SLAM postupak s dvostrukim ultrazvučnim daljinomjerom

Smolec, Dominik

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Dominik Smolec

Zagreb, 2024 godina.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Mladen Crneković, dipl. ing.

Student:

Dominik Smolec

Zagreb, 2024 godina.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se svim profesorima koji su me pratili kroz moje obrazovanje, te se zahvaljujem na svom prenesenom znaju. Posebno se zahvaljujem profesoru Mladenu Crnekoviću na mentorstvu i ukazanom strpljenju.

Dominik Smolec

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
Proizvodno inženjerstvo, inženjerstvo materijala, industrijsko inženjerstvo i menadžment, mehatronika i robotika, autonomni sustavi i računalna inteligencija

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 - 04 / 24 - 06 / 1**	
Ur.broj: **15 - 24 -**	

DIPLOMSKI ZADATAK

Student: **Dominik Smolec** JMBAG: **0035219152**

Naslov rada na hrvatskom jeziku: **SLAM postupak s dvostrukim ultrazvučnim daljinomjerom**

Naslov rada na engleskom jeziku: **SLAM procedure with dual ultrasonic range finder**

Opis zadatka:

Preduvjet uspješnog rada svakog mobilnog robota je poznavanje okoline u kojoj se robot kreće te njegovog položaja u toj okolini. To se postiže senzorima unutarnjih stanja (odometrija) te senzorima vanjskih stanja (npr. rotirajući ultrazvučni daljinomjer). Kombinacijom dobivenih informacija računa se vjerojatan položaj i orjentacija mobilnog robota u zadanoj okolini. Taj je postupak u literaturi poznat kao SLAM postupak.

Potrebno je procijeniti mogućnost korištenja jeftinog rotirajućeg ultrazvučnog daljinomjera za uspješno rješenje SLAM postupka.

U radu je potrebno:

- pomoću odabranog dvostrukog ultrazvučnog daljinomjera generirati podatke o stanju okoline (smjer i udaljenost prepreke),
- pomoću enkodera preuzeti odometrijske podatke o prijeđenom putu i skretanju,
- grafički prikazati podatke o zadanoj okolini i rezultatu SLAM postupka,
- grafički prikazati procijenjeni položaj i orijentaciju robota.

Potrebno je navesti korištenu literaturu i ostale izvore informacija te eventualno dobivenu pomoć.

Zadatak zadan:	Datum predaje rada:	Predviđeni datumi obrane:
18. siječnja 2024.	21. ožujka 2024.	25. – 29. ožujka 2024.

Zadatak zadao:

Prof. dr. sc. Mladen Crneković

Predsjednik Povjerenstva:

Prof. dr. sc. Ivica Garašić

SADRŽAJ

POPIS SLIKA

SAŽETAK

Diplomski rad istražuje primjenu SLAM (Simultaneous Localization and Mapping) postupka primjenom Lidar senzora, ključnog za precizno određivanje položaja i mapiranje okoline mobilnog robota. Kombinirajući informacije o unutarnjim stanjima kroz odometriju i vanjskim stanjima putem Lidar-a, postiže se precizna procjena položaja i orijentacije robota u zadanoj okolini. Korištenjem odabranog Lidar-a, generiraju se podaci o stanju okoline poput smjera i udaljenosti prepreka, dok se pomoću enkodera prikupljaju odometrijski podaci o prijeđenom putu i skretanju. Konačno, rezultati SLAM postupka i procijenjeni položaj i orijentacija robota grafički se prikazuju, pružajući vizualni uvid u okolinu i lokaciju robota.

Rad ima za cilj doprinijeti razumijevanju i implementaciji SLAM postupka, pomoću jednostavnih i široko dostupnih uređaja, što može imati značajne implikacije u području razvoja mobilnih robota. Kroz praktičnu primjenu u realnom okruženju, ovaj rad može potaknuti daljnja istraživanja i razvoj u području robotike, posebno u autonomnoj navigaciji i mapiranju okoline.

Ključne riječi: Mobilni robot, SLAM, senzori, mapiranje

SUMMARY

The master's thesis explores the application of the SLAM (Simultaneous Localization and Mapping) procedure utilizing Lidar sensors, crucial for accurately determining the position and mapping the environment of a mobile robot. By combining information on internal states through odometry and external states via Lidar, a precise estimation of the robot's position and orientation in the given environment is achieved. Using the selected Lidar, environmental data such as direction and obstacle distance are generated, while odometric data on traveled distance and steering are collected using encoders. Finally, the results of the SLAM procedure and the estimated position and orientation of the robot are graphically displayed, providing a visual insight into the environment and robot location.

The aim of the paper is to contribute to the understanding and implementation of the SLAM procedure, using simple and widely available devices, which could have significant implications in the field of mobile robot development. Through practical application in a real environment, this paper may stimulate further research and development in the field of robotics, particularly in autonomous navigation and environment mapping.

Keywords: Mobile robot, SLAM, sensors, mapping

1. UVOD

Navigacija mobilnih robota kroz okolinu predstavlja kompleksan izazov u području robotike i automatizacije. Ovaj problem obuhvaća niz izazova, uključujući percepciju okoline, planiranje putanje i kontrolu kretanja robota. Evo nekoliko ključnih aspekata problema navigacije mobilnih robota:

- **Percepcija okoline**: Mobilni roboti moraju imati sposobnost opažanja i razumijevanja svoje okoline kako bi donosili informirane odluke o kretanju. Senzori poput kamere, lidara, ultrazvučnih daljinomjera i drugih igraju ključnu ulogu u prikupljanju podataka o okolini, preprekama i drugim relevantnim informacijama.
- **Planiranje putanje**: Na temelju informacija o okolini, mobilni roboti trebaju razviti strategiju kako sigurno i učinkovito doći od trenutačne pozicije do cilja. Ovaj proces uključuje analizu dostupnih putanja, izbjegavanje prepreka, optimizaciju rute te uzimanje u obzir dinamičkih promjena u okolini.
- **Simultano određivanje položaja i mapiranje (SLAM)**: Ovaj problem se odnosi na sposobnost robota da istovremeno određuje svoj položaj u okolini i mapira okolinu. SLAM postupci su ključni za mobilne robote koji se kreću u nepoznatim ili dinamičnim okruženjima, omogućujući im stvaranje i ažuriranje karte okoline dok istovremeno održavaju preciznu lokalizaciju.
- **Prilagodba dinamičkim okruženjima**: Okoline se mogu mijenjati tijekom vremena, što uključuje pojavu ili nestanak prepreka, promjene u osvjetljenju ili ponašanju drugih objekata. Mobilni roboti moraju biti sposobni prilagoditi svoje strategije kretanja kako bi učinkovito reagirali na ove promjene.
- **Komunikacija među robotima**: U scenarijima gdje više mobilnih robota surađuje ili dijeli isto radno okruženje, važna je međusobna komunikacija. To može uključivati dijeljenje informacija o položaju, planiranje zajedničkih ruta ili suradnju u rješavanju problema navigacije.

Rješavanje problema navigacije mobilnih robota predstavlja multidisciplinarni izazov koji uključuje aspekte računalnog vida, strojnog učenja, planiranja algoritama i kontrolnih sustava.

Inovacije u ovom području imaju značajan potencijal za primjenu u industriji, logistici, medicini, istraživanju prostora i drugim područjima.

1.1. Percepcija mobilnih robota

Percepcija okoline mobilnih robota igra ključnu ulogu u njihovoj autonomnoj navigaciji i prilagodbi okolini. Različiti senzori igraju ključnu ulogu u prikupljanju informacija o okolini i omogućavaju robotima da donose informirane odluke o kretanju. Evo nekoliko ključnih senzora i aspekata percepcije okoline:

- **Vizualni senzori**: Kamere i sustavi računalnog vida omogućuju robotima analizu vizualnih informacija iz okoline. Prepoznavanje objekata, ljudi i prostornih karakteristika omogućuje precizniju navigaciju.
- **LIDAR**: Koristi laserske zrake za mjerenje udaljenosti i stvaranje trodimenzionalnih ili dvodimenzionalnih mapa okoline. Precizno otkrivanje prepreka i identifikacija terena pomažu u sigurnom kretanju robota.
- **Ultrazvučni senzor**: Mjeri udaljenost pomoću zvučnih valova i koristi se za otkrivanje objekata u blizini. Posebno koristan u situacijama s niskim osvjetljetljenjem.
- **Radarski senzori**: Koriste elektromagnetske valove za detekciju objekata i mjerenje udaljenosti. Učinkoviti su na većim udaljenostima i u različitim vremenskim uvjetima.
- **Enkoderi**: Praćenje putanje kretanja pomoću enkodera na kotačima robota. Doprinose unutarnjoj lokalizaciji i praćenju relativnog pomaka robota.

Integracija informacija iz ovih senzora omogućuje robotima stvaranje cjelovite slike okoline. U ovom diplomskom radu koristiti ću LIDAR za percepciju okoline, te enkodere za praćenje pozicije robota.

1.1.1. Lidar

U ovom poglavlju ću se detaljnije dotaknuti Lidar-a, jer će on biti jedini uređaj koji će nam davati informacije o okruženju robota. Lidar tj. Light Detection and Ranging je tehnologija koja se koristi za mjerenje udaljenosti između uređaja i objekata koristeći laserske zrake. Ova tehnologija koristi princip sličan radaru, ali umjesto radiovalova koristi laserske zrake. Lidar sustav, dok se okreće relativno velikom, šalje kratke impulse laserske svjetlosti prema cilju, a zatim mjeri vrijeme potrebno da se svjetlost reflektira i vrati nazad do prijemnika. Na temelju vremena koje je potrebno za refleksiju, Lidar može precizno izračunati udaljenost do objekta, te pomoću preciznog enkodera može izračunati pod kojim kutom se taj objekt nalazi u odnosu na koordinatni sustav Lidar-a.

U posljednjih nekoliko godina, Lidar je postao ključna tehnologija u razvoju autonomnih vozila. Koristeći Lidar, vozila mogu precizno detektirati i pratiti objekte u njihovoj okolini, omogućujući im da sigurno navigiraju kroz kompleksne prometne situacije.

Lidar koji ću koristiti kao referencu svojemu radu je RPLIDAR A2, prikazan na slici dolje.

Slika 1. Slamtec-ov RPLIDAR A2

Ovaj Lidar od proizvođača Slamtec-a ima domet skeniranja 0.2 – 12 metara, te može skenirati punih 360 stupnjeva sa kutnom rezolucijom od 0.9 stupnjeva, i sve to čak 10 puta u jednoj sekundi!

Ovaj Lidar ima jedan od najboljih omjera cijene i kvalitete, te ima puno mogućnosti parametrizacije rada. Ukoliko nam ne treba detaljna slika okoline, možemo na primjer smanjiti kutnu rezoluciju ili smanjiti brzinu okretanja Lidar-a kako bi smanjili količinu podataka koja dolazi od ovog uređaja.

Unatoč svojim mnogobrojnim prednostima, Lidar također ima neka ograničenja. Na primjer, jak sunčev svjetlost ili magla mogu ometati laserske signale, što može dovesti do nepreciznih ili nedostupnih podataka. Također, Lidar sustavi mogu biti skupi za implementaciju i održavanje, što može ograničiti njihovu širu primjenu u nekim područjima. Međutim, tehnološki napredak i pad cijena opreme vjerojatno će doprinijeti široj upotrebi Lidara u različitim industrijama u budućnosti.

1.1.2. Odometrija

Odometrija je tehnika koja se koristi za praćenje kretanja mobilnih robota na osnovu podataka koje generiraju senzori na robotu. Jedan od ključnih senzora koji se koristi u odometriji mobilnih robota su enkoderi.

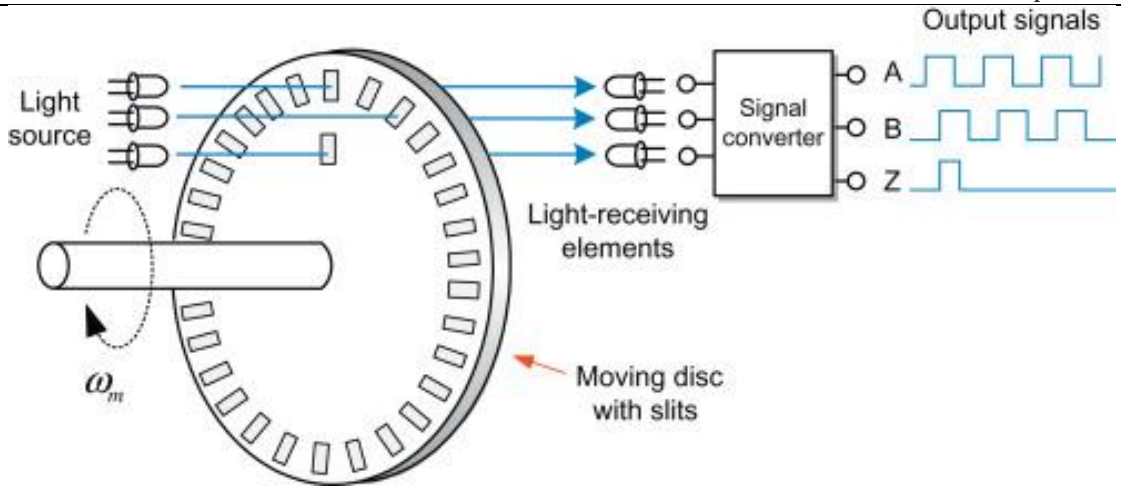
Enkoderi su senzori koji mjere rotacijsko kretanje kotača mobilnog robota. Postavljeni su na osovinama kotača i generiraju impulsne signale ovisno o brzini i smjeru rotacije kotača. Na temelju podataka dobivenih od enkodera, mobilni robot može izračunati svoj trenutni položaj i orijentaciju u odnosu na početnu točku. Korištenjem informacija o duljini kretanja i smjeru rotacije kotača, algoritmi odometrije mogu procijeniti putanju kretanja robota.

Enkoderi mogu biti jako točni oko pozicije kotača, ali ta točnost se ne prenosi na preciznost pozicije robota iz razloga:

- Klizanje kotača koje također ovisi i o podlozi, koja se može i mijenjati, po kojoj se mobilni robot kreće
- Nesavršena kružnost kotača, te neujednačenost dimenzija oba kotača
- Neujednačenost terena koja je posebno izražena u vanjskim okruženjima

Postoje dvije glavne skupine enkodera koji se koriste u robotici: Inkrementalni i apsolutni.

Apsolutni enkoderi generiraju jedinstveni kod za svaku moguću poziciju rotacije osovine. Način na koji dolaze do te jedinstvene vrijednosti varira ovisno o tipu enkodera. Jedan od načina generiranja jedinstvenog koda je korištenje optičkih ili magnetskih traka koje su kodirane na specifičan način. Svaki enkoder ima ugrađene senzore koji čitaju kodirane trake i prevode ih u digitalni ili analogni signal koji predstavlja poziciju rotacije. Kodirane trake mogu biti dizajnirane na različite načine kako bi se osigurala jedinstvenost koda za svaku poziciju. Ovaj jedinstveni kod omogućuje precizno praćenje položaja rotacijskih komponenti u različitim aplikacijama.

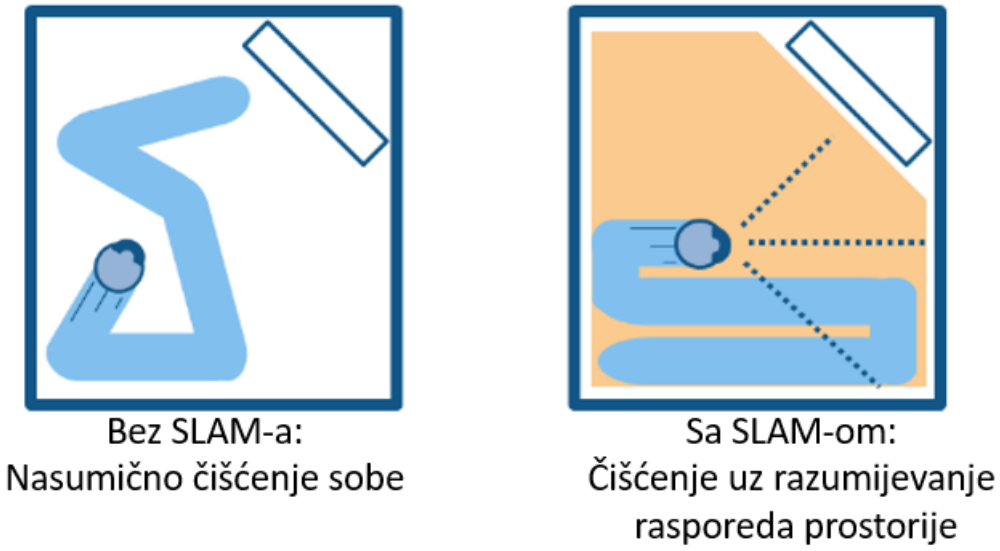


Slika 2.Shematski prikaz enkodera

Inkrementalni enkoderi su senzori koji bilježe rotacijsko kretanje i generiraju signale kako bi se odredila brzina i pomak. Oni pružaju informacije o relativnom pomaku ili brzini rotacije, ali ne daju apsolutnu poziciju. Njihova struktura obično uključuje rotirajući disk s rupama ili prorezima, koji se čita pomoću senzora. Postoje dvije glavne vrste inkrementalnih enkodera: kvadratni, koji daju četiri signala po rotaciji, omogućujući određivanje smjera i brzine, te sinusoidalni, koji daju glađe signale, korisne u visokopreciznim aplikacijama. Ovi enkoderi se široko koriste u industriji, robotici i drugim područjima gdje je potrebno precizno praćenje rotacijskog kretanja. Iako su brzi, jednostavni za izradu i relativno jeftini, važno je imati na umu da ne daju informacije o apsolutnoj poziciji, što može biti ograničavajuće u nekim situacijama.

1.2. SLAM

Simultano određivanje položaja i mapiranje (SLAM) je ključni postupak u području robotike, posebice kada je riječ o autonomnim sustavima kretanja, poput mobilnih robota ili bespilotnih letjelica. Osnovna svrha SLAM-a je omogućiti robotu da istovremeno određuje svoj položaj u nepoznatoj okolini i stvara mapu te okoline.



Slika 3. Primjer SLAM algoritma

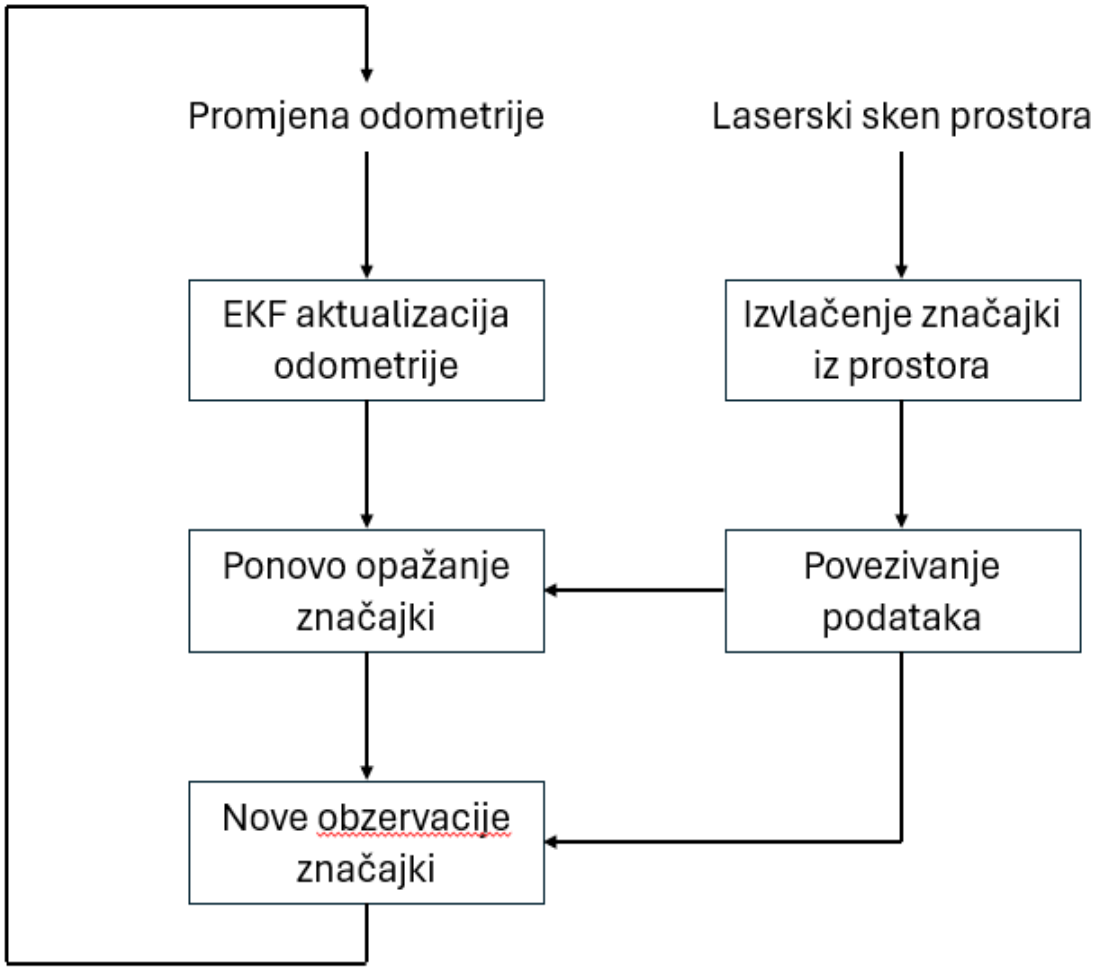
Postupak SLAM-a koristi senzore i podatke o kretanju robota kako bi sustav održao preciznost svoje lokalizacije i konstruirao mapu prostora kroz koji se kreće. Senzori poput kamera, Lidar-a, radara i ultrazvučnih daljinomjera koriste se za prikupljanje informacija o okolini, dok se podaci o kretanju mogu dobivati pomoću odometrije ili drugih senzora.

Proces SLAM-a obično uključuje dva ključna koraka: predviđanje i korekciju. U koraku predviđanja, robot koristi informacije o svojoj prethodnoj poziciji i kretanju kako bi procijenio svoju sljedeću poziciju. Nakon toga, u koraku korekcije, robot uspoređuje svoje stvarne opažaje okoline s očekivanjima temeljenima na prethodnom predviđanju te prilagođava svoju procjenu položaja.

Ova tehnika omogućuje robotima da navigiraju i mapiraju nepoznate ili dinamične okoline, kao što su prostori u kojima se okolina može mijenjati ili prostori koji su nedovoljno kartirani. SLAM postupak ima široku primjenu, uključujući autonomna vozila, bespilotne letjelice, robotsko istraživanje nepoznatih područja te druge scenarije gdje je preciznost lokalizacije i mapiranja ključna za uspješno izvršavanje zadataka.

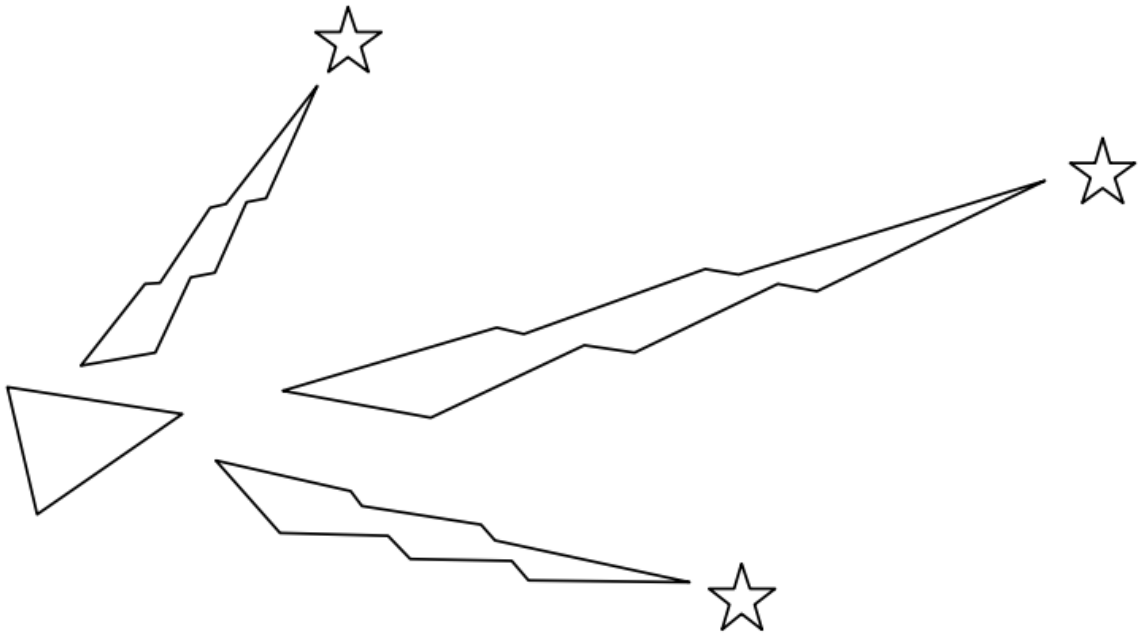
2. SLAM PROCES

SLAM proces se sastoji od nekoliko koraka, a u ovom poglavlju ću svaki korak detaljno objasniti. Ovaj moj primjer je sam jedan od mnogih mogućih izvedbi SLAM algoritma, ali svaki algoritam ima isti krajnji cilj, a to je lokalizacija robota uz pomoć njegove okoline i mapiranje prostora. Lokalizacija robota se vrši uz pomoć laserskog skeniranja okoline, te izvlačenja značajki iz okoline i ponovnog opažanja već viđenih i zapamćenih značajki. Prošireni Kalmanov filtar (EKF) je ključan za SLAM proces. Pomoću njega algoritam aktualizira poziciju robota uzimajući u obzir i odometriju i ponovo viđene značajke prostora. Osnovni prikaz SLAM postupka se može vidjeti na slici ispod.

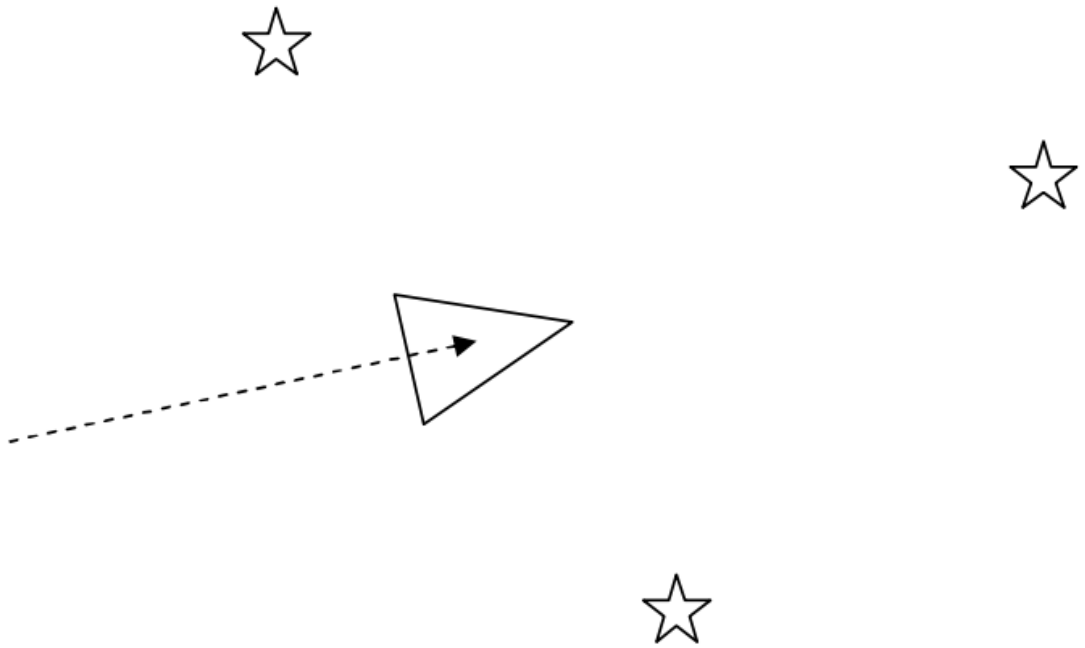


Slika 4. Pregled SLAM procesa

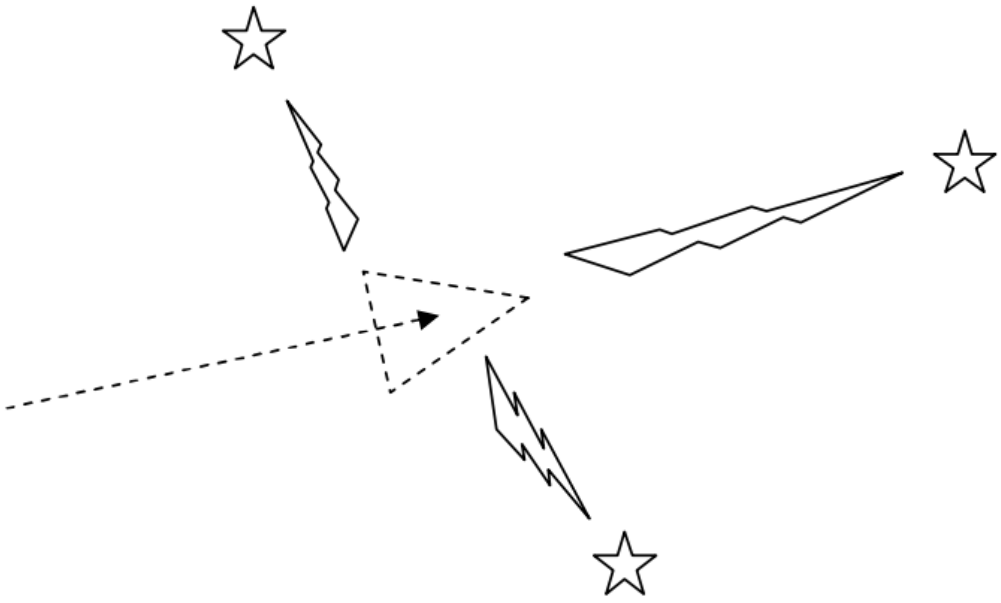
Kada se odometrija promijeni zbog pomaka robota, nova pozicija robota je upisana u EKF. Tada se, uz trenutnu poziciju robota, iz laserskog skena izvlače prostorne značajke. Algoritam tada pokuša povezati već viđene značajke sa novo viđenim značajkama te ih pokušava povezati. Ponovo opažene značajke pomažu pri ažuriranju pozicije robota, dok se novo viđene značajke pamte kako bi ih kasnije mogli ponovo opaziti i povezati. Svi ovi koraci će u daljnjem tekstu biti detaljno objašnjeni. Skup slika ispod vizualizira SLAM proces.



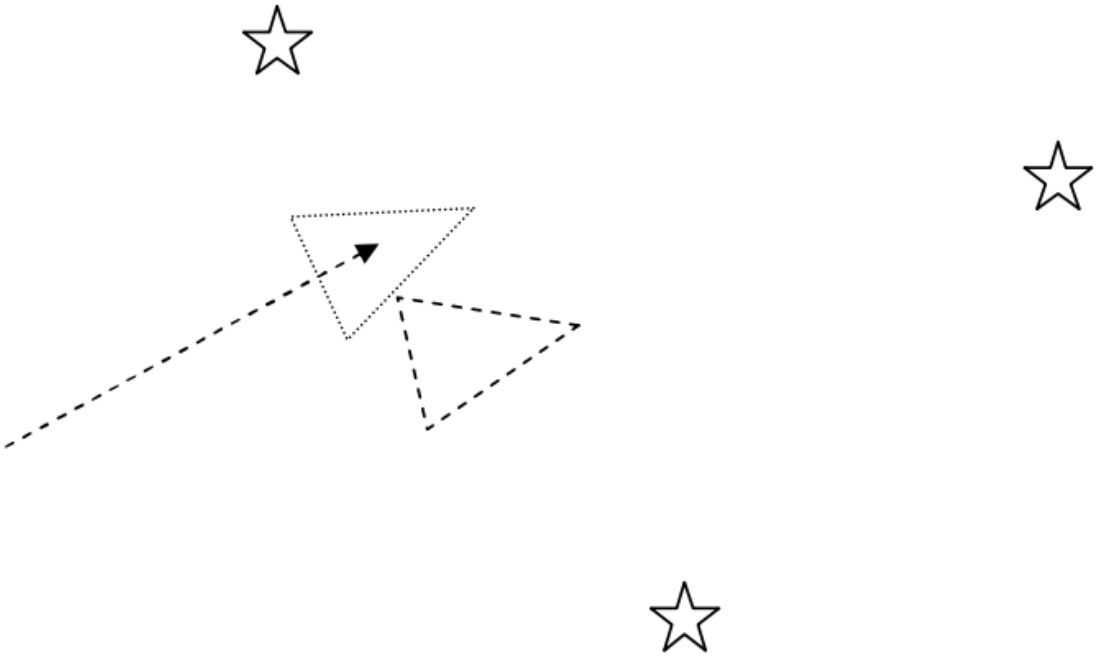
Slika 5. Trokut označava robota, a zvijezde značajke. Robot je pomoću Lidar-a odredio pozicije značajki



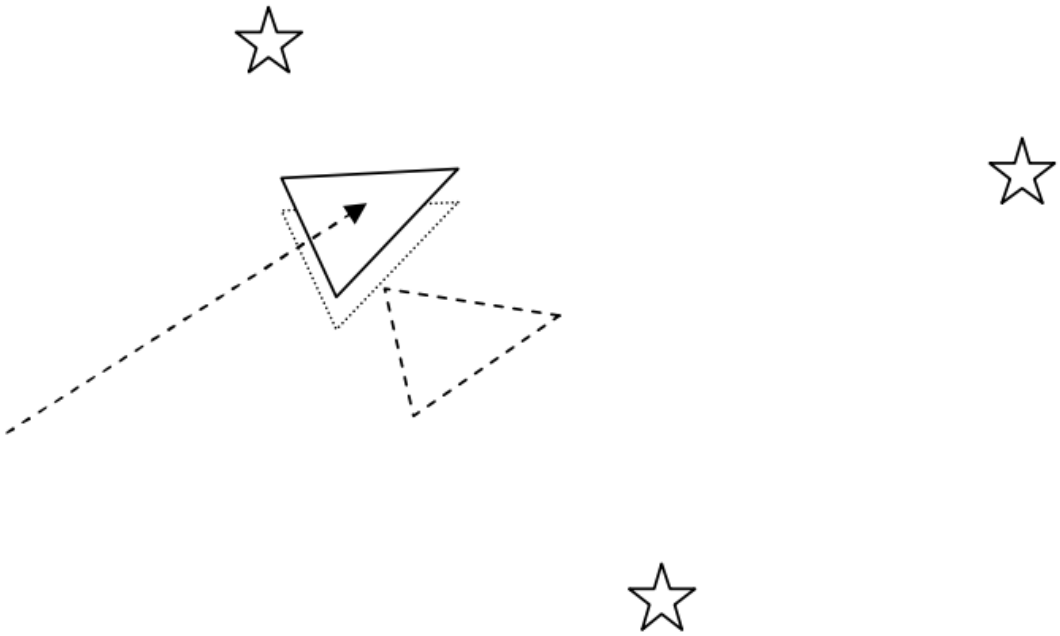
Slika 6. Robot se kreće, te odometrijom određujemo gdje se otprilike nalazi



Slika 7. Lidar-om ponovo skeniramo prostor ponovo određujemo poziciju značajki, te zaključujemo da se robot nalazi na drugom mjestu od planiranog



Slika 8. Pošto je Lidar precizniji, pomoću pozicije značajki robot određuje gdje se nalazi. Pozicija preko odometrije je



Slika 9. Kako ni Lidar nije savršen, robot se nalazi na različitoj poziciji od predviđene, označene sa debljom crtom

2.1. Sken prostora pomoću Lidar-a

SLAM proces započinje skeniranjem prostora uz pomoć Lidar-a kako bi se dobila informacija o okruženju robota. Podatke okruženja dobivamo, od uređaja koji upravlja Lidar-om (mikroupravljač), u obliku matrice, gdje je prvi stupac kut, a drugi stupac udaljenost, dok redovi označavaju po jedno mjerenje. Mikroupravljač daje naredbe Lidar-u u obliku {0xA5, naredba, opcionalna sekcija}. Jedne od glavnih naredbi su STOP({0xA5, 0x25}), SCAN({0xA5, 0x20}), GET_INFO({0xA5, 0x50}), itd. Postoje dvije glavne opcije kako ćemo dobiti informacije od Lidar-a, jedna je SCAN, druga je EXPRESS_SCAN. Kada mu damo naredbu SCAN, Lidar će napraviti jedno mjerenje i vratiti rezultat i čekati će iduću naredbu. Kada napravimo EXPRESS_SCAN, Lidar će krenuti raditi sken za skenom, te će konstantno vraćati informacije, te kada prođe 360° to će signalizirati jednim bitom „S“.

SLAMTEC-ov RPLidar definira 360° sa oko 400 čitanja, što znači da imamo rezoluciju od oko 0.9°, dok udaljenost koju može izmjeriti se nalazi između 0.15 i 8 metara (8 metara je kod uvjeta da skeniramo bijeli predmet sa 70% refleksije), sa greškom od 0.75% vrijednosti udaljenosti. Pokušati ću ovo ponašanje što stvarnije simulirati u svojem programu dodavanjem raznih Gausovih šumova

2.2. Odometrijski podaci

Cilj odometrijskih podataka je da opiše približnu poziciju mobilnog robota, koja će se onda koristiti za početnu pretpostavku globalne pozicije prostornih značajki. Odometrijski podaci se mogu dobivati kontinuirano, ciklično ali i po raznim dogovorenim protokolima, kao npr. nadležno računalo zatraži odometrijske podatke kada mu je to potrebno. U ovom radu računalo će u specifičnim trenutcima tražiti informacije o odometriji od robota.

2.3. Prostorne značajke

Prostorne značajke se moraju moći jednostavno pronaći u okruženju robota. Pomoću njih robot može točnije saznati svoju trenutnu poziciju. Kao što je rečeno prije, značajke moraju biti nepokretne i očite, kao što ću ja u ovom radu koristiti zidove prostorije kao prostorne značajke. Iz okoliša bi trebali izvući sve moguće značajke kako bi probali robotu u svakom njegovom koraku omogućiti ponovni pronalazak značajki, jer ako prođe korak a da

nema ni jedne značajke, u idućem koraku će potencijalno odometrijska pogreška biti jako velika, što bi moglo uzrokovati nemogućnošću uparivanja novih i starih značajki. Također problem koji može nastati je ako su dvije značajke jako blizu i jako slične može se dogoditi da uparimo novu značajku sa krivom starom značajkom. Ako sumiramo, glavne točke prostornih značajki su sljedeće:

- Značajke bi se trebale jednostavno ponovo uočiti.
- Pojedinačne značajke bi trebale biti razlikovljive jedne od drugih
- Značajki bi trebalo biti puno u okruženju robota
- Značajke moraju biti nepokretne

2.4. Izdvajanje prostornih značajki

Izdvajanje prostornih značajki je prvi veliki aspekt algoritma koji se razlikuje između različitih SLAM algoritama. Koje značajke ćemo izdvajati ovisi o primjeni i okolišu u kojem će naš robot raditi. U nastavku ovoga teksta ću objasniti dva algoritma za dvije različite primjene: „Spike“ i RANSAC algoritam izdvajanja prostornih značajki.

2.4.1. „Spike“ izdvajanje prostornih značajki

„Spike“ algoritam pronalazi prostorne značajke nalaskom ekstremiteta u prostoru, tj. traži gdje se u prostoru oko robota dvije točke razlikuju u udaljenosti za više od neke definirane vrijednosti. Algoritam pronalazi velike skokove udaljenosti kada se npr. jedan dio laserskog skena odbije od zida, a jedan mali dio od noge stola koji stoji ispred zida. „Spike“ algoritam nije povoljan za „glatka“ okruženja gdje se nalazi puno ravnih ploha, tj. zidova. Pošto će naš robot biti baš u takvom okruženju, ovaj algoritam nećemo koristiti.

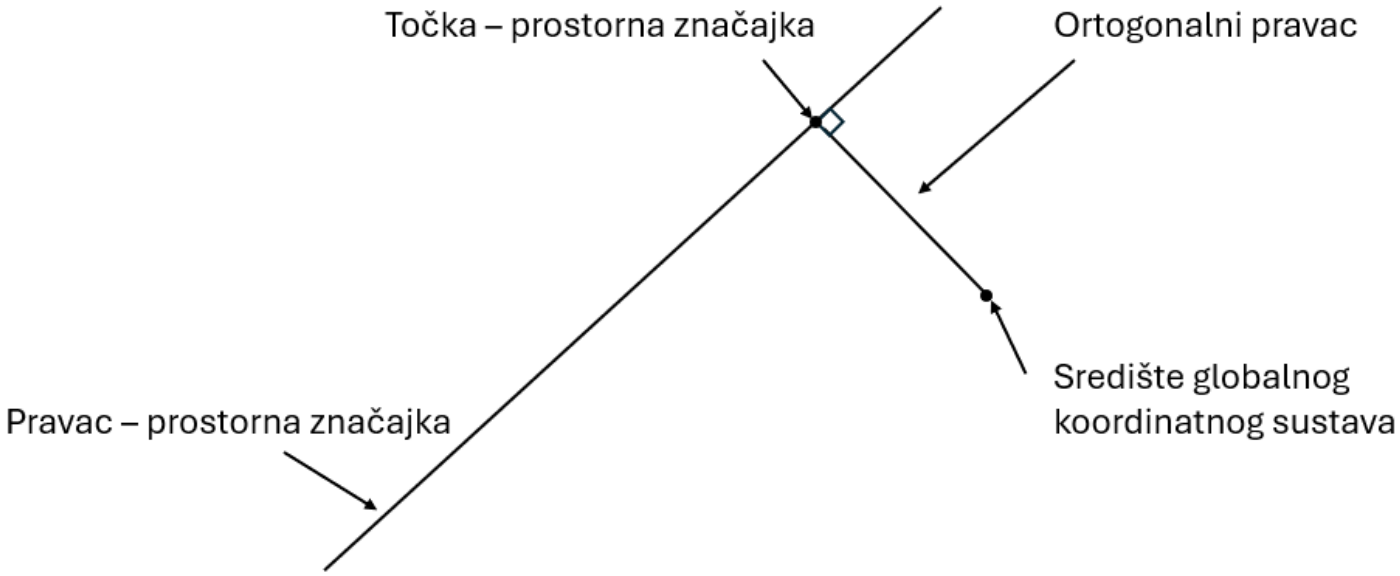
2.4.2. RANSAC algoritam

RANSAC (Random Sampling Consensus) je algoritam koji služi za pronalazak ravnih linija u laserskom skenu, kojih u zatvorenom prostoru ima na pretek. Ove linije kasnije koristimo kao prostorne značajke po kojima se robot može orijentirati u prostoru.

RANSAC algoritam će ove linije pronaći tako da uzme nasumične uzorke u podacima laserskog skena, te pokuša provući liniju kroz te točke koristeći metodu aproksimacije najmanjih kvadrata. Ovo može raditi jer će okolina, u kojoj će se robot nalaziti, biti prepuna značajki, te ćemo uzeti poprilično veliki uzorak (u ovom radu to će biti 25 točaka). Nakon što ima liniju, RANSAC će provjeriti koliko je točaka unutar neke definirane udaljenost. Ako pređemo definirani prag koliko točaka mora biti blizu linije, možemo zaključiti da to doista jest linija, tj. u našem slučaju zid. Algoritam pretpostavlja da informacije dobiva u obliku matrice sa dva stupca i n redaka, gdje je n broj mjerenja. Također pretpostavlja da su sve točke u globalnom kartezijevom sustavu. Pošto ću koristiti ovaj algoritam za otkrivanje i izdvajanje prostornih značajki, dalje u tekstu ću dati primjer pseudo-koda, a u idućem poglavlju ću pokazati kako sam to isprogramirao u programskom jeziku „Python“.

- Uzmi nasumičan uzorak iz laserskog skena, uzorak se sastoji od nekoliko bliskih mjerenja
- Pomoću metode najmanjih kvadrata napravi pravac koji najbolje opisuje te točke
- Odredi koliko je točaka je udaljeno manje od X (u ovom radu će taj broj biti 15 mm) od tog pravca
- Gornji uvjet zadovoljava Y (u ovom radu to će biti 4 točke manje nego što se uzima u uzorku) ili više točaka napravi:
    - Izračunaj novi pravac metodom najmanjih kvadrata pomoću točaka koje smo odredili da su dovoljno blizu prvobitnom pravcu
    - Pravac dodaj bazi prostornih značajki

Moja implementacija EKF-a pretpostavlja da su prostorne značajke točke, tako da ćemo morati pravce pretvarati u točke, to ćemo napraviti ovako:



Slika 10. Definicija točke – prostorne značajke

2.5. Povezivanje podataka

Problem povezivanja podataka je pronalazak istih značajki iz dva različita skena prostora. Ovaj problem smo prije zvali i ponovo opažanje značajki. Kao što smo već spomenuli, problemi kod povezivanja podataka su sljedeći:

- Ne opažanje značajke u svakom skenu
- Krivo opažanje značajke koju više nikada nećemo opet opaziti
- Pogrešno povezivanje značajki

Ove prve dvije točke za nas nisu problematične. Ali zato je treća točka veliki problem, jer ukoliko robot pogrešno poveže dvije značajke, misliti će da je na različitom mjestu od onoga na kojem doista jest.

U nastavku teksta ću u pseudokodu objasniti princip rada algoritma za povezivanje podataka. Prva pretpostavka algoritma je da postoji baza podataka već viđenih značajki. Također imamo pravilo da SLAM ne uzim značajku za lokalizaciju ukoliko ta značajka nije već bila viđena N puta. Ovo pravilo otklanja pogrešno opažene značajke.

- Sa novim laserskim skenom prođemo kroz algoritam izdvajanja prostornih značajki
- Novo viđenim značajkama pokušamo pridodati para iz baze podataka već viđenih značajki
- Ukoliko smo značajki iz baze podataka već viđenih značajki pronašli para u novo viđenim značajkama, onda povećavamo broj koliko je puta ta značajka viđena
- Ukoliko novo viđenoj značajki nismo uspjeli pronaći para, onda ju zapisujemo u bazu podataka viđenih značajki i pokušavamo ju ponovo pronaći u idućem skenu.

Ova tehnika se naziva pristupom najbližeg susjeda jer povezuje značajku s najbližom značajkom u bazi podataka. Najjednostavniji način za izračunati udaljenost dvije točke je pomoću Euklidske udaljenosti. Postoje razne druge metode asocijacije značajki, ali pošto će i zbog RANSAC algoritma i zbog okoline u kojoj će se robot nalaziti značajke biti međusobno na velikim udaljenostima, metoda računanja Euklidske udaljenosti će biti zadovoljavajuća.

2.6. EKF

Metoda proširenog Kalmanovog filtra koristi se za procjenu položaja robota na temelju odometrijskih podataka i opažanja prostornih značajki. Općenito je opisan samo u kontekstu procjene položaja (pretpostavlja se potpuna karta okoline). To znači da ne uključuje ažuriranje karte, što je ključno za SLAM. Implementacija EKF-a za SLAM može biti izazovna zbog promjena u matricama.

Kada se izdvoje značajke i podaci o njima, SLAM se može podijeliti u tri koraka:

- Ažuriranje trenutne procjene položaja uz pomoć odometrije
- Ažuriranje procijenjenog položaja na temelju ponovnih opažanja značajki
- Dodavanje novih značajki u trenutno stanje

Prvi korak je prilično jednostavan: pomoću odometrije robota se procjenjuje inicijalni položaja. Na primjer, ako je robot na točki (x, y) s rotacijom theta, a iz odometrije vidimo pomak dx, dy i promjenu u rotaciji dtheta, novi položaj robota bit će (x+dx, y+dy) s rotacijom theta+dtheta.

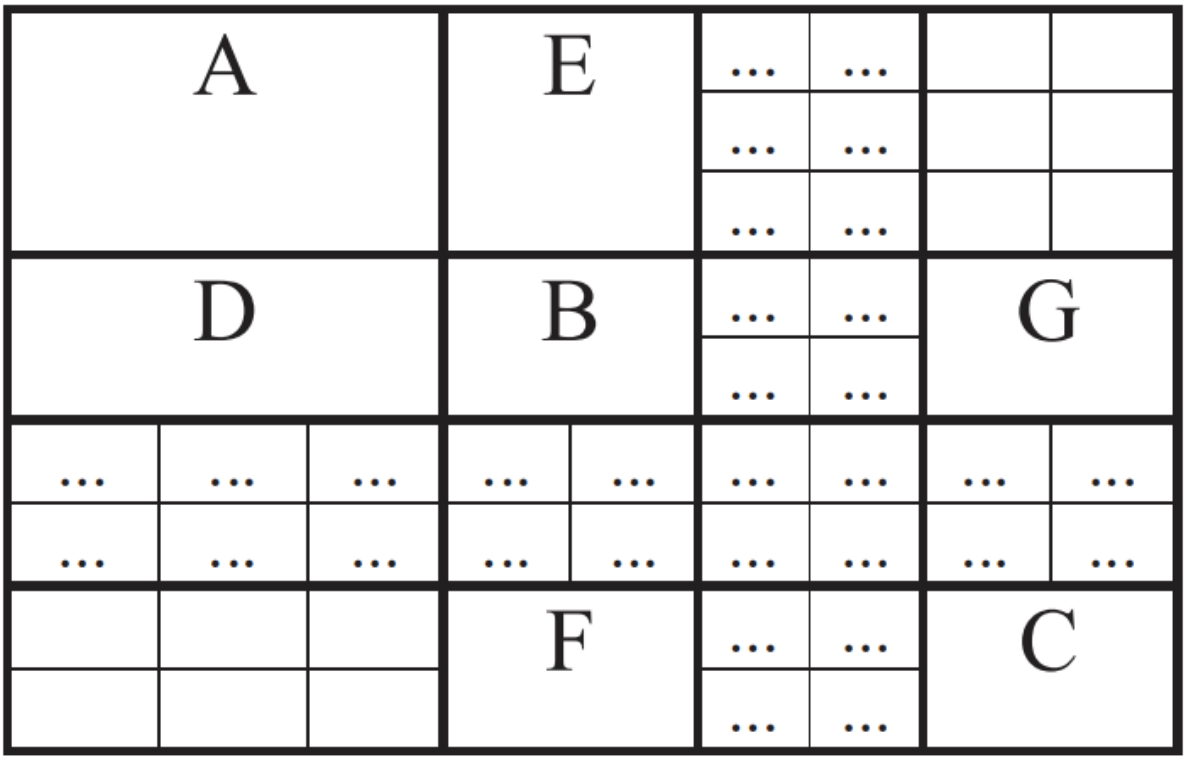
U drugom koraku, razmatraju se ponovno opažene značajke. Procjenjuje se gdje bi trebale biti u odnosu na trenutnu poziciju, no obično postoji razlika, koja se naziva inovacija. To je bitna razlika između procijenjene i stvarne pozicije robota, temeljena na onome što robot može vidjeti. Osim toga, u ovom koraku se ažurira i nesigurnost svake opažene značajke kako bi se odrazile nedavne promjene. Na primjer, ako je nesigurnost trenutne pozicije značajke mala, ponovno opažanje značajke s malom nesigurnošću povećat će sigurnost značajke u odnosu na trenutnu poziciju robota.

U trećem koraku, nove značajke se dodaju u trenutno stanje – kartu globalnog sustava koju iz koraka u korak nadopunjujemo novim informacijama. Karta se sastoji od izmjerenjih točaka i prostornih značajki koje je robot pronašao. Dodavanje značajki u kartu se postiže uz pomoć informacija o trenutnoj poziciji i dodavanjem informacija o odnosu novo viđenih značajki s već postojećim. Dalje u tekstu ću nabrojati vektore i matrice koje ću koristiti u EKF-u. Primjenu slijedećih vektora i matrica, te matematičke jednadžbe za sve što smo do

sada teoretski prošli, ćemo vidjeti i objasniti u poglavlju 4 kada dođemo do implementacije SLAM algoritma.

Vektor stanja sustava: X. Matrica X, ili u ovom našem slučaj vektor, vjerojatno je jedna od najvažnijih matrica u sustavu zajedno s matricom kovarijance. Ona sadrži poziciju robota, x, y i theta. Osim toga, sadrži x i y poziciju svake značajke. Važno je da vektor bude vektor stupac kako bi se osiguralo da sve jednadžbe funkcioniraju. Duljina vektora X je 3+2*n redova, gdje je n broj značajki. Vrijednosti će biti spremljene u milimetrima, dok je kut spremljen u stupnjevima.

Matrica kovarijance: P. Kratko prisjećanje iz matematike, kovarijanca dvije varijable pruža mjeru koliko su te dvije varijable međusobno povezane. Korelacija je koncept koji se koristi za mjerenje stupnja linearne ovisnosti između varijabli. Matrica kovarijance P je središnja matrica u sustavu. Sadrži kovarijancu položaja robota, kovarijancu značajki, kovarijancu između položaja robota i značajki te konačno kovarijancu između raznih značajki.



Slika 11. Matrica P

Slika iznad prikazuje sadržaj matrice kovarijance P. Prva ćelija, označena s A, sadrži kovarijancu položaja robota. To je matrica dimenzija 3x3 (x, y i theta). B označava kovarijancu prve značajke. To je matrica dimenzija 2x2, budući da značajke nemaju orijentaciju theta. Ovo se nastavlja do C, što je kovarijanca za posljednju značajku. Ćelija D sadrži kovarijancu između stanja robota i prve značajke. Ćelija E sadrži kovarijancu između prve značajke i stanja robota. E se može deducirati iz D transponiranjem podmatrice D. F sadrži kovarijancu između posljednje i prve značajke, dok G sadrži kovarijancu između prve i posljednje značajke što se također može deducirati transponiranjem F podmatrice. Dakle, iako matrica kovarijance može izgledati komplicirano, zapravo je vrlo sistematično izgrađena. Na početku, kada robot nije vidio nijednu značajku, matrica kovarijacije P uključuje samo podmatricu A. Kovarijacijska matrica mora biti inicijalizirana korištenjem nekih zadanih vrijednosti za dijagonalu. To odražava nesigurnost u početnom položaju. Ovisno o stvarnoj implementaciji, često će se pojaviti singularna pogreška ako se početna nesigurnost ne uključi u neke od izračuna, stoga je dobra ideja uključiti neku početnu pogrešku iako postoji razlog za vjerovanje da je početni položaj robota točan.

x_r	x_b
y_r	y_b
t_r	t_b
$x_{1,r}$	$x_{1,b}$
$y_{1,r}$	$y_{1,b}$
…	…
…	…
$x_{n,r}$	$x_{n,b}$
$y_{n,r}$	$y_{n,b}$

Slika 12. Kalmanovo pojačanje K

Kalmanovo pojačanje K izračunava se kako bismo odredili koliko ćemo vjerovati opaženim značajkama i koliko želimo da novo opažene značajke utječu na rezultat. Ako primijetimo da se robot treba pomaknuti 10 cm udesno prema opaženim značajkama, koristimo Kalmanovo pojačanje kako bismo utvrdili koliko ćemo zapravo korigirati položaj; to može biti samo 5 cm jer ne vjerujemo potpuno značajkama, već tražimo kompromis između odometrije i korekcije značajki. To postižemo korištenjem nesigurnosti opaženih značajki zajedno s mjerom kvalitete uređaja za skeniranje prostora i performansi odometrije robota. Ako je uređaj za skeniranje prostora stvarno loš u usporedbi s performansama odometrije robota, naravno da mu nećemo puno vjerovati, pa će Kalmanovo pojačanje biti nisko. Nasuprot tome, ako je uređaj za skeniranje prostora vrlo dobar u usporedbi s performansama odometrije robota,

Kalmanovo pojačanje bit će visoko. Matricu možete vidjeti na slici desno. Prvi red prikazuje koliko se treba dobiti iz inovacije za prvi red stanja X. Prvi stupac u prvom retku opisuje koliko treba dobiti iz inovacije u smisli x koordinate, dok drugi stupac u prvom retku opisuje koliko treba dobiti iz inovacije u smislu smjera. Ponovno, oba su za prvi red u stanju, što je x vrijednost položaja robota. Matrica se nastavlja kroz položaj robota; prva tri retka, i značajke; svaka dva nova retka. Veličina matrice je 2 stupca i 3+2*n redaka, gdje je n broj značajki.

Jacobian modela mjerenja: H usko je povezan s modelom mjerenja, naravno, stoga prvo pregledajmo model mjerenja. Model mjerenja definira kako izračunati očekivanu udaljenost i kut pod kojim vidimo značajku u odnosu na prijašnju poziciju robota i upravljački signal. To se radi pomoću sljedeće formule:

$$\begin{bmatrix} udaljenost \\ kut \end{bmatrix} = \begin{bmatrix} \sqrt{(\lambda_x - x)^2 + \left(\lambda_y - y\right)^2} + v_r \\ tan^{-1}\left(\frac{\lambda_y - y}{\lambda_x - x}\right) - \theta + v_\theta \end{bmatrix} \quad \textbf{(1)}$$

Gdje je lambda x x-pozicija značajke, dok je x je trenutno procijenjena x-pozicija robota. Lambda y je y-pozicija značajke, a y je trenutno procijenjena y-pozicija robota. Theta je rotacija robota, te r je izmjerena udaljenost robota od značajke. To će nam dati predviđeno mjerenje udaljenosti i kuta do značajke . Jacobian izraza gore s obzirom na x, y i theta, jest:

$$H = \begin{bmatrix} \frac{x - \lambda_x}{r} & \frac{y - \lambda_y}{r} & 0 \\ \frac{\lambda_y - y}{r^2} & \frac{\lambda_x - x}{r^2} & -1 \end{bmatrix} \quad \textbf{(2)}$$

H nam pokazuje koliko se udaljenost i kut mijenjaju kako se mijenjaju x, y i theta. Prvi element u prvom retku je promjena udaljenosti s obzirom na promjenu x koordinate robota. Drugi element je s obzirom na promjenu y koordinate. Posljednji element je s obzirom na promjenu rotacije robota. Naravno, ova vrijednost je nula jer se udaljenost ne mijenja dok se robot okreće. Drugi red daje iste informacije, samo o kutu pod kojim se značajka nalazi, znači promjenu kuta u ovisnosti o promjeni stanja robota. To je sadržaj uobičajene matrice H za redovitu procjenu stanja EKF-a, ali pošto se u ovom radu razmatra SLAM algoritam, matrica

H se proširuje kako bi mogli procjenjivati poziciju značajki u odnosu na položaj robota. Matrica H se proširuje na dimenziju 2 x 3+2*n, gdje je n broj značajki. Te matrica H izgleda otprilike ovako:

$$H = \begin{bmatrix} A & B & C & 0 & 0 & -A & -B & 0 & 0 \\ D & E & F & 0 & 0 & -D & -E & 0 & 0 \end{bmatrix} \quad \textbf{(3)}$$

Matrica H će sada sadržavati i informaciju kako će se mijenjati udaljenost i kut značajke u odnosu na promjenu x i y koordinate robota. U izrazu iznad prikazan je slučaj kada imamo 3 prostorne značajke, te sada želimo odrediti procijenjenu udaljenost i kut pod kojim ćemo vidjeti 2. prostornu značajku. Prva dva stupca koji sadrže nule, te zadnja dva stupca sa nulama se odnose na prvu i zadnju značajku. Ovi stupci su popunjeni nulama jer nas trenutno zanima samo kut i udaljenost 2. značajke. Obraćam pozornost na brojeve koji su sada upisani na mjestu druge značajke. Parametri se poklapaju sa parametrima Jakobijana za promjenu udaljenosti i kuta za x i y koordinate robota, ali sa negativnim predznakom. Ovo je iterativni postupak te ćemo u svakoj iteraciji koristiti po samo jednu značajku.

Jacobian predikcijskog modela: A. Poput H, Jacobian predikcijskog modela usko je povezan s predikcijskim modelom, naravno, stoga prvo pregledajmo predikcijski model. Predikcijski model definira kako izračunati očekivanu poziciju robota uzetu staru poziciju i upravljački ulaz. To se radi pomoću sljedeće formule, koja je označena s f:

$$f = \begin{bmatrix} x + cos\theta + q * cos\theta \\ y + sin\theta + q * sin\theta \\ \theta + \Delta\theta + q\Delta\theta \end{bmatrix} \quad \textbf{(4)}$$

Gdje su x i y položaj robota, theta rotacija robota, te q koji opisuje nesigurnost ovog modela. Izračuni su isti kao i za matricu H, osim što sada imamo još jedan red koji služi za predikciju rotacije robota. Dakle, možemo jednostavno zapisati:

$$A = \begin{bmatrix} 1 & 0 & -\Delta y * \sin(\theta) \\ 0 & 1 & \Delta x * \cos(\theta) \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5)}$$

Procesni šum: Q i W. Pretpostavlja se da proces ima Gaussov šum proporcionalan upravljačkim ulazima Δx, Δy i Δt. Šum je označen s Q, što je matrica dimenzija 3x3. Obično se izračunava množenjem nekog Gaussovog uzorka C s W i transponiranog W:

$Q = WCW^T$

C predstavlja točnost odometrije. Vrijednost bi trebala biti postavljena prema performansama odometrije robota i obično je najlakše postaviti eksperimentima i podešavanjem vrijednosti.

U većini radova procesni šum označava se samo kao Q ili kao WQW^T. Pojam C zapravo se gotovo nikad ne koristi, ali je potreban ovdje kako bi se prikazala ta dva pristupa. Za primjer ovog rada možemo napisati da je:

$$C = \begin{bmatrix} c\Delta x^2 & 0 & 0 \\ 0 & c\Delta y^2 & 0 \\ 0 & 0 & c\Delta t^2 \end{bmatrix} \quad \textbf{(6)}$$

$$W = [\Delta t cos(\theta) \quad \Delta t sin(\theta) \quad \Delta\theta]^T \quad \textbf{(7)}$$

$$Q = \begin{bmatrix} c\Delta x^2 & c\Delta x\Delta y & c\Delta x\Delta t \\ c\Delta y\Delta x & c\Delta y^2 & c\Delta y\Delta t \\ c\Delta t\Delta x & c\Delta t\Delta y & c\Delta t^2 \end{bmatrix} \quad \textbf{(8)}$$

Mjerna pogreška: R i V. Pretpostavlja se da uređaj za skeniranje također ima Gaussov šum proporcionalan udaljenosti i kutu. Izračunava se kao VRV^T. V je matrica identiteta dimenzija 2x2. R je također matrica dimenzija 2x2 s brojevima samo na dijagonali.

$$VRV^T = \begin{bmatrix} rc & 0 \\ 0 & bd \end{bmatrix} \quad \textbf{(9)}$$

U gornjem lijevom kutu imamo raspon, r, pomnožen s nekim konstantama c i d. Konstante bi trebale predstavljati točnost uređaja za mjerenje. Primjerice, ako varijanca pogreške u rasponu iznosi 1 cm, c bi trebao biti Gaussov s varijancom 0,01. Ako je pogreška u kutu uvijek 1 stupanj, varijabla „bd“ bi trebala biti zamijenjena brojem 1, pretpostavljajući da se koriste stupnjevi za mjerenja. Obično neće imati smisla grešku napraviti da bude

proporcionalna kutu, kao što bi mogli reći da laser griješi u mjerenju udaljenosti 1% od izmjerenje udaljenosti, nego ćemo reći da kut ima fiksnu pogrešku od jednog stupnja.

Jakobijani specifični za SLAM: J_{xr} i J_z. Ova dva Jakobijana se mogu pronaći samo u SLAM algoritmima. Koriste se samo prilikom ažuriranja matrice kovarijacije P kada pridodajemo nove prostorne značajke u sustav. Pošto se dodavanjem prostornih značajki u sustav povećava matrica stanja X, kako bi sve jednadžbe bile uredu moramo povećati i matricu P. Prvi od te dvije je Jakobijan J_{xr} koji je praktički isti Jakobijanu predikcijskog modela A. Jedina razlika je da nam ne treba zadnji red pošto prostorne značajke imaju samo x i y koordinatu, a nemaju usmjerenje kao što robot ima. Ovo je Jakobijan predikcije značajke u odnosu na stanje robota X

$$J_{xr} = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & 1 & \cos(\theta) \end{bmatrix} \quad \textbf{(10)}$$

Jakobijan J_z je također Jakobijan prediktivnog modela za značajke, ali ovaj put je u odnosu na udaljenost i kut, što kao rezultat daje:

$$J_z = \begin{bmatrix} \cos(\theta + \Delta\theta) & -\sin(\theta + \Delta\theta) \\ \sin(\theta + \Delta\theta) & \cos(\theta + \Delta\theta) \end{bmatrix} \quad \textbf{(11)}$$

3. SIMULACIJA

Kako bi ovaj rad i koncept SLAM-a bio dostupan svima, napravio sam malu simulaciju rada robota sa Lidarom, te ću dalje u tekstu objasniti kako radi, te kako se koristi. Sve što ću prikazati sam napisao u programu Visual Studio Code, u programskom jeziku Python. Kroz kod ću se referirati na „retke koda“ koji su naznačeni na lijevoj strani. Sve popratne funkcije koje koristim u kodu će se moći pronaći u samom kodu, jer ih ima dosta, a njihova funkcija je poprilično očita iz samog naziva funkcije.

3.1. Simulacija Lidar-a

Kao prvi korak, moramo napraviti simulaciju Lidar-a. Lidar možemo pojednostaviti te reći da je to uređaj koji nama šalje savršeno formatirane podatke. Prvi stupac će biti stupnjevi pod kojim je napravio mjerenje, a drugi stupac će biti izmjerena udaljenost. Također ćemo reći da su sva mjerena posložena po redoslijedu stupnjeva. Matrica koju dobijemo je uvijek jednakog oblika. Stupnjevi i udaljenosti su naravno podložni pogreškama koje ćemo opisati kao Gaussov šum, tj. pogrešku koja prati Gaussovu standardnu razdiobu. Okolina robota je definirana situacijom koju smo odabrali u prošlom poglavlju, te je okolina opisana kao početne i krajnje točke zidova u okolini. Dalje će biti priložena programska izvedba ovdje opisane simulacije.

```python
def getMeasure(robot_x, robot_y, robot_theta):
    zidovi = tocke_sa_pravcima
    reading = []
    for kut in np.arange(0, 360, 0.1):  # Od 0 do 359.9 s korakom 0.1
        kut = random.gauss(kut, 0.01)   # Gausov šum za kut
        # Parametri pravca koji predstavlja sken
        kut_za_m = 180 + kut + robot_theta
        m = math.tan(math.radians(kut_za_m))
        kvadrant_skena = provjeri_kvadrant_za_kut(kut_za_m)
        b = b_pravca([robot_x, robot_y], m)

        dist_min = 999999   # pocetna vrijednost za dist min kako bi mogli izracunat koji presjek zidova je najblizi robotu

        for zid in zidovi:
            try:
                x_s, y_s = sjeciste_dva_pravca(m, b, zid[4], zid[5])
            except ValueError as e:
                pass

            kvadrant_sjecista = provjeri_kvadrant_za_tocku(x_s, y_s, robot_x, robot_y)

            if kvadrant_skena == kvadrant_sjecista:
                if nalazi_se_izmedju([zid[0], zid[1]], [x_s, y_s], [zid[2], zid[3]]):
                    dist = udaljenost_tocaka([x_s, y_s], [robot_x, robot_y])
                    if dist < dist_min:
                        dist_min = dist
        if dist_min > 999900:       # ako nismo našli nijedno sjeciste, tj. dist_min je ostao ne promijenjen, onda automatski izbacijuemo to mjerenje
            pass
        else:
            dist_min = random.gauss(dist_min, dist_min*0.01)        # dodati Gaussov šum od 1% udaljenosti, prema datasheetu od Slamtec-ovog lidara A2
            reading.append([kut, dist_min])

    return reading
```

Slika 13. Lidar simulacija

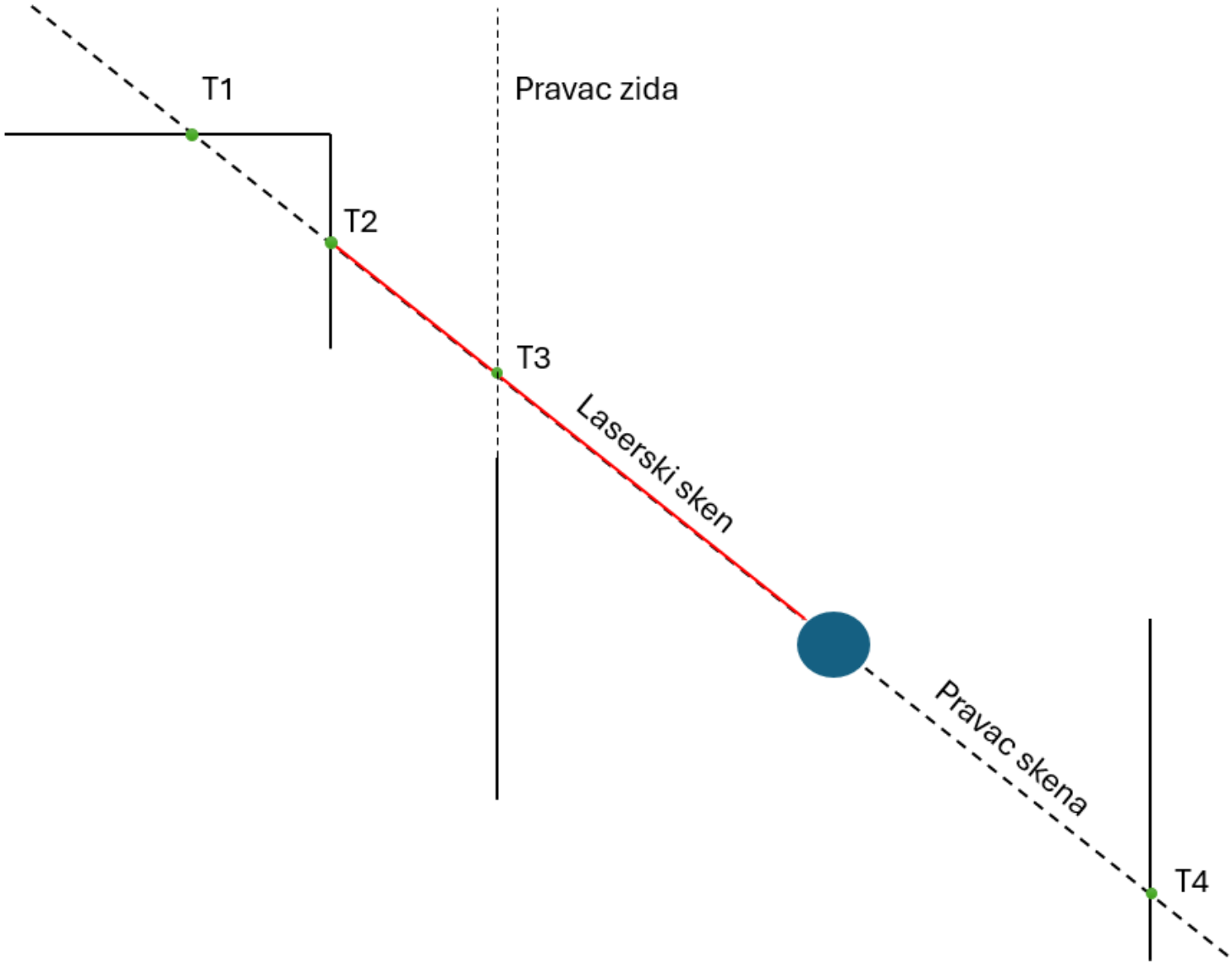
Funkcija započinje u redu 193 gdje definiramo ulazne varijable funkcije, a to će biti x, y, i theta pozicija robota. Nakon toga, red 194, preuzimamo iz varijable „tocke_sa_pravcima“ početne i krajnje točke zidova okoline, skupa s njihovim parametrima pravca, koje sam bio ručno upisao, te ću kasnije pokazati kako izgledaju. Idući red samo inicijaliziramo varijablu „reading“ koja će služiti kao izlaz iz funkcije gdje će biti spremljena čitanja. U redu 196 ulazimo u for petlju u kojoj ću stvoriti kutove od 0 do 359.9 sa korakom 0.1. Ovaj kut ćemo koristiti kut pod kojim Lidar snima okolinu. Pošto smo u for petlji, sada ću za svaki kut odrediti pripadnu udaljenost. Prvo kutu dodajemo Gaussov šum kako bi replicirali ponašanje stvarnog Lidar-a.

Dalje stvaramo kut kojim ćemo onda u redu 200 izračunati parametar m pravca koji predstavlja lasersku zraku jednog skena. Kasnije će biti jasno zašto, ali moramo odrediti u kojem se kvadrantu robota nalazi ta zraka. Te dovršavamo pravac računajući b parametar pravca. Red 204 je samo inicijalizacija, te u redu 206 počinjemo sa otkrivanjem udaljenosti koju će laserska zraka izmjeriti.

Ulazimo u for petlju gdje iz skupa točaka „zidovi“ uzimamo pojedinačni zid, sada je zid vektor redak koji sadrži početnu i krajnju točku toga zida, te parametre pravca toga zida, a izgleda ovako:

$$zid = [x_{z1} \quad y_{z1} \quad x_{z2} \quad y_{z2} \quad m_z \quad b_z]$$

Prvo, u redu 207 probamo pronaći x i y sjecišta pravca laserske zrake i pravca zid-a. Ukoliko dobijemo „ValueError“, preskočimo ovaj dio jer sjecište ne postoji. Nakon toga određujemo u kojem kvadrantu u odnosu na poziciju robota se nalazi sjecište. Te u redu 214, ukoliko se poklapaju kvadrant sjecišta te kvadrant u kojem se nalazi laserska zraka skena, možemo provjeriti dali se sjecište zida i skena nalazi između početne i krajnje točke zida. Ukoliko se točka doista nalazi na pravcu izračunamo udaljenost između sjecišta i robota. Te tu izračunatu udaljenost spremimo kao najmanju, jer moramo provjeriti ima li koji zid koji također zadovoljava sve uvjete ali se nalazi bliže. Slika ispod grafički prikazuje logiku iza ovog algoritma.



Slika 14. Algoritam određivanja udaljenosti

Plavi krug je naš robot. Crveni pravac je laserski sken za koji moramo odrediti udaljenost. Točke 1,2,3,4 su sve točke s kojima smo uspjeli pronaći sjecište. Kao potencijalno rješenje, prvo otpada točka 4, jer prva provjera je dali se laserski sken i točka sjecišta nalaze u istom kvadrantu. Iz slike je očito da se laserski sken nalazi u 2. kvadrantu, dok se točka 4 nalazi u 4. kvadrantu. Nadalje, imamo provjeru dali se sjecište nalazi između dvije točke koje definiraju zid. Tu otpada točka 3, jer kao što možemo vidjeti, imamo sjecište pravac laserskog skena i pravca zida, ali točka sjecišta se ne nalazi između dvije točke koje definiraju zid (deblja crna linija). Točke 1 i 2 su obje zadovoljile sve uvjete i možemo biti sigurni da ih laserski sken sječe. Ali, samo će jedan na kraju ostati, te vidimo da nakon što izračunamo udaljenost robota i točke 1, te udaljenost robota i točke 2, dolazimo do zaključka da je udaljenost između robota

i točke 2 najmanja, što automatski znači, da će naše rješenje biti udaljenost između robota i točke 2.

Ako se vratimo nazad na naš kod u redu 219 vidimo da ukoliko jesmo pronašli neki novi minimum, onda toj dobivenoj udaljenost dodamo Gausov šum, koji je prema dokumentaciji Lidar-a jednak 1% od dobivene udaljenosti, te u matricu Reading dodamo očitanje koje se sastoji od kuta i udaljenosti.

3.2. Simulacija robota

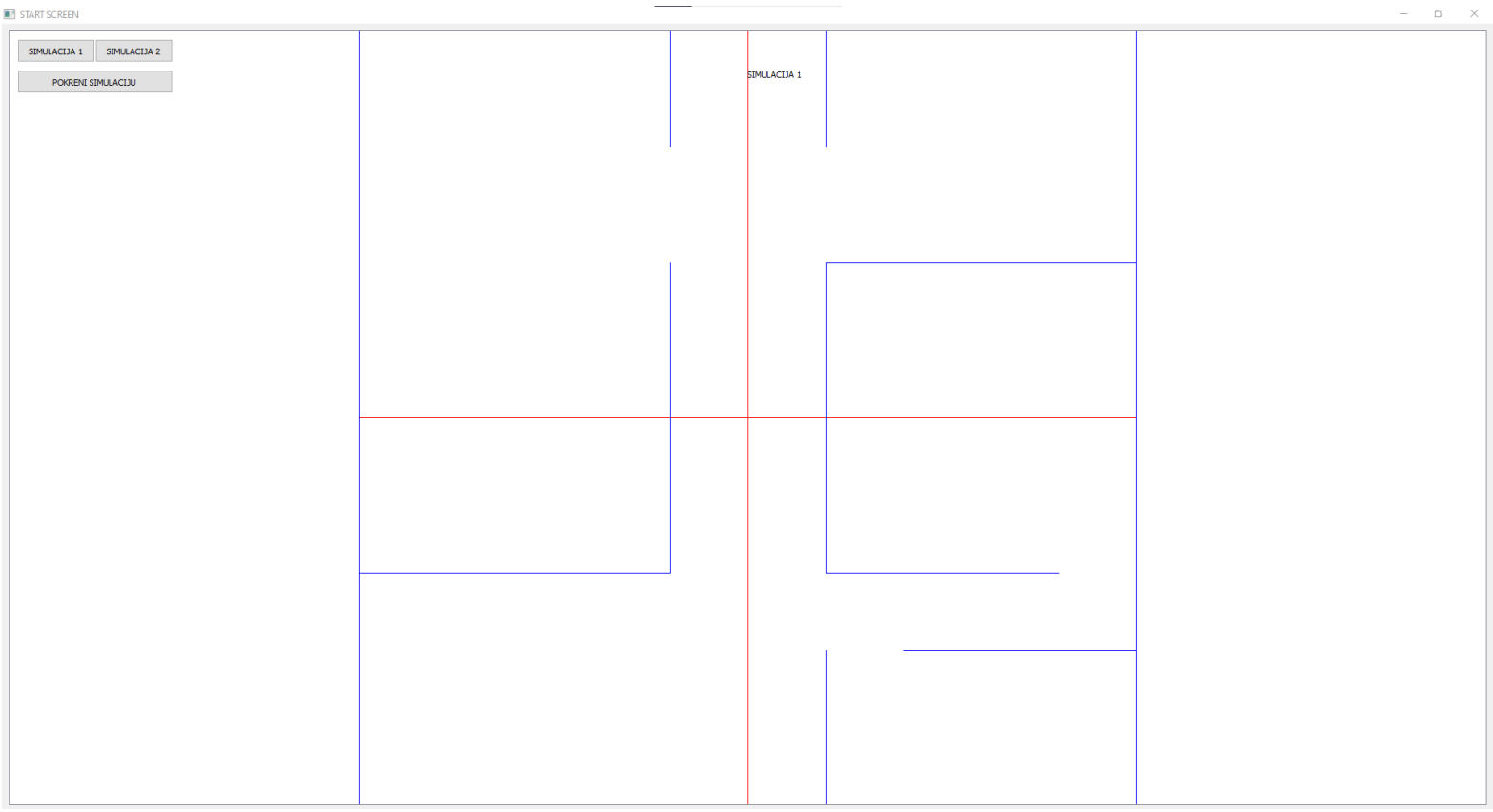
Kao i simulaciju Lidar-a, simulaciju mobilnog robota ćemo svesti na apsolutno najjednostavniji mogući model. Simulaciji predamo informaciju o tome gdje se robot sada nalazi, te instrukciju koja govori što želimo da robot napravi. Instrukcija je jednostavno broj od 1 do 4 gdje 1 označuje naprijed, 2 iza, 3 okret u smjeru kazaljke na satu, te 4 koji označuje okret suprotno od kazaljke na satu. Algoritam simulacije će jednostavno trigonometrijski izračunati gdje bi se robot s tim upravljačkim signalima nalazio u idealnom svijetu. U stvarnom svijetu postoji jako puno faktora koji utječu na pogrešku u pozicioniranju robota, te je gotovo nemoguće sve te faktore uračunati kako bi se dao što vjerniji opis modela kretanja mobilnog robota. Iz tog razloga sam radi jednostavnosti odlučio samo dodati Gaussov šum na izlaznu varijablu. Na idućoj slici se može u programu vidjeti realizacija simulacije.

```
424
425 def giveInst(robot_x_st, robot_y_st, robot_theta_st, naredba):
426     match naredba:
427         case 1:
428             robot_x_nov, robot_y_nov, robot_theta_nov = naprijed([robot_x_st, robot_y_st, robot_theta_st])
429         case 2:
430             robot_x_nov, robot_y_nov, robot_theta_nov = nazad([robot_x_st, robot_y_st, robot_theta_st])
431         case 3:
432             robot_x_nov, robot_y_nov, robot_theta_nov = rotacija_plus([robot_x_st, robot_y_st, robot_theta_st])
433         case 4:
434             robot_x_nov, robot_y_nov, robot_theta_nov = rotacija_minus([robot_x_st, robot_y_st, robot_theta_st])
435     return robot_x_nov, robot_y_nov, robot_theta_nov
436
437 def naprijed(odometrija):
438     robot_x_stari, robot_y_stari, robot_theta = odometrija
439     robot_x_novi = random.gauss(robot_x_stari + 100*math.cos(math.radians(robot_theta)), 10)
440     robot_y_novi = random.gauss(robot_y_stari + 100*math.sin(math.radians(robot_theta)), 10)
441     return [robot_x_novi, robot_y_novi, robot_theta]
442
443 def nazad(odometrija):
444     robot_x_stari, robot_y_stari, robot_theta = odometrija
445     robot_x_novi = random.gauss(robot_x_stari - 100*math.cos(math.radians(robot_theta)), 10)
446     robot_y_novi = random.gauss(robot_y_stari - 100*math.sin(math.radians(robot_theta)), 10)
447     return [robot_x_novi, robot_y_novi, robot_theta]
448
449 def rotacija_plus(odometrija):
450     robot_x_stari, robot_y_stari, robot_theta_stari = odometrija
451     robot_theta_novi = random.gauss(robot_theta_stari - 15, 1)
452     return [robot_x_stari, robot_y_stari, robot_theta_novi]
453
454 def rotacija_minus(odometrija):
455     robot_x_stari, robot_y_stari, robot_theta_stari = odometrija
456     robot_theta_novi = random.gauss(robot_theta_stari + 15, 1)
457     return robot_x_stari, robot_y_stari, robot_theta_novi
458
```

Slika 15. Funkcija giveInst()

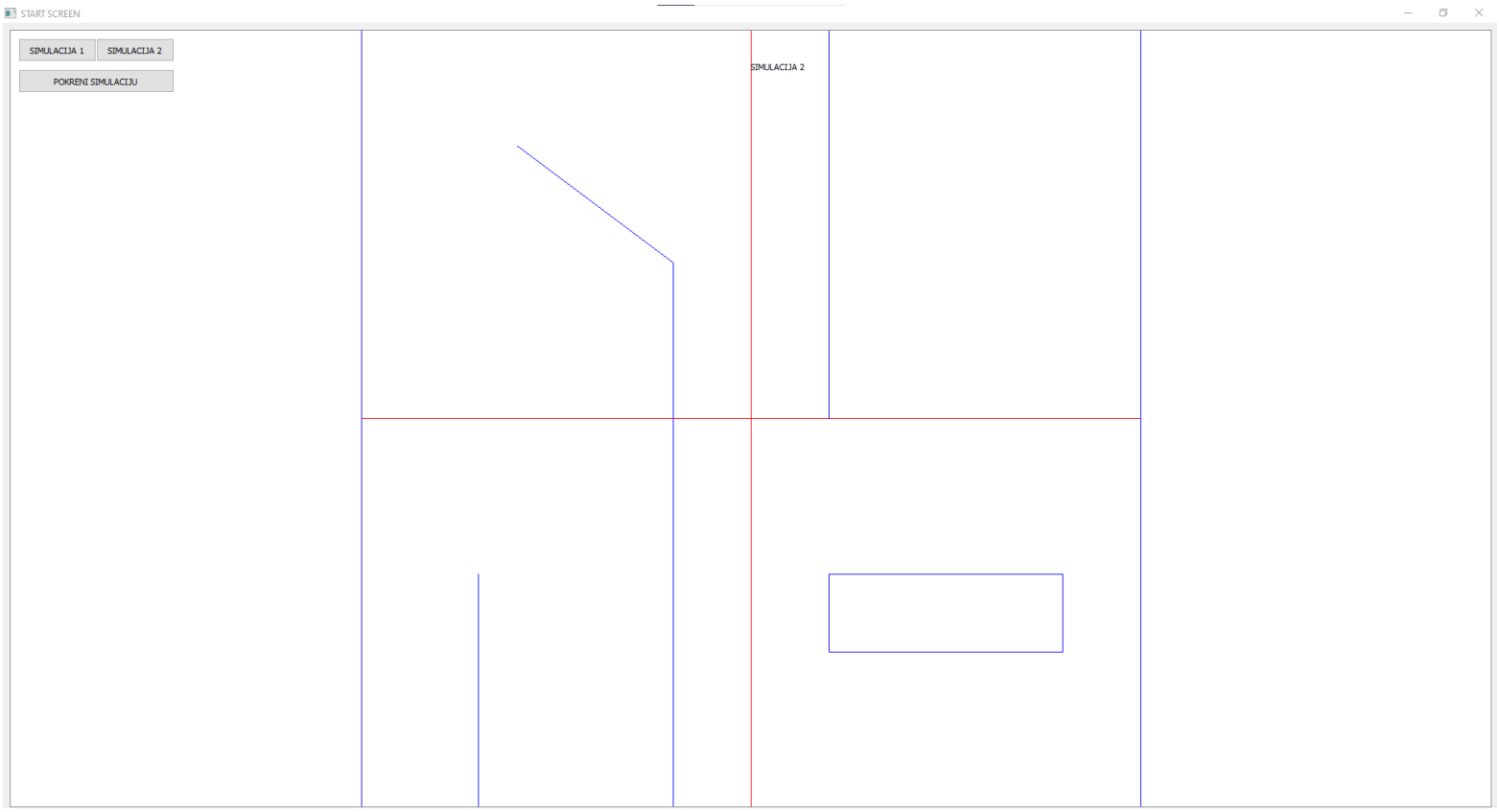
3.3. Pokretanje simulacije

Prilikom pokretanja skripte, napravio sam jednostavno korisničko sučelje na kojem možemo odabrati jednu od dvije situacije koje ćemo simulirati.



Slika 16. Simulacija prve situacije

Na slici iznad prikazano je što se vidi čim se pokrene skripta. U sredini je prikazan slučaj koji će se simulirati. Plave linije su zidovi, crvene linije označuju globalni koordinatni sustav. U gornjem lijevom kutu postoje 3 gumba, dva gumba mijenjaju između simulacije 1 i simulacije 2, dok treći gumb pokreće simulaciju. Iduća slika prikazuje simulaciju 2.



Slika 17. Simulacija druge situacije

Na slici iznad prikazano je što se vidi na računalu nakon stiska gumba „SIMULACIJA 2“, te, ako smo se recimo odlučili na ovu situaciju, pritiskom na gumb „POKRENI SIMULACIJU“ za trenutno odabranu situaciju uzimamo matricu u kojoj su definirane sve početne i krajnje točke zidova u prostoru. Matrica je u obliku:

$$\begin{bmatrix} X_{1z1} & Y_{1z1} & X_{2z1} & Y_{2z1} \\ X_{1z2} & Y_{1z2} & X_{2z2} & Y_{2z2} \\ X_{1z3} & Y_{1z3} & X_{2z3} & Y_{2z3} \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

4. REALIZACIJA SLAM-a

U ovom poglavlju ću pokazati kako sam SLAM proces, koji je bio opisan u drugom poglavlju ovoga rada, pretvorio u programski kod. Ideja je da se napravi algoritam koji bi mogao raditi i sa fizičkim komponentama.

Algoritam koji je u potpunosti ne zavisan o Lidar-u ili nekom drugom mjernom uređaju, te kojem nije važno kako upravljati robotom (fizička izvedba robota ipak ima zavisnosti sa EKF-om jer nema svaki robot isti predikcijski model, tj. ne kreće se svaki robot jednako).

4.1. Funkcije SLAM-a : __init__() i predict()

Na idućoj slici je prikazan početni dio SLAM algoritma, a to je inicijalizacija. Inicijalizacija je potrebna jer mnogo elemenata SLAM-a odmah na početku zahtijevaju neke informacije kako bi mogli zadovoljiti početne matematičke jednadžbe.

Prilikom prvog pozivanja ovog algoritma idemo kroz __init__ funkciju u kojoj stvaramo neke prazne baze podataka, te neke matrice koje ćemo kasnije puniti ili matrice koje označavaju konstante u sustavu.

Nakon inicijalizacije dolazi jedan od dva glavna dijela EKF-a, a to je funkcija predict. Ova funkcija, u odnosu na onu iduću, je relativno vrlo jednostavna, pošto preko odometrije dobivamo Δx, Δy, $\Delta\theta$. Iako, zbog toga kako je napravljen model gibanja robota, odometrija nije u potpunosti točna, već odstupa od prave vrijednosti, kao i u pravom životu. Odometrija je ovdje jako korisna jer će naša predikcija biti da smo stanje X promijenili za odometriju, što možemo i napisati kao:

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} \quad \textbf{(12)}$$

Gdje je x_k, y_k, $theta_k$ x,y pozicija i theta rotacija robota u ovom diskretnom koraku, a x_{k-1}, y_{k-1}, $theta_{k-1}$ su x,y pozicija i theta rotacija robota u prošlom koraku. Posljednji vektor stupac je vektor odometrije, tj. vektor koji pokazuje koliko se robotu promijenila x i y pozicija te

rotacija theta u nekom diskretnom vremnu .Također u ovom koraku, aktualiziramo matricu P sa matricom A koja se zove Jakobijan predikcijskog modela, koja se računa ovako:

$$A = \begin{bmatrix} 1 & 0 & -\Delta y \\ 0 & 1 & \Delta x \\ 0 & 0 & 1 \end{bmatrix}$$

```
248 class SLAM():
249     def __init__(self):
250         print("INIT SLAM-a")
251         self.LandmarkDB = []
252         self.landmarkNum = 0
253         # Inicijalizacija EKF-a
254         self.X = np.array([[0],[0],[0]])  # Vektor stanja sustava
255         self.X = self.X.astype(float)
256         self.P = np.diag([0, 0, 0])  # Matrica kovarijance
257         self.Q = np.diag([[0.01],[0.01],[0.01]])  # Procesni šum
258         self.R = np.diag([0.1, np.radians(1.0)])**2
259         self.H = np.array([[1.0,  0,   0],
260                            [  0,1.0,   0],
261                            [  0,  0, 1.0]])
262         self.innovations = []
263 
264     def predict(self, odometrija):
265         # Jacobian predikcijskog modela
266         delta_x = odometrija[0][0]
267         delta_y = odometrija[1][0]
268         delta_theta = odometrija[2][0]
269         A = np.array([[1, 0, -delta_y],
270                       [0, 1, delta_x],
271                       [0, 0, 1]])
272         self.A = A
273         # Predikcija
274         self.X[:3] = self.X[:3] + np.array(odometrija)
275         self.P[:3, :3] = A.dot(self.P[:3, :3]).dot(A.T) + self.Q
276         self.P[:3] = A.dot(self.P[:3])
```

Slika 18. Funkcije SLAM-a, inicijalizacija i predict

4.2. Funkcija SLAM-a : update()

```
277     def update(self, measurement, landmarks):
278         num_landmarks = len(landmarks)
279 
280         if num_landmarks == 0:
281             # Nema pronađenih značajki, ne treba ažurirati
282             return
283 
284         # Izračun inovacija, Jacobijana i mjernih matrica za svaki landmark
285         for i in range(num_landmarks):
286             if landmarks[i].totalTimesObserved > 5:
287                 self.H = []
288                 landmark_x, landmark_y = landmarks[i].pos
289                 real_range, real_bearing = measurement[i]
290                 robot_x = self.X[0][0]
291                 robot_y = self.X[1][0]
292                 predicted_range = np.sqrt((landmark_x - robot_x)**2 + (landmark_y - robot_y)**2)
293                 predicted_bearing = np.arctan2(landmark_y - robot_y, landmark_x - robot_x) - self.X[2]
294 
295                 a = (robot_x - landmark_x)/real_range
296                 b = (robot_y - landmark_y)/real_range
297                 c = np.array([0])
298                 d = (landmark_y - robot_y)/real_range**2
299                 e = (landmark_x - robot_x)/real_range**2
300                 f = np.array([-1])
301 
302                 gornji_H = [a, b, c]
303                 donji_H = [d, e, f]
304                 for j in range(num_landmarks):
305                     if j == i:
306                         gornji_H.append(-a)
307                         gornji_H.append(-b)
308                         donji_H.append(-d)
309                         donji_H.append(-e)
310                     else:
311                         gornji_H.append(0)
312                         gornji_H.append(0)
313                         donji_H.append(0)
314                         donji_H.append(0)
315 
316                 self.H = np.array(self.H)
```

Slika 19. Funkcija SLAM-a, update

U idućem koraku ću pokazati drugu funkciju EKF-a u kojoj ćemo naš početni „pokušaj“ da pogodimo točno stanje X, nadograditi znanjem o prostornim značajkama. Ideja postupka je iduća: za svaku značajku, prvo provjeri dali je značajka viđena dovoljno puta da ju uopće smijemo koristiti, nakon toga napravi predikciju na kojoj udaljenosti i pod kojim kutom bi

trebali moći vidjeti tu značajku. Onda nadopuni matricu H, tj. Jakobijan modela mjerenja sa trenutno aktivnom značajkom. Te posljednje ažuriraj procijenjenu poziciju robota. Iduće slijedi opis programskog koda, te matrične jednadžbe EKF-a.

U redu 277 vidimo da smo ušli u funkciju, te imamo neke ulaze measurement, landmarks. Ulaz landmarks su prostorne značajke koje smo uspjeli upariti, što i budem kasnije pokazao. Dok je ulaz measurements isto matrica koja sadrži udaljenosti i kutove prostornih značajki u odnos na poziciju robota.

U redu 285 vidimo da ulazimo u for petlju, te da želimo odraditi for petlju onoliko puta koliko imamo uparenih prostornih značajki. Odmah prvi uvjet u for petlji je taj da smo značajku, s kojom ćemo sada raditi, vidjeli barem 5 puta. U redu 292 i 293 izračunamo što bi očekivali, gdje se nalazi prostorna značajka, ali vidjeti ćemo to neće biti tako. Sada od reda 295 pa na dalje ide slaganje matrice H. Pošto su prva 3 stupca uvijek ista, njih složimo, te uđemo u for petlju, koja će razvrtiti sve značajke kako bi napunili sve sa nulama, osim na onoj poziciji na kojoj se i trenutno nalazimo sa varijablom „i“. Kada naiđemo na to mjesto tamo upišemo naše parametre i nastavimo dalje dok ne napunimo cijelu matricu, te na kraju, u redu 316 spojimo ta dva reda u jednu matricu.

```
316             self.H = np.array(self.H)
317
318             self.S = self.H.dot(self.P).dot(self.H.T)
319             K = self.P.dot(self.H.T).dot(np.linalg.inv(self.S))
320             innovation = [[real_range - predicted_range], [real_bearing - predicted_bearing]]
321             self.X += K.dot(innovation)
322
```

Slika 20. Ažuriranje vektora stanja X

Na slici iznad možemo vidjeti posljednji dio funkcije update u kojoj se i nalazi sve važno. Prvo, u redu 318, radi jednostavnosti izračunamo pomoću matricu S, koja je također i poznata kao kovarijanca inovacije, koja ima formulu:

$$S = H * P * H^T + V * R * V^T \quad \textbf{(13)}$$

Gdje je S kovarijacija matrice, H je Jakobijan modela mjerenja, P je matrica kovarijacije, VRVT je mjerna greška. Kada smo odredili S, preostaje samo odrediti matricu K, koju računamo prema formuli:

$$K = P * H^T * S^{-1} \quad \textbf{(14)}$$

I vektor inovacije, koji sadrži informacije o koliko smo promašili prilikom pomaka robota i novog skena robota.

$$innovation = \begin{bmatrix} d_s - d_p \\ b_s - b_p \end{bmatrix}$$

Gdje je d_s i b_s udaljenost i kut iz skena, a d_p i b_p udaljenost i kut koju smo pokušali pogoditi. Na kraju nam jedino preostaje, aktualizirati X koji je dot množenje matrice K i vektora innovation:

$$X = K * innovation \quad \textbf{(15)}$$

Gdje je X vektor stanja sustava i K je Kalmanova matrica pojačanja. Ovime smo završili glavne funkcije EKF-a, sad jedino preostaje da pokažem kako ih pozvati, ali prije toga da pokažem kako uopće dođem do uzorka u mjerenju.

4.3. SLAM proces i RANSAC

Kako bi iskoristili nabrojane funkcije, moramo imati i neke popratne algoritme za npr. detekciju značajki, te povezivanje podataka. Već sam i pisao o tome, ali jednom opet, koristiti ćemo RANSAC algoritam za otkrivanje značajki. Jako detaljno sam ga objasnio u drugom poglavlju pa ukoliko ima nejasnoća vjerojatno će svi odgovori biti tamo. Na idućoj slici pokazana je programska implementacija toga algoritma. U ovom prvom dijelu ovoga glavnoga koda je bitno primijetiti kako uzimamo uzorak za RANSAC algoritam prepoznavanja prostornih značajki. To radimo random.sample metodom koja se nalazi u

```
333     def SLAM_proces(self, reading, robot_x, robot_y, robot_theta, odometry):
334
335         self.newLandmarkDB = []
336         sample = random.sample(reading, RANSAC_SAMPLE)   # uzmi random sample
337
338         angles, distances = zip(*reading)
339
340         x_global = [robot_x + distance * math.cos(math.radians(angle + robot_theta)) for angle, distance in reading]
341         y_global = [robot_y + distance * math.sin(math.radians(angle + robot_theta)) for angle, distance in reading]
342
343         global_tocke = [[xi, yi] for xi, yi in zip(x_global, y_global)]
344
345
346         angles_sample = [row[0] for row in sample]  # Izdvajamo stupnjeve iz sample
347
348         for stupanj in angles_sample:
349             index_random_mjerenja = angles.index(stupanj)  # Pronalazimo indeks stupnja u originalnom popisu
350             distance_sample = distances[index_random_mjerenja]
351             indexi_oko_landmarka = range(index_random_mjerenja - SAMPLE_OFFSET, index_random_mjerenja + SAMPLE_OFFSET)  # Indeksi oko odabranog stupnja
352             korektirani_indexi = [(index-len(reading)) if index >= len(reading) else index for index in indexi_oko_landmarka ]
353
354             sample_tocke_global = [global_tocke[i] for i in korektirani_indexi]
355
356             a_temp, b_temp = parametri_pravca(sample_tocke_global)
```

Slika 21. SLAM, uzimanje uzorka prostora

pythonovom libraryu random. Ova metoda uzima pod prvi parametar sve točke koje smo dobili od Lidar-a, tj. u našem slučaju od simulacije, te uzima onoliko uzoraka koliko je to definirano konstantom RANSAC_SAMPLE. Nakon toga imamo: rastavljanje matrice reading na kutove i udaljenosti, pretvaranje tih kutova i udaljenosti u globalne koordinate, te stvaranje matrice globalnih točaka. Nakon toga, u redu 348 ulazimo u for petlju gdje iteriramo kroz svaki uzorak koji smo nasumično dobili. Kao što sam i prije opisao ovako radi RANSAC algoritam. Uzmemo nekoliko mjerenja oko tog mjerenja koje nam služi kao uzorak, te pomoću funkcije parametri_pravca() dobijemo a i b parametar pravca koji je dobiven metodom najmanjih kvadrata za taj skup točaka. Imamo i liniju za korektirane indexe, jer ako dobijemo uzorak od ,npr. 359.8 stupnjeva, onda bi nam točke uz taj uzorak prešle u 360.1 što ne postoji, nego se moramo vraćati nazad na početak mjerenja.

Dalje imamo odluku dali se te točke nalaze na jednom pravcu ili ne. U redu 360 pomoću funkcije udaljenost_od_pravca() računamo udaljenost između pravca kojeg smo dobili metodom najmanjih kvadrata i točke koja se možda nalazi na tom pravcu. Potvrđujemo dali se nalazi na pravcu u redu 361 gdje gledamo ako je udaljenost manja od konstante LINETHRESHOLD onda možemo sa sigurnošću reći da je to točka na pravcu, te ju dodajemo u vektor „tocke_na_pravcu“. Kada smo tako prošli kroz sve točke provjeravamo ima li dovoljno točaka da bi to stvarno smatrali pravcem. Pošto vjerojatno nisu sve točke iz uzorka, također i na pravcu, a originalni pravac je bio napravljen pomoću točaka iz uzorka, moramo

napraviti novu liniju pomoću samo točaka koje doista jest na pravcu, kako bi naš pravac bio precizniji. Tako da u redu 365 ponovo računamo parametre a i b za pravac. Pošto nam EKF prima prostorne značajke kao točke, pretvoriti ću značajku pravac u značajku točku, kako je i prikazano na []. Prvo računamo ortogonalni pravac koji prolazi kroz središte globalnog koordinatnog sustava. Te tražimo sjecište tog pravca i pravca prostorne značajke. Tada možemo reći da smo pronašli novu značajku te ju dodajemo u bazu podataka „newLandmarkDB“.

```
358         tocke_na_pravcu = []
359         for i in range(len(sample_tocke_global)):
360             udaljenost = udaljenost_od_pravca(sample_tocke_global[i], a_temp, b_temp)
361             if udaljenost < LINETHRESHOLD:              # provjera dali je tocka predaleko od pravca, ako je dovoljno blizu zapisuje se u rezultantni array
362                 tocke_na_pravcu.append(sample_tocke_global[i])
363 
364         if(len(tocke_na_pravcu) >= LINE_CONFIRM):       # ako ima dovoljno tocaka dovoljno blizu pravca, onda se stvara novi pravac
365             a, b = parametri_pravca(tocke_na_pravcu)
366 
367             a_ortog = -1/a                              # pretvaranje landmark pravca u landmark tocku, posto nam EKF koristi tocke a ne pravce
368             b_ortog = 0
369             x_presjek = (b_ortog - b) / (a - a_ortog)   # racunanje x_s = (b_o - b)/(a- a_o)
370             y_presjek = a * x_presjek + b               # i ovdje dobimo landmark u obliku [x,y], s time da znamo i iz kojeg je kuta i udaljenosti proizasa
371 
372 
373             noviLandmark = NewLandmark([x_presjek, y_presjek], stupanj, distance_sample, a, b)
374             self.newLandmarkDB.append(noviLandmark)
375 
```

Slika 22. SLAM, RANSAC Algoritam

4.4. Povezivanje podataka

Kako sam već napomenuo, morati ćemo imati i povezivanje podataka kako bi probali spojiti postojeće i novo viđene značajke. Ovo radimo tako da izračunamo udaljenost između novo viđene značajke i svake značajke koju smo već bili vidjeli. Ukoliko smo našli neku značajku koja je blizu nove značajke, možemo reći da je to ista ta značajka. Ako nismo pronašli novo viđenu značajku među postojećima ne radimo ništa, nego ćemo tek kasnije nju razriješiti. Ukoliko smo našli značajku koju možemo upariti, ali postojeća značajka je već uparena, jednostavno brišemo novu značajku jer znači da smo dva puta pronašli istu značajku. Slijedi opis programa povezivanja podataka.

U liniji 381 počinjemo sa algoritmom povezivanja podataka. Ukoliko smo našli novu značajku u ovom trenutnom skenu, ulazimo u for petlju, te uzimamo svaku već postojeću prostornu značajku, i računamo Euklidsku udaljenost između nove značajke i postojeće značajke. Ukoliko je ta udaljenost manja od konstante LANDMARKOFFSET, tj. ukoliko

možemo reći da je to ta ista značajka, te ta već postojeća značajka nije uparena u ovom krugu, onda: kažemo da smo uparili i novu i staru značajku, kažemo da smo značajku vidjeli još jedan put, te ukoliko je značajka izgubila koji život (jer ju nismo bili pronašli prošli krug) vraćamo joj sve živote. Uparenu staru značajku stavljamo u vektor zvan „upareniLandmark“. Odredimo pod kojim kutom i na kojoj udaljenosti vidimo značajku. Ukoliko je stara značajka već bila uparena, što znači da smo u istom laserskom skenu dva puta pronašli istu značajku, uklanjamo tu značajku iz vektora novo pronađenih značajki.

```
        if(len(self.newLandmarkDB) == 0):
            print('Nije pronađen ni jedan landmark')
        else:
            print(f'Pronađen/o je {len(self.newLandmarkDB)} Landmarkova')
        uparenLandmark = []
        for land in self.LandmarkDB:
            land.uparen = False
        novi_measurement = []
        for new_landmark in self.newLandmarkDB:
            for landmark in self.LandmarkDB:
                udaljenost = ((new_landmark.pos[0] - landmark.pos[0])**2 + (new_landmark.pos[1] - landmark.pos[1])**2)**0.5     # formula za 2D udaljenost [(x1
                if (udaljenost < LANDMARKOFFSET):
                    if (landmark.uparen == False):
                        new_landmark.uparen = True
                        landmark.uparen = True
                        landmark.totalTimesObserved += 1    # ako smo našli postojeći Landmark koji je jako blizu ovom novom landmarku
                        landmark.life = STARTLIFE           # onda tom postojecem landmarku podizemo vrijednost totalTimesObserved i vracamo life na pocetnu vr
                        uparenLandmark.append(landmark)
                        novi_range, novi_bearing = range_n_bearing(new_landmark.pos, self.X[0], self.X[1], self.X[2])
                        novi_measurement.append([novi_range, novi_bearing])
                        print("OVO JE DEBUG####################################")
                    else:
                        self.newLandmarkDB.remove(new_landmark)
                        break
```

Slika 23. SLAM, povezivanje podataka

```
403
404     self.uspareni_landmarkovi = uparenLandmark
405     predicted_state = self.predict(odometry)
406     updated_ekf = self.update(novi_measurement, uparenLandmark)
407
408     # racunanje popratnih matrica
409     Jxr = np.array([[1, 0, math.sin(math.radians(self.X[2]))], [0, 1, math.cos(math.radians(self.X[2]))]])
410     Jz = np.array([[math.cos(math.radians(self.X[2]+odometry[2])), -math.sin(math.radians(self.X[2]+odometry[2]))], [math.sin(math.radians(self.X[2]+odometry[2
411
412     for novi_landmark in self.newLandmarkDB:
413         # Određujemo dimenzije nove matrice
414         existing_rows, existing_cols = self.P.shape
415         desired_rows = existing_rows + 2  # dodajemo 2 retka
416         desired_cols = existing_cols + 2  # dodajemo 2 stupca
417         for stari_landmark in self.LandmarkDB:
418             udaljenost = ((novi_landmark.pos[0] - stari_landmark.pos[0])**2 + (novi_landmark.pos[1] - stari_landmark.pos[1])**2)**0.5
419             if udaljenost < LANDMARKOFFSET:
420                 novi_landmark.uparen = True
421         if novi_landmark.uparen == False:
422             print("povecavamo P!!!!!!!!!!!!1")
423             self.LandmarkDB.append(Landmark(self.landmarkNum, novi_landmark.pos, novi_landmark.angle, novi_landmark.distance, novi_landmark.a, novi_landmark.b)
424
425             self.X = np.vstack((self.X, novi_landmark.pos[0], novi_landmark.pos[1]))
426             prosireni_P = np.zeros((desired_rows, desired_cols))
427             prosireni_P[existing_rows:, existing_cols:] = Jxr.dot(self.P[0:3, 0:3]).dot(Jxr.T) + Jz.dot(self.R).dot(Jz.T)
428             prosireni_P[0:existing_rows, 0:existing_cols] = self.P
429
430             prosireni_P[0:3, existing_cols:] = self.P[0:3, 0:3].dot(Jxr.T)
431             prosireni_P[existing_rows:, 0:3] = prosireni_P[0:3, existing_cols].T
432             for z in range(self.landmarkNum):
433                 temp = self.P[0:3, z*2+3:z*2+5]
434                 prosireni_P[existing_rows:, z*2+3:z*2+5] = Jxr.dot(temp)
435             self.P = prosireni_P
436             self.landmarkNum += 1
```

Slika 24. SLAM, EKF

Nakon što smo uspjeli povezati podatke, idemo nadalje sa EKF-om. Dvije važne funkcije EKF-a smo već vidjeli i razjasnili. Preostalo nam je vidjeti što slijedi nakon aktualiziranja vektora stanja X. Prvo moramo izračunati matrice specifične za SLAM algoritam, koje smo spomenuli u 2. poglavlju, kako bi mogli proširiti matricu kovarijacije P za sve nove prostorne značajke. Također proširenjem matrice P, moramo proširiti vektor stanja X i dodati značajku u bazu podataka. Slijedi objašnjenje programskog koda.

Na slici iznad vidimo prve pozive EKF-a. Prvo pozivamo funkciju predict sa ulazom odometry koji smo dobili od modela gibanja robota. Funkcija predict, aktualizira vektor stanja X. Nakon toga pozivamo funkciju update koja također aktualizira vektor stanja X, ali ovoga puta uz pomoć mjerenja Lidar-a. Dalje, također nam trebaju matrice Jxr i Jz koje su specifične za SLAM algoritam, koje ćemo koristiti za proširenje matrice P.

Nakon toga provjeravamo dali novo viđena značaka postoji u bazi podataka prostornih značajki. Ukoliko ne postoji, onda ju moramo dodati u bazu podataka. Ali ovo nije samo u bazu podataka, već značajku moramo dodati u: vektor stanja X, te moramo proširiti matricu P. Dodavanje u bazu podataka je jednostavno i može se vidjeti u redu 423. Dalje, dodavanje u

vektor stanja X, uzmemo X, tj. vektor stupac te mu na dno stavimo x, pa y koordinatnu prostorne značajke. Proširenje matrice P je već bilo opisano ali u redu 426 možemo vidjeti da prvo dodamo 2 reda i 2 stupca nula. Onda u donji desni kut matrice upisujemo:

$$P^{N+1,N+1} = J_{xr} P^{rr} J_{xr}^T + J_z R J_z^T \quad \textbf{(16)}$$

Gdje je $P^{N+1,N+1}$ pod matrica u donjem desnom kutu matrice kovarijacije P, J_{xr} Jakobijan predikcije značajke u odnosu na stanje robota X, P^{rr} je 3x3 podmatrica u gornjem lijevom kutu matrice P, te $J_z R j_z^T$ je izraz koji predstavlja doprinos mjerenja Lidar-a za novo otkrivenu značajku u matrici kovarijacije. Nadalje, u gornji desni kut upisujemo matricu:

$$P^{r,N+1} = P^{rr} J_{xr}^T \quad \textbf{(17)}$$

A u donji lijevi kut, upisujemo tu istu matricu, ali transponiranu. I jedino što nam preostaje u 2x2 matrice koje opisuju kovarijancu između svake prostorne značajke. Prvo punimo najdonji red formulom:

$$P^{N+1,i} = J_{xr} P^{ri^T} \quad \textbf{(18)}$$

Dok skroz desni stupac puno istim tim matricama, ali transponiranim. U redu 435 proširenu P matricu upisujemo u stvarnu P matricu, te broj značajki u sustavu povećavamo za jedan.

```
438     # Krajnje stanje EKF-a nakon predikcije i ažuriranja
439     final_state = self.X
440     print("Krajnje stanje robota:", final_state)
441
442     for landmark in self.LandmarkDB:                        # smanjivanje života neuparenim landmarkovima
443         uparen = False
444         for uparen_landmark in uparenLandmark:
445             if landmark == uparen_landmark:
446                 uparen = True
447                 break
448         if not uparen:
449             landmark.life -= 1
450             if landmark.life == 0:                          # ako je ponestalo života, uništi landmark
451                 print("UNISTAVAMO LANDMARK ##############################")
452                 index = np.where(landmark == self.LandmarkDB)
453                 print(index)
454                 self.X = self.X[:-2]
455                 self.LandmarkDB.remove(landmark)
456                 self.landmarkNum -= 1
457
458                 print("SADA IH IMA: ", self.landmarkNum)
459                 self.P = self.P[:-2, :-2]
460                 print("TE JE SADA X DUG", len(self.X))
461
462     print("Trenutno imamo: ", len(self.LandmarkDB), "Landmarkova u LandmarkDB-u")
463     return global_tocke
464
```

Slika 25. Uklanjanje loših značajki

I na kraju ovoga procesa nam je još samo ostalo oduzeti i ukloniti loše značajke. Značajke ćemo uklanjati kada im ponestane života. Živote ćemo im smanjivati svaki put kada ih ne uspijemo pronaći u skenu prostora. Ukoliko u potpunosti uklonimo značajku moramo skratiti matricu P i vektor X za tu značajku koju smo uništili.

Na slici gore možemo vidjeti da ukoliko nam značajka u bazi podataka LandmarkDB nije uparena, znači da je nismo pronašli u najnovijem skenu, što znači da joj moramo oduzeti jedan život, te ukoliko ju je to oduzimanje smanjilo na 0 života, značajku uklanjamo iz baze podataka, uklanjamo je iz vektora stanja X, te moramo smanjiti matricu P. Te posljednja linija, SLAM algoritam vraća točke u globalnom koordinatnom sustavu.

4.5. Algoritam crtanja točaka

Iduće što nam sada dolazi je prikazivanje tih točaka na zaslonu. Kako bi se unaprijedile performanse programa, te kada bi bilo puno više točaka, spriječilo rušenje programa, moramo vidjeti koje točke su već nacrtane kako ne bi crtali duple točke.

```
691     def update(self):
692         reading = getMeasure(self.robot_x, self.robot_y, self.robot_theta)
693         nove_glob_tocke = np.array(self.proces.SLAM_proces(reading, self.robot_x, self.robot_y, self.robot_theta, self.odometrija))
694         self.glob = provjeri_postojanje(self.glob, nove_glob_tocke)
695         self.robot_x, self.robot_y, self.robot_theta, self.odometrija = giveInst(self.robot_x, self.robot_y, self.robot_theta, 1)
696         self.draw(self.glob)
697
```

Slika 26. Ciklični program

Na slici iznad je prikazan dio koda koji se izvršava u određenom vremenskom intervalu. Prvo od Lidar-a preuzimamo sken prostora, te nakon toga zovemo SLAM proces. Sada moramo provjeriti koje nove točke su doista nove, a koje su već postojale, a to ću napraviti funkcijom provjeri_postojanje().

```python
def provjeri_postojanje(glob, nove_glob_tocke):
    # Izračunavanje kvadrata razlike između koordinata x i y
    squared_diff = (glob[:, np.newaxis, :] - nove_glob_tocke[np.newaxis, :, :]) ** 2

    # Zbroj kvadrata razlika po koordinatama x i y
    sum_squared_diff = np.sum(squared_diff, axis=2)

    # Izračunavanje Euklidske udaljenosti
    distances = np.sqrt(sum_squared_diff)
    distances = distances.T
    nova_tocka = distances > 60
    i = 0
    for tocka in nova_tocka:
        if np.all(tocka):
            glob = np.append(glob, [nove_glob_tocke[i]], axis=0)
        i += 1
    return glob

```

Slika 27. Funkcija provjeri_postojanje

U prva tri reda koda računamo udaljenosti između svih postojećih i svih novih značajki. Matrica „nova_tocka“ je logička matrica koja nam pokazuje gdje se nalaze vrijednosti udaljenosti veće od 60 mm. Kada to znamo, jednostavno na kraj glob matrice dodajemo novo nađene točke. I na samom kraju te nove točke moramo nacrtati.

4.6. Algoritam crtanja

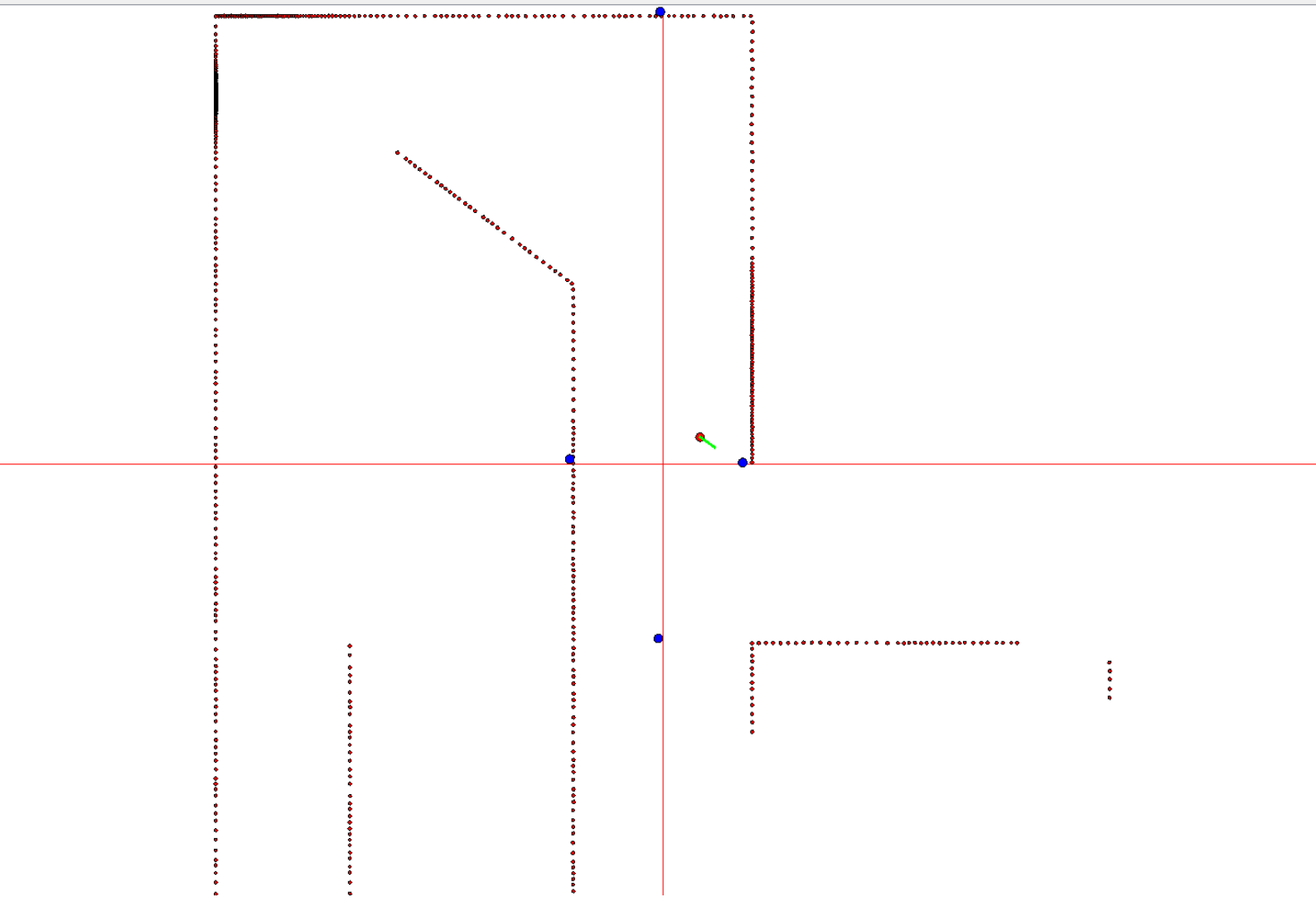
Na slici ispod je prikazan ukratko algoritam grafičkog crtanja na ekranu. Prvo počistimo ekran od svih elemenata, onda idemo u crtanje središnjih linija koje predstavljaju globalni koordinatni sustav. Nadalje ide najzahtjevnije crtanje, što se tiče računalnog napora, a to je crtanje točki iz Lidar-a. Nakon toga ulazimo u crtanje robota. Robota ćemo predstaviti kao krug, te će se njegovo usmjerenje prikazivati jednom kratkom crtom. I posljednje, crtanje točki značajki. Algoritam crtanja značajki također ima i mogućnost crtanja pravaca, ali se prikaz zapuni puni crtama pa je teško vidjeti išta drugo.

```
768     def draw(self, reading):
769         self.clear_screen()
770         self.draw_center_lines()
771         self.draw_lidar_points(reading)
772         self.draw_robot()
773         self.draw_landmarks()
```

Slika 28. Algoritam grafičkog crtanja

5. REZULTATI

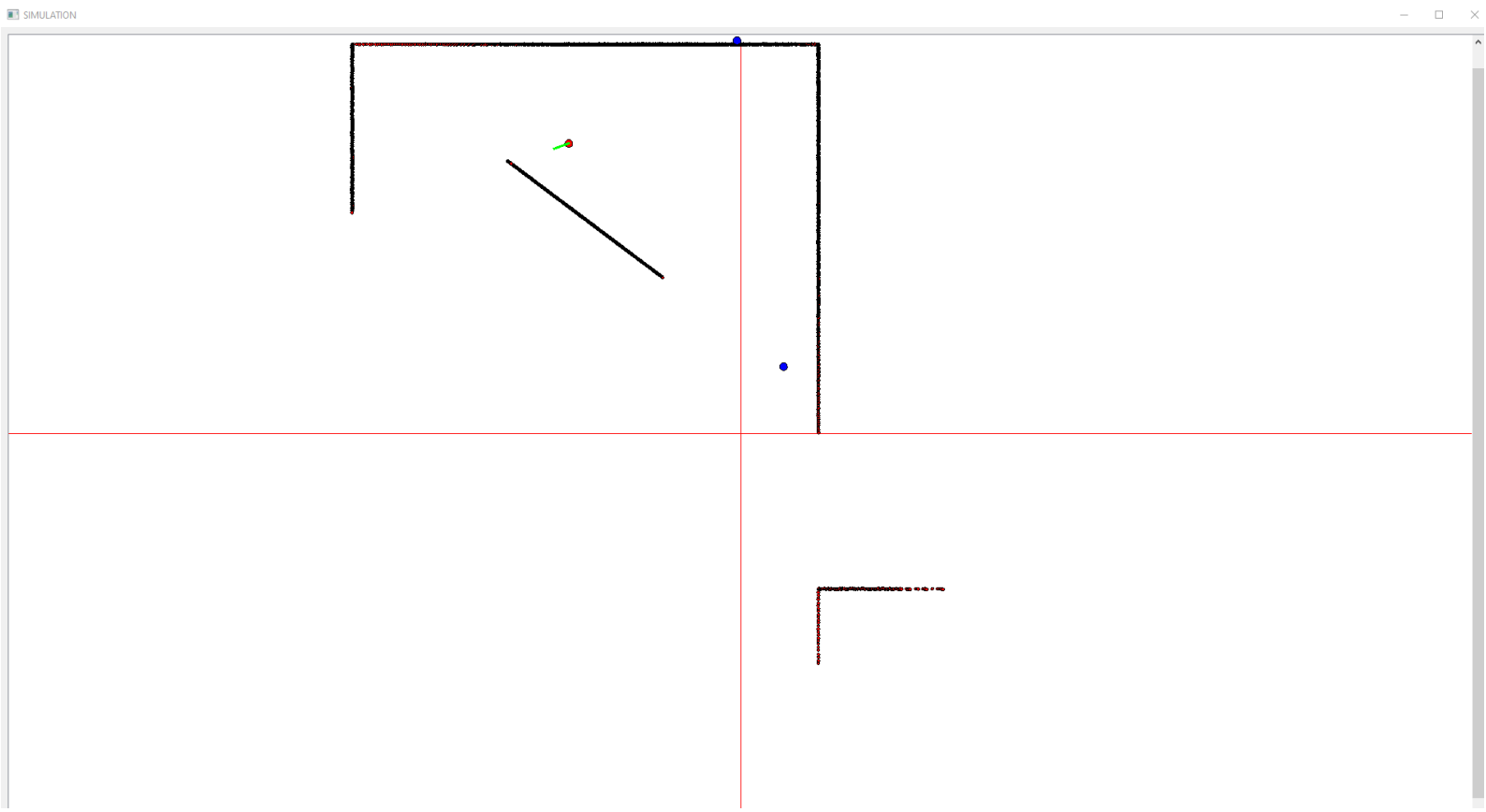
Te posljednje što nam ostaje je prikaz rezultata simulacije. Sljedeće slike su izresci zaslona računala nakon što sam pritisnuo gumb početak simulacije na početnom ekranu.



Slika 29. Situacija 2, veći dio područja poznat

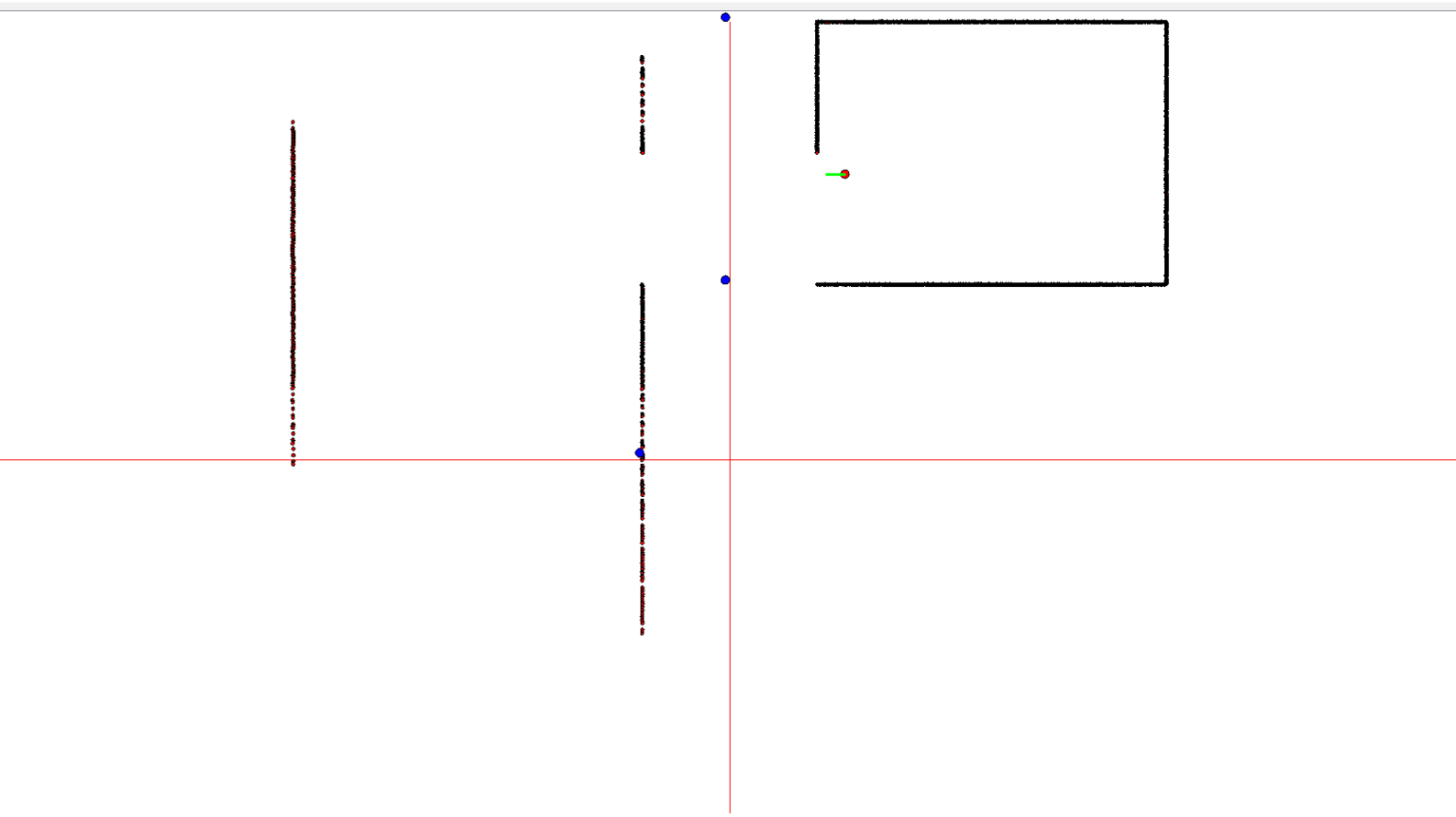
Gore na slici se može vidjeti kako izgleda situacija 2 nakon što smo malo prošli kroz prostor. Stavio sam da Lidar daje malo manje točaka nego stvarni, čisto zbog performansi simulacije. Crvene crte označuju središte globalnog koordinatnog sustava. Crvene točke označuju točke koje smo dobili od Lidar-a, te se može vidjeti kako lijepo opisuju cijeli

prostor. Plave točke su prostorne značajke u obliku točke. Lijepo se i vidi kako je donja točka značajka, iako teoretski tamo nema ništa. Ta značajka je vodoravni zid koji se vidi u 4. kvadrantu globalnog koordinatnog sustava, ali se nalazi na y osi jer smo tako definirali prostorne značajke točke. Također se može vidjeti od kuda je robot krenuo, jer u gornjem desnom kutu su jako gusta mjerenja, a prvo mjerenje koje obavim ne mogu usporediti sa postojećim točkama na ekranu kako ne bih crtao praktički istu točku, tako da se prvo mjerenje direktno preslika na zaslon, a svako iduće provjerava da ne crta ponovo istu točku.



Slika 30. Situacija 2, gusti sken

Na slici iznad sam stavio još jedan prikaz iz situacije 2, gdje sam stavio da Lidar snima gušće. Ovo je nešto slično što bi stvarni fizički Lidar dao kao rezultat. Ova donja plava točka je značajka kosog zida, jer ako si zamislimo ortogonalnu liniju na pravac zida koja također prolazi kroz središte globalnog sustava, možemo zamisliti da je ta plava točka presjek te dvije linije.



Slika 31. Situacija 1, prolaz kroz vrata

Na slici iznad se vidi primjer iz situacije 1 gdje robot izlazi iz sobe, te vidi vrata koja se nalaze nasuprot sobe u kojoj je on, te vidi zid u toj nasuprotnoj sobi. Možemo vidjeti da je uzeo gornji i donji zid kao prostornu značajku, isto kao i zid koji na kojem se nalaze nasuprotna vrata. Ova slika lijepo prikazuje vidno polje robota, te kako ne može vidjeti dublje u hodnik, te ne vidi što se zapravo nalazi u sobi nasuprot njemu.

Jedan od trendova koji se može vidjeti na svim rezultatima je taj da iako postoje neki jako očiti zidovi blizu robota, čini se kao da ih ne prepoznaje, tj. nema plavih točki koje označuju prostorne značajke. Originalno sam iscrtavao sve prepoznate prostorne značajke, ali su znale biti čitave hrpe istih značajki, jer nisam sortirao kroz značajke kako bih pazio da ne crtam dva puta „istu“ značajku.

Ovo sam riješio tako da sam odlučio crtati samo uparene značajke, što znači da se je ista značajka morala nekoliko puta za redom pojaviti, te da smo ju morali upariti kako bi se prikazala na zaslonu.

U ovim simulacijama nisam prikazao ponašanje SLAM-a u prostorima sa zakrivljenim stranicama. Robot bi možda i uspio pronaći značajke u prostoru, ukoliko su radijusi zakrivljenosti dovoljno veliki, ali jako teško da bi uspio povezati značajke, jer se mora dogoditi da stalno isti zakrivljeni dio zida uzima kao značajku.

6. ZAKLJUČAK

U ovom diplomskom radu istražen je postupak Simultaneous Localization and Mapping (SLAM) kroz računalnu simulaciju, s ciljem proučavanja njegove primjenjivosti i učinkovitosti u kontekstu autonomnih robotskih sustava. Kroz analizu dostupnih algoritama i tehnika, provedena je simulacija koja je omogućila dublje razumijevanje složenih procesa lokalizacije i mapiranja u nepoznatim okolinama. Rezultati simulacije pružili su uvid u performanse različitih SLAM-a u različitim scenarijima, ističući njihove prednosti, nedostatke i potencijalna područja poboljšanja.

Prikazana je efektivnost, te jače i slabije strane raznih algoritama korištenih u ovome radu, a među kojima se nalaze EKF algoritam, RANSAC algoritam i sl. Ovaj rad omogućuje testiranje probu raznih algoritama koji možda imaju bolje performanse, ili neke algoritme za specifične slučajeve.

Ova studija potvrđuje važnost SLAM-a kao ključnog alata u području robotike, posebice u razvoju autonomnih robotskih sustava koji zahtijevaju preciznu lokalizaciju i mapiranje u dinamičnim i nepredvidivim okolinama. Iako su simulacije pružile korisne uvide, daljnje istraživanje i eksperimentacija u stvarnim okolinama bit će ključni za verifikaciju i validaciju rezultata te primjenu u praktičnim scenarijima.

Nadalje, rezultati ove studije pružaju temelj za daljnji razvoj i optimizaciju SLAM algoritama, s ciljem poboljšanja preciznosti, brzine izvođenja i skalabilnosti. Integracija naprednih tehnologija poput dubokog učenja ili poboljšanih senzorskih sustava može doprinijeti razvoju naprednih SLAM rješenja koja su sposobna nositi se s izazovima stvarnih okolina i ostvariti autonomne robotske sustave visoke pouzdanosti i učinkovitosti.

Konačno, ovaj diplomski rad pruža temelj za daljnje istraživanje i razvoj u području SLAM-a, potičući napredak u robotici i otvarajući nove mogućnosti za primjenu autonomnih robotskih sustava u različitim industrijskim, istraživačkim i društvenim područjima.

LITERATURA

[1] Extended Kalman filter – Wikipedia, https://en.wikipedia.org/wiki/Extended_Kalman_filter, (29.2.2024.)

[2] Automaticaddison, Extended Kalman Filter (EKF) With Python Code Example, https://automaticaddison.com/extended-kalman-filter-ekf-with-python-code-example/, (4.3.2024.)

[3] Søren Riisgaard and Morten Rufus Blas, SLAM for Dummies, https://dspace.mit.edu/bitstream/handle/1721.1/119149/16-412j-spring-2005/contents/projects/1aslam_blas_repo.pdf, (16.2.2024.)