prikaz SLAM-a. Slika preuzeta sa [1]

Dodatno, definirani su i sljedeći skupovi:

- $X_{0:k} = \{x_0, x_1, \cdots, x_k\} = \{X_{0:k-1}, x_k\}$: Povijest lokacija vozila.
- $U_{0:k} = \{u_1, u_2, \cdots, u_k\} = \{U_{0:k-1}, u_k\}$: Povijest upravljačkih ulaza.
- $m = \{m_1, m_2, \cdots, m_n\}$: Skup svih orijentira.
- $Z_{0:k} = \{z_1, z_2, \cdots, z_k\} = \{Z_{0:k-1}, z_k\}$: Skup svih promatranja orijentira.

Postoje razne vrste SLAM-a, no u ovom radu ćemo promotriti vjerojatnosni SLAM. On zahtjeva izračun distribucije vjerojatnosti

$$P(X_k, m | Z_{0;k}, U_{0;k}, X_0)$$

za sva vremena k. Ova distribucija opisuje zajedničku posteriornu gustoću stanja orijentira i vozila u trenutku k, s obzirom na sva opažanja i upravljanja do tog

trenutka, zajedno s početnim stanjem vozila. Zajednička posteriorna gustoća zapravo predstavlja vjerojatnost da su određene lokacije orijentira točno tamo gdje se pretpostavlja da se nalaze, s obzirom na sva opažanja prikupljena do tog trenutka. Drugim rječima, SLAM algoritam pokušava procijeniti gdje se robot i orijentiri nalaze, s obzirom na sve informacije koje je prikupio do tada. Ta procjena se ažurira svaki puta kada robot dobije nove informacije.

Kod rješenja SLAM problema koristi se rekurzivni pristup. Počevši s procjenom distribucije

$$P(X_k - 1, m | Z_{0;k-1}, U_{0;k-1})$$

u trenutku k - 1, zajednička posteriorna distribucija nakon upravljanja i opažanja računa se pomoću Bayesova teorema. Računanje zahtjeva definiranje modela prijelaza stanja i modela opažanja.

Model opažanja opisuje vjerojatnost $P(z_k|x_k, m)$ da će robot napravit opažanje z_k kada su poznate lokacije vozila i orijentiri. Model gibanja vozila opisuje se pomoću distribucije vjerojatnosti $P(X_k|x_k - 1, u_k)$ koja predpostavlja da sljedeće stanje x_k ovisi samo o prethodnom stanju $x_k - 1$ i upravljanju u_k, to se još naziva i Markovljevo svojstvo.

SLAM algoritam implementiran je u rekurzivnom obliku koji se sastoji od predikcije (eng. time-update), što je zapravo evaluacija modela gibanja i korekcije (eng. measurement-update), što predstavlja evaluaciju modela mjerenja. Ovom procedurom izračunavamo zajedničku posteriornu distribuciju $P(X_k, m|Z_{0;k}, U_{0;k}, X_0)$ za stanje robota x_k i mapu m u trenutku k na temelju dosadašnjih opažanja i upravljanja.

4.3 Rješenja SLAM problema

Kao što je već rečeno postoji puno vrsta rješenja, no ovdje ćemo se fokusirati na dvije najpopularnije metode: Prošireni Kalmanov filter i čestični filter.

Rješenje za vjerojatnosni SLAM uključuje pronalazak odgovarajućeg modela opažanja i modela gibanja. Najčešća reprezentacija je u obliku modela stanja s dodatnim Gaussovim šumom što rezultira korištenjem proširenog Kalmanovog filtra. Alternativa je opisivanje modela gibanja kao skup uzoraka ne-Gaussove distribucije vjerojatnosti, što dovodi do korištenja čestičnog filtra za rješavanje SLAM problema.

4.3.1 Prošireni Kalmanov filter

Kalmanov filter je algoritam za procjenu stanja dinamičkog sustava. Algoritam kombinira predikciju stanja sustava i korekciju te procjenu na temelju mjerenja. Dvije osnovne komponente su:

- Model gibanja (eng. state transition model) koji opisuje mijenjanje sustava kroz vrijeme uzevši u obzir prethodno stanje i upravljanje:

$$x_k = F_k x_{k-1} + B_k u_k + w_k$$

gdje su:

  - x_k vektor stanja u trenutku k
  - F_k matrica prijelaza stanja
  - u_k vektor upravljanja u trenutku k
  - B_k matrica koja povezuje upravljanje sa stanjem
  - w_k vektor smetnje

- Model mjerenja (eng. measurement model) koji povezuje trenutno stanje sustava sa mjerenjima:

$$z_k = H_k x_k + v_k$$

gdje su:

  - z_k vektor mjerenja u trenutku k
  - H_k matrica koja povezuje stanje s mjerenjima
  - v_k vektor smetnje

Kalmanov filter ima dva glavna koraka: predikciju i ažuriranje.

1. Predikcija: koristi se model gibanja za procjenu stanja sustava u trenutku k na temelju stanja iz prethodnog trenutka k - 1:

$$\hat{x}_k^- = F_k \hat{x}_{k-1} + B_k u_k$$

Procjenjuje se i kovarijancijska matrica pogreške u procjeni:

$$P_k^- = F_k P_{k-1} F_k^T + Q_k$$

Ovdje $\hat{x}_k^-$ predstavlja predikciju stanja, a P_k^- predstavlja predikciju kovarijancijske matrice pogreške.

2. Ažuriranje: koristi se novo mjerenje z_k kako bi se korigirala predikcija:

$$K_k = P_k^- H_k^T \left(H_k P_k^- H_k^T + R_k \right)^{-1}$$

gdje je K_k Kalmanov dobitak koji određuje koliko će novo mjerenje utjecati na procjenu stanja. Ažurirana procjena stanja i kovarijacijska matrica pogreške računaju se kao:

$$\hat{x}_k = \hat{x}_k^- + K_k \left(z_k - H_k \hat{x}_k^- \right)$$

$$P_k = \left(I - K_k H_k \right) P_k^-$$

Ovim korakom kombiniraju se informacije iz predikcije i mjerenja, pri čemu se uzima u obzir njihova pouzdanost.

Glavni nedostatak Kalmanova filtra je njegova primjena samo na linearne sustave. Zbog tog razloga razvijen je prošireni Kalmanov filter (eng. extended Kalman filter, abbr. EKF) koji omogućuje rad s nelinearnim sustavima.

Osnova za EKF-SLAM je opisivanje gibanja u obliku:

$$P(x_k \mid x_{k-1}, u_k) \iff x_k = f(x_{k-1}, u_k) + w_k$$

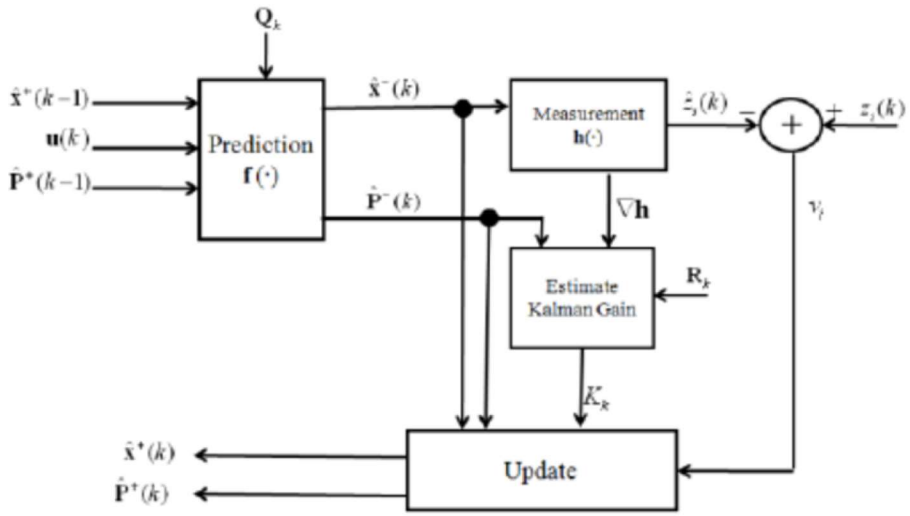
gdje $f(\cdot)$ modelira kinematiku vozila, a w_k su smetnje gibanja s kovarijancom Q_k. Model mjerenja je sljedeći:

$$P(z_k \mid x_k, m) \iff z(k) = h(x_k, m) + v_k$$

gdje $h(\cdot)$ opisuje geometriju mjerenja, a v_k su pogreške mjerenja s kovarijancom R_k. Sada možemo izračunati srednje vrijednosti $\hat{x}_k \mid k$ i $\hat{m}_k$, te kovarijance $P_{k|k}$ zajedničke posteriorne distribucije kroz dvije faze:

1. Predikcija: ažurira se poza robota i kovarijancija stanja koristeći funkciju gibanja, odnosno procjenjuje se nova poza robota i ažurira kovarijancija stanja na temelju kontrole i prošlih stanja.

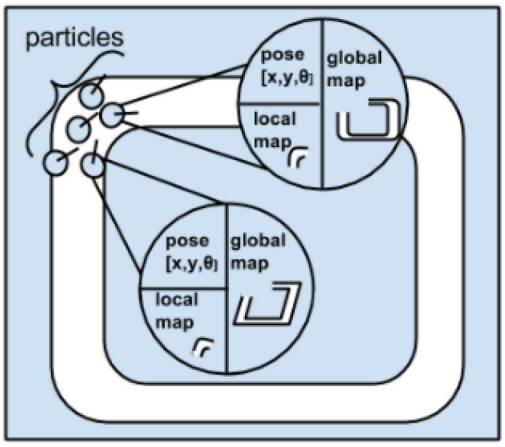
2. Ažuriranje: ažurira se procjena stanja i karte na temelju novih mjerenja koristeći funkciju mjerenja. U ovoj fazi se koristi novo mjerenje za korekciju predikcije stanja i ažuriranje položaja svih orijentira.



Slika 4.2: Prikaz EKF procesa. Slika preuzeta sa [4]

4.3.2 Čestični filter

Čestični filter koristi skup čestica koje predstavljaju vjerovanje robota $bel(x_k)$. Svaka čestica sadrži hipotezu o položaju robota, drugim riječima ona predstavlja jednu moguću poziciju robota, a pozicija može uključivati koordinate robota i orijentaciju. Dodatno, svaka čestica pretpostavlja da je njena pozicija točna i ona održava svoju vlastitu mapu.



Slika 4.3: Ilustracija čestica u različitim pozicijama. Slika preuzeta sa [5]

Postoji nekoliko koraka za implementaciju čestičnog filtra:

- Generiranje čestica
- Ažuriranje stanja robota u svakoj čestici uzevši u obzir podatke odometrije
- Prikupljanje podataka mjerenja
- Ažuriranje stanja robota u svakoj čestici uzevši u obzir podatke mjerenja
- Izračunavanje važnosti svake čestice koristeći razliku između stvarnog i pretpostavljenog mjerenja
- Ponovno uzrokovanje (eng. resample) čestica ovisno o njihovoj važnosti

Čestični filter kombinira navedene korake u korak predikcije i korak korekcije.

1. Korak predikcije: skup čestica ažurira se prema modelu gibanja koji procjenjuje novu poziciju robota. U tu svrhu koristi se model gibanja temeljen na odometriji:

$$x_k \sim p(x_k \mid x_{k-1}, a_k)$$

gdje su:

- x_k trenutna pozicija robota
- x_{k-1} prethodna pozicija
- a_k upravljački signal

Uz model gibanja temeljen na odometriji postoji i model gibanja temeljen na brzini. Dok se kod modela s odometrijom koriste informacije dobivene iz enkodera kotača, kod modela s brzinom koriste se translacijska i rotacijska brzina kako bi se procjenilo kretanje robota. Glavna razlika između njih je ta što model temeljen na brzini koristi unaprijed zadane brzine za predviđanje

kretanja robota, što može uzrokovati veće pogreške zbog neslaganja stvarnog ponašanja i modela, dok model temeljen na odometriji koristi stvarne podatke o kretanju iz enkodera kotača što pruža preciznije rezultate, ali i otežano dinamičko planiranje i kontrolu, jer se podaci dobivaju nakon pomaka robota.

Zaključno, model temeljen na odometriji pruža preciznije informacije i zato je odabran u ovom radu.

2. Korak korekcije: težine ili važnost svake čestice ažurira se na temelju razlike između stvarnih i predviđenih promatranja. Čestice koje bolje odgovaraju stvarnim mjerenjima imati će veću važnost.

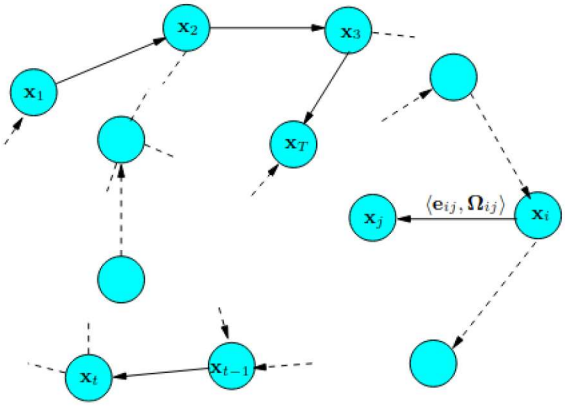
Nakon toga proces se ponavlja, gdje se uzimaju u obzir čestice sa većim težinama, dok čestice sa manjim se uklanjaju.

4.4 ROS SLAM

Postoji gotovo rješenje SLAM-a u ROS-u koje se može pronaći u *Cartographer* paketu. Metoda koja se koristi naziva se Pose Graph SLAM. Pose Graph SLAM je još jedno rješenje koje koristi graf gdje čvorovi grafa x_i predstavljaju različite poze robota, a bridovi e_{ij} označavaju transformaciju između tih poza temeljene na odometriji i mjerenjima senzora. Cilj je minimizirati ukupnu pogrešku $E(x)$, koja proizlazi iz tih transformacija:

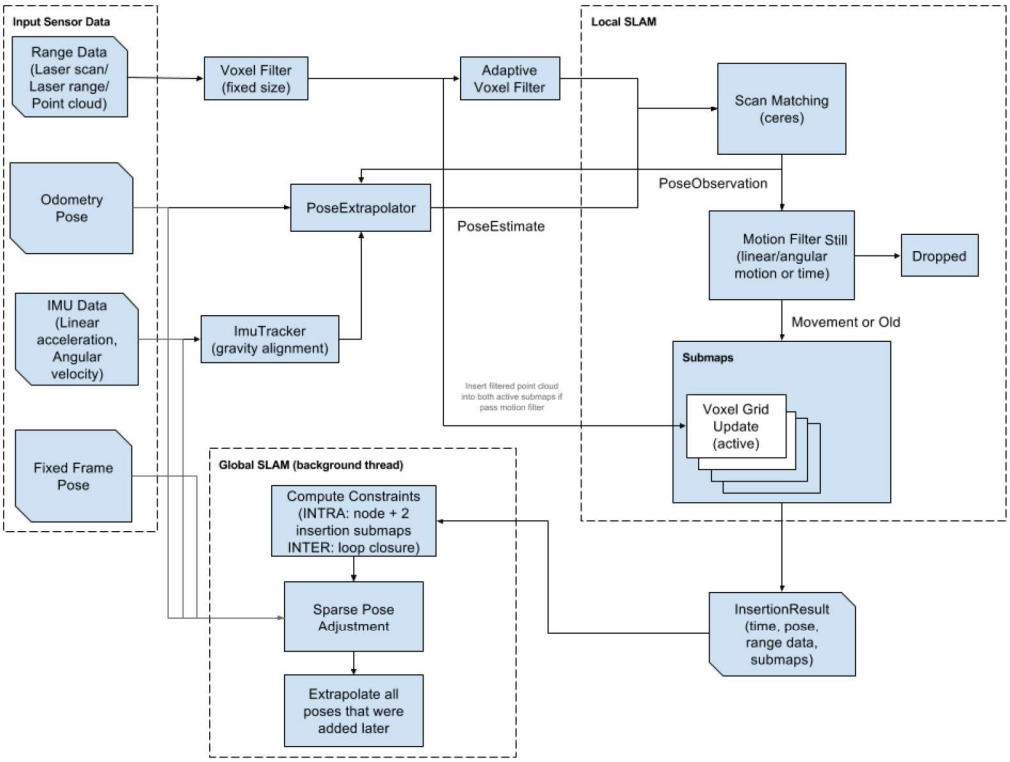
$$E(\mathbf{x}) = \sum_{(i,j)\in\mathcal{C}} \left\|\mathbf{e}_{ij} - \mathbf{z}_{ij}\right\|^2_{\mathbf{\Omega}_{ij}}$$

gdje z_{ij} predstavlja mjereni odnos između poza x_i i x_j, dok je $\mathbf{\Omega}_{ij}$ matrica informacija koja definira pouzdanost tog mjerenja.



Slika 4.4: SLAM proces u obliku grafa. Bridovi $e_{t-1,t}$ između uzastopnih poza predstavljaju odometrijska mjerenja, dok su ostali bridovi prostorna ograničenja nastala zbog višestrukih mjerenja istih dijelova. Slika preuzeta sa [2]

Vidljivo je kako je ROS-ova implementacija iznimno složena, s efikasnijim i suvremenijim metodama koje nadilaze okvir ovog rada. Unatoč tome, slika 4.5 prikazuje sažeti proces koji se odvija prilikom pokretanja ROS-ovog SLAM sustava.



Slika 4.5: Tehnički pregled Cartographer SLAM-a. Slika preuzeta sa [7]

5 | Izrada i primjena vlastitog ROS paketa

U ovoj cjelini prikazat ćemo vrlo jednostavan specijalizirani čvor koji komunicira sa TurtleBot-om i šalje mu naredbe kretnje uzevši u obzir mjerenja sa senzora. Svrha ovog poglavlja je pokazati lakoću razvoja vlastitih aplikacija sa ROS-om.

5.1 Proces izrade vlastitog čvora

Kako bismo razvili vlastiti čvor, prvo moramo kreirati vlastiti paket u kojem će se nalaziti čvor. Sljedeća naredba kreira paket:

```
$ ros2 pkg create name_of_package --build-type ament_python --
  dependencies rclpy
```

U radu je čvor napisan u Pythonu, pa dodatnim argumentom *build-type* specificiramo sistem izgradnje, dok *dependencies* argumentom navodimo da paket koristi biblioteku *rclpy*, koja je zaslužna za razvoj ROS aplikacija u Pythonu.

Sada kada je paket napravljen, potrebno je samo kreirati Python datoteku u paketu koja će se ponašati kao čvor. Za svrhu ovog rada kreiran je idući čvor:

```
#!/usr/bin/python3

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
import math
import random

class ObstacleAvoidance(Node):
    def __init__(self):
        super().__init__('obstacle_avoidance')

        qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            depth=10
        )
```

```
19
20              self.scan_subscriber = self.create_subscription(
21                  LaserScan,
22                  '/scan',
23                  self.scan_callback,
24                  qos_profile
25              )
26
27              self.cmd_vel_publisher = self.create_publisher(
28                  Twist,
29                  '/cmd_vel',
30                  10
31              )
32
33              self.safe_distance = 0.25
34
35              self.is_rotating = False
36              self.rotation_timer = None
37
38              self.get_logger().info('Obstacle Avoidance Node
    Initialized')
39
40              self.move_forward()
41
42          def scan_callback(self, msg):
43              if self.is_rotating:
44                  return
45
46              num_readings = len(msg.ranges)
47
48              front_left_idx = int(num_readings * 0.45)
49              front_right_idx = int(num_readings * 0.55)
50
51              front_ranges = msg.ranges[front_left_idx:
    front_right_idx]
52
53              valid_ranges = [r for r in front_ranges if msg.
    range_min <= r <= msg.range_max and not math.isinf(r) and not
    math.isnan(r)]
54
55              if not valid_ranges:
56                  self.get_logger().warn('No valid front range data
    available.')
57                  return
58
59              min_distance = min(valid_ranges)
60              self.get_logger().info(f'Minimum distance in front: {
    min_distance:.2f} meters')
61
62              if min_distance < self.safe_distance:
63                  self.get_logger().info('Obstacle detected!')
64                  self.rotate_robot()
65              else:
66                  self.move_forward()
67
68          def move_forward(self):
```

```
69              twist_msg = Twist()
70              twist_msg.linear.x = -0.05
71              twist_msg.angular.z = 0.0
72              self.cmd_vel_publisher.publish(twist_msg)
73              self.get_logger().info('Moving forward.')
74
75          def rotate_robot(self):
76              if self.is_rotating:
77                  return
78
79              self.is_rotating = True
80
81              random_angle = random.uniform(45, 135)
82              rotation_duration = random_angle / 45.0
83
84              angular_speed = 0.785
85              if random.choice([True, False]):
86                  angular_speed = -angular_speed
87
88              twist_msg = Twist()
89              twist_msg.linear.x = 0.0
90              twist_msg.angular.z = angular_speed
91              self.cmd_vel_publisher.publish(twist_msg)
92              self.get_logger().info(f'Rotating robot for {
    random_angle:.2f} degrees.')
93
94              self.rotation_timer = self.create_timer(
    rotation_duration, self.stop_rotation)
95
96          def stop_rotation(self):
97              self.rotation_timer.cancel()
98              self.rotation_timer = None
99
100             self.is_rotating = False
101             self.stop_robot()
102             self.move_forward()
103
104         def stop_robot(self):
105             twist_msg = Twist()
106             twist_msg.linear.x = 0.0
107             twist_msg.angular.z = 0.0
108             self.cmd_vel_publisher.publish(twist_msg)
109             self.get_logger().info('Robot stopped.')
110
111     def main(args=None):
112         rclpy.init(args=args)
113         obstacle_avoidance = ObstacleAvoidance()
114         rclpy.spin(obstacle_avoidance)
115         obstacle_avoidance.destroy_node()
116         rclpy.shutdown()
117
118     if __name__ == '__main__':
119         main()
```

Kako bi ROS aplikacija bila korektna, u svakoj datoteci treba se držati sljedećih

stvari:

- Svaka nova klasa zapravo predstavlja jedan čvor i zato svaka definirana klasa mora biti naslijeđena od klase *Node*. Dodatno se u konstruktoru mora inicijalizirati konstruktor izvorne klase, gdje kao argument prosljeđujemo ime čvora.
- Main funkcija uvijek mora biti idućeg oblika, gdje između inicijaliziramo instance naših čvorova:

```
1         rclpy.init(args=args)
2         ...
3         rclpy.shutdown()
4
```

Naš čvor se ponaša i kao izdavač i kao pretplatnik teme. Može se uočiti kako je čvor pretplaćen na temu *scan* koja ima vrstu poruke *LaserScan*. Drugim riječima, naš čvor prima podatke koje senzor sa TurtleBot-a odašilje. Uz to on je i izdavač teme *cmd-vel* koja ima vrstu poruke *Twist*. *Twist* predstavlja komandu kretnje, koju naš čvor šalje TurtleBot-u. Nakon transakcije TurtleBot se kreće obzirom na dobivenu poruku.

Glavni dio čvora je *scan-callback* funkcija koja se poziva svaki puta kada naš čvor primi poruku sa senzora. Funkcija uzima čitanja senzora ispred robota i provjerava ima li robot prepreku ispred sebe. Ukoliko ima, on se okreće za nasumično odabran kut i nastavlja se kretati naprijed.

5.2 Pokretanje čvora i TurtleBot-a

Za uspješno pokretanje SLAM-a na TurtleBot-u potrebno je slijediti nekoliko koraka:

1. Na TurtleBot-u pokrenut sljedeću naredbu u konzoli:

```
1         $ export TURTLEBOT3_MODEL=burger
2
```

2. Na TurtleBot-u pokrenuti *bringup* launch datoteku koja inicijalizira sve potrebne senzore i funkcionalnosti robota.
3. Pokrenuti naš specijalizirani čvor za autonomnu vožnju robota (ili možemo pokrenuti *turtlebot3-teleop-key* čvor za kontroliranje robota uz pomoć tipkovnice).
4. Pokrenuti *Cartographer* čvor, koji pokreće SLAM.

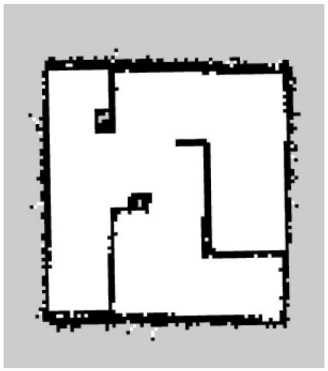
Kako bi olakšali pokretanje svih potrebnih čvorova na računalu, kreiramo vlastitu launch datoteku koja ujedinjuje sve potrebne korake i uzbrava proces.

```
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
PythonLaunchDescriptionSource
import os
from ament_index_python import get_package_share_directory

def generate_launch_description():
    ld = LaunchDescription()

    robot_launch_file = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('
turtle_bringup'), 'launch/bringup.launch.py')
        )
    )

    slam_launch_file = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('
turtlebot3_cartographer'), '/opt/ros/humble/share/
turtlebot3_cartographer/launch/cartographer.launch.py')
        )
    )

    ld.add_action(robot_launch_file)
    ld.add_action(slam_launch_file)

    return ld
```

Uočimo kako u *robot-launch-file* i *slam-launch-file* navodimo putanju gdje se ta datoteka može pronaći i samo funkcijom *add-action* navodimo koje sve launch datoteke želimo ujediniti. Sada se cijeli proces svodi na pokretanje samo jedne naredbe:

```
$ ros2 launch turtle_bringup start.launch.py
```

Ukoliko su svi navedeni koraci izvršeni, robot će se uspješno pokrenuti i prikazivati će generiranu mapu na računalu. Primjer generirane mape je idući:



Slika 5.1: Primjer generirane mape uz pomoć SLAM-a. Slika preuzeta sa [8]

5.3 Zaključak

Zaključno, ovaj rad demonstrira osnove ROS-a i TurtleBot-a i isto tako pruža temelj za razumijevanje SLAM-a i njegove primjene u robotici. Nadamo se da će ovaj rad motivirati druge da se uključe u svijet robotike i potaknuti daljnja istraživanja, bilo kroz optimizaciju SLAM algoritama ili integraciju s novim tehnologijama.

Literatura

[1] H. Durrant-Whyte, Fellow, IEEE, T. Bailey, *Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms*.

[2] G. Grisetti, R. Kummerle, C. Stachniss, W. Burgard, *A Tutorial on Graph-Based SLAM*, Department of Computer Science, University of Freiburg, 79110 Freiburg, Germany.

[3] G. Welch, G. Bishop, *An Introduction to the Kalman Filter*, Department of Computer Science University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175.

[4] J. Kang, W. Choi, S. An, S. Oh, *Augmented EKF based SLAM method for improving the accuracy of the feature map*.

[5] Yatim, Norhidayah, Norlida, *Particle filter in simultaneous localization and mapping (Slam) using differential drive mobile robot*, Jurnal Teknologi, 2015, 77. DOI: 10.11113/jt.v77.6557.

[6] Y. Pyo, H. Cho, R. Jung, T. Lim, *ROS Robot Programming*, ROBOTIS Co., Ltd., Seoul, 2017.

[7] ROS2 Documentation.
URL: https://docs.ros.org/en/humble/index.html.

[8] TurtleBot3 Documentation.
URL: https://www.turtlebot.com/turtlebot3/.

Sažetak

U ovom radu uspješno smo demonstrirali SLAM algoritam na platformi TurtleBot3 koristeći Robot Operating System (ROS). Kroz proces razvijanja vlastitih čvorova i konfiguriranja potrebnih launch datoteka, prikazali smo jednostavnost i efikasnost ROS-a, dok se TurtleBot3 pokazao kao izvrsna platforma za ulazak u svijet robotike svojim jednostavnim korištenjem i fleksibilnošću. Isto tako, teorijski smo prikazali problem istovremene lokalizacije i mapiranja, koji se svrstava u najvažnije koncepte robotike. Pregledali smo različite pristupe rješavanja tog problema, koji i dan danas predstavlja aktivno područje istraživanja i optimizacije.

Ključne riječi

SLAM, TurtleBot3, Robot Operating System, ROS, robotika, optimizacija algoritama, robotske aplikacije, autonomna vožnja

Naslov rada na engleskom jeziku

Summary

In this work, we successfully demonstrated the SLAM (Simultaneous Localization and Mapping) algorithm on the TurtleBot3 platform using the Robot Operating System (ROS). Through the development of custom nodes and the configuration of necessary launch files, we showcased the simplicity and efficiency of ROS, while TurtleBot3 proved to be an excellent platform for entering the field of robotics due to its ease of use and flexibility. Additionally, we provided a theoretical overview of the SLAM problem, which is considered one of the fundamental concepts in robotics. We reviewed various approaches to solving this problem, which remains an active area of research and optimization today.

Keywords

SLAM, TurtleBot3, Robot Operating System, ROS, robotics, algorithm optimization, robot aplications, autonomous driving