SLAM postupak na eMIR mobilnom robotu

Tuček, Leon

Sveučilište u Zagrebu
Fakultet strojarstva i brodogradnje

DIPLOMSKI RAD

Leon Tuček

Zagreb, 2022.

Sveučilište u Zagrebu
Fakultet strojarstva i brodogradnje

DIPLOMSKI RAD

SLAM POSTUPAK NA EMIR MOBILNOM ROBOTU

Mentor:
prof. dr. sc. Mladen Crneković

Student:
Leon Tuček

Zagreb, 2022.

Najveću zaslugu za ono što sam postigao pripisujem mojoj majci koja je uvijek bila tu uz mene, te se zahvaljujem mojoj djevojci na nesebičnoj podršci i razumijevanju moje prgave naravi tijekom studija.

Izjava

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

Zahvaljujem se mentoru prof. dr. sc. Mladenu Crnekoviću na savjetima i pomoći tijekom izrade rada te asistenu Marinu Lukasu na pomoći pri implementaciji rada na eMiR mobilnog robota.

Zagreb, 2022. Leon Tuček

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske radove studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala te mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum:	Prilog:
Klasa: 602-14/22-6/1	
Ur. broj: 15-1703-22-	

DIPLOMSKI ZADATAK

Student: **LEON TUČEK** Mat. br.: 0035216759

Naslov rada na hrvatskom jeziku: **SLAM postupak na eMIR mobilnom robotu**

Naslov rada na engleskom jeziku: **SLAM procedure on the eMIR mobile robot**

Opis zadatka:

Preduvjet uspješnog rada svakog mobilnog robota je poznavanje okoline u kojoj se kreće i njegovog položaja u toj okolini. To se postiže senzorima unutarnjih stanja (odometrija) te senzorima vanjskih stanja (npr. LIDAR). Kombinacijom dobivenih informacija računa se vjerojatan položaj i orjentacija mobilnog robota u zadanoj okolini. Taj je postupak u literaturi poznat kao SLAM i potrebno ga je primijeniti na eMIR mobilne robote.

U radu je potrebno:
- za odabrani LIDAR preuzeti podatke o stanju okoline (smjer i udaljenost prepreke),
- za eMIR robota 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.

Sve grafičke podatke treba prikazivati u tzv. "realnom vremenu".

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan: 29. rujna 2022.

Rok predaje rada: 1. prosinca 2022.

Predviđeni datum obrane: 12. prosinca do 16. prosinca 2022.

Zadatak zadao: prof. dr. sc. Mladen Crneković

Predsjednica Povjerenstva: prof. dr. sc. Biserka Runje

Sadržaj

Popis slika

Popis tablica

Popis oznaka

Oznaka	Jedinica	Opis
α	$^\circ$	Zakret od x ili y osi
x	mm	Pozicija po x osi
y	mm	Pozicija po y osi
T	–	Matrica homogenih transformacija
P	–	Matrica translacije
R	–	Matrica rotacije

Sažetak

Problem mapiranja i istovremenog lociranja u nepoznatom prostoru u robotici naziva se SLAM (*engl.* Simultaneous localization and mapping) te je jedan od temeljnih problema u autonomnoj robotici.

SLAM omogućuje mapiranje nepoznatog okruženja robota te se nastala mapa može koristiti za izvršavanje raznovrsnih zadataka uz pomoć poznavanja okoline robota. Neki od zadatka koje SLAM rješava su pronalazak optimalne putanje robota u mapiranom prostoru te izbjegavanje prepreka u stvarnom vremenu, odnosno robot može izbjegavati prepreke u promjenjivoj okolini. Temelj interakcije s okolinom velike većine autonomnih robota te automobila je upravo njihov SLAM sustav.

U ovom radu bit će obrađena izrada SLAM sustava uz pomoć 2D LIDAR senzora te implementacija istog na eMIR robota.

Ključne riječi: SLAM, Simultano lociranje i mapiranje, Lidar, Mobilni robot

Summary

The problem of mapping and simultaneous localization in an unknown space in robotics is called SLAM (Simultaneous localization and mapping) and is one of the fundamental problems in autonomous robotics.

SLAM enables mapping of the robot unknown environment, the resulting map can be used to perform various tasks with the information about robot environment. Some of the tasks that SLAM enables solving are finding the optimal path of the robot in the mapped unknown space and avoiding obstacles in real time, i.e. the robot can avoid obstacles in a changing environment. The vast majority of autonomous robots and cars base their interaction with the environment on their SLAM systems.

This paper will deal with the development of a SLAM system with the help of a 2D LIDAR sensor and its implementation on the eMIR robot.

Keywords: SLAM, Simultaneous localization and mapping, Lidar, Mobile robot

Poglavlje 1.

Uvod

SLAM (*engl.* Simultaneous localization and mapping, SLAM) omogućuje mapiranje nepoznatog okruženja te se nastala mapa kasnije može koristiti za rješavanje zadataka kao što su pronalazak optimalne putanje robota u prostoru te izbjegavanje prepreka. Jedna od prednosti SLAM sustava je to što za razliku od sustava s unaprijed zadanim mapa prostora SLAM sustavi su fleksibilni te se mogu prilagoditi novonastalim situacijama, npr. ako ispred robota postavimo prepreku koja nije definirana na zadanoj karti robot će ju moći sam zaobići te se vratiti na svoju definiranu putanju.

SLAM se već dugi niz godina istražuje no ne postoji krajnje rješenje problema već se teži optimizaciji za specifične slučajeve. Primjer je SLAM sustav koji se koristi u robotskim usisivačima koji je dovoljno dobro optimiziran za taj problem no ukoliko bi se isti sustav primijenio na autonomnim automobilima vrlo brzo bi došlo do problema u radu sustava.

SLAM je u današnje doba široko primjenjiv te i u raznolikoj upotrebi kao što je navigacija flotom mobilnih robota za raspoređivanje predmeta u skladištu, upravljanje samovozećih automobila, isporuka predmeta pomoću drona u nepoznatom prostoru te u još mnogo različitih područja primjena.
SLAM sustav se uvijek sastoji od barem dva senzora te se njihovi podaci preklapaju kako bi se dobila ispravna slika prostora. Jedan od senzora je uvijek senzor uz pomoć kojega možemo na neki način skenirati prostor, odnosno odrediti njegov oblik. Drugi senzor može također biti istog tipa kao prvi senzor no mora mjeriti drugo mjerno stanje robota, odnosno ne može se osloniti na samo jednu vrstu mjerenja. U najvećem broju

slučaja SLAM sustav se sastoji od konfiguracije 2D ili 3D Lidar senzora s odometrijom robota.

U pravilu se SLAM sustav dijelu na dva dijela, frontend i backend. Frontend dio sustava učitava ulazne signale senzora dok ih backend dio obrađuje, stvara mapu prostora i pozicionira robota u prostoru. Za praćenje kretanja robota kroz prostor koristi se odometrija te za mapiranje i korekciju podataka Lidar senzor ili kamera (moguća je i kombinacija).

Danas se SLAM tehnologija koristi u mnogim industrijama jer otvara mogućnosti za bolje mapiranje i razumijevanje okruženja robota, bilo u zatvorenom ili otvorenom prostoru te u zraku ili pod zemljom.

1.1. Senzori

SLAM će skoro uvijek koristiti nekoliko senzora, razvoji različitih tipova senzora bili su glavni pokretač stvaranja novih algoritama koji se koriste u SLAM-u.
Neovisnost mjernih veličina senzora obavezni je zahtjev pri odabiru senzora. Različite vrste senzora zahtijevaju različite SLAM algoritme čije su pretpostavke najprikladnije za senzore. U jednoj krajnosti, lasersko skeniranje ili vizualne značajke daju detalje mnogih točaka unutar područja, ponekad čineći SLAM zaključivanje nepotrebnim jer se oblici u tim oblacima točaka mogu lako i nedvosmisleno poravnati u svakom koraku putem registracije slike. Na suprotnoj krajnosti, senzori dodira su izuzetno rijetki jer sadrže samo informacije o točkama vrlo blizu robota, tako da zahtijevaju jake prethodne modele za kompenzaciju u čisto taktilnom SLAM-u. Većina praktičnih SLAM zadataka nalazi se negdje između ovih vizualnih i taktilnih krajnosti.

1.2. Vizualni SLAM (vSLAM)

Kao što naziv sugerira, vizualni SLAM (ili vSLAM) koristi slike dobivene s kamera i drugih senzora slike. Visual SLAM može koristiti jednostavne kamere (širokokutne, riblje oko i sferne kamere), kamere sa složenim okom (stereo i više kamera) i RGB-D kamere (dubinske i ToF kamere).

Visual SLAM može se implementirati po niskoj cijeni s relativno jeftinim kamerama. Osim toga, budući da kamere pružaju veliku količinu informacija, mogu se koristiti za otkrivanje vizualnih orijentira (prethodno izmjerene pozicije). Detekcija orijentira također se može kombinirati s optimizacijom temeljenom na upotrebom algoritama koji koriste grafove, čime se postiže fleksibilnost u implementaciji SLAM-a.

Monokularni SLAM je kada vSLAM koristi jednu kameru kao jedini senzor, što čini definiranje dubine izazovnim. To se može riješiti otkrivanjem AR markera, šahovnica ili drugih poznatih objekata na slici radi lokalizacije ili spajanjem informacija kamere s drugim senzorom kao što su inercijske mjerne jedinice (IMU), koje mogu mjeriti fizičke veličine tipa brzina i orijentacija.

Slika 1.1: Primjer rada vSLAM sustava [5]

1.3. Lidar SLAM

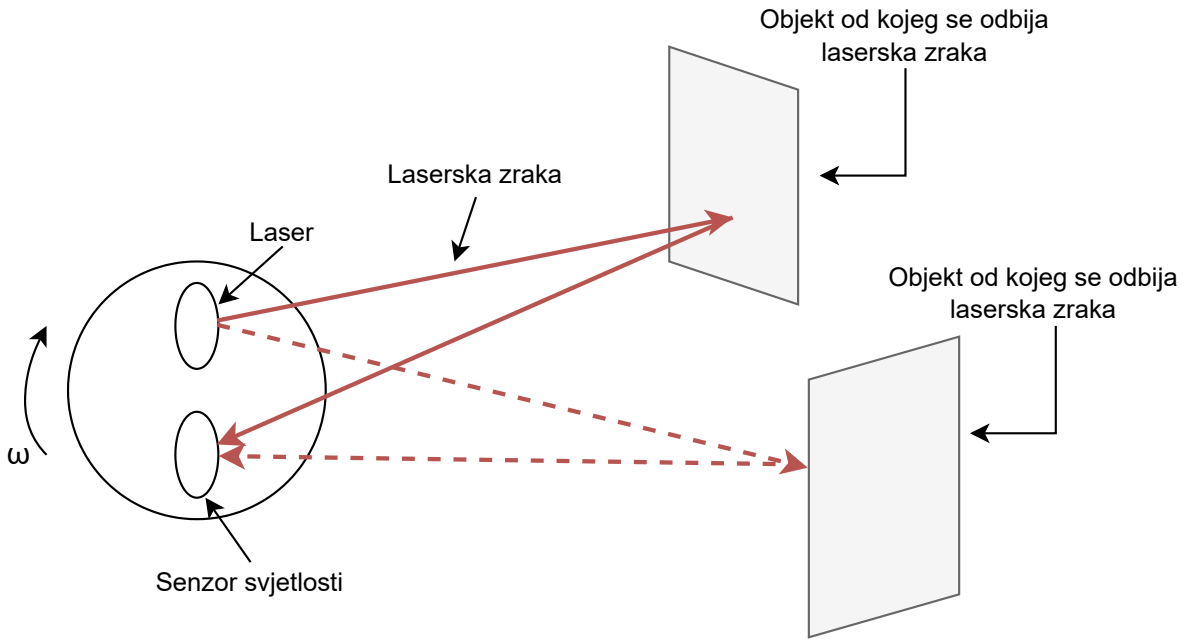
U usporedbi s kamerama laseri su znatno precizniji i koriste se na robotima koji se kreću velikom brzinom kao što su samovozeći automobili i dronovi. Izlazne vrijednosti laserskih senzora općenito su 2D (x, y) ili 3D (x, y, z) podaci oblaka točaka. Oblak točaka laserskog senzora pruža visoko precizna mjerenja udaljenosti koja se mogu iskoristiti za izradu karte i pozicioniranje robota sa SLAM-om. Općenito, kretanje se procjenjuje sekvencijalno uspoređivanjem oblaka točaka te se uz pomoć prijeđenog puta određuje lokalizacija robota. Za usklađivanje oblaka lidarskih točaka koristi se algoritam registracije kao što je iterativni algoritam najbliže točke (ICP). 2D ili 3D karte oblaka točaka mogu se prikazati kao mrežna karta.

S druge strane, oblaci točaka ne sadrže toliko finih detaljna kao slike u smislu gustoće i ne pružaju uvijek dovoljno značajki za podudaranje. Na primjer, na mjestima gdje ima malo prepreka, teško je poravnati oblake točaka i to može rezultirati gubitkom traga o lokaciji vozila. Osim toga, podudaranje oblaka točaka općenito zahtijeva veliku procesorsku snagu, stoga je potrebno optimizirati procese kako bi se poboljšala brzina.

Zbog ovih izazova, lokalizacija za autonomna vozila može uključivati spajanje drugih rezultata mjerenja kao što su odometrija kotača, globalni satelitski navigacijski sustav (GNSS) i IMU podaci. Za aplikacije kao što su skladišni roboti, obično se koristi 2D Lidar SLAM, dok se SLAM koji koristi 3D lidar oblake točaka može koristiti za UAV-ove i automatiziranu vožnju.

1.3.1. Lidar senzor

U ovom radu kao senzor za određivanje oblika prostora, odnosno mjerenje udaljenosti prepreka oko robota koristi se Lidar senzor. Lidar je senzor za mjerenje udaljenosti ciljanjem objekta ili površine laserom i mjerenjem vremena potrebnog za povratak reflektirane svjetlosti do prijemnika, tj. senzora svjetlosti. Prednost Lidar senzora je što može mjeriti udaljenosti predmeta od robota u svim smjerovima robota u skoro pa istom trenutku, odnosno može vidjeti sve prepreke oko robota u mjernom radijusu u trenutku mjerenja.



Slika 1.2: Princip rada Lidar senzora

1.3.2. Enkoder

Enkoder je senzor koji mjeri pomak, u ovom slučaju kutni pomak kotača mobilnog robota. Enkoderi se dijele na apsolutne i inkrementalne. Kod apsolutnih enkodera svaka pozicija senzora ima svoj jedinstveni kod te senzor u svakom trenutku zna svoj položaj. U pravilu apsolutni enkoderi za bilježenje svojeg položaja koriste Grayev kod.
Inkrementalni enkoderi daju informaciju o relativnom pomaku u odnosu na referentnu točku te ako senzor izgubi napajanje gubi se podatak o položaju u odnosu na referentnu točku. Mobilni robot eMIR opremljen je s dva enkodera te se svaki nalazi na pogonskim kotačima.

Poglavlje 2.

Prikupljanje i obrada podataka senzora

2.1. Lidar senzor

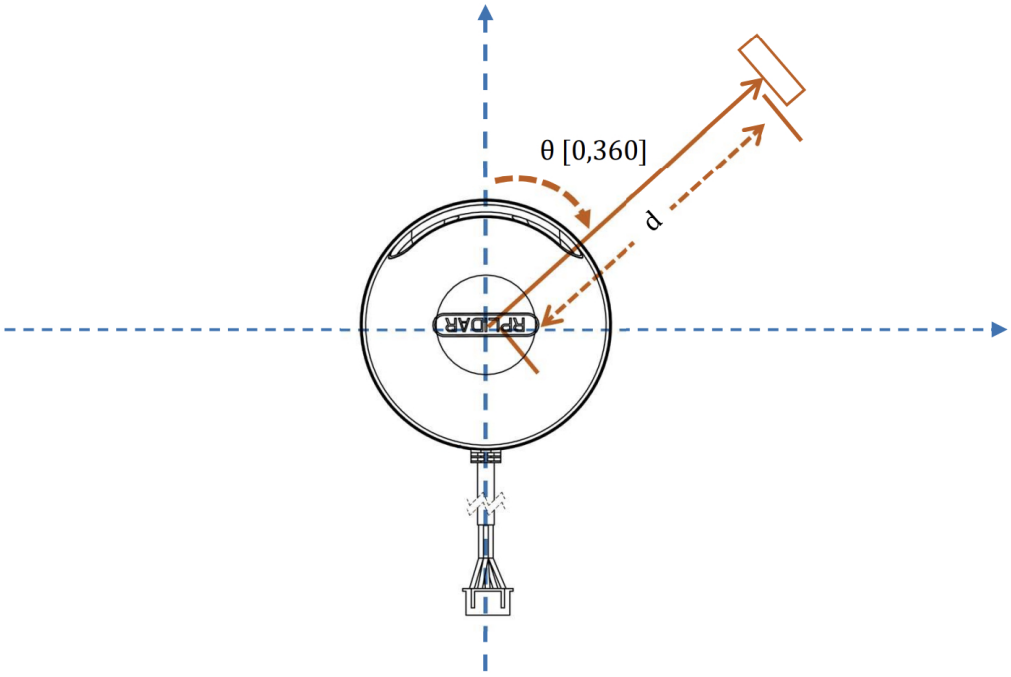
Za prikupljanje i obradu podataka koristi se programski jezik Python. Python je odabran kao jezik zbog njegove jednostavnosti koja pomaže u brzini izrade programa te široke podrške. U radu je korišten Lidar senzor RPLiDAR A2M8, a za serijsku komunikaciju s Lidar senzorom koristi se Python biblioteka RPLidar [2].

Slika 2.1: Senzor RPLiDAR A2M8 [3]

Kako bi se olakšao rad s Lidar senzorom napisana je programska klasa Lidar koja omoguće lakšu integraciju u postojeći i druge sustave. Na primjer ako odaberemo drugačiji Lidar senzor nije potrebno mijenjati i prilagođavati druge dijelove SLAM sustava već samo Lidar klasu koja obavlja komunikaciju s Lidar senzorom te obrađuje prikupljene podataka.

Navedeni Lidar senzor prikupljene podatke vraća kao niz podataka koji se sastoji od

kuta zakreta od y osi i udaljenosti skeniranog predmeta od senzora u milimetrima. Odnosno, Lidar senzor podatke vraća u polarnom koordinatnom sustavu zakrenutom za 90°.



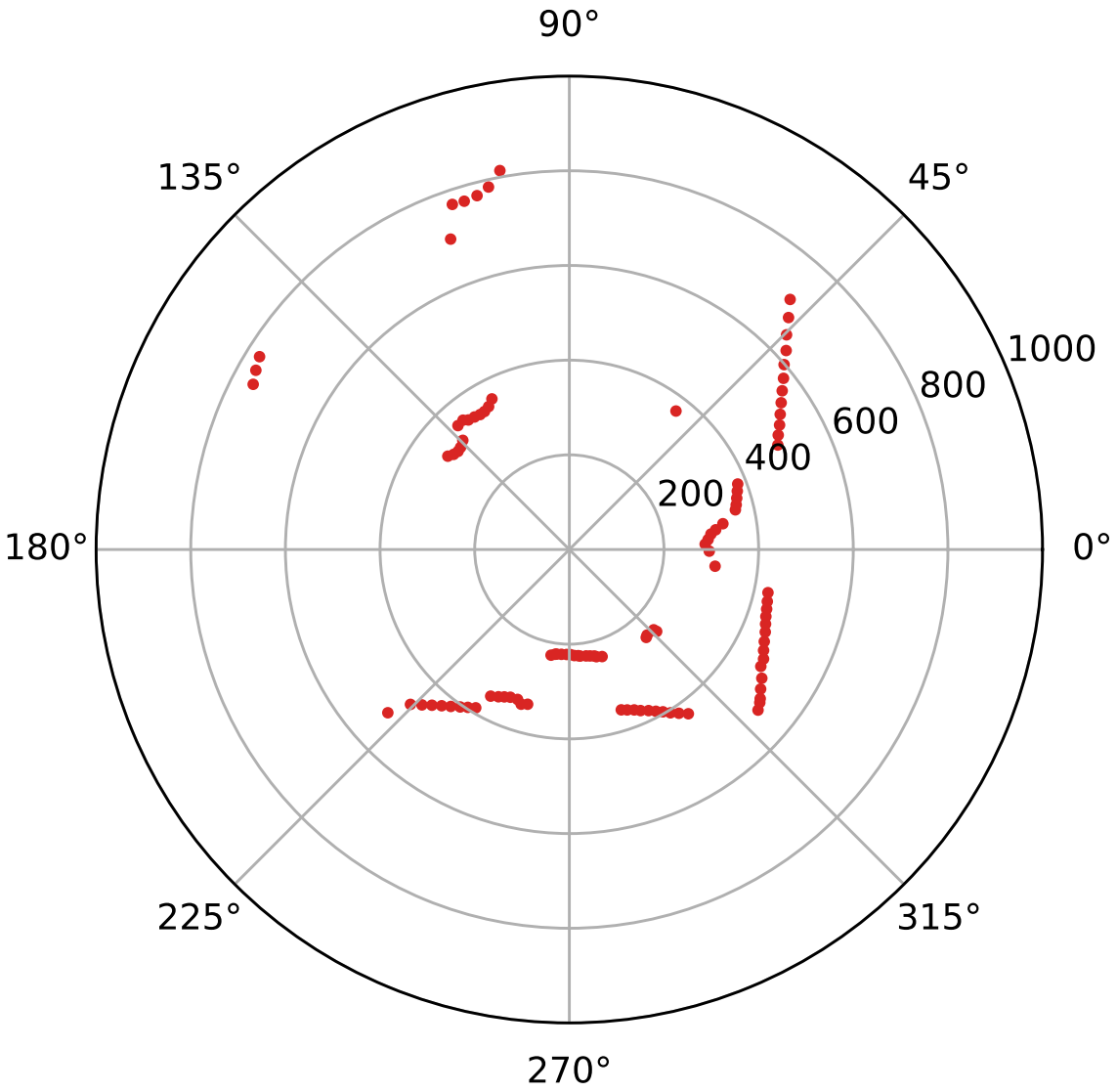
Slika 2.2: Definicija koordinatnog sustava RPLiDAR A2M8 senzora [4]

Kako bi prikupljene podatke prikazali u kartezijevom koordinatnom sustavu potrebno je točke iz gore definiranog koordinatnog sustava pretvoriti u kartezijev na sljedeći način:

$$x = d\sin(\alpha) \tag{2.1}$$

$$y = d\cos(\alpha) \tag{2.2}$$

Gdje je d udaljenosti skeniranog predmeta od senzora u milimetrima, a α kuta zakreta senzora u stupnjevima od osi y.



Slika 2.3: Neobrađeni Lidar podaci

2.1.1. Tehnička specifikacija Lidar senzora

Tablica 2.1: Tehnička specifikacija Lidar senzora

Masa	190 g
Radna temperatura okoline	0 – 40 °C
Radni napon	5 V
Radna struja	450 – 600 mA
Minimalna mjerna udaljenost	0,2 m
Maksimalna mjerna udaljenost	12 m
Rezolucija mjerenja udaljenosti	$< 0,5$ mm
Rezolucija mjerenja kutnog zakreta	$0,9 \pm 0,45°$
Učestalost mjerenja	10 ± 5 Hz

Navedeni senzor komunikaciju s računalom vrši preko 3.3V-TTL serijskog porta (UART).

Također važno je naglasiti kako zbog principa rada Lidar senzora nije moguć rad senzora u okruženju u kojem se nalazi puno reflektirajućih objekata tipa zrcala ili sjajnih metalnih površina jer to dovodi do pogrešnog mjerenja te određeni objekti mogu postati nevidljivi Lidar senzoru.

2.1.2. Lidar potprogram

Lidar potprogram, odnosno programska klasa Lidar napisana je na način kako bi bila što fleksibilnija u različitim sustavima, odnosno ista klasa se može implementira u više raznolikih sustav te nije ograničena na samo ovaj SLAM sustav.
Uloga Lidar programske klase je komunikacija s Lidar senzorom te prikupljanje podataka uz pomoć Lidar senzora.
Prilikom inicijalizacije klase potrebno je proslijediti parametar *port_name* tipa string s nazivom serijskog porta na koji je Lidar senzor spojen na računalo te *number_of_scans* tipa integer koji određuje koliko okretaja, odnosno očitanja senzora će Lidar klasa izvršiti

prije nego što obradi i spremi podatke senzora. Parametar *number_of_scans* se može koristiti kako bi Lidar senzor dobio točniju i puniju sliku prostora oko robota jer će se isti prostor skenirati broj puta jednak vrijednosti parametra *number_of_scans*.
Treba imati na umu kako navedeni parametar ne bi trebao biti veći od 1 ako se robot prilikom skeniranja kreće.

Klasa Lidar se sastoji od inicijalizacije te četiri metode: *start*, *stop*, *get_coordinates* te *scan*.

Kako prikupljanje i obrada podataka ne bi utjecalo na blokiranje glavne procesorske jezgre metoda *scan* se uvijek poziva na posebnoj dretvi.

2.1.3. Inicijalizacija klase

Inicijalizacija klase je vrlo jednostavna te se u njoj samo postavljaju vrijednosti proslijeđenih parametara pri inicijalizaciji.

```
procedure INIT(passed_port_name, passed_number_of_scans)
    port_name ← passed_port_name
    number_of_scan ← passed_number_of_scan
end procedure
```

2.1.4. Metoda *start*

Metoda *start* koristi se kako bi Lidar klasa počela prikupljati i obrađivati podatke sa senzora. Unutar navedene metode kreira se posebna dretva koja se koristi za obrađivanje podataka kako bi samo prikupljanje podataka bilo asinkrono, odnosno kako ne bi blokiralo glavnu procesorsku jezgru. U novo stvorenoj dretvi poziva se metoda *scan*.

```
procedure START
    is_running ← true
    CreateNewThreadAndRunMethod(scan())
end procedure
```

2.1.5. Metoda *stop*

Metoda *stop* koristi se za zaustavljanje prikupljanja i obrade podataka. Kako bi se Lidar senzor pravilno ugasio potrebno je pozvati ovu metodu pri završetku rada.

```
procedure STOP
    is_running ← false
    stopLidarSensor()
end procedure
```

2.1.6. Metoda *scan*

Metoda *scan* vrši komunikaciju s Lidar senzorom i te obradu podataka. Samo prikupljanje podataka se vrši svakih 50 ms. Zapravo skoro pa sva logika ove klase se nalazi u metodi scan te je ovo računski najzahtjevnija metoda ove klase.

```
procedure SCAN
    while is_running is true do
        new_scan_points ← []
        for i = 0 to number_of_scan do
            for j = 0 to lidar_measurements do
                angle ← ReadLidarAngleAtIndex(j)
                distance ← ReadLidarDistanceAtIndex(j)

                radians_angle ← radians(angle)
                x ← distance * sin(radians_angle)
                y ← distance * cos(radians_angle)

                new_scan_points.append([x, y])
            end for
        end for

        points ← new_scan_points
        sleep(0.05)

    end while
end procedure
```

2.1.7. Metoda *get_coordinates*

Metoda *get_coordinates* je vrlo jednostavna te je njezina jedina uloga je ta da vraća listu koordinata prikupljenih i obrađenih točaka. Sama lista koordinata skeniranih točaka se kreira u *scan* metodi.

```
procedure GET_COORDINATES
    return points
end procedure
```

2.2. Odometrija

Odometrija igra važnu ulogu u SLAM procesu jer se s podaci dobiveni pomoću odometrije preklapaju s podacima Lidar senzora, odnosno poravnavaju. Nažalost sama odometrija uz Lidar nije dovoljna jer dolazi do greške u mjerenju odometrije. Greška mjerenja može nastati iz više razloga, od nepreciznosti enkodera na kotačima pa do proklizavanja samih kotača radi niskog faktora trenja podloge po kojoj se mobilni robot kreće. Dodatni problem kod greške odometrije je ta što se greška s vremenom nakuplja te postaje sve veća. U nastavku rada bit će objašnjeno na koji način se nastale greške odometrije kompenziraju uz pomoć Lidar podataka.

Implementacija odometrije i senzora koji se koriste za istu neće biti obrađeni u ovom radu jer se koristi već gotovo rješenje koje je postavljeno na eMIR mobilnog robota. Jedina izmjena od gotovog rješenja je ta što je za potrebe ovog rada Python skripta za komunikaciju s eMiR robotom refaktorirana kako bi se modernizirala te optimizirala.

Poglavlje 3.

Zatvaranje petlje

Za mjerenje udaljenosti predmeta od robota koristi se LIDAR ili kamera (moguća je i kombinacija) dok odometrija procjenjuje kretanje robota, no zbog proklizavanja kotača i ostalih smetnji u mjerenju pogreška mjerenja se tijekom vremena akumulira te raste. Također treba naglasiti kako je kod mjerenja moguća i pojava šuma koja unosi još dodatnu grešku u mjerenju. Nastala pogreška uzrokuje pogrešno preklapanje mjerenja dobivenima uz pomoć LIDAR-a ili kamere te rezultira krivom ili čak fizički nemogućom kartom prostora. Uzmimo za primjer prolazak mobilnog robota oko pravokutnog oblika, odnosno hodnikom oko druge prostorije.

Slika 3.1: Akumulirana greška odometrije [1]

Kao što je vidljivo sa slike 3.1 početna i završna točka robota se ne preklapaju iako je robot započeo i završio svoje kretanje u istoj točki. Isto tako nastala mapa prostora nije točna. Ovakve pogreške su neizbježne te ih je potrebno kompenzirati s nekom drugom vrstom mjernih podataka, na primjer s Lidar senzorom ili kamerom.

Kako bi riješili nastali problem koristi se tehnika zatvaranja petlje te uz pomoć navedene tehnike ispravljamo pogrešne poze robota, odnosno pogrešno preklopljene Lidar podatke.
Tehnika zatvaranje petlje funkcionira na principu da algoritam pronalazi zajedničke točke između dvije uzastopne Lidar „slike", odnosno dva uzastopno prikupljena oblaka točaka te proračunava pogrešku odometrije.
Samo preklapanje prikupljenih podataka nije jednostavan problem te također ne postoji jedinstveno rješenje, pogotovo jer se podaci ponekad mogu preklopiti na više različitih načina.

Slika 3.2: Ispravljena pogreška odometrije uz pomoć LIDAR podataka i odometrije [1]

Kao što je vidljivo sa slike 3.2 po završetku procesa zatvaranje petlje skenirana prostorija je poprimila točan oblik te je polazna točka robota jednaka završnoj točki. Iz ovog razloga sam proces zatvaranja petlje je zapravo i najvažniji dio SLAM sustava jer bez njega SLAM sustav postaje beskoristan zbog velike greške pri spajanju dvije vrste podataka, odnosno podataka Lidar senzora i odometrije.

Jedno od ograničenja SLAM sustava je rad u velikim otvorenim prostorima koji uzrokuje nemogućnost detektiranja točaka u prostoru te navedeno dovodi do gubitka informacije o položaju robota. Na primjer ukoliko bi stavili robota u sredinu velike prazne hale u kojoj je razmak između zidova veći od 12 metara njegov Lidar senzor s dometom od 12 metara ne bi uspio prikupiti niti jednu točku u prostoru te ne bi imali nikakvu informaciju o položaju robota osim odometrije.

Uporaba računalnih resursa je još jedan problem pri implementaciji SLAM-a na hardver robota. Računanje se obično izvodi na kompaktnim ugrađenim mikroprocesorima niske potrošnje energije koji imaju ograničenu procesorsku snagu. Kako bi se postigla točna lokalizacija, bitno je izvršiti obradu slike i podudaranje oblaka točaka u relativno kratkom vremenu. Osim toga, optimizacijski izračuni kao što je zatvaranje petlje procesi su koji koriste puno računalnih resursa. Izazov je kako izvršiti tako računalno skupu obradu podataka na ugrađenim mikroračunalima.

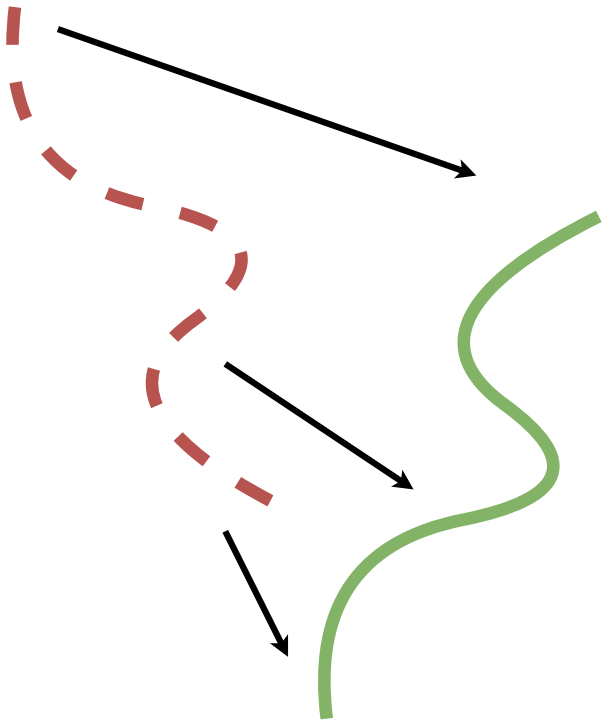
Jedna od protumjera je pokretanje različitih procesa paralelno odnosno asinkrono. Procesi kao što je očitanje vrijednosti senzora i procesiranje podudaranja oblaka točaka relativno su prikladni za paralelizaciju. Korištenje višejezgrenih procesora za prikupljanje i obradu podataka te ugrađenih grafičkih procesora može dodatno poboljšati brzine u nekim slučajevima, pogotovo kada je riječ o vSLAM sustavu, odnosno sustavu koji koristi kameru za prikupljanje informacija o položaju.

3.1. Iterativni algoritam najbliže točke

Iterativni algoritam najbliže točke (*engl.* Iterative closest point, ICP) je algoritam koji se koristi za preklapanje dvaju različitih oblaka točaka. ICP se često koristi za rekonstrukciju 2D ili 3D površina iz različitih skeniranja, za lokalizaciju robota i postizanje optimalnog planiranja putanje (osobito kada je odometrija kotača nepouzdana zbog skliskog terena), za registriranje modela kostiju te na još mnogo drugih mjesta.

Kod iterativnog algoritama najbliže točke, jedan oblak točaka ostaje fiksan dok se drugi referentni oblak točaka transformira kako bi najbolje odgovarao fiksnom oblaku točaka. Odnosno, algoritam iterativno revidira transformaciju (kombinaciju translacije i rotacije) potrebnu za minimiziranje pogreške, obično zbroj kvadrata razlika između koordinata uparenih parova fiksnog oblaka točaka od referentnog oblaka točaka.

Iterativni algoritam najbliže točke je jedan od naširoko korištenih algoritama za usklađivanje 2D i 3D oblaka točaka s obzirom na početnu pretpostavku potrebne transformacije. Algoritam su prvi predstavili Chen i Medioni, te Besl i McKay. [6] Tijekom vremena razvilo se još inačica algoritma kao što su točka u točku, točka u ravninu te linearni i nelinearni.



Slika 3.3: Zelena linija predstavlja fiksni oblak točaka dok crvena referentni oblak točaka koji treba uz pomoć algoritma poravnat s fiksnim oblakom točaka

Iterativni algoritam najbliže točke kao ulaz prima referentni i fiksni oblak točaka te estimiranu matricu transformacija za preklapanje dva oblaka točaka te vraća matricu transformacija koja preklapa referentni oblak točaka na fiksni pri minimzaciji pogreške. Nadalje u algoritam je moguće i implementirati još dva dodatna ulazna parametra, maksimalni broj iteraciji te maksimalna dozvoljena pogreška. Oba dva parametra služe kao izlazni uvjeti za algoritam.

Algoritam se sastoji od sljedećih koraka:

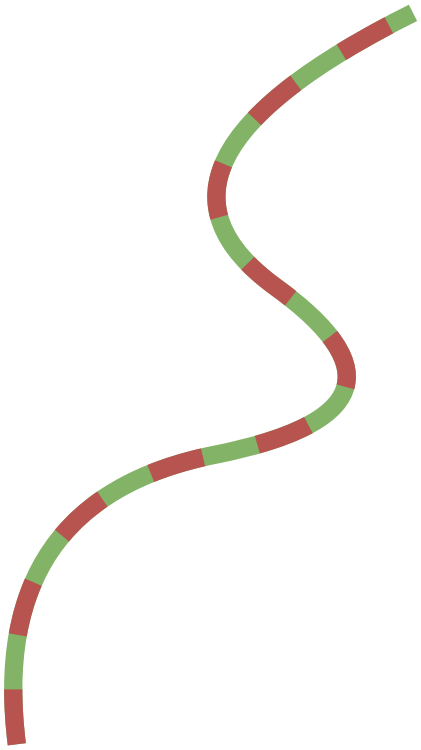
1. Za svaku točku referentnog oblaka točaka pronađi najbližu točku u fiksnom oblaka točaka

2. Odredi težište referentnog i pronađenih točaka u 1. koraku iz fiksnog oblaka točaka

3. Translatiraj referentni oblak točaka tako da se težišta pronađena u 2. točki fiksnog i referentnog oblaka točaka preklope

4. Odredi rotaciju referentnog oblaka točaka koja minimizira udaljenosti pronađenih točaka iz 1. koraka i referentnih točaka uz pomoć dekompozicija na singularne vrijednosti matrice

5. Iteriraj ako je pogreška veća od maksimalne dozvoljene pogreške i ako je broj iteracija manji od maksimalnog dopuštenog broja iteracija, u suprotnom vrati izračunatu matricu homogenih transformacija.

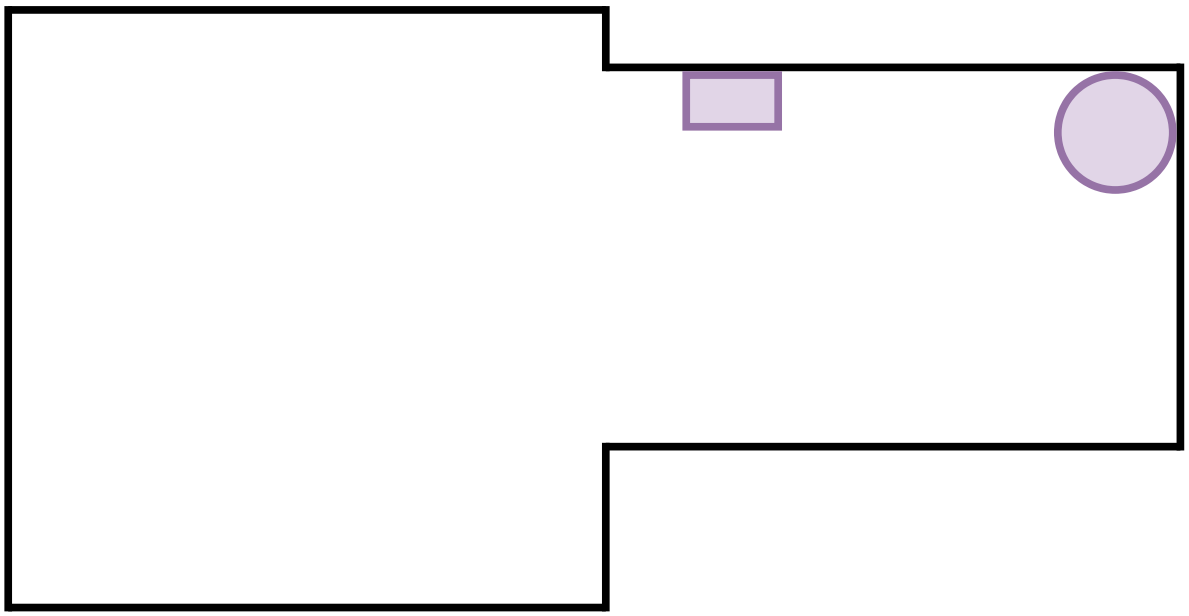


Slika 3.4: Poravnati oblaci točaka iz slike 3.3

U ovom radu iterativni algoritam najbliže točke koristi se kako bi se preklopile točke prikupljene uz pomoć Lidar podataka te se odometrijski podaci koriste za generiranje estimirane matrice transformacije. Korištena je modificirana javno dostupna implementacija algoritma [7].

3.2. Razvojni poligon

Prilikom izrade SLAM sustava bilo je potrebno izraditi konstantno i nepromjenjivo okruženje u kojem će se sustav moći testirati. Iz navedenog razloga izrađen je poligon od kartona sljedećeg oblika.

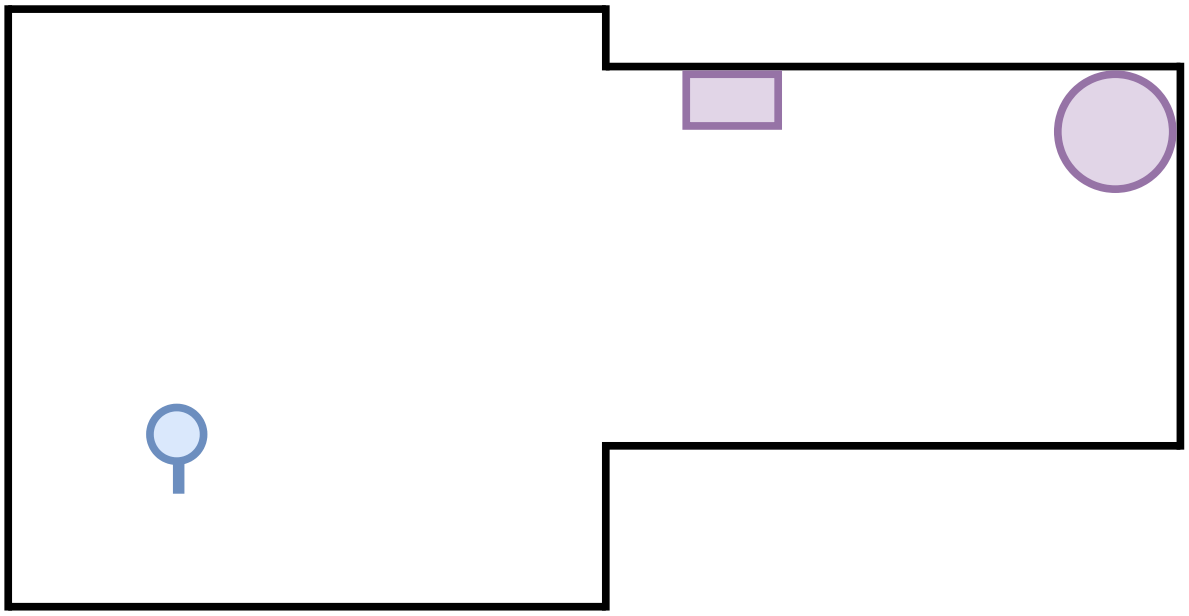


Slika 3.5: Oblik testnog poligona

U poligon je postavljena još jedna manja kutija i valjak. Navedeni objekti su postavljeni u poligon iz razloga jer u poligonu ne postoji jedna točka u kojoj su vidljive sve strane postavljenih objekata te da bi se dobila potpuna „slika" svakog objekta odnosno 2D sken potrebno je preklopiti više oblaka točaka prikupljenih Lidar senzorom.

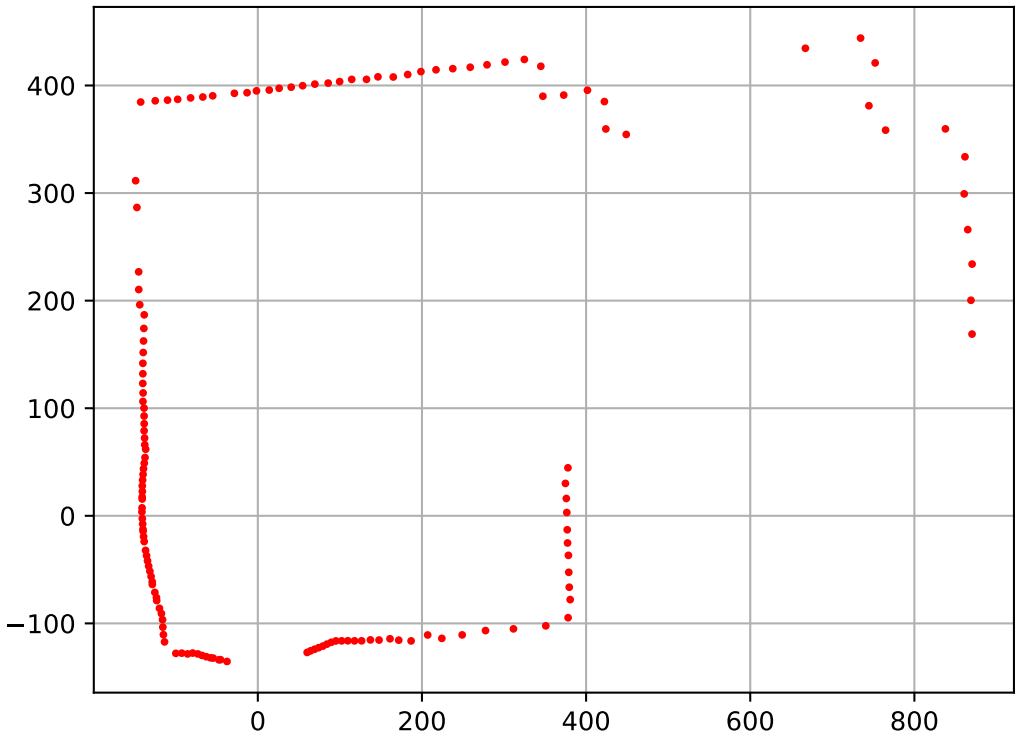
Poligon je skeniran s Lidar senzorom iz četiri različite točke te je razmak i zakret između svake od četiri točke izmjeren kako bi mogli simulirati odometriju. Mjerenje položaja senzora prilikom skeniranja poligona ne mora biti točno jer kao što je i ranije rečeno, sama odometrija često daje netočna odnosno vrlo neprecizna mjerenja. U nastavku će biti prikazani prikupljeni podaci iz svake od četiri točke poligona.

1. točka



Slika 3.6: Položaj i orijentacija Lidar senzora u poligonu

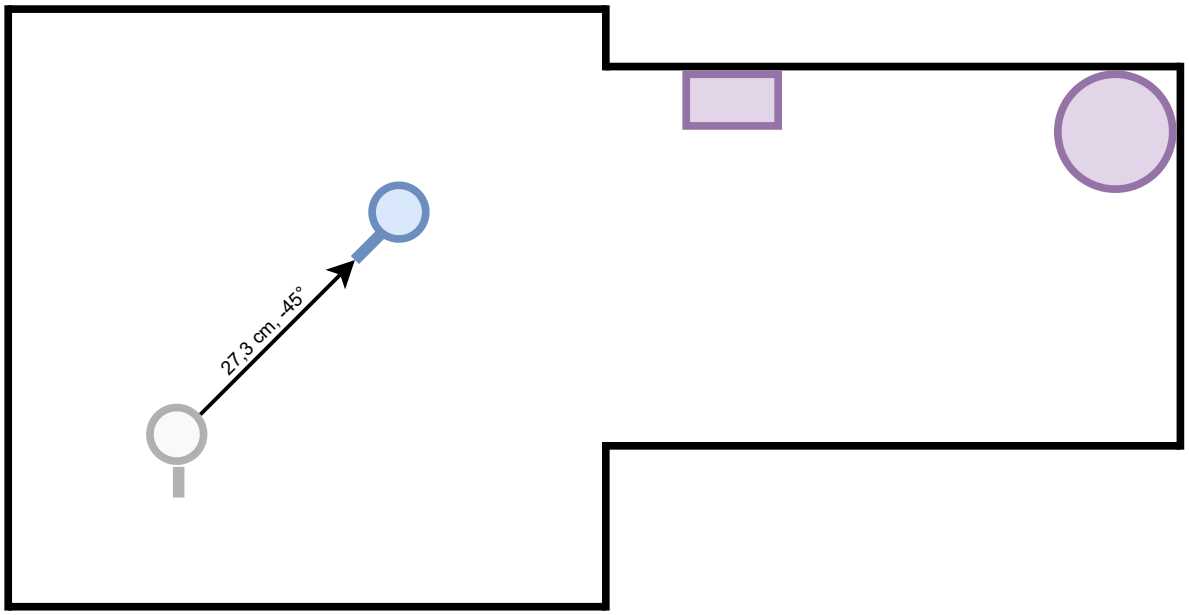
Prva točka mjerenja će također biti i ishodište globalnog koordinantnog sustava na koji će se preklapati svi podaci u narednim mjerenjima.



Slika 3.7: Prikupljeni podaci Lidar senzorom

Kao što je vidljivo na slici 3.7 Lidar senzor se nalazi u ishodištu koordinantnog sustava, crvene točke predstavljaju detektirane bridove poligona te se može primijetit kako je samo dio poligona vidljiv dok se ostatak ne vidi zbog oblika poligona.

2. točka



Slika 3.8: Položaj i orijentacija Lidar senzora u poligonu

Nakon prikupljanja podataka u 1. točki Lidar senzor je zaokrenut za -45°, odnosno 45° udesno te je pomaknut za 27,3 cm unaprijed. Pomak Lidar senzora od točke 1 do točke 2 potrebno je prikazati u matrici homogenih transformacije T_{12}. Sama matrica homogenih transformacija se sastoji od matrice rotacije R_{12} te matrice translacije P_{12}. Gdje je matrica T_{12} definirana kao

$$T_{12} = \begin{bmatrix} R_{12} & P_{12} \\ 0 & 1 \end{bmatrix} \tag{3.1}$$

matrica R_{12} je definirana kao

$$R_{12} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{3.2}$$

gdje je α kut pomaka Lidar senzora, u ovom slučaju -45°.

matrica P_{12} je definirana kao

$$P_{12} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.3}$$

gdje su x i y pomaci u smjeru osi x i y.

Uvrstimo li izraze definirane u 3.2 i 3.3 u 3.1 dobivamo konačni oblik matrice homogenih transformacija, odnosno:

$$T_{12} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & x \\ \sin(\alpha) & \cos(\alpha) & y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

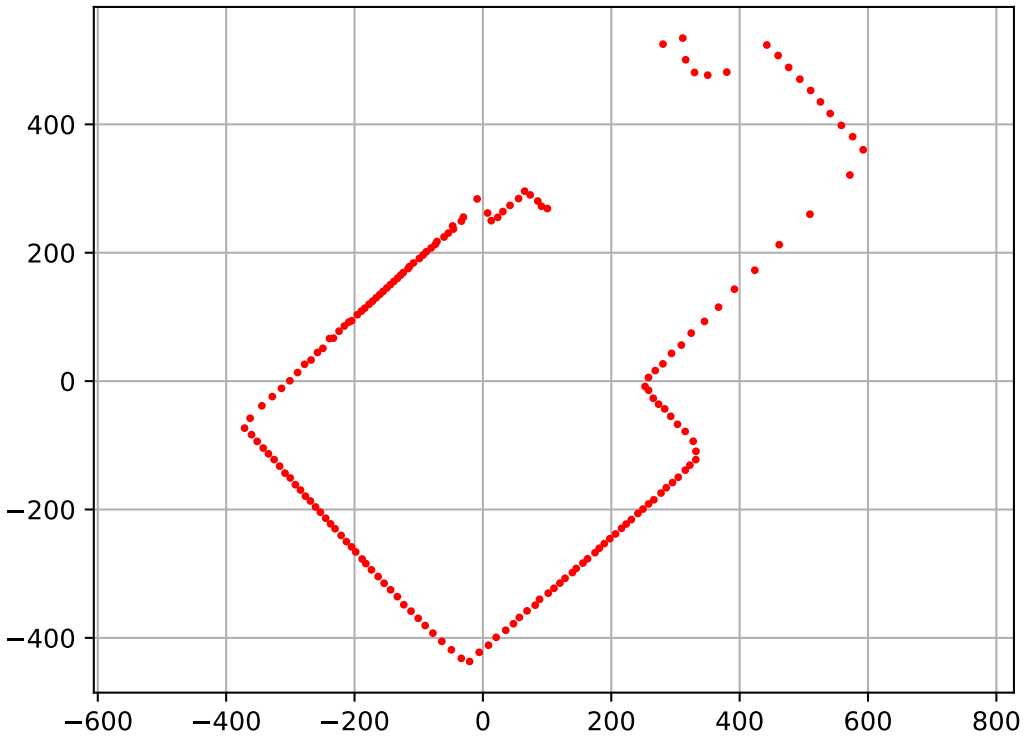
Jednostavnom trigonometrijom možemo izračunati koordinate točke 2 koje vrijede dok se točka 1 smatra ishodištem koordinatnog sustava

$$x = 27,3 \sin(45°) = 19,304 \tag{3.5}$$

$$y = 27,3 \cos(45°) = 19,304 \tag{3.6}$$

Odnosno matrica T_{12} poprima konačni oblik

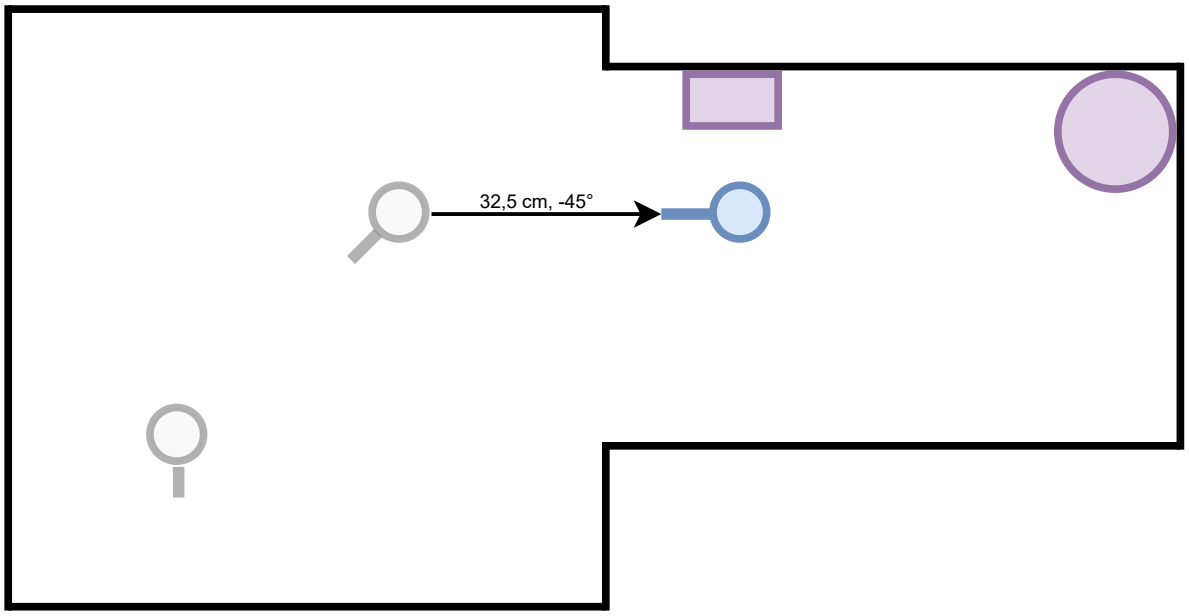
$$T_{12} = \begin{bmatrix} \cos(-45°) & -\sin(-45°) & 19,304 \\ \sin(-45°) & \cos(-45°) & 19,304 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$



Slika 3.9: Prikupljeni podaci Lidar senzorom

Kao i na slici 3.7 vidljivo je kako se Lidar opet nalazi u ishodištu koordinatnog sustava no ovaj put su prikupljeni podaci drugačijeg oblika te je vidljiva drugačija količina bridova izrađenog poligona.

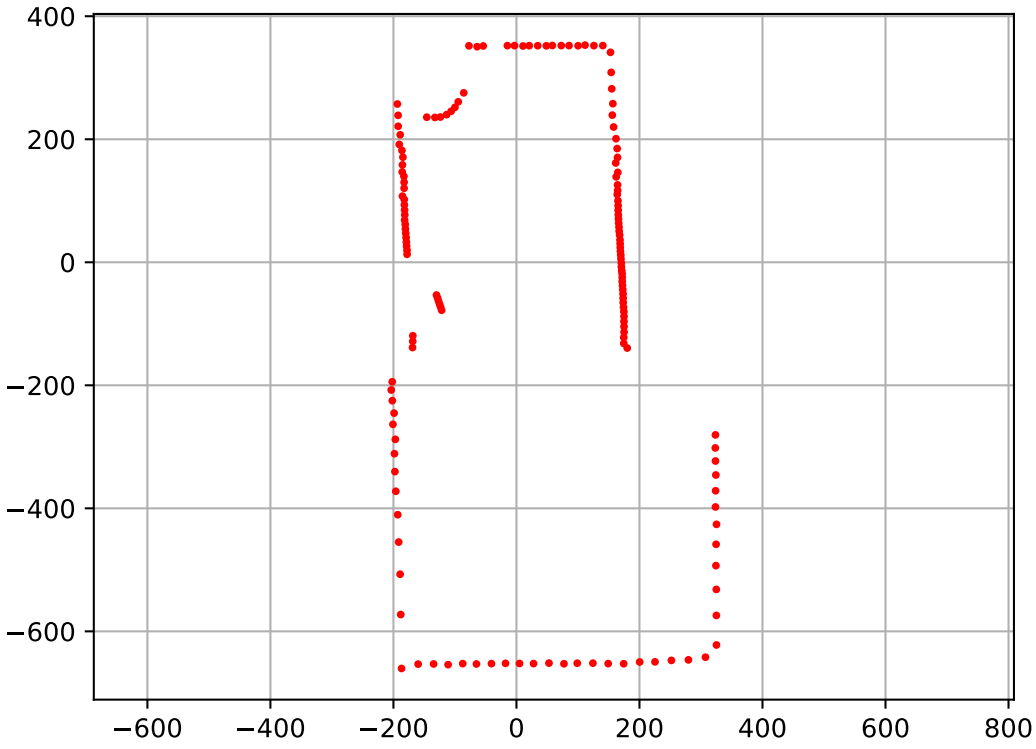
3. točka



Slika 3.10: Položaj i orijentacija Lidar senzora u poligonu

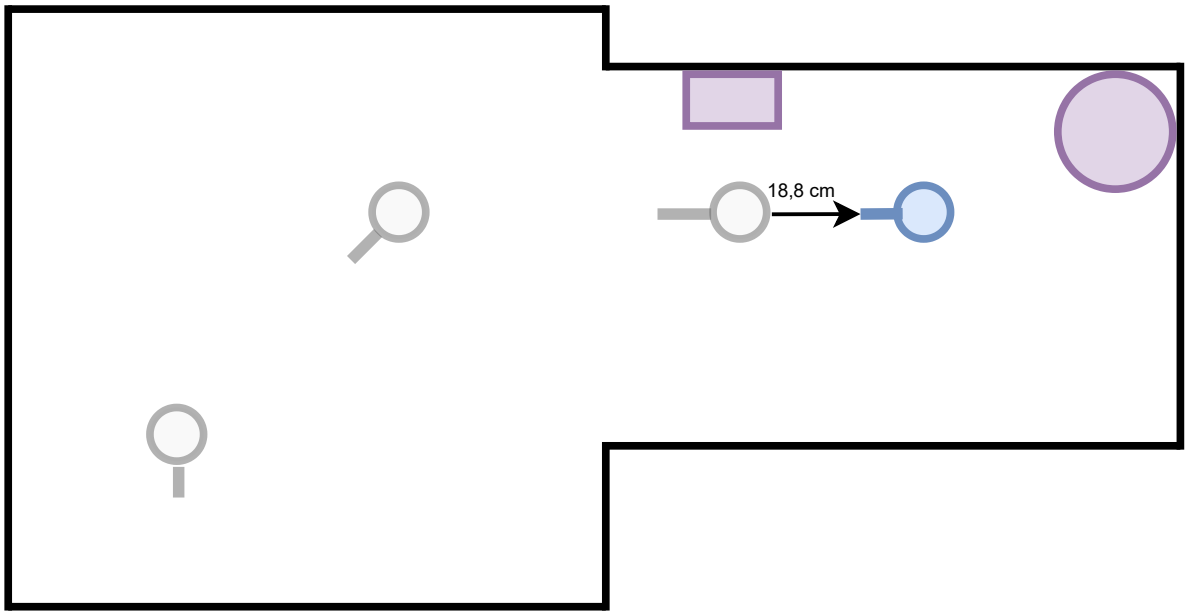
Nakon prikupljanja podataka u 2. točki Lidar senzor je još jednom zaokrenut za -45°, odnosno 45° udesno te je pomaknut za 32,5 cm unaprijed. Pomak Lidar senzora od točke 2 do točke 3 zapisan u obliku matrice homogene transformacije glasi:

$$T_{23} = \begin{bmatrix} \cos(-45°) & -\sin(-45°) & 22,981 \\ \sin(-45°) & \cos(-45°) & 22,981 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$



Slika 3.11: Prikupljeni podaci Lidar senzorom

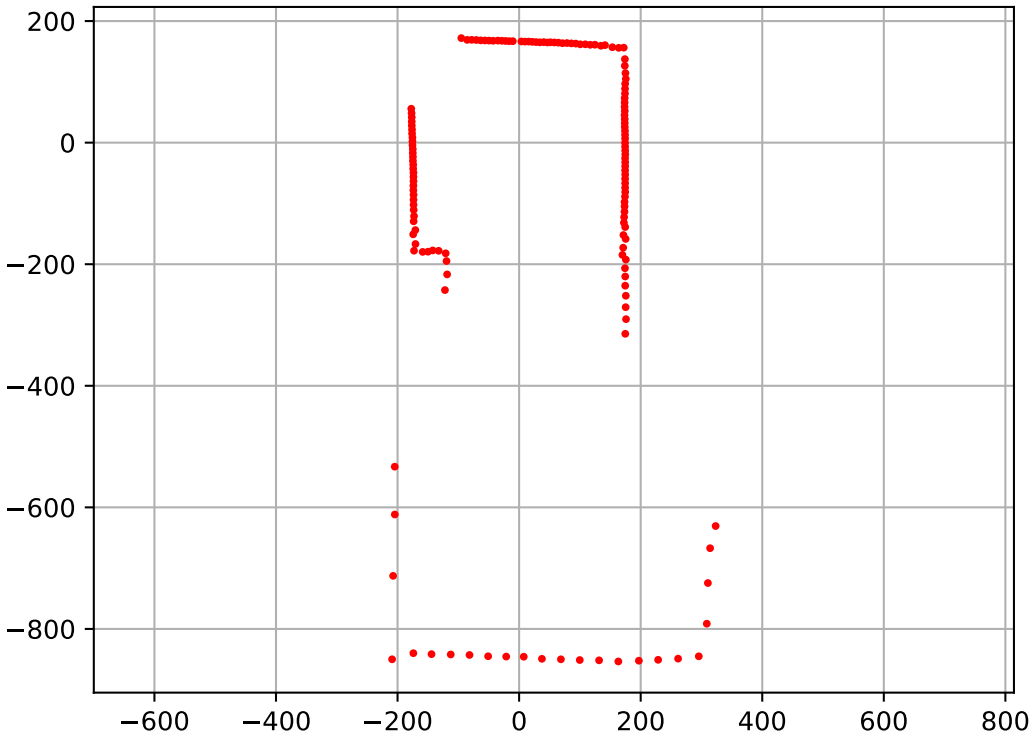
4. točka



Slika 3.12: Položaj i orijentacija Lidar senzora u poligonu

Lidar senzor je pomaknut 18,8 cm od točke 3 do 4 te je ovo ujedno i zadnja točka u kojoj su prikupljeni podaci Lidar senzora na poligonu.

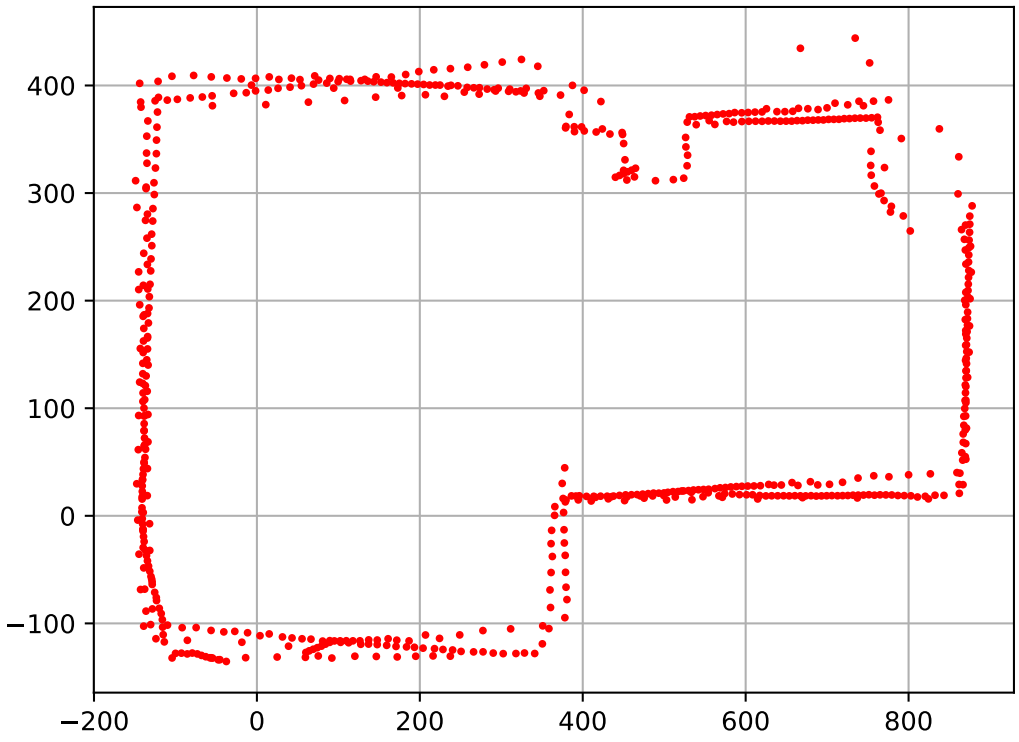
$$T_{34} = \begin{bmatrix} \cos(0) & -\sin(0) & 0 \\ \sin(0) & \cos(0) & 18,8 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$



Slika 3.13: Prikupljeni podaci Lidar senzorom

3.3. Preklapanje prikupljenih podataka

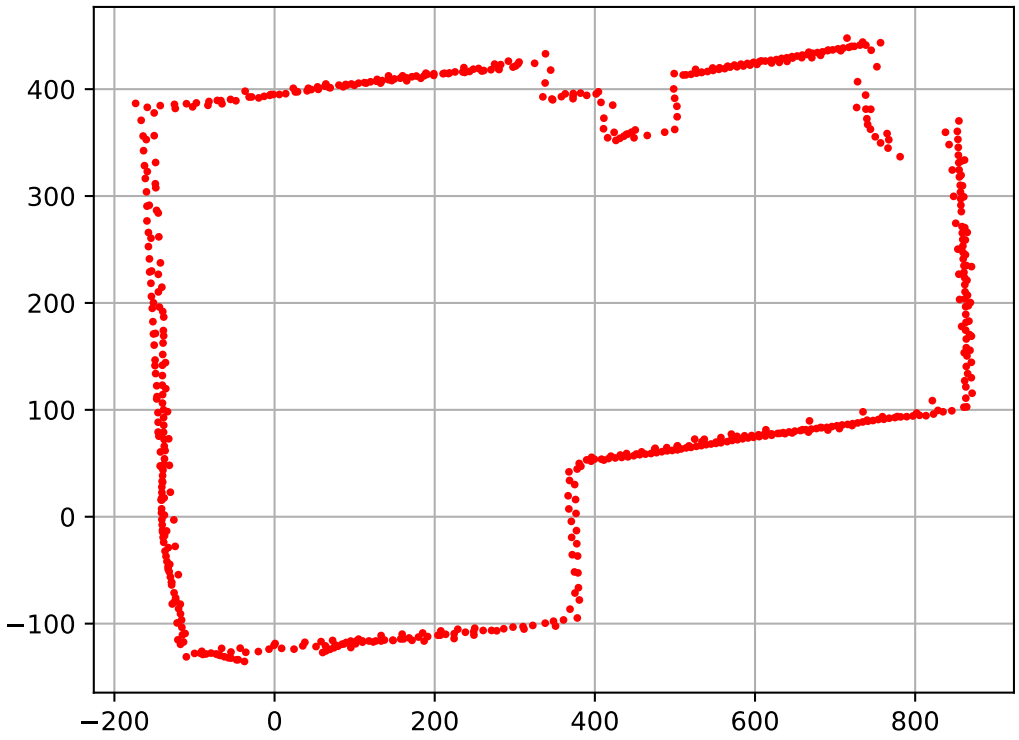
Pokušamo li prikupljene podatke preklopiti samo na temelju mjerenja, odnosno simulacije odometrije dobivamo sljedeći rezultat:



Slika 3.14: Preklopljeni prikupljeni podaci Lidar senzorom

Kao što je vidljivo na slici 3.14 podaci nisu u potpunosti točno preklopljeni te postoje određena odstupanja i točke na određenim dijelovima izgledaju kao da su rasipane no generalna slika podsjeća na sam oblik poligona. Valja napomenuti kako je ovo slučaj preklapanja s vrlo točnim podacima odometrije te kako u realnoj uporabi odometrija često ima puno veće greške.

Ponovimo li preklapanje točaka ali koristeći iterativni algoritam najbliže točke dobivamo sljedeće:



Slika 3.15: Preklopljeni prikupljeni podaci Lidar senzorom s iterativnim algoritmom najbliže točke

Na slici 3.15 je vidljivo kako su preklopljeni podaci u potpunosti poprimili oblik poligona, postavljeni predmeti su također poprimili ispravni oblik te greške u preklapanju koje su bile vidljive na slici 3.14 su nestale. Ishodište koordinatnog sustava predstavlja početnu poziciju s koje je Lidar senzor prikupio podatke. Vrijeme preklapanja točaka je relativno malo te u prosjeku iznosi 0,0109 s, odnosno 10,95 ms što je i više nego dovoljno brzo za preklapanje točaka u „realnom vremenu". Samo vrijeme izvršavanja algoritma nije povezano s veličinom skeniranog poligona već ovisi samo o broju prikupljenih točaka koje je Lidar senzor prikupio u prethodnom i najnovijem mjerenju što znači da ovakav SLAM sustav može bez problema stvarati karte velikih prostora s puno značajki bez opasnosti od prevelike upotrebe računalnih resursa.

Kao što je i ranije napomenuto ovako točno preklapanje bez ikakvih ispravaka kao što je vidljivo na slici 3.14 nije realno zbog greške odometrije. Iz navedenog razloga uvodimo namjernu greške u matrice homogenih transformacija koje smo definirali u poglavlju 3.2.. Tako matrica T_{12} poprima oblik

$$T_{12} = \begin{bmatrix} \cos(-35°) & -\sin(-35°) & 8,604 \\ \sin(-35°) & \cos(-35°) & 12,287 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Odnosno Lidar senzor je pomaknut s točke 1 u točku 2 tako da je zakrenut udesno za 35°, odnosno -35°, umjesto za -45° te je pomaknut unaprijed za 15 cm, umjesto 27,3 cm.

Matrica T_{23} poprima oblik

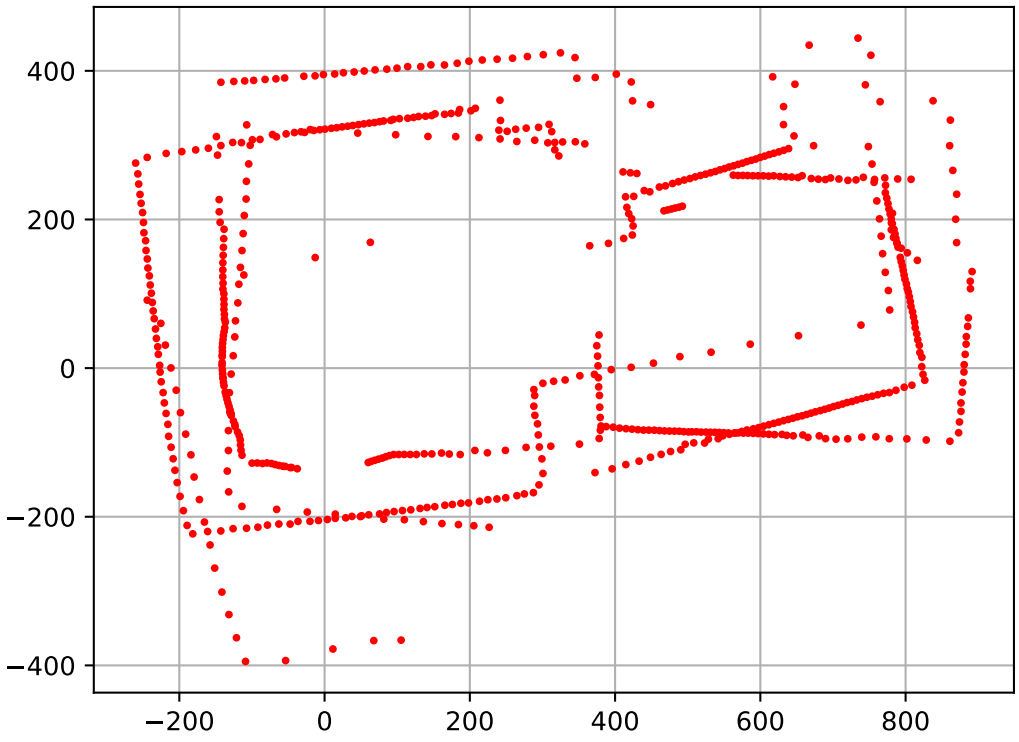
$$T_{23} = \begin{bmatrix} \cos(-60°) & -\sin(-60°) & 38,97 \\ \sin(-60°) & \cos(-60°) & 22,5 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

Odnosno Lidar senzor je pomaknut s točke 2 u točku 3 tako da je zakrenut udesno za 60°, odnosno -60°, umjesto za -45° te je pomaknut unaprijed za 45 cm, umjesto 32,5 cm.

Matrica T_{34} poprima oblik

$$T_{34} = \begin{bmatrix} \cos(20°) & -\sin(20°) & -3,42 \\ \sin(20°) & \cos(20°) & 9,397 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

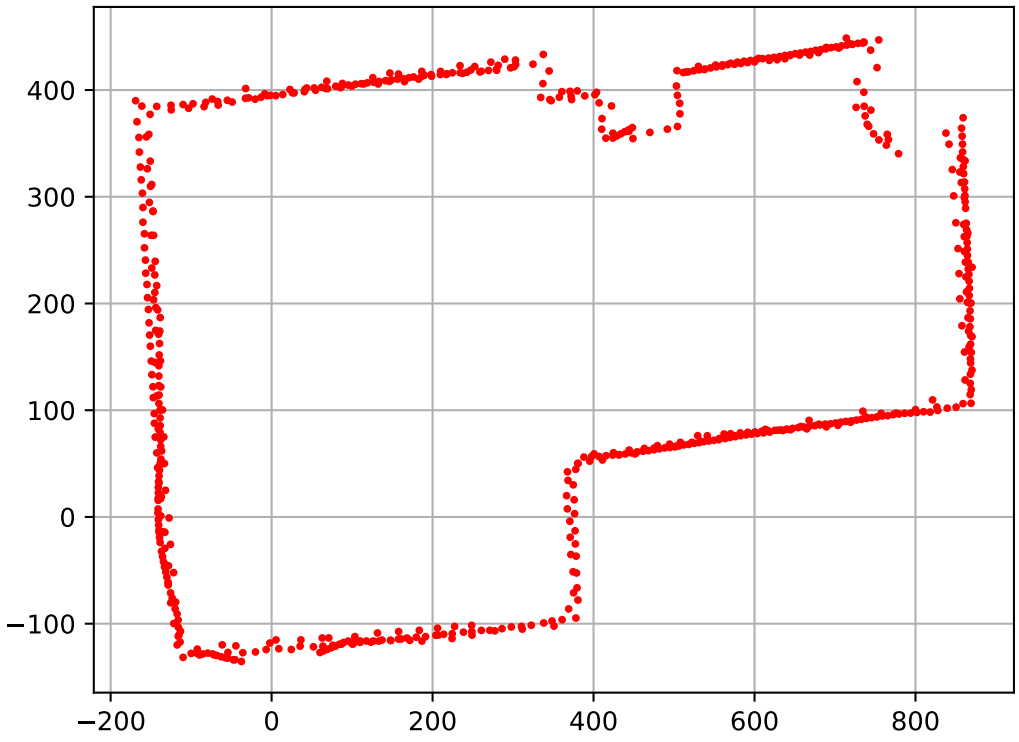
Odnosno Lidar senzor je pomaknut s točke 3 u točku 4 tako da je zakrenut ulijevo za 20°, umjesto da je zadržao istu orijentaciju te je pomaknut unaprijed za 10 cm, umjesto 18,8 cm.



Slika 3.16: Preklopljeni prikupljeni podaci Lidar senzorom s namjerno unesenom greškom

Na slici 3.16 je vidljivo kako preklapanje Lidar podataka senzora bez ispravaka odstupanja s velikom greškom odometrije rezultira fizički nemogućim oblikom poligona te nastala karta poligona ne odgovara stvarnom obliku poligona. Bitno je napomenuti kako ovako velika greška odometrije nije realna te služi samo za svrhu demonstracije važnosti iterativnog algoritma najbliže točke.

Ispravimo li podatke sa slike 3.16 uz pomoć iterativnog algoritma najbliže točke dobivamo sljedeće:



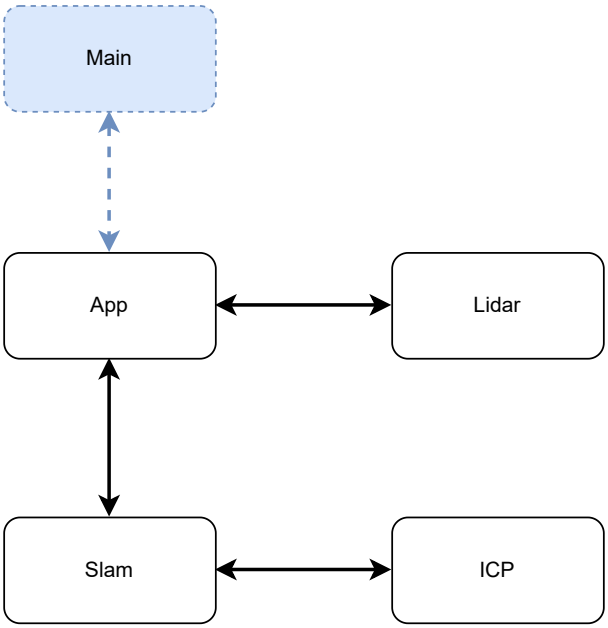
Slika 3.17: Preklopljeni prikupljeni podaci Lidar senzorom s iterativnim algoritmom najbliže točke

Na slici 3.17 je vidljivo kako ispravljeni preklopljeni Lidar podaci imaju pravilan oblik poligona te kako je algoritam uspio ispraviti podatke s čak ovoliko velikom greškom odometrije. Ovoliko velika greška odometrije nije realna no ona je dokaz kako je implementacija algoritma ispravna te kako algoritam funkcionira vrlo dobro.

Poglavlje 4.

Arhitektura SLAM sustava

Sama arhitektura sustava nastala je s ciljem kako bi SLAM sustav bio što više fleksibilan, odnosno kako bi se mogao implementirati i na druge mobilne robote bez prevelikih izmjena. Slam sustav se sastoji od četiri podsustava odnosno programa: Main, App, Lidar, Slam te ICP podsustav. Lidar podsustav, odnosno Lidar programska klasa te način prikupljanje podataka uz pomoć Lidar senzora i pretvorba prikupljenih podataka u oblak točaka već su objašnjeni u poglavlju 2.1.2..



Slika 4.1: Arhitektura SLAM sustava

Na slici 4.1 vizualizirana je arhitektura SLAM sustava te način komunikacije podsustava.

4.1. ICP potprogram

ICP je potprogram u kojem se nalazi iterativni algoritam najbliže točke. Sam potprogram se upotrebljava kako bi se zadnja dva prikupljena oblaka točaka poravnala, odnosno kako bi se kompenzirala greška odometrije.
ICP potprogram komunicira sa Slam potprogramom tako da Slam potprogram poziva *icp* metodu unutar ICP potprograma. Pozvana metoda vraća matricu transformacija koja minimizira odstupanje referentnog od fiksnog oblaka točaka.
Princip rada iterativnog algoritam najbliže točke, odnosno ICP potprograma već je objašnjen u poglavlju 3.1..

4.2. Slam potprogram

Unutar slam potprograma nalazi se programska klasa Slam. Potprogram služi kao neka vrsta komunikacijskog kanala između App potprograma te ICP potprograma. Njegova uloga je prihvaćanje novih skeniranih točaka, odnosno oblaka točaka, te proračun ukupne matrice homogenih transformacija.
Ukupna matrica homogenih transformacija u sebi sadrži sveukupnu transformaciju (translaciju i rotaciju) koju je mobilni robot napravio od početka rada, odnosno od pokretanja SLAM sustava. Slam podsustav također sprema sve poravnate točke koje je Lidar senzor tijekom rada prikupio te isto tako sadrži i „naivnu" sveukupnu matricu homogenih transformacija i točke.
„Naivna" matrica homogenih transformacija odnosno „naivne" prikupljene točke su podaci koji nisu ispravljeni s ICP potprogramom, tj. iterativnim algoritmom najbliže točke te se koriste za prikaz kontrasta između ispravljenih i neispravljenih podataka.

Slam potprogram u sebi također sadrži i pomoćne metode koje se koriste za pretvorbu koordinata iz kartezijevog koordinatnog sustava u sustav homogenih koordinata te obratno.

Slam potprogram, odnosno programska klasa Slam se sastoji od sljedećih metoda: *homogeneous_to_array*, *array_to_homogeneous*, *apply_transformation*, *add_new_scan*, *calculate_transformation*, *transformation_is_zero* ,*get_points*, *get_naive_points*, *get_current_position*, *get_current_naive_position*, *get_current_orientation*, te *get_current_naive_orientation*.

4.2.1. Metoda *homogeneous_to_array*

Metoda *homogeneous_coordinates* kao ulazni parametar prima listu koordinata u sustavu homogenih koordinata te vrši pretvorbu u listu kartezijevih koordinata.

```
procedure HOMOGENEOUS_TO_ARRAY(homogeneous_coordinates)
    array ← []
    array[:, 0] ← homogeneous_coordinates[0, :]
    array[:, 1] ← homogeneous_coordinates[1, :]
    return array
end procedure
```

4.2.2. Metoda *array_to_homogeneous*

Metoda *array_to_homogeneous* kao ulazni parametar prima listu koordinata u kartezijevom koordinatnom sustavu te vrši pretvorbu u listu koordinate u sustavu homogenih koordinata.

```
procedure ARRAY_TO_HOMOGENEOUS(array_coordinates)
    homogeneous ← []
    homogeneous[:, 0 : 2] ← array_coordinates
    homogeneous ← Transpose(homogeneous)
    return homogeneous
end procedure
```

4.2.3. Metoda *apply_transformation*

Metoda *apply_transformation* kao ulazni parametar prima listu koordinata u kartezijevom koordinatnom sustavu i matricu homogenih transformacija te uz pomoć skalarnog umnoška transformira predanu listu koordinata.

procedure APPLY_TRANSFORMATION($points, transformation\_matrix$)
$homogeneus\_points \leftarrow array\_to\_homogeneou(points)$
$transformed\_homogeneus\_points$
$\leftarrow DotProduct(homogeneus\_points, ransformation\_matrix)$
$array\_points \leftarrow homogeneous\_to\_array(transformed\_homogeneus\_points)$
return aray_points
end procedure

4.2.4. Metoda *add_new_scan*

Metoda *add_new_scan* koristi se za dodavanje novog oblaka točaka u dosad prikupljene točke. Metoda kao ulazne parametre prima listu koordinata u kartezijevom koordinatnom sustavu prikupljenih uz pomoć Lidar potprograma te matricu homogenih transformacija dobivenu uz pomoć odometrije robota.
Ako se ova metoda poziva prvi put odnosno ukoliko je broj dosad prikupljenih točaka jednak nuli tada predana lista koordinata postaje referentni i fiksni oblak točaka te se sprema kao prethodno prikupljeni oblak točaka. Isto tako predane točke se spremaju kao i „naivne" prikupljene točke, odnosno točke koje se prikupljene i preklopljene bez kompenzacije pogreške odometrije.
U suprotnom se poziva metoda *calculate_transformation* te se predana lista koordinata odnosno oblak točaka preklapa s prethodno predanim oblakom točaka.
Matrica homogenih transformacija koja se predaje kao parametar metodi određuje se na temelju podataka odometrije te predstavlja transformaciju između posljednja dva oblaka točaka, odnosno između referentnog i fiksnog oblaka točaka.

```
procedure ADD_NEW_SCAN(points, transformation_matrix)
    if count(self.points) is 0 then
        self.points ← points
        self.naive_points ← points
    else if count(self.previous_points) is 0 then
        self.previous_points ← points
    else
        calculate_transformation(points, transformation_matrix)
    end if
end procedure
```

4.2.5. Metoda *calculate_transformation*

Metoda *calculate_transformation* je zapravo srž samog Slam sustava. Metoda prima novi oblak točaka koji je Lidar klasa prikupila uz pomoć Lidar senzora i obradila te matricu homogenih transformacija koja sadrži prijeđeni put i kutni zakret koji se dogodio od prošlog poziva metode *calculate_transformation* prikupljenu uz pomoć odometrije robota.
Ako prikupljeni oblak točaka sadrži manje od tri točke tada se oblak točaka odbacuje iz razloga jer se smatra da Lidar senzor nije uspio prikupiti dovoljno podataka te se položaj mobilnog robota estimira uz pomoć odometrije bez ispravke greške uz pomoć Lidar senzora.
Jedna od mogućnosti zbog čega Lidar senzor nije uspio prikupiti podatke je ta da se mobilni robot nalazi u velikoj prostoriji ili u otvorenom te oko robota ne postoji niti jedan objekt u radnom radijusu Lidar senzora, u slučaju ovog Lidar senzora 12 m.
Rad robota u načinu rada gdje se podaci ne ispravljaju s Lidar senzorom je moguć no ne u velikom vremenskom intervalu jer će akumulirana pogreška odometrije nakon nekog vremena postati prevelika te iterativni algoritam najbliže točke više neće moći ispravno preklopiti referentni i fiksni oblak točaka.

Kako prikupljeni oblak točaka sadrži manje od 3 točke potrebno je ažurirati ukupnu matricu homogenih transformacija uz pomoć skalarnog produkta predane matrice homogenih transformacija i postojeće sveukupne matrice homogene transformacije. Isti postupak se ponavlja i za matricu „naivnih" transformacija. Također potrebno je obrisati sve točke koje su spremljene kao prethodni oblak točaka jer više nemamo nikakve Lidar podatke te se zapravo postupak preklapanja referentnog i fiksnog oblaka točaka resetira. Odnosno, kada Lidar krene opet prikupljati podatke ne možemo novi oblak točaka pokušati preklopiti s prethodnim jer podaci odometrije mjere pomak i zakret između prethodnog i trenutnog oblaka točaka, a ne od zadnjeg ispravnog te trenutnog oblaka točaka.

Ako prikupljeni oblak točaka sadrži više ili tri točke tada se prvo uz pomoć podataka odometrije proračunava „naivna" matrica sveukupnih homogenih transformacija. Nakon toga se novo prikupljeni oblak točaka transformira uz pomoć matrice transformacije

koja je generirana iz podataka odometrije te se potom transformirani oblak točaka dodaje na već prikupljene „naivne" točke.
Postupak je sličan i za referentni oblak točaka koji se preklapa s fiksnim oblakom točaka. Prvo se prikupljeni referentni oblak točaka poravnava s prošlim oblakom točaka uz pomoć algoritma iterativne najbliže točke i podataka odometrije. Po završetku izvršavanja algoritma algoritam vraća matricu homogenih transformacija koja odgovara transformaciji potrebnoj da se novi oblak točaka preklopi s prošlim oblakom točaka, odnosno transformaciju koju je mobilni robot napravio između prošlog i novog oblaka točaka.
U idealnom sustavu odometrije koji radi bez greške matrica homogenih transformacija generirana iz podataka odometrije bi bila jednaka matrici homogenih transformacija koju algoritam vraća.

Po završetku izvršavanja algoritma nova matrica homogenih transformacija se skalarno množi sa sveukupnom matricom homogenih transformacija kako bi dobili novu matricu sveukupnih homogenih transformacija te se novo prikupljeni oblak točaka transformira s novom matricom sveukupnih homogenih transformacija. Nakon toga se transformirani oblak točaka nadodaje na prethodno prikupljene oblake točaka te se dobiva nova mapa prostora.
Na kraju se novi predani oblak točaka sprema kao prošli oblak točaka kako bi se u sljedećem pozivu metode on koristio kao prošli oblak točaka.

```
procedure ADD_NEW_SCAN(new_points, transformation_matrix)
    if count(new_points) < 3 then
        self.current_transformation
← DotProduct(self.current_transformation, transformation_matrix)
        self.current_naive_transformation
← DotProduct(self.current_naive_transformation, transformation_matrix)
        self.previous_points ← []
        return
    else
        self.current_naive_transformation
← DotProduct(self.current_naive_transformation, transformation_matrix)
        naive_transformed_points
← apply_transformation(new_points, transformation_matrix)
        self.naive_points
← Concatenate(self.naive_points, naive_transformed_points)

        calculated_current_transformation ← None
        if transformation_is_zero(transformation_matrix) is True then
            calculated_current_transformation ← transformation_matrix
        else
            calculated_current_transformation
            ← icp(new_points, self.previous_points, transformation_matrix)
        end if
        self.current_transformation
← DotProduct(self.current_transformation, calculated_current_transformation)
        transformed_points
← apply_transformation(new_points, self.current_transformation)
        self.points ← Concatenate(self.points, transformed_points)

        self.previous_points ← new_points
    end if
end procedure
```

4.2.6. Metoda *get_points*

Metoda *get_points* vraća listu koordinata u kartezijevom koordinatnom sustavu od dosad prikupljenih i poravnatih oblaka točaka. Odnosno, ova metoda vraća kartu prostora oko mobilnog robota. Primjer karte koju ova metoda vraća, odnosno oblaka točaka je vidljiv na slici 3.17.

```
procedure GET_POINTS
    return self.points
end procedure
```

4.2.7. Metoda *get_naive_points*

Metoda *get_naive_points* funkcionira na istom principu kao i metoda *get_points* no za razliku od metode *get_points* metoda *get_naive_points* vraća listu koordinata u kartezijevom koordinatnom sustavu od dosad prikupljenih oblaka točaka koji nisu poravnati, odnosno ispravljeni s iterativnim algoritmom najbliže točke te se greška odometrije ne kompenzira. Primjer karte koju ova metoda vraća, odnosno oblaka točaka je vidljiv na slici 3.16.

```
procedure GET_NAIVE_POINTS
    return self.naive_points
end procedure
```

4.2.8. Metoda *get_current_position*

Metoda *get_current_position* vraća koordinate u kartezijevom koordinatnom sustavu od trenutne ispravljene pozicije robota. S obzirom kako znamo da je matrica homogenih transformacija definirana na sljedeći način

$$T = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & x \\ \sin(\alpha) & \cos(\alpha) & y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

vidljivo je da su x i y koordinate robota definirane ukoliko znamo matricu sveukupnih homogenih transformacija. U poglavlju 4.2.5. objašnjeno je na koji način se matrica sveukupnih homogenih transformacija računa.

```
procedure GET_CURRENT_POSITION
    return [self.current_transformation[0, 2], self.current_transformation[1, 2]]
end procedure
```

4.2.9. Metoda *get_current_naive_position*

Metoda *get_current_naive_position* radi na istom principu kao i metoda *get_current_position* no za razliku od metode *get_current_position* ova metoda vraća trenutnu poziciju robota koja nije ispravljena uz pomoć iterativnog algoritma najbliže točke te ne kompenzira grešku odometrije.

```
procedure GET_CURRENT_NAIVE_POSITION
    return [
            self.current_naive_transformation[0, 2],
            self.current_naive_transformation[1, 2]
      ]
end procedure
```

4.2.10. Metoda *get_current_orientation*

Metoda *get_current_orientation* vraća trenutnu ispravljenu orijentaciju mobilnog robota u smjeru osi y. S obzirom kako znamo da je matrica homogenih transformacija definirana na sljedeći način

$$T = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & x \\ \sin(\alpha) & \cos(\alpha) & y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

vidljivo je ako znamo matricu sveukupnu homogenih transformacija da se kut α može izračunat na sljedeći način

$$\alpha = atan2(T[1,0], T[0,0]) \tag{4.3}$$

gdje je $T[1,0]$ vrijednost matrice T na prvom retku i nultom stupcu, a $T[0,0]$ vrijednost matrice T na nultom retku i nultom stupcu.

```
procedure GET_CURRENT_ORIENTATION
    return degrees(atan2(
                self.current_transformation[1,0],
                self.current_transformation[0,0])
            )
end procedure
```

4.2.11. Metoda *get_current_naive_orientation*

Metoda *get_current_naive_orientation* radi na istom principu kao i metoda *get_current_orientation* no za razliku od metode *get_current_orientation* ova metoda vraća trenutnu orijentaciju robota koja nije ispravljena uz pomoć iterativnog algoritma najbliže točke te ne kompenzira grešku odometrije.

```
procedure GET_CURRENT_NAIVE_ORIENTATION
    return degrees(atan2(
                    self.current_naive_transformation[1, 0],
                    self.current_naive_transformation[0, 0])
                )
end procedure
```

4.3. App potprogram

App potprogram, odnosno programska klasa App je zapravo glavni potprogram Slam sustava koji ujedinjuje sve potprograme u jednu cjelinu te upravlja s njima. Također App potprogram je neka vrsta komunikacijskog kanala između Slam sustava te sustava koji ga koristi, odnosno implementira.
Programska klasa App se sastoji od sljedećih metoda: *start*, *on_close*, *configure_plot*, *animate*, *slam_routine*, metode koja inicijalizira samu klasu te pomoćne metode *odometry_to_transfomration*.

4.3.1. Inicijalizacija programske klase App

Inicijalizaciji programske klase App prihvaća četiri parametra od koja su tri opcionalna. Obavezani parametar je metoda koja se poziva za prikupljanje podataka odometrije, odnosno matrice homogenih transformacija. Ostala tri opcionalna parametra su port na kojem se nalazi Lidar senzor, varijabla koja određuje hoće li se vizualizirati i naivni podaci Slam sustava odnosno podaci koji nisu obrađeni i ispravljeni te metoda koja se poziva za prikupljanje Lidar podataka.
Ako se parametar *draw_naive_data* postavi na vrijednost $True$ tada će App potprogram prikazivati ispravljene podatke Lidar senzora u crvenoj boji, orijentaciju i poziciju robota te ne ispravljene podatke u sivoj boji s ne ispravljenom orijentacijom i pozicijom robota.

Slika 4.2: Vizualizacija App potprograma s neispravljenim podacima

Ako se parametar *draw_naive_data* postavi na vrijednost $False$ tada će App potprogram prikazivati samo ispravljene podatke Lidar senzora te poziciju i orijentaciju robota.

Slika 4.3: Vizualizacija App potprograma bez neispravljenih podataka

Sama inicijalizacija je napravljena tako da App potprogram bude fleksibilan te da olakša daljnji razvoj programa. Iz tog razloga je moguće predati i metodu koja vraća Lidar podatke senzora te tada Slam podsustav neće koristiti Lidar klasu za prikupljanje i obradu Lidar podataka već će se pozivati proslijeđena metoda.
To je napravljeno iz razloga kako bi se omogućio razvojni način rada programa, odnosno kako bi se program mogao testirati i razvijati bez spojenog Lidar senzora. Tako je moguće očitavati vrijednosti Lidar senzora iz datoteke koja je ranije generirana, odnosno u koju su ranije spremljene vrijednosti Lidar senzora.

4.3.2. Metoda *start*

Metoda *start* se koristi za pokretanje Slam sustava. Sama metoda inicijalizira Lidar klasu i pokreće prikupljanje Lidar podataka, ako je pri inicijalizaciji klase App predan opcionalan parametar metode koja se poziva za prikupljanje Lidar podataka tada se Lidar klasa neće inicijalizirati već će se Lidar podaci prikupljati preko predane metode.

Isto tako ova metoda kreira novu dretvu koja se koristi za paralelizaciju procesa slam rutine te u novoj dretvi pokreće metodu slam_routine.

Također metoda otvara i novi programski prozor u kojem se prikazuju rezultati Slam sustava u realnom vremenu.

Slika 4.4: Programski prozor Slam sustava

```
procedure START
    if self.get_lidar_data_method is not None then
        StartCollectingLidarData()
    end if
    CreateNewThreadAndRun(slam_routine)
    CreateAndShowNewWindow()
end procedure
```

4.3.3. Metoda *on_close*

Metoda *on_close* se poziva prilikom zatvaranja programskog prozora Slam sustava te je njezina jedina uloga zaustavljanje prikupljanja podataka Lidar senzora.

4.3.4. Metoda *configure_plot*

Metoda *configure_plot* koristi se za konfiguraciju programskog prozora Slam sustava i koordinatnog sustava u kojem se crtaju Lidar podaci te pozicija i orijentacija mobilnog robota.

4.3.5. Metoda *animate*

Metoda *animate* koristi se za vizualizaciju Slam podataka. Metoda se poziva svaki put prilikom osvježavanja programskog prozora Slam sustava.

4.3.6. Metoda *slam_routine*

Metoda *slam_routine* se poziva na zasebnoj dretvi radi paralelizacije procesorski zahtjevnih operacija koje vrši instanca Slam klase koja se koristi unutar ove metode. Metoda prikuplja podatke odometrije te podatke Lidar senzora, prikupljene podatke prosljeđuje instanci Slam klase te nakon toga čeka zadanu količinu vremena prije nego se cijeli postupak ponovi.

```
procedure SLAM_ROUTINE
    while self.is_running is not True do
        new_points ← []
        odometry ← self.get_odometry_data_method()

        if self.get_lidar_data_method is None then
            new_points ← self.lidar_instance.get_coordinates()
        else
            new_points ← self.lidar_instance.get_lidar_data_method()
        end if

        if count(new_points) is 0 or count(odometry) is 0 then
            return
        end if

        self.slam_instance.add_new_scan(new_points, odometry)
        sleep(self.SLAM_TIMEOUT)
    end while
end procedure
```

4.3.7. Metoda *odometry_to_transfomration*

Metoda *odometry_to_transfomration* je pomoćna metoda koja se ne koristi unutar klase App već je zamišljena kao metoda koja se poziva iz programa koji implementira Slam sustav, u ovom slučaju main i main_debug potprogram.
Metoda prima dva ulazna parametra, udaljenost u milimetrima i kutni zakret u stupnjevima, odnosno podatke odometrije mobilnog robota te potom predane vrijednosti preračunava u matricu homogenih transformacija.

4.4. Main_debug potprogram

Main_debug potprogram zapravo i nije dio Slam sustava već primjer upotrebe sustava. U Main_debug potprogramu vidljivo je kako se implementira opisani Slam sustav te su vidljive metode koje se pozivaju za prikupljanje podataka Lidar senzora te odometrije robota.
Kao što je već opisano u poglavlju 4.3.1. prilikom inicijalizacije App objekta, odnosno klase moguće je predati i metodu koja vraća Lidar podatke. U ovom primjeru Main potprograma vidljivo je kako su Lidar podaci unaprijed prikupljeni te spremljeni u datoteke. Kreirana je metoda *mock_lidar* koja iterira po unaprijed zadanim putanjama do datoteka u kojima se nalaze spremljeni podaci te umjesto Lidar senzora ona vraća prethodno prikupljene Lidar podatke. Po sličnom principu funkcionira i metoda *mock_odometry* koja vraća lažne podatke odometrije odnosno podatke odometrije koji su uneseni u program ručno te simulirani.

Ovaj potprogram je razvijen kako bi se olakšao daljnji razvoj Slam sustava, uklanjanje pogrešaka u radu sustava te omogućilo simuliranje SLAM sustava bez robota. Sama statična okolina, odnosno statični podaci koji se očitavaju iz programa i datoteka korisni su jer se program može testirati na identičnom setu ulaznih podataka te se time otklanja mogućnost u devijacijama rezultata Slam sustava radi testiranja na nekonzistentnim ulaznim podacima, odnosno različitim okolinama.

4.5. Main potprogram

Main potprogram koristi se na sličan način kao ranije opisani Main_debug potprogram u poglavlju 4.4. te je jedina razlika u tome što ovaj potprogram ne koristi simulirane i ranije prikupljene podatke već komunicira s eMiR robotom i Lidar senzorom te prikuplja stvarne podatke uz pomoć senzora.
Ovaj potprogram se koristi kako bi se SLAM sustav povezao u jednu cjelinu zajedno s eMiR robotom.

Poglavlje 5.

Implementacija SLAM sustava na eMiR robota

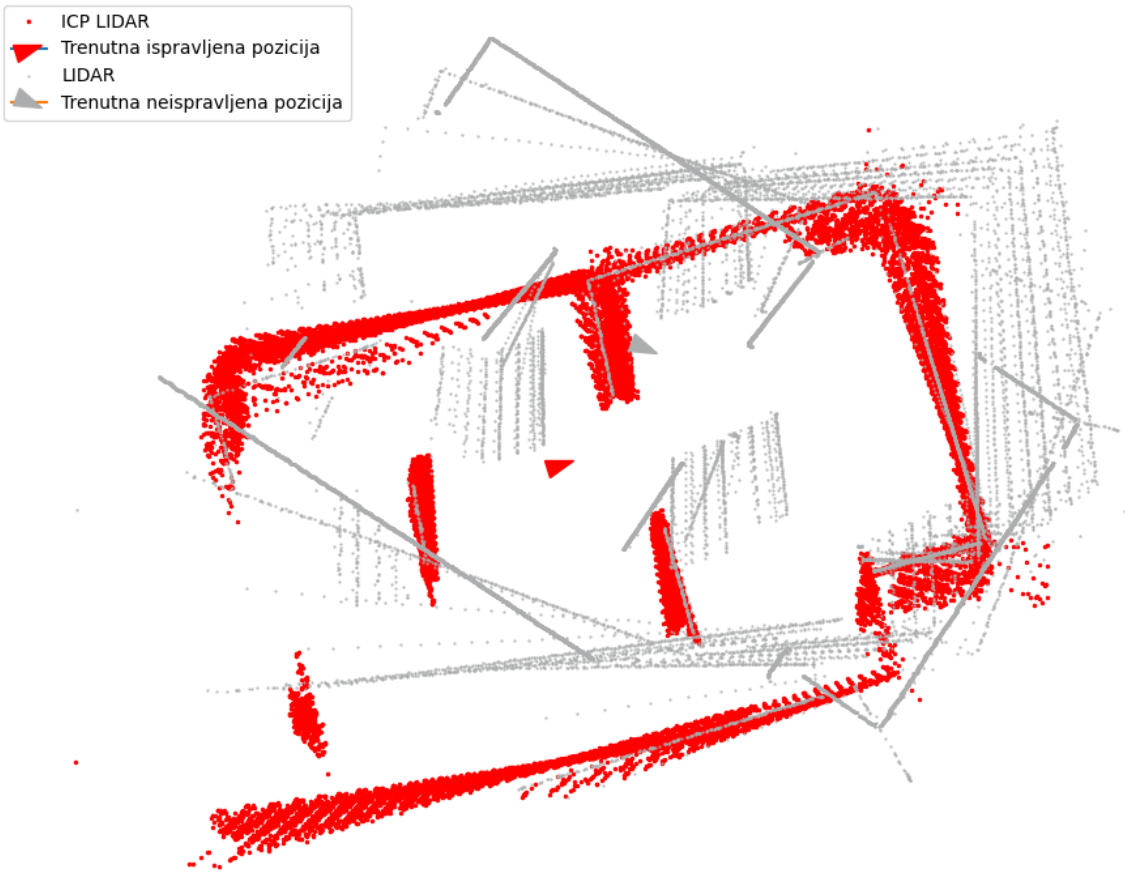
Kako bi se dokazala ispravna funkcionalnost u radu opisanog SLAM sustava napravljen je test SLAM sustava u realnim uvjetima na eMiR robotu.

Prije puštanja SLAM sustava u rad bilo je potrebno napraviti novu Python skriptu (eMiRcon.py) koja se korisit za komunikaciju s eMiR robotom na temelju postojeće ali zastarjele skripte.

Nakon izrade skripte za komunikaciju odabrani Lidar senzor postavljen je na vrh eMiR robota te je sam robot postavljen u poligon fakulteta te je SLAM sustav pokrenut.

Slika 5.1: Lidar senzor RPLiDAR A2M8 postavljen na eMiR robota

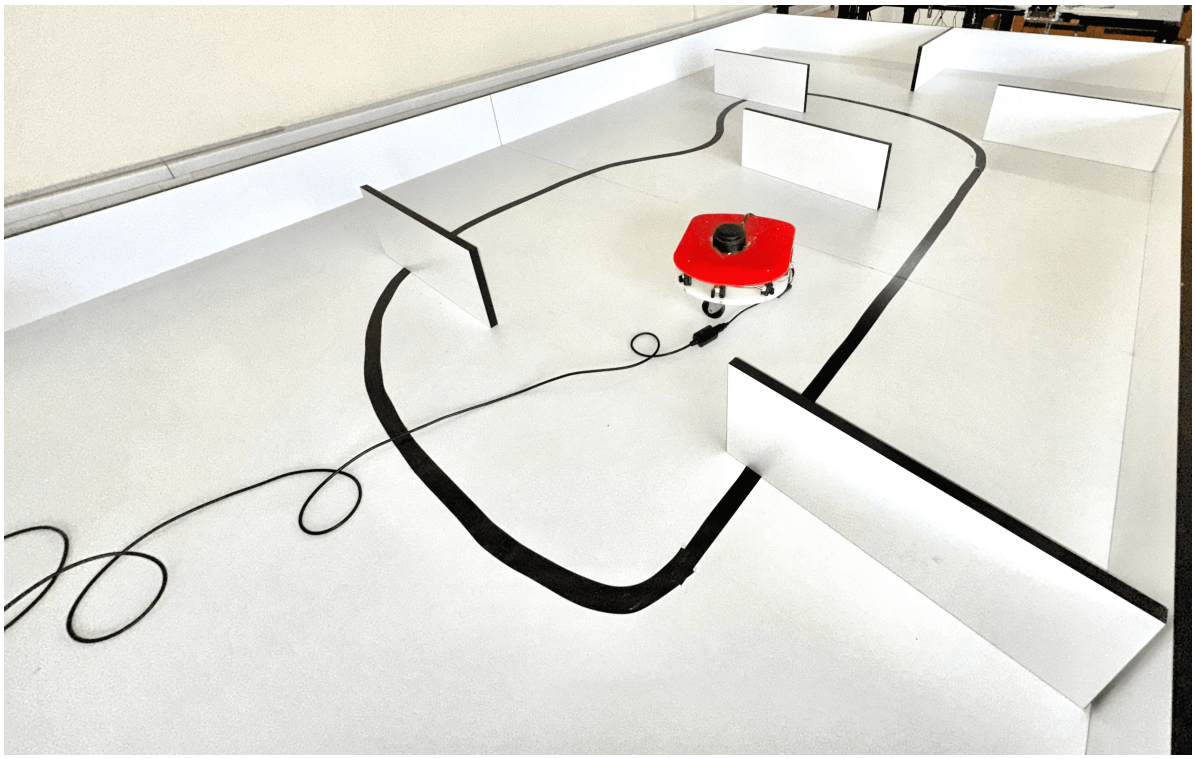
Uz pomoć ranije napisane skripte za komunikaciju s robotom eMiR robot je povezan sa SLAM sustavom te je upravljan tako da napravi krug oko prepreka u poligonu. Nakon određenog vremena akumulirana greška odometrije iz ranije opisanih razloga je značajno narasla te se jasno vidi kako SLAM sustav koji bi se oslanja samo na odometriji bez iterativnog algoritma najbliže točke ne bi ispravno radio.



Slika 5.2: Rezultat testiranja SLAM sustava na eMiR robotu

Na slici 5.2 vidljivo je kako crveno označeni rezultat SLAM sustava odgovara stvarnom obliku poligona te stvarnoj poziciji robota na poligonu. Na slici 5.3 možemo vidjeti poziciju robota u poligonu u trenutku spremanja rezultata SLAM sustava sa slike 5.2.

Iz navedenog možemo zaključiti kako SLAM sustav koji je izrađen u sklopu ovog rada ispravno funkcionira.



Slika 5.3: Stvarna pozicija eMiR robota u poligonu

Poglavlje 6.

Zaključak

U ovom radu prikazan je postupak izrade SLAM sustav uz pomoć 2D Lidar senzora. Izrađen je sustav za prikupljanje Lidar podataka uz pomoć 2D Lidar senzora te obradu prikupljenih podataka. Napravljen je potprogram za ispravak prikupljenih podataka uz pomoć podataka odometrije te su svi potprogrami povezani u jedan cjeloviti SLAM sustav.
Izrađeni SLAM sustav napravljen je da bude što apstraktniji, odnosno da je uz malu modifikaciju primjenjiv s bilo kojim 2D Lidar senzorom te mobilnim robotom.

Izrađen je poseban način rada koji omogućuje simuliranje kretanja robota uz pomoć unaprijed prikupljenih podataka te je na temelju prikupljenih podataka odometrije i Lidar senzora prikazano kako izrađeni Slam sustav ispravno funkcionira.
Cijeli sustav je implementiran na eMiR mobilnog robota te uspješno testiran, odnosno rezultati testiranja su pokazali kako izrađeni sustav ispravno funkcionira.

Jedan od nedostataka ovog SLAM sustava je taj što je ograničen s 2D Lidar senzorom te kao daljnji razvoj ovog SLAM sustava moguće je implementirati 3D Lidar senzor kako bi dobili što precizniju reprezentaciju okoline robota.
Isto tako jedan od sljedećih koraka razvoja ovog sustava je i implementacija samog sustava na ugrađenim mikroračunalima mobilnih robota ili implementacija bežičnog prikupljanja Lidar podataka uz pomoć mobilnog robota.
Jedna od mogućih dorada samog sustava prikupljanja podataka koja bi još više poboljšala točnost sustava je ažuriranje prikupljenih podataka, odnosno trenutna im-

plementacija sumira prikupljene točke te je moguće napraviti implementaciju koja bi ažurirala područja na karti koja bi mobilni robot ponovno obišao, odnosno Lidar senzor skenirao.

Literatura

[1] The MathWorks, Inc., SLAM (Simultaneous Localization and Mapping), Pristupljeno 9. rujna 2022., https://www.mathworks.com/discovery/slam.html

[2] Roboticia, RPLidar, Pristupljeno 25. srpnja 2022., https://github.com/Roboticia/RPLidar

[3] BOTLAND B. DERKACZ SP.K., Laser Scanner 360 Degree RPLidar A2M8, Pristupljeno 3. listopada 2022. https://botland.store/laser-scanners-lidar/7036-laser-scanner-360-degree-rplidar-a2m8-5904422361433.html

[4] Slamtec Co., Ltd., Documents, Pristupljeno 3. listopada 2022. https://www.slamtec.com/en/Support#rplidar-a-series

[5] ACM SIGMM Records, OpenVSLAM: A Versatile Visual SLAM Framework, Pristupljeno 15. listopada 2022. https://records.sigmm.org/?open-source-item=openvslam-a-versatile-visual-slam-framework

[6] Besl, Paul J.; N.D. McKay (1992). "A Method for Registration of 3-D Shapes". IEEE Transactions on Pattern Analysis and Machine Intelligence. 14 (2): 239–256.

[7] Clay Flannigan, icp, Pristupljeno 10. kolovoza 2022. https://github.com/ClayFlannigan/icp

7.

Prilog

7.1. Izvorni kod

7.1.1. lidar.py

```
from time import sleep
from rplidar import RPLidar
from threading import Thread
from math import sin, cos, radians
import numpy as np

class Lidar:
    lidar = None
    is_running = False
    points = []
    number_of_scans = 1

    def __init__(self, port_name, number_of_scans = 1):
        self.number_of_scans = number_of_scans
        self.lidar = RPLidar(port_name)



    def start(self):
```

```
        self.is_running = True
        Thread(target=self.scan, args=()).start()

    def stop(self):
        self.is_running = False

        self.lidar.stop()
        self.lidar.stop_motor()
        self.lidar.disconnect()

    def scan(self):
        while self.is_running:
            try:
                new_scan_points = []

                for i in range(0, self.number_of_scans):
                    for measurment in next(self.lidar.iter_scans()):
                        quality, angle, distance = measurment[0],
                            ↪ measurment[1], measurment[2]

                        radians_angle = radians(angle)
                        x = distance * sin(radians_angle)
                        y = distance * cos(radians_angle)

                        new_scan_points.append([x, y])

                self.points = np.array(new_scan_points)

                sleep(0.05) # 50ms
            except Exception as e:
                print("Lidar.py - Failed to read lidar data. Error: ", e)
                sleep(0.05)
```

```
    def get_coordinates(self):
        if self.is_running and len(self.points) == 0:
            self.scan()

        return self.points
```

7.1.2. slam.py

```
import icp
import numpy as np
from math import asin, degrees

class Slam:
    MAX_ICP_ITERATIONS = 500

    points = [] # Current merged points of all scans
    naive_points = []
    previous_points = []

    current_transformation = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    current_naive_transformation = np.array([[1, 0, 0], [0, 1, 0], [0,
        ↪ 0, 1]])



    def __init__(self, max_icp_iterations = 500):
        self.MAX_ICP_ITERATIONS = max_icp_iterations

    def homogeneous_to_array(self, homogeneous):
        array = np.ones((homogeneous.shape[1], 2))
        array[:, 0] = homogeneous[0, :]
        array[:, 1] = homogeneous[1, :]

        return array

    def array_to_homogeneous(self, array):
        homogeneous = np.ones((array.shape[0], 3))
        homogeneous[:, 0:2] = np.copy(array)

        return homogeneous.T
```

```
def apply_transformation(self, points, T):
    homogeneus = self.array_to_homogeneous(points)
    transformedHomogeneus = np.dot(T, homogeneus)
    arrayPoints = self.homogeneous_to_array(transformedHomogeneus)

    return arrayPoints

def add_new_scan(self, new_points, position = None):
    if len(self.points) == 0:
        self.points = new_points
        self.naive_points = new_points
    elif len(self.previous_points) == 0:
        self.previous_points = new_points
    else:
        self.calculate_transformation(new_points, position)

def calculate_transformation(self, new_points, position):
    if len(new_points) < 3 :
        self.current_transformation = np.dot(self.
            ↪ current_transformation, position)
        self.current_naive_transformation = np.dot(self.
            ↪ current_naive_transformation, position)
        self.previous_points = []

        return

    self.current_naive_transformation = np.dot(self.
        ↪ current_naive_transformation, position)
    naive_transformed_points = self.apply_transformation(new_points,
        ↪ self.current_naive_transformation)
    self.naive_points = np.concatenate((self.naive_points,
        ↪ naive_transformed_points))
```

```
        calculated_current_transformation = None

        if self.transformation_is_zero(position):
            calculated_current_transformation = position
        else:
            calculated_current_transformation = icp.icp(new_points, self.
                ↪ previous_points, init_pose = position, max_iterations =
                ↪  self.MAX_ICP_ITERATIONS)

        self.current_transformation = np.dot(self.current_transformation,
            ↪  calculated_current_transformation)
        transformed_points = self.apply_transformation(new_points, self.
            ↪ current_transformation)
        self.points = np.concatenate((self.points, transformed_points))

        self.previous_points = new_points

    def transformation_is_zero(self, transformation):
        return transformation.sum() == 0

    def get_points(self):
        return self.points

    def get_naive_points(self):
        return self.naive_points

    def get_current_position(self):
        return [self.current_transformation[0, 2], self.
            ↪ current_transformation[1, 2]]

    def get_current_naive_position(self):
```

```
        return [self.current_naive_transformation[0, 2], self.
            ↪ current_naive_transformation[1, 2]]

    def get_current_orientation(self):
        return degrees(np.arctan2(self.current_transformation[1,0], self.
            ↪ current_transformation[0,0]))

    def get_current_naive_orientation(self):
        return degrees(np.arctan2(self.current_naive_transformation[1,0],
            ↪  self.current_naive_transformation[0,0]))
```

7.1.3. app.py

```
from lidar import Lidar
from slam import Slam

import numpy as np

import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.markers as markers
import matplotlib as mpl

from threading import Thread
from time import sleep
from math import sin, cos, radians
from time import sleep
import pickle

class App:
    SLAM_TIMEOUT = 1 # Every 1 seconds
    # LIDAR_PORT = "COM13"
    ANIMATION_FPS = 1

    is_running = True
    lidar_instance = None
    slam_instance = Slam()

    figure = None
    axis = None
    animation_instance = None

    get_odometry_data_method = None
    get_lidar_data_method = None
```

```
draw_naive_data = True

def __init__(self, get_odometry_data_method, lidar_port = "",
    ↪ draw_naive_data = True, get_lidar_data_method = None):
    self.get_odometry_data_method = get_odometry_data_method
    self.get_lidar_data_method = get_lidar_data_method
    self.draw_naive_data = draw_naive_data
    self.LIDAR_PORT = lidar_port

    self.configure_plot()

def start(self):
    if self.get_lidar_data_method == None:
        # Start collecting lidar data
        self.lidar_instance = Lidar(port_name = self.LIDAR_PORT,
            ↪ number_of_scans = 1)
        self.lidar_instance.start()

    # Wait for Lidar
    sleep(5)
    # Start SLAM procedure
    Thread(target = self.slam_routine, args = ()).start()

    animation_instance = animation.FuncAnimation(self.figure, self.
        ↪ animate, interval = 1000 / self.ANIMATION_FPS)

    plt.show()

def on_close(self, _):
    self.is_running = False

    if self.lidar_instance != None:
        self.lidar_instance.stop()
```

```
def configure_plot(self):
    mpl.rcParams["toolbar"] = "None"
    self.figure = plt.figure("SLAM")
    self.figure.canvas.mpl_connect("close_event", self.on_close)
    self.axes = self.figure.add_subplot()

    plt.xticks([])
    plt.yticks([])
    plt.axis("equal")
    plt.subplots_adjust(left = 0, bottom = 0, right = 1, top = 1,
        ↪ wspace = 0, hspace = 0)

    x_scale = (self.axes.get_xlim()[1] - self.axes.get_xlim()[0]) / 2
    y_scale = (self.axes.get_ylim()[1] - self.axes.get_ylim()[0]) / 2

    current_position = self.slam_instance.get_current_position()

    plt.xlim([current_position[0] - x_scale, current_position[0] +
        ↪ x_scale])
    plt.ylim([current_position[1] - y_scale, current_position[1] +
        ↪ y_scale])

    self.figure.canvas.toolbar_visible = False

def animate(self, _):
    slam_points = self.slam_instance.get_points()

    if self.slam_instance == None or self.is_running == False or len(
        ↪ slam_points) == 0:
        return

    self.axes.clear()
```

```
slam_points = self.slam_instance.get_points()
current_position = self.slam_instance.get_current_position()

self.axes.plot(slam_points[:, 0], slam_points[:, 1], "r.", label
    ↪ = "ICP␣LIDAR", markersize = 3)

position_marker = markers.MarkerStyle(marker = "^")
position_marker._transform.scale(1, 1.5)
position_marker._transform.rotate_deg(self.slam_instance.
    ↪ get_current_orientation())

self.axes.plot(current_position[0], current_position[1], marker =
    ↪  position_marker, markersize = 10, markerfacecolor = "#
    ↪ FF0000", markeredgecolor = "#FF0000", label = "Trenutna␣
    ↪ ispravljena␣pozicija")

if self.draw_naive_data:
    naive_slam_points = self.slam_instance.get_naive_points()
    naive_position = self.slam_instance.
        ↪ get_current_naive_position()

    self.axes.plot(naive_slam_points[:, 0], naive_slam_points[:,
        ↪ 1], "b.", label = "LIDAR", markersize = 1,
        ↪ markerfacecolor = "#acadae", markeredgecolor = "#acadae
        ↪ ",)

    naive_position_marker = markers.MarkerStyle(marker = "^")
    naive_position_marker._transform.scale(1, 1.5)
    naive_position_marker._transform.rotate_deg(self.
        ↪ slam_instance.get_current_naive_orientation())
    self.axes.plot(naive_position[0], naive_position[1], marker =
        ↪  naive_position_marker, markersize = 10,
```

```
            ↪ markerfacecolor = "#acadae", markeredgecolor = "#acadae
            ↪ ", label = "Trenutna neispravljena pozicija")

        plt.legend()

    def slam_routine(self):
        while self.is_running:
            new_points = None
            odometry = self.get_odometry_data_method()

            if self.get_lidar_data_method == None:
                new_points = self.lidar_instance.get_coordinates()
            else:
                new_points = self.get_lidar_data_method()

            if len(new_points) == 0 or len(odometry) == 0:
                return

            self.slam_instance.add_new_scan(new_points, odometry)

            sleep(self.SLAM_TIMEOUT)

    def odometry_to_transfomration(distance, angle):
        radiansAngle = radians(angle)
        cosAngle = cos(radiansAngle)
        sinAngle = sin(radiansAngle)

        x = cos(radians(angle + 90)) * distance
        y = sin(radians(angle + 90)) * distance

        return np.array([[cosAngle, -sinAngle, x], [sinAngle, cosAngle, y
            ↪ ], [0, 0, 1]])
```

7.1.4. icp.py

```
import numpy as np
from sklearn.neighbors import NearestNeighbors


def best_fit_transform(A, B):
    m = A.shape[1]

    centroid_A = np.mean(A, axis=0)
    centroid_B = np.mean(B, axis=0)
    AA = A - centroid_A
    BB = B - centroid_B

    H = np.dot(AA.T, BB)
    U, S, Vt = np.linalg.svd(H)
    R = np.dot(Vt.T, U.T)

    if np.linalg.det(R) < 0:
       Vt[m-1,:] *= -1
       R = np.dot(Vt.T, U.T)

    t = centroid_B.T - np.dot(R,centroid_A.T)

    T = np.identity(m+1)
    T[:m, :m] = R
    T[:m, m] = t

    return T, R, t


def nearest_neighbor(src, dst):
    neigh = NearestNeighbors(n_neighbors=1)
```

```
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True)
    return distances.ravel(), indices.ravel()


def icp(A, B, init_pose=None, max_iterations=20, tolerance=0.001):
    m = B.shape[1]

    src = np.ones((m+1,A.shape[0]))
    dst = np.ones((m+1,B.shape[0]))
    src[:m,:] = np.copy(A.T)
    dst[:m,:] = np.copy(B.T)

    if init_pose is not None:
        src = np.dot(init_pose, src)

    prev_error = 0

    for i in range(max_iterations):
        distances, indices = nearest_neighbor(src[:m,:].T, dst[:m,:].T)

        T,_,_ = best_fit_transform(src[:m,:].T, dst[:m,indices].T)

        src = np.dot(T, src)

        mean_error = np.mean(distances)
        if np.abs(prev_error - mean_error) < tolerance:
            break
        prev_error = mean_error

    T,_,_ = best_fit_transform(A, src[:m,:].T)

    return T
```

7.1.5. app.py

```
from lidar import Lidar
from slam import Slam

import numpy as np

import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.markers as markers
import matplotlib as mpl

from threading import Thread
from time import sleep
from math import sin, cos, radians
from time import sleep
import pickle

class App:
    SLAM_TIMEOUT = 1 # Every 1 seconds
    # LIDAR_PORT = "COM13"
    ANIMATION_FPS = 1

    is_running = True
    lidar_instance = None
    slam_instance = Slam()

    figure = None
    axis = None
    animation_instance = None

    get_odometry_data_method = None
    get_lidar_data_method = None
```

```
draw_naive_data = True

def __init__(self, get_odometry_data_method, lidar_port = "",
    ↪ draw_naive_data = True, get_lidar_data_method = None):
    self.get_odometry_data_method = get_odometry_data_method
    self.get_lidar_data_method = get_lidar_data_method
    self.draw_naive_data = draw_naive_data
    self.LIDAR_PORT = lidar_port

    self.configure_plot()

def start(self):
    if self.get_lidar_data_method == None:
        # Start collecting lidar data
        self.lidar_instance = Lidar(port_name = self.LIDAR_PORT,
            ↪ number_of_scans = 1)
        self.lidar_instance.start()

    # Wait for Lidar
    sleep(5)
    # Start SLAM procedure
    Thread(target = self.slam_routine, args = ()).start()

    animation_instance = animation.FuncAnimation(self.figure, self.
        ↪ animate, interval = 1000 / self.ANIMATION_FPS)

    plt.show()

def on_close(self, _):
    self.is_running = False

    if self.lidar_instance != None:
        self.lidar_instance.stop()
```

```
def configure_plot(self):
    mpl.rcParams["toolbar"] = "None"
    self.figure = plt.figure("SLAM")
    self.figure.canvas.mpl_connect("close_event", self.on_close)
    self.axes = self.figure.add_subplot()

    plt.xticks([])
    plt.yticks([])
    plt.axis("equal")
    plt.subplots_adjust(left = 0, bottom = 0, right = 1, top = 1,
        ↪ wspace = 0, hspace = 0)

    x_scale = (self.axes.get_xlim()[1] - self.axes.get_xlim()[0]) / 2
    y_scale = (self.axes.get_ylim()[1] - self.axes.get_ylim()[0]) / 2

    current_position = self.slam_instance.get_current_position()

    plt.xlim([current_position[0] - x_scale, current_position[0] +
        ↪ x_scale])
    plt.ylim([current_position[1] - y_scale, current_position[1] +
        ↪ y_scale])

    self.figure.canvas.toolbar_visible = False

def animate(self, _):
    slam_points = self.slam_instance.get_points()

    if self.slam_instance == None or self.is_running == False or len(
        ↪ slam_points) == 0:
        return

    self.axes.clear()
```

```
slam_points = self.slam_instance.get_points()
current_position = self.slam_instance.get_current_position()

self.axes.plot(slam_points[:, 0], slam_points[:, 1], "r.", label
    ↪ = "ICP␣LIDAR", markersize = 3)

position_marker = markers.MarkerStyle(marker = "^")
position_marker._transform.scale(1, 1.5)
position_marker._transform.rotate_deg(self.slam_instance.
    ↪ get_current_orientation())

self.axes.plot(current_position[0], current_position[1], marker =
    ↪  position_marker, markersize = 10, markerfacecolor = "#
    ↪ FF0000", markeredgecolor = "#FF0000", label = "Trenutna␣
    ↪ ispravljena␣pozicija")

if self.draw_naive_data:
    naive_slam_points = self.slam_instance.get_naive_points()
    naive_position = self.slam_instance.
        ↪ get_current_naive_position()

    self.axes.plot(naive_slam_points[:, 0], naive_slam_points[:,
        ↪ 1], "b.", label = "LIDAR", markersize = 1,
        ↪ markerfacecolor = "#acadae", markeredgecolor = "#acadae
        ↪ ",)

    naive_position_marker = markers.MarkerStyle(marker = "^")
    naive_position_marker._transform.scale(1, 1.5)
    naive_position_marker._transform.rotate_deg(self.
        ↪ slam_instance.get_current_naive_orientation())
    self.axes.plot(naive_position[0], naive_position[1], marker =
        ↪  naive_position_marker, markersize = 10,
```

```
            ↪ markerfacecolor = "#acadae", markeredgecolor = "#acadae
            ↪ ", label = "Trenutna neispravljena pozicija")

    plt.legend()

def slam_routine(self):
    while self.is_running:
        new_points = None
        odometry = self.get_odometry_data_method()

        if self.get_lidar_data_method == None:
            new_points = self.lidar_instance.get_coordinates()
        else:
            new_points = self.get_lidar_data_method()

        if len(new_points) == 0 or len(odometry) == 0:
            return

        self.slam_instance.add_new_scan(new_points, odometry)

        sleep(self.SLAM_TIMEOUT)

def odometry_to_transfomration(distance, angle):
    radiansAngle = radians(angle)
    cosAngle = cos(radiansAngle)
    sinAngle = sin(radiansAngle)

    x = cos(radians(angle + 90)) * distance
    y = sin(radians(angle + 90)) * distance

    return np.array([[cosAngle, -sinAngle, x], [sinAngle, cosAngle, y
        ↪ ], [0, 0, 1]])
```

7.1.6. main_debug.py

```
from app import App
import pickle
import numpy as np
from math import cos, sin, pi
from pathlib import Path

mock_index = 0

def get_mock_data():
        mock_points = []
        mock_data_directory = Path(__file__).with_name("mock_data")

        mock_points.append(pickle.load(open(Path.joinpath(
            ↪ mock_data_directory, "1.p"), "rb"))) # Scan at start
        mock_points.append(pickle.load(open(Path.joinpath(
            ↪ mock_data_directory, "2.p"), "rb")))
        mock_points.append(pickle.load(open(Path.joinpath(
            ↪ mock_data_directory, "3.p"), "rb")))
        mock_points.append(pickle.load(open(Path.joinpath(
            ↪ mock_data_directory, "4.p"), "rb")))

        mock_positions = []
        mock_positions.append(App.odometry_to_transfomration(0, 0)) #
            ↪ Start position
        mock_positions.append(App.odometry_to_transfomration(150, -35))
        mock_positions.append(App.odometry_to_transfomration(450, -60))
        mock_positions.append(App.odometry_to_transfomration(100, 20))

        # Simulated odometry error
        # mock_positions.append(App.odometry_to_transfomration(0, 0))
```

```
        # mock_positions.append(App.odometry_to_transfomration(273,
            ↪ -45))
        # mock_positions.append(App.odometry_to_transfomration(325,
            ↪ -45))
        # mock_positions.append(App.odometry_to_transfomration(188, 0))

        return [mock_points, mock_positions]

def mock_odometry():
    mock_data = get_mock_data()

    global mock_index
    if mock_index >= len(mock_data[0]):
        return []

    return mock_data[1][mock_index]

def mock_lidar():
    mock_data = get_mock_data()

    global mock_index
    if mock_index >= len(mock_data[0]):
        return []

    mock_index += 1

    return mock_data[0][mock_index - 1]

if __name__ == "__main__":
    App(
        get_odometry_data_method = mock_odometry,
        draw_naive_data = True,
        get_lidar_data_method = mock_lidar
```

```
    ).start()
```

7.1.7. main.py

```
from app import App
from eMiRcon import eMiR
from multiprocessing import Process
from time import sleep

robot = eMiR("COM12")

def read_odometry():
    global robot
    translation, rotation = robot.get_values_for_sensors([14, 15])
    robot.reset_encoder()
    transformation_matrix = App.odometry_to_transfomration(translation
        ↪ *10, rotation*2)

    return transformation_matrix

def robot_control():
    global robot
    new_position = input("New␣position␣(distance,␣angle␣or␣'reset'):␣")

    while new_position != "exit":
        while "␣" in new_position:
            new_position = new_position.replace("␣", "")

        if new_position == 'reset':
            robot.reset_encoder()
        else:
            translation, rotation = new_position.split(",")

            robot.move(int(translation), int(rotation))
```

```
            new_position = input("New position (distance, angle or 'reset
                ↪ '): ")

def start():
    App(
        get_odometry_data_method = read_odometry,
        lidar_port = "COM13",
        draw_naive_data = True
    ).start()
```

7.1.8. eMiRcon.py

```
from time import sleep
import serial

class eMiR:
  RS232 = None

  def __init__(self, port):
    self.RS232 = serial.Serial(port)
    self.RS232.baudrate = 57600
    self.RS232.timeout = 1
    self.RS232.flushInput()

    # Ukljucuje slanje podataka o senzorima
    self.RS232.write(b'#100100EF/')
    self.RS232.flushOutput()

  def stop(self):
    self.RS232.write(b'#010000FF/')
    self.RS232.flushOutput()
    self.RS232.close()

  def reset_encoder(self):
    self.RS232.write(b'#150000EB/')
    self.RS232.flushOutput()
    self.RS232.flushInput()

  def read_sensor(self):
    number_of_output_chars = self.RS232.inWaiting()

    if number_of_output_chars < 10:
      self.RS232.flushInput()
```

```
    return ""

  output = self.RS232.readline(number_of_output_chars).decode("UTF-8")

  end = output.rfind("/")
  output = output[:end]

  start = output.rfind("*")

  self.RS232.flushInput()
  return output[start + 1: end]

def get_values_for_sensors(self, sensor_list):
  raw_sensor_values = self.read_sensor()
  sensor_values = []

  if len(raw_sensor_values) != 34:
    print("Error, got {} charachters, expected 34". format(len(
        ↪ raw_sensor_values)))
    return []

  for i in range(0, len(raw_sensor_values), 2):
    value = int(raw_sensor_values[i: i + 2], 16)

    if value > 127:
      value -= 256

    sensor_values.append(value)

  values = []
  for selected_sensor in sensor_list:
    if selected_sensor > 0 and selected_sensor < len(sensor_values):
      values.append(sensor_values[selected_sensor])
```

```
  return values


def move(self, translation, rotation):
  translation = max(min(translation, 100), -100)
  rotation = max(min(rotation, 100), -100)

  if translation < 0:
    translation += 256

  if rotation < 0:
    rotation += 256

  checksum = 256 - 1 - translation - rotation

  translation_encoded = f"{translation:02x}"
  rotation_encoded = f"{rotation:02x}"
  checksum_encoded = f"{checksum:02x}"

  command = "#01{}{}{}/".format(translation_encoded, rotation_encoded,
     ↪  checksum_encoded).upper()

  self.RS232.write(str.encode(command))
  self.RS232.flushOutput()
```