SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4390

Određivanje parametara stubišta na osnovi mjerenja laserskim daljinomjerom

Branimir Brkić

Zagreb, lipanj 2016.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 15. ožujka 2016.

ZAVRŠNI ZADATAK br. 4390

Pristupnik: **Branimir Brkić (0108059221)**
Studij: Elektrotehnika i informacijska tehnologija
Modul: Automatika

Zadatak: **Određivanje parametara stubišta na osnovi mjerenja laserskim daljinomjerom**

Opis zadatka:

U radu je potrebno implementirati algoritam određivanja parametara stubišta koji kao ulaz koristi oblak točaka dobiven iz laserskog daljinomjera. Algoritam treba odrediti širinu, nagib i visinu stuba te njihov broj, kao i postojanje ograde uz rub stuba. Potrebno je omogućiti rad algoritma s 3D i 2D laserima postavljenim na mobilnoj platformi. Algoritam je potrebno implementirati unutar ROS-a i testirati ga u simulacijskom okruženju i po mogućnosti na stvarnoj mobilnoj platformi.

Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 17. lipnja 2016.

Mentor:
Prof. dr. sc. Ivan Petrović

Djelovođa:
Doc. dr. sc. Nikola Mišković

Predsjednik odbora za završni rad modula:
Prof. dr. sc. Stjepan Bogdan

Zahvaljujem prof. dr. sc. Ivanu Petroviću, asistentu mag. ing. Kruni Lencu i kolegici Sabrini Miškulin na potpori u izradi završnog rada.

SADRŽAJ

1. Uvod

U ovom završnom radu je implementiran algoritam određivanja parametara stubišta koji kao ulaz koristi oblak točaka dobiven iz laserskog daljinomjera. Algoritam određuje širinu, nagib te visinu stuba, kao i postojanje ograde uz rub stuba. Algoritam radi s 3D i 2D laserima postavljenim na mobilnoj platformi. Algoritam je implementiran unutar ROS-a (Robotic Operating System) i testiran na stvarnoj mobilnoj platformi. Kao mobilne platforme su korišteni roboti Robotnik Guardian s HOKUYO URG-04LX 2D laserskim daljinomjerom i CLEARPATH Husky s Velodyne-32HDL 3D laserskim daljinomjerom. Okruženja koja su koristila kao podloga za programiranje u C++ programskom jeziku, su bila ROS Hydro i ROS Indigo pod Linux Ubuntu 12.04 odnosno Linux Ubuntu 14.04 64-bitnim operativnim sustavom. Za obradu oblaka točaka se koristio alat PCL v.2.8.3 (*engl.* Point Cloud Library) koji je integriran s ROS-om.

2. Postavke

2.1. O ROS-u

ROS (*engl.* Robotic Operating System) je izvorno razvijen 2007. godine pod imenom *switchyard* od SAIL-a (*engl.* Stanford Artificial Intelligence Laboratory) u potpori stanfordskom UI robotu STAIR-u. Cilj je bio integrirati razna podpodručja umjetne inteligencije. Od 2008. do 2013. na razvoju je najviše radio robotičar Willow Garage. U to vrijeme mu se pridružilo više od 20 institucija u suradnji na ROS-u. U veljači 2013. godine ROS-ov razvoj je prepušten Open Source Robotics Foundation-u.

ROS je kolekcija *software*-skih okruženja za razvoj *software*-a za robote, pružajući funkcionalnost poput operativnog sustava u heterogenoj grupi računala. ROS pruža standardne usluge operativnog sustava poput *hardware*-ske apstrakcije, kontroliranje uređaja na nižim razinama, implementaciju često korištenih funkcionalnosti, slanje i primanje poruka između procesa i upravljanje paketima podataka. Procesi u ROS-u se odvijaju u čvorovima povezanim u *graph* arhitekturi koji primaju, šalju, multipleksiraju senzorske, kontrolne, aktuatorske poruke i stanja. Usprkos reaktivnosti i niskoj latenciji u kontroli robota, ROS-ovi procesi se ne odvijaju u stvarnom vremenu, iako je moguće integrirati ROS s programskim kodom u stvarnom vremenu. *Software* u ROS-u dijelimo u tri grupe:

- Programski jezik i platformski neovisne alate za izgradnju i distribuciju *software*-a temeljenog na ROS-u.
- Implementacije ROS biblioteke klijenata poput roscpp, rospy, roslisp itd.
- Programski paketi koji sadrže programski kod povezan s aplikacijama koji koristi jednu ili vise ROS biblioteka klijenata.

I alati neovisni o programskom jeziku i glavne biblioteke klijenata (C++, Python, LISP) su besplatne za komercijalnu i istraživačku upotrebu (*open-source software*). Većina ostalih programskih paketa je licencirana pod raznim *open-source* licencama. Ti paketi implementiraju često korištene funkcionalnosti i aplikacije kao upravljačke programe za *hardware*, modele robota, tipove podataka, vid, SLAM (*engl.* Simultaneous Localization and Mapping), alate za simulacije i druge algoritme.

Glavne ROS biblioteke klijenata (C++, Python, LISP) su pristrane sustavima nalik Unix-u, najviše zbog toga što ovise o velikim kolekcijama *open-source software* zavisnosti. Za ove biblioteke klijenata jedan od podržanih operativnih sustava je Linux Ubuntu.

Verzije koje su korištene pri izradi završnog rada su ROS Hydro i ROS Indigo pod Linux Ubuntu 12.04 odnosno Linux Ubuntu 14.04 64-bitnim operativnim sustavom.

Slika 1. ROS Hydro i ROS Indigo

2.2. O PCL-u

Razvoj PCL-a (*engl.* Point Cloud Library) je započeo u ožujku 2010. godine u Willow Garage-u. Projekt se inicijalno nalazio na poddomeni Willow Garage-a, a u ožujku 2011. godine se preselio na novu web-stranicu www.pointclouds.org. PCL-ovo prvo izdanje (verzija 1.0) je izašla na tržište dva mjeseca kasnije u svibnju 2011. godine.

Alat za obradu oblaka točaka koji je integriran s ROS-om: PCL v.2.8.3 je *open-source framework* za dvodimenzionalnu i trodimenzionalnu obradu oblaka točaka i slika. Sadrži brojne *state-of-the-art* algoritme kao što su filtriranje, procjena značajki, rekonstrukcija površina, slaganje modela, prepoznavanje i segmentacija. PCL je podijeljen u nekoliko modularnih biblioteka.

Slika 2. Point Cloud Library

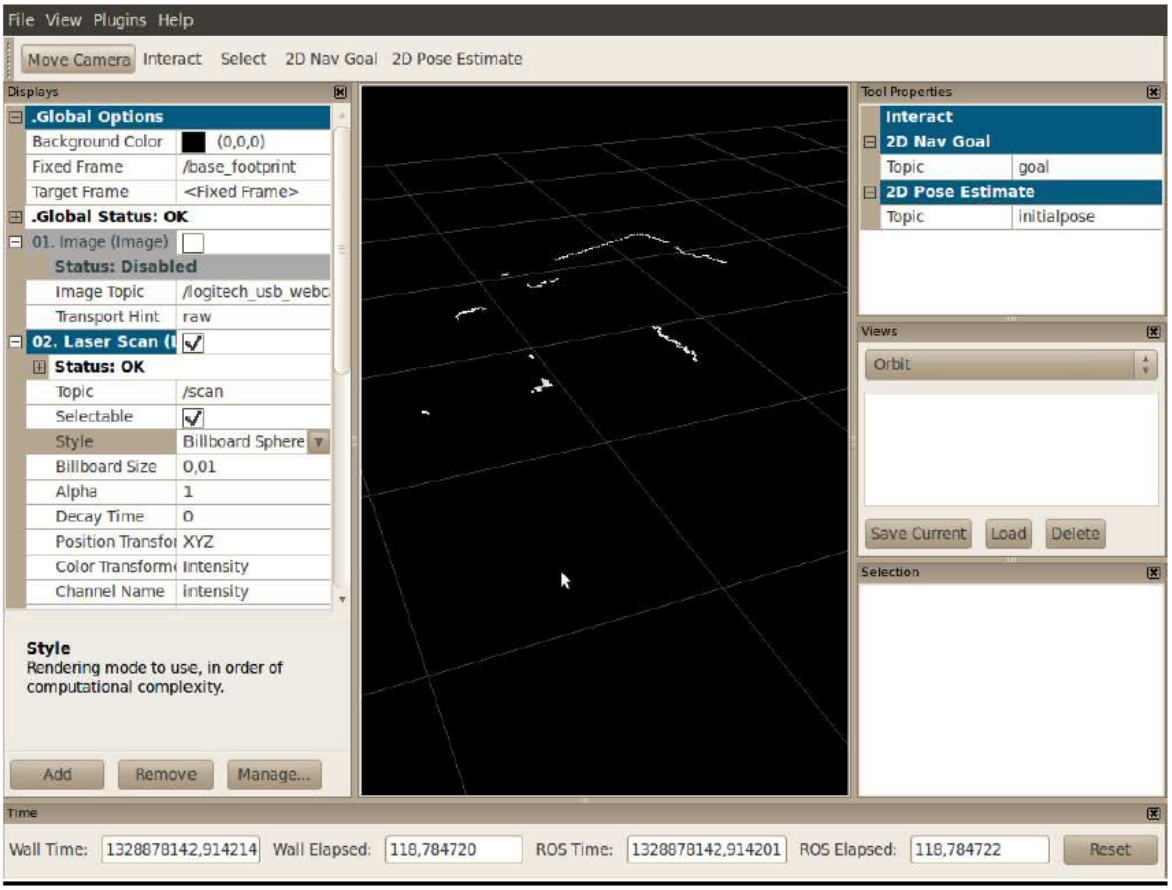
2.4. O mobilnom robotu Robotnik Guardian

Mobilni robot Robotnik Guardian je modularni robot izgrađen od tvrtke Robotnik Automation s visokom razinom mobilnosti. Ovaj mobilni robot proširuje raspon misija u kojima se robotska tehnologija koristi na više načina. Teleoperacijske mogućnosti mu omogućavaju da iz daljine izvrši različite misije i tako očuva sigurnost operatera. Dimenzije robota su dovoljno male da ga se može transportirati u konvencionalnom prtljažniku automobila, te je robot dovoljno lagan da ga se može prevoziti liftom. Guardian je pogodan i za AMCL/SLAM navigaciju. Mobilnost i velika brzina robota omogućavaju mu brz pristup građevinama. Guardian se može opremiti i laserskim daljinomjerom kako bi robot djelovao s obzirom na podražaje iz okoliša. Laserski daljinomjer koji je instaliran na robotu je HOKUYO URG-04LX koji je opisan u sljedećem potpoglavlju.

Slika 3. Mobilni robot Robotnik Guardian

2.5. O laserskim daljinomjerima

Kao mobilne platforme su korišteni roboti Robotnik Guardian s HOKUYO URG-04LX 2D laserskim daljinomjerom i CLEARPATH Husky s Velodyne-32HDL 3D laserskim daljinomjerom. Nakon što smo riješili tehničke probleme u vezi induktivnog senzora za ograničenje nagiba *pan-tilt* HOKUYO URG-04LX 2D laserskog daljinomjera, instalirali smo upravljačke programe za isti. Koristili smo rviz - 3D vizualizacijski alat za ROS i druge potrebne programske pakete koji su rekonstruirali 3D sliku iz 2D linija uslikanih zadanom frekvencijom. Naravno, za gušći oblak točaka zadajemo veću frekvenciju, no treba uzeti u obzir dulje vrijeme obrade takvog oblaka točaka.



Slika 4. Uslikana 2D linija u rviz-u

Zbog poteškoća s upravljačkim programima pogonskih motora Robotnik Guardian-a, zamijenili smo ga s CLEARPATH Husky mobilnom platformom koja ima Velodyne-32HDL 3D laserski daljinomjer. S pretpostavkom da robot ima dovoljno vremena za snimiti stube, zamjena 2D s 3D laserskim daljinomjerom je bila adekvatna.

Senzor 3D laserski daljinomjer (Velodyne-32HDL) je mali, lagan i kompaktan s 32 lasera preko 40 stupnjeva vertikalnog vidnog polja, dimenzije 5.7" x 3.4". Teži manje od dva kilograma. Dizajniran je da izdrži najizazovnije uvjete autonomne navigacije u stvarnom vremenu, za trodimenzionalno mobilno mapiranje i druge LiDAR-ske primjene.

Slika 5. Laserski daljinomjeri Velodyne-32HDL i HOKUYO URG-04LX

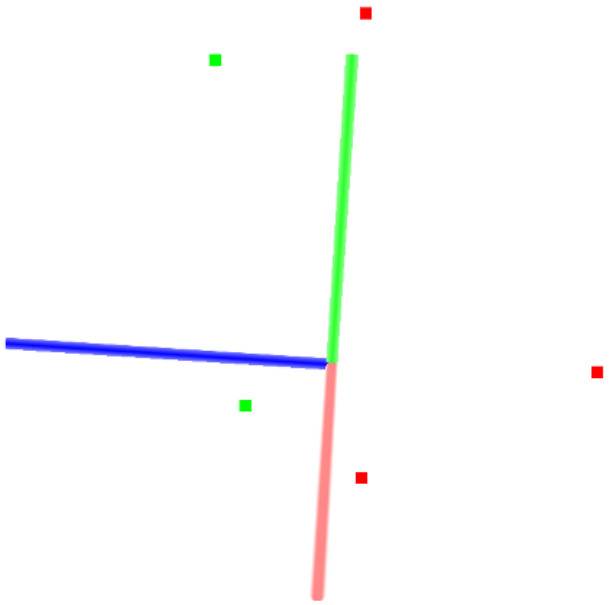
3. Algoritam

U nastavku završnog rada će biti objašnjeni svi aspekti algoritma za detekciju stuba i određivanje parametara istih. Algoritam je temeljen na nekoliko statističkih metoda te naprednijim principima računarske znanosti kao što su RANSAC (za segmentaciju ravnina iz oblaka točaka uključujući ravnine stuba), k-d stablo (za pretraživanje oblaka točaka), euklidska tehnika nakupljanja (za izdvajanje ograde stubišta) itd.

3.1. PassThrough i VoxelGrid filtri

VoxelGrid filtar gradi lokalnu 3D mrežu nad danim oblakom točaka te smanjuje uzorak održavajući konzistentnost prostora i skračujući vrijeme obrade tog oblaka točaka.

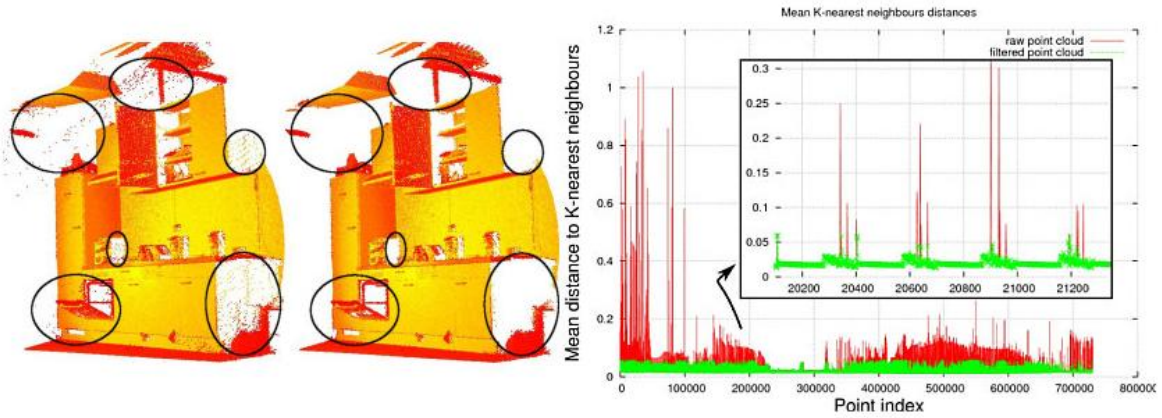
PassThrough filtar prolazi točke u oblaku po zadanim ograničenjima za jedno specifično polje tog tipa točke. Iterira kroz cijeli ulaz jednom, automatski filtrirajući ne-konačne točke i točke izvan zadanog intervala koji se samo odnosi na zadano polje. Na slici 6 možemo vidjeti primjer koordinatnog sustava s filtriranim (crvenim) točkama po "z" osi u intervalu od 0 do -∞. Primjenu u praksi možemo vidjeti na slici 17.



Slika 6. Princip rada PassThrough filtra

3.2. StatisticalOutlierRemoval filtar

Laserska mjerenja tipično generiraju setove podataka oblaka točaka s varirajućom gustoćom točaka. Uz to, greške u mjerenju mogu dovesti do raširenih *outlier*-a (točke koje ne pripadaju nijednoj grupi točaka) što može ugroziti rezultate još više. Ovo komplicira procjenu karakteristika lokalnog oblaka točaka kao što su površinske normale ili promjene zakrivljenosti, uzrokujući opadanje kvalitete što onda može uzrokovati neuspjeh u registraciji oblaka točaka. Neke nepravilnosti se mogu riješiti statističkom analizom susjeda svake od točaka i izbacivanjem onih koje ne zadovoljavaju određene kriterije. StatisticalOutlierRemoval filtar je temeljen na izračunu distribucije točaka prema udaljenosti od susjeda u ulaznom setu podataka. Za svaku točku se izračunava srednja vrijednost udaljenosti prema svim njezinim susjedima. Pretpostavljajući da je razdioba Guassova sa srednjom vrijednosti i standardnim odstupanjem, sve točke čija je srednja udaljenost izvan intervala postavljenim od globalne srednje vrijednosti i standardnog odstupanja, se mogu smatrati *outlier*-ima i miču se iz seta podataka. Na slici 7 (oblak točaka koji je poslužio kao primjer) te na slikama 11 i 15 možemo vidjeti efekte filtra.



Slika 7. Princip rada StatisticalOutlierRemoval filtra

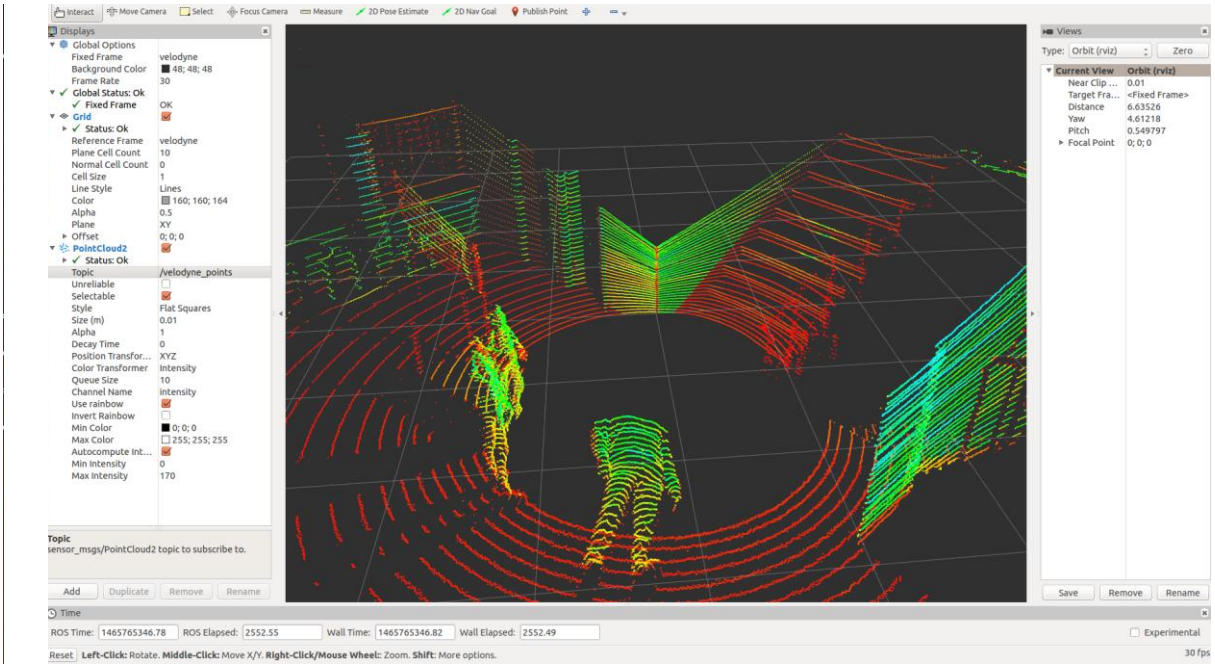
3.3. RANSAC

Za segmentaciju ravnina i linija stubišta smo koristili podalgoritam RANSAC. RANSAC (*engl.* RANdom SAmple Consesus) je iterativna metoda koja se koristi za procjenu parametara matematičkog modela iz seta podataka koji sadrži *outlier*-e. Algoritam su objavili Fischler i Bolles 1981. godine. RANSAC algoritam pretpostavlja da se svi podaci u koje gledamo sastoje od i *inlier*-a (točke koje pripadaju barem jednoj grupi točaka) i *outlier*-a. *Inlier*-i mogu biti objašnjeni modelom sa specifičnim vrijednostima parametara dok se *outlier*-i ne mogu opisati s tim modelom. Nužni uvjet koji mora biti zadovoljen kako bi RANSAC dao dobre rezultate je da se iz odabranog skupa podataka mogu odrediti parametri modela koji se koristi za odbacivanje *outlier*-a. Ulaz u RANSAC algoritam je set promatranih vrijednosti podataka, parametrizirani model koji pristaje promatranim vrijednostima i postavke algoritma. RANSAC postiže cilj iterativno izabirući podset izvornih podataka. Ti podaci su hipotetski *inlier*-i i ta hipoteza se testira na sljedeći način:

1. Testira se je li model pristajao hipotetskim *inlier*-ima npr. svi slobodni parametri modela se rekonstruiraju iz *inlier*-a
2. Svi drugi podaci se testiraju protiv tog modela i ako točka pristaje procjenjenom modelu, ona postaje hipotetski *inlier*
3. Procjenjeni model je razumno dobar ako se dovoljan broj točaka kvalificirao za hipotetske *inlier*-e
4. Model se preprocjenjuje iz hipotetskih *inlier*-a zato što se samo procjenio iz inicijalnog seta hipotetskih *inlier*-a
5. Konacno, model se ocjenjuje proračunavajući greške *inlier*-a s obzirom na model

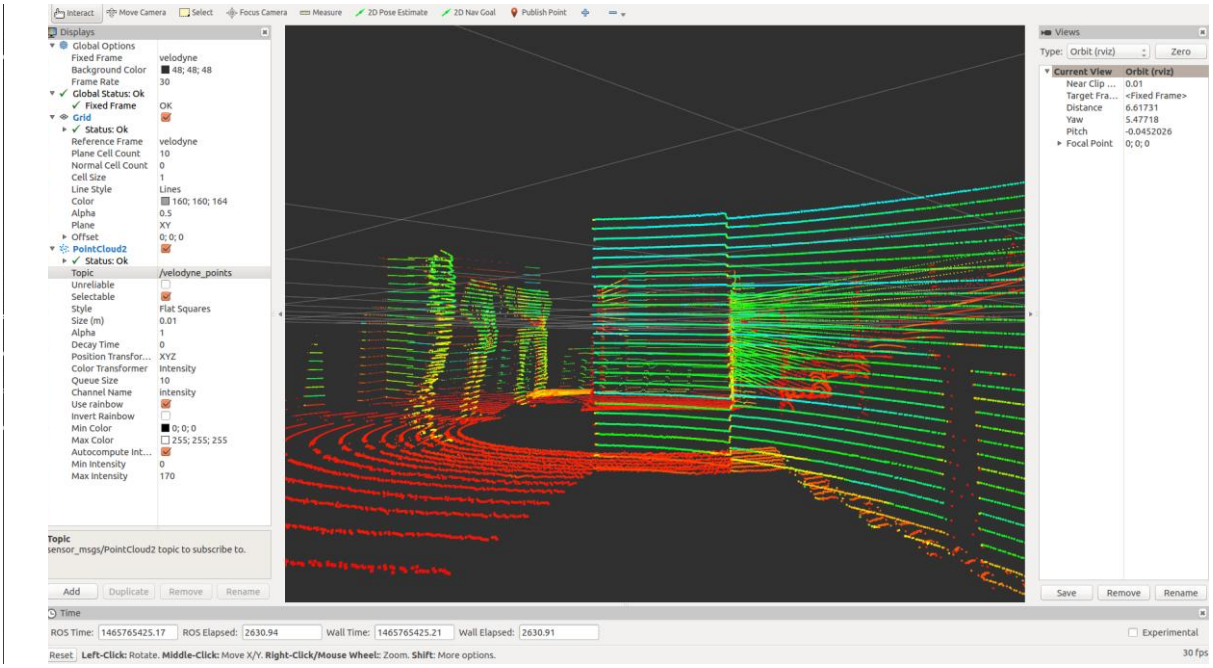
Prednost RANSAC-a je sposobnost da radi robusnu procjenu parametara modela npr. može procjeniti parametre s visokom razinom točnosti čak i kad je prisutan znatan broj *outlier*-a u setu podataka. Mana mu je što ne postoji gornja granica za izračun tih parametara. Kad je broj izračunatih iteracija ograničen, dobiveno rješenje ne mora biti optimalno. Na ovaj način RANSAC nudi kompromis: izračunavajući veći broj iteracija, vjerojatnosti stvaranja razumnog modela raste. Još jedna mana RANSAC-a je da mu se uvijek trebaju postaviti pragovi specifični za problem. RANSAC može računati samo jedan model za pojedini set podataka inače se mogu pojaviti problemi u nalaženju drugog.

U nastavku slijede eksperimentalni rezultati segmentacije ravnina i linija stubišta iz oblaka točaka dobivenih snimanjem Velodyne-32HDL 3D laserskim daljinomjerom. Rezultati su prikazani u rviz-u i pcl_viewer-u (u .pcd formatu).



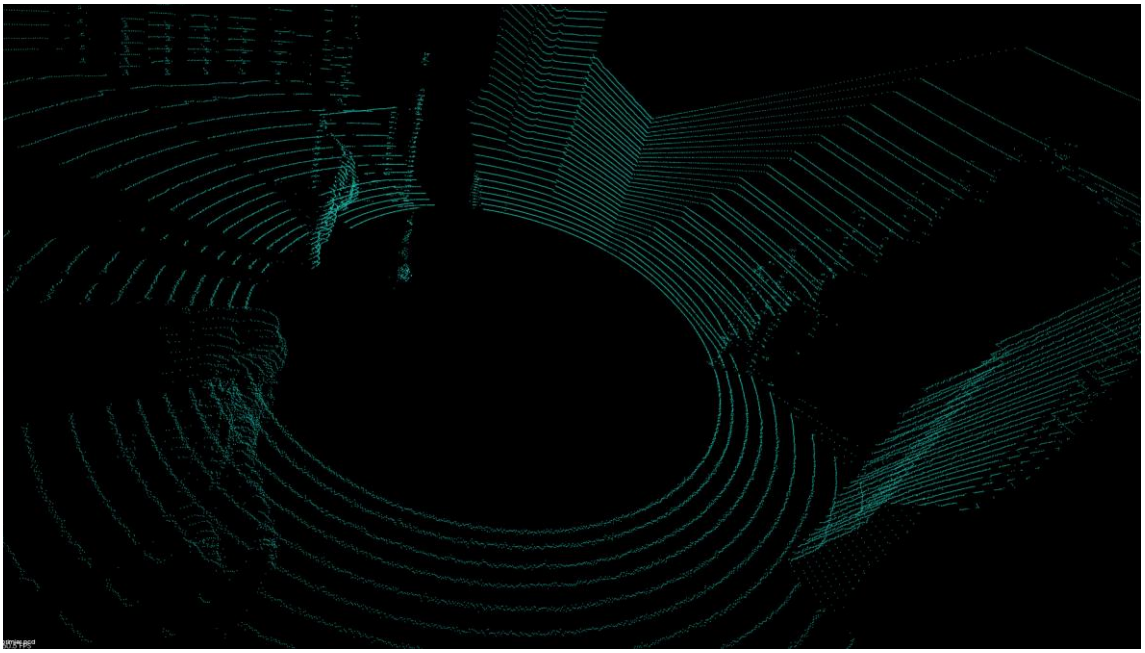
Slika 8. Oblak točaka u rviz-u (ptičja perspektiva)

Na slici 8 vidimo prizemlje C zgrade Fakulteta elektrotehnike i računarstva snimljeno pomoću 3D laserskog daljinomjera.



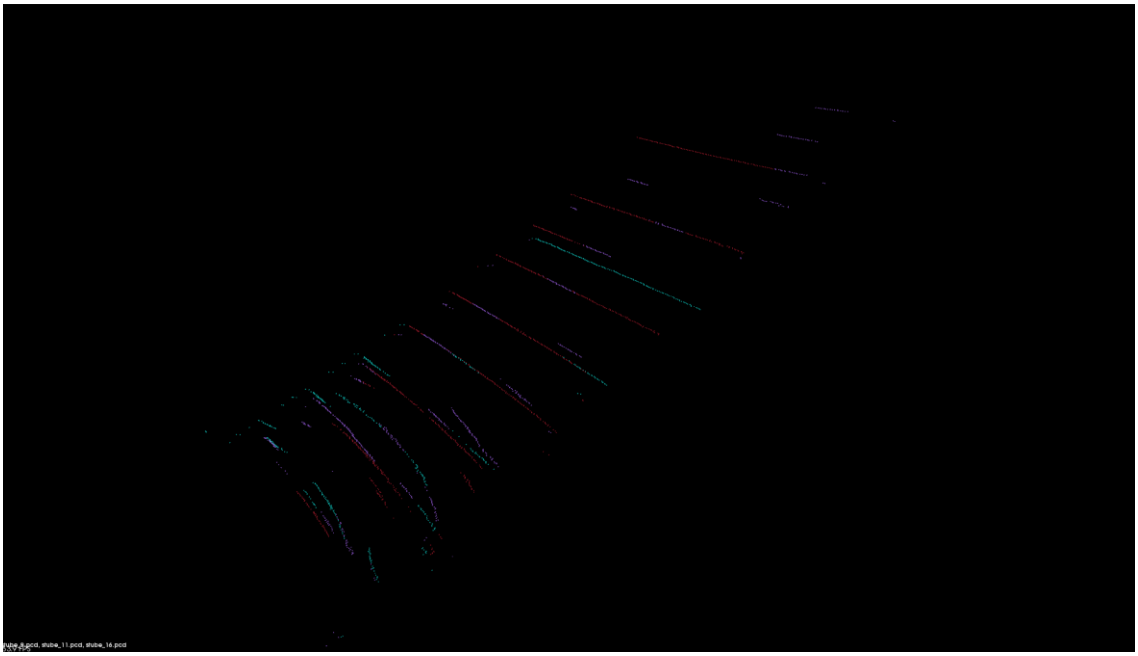
Slika 9. Oblak točaka u rviz-u (profil)

Kako je prikazano na slici 9, pod (referentna ravnina) nalazi se na -1 metar po z-osi u odnosu na laserski daljinomjer.



Slika 10. Oblak točaka u pcl_viewer-u (ptičja perspektiva)

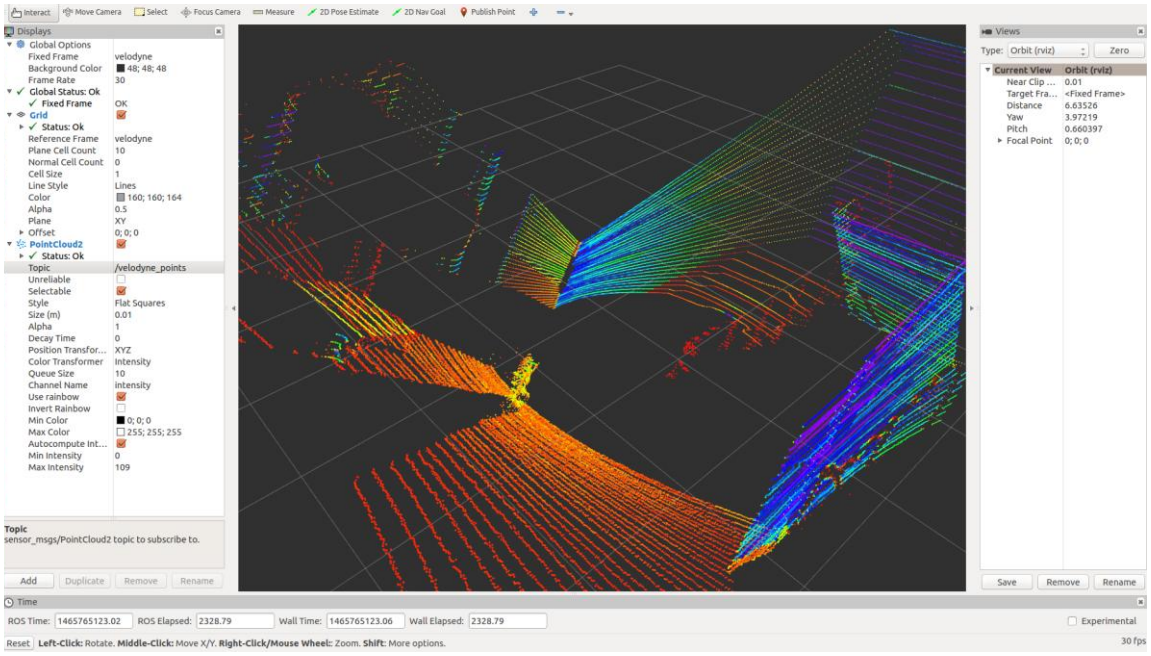
Na slici 10 vidimo prizemlje C zgrade Fakulteta elektrotehnike i računarstva snimljeno pomoću 3D laserskog daljinomjera i spremljeno u .pcd formatu.



Slika 11. Segmentirano stubište u pcl_viewer-u

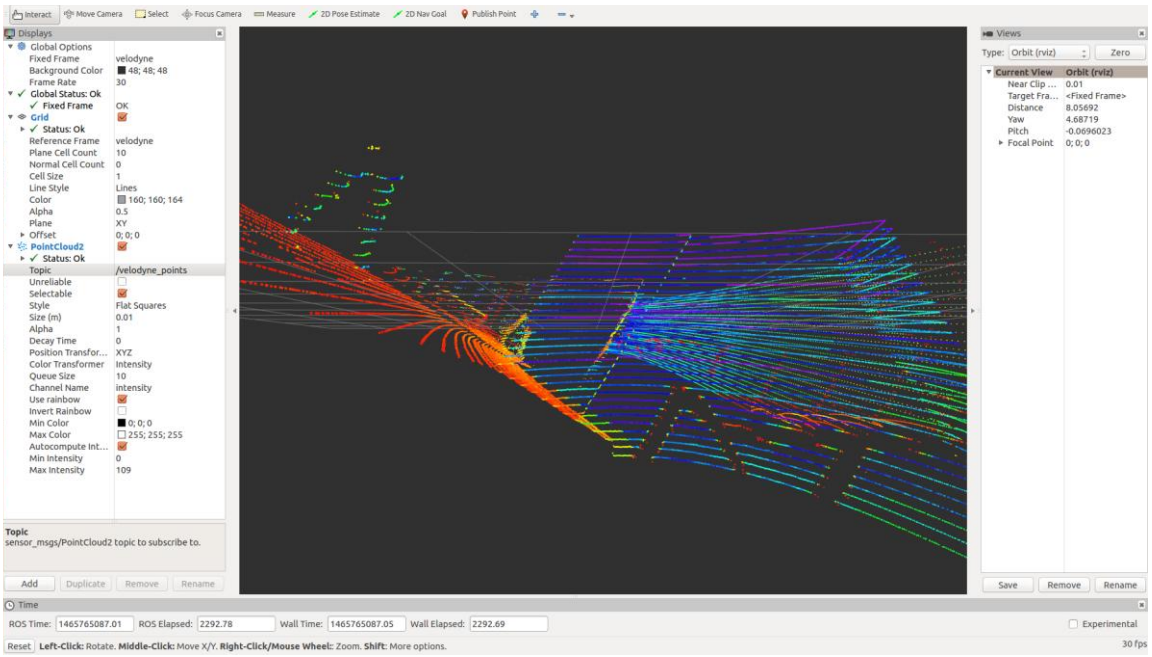
Na slici 11 mozemo vidjeti efekte RANSAC-a pri segmentaciji ravnina.

Kako bi se pripremili za probleme što nosi autonomno upravljanje mobilne platforme uz stube, modificirali smo algoritam koji uzima u obzir zakrenuti koordinatni sustav za parametar prethodno izračunat u prvotnom algoritmu. Eksperimentalni rezultati slijede u slikama 12 do 15.



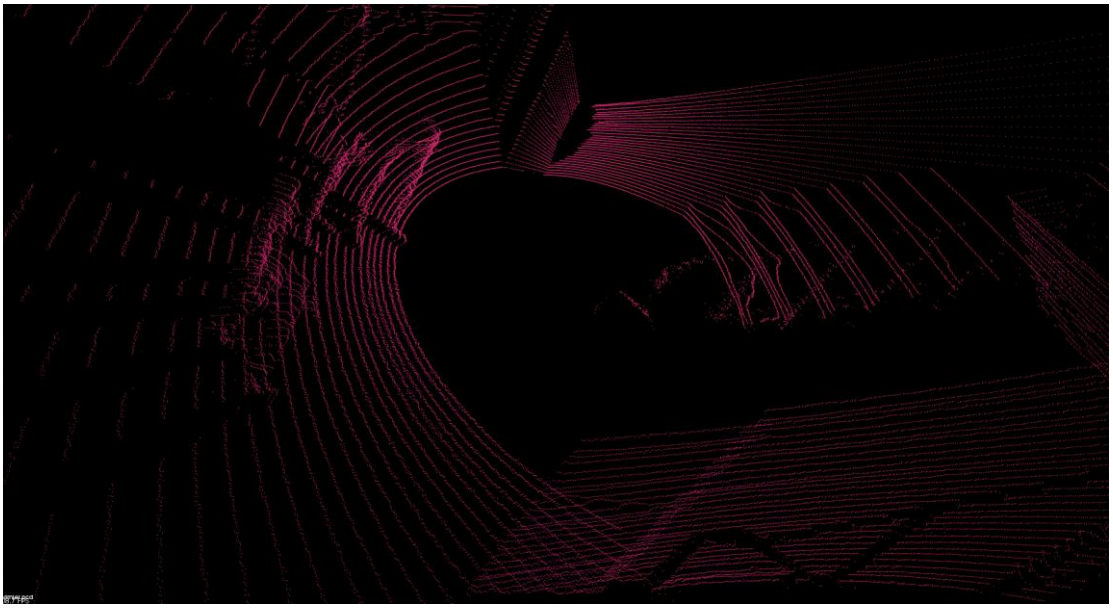
Slika 12. Oblak točaka u rviz-u (ptičja perspektiva)

Na slici 12 možemo vidjeti kako je slijepo područje 3D laserskog daljinomjera isključilo dosta točaka iz mjerenja ispred mobilnog robota.



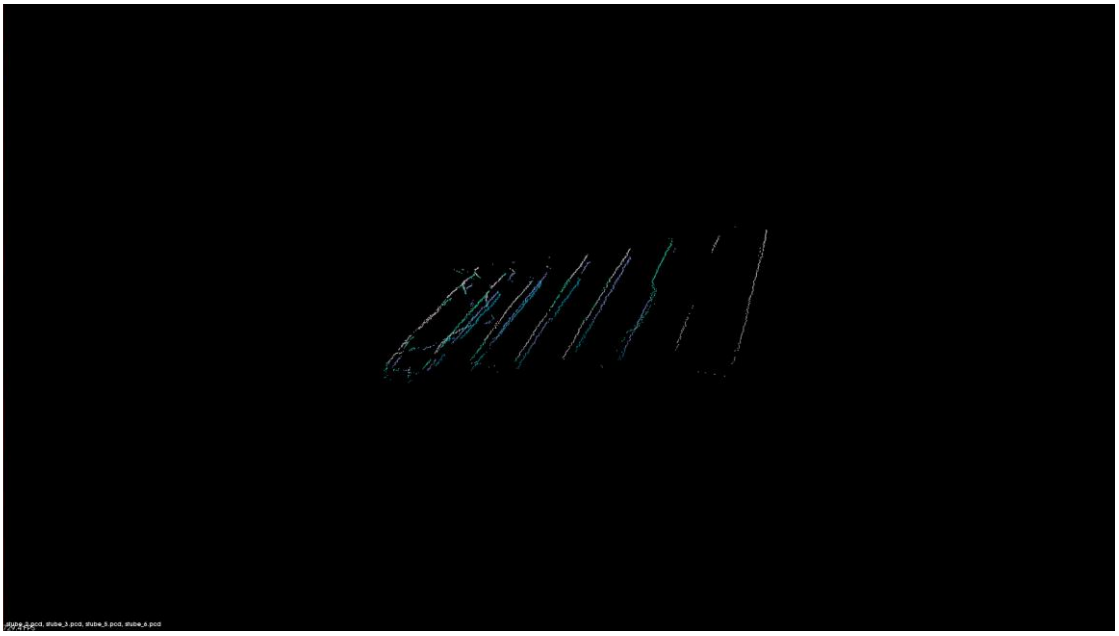
Slika 13. Oblak točaka u rviz-u (profil)

Na slici 13 se može vidjeti zakrenuti koordinatni sustav što smo trebali uzeti u obzir pri traženju ravnina stuba.



Slika 14. Oblak točaka u pcl_viewer-u (ptičja perspektiva)

Na slici 14 vidimo snimku laserskog daljinomjera u .pcd formatu kad je mobilni robot započeo uspinjanje po stubama.

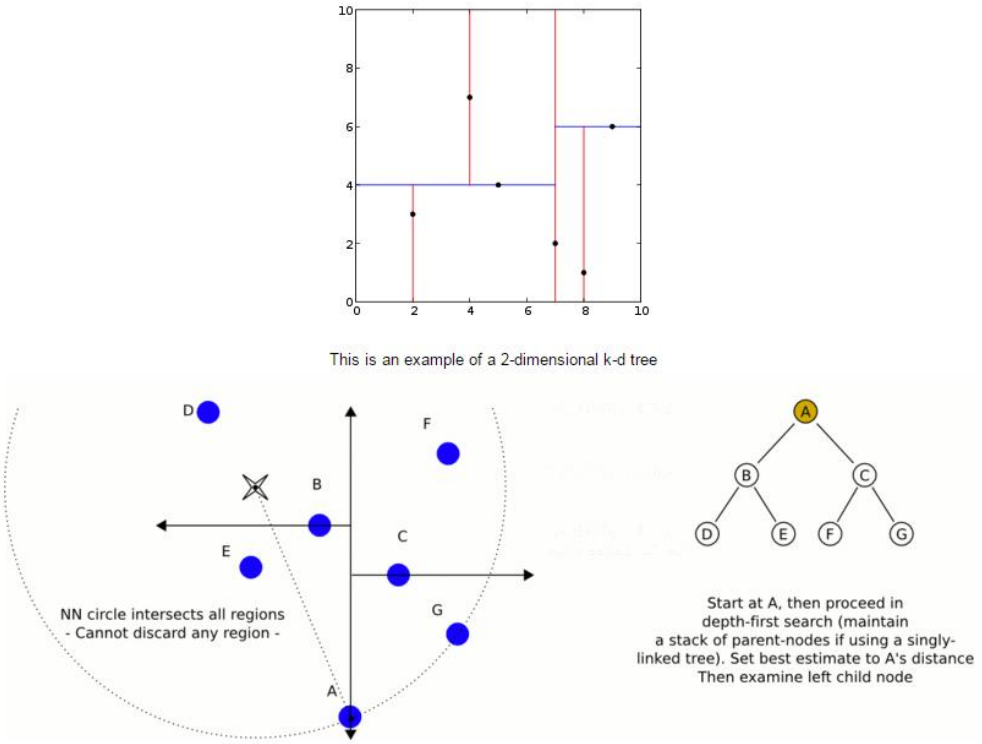


Slika 15. Segmentirano stubište u pcl_viewer-u

Na prošloj slici vidimo manju ravninu zbog slijepog područja 3D laserskog daljinomjera kad je mobilni robot započeo uspinjanje po stubama.

3.4. KdTree

K-d stablo ili k-dimenzionalno stablo je struktura podataka korištena u računarskoj znanosti za organiziranje određenog broja točaka u prostoru s k dimenzija. To je binarno stablo s nametnutim drugim ograničenjima. K-d stabla su korisna u traženju daljnjih i bližih susjednih točaka. Za našu namjenu, mi smo koristili samo oblake točaka u tri dimenzije tako da je naše k-d stablo trodimenzionalno. Svaka razina k-d stabla dijeli svu svoju djecu uzduž specifične dimenzije koristeći hiper-ravninu koja je okomita na odgovarajuću os. U korijenu stabla sva djeca su podijeljena s obzirom na prvu dimenziju (npr. koordinata prve dimenzije je manja od korijena pa je smještena u lijevo podstablo, inače bi bila smještena u desno). Svaka razina prema dolje u stablu se dijeli na sljedećoj dimenziji vračajući se u prvu dimenziju kada se sve ostale iscrpe. Najefikasniji način da se sagradi k-d stablo je koristiti metodu pregrade kao kod quicksort algoritma gdje stavljamo medijan točku u korijen i sve s manjom dimenzijskom vrijednošću lijevo odnosno većom desno. Ponavljamo proceduru i na lijevom i na desnom podstablu sve dok se zadnja preostala stabla ne sastoje od samo jednog elementa.



Slika 16. Princip rada KdTree-a

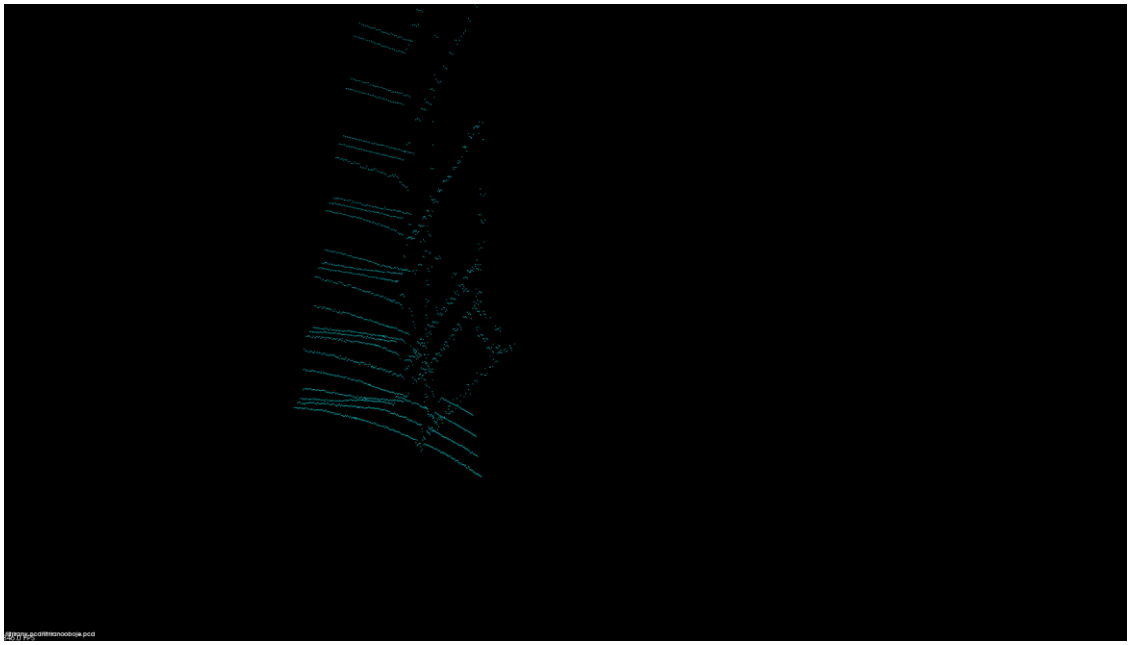
3.5. EuclideanClusterExtraction

Metoda nakupljanja (*clustering*) treba podijeliti neorganizirani model oblaka točaka *P* u manje dijelove tako da se sveukupno vrijeme obrađivanja za *P* skrati. Jednostavan pristup nakupljanju podataka na euklidski način može se implementirati koristeći podjelu prostora na 3D podmreže (kutije stalne širine) ili još češće k-d stablo. Takva struktura se može vrlo brzo izgraditi i korisna je u situacijama gdje je ili potrebna volumetrijska reprezentacija zauzetog prostora ili se podaci u svakoj rezultantnoj 3D kutiji (ili listu k-d stabla) mogu aproksimirati s drugom strukturom. Općenito govoreći, možemo iskoristiti najbliže susjede točaka i implementirati tehniku nakupljanja koja je u biti slična flood-fill algoritmu.

Nakon što smo izračunali sve parametre stubišta, koncentrirali smo se samo na parametar širine stubišta koji smo iskoristili za pretpostavku položaja ograde stubišta. Ovom smo PCL ugrađenom funkcijom našli ogradu stubišta kao nakupinu točaka prativši sljedeće korake:

1. stvoriti k-d stablo od ulaznog seta podataka oblaka točaka *P*
2. postaviti praznu listu nakupina točaka *C* i poredak točaka koje treba provjeriti *Q*
3. onda za svaku točku $\mathbf{p_i}$ koja je element *P*:
    a) dodati $\mathbf{p_i}$ poretku *Q*
    b) za svaku točku $\mathbf{p_i}$ koja je element *Q*:
        - pretraži set *P* od *i* do *k* u sferi polumjera $r < r_{zadano}$
        - za svaku susjednu točku $\mathbf{p_i}$ koja je element seta *P* od *i* do *k*, provjeriti je li točka već obrađena i ako nije, dodati ju u *Q*
    c) kad je lista svih točaka u *Q* obrađena, dodati *Q* u listu nakupina točaka *C* i resetirati *Q* na praznu listu
4. algoritam je gotov kada su sve točke $\mathbf{p_i}$, koje su elementi *P*, obrađene i dio su liste nakupina točaka *C*

U nastavku slijede eksperimentalni rezultati pronalaženja ograde uz rub stuba iz oblaka točaka koristeći k-d stablo i EuclideanClusterExtraction. Valja napomenuti kako bi rezultati bili još bolji da smo snimili gušći oblak točaka.



Slika 17. Filtrirani oblak točaka u pcl_viewer-u

Na slici 17 možemo vidjeti oblak točaka koji je prošao kroz PassThrough filtar s granicama postavljenim na temelju širine stuba.

Slika 18. Obrađeni oblak točaka u pcl_viewer-u (ograda stubišta)

Na slici 18 primjećujemo izdvojenu nakupinu točaka (ogradu stubišta) koristeći k-d stablo i EuclideanClusterExtraction.

3.6. Analitička geometrija

Nakon što smo segmentirali ravninu poda te odredili njezinu jednadžbu, tražimo sve one ravnine koje s ravninom poda zatvaraju kut do 45 stupnjeva – stubište. Za to su nam poslužile jednadžbe iz analitičke geometrije.

Kut ω između dvije ravnine

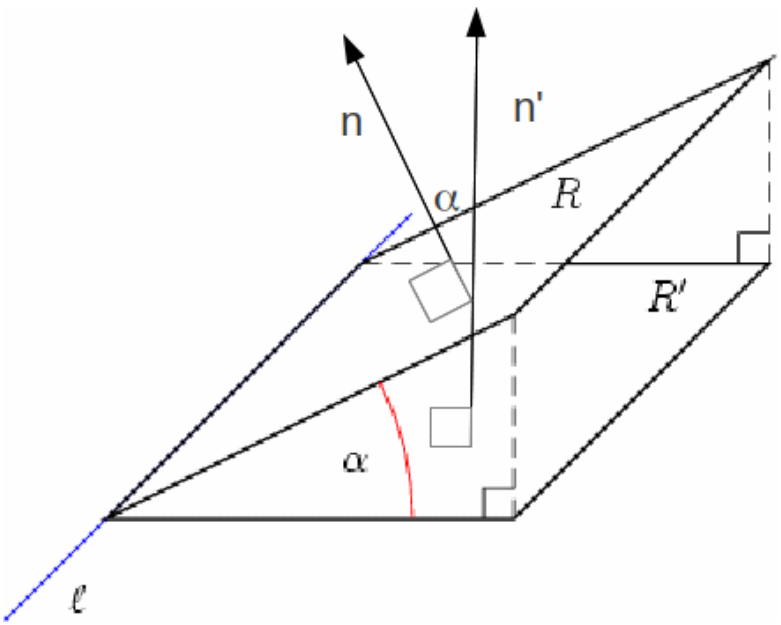
$$\begin{aligned} A_1x + B_1y + C_1z + D_1 &= 0 \\ A_2x + B_2y + C_2z + D_2 &= 0 \end{aligned} \qquad (1)$$

izračunava se pomoću jednadžbe

$$\cos\omega = \frac{A_1A_2 + B_1B_2 + C_1C_2}{\sqrt{A_1^2+B_1^2+C_1^2}\sqrt{A_2^2+B_2^2+C_2^2}} \qquad (2)$$

Ako su okomiti vektori na ravninu poznati, skalarni proizvod okomitih vektora se može upotrijebiti da bi se izračunao kut između ravnine:

$$\cos\omega = \frac{\mathbf{n_1 n_2}}{|\mathbf{n_1}||\mathbf{n_2}|} \qquad (3)$$



Slika 19. Kut između dvije ravnine

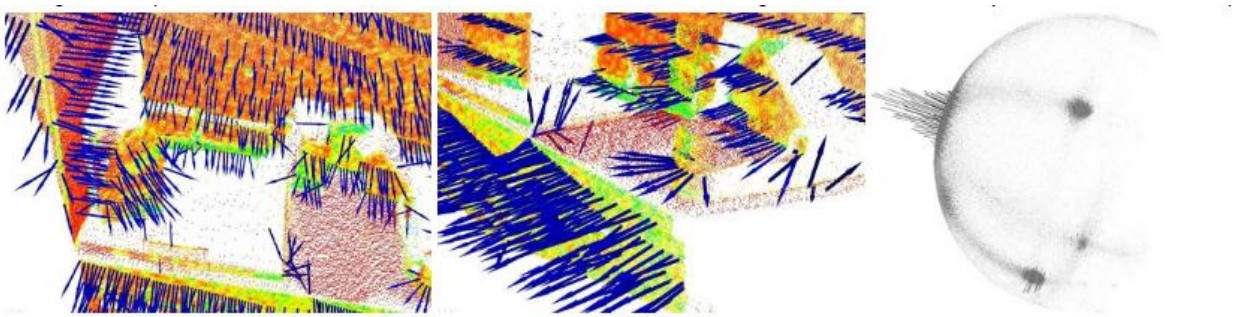
3.7. PCA

Naizgled jednostavan problem određivanja širine stuba se pokazao puno kompliciranijim. Osim segmentiranja linija iz ravnine stubišta, postupak je uključivao provođenje Principal Component Analysis ili PCA. Inače, ta se analiza provodi i pri procjeni površinskih normala i svodi se na određivanje svojstvenih vektora i svojstvenih vrijednosti matrice kovarijanci, stvorene od najbližih susjednih točaka ispitivane točke. Još preciznije, za svaku točku $\mathbf{p_i}$ sastavljamo matricu kovarijanci *C* po sljedećoj formuli:

$$C = \frac{1}{k}\sum_{i=1}^{k}(\mathbf{p_i} - \overline{\mathbf{p}}) \cdot (\mathbf{p_i} - \overline{\mathbf{p}})^T, \quad (4)$$

$$C \cdot \vec{v_j} = \lambda_j \cdot \vec{v_j}$$

Gdje je *k* broj susjednih točaka „u susjedstvu“ točke $\mathbf{p_i}$, $\overline{\boldsymbol{P}}$ je 3D centroid (centar „mase”) najbližih susjednih točaka, λ_j je j-ta svojstvena vrijednosti matrice kovarijanci i $\vec{v_j}$ je j-ti svojstveni vektor (j ∈ {0,1,2}).



Slika 20. Primjer PCA pri procjeni površinskih normala

4. Zaključak

Svi ciljevi u ovom završnom radu su bili ispunjeni tj. određeni su svi parametri stubišta i uspješno je izdvojena ograda uz rub stuba kao nakupina točaka iz ulaznog oblaka točaka u stvarnom okruženju. To nam pokazuje koliko je Point Cloud Library moćan alat mada smo tek „ogrebali površinu" što se tiče njegovih mogućnosti i brzine trodimenzionalne obrade oblaka točaka. Pri izradi ovog završnog rada uvidjeli smo važnost koliko znači kvaliteta laserskog daljinomjera, jer njime upravljamo jednim od najvažnijih aspekata robotike – robotskim vidom, kao oblikom umjetne inteligencije. Nažalost, zbog nemogućnosti osposobljenja mobilnog robota Robotnik Guardian za rad u svom punom opsegu, nije testiran puni spektar mogućnosti ovog algoritma kao npr. u autonomnom upravljanju.

LITERATURA

[1] E. Mihankhah, A. Kalantari, E. Aboosaeedan, H.D. Taghirad, S.Ali.A. Moosavian, "Autonomous Staircase Detection and Stair Climbing for a Tracked Mobile Robot using Fuzzy Controller", Proceedings of the 2008 IEEE International Conference on Robotics and Biomimetics, Bangkok, Thailand, veljača 21 - 26, 2009

[2] Joel A. Hesch, Gian Luca Mariottini, Stergios I. Roumeliotis, "Descending-stair Detection, Approach, and Traversal with an Autonomous Tracked Vehicle", IROS, 2010

[3] Mike Fair, David P. Miller, "Automated Staircase Detection, Alignment & Traversal", Proceedings of the IASTED Int'l Conference on Robotics and Manufacturing, prosinac, 2000

[4] Robotnik Automation, S.L.L., "GUARDIAN Robot Manual", kolovoz 10, 2012

[5] http://wiki.ros.org/Robots/Guardian, lipanj 14, 2016

[6] http://pointclouds.org/documentation/, lipanj 14, 2016

Naslov: Određivanje parametara stubišta na osnovi mjerenja laserskim daljinomjerom

Sažetak: U ovom završnom radu je implementiran algoritam određivanja parametara stubišta koji kao ulaz koristi oblak točaka dobiven iz laserskog daljinomjera. Algoritam određuje širinu, nagib te visinu stuba, kao i postojanje ograde uz rub stuba. Algoritam radi s 3D i 2D laserima postavljenim na mobilnoj platformi. Algoritam je implementiran unutar ROS-a (Robotic Operating System) i testiran na stvarnoj mobilnoj platformi.

Ključne riječi:
ROS, PCL, Robotnik Guardian, stube, laserski daljinomjer

Title: Staircase parameters determination based on laser rangefinder measurements

Abstract: In this bachelor`s thesis, an algorithm is implemented that determines staircase measurements using input point cloud recorded with a laser rangefinder. Algorithm determines width, slope and height of the staircase, as well as a staircase handle on the edge of stairs. Algorithm works with 3D and 2D lasers mounted on a mobile platform. Algorithm is implemented in ROS and tested on a real mobile platform.

Keywords:
ROS, PCL, Robotnik Guardian, stairs, laser rangefinder

Privitak

Zavrsni.cpp

```
#include <iostream>
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/extract_indices.h>
#include <math.h>
#include <cmath>
#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <boost/foreach.hpp>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/filters/project_inliers.h>
#include <pcl/surface/convex_hull.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/impl/point_types.hpp>
#include <pcl/sample_consensus/sac_model_perpendicular_plane.h>
#include <pcl/segmentation/extract_polygonal_prism_data.h>
#include <Eigen/Dense>
#include <pcl/sample_consensus/sac_model_parallel_line.h>
#include <pcl/sample_consensus/sac_model_line.h>
#include <pcl/common/angles.h>
#include <pcl/common/pca.h>
#include <vector>
#include <pcl/common/common.h>
#include <boost/thread/thread.hpp>

#define PI 3.14159265

/*za real-time, umjesto int main (int argc, char** argv) stavljamo void
cloud_cb (const pcl::PCLPointCloud2ConstPtr& cloud_blob)*/

//za real-time, ros::Publisher pub1, pub2;

int
main (int argc, char** argv)
{
int br=0;
 float line_length, A2, B2, C2, A3, B3, C3, D3, result, param, suma = 0, summinx =
```

```
0, summiny = 0, summinz = 0, summaxx = 0, summaxy = 0, summaxz = 0,
aritmeticka = 0, najkut = 0, najnaj = 0, vismin = 100, vismax = 0;
pcl::PointXYZ minimum, maximum;
pcl::PCLPointCloud2::Ptr cloud_blob (new pcl::PCLPointCloud2);
pcl::PCLPointCloud2::Ptr cloud_filtered_blob (new pcl::PCLPointCloud2);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new
pcl::PointCloud<pcl::PointXYZ>), cloud_p (new
pcl::PointCloud<pcl::PointXYZ>), cloud_f (new
pcl::PointCloud<pcl::PointXYZ>), cloud_l (new
pcl::PointCloud<pcl::PointXYZ>), ground_filtered (new
pcl::PointCloud<pcl::PointXYZ>);

//za real-time, treba jos dodati klase koje ce se publish-ati

// Fill in the cloud data
pcl::PCDReader reader;
reader.read ("primjer.pcd", *cloud_blob);

//std::cerr << "PointCloud before filtering: " << cloud_blob->width * cloud_blob-
>height << " data points." << std::endl;

// Create the filtering object: downsample the dataset using a leaf size of 1cm
 pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
sor.setInputCloud (cloud_blob);
sor.setLeafSize (0.01f, 0.01f, 0.01f);
sor.filter (*cloud_filtered_blob);

// Convert to the templated PointCloud
pcl::fromPCLPointCloud2 (*cloud_filtered_blob, *cloud_filtered);

//std::cerr << "PointCloud after filtering: " << cloud_filtered->width *
cloud_filtered->height << " data points." << std::endl;

// Write the downsampled version to disk
//pcl::PCDWriter writer;
//writer.write<pcl::PointXYZ> ("primjer_downsampled.pcd", *cloud_filtered,
false);

pcl::PassThrough<pcl::PointXYZ> nan_remover;
nan_remover.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZ>
>(*cloud_filtered));
nan_remover.setFilterFieldName("z");
nan_remover.setFilterLimits(-5.0, -0.5);
nan_remover.filter(*ground_filtered);

pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
pcl::PointIndices::Ptr inliers (new pcl::PointIndices ());
// Create the segmentation object
pcl::SACSegmentation<pcl::PointXYZ> seg;
```

```
seg.setOptimizeCoefficients (true);
// Mandatory
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (1000);
seg.setDistanceThreshold (0.01);

// Create the filtering object
pcl::ExtractIndices<pcl::PointXYZ> extract;

seg.setInputCloud (ground_filtered);
seg.segment (*inliers, *coefficients);

extract.setInputCloud (ground_filtered);
extract.setIndices (inliers);
extract.setNegative (false);
extract.filter (*cloud_p);

float A1 = coefficients->values[0];
float B1 = coefficients->values[1];
float C1 = coefficients->values[2];

int i = 0, nr_points = (int) cloud_filtered->points.size ();
// While 30% of the original cloud is still there
while (cloud_filtered->points.size () > 0.3 * nr_points)
{
// Segment the largest planar component from the remaining cloud
seg.setInputCloud (cloud_filtered);
seg.segment (*inliers, *coefficients);
if (inliers->indices.size () == 0)
{
std::cerr << "Could not estimate a planar model for the given dataset." << std::endl;
break;
}

// Extract the inliers
extract.setInputCloud (cloud_filtered);
extract.setIndices (inliers);
extract.setNegative (false);
extract.filter (*cloud_p);
//std::cerr << "PointCloud representing the planar component " << i << ": " <<
cloud_p->width * cloud_p->height << " data points." << std::endl;
/*std::cerr << "Model coefficients: " << coefficients->values[0] << " "
                    << coefficients->values[1] << " "
                    << coefficients->values[2] << " "
                    << coefficients->values[3] << std::endl;*/

A2 = coefficients->values[0];
B2 = coefficients->values[1];
C2 = coefficients->values[2];
```

```
param = std::abs(A1*A2 + B1*B2 + C1*C2)/sqrt((pow(A1,2) + pow(B1,2) +
pow(C1,2))*(pow(A2,2) + pow(B2,2) + pow(C2,2)));
result = acos (param) * 180.0 / PI;

std::cerr << "Kut: " << result << "\n" << std::endl;

if ((result < 45) && (result > 15)) {

if (result>najkut) {
        najkut=result;
        A3 = coefficients->values[0];
        B3 = coefficients->values[1];
        C3 = coefficients->values[2];
        D3 = coefficients->values[3];
}

pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statistical_filter;
statistical_filter.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXY
Z> >(*cloud_p));
statistical_filter.setMeanK(50);
statistical_filter.setStddevMulThresh(1.0);
statistical_filter.filter(*cloud_p);

pcl::PCDWriter writer;
std::stringstream ss;
ss << "stube_" << i << ".pcd";
writer.write<pcl::PointXYZ> (ss.str (), *cloud_p, false);

pcl::ModelCoefficients::Ptr coefficients1 (new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers1 (new pcl::PointIndices);
pcl::SACSegmentation<pcl::PointXYZ> lseg;
lseg.setOptimizeCoefficients(true);
lseg.setMethodType(pcl::SAC_RANSAC);
lseg.setModelType(pcl::SACMODEL_PARALLEL_LINE);
lseg.setDistanceThreshold(0.01);
lseg.setAxis(Eigen::Vector3f(0,0,1)); // Vertical axis
lseg.setEpsAngle(pcl::deg2rad(10.0));

int j=0, nr2_points = (int) cloud_p->points.size ();
        while (cloud_p->points.size () > 0.3 * nr2_points)
        {
        // Segment the largest planar component from the remaining cloud
        lseg.setInputCloud (cloud_p);
        lseg.segment (*inliers1, *coefficients1);
        if (inliers1->indices.size () == 0)
{
std::cout << "Could not estimate a planar model for the given dataset." << std::endl;
break;
}
```

```
// Extract the planar inliers from the input cloud
pcl::ExtractIndices<pcl::PointXYZ> ext;
        ext.setInputCloud (cloud_p);
ext.setIndices (inliers1);
ext.setNegative (false);

// Get the points associated with the planar surface
ext.filter (*cloud_l);
std::cout << "PointCloud representing the planar component: " << cloud_l-
>points.size () << " data points." << std::endl;

        // pca line dimension
        pcl::PCA<pcl::PointXYZ> pca;
        pcl::PointCloud<pcl::PointXYZ> proj;

        pca.setInputCloud (cloud_l);
        pca.project (*cloud_l, proj);

        pcl::PointXYZ proj_min;
        pcl::PointXYZ proj_max;
        pcl::getMinMax3D (proj, proj_min, proj_max);

        pcl::PointXYZ min;
        pcl::PointXYZ max;
        pca.reconstruct (proj_min, min);
        pca.reconstruct (proj_max, max);

        line_length = sqrt(pow(max.x-min.x, 2)+pow(max.y-min.y,2)+pow(max.z-
min.z, 2));
        std::cerr << "lenght: " << line_length << std::endl;
   std::cerr << "max.z: " << max.z << std::endl;
   std::cerr << "min.z: " << min.z << std::endl;

// Remove the planar inliers, extract the rest
ext.setNegative (true);
ext.filter (*cloud_f);
cloud_p.swap (cloud_f);
        j++;

        if (min.z<vismin) {
                        vismin=min.z;
        }

        if (max.z>vismax) {
                        vismax=max.z;
        }

        if (line_length > suma) {
                        suma = line_length;
```

```
if (line_length>najnaj) {
                                najnaj=line_length;
                                maximum = max;
                                minimum = min;
                                }
        }

        }

        aritmeticka += suma;
        summinx += minimum.x;
        summiny += minimum.y;
        summinz += minimum.z;
        summaxx += maximum.x;
        summaxy += maximum.y;
        summaxz += maximum.z;

        br++;

        suma = 0;

}

// Create the filtering object
extract.setNegative (true);
extract.filter (*cloud_f);
cloud_filtered.swap (cloud_f);
i++;

}

aritmeticka = (float) aritmeticka/br;
summinx = (float) summinx/br;
summiny = (float) summiny/br;
summinz = (float) summinz/br;
summaxx = (float) summaxx/br;
summaxy = (float) summaxy/br;
summaxz = (float) summaxz/br;

std::cerr << "Arimeticka sredina sirine stuba: " << aritmeticka << "m od " << br <<
" ravnine" << std::endl;
std::cerr << "\nKut stuba: " << najkut << " stupnja" << std::endl;
std::cerr << "\nUdaljenost od lijeve strane: " << sqrt(pow(minimum.x, 2) +
pow(minimum.y, 2) + pow(minimum.z, 2)) << "m\nUdaljenost od desne strane: "
<< sqrt(pow(maximum.x, 2) + pow(maximum.y, 2) + pow(maximum.z, 2)) << "m"
<< std::endl;
std::cerr << "\nVisina stubista je: " << vismax-vismin  << "m." << std::endl;
std::cerr << "\nA3: " << A3 << std::endl;
std::cerr << "\nB3: " << B3 << std::endl;
std::cerr << "\nC3: " << C3 << std::endl;
```

```
std::cerr << "\nD3: " << D3 << std::endl;

/*za real-time, stavljamo        pcl::PCLPointCloud2 outcloud;

                              pcl::toPCLPointCloud2 (*pod, outcloud);
                              pub1.publish (outcloud);

                              pcl::toPCLPointCloud2 (*cloud_pub, outcloud);
                              pub2.publish (outcloud);*/

return 0;
}

/*za real-time,

int
main (int argc, char** argv) {

// Initialize ROS
ros::init (argc, argv, "planes");
ros::NodeHandle nh;

// Create a ROS subscriber for the input point cloud
ros::Subscriber sub = nh.subscribe ("input", 1, cloud_cb);

// Create a ROS publisher for the output point cloud
pub1 = nh.advertise<pcl::PCLPointCloud2> ("pod", 1);
pub2 = nh.advertise<pcl::PCLPointCloud2> ("stube", 1);

// Spin
ros::spin ();
}*/
```

Ograda.cpp

```
#include <pcl/ModelCoefficients.h>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/filters/passthrough.h>

int
main (int argc, char** argv)
{
// Read in the cloud data
pcl::PCDReader reader;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new
pcl::PointCloud<pcl::PointXYZ>), cloud_f (new
pcl::PointCloud<pcl::PointXYZ>), ground_filtered (new
pcl::PointCloud<pcl::PointXYZ>);
pcl::PCDWriter writer;
std::stringstream ss;
reader.read ("primjer.pcd", *cloud);
std::cout << "PointCloud before filtering has: " << cloud->points.size () << " data
points." << std::endl; //*

// Create the filtering object: downsample the dataset using a leaf size of 1cm
pcl::VoxelGrid<pcl::PointXYZ> vg;
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new
pcl::PointCloud<pcl::PointXYZ>);
vg.setInputCloud (cloud);
vg.setLeafSize (0.01f, 0.01f, 0.01f);
vg.filter (*cloud_filtered);
std::cout << "PointCloud after filtering has: " << cloud_filtered->points.size ()  <<
" data points." << std::endl; //*

//PassThrough s obzirom na varijablu -((aritmeticka/2) + 0.25)
pcl::PassThrough<pcl::PointXYZ> nan_remover;
nan_remover.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZ>
>(*cloud_filtered));
nan_remover.setFilterFieldName("y");
nan_remover.setFilterLimits(-1, -0.25);
nan_remover.filter(*cloud_filtered);

pcl::PassThrough<pcl::PointXYZ> nan_remover2;
nan_remover2.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZ
```

```
> >(*cloud_filtered));
nan_remover2.setFilterFieldName("x");
nan_remover2.setFilterLimits(0, 100);
nan_remover2.filter(*cloud_filtered);

ss << "filtriranooboje.pcd";
writer.write<pcl::PointXYZ> (ss.str (), *cloud_filtered, false); //*

// Create the segmentation object for the planar model and set all the parameters
pcl::SACSegmentation<pcl::PointXYZ> seg;
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new
pcl::PointCloud<pcl::PointXYZ> ());
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (100);
seg.setDistanceThreshold (0.05);

int i=0, nr_points = (int) cloud_filtered->points.size ();
while (cloud_filtered->points.size () > 0.3 * nr_points)
{
// Segment the largest planar component from the remaining cloud
seg.setInputCloud (cloud_filtered);
seg.segment (*inliers, *coefficients);
if (inliers->indices.size () == 0)
{
std::cout << "Could not estimate a planar model for the given dataset." << std::endl;
break;
}

// Extract the planar inliers from the input cloud
pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud (cloud_filtered);
extract.setIndices (inliers);
extract.setNegative (false);

// Get the points associated with the planar surface
extract.filter (*cloud_plane);
std::cout << "PointCloud representing the planar component: " << cloud_plane-
>points.size () << " data points." << std::endl;

// Remove the planar inliers, extract the rest
extract.setNegative (true);
extract.filter (*cloud_f);
*cloud_filtered = *cloud_f;
}

// Creating the KdTree object for the search method of the extraction
```

```
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud (cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (1); // 1m
ec.setMinClusterSize (100);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);

int j = 0;
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it !=
cluster_indices.end (); ++it)
{
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->indices.end
(); ++pit)
cloud_cluster->points.push_back (cloud_filtered->points[*pit]); //*
cloud_cluster->width = cloud_cluster->points.size ();
cloud_cluster->height = 1;
cloud_cluster->is_dense = true;

std::cout << "PointCloud representing the Cluster: " << cloud_cluster->points.size
() << " data points." << std::endl;
ss << "cloud_cluster_" << j << ".pcd";
writer.write<pcl::PointXYZ> (ss.str (), *cloud_cluster, false); //*
j++;
}

return (0);
```