SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1935

Lokalizacija autonomnog vozila u simuliranom urbanom okruženju

Matija Vukić

Zagreb, lipanj 2019.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Zahvala

SADRŽAJ

1. Uvod

U zadnjih 30ak godina se može vidjeti ubrzan napredak u automatici i računarstvu. Tome su prethodile godine teoretskog razvoja iz kojih su kasnije nastali razni algoritmi za optimizacije i obradu podataka. Sada s razvojem računalnih aspekata poput memorija te procesorske snage možemo obrađivati sve više podataka u što manje vremena. To omogućava današnjim vozilima da u potpunosti budu električna što znači da su stalno u kontaktu s okolinom te mogu bez prestanka skupljati podatke iz okoline. Ta vozila su zapravo skupine senzora i urađaja konstantno povezanih na internet.

Skoro sva takva vozila danas imaju mogućnost autonomne vožnje. To omogućuju ljudima ugodnije iskustvo te veliko smanjenje broja prometnih nesreća koje uzrokuju vozači u stanjima koja nisu prigodna za vožnju ili čistome nemaru. S obzirom da je sigurnost u prometu jedna od najvažnijih stvari po pitanju svih sudionika ,ona mora biti prioritet. To znači da svi algoritmi za bilo kavo upravljanje vozilom moraju biti u potpunosti testirani te bez ikakvih pograšaka. Naravno to je nemoguć zahtjev zato što u takvoj domeni ti algoritmi za rad imaju previše varijabli koje se ne mogu uzeti u obzir.

Cilj ovoga rada je objesniti i prikazati rezultate nekoliko algoritama te njihove točnosti. Ulaz u algoritam su senzorska očitanja dok su izlazi lokacija vozila tj. relativna promjena lokacije između dva očitanja. Za usporedbu rezultata koristimo referentne podatke koji su prikupljni iz simulatora te je tako garantirana njihova točnost. Nekoliko primjera podataka je provedeno kroz algoritme te uspoređeno s referentnim podacima.

2. Problem lokalizacije autonomnog vozila

2.1. Lokalizacija

Roboti i vozila u većini slučajeva se primjenjuju za izvođenje repetitivnih ili opasnih po život poslova. Čovjeka zamjenjuje robot ali to znači da je za upravljanje robota zadužen taj isti robot ili neki udaljeni sustav, čovjek više nema tu ulogu. U tu svrhu su roboti i vozila opremljeni raznim senzorima da bi se to omogućilo. Svi podaci prikupljeni iz tih senzora se koriste prilikom lokalizacije robota ili vozila.

Lokalizacija je postupak određivanja lokacije objekta u prostoru iz senzorskih podataka. Lokalizacija može biti vrlo zahtjevan zadatak te se u tu svrhu mogu koristiti algoritmi različitih složenosti. Što je algoritam složeniji, to se sporije izvodi ali je točiji dok se neki više optimizirani tj. brži algoritmi brže izvode ali se događaju veće greške te se one akumuliraju prilikom izvođenja algoritma.

Koriste se algoritmi za istovremenu lokalizaciju i mapiranje [dodaj SLAM link] tj. za stvaranje karte nepoznatog prostore kojime se robot kreće te koordinate u tome prostoreu. Lokalizacija odgovara 'Gdje je robot sada?' tj. gdje je sada naspram prethodne lokacije. Na to pitanje se može odgovoriti ovisno o tome radi li se o lokalizaciji u otvorenome ili zatvorenome prostoru. Lokacija robota je oglavnom prikazana u kartezijskom koordinatnom sustavu, bilo to u 2d ili 3d prostoru.

Postoje dvije vrste lokalizacije:

- Lokalna - informacije se prikupljaju pomoću senzora robota iz njegove okoline
- Globalna - informacije se dobiju iz GPS-a ili slično

Pregled pristupa problemu lokalizacije

Jedna od najjednostavnijih metoda je "Metoda najmanjih kvadrata" (eng. Least Squares Error) gdje se koristi metoda najmanjih kvadrata za regresijsku analizu podataka. Cilj te metode jest minimizacija pogreške gdje robot jest i gdje bi robot trebao biti tj. ona okvirno procjenjuje gradijent funkcije pomaka robota.

Praćenje pozicije (eng. Pose Tracking) metoda se koristi kada je poznata početna pozicija robota pa je potrebno samo pratiti njegovu poziciju kroz vrijeme. Metoda koristi ekstrakciju tj. izdvajanje značajki okoline koje se mogu uspoređivati te se tako kroz vrijeme može pratiti promjena položaja nekih uočljivih objekata.

Metoda višestrukih hipoteza (eng. Multiple Hypotesis Localization) pretpostavlja da početna pozicija nije poznata ali je poznata topografija mape. U ovome slučaju početnu poziciju može robotu pridodati korisnik ili robot uvijek može započeti iz iste pozicije. Ideja iza ove metode je da se detektira svojstvo te se preko njega stvaraju hipoteze o položaju robota naspram toga objekta kojemu pripada to svojstvo. Može se stvoriti nova hipoteza ili se može poboljšati neka od prethodnih hipoteza ili ipak eliminirati.

Metoda iteracije najbližih točaka (eng. Iterative Closest Point) minimizira razliku između dvije skupine točaka tako da iterira između svake dvije točke te pronalazi onu kombinaciju koja daje najmanju grešku. Često se koristi pri rekonstrukciji 2D ili 3D površina nakon skeniranja. Tijekom izvođenja te metode jedna skupina točaka je fiksna tj. referentna dok se druga transformira tako da se najbolje slažu koordinatama u referentnom skupu. Postoje mnoge varijante ICP-a od kojih su point-to-point (usporedba točka-točka) , point-to-plane (usporedba točke-površina) i point-to-line (usporedba točka-linija) najpopularnije.

Metoda usporedbe očitanja (eng. scan matching) koristi dva uzastopna očitanja senzora robota poput lasera, sonara, ... da se pronađe relativan pomak robota u prostoru. Razlike između dva očitanja senzora se mogu uočiti vrlo lako zbog učestalosti skeniranja tj. frekvencije dohvaćanja senzorskih podataka te o gustoći lasera kojih uglavnom ima od nekoliko stotina do nekoliko tisuća. Načina na koji se zapravo traže razlike između dva očitanja ima mnogo. Koriste se laseri (eng. laser range finders) da bi vidio prepreke i odometrija kotača (eng. wheel odometry) da dobije okvirno stanje robota. Odometrija iz kotača ima određenu grešku zbog proklizavanja kotača ili nekog drugog razloga te se ona tada ispravlja pomoću izračunatih vrijednosti odometrije iz lasera. Ova metoda je zapravo metoda iteracije najbližih točaka ali ograničena u dvije dimenzije.

2.2. Problem lokalizacije

Sve prethodno navedene imaju nešto zajedničko, a to je da koriste algoritme čiji rezultati nikada nisu posve točni.
Ta netočnost može proizaći zbog sljedećih razloga:

- Šum u očitanjime - u podacima koji se dobiju iz senzora uvijek ima i podataka koji su nastali zbog privremenih objekata (npr. pas koji prolazi pokraj vozila)
- Sinkronizacija obrade i očitanja podataka - zbog prebrzog slanja podataka algoritmu, te se tako mogu neka očitanja preskočiti
- Samog načina izvedbe senzora - možda senzor zbog samog načina fizičke izvedbe ima uračunat šum
- ...

Razne metode lokalizacije već unutar svog tijeka izvođenja imaju metode koje prate veličinu relativne pogreške te ju pokašaju minimizirati nakon svake iteracije ali ta pograška i dalje postoji te će uvijek i postojati. Te pogreške se robota koji rade u skladištima ne moraju uzeti previše ozbiljno, dok se kod autonomnih vozila u svakodnevnome cestovnom prometu ili industrijskih robota te pogreške moraju uvijek uzeti u obzir.

3. Priprema podataka

3.1. Simulator

Za točne referentne podatke potrebno je imati simulirano okruženje. Takvo simulirano okruženje se zove simulator. Potreban je simulator koji već ima integrirane razne mape, razne senzore, vozila te način komunikacije s tim vozilima iz vanjskih skripti. Neki od simulatora su opisani u sljedećem tekstu.

Carla

Slika 3.1: Carla logo

Carla je simulacijsko okruženje koje služi za testiranje metoda i algoritama prilikom razvoja autonomnih vozila. U pozadini koristi Unreal Engine za izvršavanje simulacije. Simulator se ponaša kao poslužitelj koji prima naredbe iz vanjskih klijentskih programa. Ti klijentski programi su pisani u programskom jeziku pytohn. Carla ima integrirane razne senzore te su neki od njih:

- RGB kamera
- LIDAR senzor
- Senzor dubine
- GNSS

Sponzori projekta su Intel, Toyota, GM i Computer vision Center. Više o ovome simulatoru će biti u sljedećem poglavlju.

Apollo

Slika 3.2: Apollo logo

Apollo je također rješenje za testiranje autonomnoh vozila. Sadrži simulator ali također je i potpuno komercijalno rješenje. Podržava razne scenarije, ima sustav ocjenjivanja koji daje ocjenu na temelju desetak metrika. Simulacije zapravo provodi u oblaku tj. koristi Microsoft Azure. Sponzori projekta su mnoge azijske tvrtke kao i Ford, Microsoft, Daimler, Honda, Intel i ostali.

rFpro

Slika 3.3: rFpro logo

rFpro je kompletno rješenje za testiranje autonomnih vozila. U potpunosti je komercijalno rješenje ali je zato jedno od najboljih u svijetu. Uglavnom je usredotočeno na primjenu strojnog učenja u autonomnim vozilima. Ima jednu od najvećih baza digitaliziranih stvarnih likacija diljem svijeta. Dinamički sustav vremena omogućuje testiranje ponašanja vozila u raznim vremenskim uvjetima. Sponzori projekta su BMW, Shell, GM, Renault i ostali.

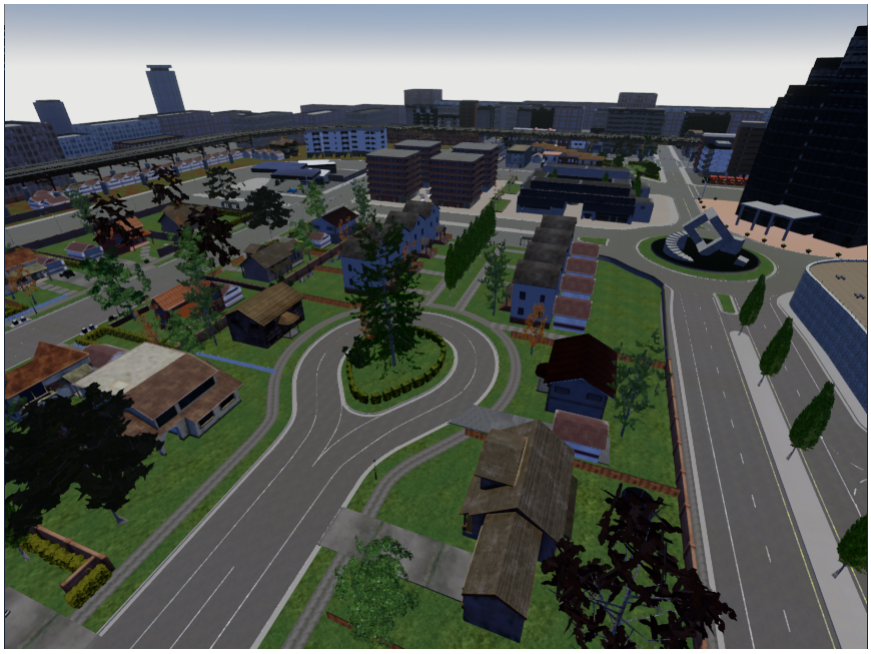
AVSimulation

Slika 3.4: AVSimulation logo

AVSimulation se zapravo sastoji od simulatora vožnje i samog simulatora SCANeR. SCANeR je skup aplikacija koji pružaju rute, senzore, vozila, dinamičko vrijeme, pisanje skripti. Vrlo je modularan. Smulator vožnje je zapravo kupola koja se sastoji od cijelog vozila te se zapravo kretanje tog vozila simulira unutar te kupole. Sponzori su Renault, PSA, Volvo, Microsoft, Mazda i ostali.

3.2. Opis okruženja

Simulacijsko okruženje će biti Carla zato što ima vrlo široko programsko sučelje za upravljanje aspektima simulacije te je besplatno za korištenje. Simulator se pokreće kao poslužitelj te se vozila dodaju pomoću skripte koja je napisana u programskom jeziku python.



Slika 3.5: Primjer karte pod nazivom Town03

Na slici 3.5 se vidi pogled na jedan od 7 mapa iz perspektive slobodne kamere. Korištene mape su definirane OpenDrive standardom. Simulator podržava raznolike senzore. Svi ti senzori se mogu postaviti samostalno na mapu, ali su najkorisniji kada se postave na drugo vozilo.

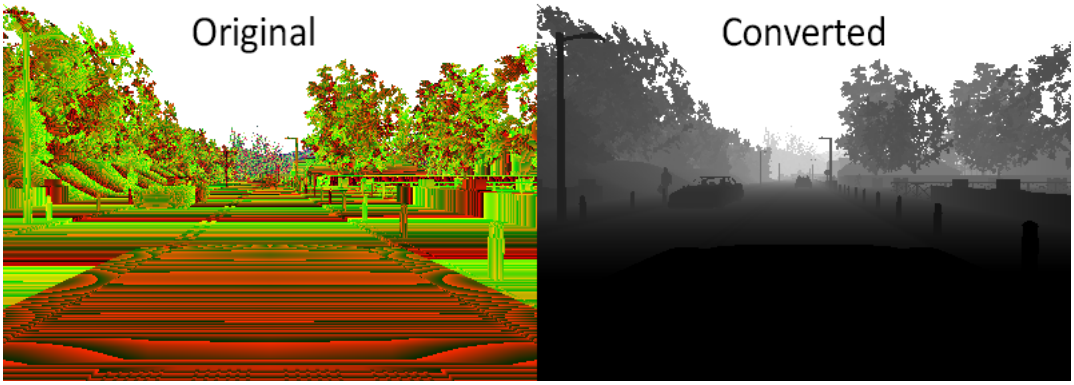
3.2.1. Senzori

RGB senzor

Slika 3.6: Primjer regularne kamere

RGB senzor simulira kameru koja može snimati sliku koristeću crvenu, zelenu i plavu boju, tj. regularnu kameru. Ovaj senzor može prikupljati podatke iz simulatora u video obliku ili kao niz slika. Ovaj senzor se može koristiti u metodama lokalizacije koje kao temeljni algoritam koriste prepoznavanje ostalih sudionika u prometu prema obliku tj algoritmi prepoznavaju kontekst slike.

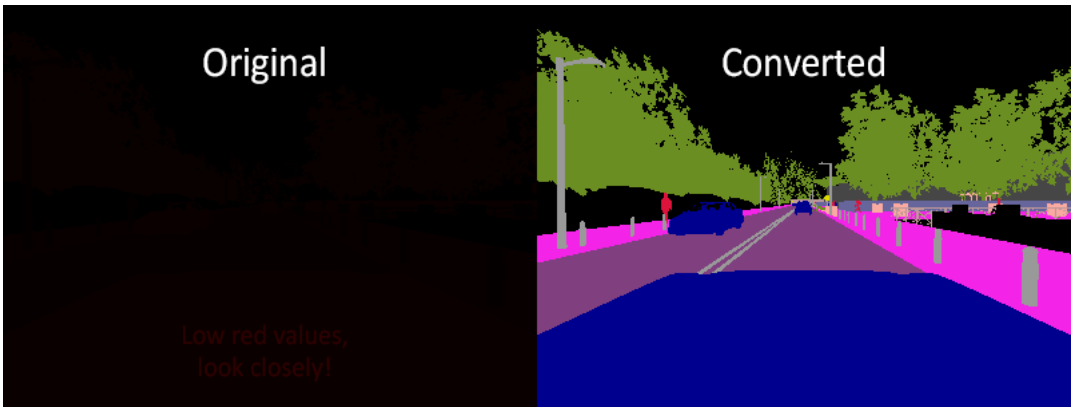
Senzor dubine



Slika 3.7: Primjer rezultata senzora dubine

Ovaj senzor koristi nizove projicirajućih točaka da bi ilustrirao udaljenosti objekata, original na slici 3.7. Tada se ti podaci pretvaraju u crno bijelu sliku gdje je svaki pixel u nijansama sive boje tj. ovisno koliko je objekt na određnome pixelu odaljen od kamere imati će svijetliju nijansu, konvertirano na slici 3.7.

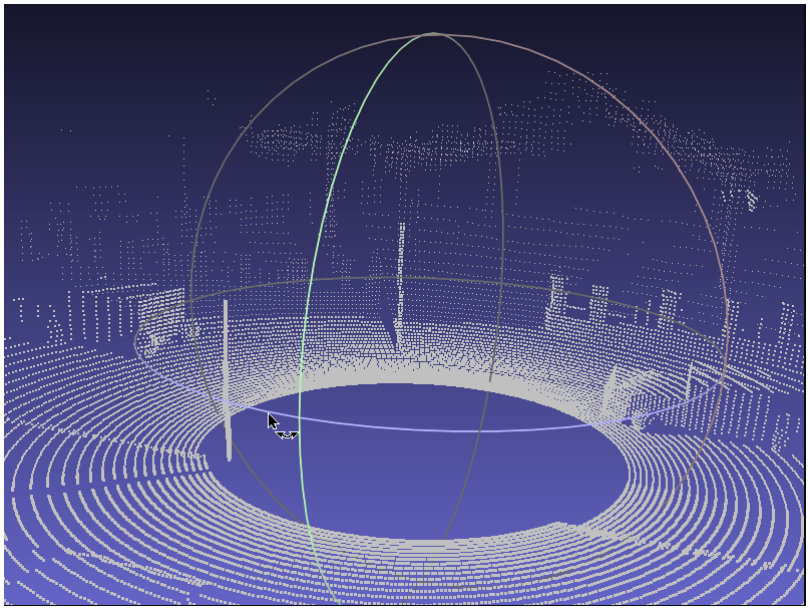
Semantičke segmentacije



Slika 3.8: Primjer semantičke segmentacija

Ovaj senzor zapravo nije senzor ali je grupiran u klasu senzora zato što radi na vrlo sličan način kao i ostali senzori u simulatoru. Ovaj senzor dijeli sliku kamere na semantičke dijelove tj. objekte različitog tipa predstavlja drugim bojama. Na slici 3.8 se vidi da je nebo crne boje, auti su plave boje, drveće je zeleno itd. OVaj način raspoznavanja objekata je moguć samo u simulator zato što nije potrebno razpoznavati objekte na slici već su oni definirani u simuliranoj okolini.

LIDAR senzor



Slika 3.9: LIDAR podaci

LIDAR (eng. light detecting and raging) senzor je zapravo vertikalan skup lasera koji simuliraju skeniranje od 360 stupnjeva tako da se rotiraju određeni broj puta u sekundi. Povratni podaci senzora su zapravo točke tj. koordinate relativne naspram samoga senzora do kojih su laseri uspjeli doći. Na slici 3.9 se mogu vidjeti takvi podaci vizualizirani u programu MeshLab. Više o ulaznim parametrima senzora kasnije u radu.

Senzor sudara

Ovaj senzor dojavljuje klijentskome programu ako se vozila sudarilo s drugim objektom u simulaciji.

Senzor prijelaza trake

Ovaj senzor dojavljuje klijentskome programu ako je vozilo prošlo preko trake na cesti.

GNSS senzor

Senzor koji dojavljuje klijentskome programu trenutnu GNSS lokaciju vozila. Ta lokacija se interno računa tako da se lokacija vozila dodaje na geografsku referentnu lokaciju definiranu za cijelu mapu.

Senzor prepreke

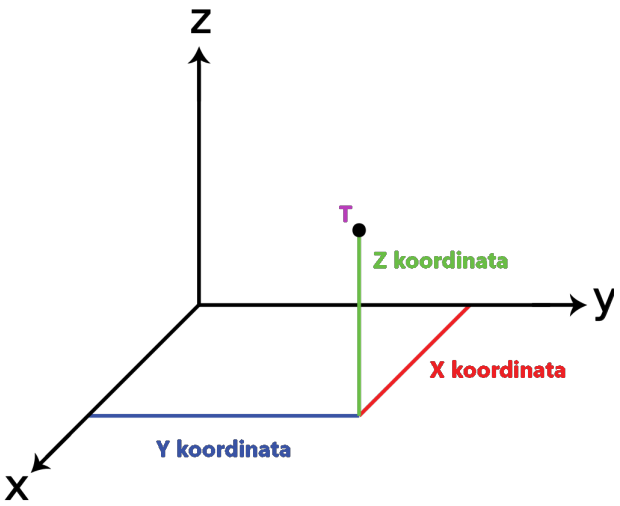
Ovaj senzor javlja klijentskome programu ako se ispred vozila nalazi prepreka.

3.2.2. Promet

Postoji poveći broj već unaprijed definiranih vozila koja se mogu koristiti. Mogu se koristiti kao nositelji senzora ili kao ostali sudionici u prometu. Carla ima dobro definirana prometna pravila te semafore da bi simulacija izgledala što vjernije.

3.3. Oblak točaka

Oblak točaka je skupina podataka koji definiraju neki objekt u prostoru. Oblak točaka se obično generira pomoću trodimenzionalnog skenera. Oblaci točaka imaju veliku primjenu u rekonstrukcijama predmeta, vizualizaciji, animaciji, virtualnoj i proširenoj stvarnosti te industrijskoj proizvodnji i kontroli kvalitete. U trodimenzionalnome kartezijevom sustavu svaka točka je definirana s tri atributa, a to su njene x, y i z koordinate. Uz te osnovne podatke svaka točka također može sadržavati i podatke o njenoj boji.



Slika 3.10: Ilustracija točke u kartezijevom koordinatnome prostoru

Slika 3.11: Oblak točaka katedrale u Zagrebu

Na slici 3.11 se vidi skup točaka koji opisuje katedralu u Zagrebu te okolne objekte. Skup se sastoji od oko 22 milijuna točaka. Skup točaka je mnogo jednostavnije koristiti za mapiranje objekata od slika zato što se lakše može obraditi na računalu. Ti podaci su zapravo spremljeni u tekstualne datoteke pa su lako prenosivi i čitljivi. Konkretno naše metode koriste oblak točaka prikupljen s rotirajućim laserima na vozilu. Jedan taj skup točaka predstavlja stanje okoline vozila u jednome trenutku.

3.4. Referentni podaci

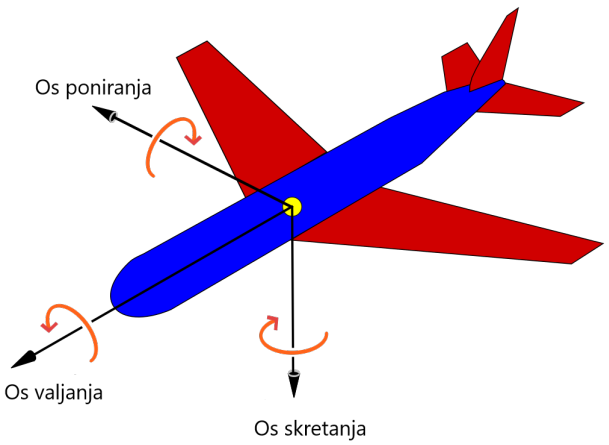
Referentni podaci su oni podaci s kojima se uspoređuju rezultati metoda. Ti referentni podaci su generirani u simulatoru te predstavljaju lokaciju i rotaciju vozila u jednome trenutku. Referentni podaci se zapravo sastoje od lokacije i rotacije vozila.

Lokacija

Lokacija vozila je također definirana kao točka u kartezijevom koordinatnome prostoru. Sastoji se od x, y i z koordinata. Slično kao prikazano na slici 3.10.

Rotacija

U trodimenzionalnome prostoru objekt se zapravo može rotirati oko beskonaćnoga broja osi ali se u pravilu uzimaju 3 statičke osi. Te osi se nazivaju os skretanja (eng. yaw), os poniranja (eng. pitch) i os valjanja (eng. roll).

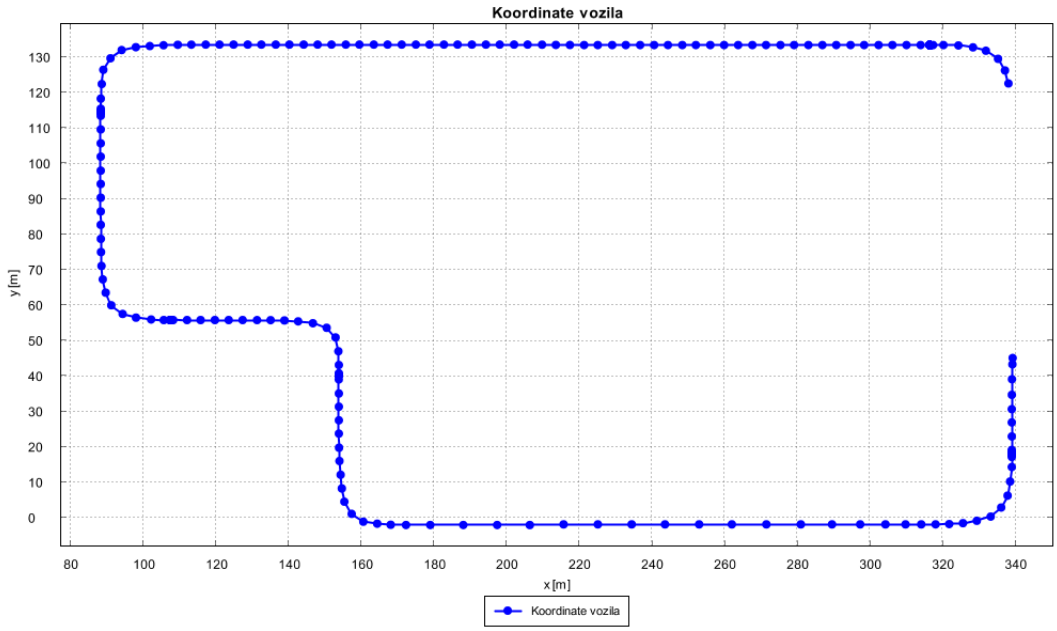


Slika 3.12: Ilustracija osi rotiranja

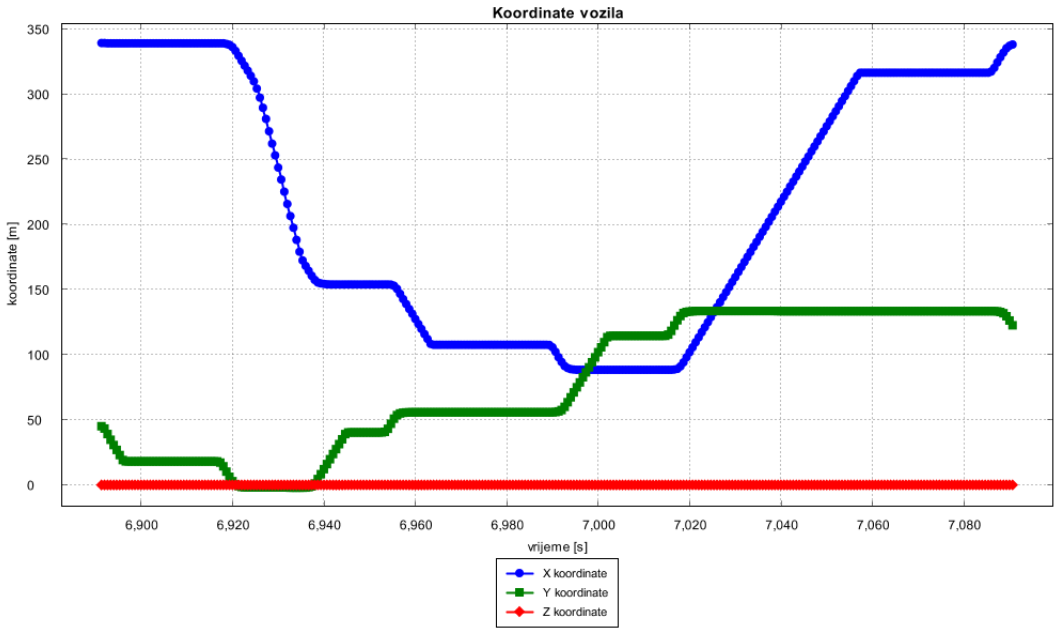
Os rotacije je os koja prolazi u smjeru kretanja vozila (x os), os poniranja je zapravo os okomita s os rotacija (z os), dok je os skretanja okomita na obje prethodne osi (y

os). Te osi su ilustrirane na slici 3.12.

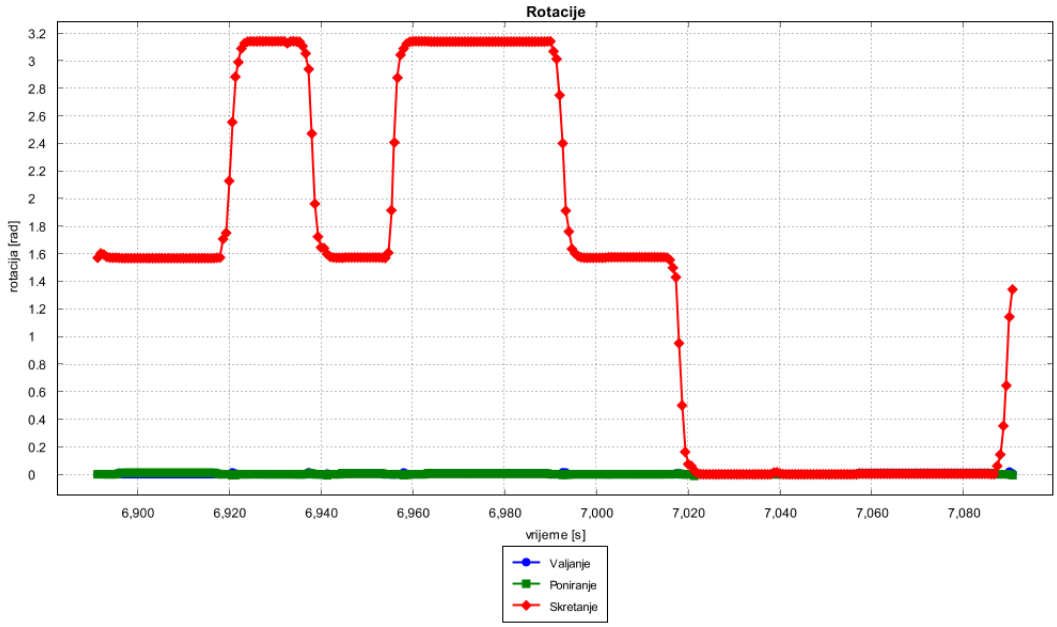
Vizualizacije referentnih podataka za jedan primjer kretanja vozila možemo vidjeti na sljedećim slikama. Na svim grafovima svaka točka predstavlja jedno očitanje. Grafovi su izgrađeni pomoću programskog jezika Kotlin koristeći biblioteku XChart. Slika 3.13 prikazuje graf lokacija vozila. Slika 3.14 pokazuje koordinate vozila u vremenu. Možemo vidjeti da se mijenjaju samo x i y koordinate zato što vozilo može samo skretati. Slika 3.15 valjanje, poniranje i skretanje oko statičnih osi vozila u vremenu. Također se može uočiti da se samo skretanje mijenja zato što se vozilo ne može valjati niti ponirati. Slika 3.16 također pokazuje rotaciju vozila u vremenu ali predstavljenu u obliku kvaterniona.



Slika 3.13: Graf lokacije vozila

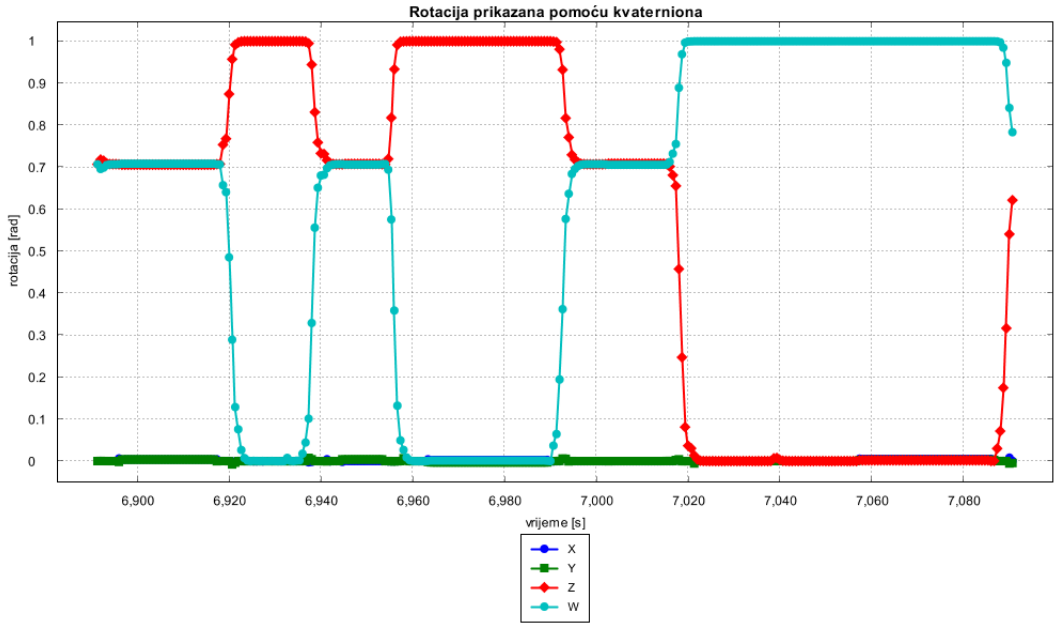


Slika 3.14: Graf x, y i z koordinata vozila u vremenu



Slika 3.15: Valjanje, skretanje i poniranje vozila u vremenu

Rotacija između dva sustava može biti reprezentirana pomoću rotacijskih matrica. Te matrice su veličine 3x3 te predstavljaju rotaciju oko određene osi.



Slika 3.16: Rotacija vozila u obliku kvaterniona

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{3.1}$$

$$R_y(\beta) = \begin{pmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{pmatrix} \tag{3.2}$$

$$R_z(\gamma) = \begin{pmatrix} cos(\gamma) & -sin(\gamma) & 0 \\ sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

Za razliku od rotacijskih matrica Eulerova reprezentacija rotacije se sastoji od samo 3 broja. Ta tri broja opisuju sekvencu kojom je objekt rotiran. Rotacija se može opisati na 12 načina zato što se prikazuje dinamička rotacija. Primjeri su XYX, ZZX, ZYZ, itd. Objekt u trodimenzionalnome prostoru možemo rotirati pomoću matrica 3.1, 3.2 i 3.3 u prethodni dogovor da se prvo rotira oko x osi, tada oko y osi i naposljetku oko z osi pomoću formule 3.4 gdje su α, β i γ kutevi rotacije oko x, y i z osi.

$$\begin{bmatrix} X_t \\ Y_t \\ Z_t \end{bmatrix} = R(\gamma)R(\beta)R(\alpha) \begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix} \tag{3.4}$$

Kao kompromis između prednosti i nedostataka prethodnih reprezentacija rotacije objekata koristi se sustav kvaterniona (eng. Quaternion). U matematici kvaterniona su skup brojeva koji proširuju kompleksne brojeve. Rotacija je prikazana kao vektor od 4 komponente. Formula 3.5 prikazuje kako se množe kvatrnioni.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{3.5}$$

Jedan kvaternion (jedinični) je definiran kao zbroj skalara a i vektora $[abc]$ te iznosi:

$$q = a + b\hat{i} + c\hat{j} + d\hat{k} \tag{3.6}$$

Ako os rotacije napišemo u obliku $u = u_x\hat{i} + u_y\hat{j} + u_z\hat{k}$ gdje su u_x, u_y i u_z kalarne veličine te je θ kut zakreta te iste osi tada se rotacija može zapisati kao:

$$q = \cos(\frac{\theta}{2}) - (u_x i + u_y j + u_z k)\sin(\frac{\theta}{2}) \tag{3.7}$$

Pretvorba kvaterniona u generalnu rotacijsku matricu se provodu na sljedeći način:

$$R = \begin{pmatrix} a^2 + b^2 - c^2 - d^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & a^2 - b^2 + c^2 - d^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & a^2 + b^2 - c^2 + d^2 \end{pmatrix} \tag{3.8}$$

3.5. Izvor podataka

Kao izvor podataka za algoritme se koristi već prije spomenuti LIDAR senzor. Podaci koje nam vrati senzor su u obliku već spomenutog skupa točaka (eng. point cloud). Lidar senzor je zapravo vertikalan skup lasera.

Građa lidara

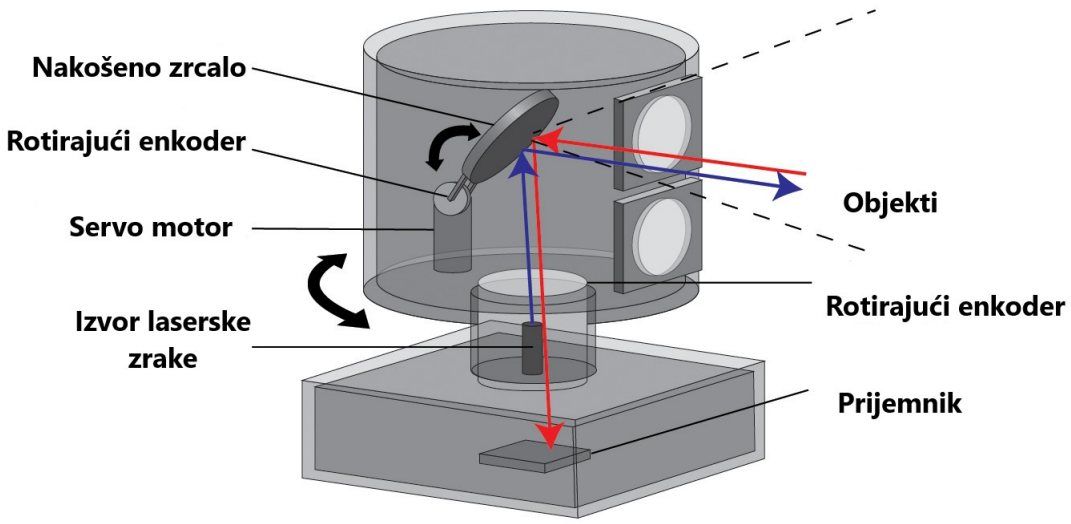
Lidar se sastoji od sljedećih glavnih dijelova:

1. Nakošeno zrcalo
2. Servo motor

3. Izvor laserske zrake

4. Detektor laserske zrake

5. Optički rotirajući enkoder



Slika 3.17: Ilustracija građe lidara

Izgled je ilustriran na slici 3.17. Ovaj primjer ima samo jedan laser. Servo motor rotira kučište gdje se nalazi laser te tako omogućuje očitanje u svih 360 stupnjeva. Danas se zahtjeva da brzine rotacije budu veće od 1000 okretaja u minuti. Nakošeno zrcalo služi za usmjeravanje zrake. Izvor laserske zrake proizvodi lasere u pulsevima. Te zrake se nalaze u infracrvenome valnome području te su sigurne za vid i male energije. Laserski detektor detektira laserske pulseve te dekodira podatke.

Rad lidara

Lidar može sadržavati razne sustave za lokalizaciju i orijentaciju poput GPS-a, inercijskih sustava (IMU) i ostalih. Koristi dvodimenzionalno očitanje u horizontalnom smijeru i vertikalno polje vida za kreiranje panorame u 360 stupnjeva. Za dekodiranje podataka se koristi vrijeme povratka laserske zrake i valna duljina ako podržava detekciju boje. Postoji mogućnost nepotpunih očitanja zbog razlike između frekvencije rotacije i frekvencije očitanja senzora. Općenito se udaljenost predmeta s laserom

može izračunati formulom 3.9 gdje je c brzina svijetlosti, a t vrijeme povratka pulsa zrake.

$$D = \frac{ct}{2} \tag{3.9}$$

Ovakav način je pogodan za srednje do velike udaljenosti, dok za vrlo male udaljenosti se koriste druge metode zato što se teško računaju vremena povratka zrake.

Tipovi lidara

Jedna od podjela lidara je po tome gdje se koristi. Tako imamo zračne i zemaljske lidare. Zračni lidari se koriste za mapiranje topologije prostora. Zasada je to najbolji način za mapiranje površina zato što omogućuje filtriranje vegatacije te sa tako lako mogu dobiti topološki podaci. Takvi lidari su pričvršćeni za avion, dron ili helikopter. Zemaljski lidari su postavljeni statički na zemlji. Koriste se u forenzici i rekonstrukciji objekata.

Primjena

Lidar se koristi u geodeziji, geografiji, geologiji, seizmologiji, fizici a u zadnje vrijeme prilikom lokalizacije autonomnih vozila. Također se koriste u agrokulturi prilikom navodnjavanja ili sadnje. Koristi se prilikom klasifikacije biljaka uz pomoć strojnog učenja. U arheologiji se primjenjuje od mapiranja arheoloških nalazišta koja su fizički nedostupna pa do mapiranja oštećenih predmeta.

Prednosti i nedostatci

Prednosti su što se može iskoristiti u mnoge svrhe i možemo dobiti razne informacije o okolini. Također su podaci koji se dobiju vrlo lako obradivi i čitljivi. Jedini nedostatak je brzina obrade tih podataka. Za velike skupove točaka se rijetko koristi obrada u stvarnome vremenu zato što je potrebna velika računalna snaga.

Parametri simuliranoga lidara

S obzirom da je lidar simuliran njegovi parametri nisu vezani uz njegovu izvedbu nego se mogu postaviti po želji. Jedina razlika je što simulirani lidar nema kašnjenje tj. njegova očitanja su gotovo trenutna što u stvarnome svijetu nikako ne može biti. Možemo mu postaviti sljedeće parametre:

- broj kanala - broj vertikalnih laserskih zraka

- donja granica polja vida - koliko nisko su orijentirani laseri
- gornja granica polja vida - koliko visoko su orijentirani laseri
- ukupan broj točaka - ukupan broj točaka po laseru u pojedinom očitanju
- frekvencija rotacije - koliko često se laseri rotiraju
- vremenski korak - koliko često se podaci prikupljaju

3.6. Prikupljanje podataka

Podaci su prikupljeni tako da se simulator pokrene u poslužiteljkom načinu rada te se tada pokreće klijentska skripta napisana u python jeziku. Ta skripta uspostavi kontakt s poslušiteljem te se tako šalju naredbe. Te naredbe će zapravo stvoriti naše vozilo, ostale sudionike i senzor. Nakon što smo prikupili dovoljno podataka skripta će obrisati stvorene objekte i spremiti podatke u datoteke. Tada simulator može prekinuti s radom te se ti podaci mogu obrađivati na bilo koji način.

Pokretanje simulatora

```
CarlaUE4.exe \
  /Game/Carla/Maps/Town01 \
  -quality-level=Low \
  -benchmark -fps=15 \
  -windowed -ResX=800 -ResY=600 \
  -carla-port=2000 \
```

Izvorni kod 1: Carla naredba

Simulator Carla se pokreće pomoću python skripte zbog boljeg upravljanja parametrima ali se zapravo sastoji od naredbe pokazane u primjeru 1. Simulacija se pokreće u mapi pod nazivom Town01. Kvaliteta je postavljena na najnižu vrijednost kao i broj slika u sekundi zbog boljih performansi izvođenja. Prozor smo postavili na vrlo malu rezoluciju od 800 pixela širine i 600 pixela visine također zbog boljih performansi. Vrlo važan parametar je sučelje preko kojega klijentski program komunicira s poslužiteljem. Ovdje je definiran kao 2000.

Klijentska kripta

Referentni i testni podaci su prikupljeni iz simualtora ali iz različitih izvora. Referentni podaci su prikupljeni iz samoga simulatora dok su testni podaci prikupljeni pomoću senzora.

```
class CarlaProp:
  spawn_delay = 1.0
  host = "localhost"
  port = 2000
```

Izvorni kod 2: Carla postavke

```
def connect_to_carla(self):
  self.client = carla.Client(
    CarlaProp.host,
    CarlaProp.port
  )
  self.client.set_timeout(2.0)
  self.world = self.client.get_world()
  settings = self.world.get_settings()
  settings.synchronous_mode = True
  self.world.apply_settings(settings)
```

Izvorni kod 3: Uspostava konekcije s poslužiteljem

U primjeru izvornoga koda 3 klijent se spaja na Carla poslužitelj čiju smo lokaciju (IP adresu i sučelje) definirali u klasi `CarlaProp`. Također postavljamo sinkroni način rada simulatora, a razlog je taj što želimo upravljati frekvencijom slanja podataka iz poslužitelja prema klijentima. Varijabla `self.world` služi za izvođenje svih operacija koje su vezane uz svijet.

Svaka mapa ima već unaprijed definirane točke stvaranja tj. koordinate u svijetu na kojima možemo stvoriti objekte. Te koordinate se nalaze na cestama. Njih možemo dobiti naredbom prikazanom na primjeru izvornoga koda 4.

Sljedeće što slijedi je stvaranje ostalih sudionika prometa tj. ostalih vozila. Carla ima već unaprijed definirane nacrte raznih objekata.

```
1     def get_spawn_points(world):
2       return list(world.get_map().get_spawn_points())
```

Izvorni kod 4: Dohvaćanje liste koordinata stvaranja

```
1  def get_vehicle_blueprints(world):
2    blueprints = world.get_blueprint_library()
3                       .filter('vehicle.*')
4    blueprints = [
5      x for x in blueprints
6        if int(x.get_attribute('number_of_wheels')) == 4
7    ]
8    return [
9      x for x in blueprints
10       if not x.id.endswith('isetta')
11   ]
```

Izvorni kod 5: Dohvaćanje nacrta vozila

Na kodu 5 se vidi kako koristimo knjižnicu nacrta da bi filtrirali nama potrebne nacrte. Koristiti će se samo vozila koja imaju 4 kotača.

Koristeću točke stvaranja i nacrte vozila sada se mogu ta vozila stvoriti u svijetu. Na 6 se koristeći petljom stvara unaprijed zadan broj ostalih sudionika definiranih u varijabli klase `self.npc_number`. Njihove reference se tada spremaju u listu zato što se na kraju izvođenja moraju uništiti. Stvaranje instance nacrta se izvodi naredbom na liniji 7. Također smo svakoj instanci definirali autonomni način rada na liniji 11.

Sada se definira vozilo koje zapravo promatramo tj. koje ima na sebi lidar senzor. To se radi na približno jednak način kao u primjeru 6. Izvorni kod je prikazan na primjeru 7.

Sada slijedi pronalazak nacrta za lidar senzor, postavljanje njegovih atributa, njegovo instanciranje i postavljanje na promatrano vozilo. Izvorni kod je prikazan na 9. Postavke LIDAR senzora se nalaze u klasi `LIDARProp`.

Pomoću `sensor_tick` parametra se definira da simulator treba prikupljati podatke najbrže što može. Varijabla `laser_range` definira koliko daleko laser može doprijeti, ovdje je definirano na 1500 centimetara ili 15 metara. Frekcencija rotacije lidara je definirana varijablom `rotation_frequency` i iznosi 120 rotacija u minuti.

```
def spawn_npcs(self):
  blueprints = utils.get_vehicle_blueprints(self.world)
  points = self.spawn_points[1:self.npc_number+1]
  for i in range(self.npc_number):
    actor_blueprint = random.choice(blueprints)
    actor_spwn_point = points[i]
    spawned_actor = self.world.try_spawn_actor(
      actor_blueprint,
      actor_spwn_point
    )
    spawned_actor.set_autopilot()
    self.npcs.append(spawned_actor)
  self.tick()
```

Izvorni kod 6: Stvaranje ostalih vozila

```
def spawn_actor(self):
  lib = self.world.get_blueprint_library()
  spawn_point = self.spawn_points[0]
  actor_blueprint = utils.get_vehicle_blueprint(lib)
  self.actor = self.world.spawn_actor(
    actor_blueprint, spawn_point
  )
  self.actor.set_autopilot()
  self.tick()
```

Izvorni kod 7: Stvaranje promatranoga vozila

Gornja granica mjerenja lasera je 45°, a donja je -80°. parametar `location` definira lokaciju lasera tj. bit će u središtu koordinatnog sustava ali na visini od 4 metra. Sredina tog koordinatnog sustava je zapravo sredina unutarnjeg sustava vozila na kojemu će taj senzor biti pričvršćen.

```
1  import carla
2
3  class LIDARProp:
4    sensor_tick = str(0.0)
5    channels = str(180)
6    laser_range = str(1500.0)
7    rotation_frequency = str(120.0)
8    points_per_second = str(600_000)
9    upper_fov = str(45.0)
10   lower_fov = str(-80.0)
11   location = carla.Transform(
12     carla.Location(x=0, y=0, z=4)
13   )
```

Izvorni kod 8: LIDAR atributi

Na liniji 3 primjera 9 dohvaćamo nacrt lidar senzora. Tada od linije 4 do 29 postavljamo zadane postavke nad nacrtom. Konačno na liniji 31 stvaramo instancu senzora ali metodi predajemo dodatan parametar `attach_to` koji je je jednak referenci na naše vozilo. Također umjesto stvarnih koordinata, za lokaciju senzora postavljamo lokaciju relativnu naspram lokacije vozila. Na liniji 35 postaljamo metodu `lidar_callback()` kao metodu koju će simulator pozvati svaki puta kada senzor očita okolinu i pošalje podatke. Spremanje podataka se izvršava tek nakon što smo sakupili konačan broj očitanja. Za spremanje podataka u datoteke se koristi posebna klasa `DataSaver` pokazana na primjeru 10.

Primjeri 10 i 11 prikazuju klasu `DataSaver` koja služi za spremanje podataka u datoteke. Datoteke s podacima iz senzora se nalaze u direktoriju s nazivom pointclouds, dok se datoteke s podacima o lokaciji vozila nalaze u direktoriju s nazivom actortransforms. Oba ta direktorija se nalaze u direktoriju s nazivom *output*. Datoteke s informacijama o lokaciji vozila se sastoje od 2 reda. Prvi red sadrži vremensku oznaku a drugi sadrži lokaciju i transformaciju koji su opisani u prijašnjem poglavlju. Datoteke koje sadrže podatke o jednome očitanju se nalaze u tekstualnim datotekama s ekstenzijom .ply te se sastoji od zaglavlja i po jedan redak za svaku točku u očitanju. Također postoji još jedna datoteka koja samo sadrži relativnu transformaciju između senzora i vozila. Ona se nalazi u direktoriju *output*.

```
    def connect_LIDAR(self):
      lib = self.world.get_blueprint_library()
      blueprint = utils.get_lidar_sensor_blueprint(lib)
      blueprint.set_attribute(
        'sensor_tick',
        LIDARProp.sensor_tick
      )
      blueprint.set_attribute(
        'channels',
        LIDARProp.channels
      )
      blueprint.set_attribute(
        'range', LIDARProp.laser_range)
      blueprint.set_attribute(
        'rotation_frequency',
        LIDARProp.rotation_frequency
        )
      blueprint.set_attribute(
        'points_per_second',
        LIDARProp.points_per_second
        )
      blueprint.set_attribute(
        'upper_fov',
        LIDARProp.upper_fov
        )
      blueprint.set_attribute(
        'lower_fov',
        LIDARProp.lower_fov
      )
      utils.print_sensor_blueprint_data(blueprint)
      self.lidar = self.world.try_spawn_actor(
        blueprint, LIDARProp.lidar_relative_postion,
        attach_to=self.actor
      )
      self.lidar.listen(
        lambda data: self.lidar_callback(data)
      )
      self.tick()
```


Izvorni kod 9: Stvaranje LIDAR senzora

```python
import utils as utils
import time
import threading
from concurrent.futures import
  ThreadPoolExecutor, as_completed
from lidar_properties import
  LIDARProperties

class DataSaver:

  def __init__(self):
    self.pc_path = 'output/pointclouds'
    self.trans_path = 'output/actortransforms'
    self.rel_path = 'output/relative_transform.txt'

  def initialize_folders(self):
    utils.create_directory(self.trans_path)
    utils.create_directory(self.pc_path)

  def save(self, scans):
    self.initialize_folders()
    self.save_rel_trans()
    start_time = time.time()

    with ThreadPoolExecutor(max_workers=10) as executor:
      jobs = list()
      for i, (scan, transform) in enumerate(scans[1:]):
        job = executor.submit(
          self.process_pair, scan, transform, i
        )
        jobs.append(job)
      for future in as_completed(jobs):
        future.result()
```

Izvorni kod 10: Klasa za spremanje podataka

```
  def process_pair(self, scan, trans, i):
    frame_number = scan.frame_number
    s_path = f'{self.pc_path}/{frame_number:06d}.ply'
    self.save_scan(scan, s_path, i + 1)
    tpath =  f'{self.trans_path}/{frame_number:06d}.txt'
    self.save_transform(
      trans, scan.timestamp, tpath, i + 1
    )

  def save_rel_trans(self):
    with open(self.rel_path, 'w+') as file:
      file.write(
        utils.transform_to_string(LIDARProp.location)
      )

  def save_scan(self, scan, path, i = 0):
      scan.save_to_disk(path)

  def save_transform(self, trans, timestamp, path, i = 0):
    with open(path, 'w+') as file:
      file.write(
        f"{timestamp}\n{utils.transform_to_string(trans)}"
      )
```

Izvorni kod 11: Klasa za spremanje podataka - nastavak

```
ply
format ascii 1.0
element vertex 17252
property float32 x
property float32 y
property float32 z
end_header
-6.6331 4.7159 -3.6613
-7.6668 3.6868 -3.6721
-6.8233 4.8510 -3.6132
-7.2357 4.3627 -3.6467
-7.5643 3.8149 -3.6567
-6.8332 4.8581 -3.5678
.
.
.
```

Izvorni kod 12: Izgled sadržaja .ply datoteke

4. Algoritmi lokalizacije

4.1. Obitelj algoritama

Algoritmi korišteni za obrađivanje senzorskih podataka pripadaju ICP (eng. iterative closest point) obitelji algoritama. Koristi se za minimizaciju razlika dvije skupine točaka. U ovome slučaju se koristi za minimizaciju razlike između dva oblaka točaka prikupljenih pomoću lidar senzora. Ovaj algoritam radi tako da se jedan skup točaka postavi kao referentni dok se drugi skup pokušava transformirati, uz minimiziranje razlike, u referentni skup. Algoritam iterativno poboljšava transformaciju potrebnu za minimiziranje pogreške. Za računanje pogreške se mogu koristiti kvadrati razlika koordinata dviju točaka. ICP algoritam je jedan od najkorištenijih algoritama za rekonstrukcije trodimenzionalnih objekata. ICP algoritam su prvi puta predstavili Besl i Mckay (ref: https://ieeexplore.ieee.org/document/121791).

Ulaz i izlaz algoritma

Ulazi algoritma su referentni i ciljni skupovi točaka, kriterij zaustavljanja iteriranja algoritma te opcionalna inicijalna transformacija tj. translacija i rotacija. Izlaz algoritma je u pravilu matrica koja se sastoji od rotacijskih i translacijskih podataka te karakterističan fitness broj koji prikazuje koliko dobro je poravnanje dvaju skupa točaka zapravo bilo. Izgled izlaza je prikazan na martici 4.1.

$$T = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

Matrica se zapravo sastoji od rotacijske matrice veličine 3x3 koja se sastoji od elemenata r_{11} do r_{33}, te od translacijske matrice koja se sastoji od elemenata t_x, t_y i t_z.

Pseudokod ICP algoritma

Result: Transformacijska matrica
priprema skupa točaka;
for *Za svaku točka ciljnog skupa* **do**
    pronađi najbližu točku u referentnome skupu ;
    estimiraj translacijsku matricu koristeći srednju pogrešku udaljenosti točaka;
    transformiraj ciljne točke;
    if *uvjet zaustavljanja zadovoljen* **then**
        vrati trenutnu transformacijsku matricu;
    else
        iteriranje algoritma;
    end
end

Ukratko ovaj algoritam prolazi kroz svaku točku ciljnoga skupa ili podskupa točaka te traži najbližu točku u referentnome skupu. Tada estimira kombinaciju rotacije i translacije ta bi poravnao te dvije točke tako što računa srednju vrijednost kvadrata razlike koordinata točaka. Algoritam tada to izvodi za ostale točke te se zaustavlja kada dosegne uvjet zaustavljanja. Taj uvjet može biti postavljen kao broj iteracija ili kao minimalna dopuštena vrijednost pogreške.

Ovaj algoritam se može optimizirati na razne načine, a jedan od od najčešćih je da se prvo u skupovima točaka detektiraju neki lako prepoznatljivi oblici poput rubova te se tada uspoređuju samo točke unutar tih oblika. Takav način optimizacije se promatra u poglavlju 4.2.

PCL biblioteka

Za izvođenje ICP algoritma u oba primjera se koristi biblioteka PCL (eng. Point Cloud Library) koja je postala standardan način za obradu oblaka točaka. To je biblioteka otvorenoga koda koja se sastoji od implementacija raznih algoritama za obradu dvodimenzionalnih i trodimenzionalnih podataka. Sastoji se od algoritama za detekciju oblika, rekonstrukciju površina, obradu oblaka točaka. Sama biblioteka je napisana u jeziku C++ zbog vrlo visokih performansi te omogućuje izvođenje na raznim platformama od autonomnih automobila do ugrađenih računalnih rješenja u prijenosnim uređajima poput mobilnih uređaja.

4.2. Algoritmi

4.2.1. Generalizirani ICP algoritam

Opis algoritma

Ovaj način uspoređivanja skupova točaka je najjednostavniji. Ne tražimo prepoznatljive oblike niti imamo ikakve posebne optimizacije. Kao ulaz koristimo niz .ply datoteka. Svaka ta dototeka predstavlja jedan skup točaka tj. jedno očitanje lidar-a. Oblik datoteke je prikazan na slici 12. Program otvari dvije datoteke koje predstavljaju dva sljedna očitanja. Tada njihov sadržaj preda metodi koja vraća transformacijsku matricu i fitnes veličinu. Za rad s datotekama se koristi Boost biblioteka otvorenoga koda.

Izvorni kod algoritma

```
1  typedef PointXYZ PT;
2  typedef PointCloud<PT> PointCloudType;
3  typedef IterativeClosestPoint<PT, PT, double> ICP;
4
5  PointCloudType::Ptr cloud_ref(new PointCloudType());
6  PointCloudType::Ptr cloud_target(new PointCloudType());
7  PointCloudType::Ptr cloud_reg(new PointCloudType());
8
9  string root_point_clouds = "\\point_clouds\\";
10 string root_results = "\\icp_results\\";
```

Izvorni kod 13: Generalizirani ICP - konstante

U primjeru izvornoga koda 13 su definirane konstante poput putanje za spremanje rezultata i putanje s ulaznim datotekama. Također su definirani tipovi točaka `PT` kao `PointXYZ` koje će algoritam koristit te sadrže samo x, y i z koordinate. Mogu se koristiti i drugi oblici točaka. Oblak točaka `PointCloudType` je definiran pomoću prethodne definicije točke. Naposljetku se definira tip `ICP` algoritma tj. s kojim timovima podataka radi. Definiran je pomoću uređene trojke `<PT, PT, double>` što znači da uspoređuje točke tipa `PT`, a rezultate u transformacijsku matricu zapisuje kao `double` vrijednosti.

Definirane su i varijable `cloud_ref` koja pokazuje na referentni skup točaka,

`cloud_target` koja pokazuje na ciljni skup točakai `cloud_reg` koja pokazuje na skup točaka nakon poravnanja. One su tipa `boost::shared_ptr` te se kao takve predaju metodama kao pokazivači.

```
1 ICP setupICP() {
2  ICP icp;
3  icp.setMaxCorrespondenceDistance(0.05);
4  icp.setMaximumIterations(500);
5  icp.setTransformationEpsilon(1e-8);
6  icp.setEuclideanFitnessEpsilon(1);
7  return icp;
8 }
```

Izvorni kod 14: Generalizirani ICP - definicija ICP

U primjeru 14 se definira ICP algoritam tako da mu se predaju uvjeti zaustavljanja te ostali parametri. Trenutno su zadana tri uvjeta zaustavljanja, a oni su:

1. `setMaxCorrespondenceDistance` - uzima u obzir samo točke unutar zadanoga promjera u metrima
2. `setMaximumIterations` - maksimalan broj iteracija prilikom estimacije matrice za neku točku
3. `setTransformationEpsilon` - maksimalna dozvoljena pogreška

Kod u primjeru 15 koristi metode biblioteke Boost za iteriranje datoteka sa skupovima točaka te vrača vektora s njihovim apsolutnim putanjama.

Metodom `process_files` u primjeru 16 se iterira kroz datoteke te se otvaraju u parovima i njihov sadržaj tj. informacije o oblaku točaka se spremaju u globalne varijable `cloud_ref` i `cloud_target`. Na linijama 11 i 12 postavljam ICP algoritmu dodatne ulazne parametre, a to su te varijable. Konačno se pokreće algoritam te se ispituje ako je došlo do konvergencije. Do konvergencije dolazi ako su dva skupa oblaka slična tj. ako predstavljaju isti objekt ili scenu. Očekuje se da uvijek dođe do konvergencije u ovome primjeru. Ako je došlo do konvergencije, spremamo podatke u datoteku. Naposljetku se iz optimizacijskih razloga vrijednost matrice `cloud_target` sprema

```
vector<path> get_files() {
 vector<path> paths;
 path p(root_point_clouds);
 directory_iterator end_itr;
 for (directory_iterator itr(p); itr != end_itr; ++itr) {
  if (is_regular_file(itr->path())) {
   paths.push_back(itr->path());
  }
 }
 return paths;
}
```

Izvorni kod 15: Generalizirani ICP - skupljanje datoteka

Rezultat algoritma

Spremanje rezultata se vrši metodom `save_matrix` u primjeru 17. Matricu transformacije možemo dobiti pozivom `icp.getFinalTransformation()` te je ona oblika `Eigen::Matrix4d` tj. ima 4 redaka i 4 stupaca dok su elemnti tipa **`double`**. Ta matrica se tada transformira u matricu veličine 3x3 tj. izdvaja se rotacijska matrica zato što takav tip matrice ima ugrađenu metodu `eulerAngles()`. Ta metoda kao argumente prima redosljed rotacija objekta tj. redosljed osi rotacija. U ovome slučaju se prvo predaje 0 što znači da se objekt prvo rotirao oko x osi, tada se predaje što znači da se tada rotirao oko y osi i naposljetku se predaje 2 što znači da je zadnja rotacija bila oko z osi. Metoda vraća vektor od tri elementa koji predstavljaju valjanje, poniranje i skretanje. Konačno se sve te informacije spremaju u datoteku s imenom sastavljenim od dva indetifikatora očitanja tako da se zna koji skupovi točaka su uspoređivani. Struktura te datoteke je prikazana na primjeru 18. Prva linija sadrži fitnes vrijednost. Od druge do pete linije se nalazi transformacijska matrica dok se na zadnjoj liniji nalaze Eulerovi kutevi u radijanima.

Evaluacija rezultata

opis konkretnih ulaza podataka: broj datoteka duljina trajanja simulacije

prikazani na grafovima s referentnim podacima kotlin kod za generiranje estimiranih točaka grafovi za : lokaciju, euler kuteve, razlike x,y,z razlike eulerovih kuteva ispiši stvarnu duljinu putovanja ispiši estimiranu duljinu putovanja ispiši MEA

```
void load_point_cloud(string path, PointCloudType& cloud) {
 pcl::io::loadPLYFile(path, cloud);
}

void process_files(vector<path> paths, ICP icp) {
 for (long i = 0; i < paths.size() - 1; i++) {
  load_point_cloud(paths.at(i).string(), *cloud_ref);
  load_point_cloud(paths.at(i + 1).string(), *cloud_target);
  string first = paths.at(i).stem().string();
  string second = paths.at(i + 1).stem().string();
  icp.setInputCloud(cloud_ref);
  icp.setInputTarget(cloud_target);
  icp.align(*cloud_reg);
  if (icp.hasConverged()) {
   save_matrix(icp, first, second);
  }
  *cloud_ref = *cloud_target;
 }
}
```

Izvorni kod 16: Generalizirani ICP - procesiranje datoteka

```
void save_matrix(ICP icp, string first, string second) {
 Eigen::Matrix4d transformation = icp.getFinalTransformation();
 double fitness = icp.getFitnessScore();
 string filename = first + "-" + second + ".txt";
 Eigen::Matrix3d mat = mat4x4_to_3x3(transformation);
 Eigen::Vector3d rpy = mat.eulerAngles(0, 1, 2);
 save_to_file(filename, mat_to_string(transformation), fitness, rpy);
}
```

Izvorni kod 17: Generalizirani ICP - spremanje matrice

4.2.2. Algoritam 2

Opis algoritma

Ovaj algoritam radi tako da ...

```
0.0263544
    0.999735   -0.0230172  -0.00046344  -0.00394583
   0.0230173     0.999735  0.000237121  0.000806379
 0.000457859 -0.000247725            1    0.0091155
           0            0            0            1
3.14136 -3.14113 -3.11857
```

Izvorni kod 18: ICP - datoteka s rezultatom

Rezultat algoritma

Rezultati algoritma su ...

Evaluacija rezultata

U usporedbi s ground truth ovaj algoritam ...

5. Eksperimentalni rezultati

Eksperimentalni rezultati ...

6. Zaključak

Zaključak.

LITERATURA

Lokalizacija autonomnog vozila u simuliranom urbanom okruženju

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: lokalizacija, simulacija

Title

Abstract

Abstract.

Keywords: simulation, localization.