Robotska paletizacija primjenom računalnog vida

Rekić, Tomislav

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I
INFORMACIJSKIH TEHNOLOGIJA

Diplomski sveučilišni studij Računarstvo, izborni blok Robotika i umjetna inteligencija

Robotska paletizacija primjenom računalnog vida

Diplomski rad

Tomislav Rekić

Osijek, 2022.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 03.09.2022.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Tomislav Rekić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1082R, 06.10.2019.
OIB studenta:	06929739237
Mentor:	Prof.dr.sc. Robert Cupec
Sumentor:	Valentin Šimundić, mag. ing. comp.
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Damir Filko
Član Povjerenstva 1:	Prof.dr.sc. Robert Cupec
Član Povjerenstva 2:	Izv. prof. dr. sc. Emmanuel-Karlo Nyarko
Naslov diplomskog rada:	Robotska paletizacija primjenom računalnog vida
Znanstvena grana diplomskog rada:	**Umjetna inteligencija (zn. polje računarstvo)**
Zadatak diplomskog rada:	Izraditi računalni program za robotsku ruku UR5 opremljenu RGB-D kamerom, koji omogućuje prepoznavanje ciljnih objekata na radnoj površini, njihovo hvatanje i slaganje na paletu ili u kutiju. Izrađeni program ispitati odgovarajućim pokusima. Tema rezervirana za: Tomislav Rekić Sumentor s FERIT-a: Valentin Šimundić Sumentor iz tvrtke: Filip Novoselnik (Protostar Labs)
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	03.09.2022.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 13.09.2022.

Ime i prezime studenta:	Tomislav Rekić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1082R, 06.10.2019.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Robotska paletizacija primjenom računalnog vida**

izrađen pod vodstvom mentora Prof.dr.sc. Robert Cupec

i sumentora Valentin Šimundić, mag. ing. comp.

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Sadržaj

1. UVOD

Stupanj automatizacije unutar industrijskih postrojenja znatno je povišen u zadnjim desetljećima. Stvaraju se strojevi koji odrađuju ponavljajuće zadaće brže i preciznije od ljudi. Od strojeva ističu se robotski manipulatori, pogotovo u autoindustriji, gdje najčešće imaju unaprijed definirane kretnje koje se precizno ponavljaju. Robotski manipulatori su fleksibilnija vrsta strojeva jer se njihova funkcija može isprogramirati te tako mogu odrađivati više različitih zadaća. S napretkom dubinskih senzora, kao što su LiDAR, RGB-D i stereo kamere, roboti u industriji mogu se isprogramirati tako da dobivaju informaciju iz okoline. Mobilni roboti iz dubinske kamere mogu dobiti informaciju o preprekama ispred robota. Robotski manipulatori mogu kamerom locirati objekt kojeg je potrebno uhvatiti. Podatci sa senzora i njihova kvalitetna obrada smanjuju potrebu da se kretnja robota unaprijed definira jer se kretnja robota može dobiti inverznom kinematikom nakon što se kamerom objekt locira. Takvi pametni sustavi omogućuju veći stupanj automatizacije za zadaće koje stroj ne može lako riješiti. U ovome se radu obrađuje i implementira jedan takav sustav, za potrebe tvrtke Protostar Labs d.o.o. Sustav ima zadaću locirati objekte na radnoj površini, uhvatiti ih i odložiti na predefinirano mjesto, odnosno palete. Taj proces se još zove robotska paletizacija. Robotska paletizacija je jedan od sustava koji su ključni za naprednu automatizaciju industrijskog postrojenja. Sklopovlje sustava sastoji se od robotske ruke UR5, robotske šake Robotiq 3-Finger i od LiDAR kamere L515. Za upravljanje robotom koristi se ROS, a za rad s oblacima točaka koristi se biblioteka Open3D. Za lociranje objekata na sceni koristi se ICP algoritam.

Rad je strukturiran na sljedeći način. U drugome poglavlju dan je uvid u područje detekcije objekata na trodimenzionalnim podacima, najčešće nad oblacima točaka. Treće poglavlje sadrži opise korištenih algoritama bitnih za ovaj rad. U četvrtom poglavlju opisan je korišteni robotski sustav. Opisani su dijelovi programske podrške kao što su ROS i Open3D te dijelovi sklopovlja kao što su robotski manipulator UR5, robotska šaka Robotiq 3-Finger i LiDAR kamera L515. Unutar petog poglavlja opisana je implementacija algoritama i uspostava simulacije unutar simulatora Gazebo. Eksperimentalna evaluacija i rezultati su opisani u šestom poglavlju. Na kraju, zaključak rada nalazi se u sedmom poglavlju.

2. ISTRAŽIVANJA U PODRUČJU

Problem 3D detekcije uključuje segmentaciju scene i klasificiranje objekata na sceni. Prije nego što je razvoj računalnog sklopovlja omogućio napredne i resursno zahtjevne algoritme poput konvolucijskih neuronskih mreža (dalje CNN, engl. *Convolutional Neural Network)*, problemi 2D i 3D detekcije morali su biti riješeni klasičnijim metodama, tj. bez uporabe neuronskih mreža. U [1] Opisan je algoritam za pronalazak kvadara iz RGBD slika. Prvo, grupira slične piksele po nekoliko različitih kriterija. Iz grupa piksela pronalazi plohu koja ih najbolje opisuje koristeći RANSAC (engl. *Random Sample Consensus*) algoritam. Sljedeće, algoritam pronalazi parove ploha koje su međusobno okomite. Najmanji kvadar koji može obuhvatiti taj par plohi je potencijalni kandidat za kvadar na sceni. Pronađeni kvadri optimiziraju se i eliminiraju se netočna rješenja. Rezultat algoritma je optimalni skup kvadara pronađenih na sceni, gdje je svaki kvadar opisan svojim vrhovima. Rad [2] također predlaže rješenje za 3D detekciju objekata koristeći klasične metode. Koriste SVM (engl. *Support Vector Machine*) kao klasifikator objekata. SVM se trenira nad sintetičnim oblacima točaka koji su dobiveni iz 3D CAD (engl. *Computer-Aided Design*) modela. Pronalazak objekata na sceni radi s filterom u obliku kocke koji kliže kroz scenu i pronalazi dijelove oblaka točaka koji se podudaraju s objektima na kojima je SVM učen.

Od klasičnih metoda može se spomenuti još [3] koji prepoznaje objekte pomoću histograma. Objekti koji se prepoznaju moraju biti na dominantnoj plohi, poput stola ili poda. Dominantna ploha se u procesu lokaliziranja objekta uklanja iz oblaka točaka, nakon čega u oblaku točaka ostaju samo objekti na stolu. Takve objekte jednostavno je lokalizirati i lakše klasificirati. Za objekte na sceni generiraju se histogrami te se uspoređuju s histogramima objekata u bazi objekata. Na temelju usporedbe donose se odluke o klasi objekta na sceni.

Jednostavnija, 2D detekcija prijašnjih godina uvelike se temeljila na CNN-ovima, iako se s vremenom vizualni pretvarači [4] (engl. *Vision Transfomer*) sve više pojavljuju kao dobra rješenja za problem 2D detekcije. Uspjeh CNN-ova kod 2D detekcije inspirirao je mnoge radove da CNN-ove prilagode za 3D detekciju. Primjerice, [5] koristi CNN za generiranje pretpostavki o poziciji objekata na sceni. Kada se dobije okvirna pozicija objekta na slici, nad tim objektom pokuša se preklopiti 3D model iz baze objekata. Dobro preklapanje znači da su točke 3D modela iz baze objekta blizu odgovarajućim točkama objekta na sceni. Ovaj algoritam uspijeva klasificirati objekte i izgrađuje 3D model scene koristeći 3D modele iz baze. U [6] predstavljen je SSCNet (engl. *Semantic Scene Completion Network)*, koji kao ulazni podatak prima dubinsku sliku, a kao izlaz daje klasu svakome vokselu (engl. *Voxel*) na sceni, tj. prikazuje objekte na sceni kao grupe voksela iste klase. Time nastoji predvidjeti i dijelove objekta koji nisu

vidljivi na slici. SSCNet za učenje zahtijeva dubinske slike s volumenskim anotacijama, što znači da je potrebno anotirati izgled cijelog objekta, ne samo površine objekta koje su vidljive kameri. Za potrebe ovog rada prikupljena je baza podataka SUNCG. Baza podataka u sebi sadrži dubinske slike koje su generirane iz 3D modela raznih okruženja. Ta okruženja sintetična su, odnosno ljudski su dizajnirana unutar računalnih programa. Daljnji napredak ostvario je [7] koji predstavlja HGNet (engl. *Hierarchical Graph Network*). HGNet sastoji se od tri dijela. Na početku se nalazi GU-net, mreža U-oblika koja koristi G-konvoluciju za detektiranje značajki u oblaku točaka. U-oblik mreže se sastoji od četiri sloja koji smanjuju razlučivost podataka i dva sloja koja poslije smanjivanja povećavaju nazad razlučivost. Podatci dobiveni između slojeva se također koriste kasnije u mreži kako bi se značajke bolje pronašle. Drugi dio HGNet-a je generator pretpostavki o lokaciji objekata na sceni. Generator pretpostavki sadrži glasački modul koji za svaki objekt generira više mogućih rješenja za lokaciju tog objekta. Uzima se rješenje s najviše glasova. Tako generator pretpostavki predviđa gdje bi objekt mogao biti na sceni i gdje bi mogla biti sredina tog objekta. Treći i zadnji dio HGNet-a je modul za rasuđivanje pretpostavki i ispitivanje njihove valjanosti, što se odrađuje pomoću G-konvolucije. Od algoritama temeljenih na operaciji konvolucije, ističe se rad [8] koji predstavlja FCAF3D (engl. *Fully Convolutional Anchor-free 3D Object Detection*). FCAF3D služi za detekciju objekata na oblacima točaka unutrašnjih scena. Posebnost njihovog rada je to što se ne koriste fiksni parametri koji bi potpomogli detekciji objekata na sceni. Primjer fiksnog parametra je polumjer kružnice koju želimo detektirati kad radimo s Houghovom transformacijom za kružnice. Svi potrebni parametri za uspješnu detekciju objekata samostalno se podešavaju iz podataka dok se mreža uči. Za označavanje objekata koriste orijentirane granične okvire, zapisane na *Mobius* način.

Rad [9] predstavlja Houghovu transformaciju izvedenu pomoću neuronskih mreža, koja generira glasove za sredinu objekata na sceni. Ovdje neuronska mreža uči i odlučuje što je od važnosti za detekciju nekog objekta, dok je u klasičnom algoritmu Houghove transformacije potrebno ručno specificirati ključne značajke. Glasovi se također donose pomoću neuronske mreže. Korištenje neuronskih mreža ovdje znatno doprinosi većoj robusnosti algoritma, pa se tako može koristiti za opću detekciju objekata.

3. TEORIJSKA PODLOGA

3.1. RANSAC

RANSAC (engl. *Random Sample Consensus*) je algoritam koji iterativno nastoji doći do najboljeg rješenja za određeni skup podataka. Kod korištenja RANSAC-a za pronalazak dominantne plohe (stol, pod, ...) unutar oblaka točaka izabiru se tri nasumične točke. Te tri točke definiraju jednu ravninu **P**. Uz te tri točke koje definiraju ravninu, dodatne točke unutar oblaka točaka mogu pripadati toj ravnini. Recimo, neka točka **A** pripadati će ravnini **P** ako normala povučena od ravnine **P** prema točki **A** ne iznosi više od neke udaljenosti **l**. Ako točka **A** pripada ravnini **P**, onda ravnina **P** dobiva jedan glas. Broj glasova govori koliko je neka pretpostavka za dominantnu ravninu dobra. Algoritam RANSAC se odvija iterativno, te se tako ovaj postupak ponavlja u **N** iteracija. Najbolje dosadašnje rješenje se čuva tijekom rada algoritma, te se nakon svake iteracije uspoređuje s novim rješenjem. Na kraju **N** iteracija vraća se ravnina s najviše glasova. Ako je algoritam dobro podešen, ravnina s najviše glasova kvalitetno će opisivati dominantnu plohu na sceni.

3.2. Uklanjanje netipičnih vrijednosti

Uklanjanje netipičnih vrijednosti (engl. *Outlier Removal*) metoda je za uklanjanje šuma i nepotrebnih točaka unutar oblaka točaka. Treba biti oprezan s korištenjem metoda za uklanjanje netipičnih vrijednosti kod rada s podacima iz simulacije. Oblaci točaka unutar simulacije mogu biti previše idealni, nekad nemaju netipične vrijednosti, pa korištenje ovakve metode s lošim postavkama može dovesti do oštećenja oblaka točaka. Uklanjanje netipičnih vrijednosti se tako koristi nad oblacima točaka zabilježenim pravim senzorom, ili namjerno iskrivljenim oblacima točaka iz simulacije.

Unutar biblioteke Open3D nalaze se dva algoritma za uklanjanje netipičnih vrijednosti [10]. Prvi algoritam je uklanjanje statističko netipičnih vrijednosti (engl. *Statistical outlier removal*). Taj algoritam uklanja točke oblaka točaka koje su dalje od svojih susjednih točaka nego od prosjeka udaljenosti točaka od susjednih točaka tog oblaka točaka. Algoritam prima dva parametra, gdje je prvi parametar broj susjednih točaka neke točke koje se koriste pri izračunu udaljenosti te točke. Drugi parametar algoritma je granična vrijednost na krivulji raspodjele udaljenosti svih točaka. Tako se, u ovisnosti o graničnoj vrijednosti, odlučuje koje se točke uklanjaju, a koje ostaju.

Drugi algoritam je uklanjanje netipičnih vrijednosti na osnovu polumjera (engl. *Radius Outlier Removal*. Algoritam uklanja točke koje u svome susjedstvu imaju manje točaka od

definirane vrijednosti. Ovome algoritmu je tako potrebno predati dvije vrijednosti. Prva je najmanji broj točaka koje neka točka mora imati unutar svoga susjedstva kako bi ostala unutar oblaka točaka. Ako ima manje, onda se uklanja iz oblaka točaka. Drugi parametar koji se predaje je polumjer kugle unutar koje se susjedi za neku točku gledaju. Podešavanjem ova dva parametra mijenja se osjetljivost algoritma. Kada je parametar koji definira najmanji broj točaka veći, i kada je kugla unutar koje se susjedi gledaju manja, broj točaka koje će se eliminirati bit će veći.

3.3. DBSCAN

Ester i sur. 1996. predlažu algoritam DBSCAN [11] (engl. *Density-Based Spatial Clustering of Applications with Noise*) koji se koristi za grupiranje podataka unutar nekog skupa ili baze podataka. Pojednostavljeno, algoritam iterira kroz sve točke skupa i pronalazi za svaku točku njoj susjedne točke. Dalje, za sve susjedne točke traži daljnje susjedne točke. Tako se stvori mreža točaka koje čine jednu grupu. Potreban je parametar koji bi definirao koliko je veliko područje unutar kojeg se točke skupa smatraju susjedima jedne određene točke. Taj parametar se označuje s ϵ te on za dvodimenzionalni skup podataka predstavlja polumjer kružnice oko jedne točke unutar koje se nalaze susjedne točke. Za podatke u trodimenzionalnom prostoru parametar ϵ definira sferu, i tako dalje za veće dimenzije. Tako se dvije točke udaljene međusobno za manje od ϵ smatraju susjedima, dok se dvije točke udaljene međusobno za više od ϵ ne smatraju susjedima. Udaljenost se najčešće definira kao euklidska udaljenost između te dvije točke, ali se mogu koristiti druge funkcije. Izbor vrijednosti parametra ϵ, kao i vrsta funkcije za udaljenost, ovisi o tipu podataka s kojima se radi. Vrijednost parametra ϵ se često odredi empirijski.

Parametar ϵ jedan je od parametara koji je potreban za algoritam DBSCAN. Za rad algoritma DBSCAN još je potreban parametar koji predstavlja najmanju veličinu jedne grupe, odnosno najmanji broj točaka koje mogu formirati jednu grupu. Parametar je u radu [11] označen s *MinPts* (engl. *Minimum Points*). Ako je parametar *MinPts* jednak 1, to znači da točka q može biti dio neke grupe W, gdje je jedini član grupe točka q. Općenito, to znači da je svaka točka skupa član neke grupe, što nije dobro rješenje ako se u skupu podataka nalaze netipične točke, odnosno šum. Jedno pravilo za dobru vrijednost parametra *MinPts* je da iznosi dvostrukom iznosu dimenzije skupa podataka [12] tj.

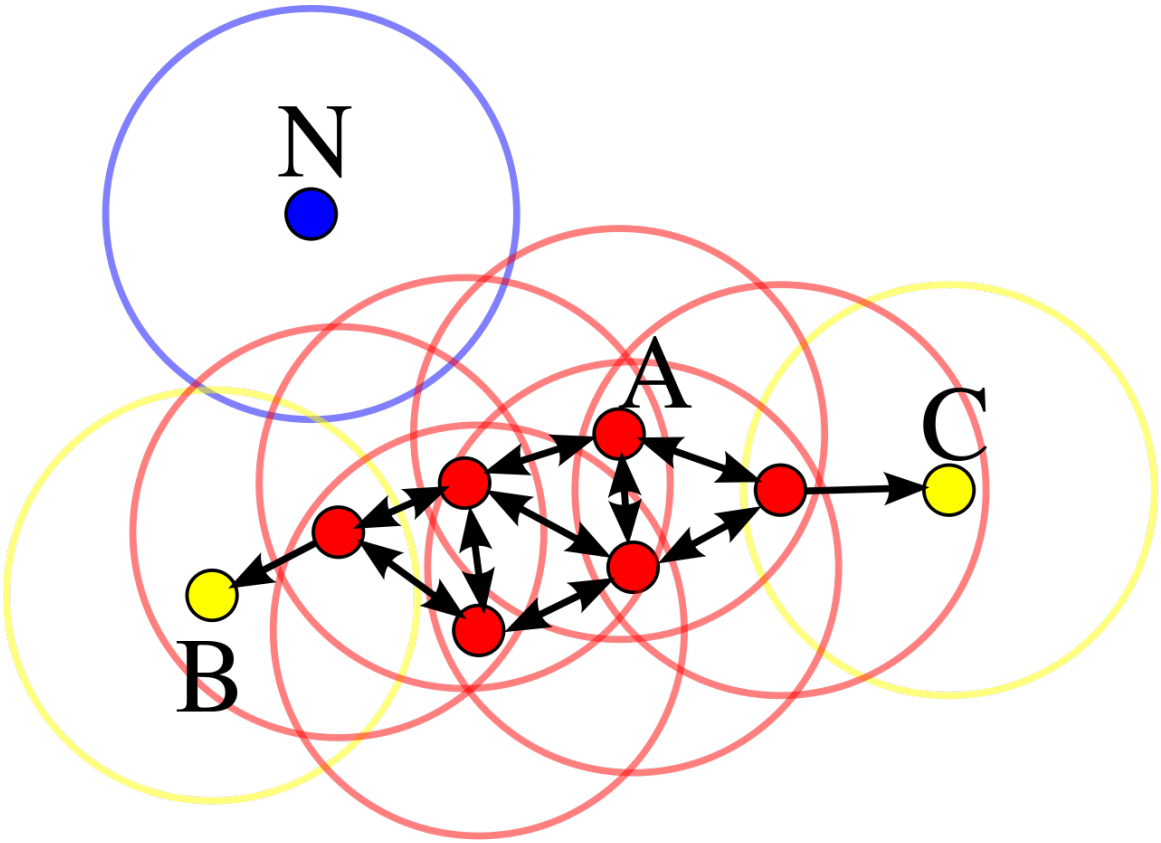
$$\mathbf{MinPts} = \mathbf{2} \cdot \mathbf{dim} \tag{3-1}$$

Gdje je **dim** broj dimenzija podataka unutar skupa. Sve točke koje u svome susjedstvu

definiranom parametrom ϵ nemaju više od *MinPts* broja točaka i ne sadržavaju u susjedstvu točku koja pripada jednoj od grupa, nisu dio jedne od grupa, te se smatraju šumom.

Sve točke koje u svome susjedstvu definiranom parametrom ϵ nemaju više od *MinPts* broja točaka, ali u susjedstvu sadržavaju točku koja pripada jednoj od grupa, smatraju se rubnim točkama (engl. *Border Points*)

Na kraju, sve točke koje u svome susjedstvu definiranom parametrom ϵ sadrže više ili jednako *MinPts* broja točaka zovu se unutarnje točke (engl. *Core Points*). Primjer rada algoritma DBSCAN može se vidjeti na slici 3.1, gdje su crvenom bojom i slovom **A** označene unutarnje točke (engl. *Core Points*) grupe. Žutom bojom i slovima **B** i **C** označene su rubne točke (engl. *Border Points*) grupe, a plavom bojom i slovom **N** označen je šum u skupu podataka.



Sl. 3.1: *Prikaz rada algoritma DBSCAN, gdje su crvene točke (A) unutarnje točke grupe, žute točke (B i C) rubne točke grupe, a plava točka (N) je šum. Slika preuzeta s Wikipedije [13].*

3.4. ICP

ICP (engl. *Iterative Closest Point*) [14] je jedan od najčešće korištenih algoritama za preklapanje trodimenzionalnih modela ili oblaka točaka. Koristi se kod trodimenzionalnog skeniranja za sparivanje snimki scene iz različitih kuteva, nakon čega se može dobiti kompletni oblak točaka

objekta, iz čega se može dobiti kvalitetan trodimenzionalni model. Koristi se i kod 3D detekcije objekata, gdje se unutar oblaka točaka traži skup točaka koji predstavlja traženi objekt. Kod ICP algoritma modeli se preklapaju na temelju njihove geometrije, iako se može koristiti i boja, kao i ostali tipovi podataka, ako je potrebno. ICP za rad treba par oblaka točaka kojeg treba preklopiti, i početnu transformaciju koja je pretpostavka stvarne transformacije. Pretpostavka ne mora biti sasvim točna, ali bolja pretpostavka uglavnom znači brži rad algoritma i kvalitetnije završno preklapanje.

U radu [15], rad ICP algoritma dijeli se u šest koraka:

- Odabir točaka:

  Kod odabira točaka postoje par različitih načina i pristupa. Prvi pristup je da se koriste sve točke modela ili oblaka točaka. Korištenje svih točaka je u pravilu računalno zahtjevno i često nepotrebno. Sljedeća metoda odabira točaka je jednoliko uzorkovanje (engl. *Uniform Sampling*). Jednolikim uzorkovanjem s pravilnim parametrima informacija koju oblak točaka prenosi ostaje uvelike nepromijenjena, ali je veličina oblaka točaka znatno smanjena. Sljedeća metoda je nasumično uzorkovanje (engl. *Random Sampling*), s novim odabranim točkama nakon svake iteracije. Postoje i metode gdje se odabiru točke s visokim gradijentom intenziteta. Takve točke često predstavljaju rubove objekta, što ih čini dobrim kandidatom. Posljednje, predstavljaju novu metodu odabira točaka gdje se točke odabiru tako da njihove normale budu što raznovrsnije. Tako se bolje uspijevaju uhvatiti manje značajke na objektu, što rezultira boljim preklapanjem objekta.

- Udruživanje točaka:

  Kod udruživanja točaka potrebno je spariti točke dva oblaka točaka na najbolji način. Prva metoda je sparivanje točke s najbližom točkom u drugom oblaku točaka. Takva metoda je jednostavna, intuitivna i radi, ali nije najbolja. Dalje je predloženo da se iz jedne točke traži najbliža točka drugog oblaka točaka u smjeru normale te točke. Metoda se tako zove gađanje normalom (engl. *Normal shooting*). Postoji i metoda gdje se točka iz prvog oblaka točaka projicira na drugi oblak točaka iz smjera pogleda kamere. Metode su ocijenjene u istome radu [15].

- Dodjeljivanje težine parovima:

  Težina se dodjeljuje parovima kako bi se odredilo koji su parovi bitniji za preciznost preklapanja. Najjednostavnija je metoda dodjeljivanje istih težina svakom paru. Druga metoda je dodjeljivanje manjih težina parovima koji su više udaljeni. Dalje postoje metode gdje se težine dodjeljuju ovisno o tome koliko su normale slično usmjerene. Postoje i metode

gdje se boje parova uzimaju u obzir. Sudeći po rezultatima dobivenim u radu [15], izbor metode dodjeljivanja težine parovima nema značajan utjecaj na kvalitetu preklapanja algoritmom ICP.

- Eliminiranje parova:

  Eliminiranje parova može pomoći kod eliminacije netipičnih (engl. *outlier*) podataka. Eliminacija netipičnih podataka može pomoći kod procesa optimizacije. Prva metoda za eliminaciju loših parova je eliminacija parova koji su udaljeni jedno od drugog više od određene udaljenosti. Ostale metode za eliminaciju loših parova su slične navedenoj, samo se razlikuju u kriteriju ili broju parova koji se eliminiraju.

- Odabir funkcije troška:

  Dvije glavne funkcije troška koje se koriste su točka-na-točku [16] (engl. *point-to-point*) i točka-na-ravninu (engl. *point-to-plane*). Neka je $\mathbf{S} = (\mathbf{p}, \mathbf{q})$ skup parova točaka iz oblaka točaka $\mathbf{P}$ i $\mathbf{Q}$, $\mathbf{T}$ transformacija koja opisuje relativni položaj tih oblaka točaka, a $\mathbf{E}(\mathbf{T})$ funkcija troška. Funkcija troška točka-na-točku [16] opisana je sljedećim izrazom.

$$\mathbf{E}(\mathbf{T}) = \sum_{(\mathbf{p},\mathbf{q})\epsilon\mathbf{S}} ||\mathbf{p} - \mathbf{T}\mathbf{q}||^{\mathbf{2}} \tag{3-2}$$

  Kod funkcije troška točka-na-ravninu postoji još normala $\mathbf{n_p}$ točke $\mathbf{p}$. Tako je funkcija troška točka-na-ravninu [16] opisana izrazom.

$$\mathbf{E}(\mathbf{T}) = \sum_{(\mathbf{p},\mathbf{q})\epsilon\mathbf{S}} ((\mathbf{p} - \mathbf{T}\mathbf{q}) \cdot \mathbf{n_p})^{\mathbf{2}} \tag{3-3}$$

  Funkcija troška točka-na-ravninu je bolja od funkcija troška točka-na-točku [16] [15].

- Optimizacija funkcije troška:

  Optimizacija funkcije troška se odnosi na način na koji će funkcija troška najbrže konvergirati minimumu. Postoje metode koje osiguravaju da funkcija troška neće završiti u lokalnom minimumu koji nije zadovoljavajući. Takve metode uglavnom rade tako da se funkcija troška optimizira više puta, pa se kao krajnji uzme bolji rezultat. Takve metode su računalno i vremenski zahtjevnije, ali se u tom slučaju funkcija troška ne mora optimizirati dok ne završi u lokalnom minimumu, već je dovoljno samo par iteracija.

3.5. Chamfer udaljenost

Chamfer udaljenost (engl. *Chamfer distance*) način je za opisivanje sličnosti između dva oblaka točaka. Chamfer udaljenost računa se tako da se za svaku točku iz oblaka točaka $\mathbf{P}$ pronađe udaljenost do najbliže točke iz oblaka točaka $\mathbf{Q}$. Također, za svaku točku iz oblaka točaka $\mathbf{Q}$ pronađe se udaljenost do najbliže točke iz oblaka točaka $\mathbf{P}$. Neka se ti nizovi udaljenosti označuju s $\mathbf{min_{PQ}}$ i $\mathbf{min_{QP}}$. Dalje se svaka vrijednost u nizu udaljenosti $\mathbf{min_{PQ}}$ i u nizu udaljenosti $\mathbf{min_{QP}}$ kvadrira i sumira. Dobivene vrijednosti $\mathbf{S_{PQ}}$ i $\mathbf{S_{QP}}$ se zbroje te se dobije Chamfer udaljenost $\mathbf{d_{CD}}$. Chamfer udaljenost [17] definirana je izrazom.

$$\mathbf{d_{CD}}(\mathbf{P}, \mathbf{Q}) = \sum(\mathbf{min_{PQ}})^{\mathbf{2}} + \sum(\mathbf{min_{QP}})^{\mathbf{2}} \tag{3-4}$$

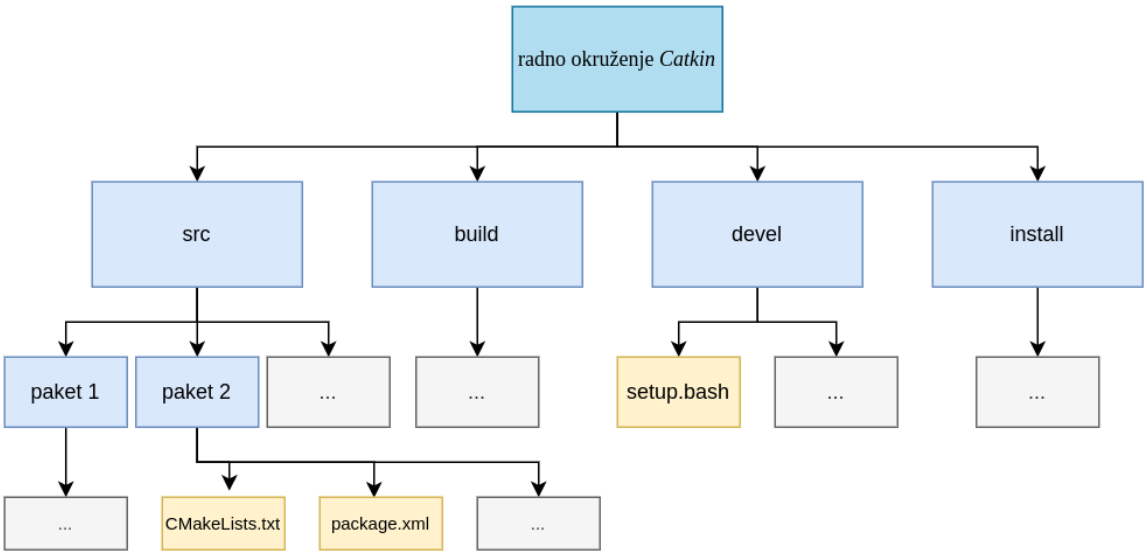
4. ROBOTSKI SUSTAV

4.1. ROS

ROS [18] (engl. *Robot Operating System*) je meta-operacijski sustav (engl. *Meta-operating System*) otvorenog koda (engl. *Open-Source*). Predstavlja sučelje za komunikaciju s fizičkim sklopovljem robota i za komunikaciju između procesa. Apstrahira fizičko sklopovlje i pomaže kod upravljanja njima. Sadrži brojne alate i gotove algoritme u obliku lagano dostupnih paketa i biblioteka. Zbog te sličnosti s operativnim sustavom u ulozi sučelja prema sklopovlju dobiva naziv meta-operacijski sustav. Programi napravljeni u ROS-u sastoje se od više zasebnih labavo spojenih (engl. *Loosely-coupled*) čvorova (engl. *Nodes*) koji mogu međusobno komunicirati. Čvorovi komuniciraju preko tema (engl. *Topic*). Čvor može objaviti temu, ili se pretplatiti na nju. Korisnik tako može napisati vlastiti programski kod koji ima određenu funkciju i pokrenuti ga kao novi čvor. Programski kodovi unutar ROS-a se uglavnom pišu u programskom jeziku *C++* ili *Python*, ali je moguće i u programskim jezicima *Lisp*, *Lua* i *Java*.

4.1.1 ROS okolina

Programi i paketi koji rade s ROS-om nalaze se u radnom okruženju *Catkin*[19] (engl. *Catkin Workspace*). Struktura radnog okruženja *Catkin* propisana je u *REP 128*[20] (engl. *ROS Enhancement Proposals*). Dijagram 4.1 prikazuje strukturu radnog okruženja *Catkin* po *REP 128*.



Sl. 4.1: *Struktura radnog okruženja Catkin*

Direktorij radnog okruženja sadrži nekoliko pod direktorija. Direktorij `src` sadrži izvorni

kod paketa. U tom direktoriju se nalaze sve konfiguracijske datoteke, datoteke za pokretanje (engl. *Launch File*), programski kodovi, opisne datoteke i drugi. Iz direktorija `src` se gradi *catkin* projekt naredbama `catkin_init` ili `catkin_make` te nastaju direktoriji `build` i `devel`. Unutar se nalaze biblioteke, paketi i ostale datoteke nužne za rad. Sadržaj tih direktorija se uglavnom ne mijenja ručno. Dodatno, postoji direktorij `install`, koji nastaje ako se pozove naredba `make_install`, te sadrži instalaciju projekta.

Roboti se unutar ROS-a opisuju URDF [21] (engl. *Unified Robot Description Format*) datotekama, koji se temelji na opisnom jeziku XML (engl. *Extensible Markup Language*). URDF sadrži nekoliko XML elementa, navedeni su oni važniji: [22]:

- **robot**, označava da se počinje opisivati struktura robota.
- **sensor**, za opisivanje senzora poput kamere, LIDAR-a i slično.
- **link**, opisuje jedan članak robota.
- **joint**, opisuje jedan zglob robota.
- **transmission**, opisuje aktuator vezan za zglob.
- **gazebo**, opisuje svojstva korištena za točniju simulaciju robota.

URDF datoteke mogu postati velike i nečitljive, pa se koristi XML makro jezik *xacro* (engl. *XML Macros*) kako bi se odvojile smislene cjeline o odvojene URDF datoteke, te kasnije povezale u jedno.

Za pokretanje programa se koriste datoteke za pokretanje (engl. *Launch File*). Datoteka za pokretanje se također piše u opisnom jeziku *XML*. Unutar datoteke za pokretanje definiraju se čvorovi i dohvaćaju se datoteke potrebne za rad programa. Neki od bitnijih elemenata su `include`, koji uključuje sadržaj druge datoteke za pokretanje, `node`, koji definira čvor koji se treba pokrenuti i `launch` koji označava početak i kraj datoteke za pokretanje.

Za pokretanje čvorova, definiranje objavitelja (engl. *Publisher*) ili pretplatitelja (engl. *Subscriber*) unutar programskog jezika *Python* može se koristiti biblioteka *rospy* [23].

4.1.2 Gazebo

Simulacija robota unutar ROS-a obavlja se u simulatoru Gazebo [24]. Gazebo nastoji ostvariti realnu simulaciju okruženja robota, gdje se objekti ponašaju po zakonima fizike. Svijet je *renderiran* u 3D.

Većina elementa simulacije sadržani su unutar `.world` datoteke, osim onih naknadno stvorenih `spawn` naredbom. Tako se recimo može unutar `.world` datoteke opisati okruženje

robota, a `spawn` naredbom stvoriti robota u tom okruženju. Svaki posebni objekt koji je naveden u `.world` datoteci uglavnom se opisuje u nekoj drugoj datoteci. Te datoteke se nazivaju datoteke modela (engl. *Model Files*). Datoteke modela i `.world` datoteke oblikovane su po uzoru na SDF (engl. *Simulation Description Format*). Gazebo se može pokrenuti naredbom `gazebo`, kojoj je potrebno predati put do željene `.world` datoteke.

4.1.3 RViz

RViz (engl. *ROS Visualization*) je 3D vizualizacijski alat za ROS. Pruža mogućnost vizualizacije podataka koje se uglavnom objavljuju na temama. Neke od stvari koje se mogu prikazati unutar RViza su trajektorije robota, model robota, karta okruženja u kojem se robot nalazi, razni podatci sa senzora, oblaci točaka, vektori i ostali [25] podatci koji pomažu robotu interpretirati svoje okruženje. Služi kao korisničko sučelje za ROS, kao i za dodatke poput MoveIt-a, koji preko RViza daje mogućnost određivanja cilja, planiranja putanje do cilja i izvršavanja putanje.

4.1.4 MoveIt

MoveIt [26] je najkorištenija programska podrška za planiranje i izvođenje kretnje robotskog manipulatora u ROS-u. Povezan je s ROS-om i koristi njegove mogućnosti i alate. Kao i ROS, kompatibilan je s većinom robotskih manipulatora; nije ciljano napisan za određene manipulatore.

MoveIt rješava problem računanja direktne i inverzne kinematike, detekcije kolizije i planiranje putanje. Zbog toga se koristi u ovome radu.

4.1.5 Vrste zglobova

Zglobovi (engl. *Joint*) unutar ROS-a povezuju dva članka (engl. *Link*). Svakome zglobu potrebno je specificirati ime, vrstu zgloba i definirati roditelj i dijete članak. Za neke zglobove je potrebno navesti dodatne parametre, poput ograničenja zgloba. Vrste zglobova su [27]:

- **revolute**: dopušta rotiranje oko jedne osi, s postavljenim ograničenjima.
- **continuous**: dopušta rotiranje oko jedne osi, bez ograničenja.
- **prismatic**: dopušta klizanje po jednoj osi, s potrebnim ograničenjem.
- **fixed**: ne dopušta kretnju. Nije zapravo zglob, ali i dalje povezuje dva članka.
- **floating**: dopušta rotiranje oko i translaciju po sve tri osi.
- **planar**: dopušta kretnju po ravnini okomitu na odabranu os.

4.1.6 Vrste kontrolera

Paket *ros_control* sadrži više vrsta kontrolera aktuatora zglobova, njihova sučelja i druge potrebne alate za njihov rad. Tri glavne vrste upravljanja su [28]:

- **upravljanje silom**: aktuatoru se predaje sila ili okretni moment koji treba izvršiti
- **upravljanje brzinom**: aktuatoru se predaje kutna ili translacijska brzina koju treba postići
- **upravljanje pozicijom**: aktuatoru se daje pozicija ili kut u kojemu treba biti

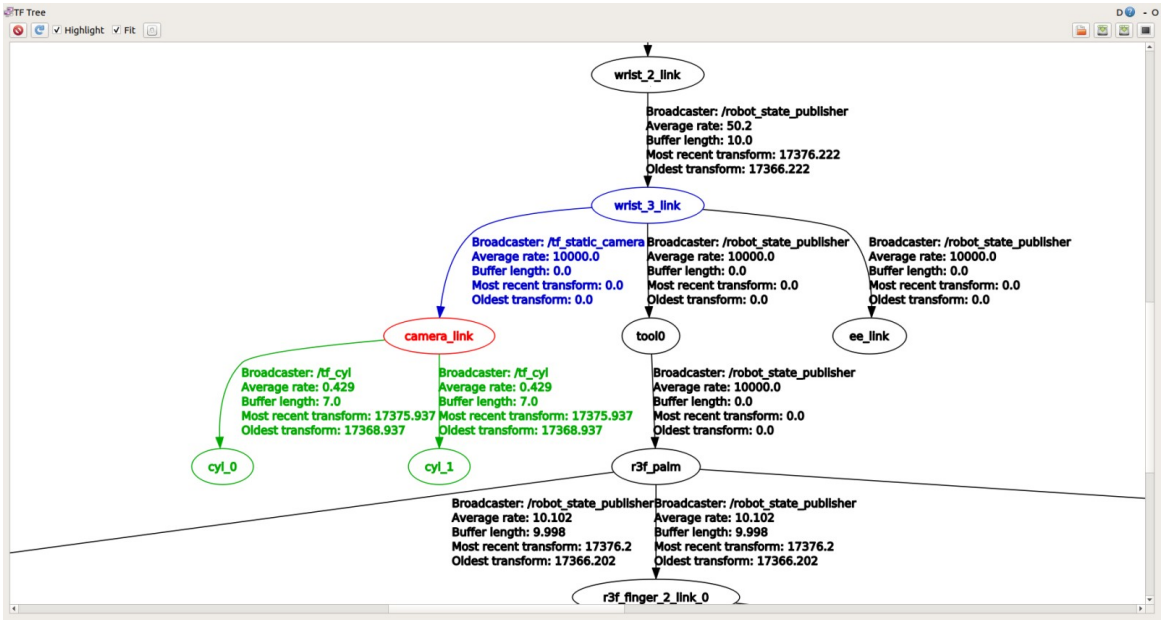
Tako u ROS-u postoje kontroleri koji su bitni za upravljanje robotom:

- **joint_state_controller**: čita stanje svih zglobova te ih objavljuje na temu */joint_states*, ne šalje podatke aktuatorima.
- **joint_trajectory_controllers**: koristi se za izvršavanje trajektorija upravljanjem grupom zglobova (npr. robotska ruka). Dalje se dijeli na:
  - **effort_controllers/JointTrajectoryController**
    Ulazna vrijednost sila ili okretni moment, potrebno definiranje PID vrijednosti
  - **velocity_controllers/JointTrajectoryController**
    Ulazna vrijednost kutna ili translacijska brzina, potrebno definiranje PID vrijednosti
  - **position_controllers/JointTrajectoryController**
    Ulazna vrijednost pozicija ili kut, **nije** potrebno definiranje PID vrijednosti

4.1.7 TF2

Biblioteka *tf2* [29], koja je nadogradnja na biblioteku *tf*, služi kako bi pojednostavila izračune kod transformacije nekog vektora u druge koordinatne sustave. Biblioteka *tf2* stvara hijerarhiju stabla koristeći okvire (engl. *Frame*) i transformaciju između njih. Okviri su najčešće koordinatni sustavi objekata na sceni, kao što su recimo koordinatni sustav kamere i koordinatni sustav objekta kojeg se hvata. Transformacija između okvira opisana je translacijskim vektorom i kvaternionom. Izgled hijerarhije stabla okvira nalazi se na slici 4.2. Okviri su opisani elipsom, dok su transformacije opisane strelicom između okvira.

Kao kod ostalih ROS paketa, komunikacija se odvija preko tema. Čitanje podataka o okvirima i transformacijama unutar stabla okvira radi se slušanjem (engl. *Listening*) transformacija, a dodavanje novih okvira i transformacija rade se objavljivanjem (engl. *Broadcasting*)



Sl. 4.2: *Hijerarhija stabla okvira*

transformacija. Kod slušanja transformacija potrebno je odrediti između koja dva okvira se transformacija traži. Tako se unutar primjera koda 4.1 nalazi funkcija `tfBuffer.lookup_transform()` kojoj se predaje imena okvira `'frame1'`, `'frame2'` i vrijeme za koje se transformacija traži.

Primjer koda 4.1: *Kod za slušanje transformacija [30]*

```
1 import tf2_ros
2
3 if __name__ == '__main__':
4     rospy.init_node('tf2_listener')
5     tfBuffer = tf2_ros.Buffer()
6     listener = tf2_ros.TransformListener(tfBuffer)
7     trans = tfBuffer.lookup_transform('frame1', 'frame2', rospy.Time())
```

Kod objavljivanja transformacija (Primjer koda 4.2) koristi se klasa `tf2_ros.TransformBroadcaster()` i njena metoda `sendTransform()` kako bi se transformacija dodala u stablo okvira. Metodi `sendTransform()` kao argument predaje se objekt klase `geometry_msgs.msg.TransformStamped()`, kojem se trebaju dodijeliti vrijednosti varijablama kao npr. `tf.header.frame_id` koji označava okvir roditelja transformacije. Okvir djeteta definira se unutra `tf.child_frame_id`, a ostale varijable su vrijeme transformacije, translacijski vektor i kvaternion.

4.2. Open3D

Biblioteka Open3D [32] je biblioteka otvorenog koda dizajnirana za rad s trodimenzionalnim podacima. To obuhvaća rad s oblacima točaka, rad s trodimenzionalnim modelima i rad s

Primjer koda 4.2: *Kod za objavljivanje transformacija [31]*

```
import rospy
import tf_conversions
import tf2_ros
import geometry_msgs.msg

if __name__ == '__main__':
    rospy.init_node('tf2_turtle_broadcaster')
    br = tf2_ros.TransformBroadcaster()
    tf= geometry_msgs.msg.TransformStamped()
    tf.header.stamp = rospy.Time.now()
    tf.header.frame_id = 'frame1'
    tf.child_frame_id = 'frame2'
    tf.transform.translation.x = 1.0
    tf.transform.translation.y = 0.0
    tf.transform.translation.z = 2.0
    q = tf_conversions.transformations.quaternion_from_euler(0.0, 1.5707, 3.1415)
    tf.transform.rotation.x = q[0]
    tf.transform.rotation.y = q[1]
    tf.transform.rotation.z = q[2]
    tf.transform.rotation.w = q[3]
    br.sendTransform(tf)
```

RGBD slikama. Sadrži kvalitetne metode za vizualizaciju i jednostavan način za navigaciju unutar tog vizualiziranog prostora. Primjerice, tijekom razvijanja algoritma detekcije objekata u oblaku točaka korištenog u ovome radu vizualizacija rješenja koristila se kako bi se vidio učinak raznih algoritama na oblak točaka. Podjelom algoritma u korake, i vizualizacija oblaka točaka između svakog koraka pomaže pri shvaćanju svakog algoritma.

Biblioteka Open3D sadrži metode za generiranje oblaka točaka iz RGBD slike, kao i iz trodimenzionalnog modela uzorkovanjem. Za generiranje oblaka točaka iz RGBD slike dodatno je potrebna matrica intrinzičnih parametara kamere. Sadrži razne metode za obradu oblaka točaka, kao što je smanjenje veličine oblaka točaka, izrezivanje oblaka točaka i određivanje dominantne ravnine pomoću algoritma RANSAC. Sadrži i metodu za izračun udaljenosti između točaka jednog oblaka točaka i točaka drugog oblaka točaka, iz čega se može izračunati Chamfer udaljenost. Unutar biblioteke nalaze se i metode za segmentaciju ili grupiranje točaka oblaka točaka algoritmom DBSCAN [11].

Dodatni korisni algoritmi za rad s oblacima točaka koje sadrži su ICP (engl. *Iterative Closest Point*) i eliminacija netipičnih (engl. *Outlier*) vrijednosti. Svi ti algoritmi često se koriste u obradi i u radu s oblacima točaka te koriste i u ovome radu u sklopu algoritma za detekciju objekata unutar oblaka točaka.

4.3. UR5

UR (engl. *Universal Robots*) proizvodi niz robota koji se dijele u CB3-seriju i E-seriju. CB3-serija se sastoji od robota UR3, UR5 i UR10, dok se E-serija sastoji od robota UR3e, UR5e, UR10e i UR16e. U svakoj oznaci, UR je kratica za *Universal Robots*, broj poslije UR je nosivost

robota u kilogramima, a *e* označuje da su roboti pripadnici E-serije. Tako je UR5, robot koji je korišten u ovome radu, pripadnik CB3-serije i ima nosivost od 5 kg.

Unutar službenih tehničkih specifikacija [33] nalaze se tehničke karakteristike UR5. Neke od izdvojenih nalaze se unutar tablice 4.1.

Tab. 4.1: *Tehničke specifikacije UR5*

Težina	18.4 kg
Nosivost	5 kg
Doseg	850 mm
Ponovljivost	0.1 mm
Stupnjevi slobode	6

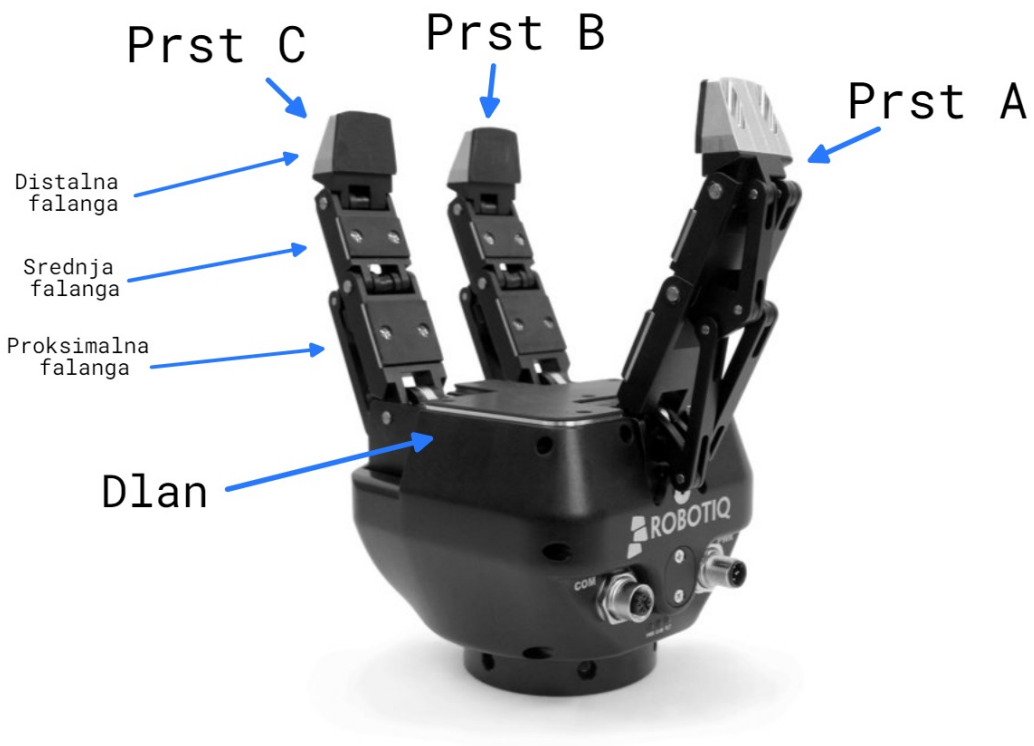
Sudeći po informacijama iz tablice 4.1, s težinom od 18.4kg i nosivošću od 5kg, UR5 je jedan od lakših robotskih ruku. Primjerice, robotska ruka FANUC R-2000 [34] s težinom od 1090 kg i nosivošću do 270 kg daleko prestiže mogućnosti UR5. Uz veću nosivost, robotska ruka FANUC R-2000 ima bolju ponovljivost (0.05 mm) te veći doseg (2655 mm). UR roboti razlikuju se od tradicionalnih industrijskih robota te se nazivaju kolaborativnim robotima (engl. *collaborative robot*). UR roboti se jednostavno programiraju, što ih čini privlačnim za edukativne ili znanstvene svrhe.

4.4. Robotiq 3-Finger

Robotiq 3-Finger je prilagodljiva robotska hvataljka s tri prsta (engl. *Three Finger Adaptive Robot Gripper*) ili robotska šaka s tri prsta. Svaki prst je dizajniran kao imitacija ljudskog prsta, te se tako sastoji od tri članka: distalna falanga (engl. *Distal Phalanx*), srednja falanga (engl. *Medial Phalanx*) i proksimalna falanga (engl. *Proximal Phalanx*). Zglobovi između tih članaka prstu daju mogućnost savijanja (engl. *Flexion*) i istezanja (engl. *Extension*). Svaki prst je također sposoban za abdukciju (engl. *Abduction*) i adukciju (engl. *Adduction*), tj. može se kretati lijevo-desno. Robotiq 3-Finger robotska šaka nalazi se na slici 4.3.

Kretanjem prstiju lijevo-desno robotska šaka omogućava više načina za hvatanje objekta. Tako ova šaka ima četiri načina rada:

- **Običan** (engl. *Basic Mode*) - Preporučeni način rada i prikazan na slici 4.3. Prsti su okomiti na dlan te se prilikom zatvaranja prekriže. Tako mogu objekt potpuno obuhvatiti, što znači da je stisak jači.



Sl. 4.3: *Robotiq 3-Finger robotska šaka [35]*

- **Široki** (engl. *Wide Mode*) - Način rada gdje se prsti C i B razmaknu kako bi omogućili hvatanje većih objekata. Prsti se također prekriže prilikom zatvaranja, kao u običnom načinu rada.
- **Štipanje** (engl. *Pinch Mode*) - Način rada gdje se prsti C i B približe jedno drugome. Kod stiska se prsti ne prekriže, već se dotaknu kod distalnih falangi. Zbog preciznosti ovaj način je prikladan za hvatanje objekata odozgo, pogotovo kad su drugi objekti u blizini objekta koji se hvata.
- **Škarice** (engl. *Scissor Mode*) - Način rada gdje se objekti hvataju između prstiju B i C, dok se prst A ne kreće. Ovaj način rada ima slabiji stisak te se koristi za manje objekte.

Težina Robotiq 3-Finger robotske šake je 2.3 kg, što je potrebno navesti prije početka rada s robotskom rukom na koju je spojena ova robotska šaka. Nosivost robotske šake prilikom obuhvaćanja objekta (engl. *Encompassing Grip*) je 10 kg, a nosivost prilikom hvatanja vrhovima prstiju (engl. *Fingertip Grip*) iznosi 2.5 kg. Takve nosivosti su primjerene za uporabu u industriji, a za istraživačke namjene sasvim dovoljne. Više mehaničkih specifikacija nalazi se unutar tablice 4.2.

Tab. 4.2: *Tehničke specifikacije Robotiq 3-Finger robotske šake [35]*

Razmak između istegnutih prstiju	167 mm
Najveći dijametar objekata koji se može obuhvatiti	155 mm
Masa robotske ruke	2.3 kg
Nosivost kod obuhvaćanja objekta	10 kg
Nosivost kod hvatanja vrhovima prstiju	2.5 kg
Najveća snaga hvata	70 N
Najveća brzina zatvaranja šake	110 mm/sec

4.5. L515 LiDAR kamera

Intel RealSense L515 LiDAR kamera [36] je kamera koja vraća RGBD informaciju o prostoru unutar njenog vidnog polja. S promjerom od 61 mm, visinom od 26 mm i masom od 100 g, jedna je od manjih RGBD kamera. Kamera se napaja preko USB 3.1, preko kojeg se vrši i prijenos podataka.

L515 LiDAR kamera koristi infracrveni laser i MEMS (engl. *Micro-Electro-Mechanical System*) koji taj laser navodi preko cijelog vidnog polja kamere, Infracrvena fotodioda detektira odbijenu lasersku zraku te informaciju o intenzitetu zrake prosljeđuje integriranom krugu posebno dizajniranom kako bi procijenio udaljenost objekta od kojeg se zraka odbila. Svi potrebni izračuni za formiranje RGBD slike se vrše na integriranim krugovima unutar L515.

Rezolucija dubinske slike dolazi u tri formata: [36]

- QVGA (engl. *Quarter Video Graphics Array*) - 320x240
- VGA (engl. *Video Graphics Array*) - 640x480
- XGA (engl. *Extended Graphics Array*) - 1024x768

Operativni opseg kamere u unutarnjem okruženju za dubinsku sliku je od 0.25 do 9 metara, osim za XGA format, gdje se operativni opseg smanjuje na vrijednost od 0.25 do 6.5 metara. Kako bi se LiDAR kamera prilagodila različitim mogućim okruženjima u kojima se može primjenjivati, snaga lasera i osjetljivost infracrvenog prijamnika može se prilagoditi. Tako postoje sljedeće predefinirane postavke: [36]

- Najveći domet (engl. *Max Range*) - Postavka koja se može koristiti u unutrašnjem okruženju u uvjetima bez pozadinskog svjetla, tj. gdje su prozori prekriveni zastorima. Ovdje se snaga laserske zrake i osjetljivost prijamnika stavljaju na najveću razinu kako bi se preciznost kamere optimizirala za velike udaljenosti.

- Kratki domet (engl. *Short Range*) - Snaga lasera i osjetljivost prijamnika smanjuju se kako se prijamnik ne bi zasitio laserskim zrakama kod rada s objektima blizu kameri.
- Nema pozadinske svjetlosti (engl. *No Ambient Light*) - Slično kao postavka za najveći domet, ali je snaga lasera smanjena da se postigne optimalni operativni opseg.
- Slaba pozadinska svjetlost (engl. *Low Ambient Light*) - Slično kao postavka za najveći domet, ali je osjetljivost prijamnika smanjena kako bi se smanjio utjecaj pozadinskog svjetla na kvalitetu slike.

RGB kamera unutar L515 vraća sliku formata YUY2 u rezolucijama 1920x1080 (30 FPS), 1280x720 (60 FPS) i 960x540 (60 FPS). [36]

5. IMPLEMENTACIJA ALGORITAMA

5.1. Uspostava robota i Gazebo simulacije

Za osposobljavanje simulacije robota u Gazebo prvo je potrebno stvoriti URDF datoteku koja će opisati strukturu robota. U ovome radu robot se sastoji od UR5 robotske ruke i Robotiq 3-Finger robotske šake. Za ruku i šaku već postoje repozitoriji koji sadrže URDF i ostale opisne datoteke potrebne za rad i simuliranje robota. Kako su ti materijali svi dostupni, potrebno je samo napisati URDF odnosno xacro datoteku koja objedinjuje ruku i šaku u novog robota. Takva datoteka prikazana je na primjeru koda 5.1.

Primjer koda 5.1: *objedinjeni UR5 i Robotiq 3f*

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="ur5_3f" >
  <xacro:arg name="transmission_hw_interface" default="hardware_interface/
      ↪ EffortJointInterface"/>

  <!-- common stuff -->
  <xacro:include filename="$(find ur5_3f)/ur5_3f_description/urdf/common.gazebo.xacro"/>

  <!-- ur5 -->
  <xacro:include filename="$(find ur5_3f)/ur5_3f_description/urdf/ur5.urdf.xacro"/>

  <!-- arm -->
  <xacro:ur5_robot prefix="" joint_limited="false"
    transmission_hw_interface="$(arg transmission_hw_interface)"/>

  <!-- Robotiq from ROS industrial repos -->
    <xacro:include filename="$(find robotiq_3f_gripper_visualization)/cfg/
        ↪ robotiq_hand_macro.urdf.xacro"/>
    <xacro:robotiq_hand prefix="r3f_" reflect="" parent="tool0">
      <origin xyz="0 0 0.055" rpy="${pi/2} 0 0"/>
    </xacro:robotiq_hand>

  <link name="world"/>
  <joint name="world_joint" type="fixed">
    <parent link="world"/>
    <child link = "base_link"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
  </joint>

  <!-- plugin for the RobotiQ hand -->
  <gazebo>
    <plugin name="robotiq_hand_plugin" filename="libRobotiqHandPlugin.so">
      <kp_position>10.0</kp_position>
      <kd_position>0.5</kd_position>
      <prefix>r3f_</prefix>
      <topic_command>/Robotiq3FGripperRobotOutput</topic_command>
    </plugin>
  </gazebo>
</robot>
```

Na početku otvaramo element `robot` i dajemo ime robotu `ur5_3f`. Onda u argumentu `transmission_hw_interface` opisujemo koju vrstu prijenosa robot treba imati. Ovdje se odabire `PositionJointInterface` zbog jednostavnosti njegova korištenja. Dalje se uključuju dodatne datoteke `common.gazebo.xacro`, `ur5.urdf.xacro` i predaje im se argument koji odgo-

vara željenom obliku prijenosa. S time je dodana robotska ruka UR5. Dalje se dodaje datoteka `robotiq_hand_macro.urdf.xacro` koja uključuje robotsku šaku Robotiq 3f. Potrebno je podesiti parametre `prefix`, koji je proizvoljan, i parametar `parent` koji definira na koji članak se ruka veže. Ovdje je to članak `tool0`, koji se nalazi na vrhu robotske ruke. Potrebno je podesiti udaljenost središta koordinatnog sustava članka `tool0` od početnog članka robotske šake. Ta udaljenost se podesi unutar elementa `origin`. Sljedeće se dodaje članak `world` i preko fiksnog zgloba `world_joint` povezuje se članak `world` s člankom `base_link`, koji predstavlja prvi članak UR5 robotske ruke. Na kraju, potrebno je navesti dodatak (engl. *plugin*) `robotiq_hand_plugin` koji se koristi za upravljanje šakom.

Kako bi se izbjegla neželjena vibracija šake potrebno je dodati unutar datoteke `ur.gazebo.xacro` kod 5.2

Primjer koda 5.2: *Potreban dodatak koda*

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://wiki.ros.org/xacro">
3
4    <xacro:macro name="ur_arm_gazebo" params="prefix">
5
6      <!-- ... -->
7
8      <gazebo reference="${prefix}tool0">
9        <selfCollide>false</selfCollide>
10     </gazebo>
11
12   </xacro:macro>
13
14 </robot>
```

Potreban dodatak koda nalazi se u linijama od 8 do 10. Navedeni kod onemogućuje samo-koliziju članka `tool0`.

Za pokretanje ruke koristi se planer kretnje MoveIt[37]. Za konfiguraciju MoveIt-a koristi se `MoveIt Setup Assistant` čija je konfiguracija podijeljena u sljedeće korake:

- **Start:** potrebno je učitati URDF datoteku koja opisuje robota.
- **Self-Collisions:** `Generate Collision Matrix` će sam generirati matricu kolizije, koja je dovoljno dobra.
- **Virtual Joints:** Preskočiti.
- **Planning Groups:** dodati grupu `arm`. Postaviti *Kinematic Solver* na `kdl_kinematics_plugin/KDLKinematicsPlugin` te dodati kinematički lanac (engl. *Kinematic Chain*) od `base_link` do `ee_link`.
- **Robot Poses:** dodati određene položaje za robota. Opcionalno.

- **End Effectors:** Preskočiti.
- **Passive Joints:** Dodati kao pasivne sve zglobove robotske šake.
- **ROS Control:** Dodati kontrolere za robotsku ruku.
  Potrebni su `arm_controller` tipa *FollowJointTrajectory* i `arm_controller` tipa *position_controllers/JointTrajectoryController*.
- **Simulation:** Preskočiti
- **3D Perception:** Preskočiti
- **Author Information:** Upisati podatke
- **Configuration Files:** Generiranje paketa s konfiguracijskim datotekama

Nakon što se odrade navedeni koraci potrebno je odraditi promjene u nekoliko datoteka. Unutar datoteke `gazebo.launch` potrebno je zamijeniti `spawn_gazebo_model` s onim unutar koda 5.3, unutar datoteke `ros_controllers.launch` kod učitavanja kontrolera potrebno je kod argumenta `args` dodati `joint_state_controller arm_controller`. Na kraju, `demo_gazebo.launch` proširujemo s kodom 5.4 te datoteku spremamo kao `custom_gazebo.launch` unutar vlastitog paketa *ur5_3f*. Dodatak koda 5.4 pokreće python skriptu koja u RViz dodaje postolje za robota, kako bi se izbjeglo planiranje putanje ispod robota.

Primjer koda 5.3: *Zamjena spawn naredbe unutar* `gazebo.launch`

```
<?xml version="1.0"?>
<launch>
  <!-- ... -->
  <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model" args="-urdf -param
      ↪  robot_description -model robot -x 0 -y 0 -z 0.05 -J shoulder_lift_joint -1.57
      ↪ -unpause"
    respawn="false" output="screen"/>
  <!-- ... -->
</launch>
```

Primjer koda 5.4: *Dodatak koda unutar* `custom_gazebo.launch`

```
<?xml version="1.0"?>
<launch>
  <!-- ... -->
  <!-- Spawn rviz moveit models -->
  <node name="spawn_models" pkg="ur5_3f" type="spawn_moveit_models.py" output="screen"/>
</launch>
```

Na kraju, potrebno je otvoriti dva terminala u početnom direktoriju radnog okruženja *Catkin* te pozvati sljedeće *bash* naredbe:

- Prvi i drugi terminal prozor inicijalizirati s:

  `$ source devel/setup.bash`

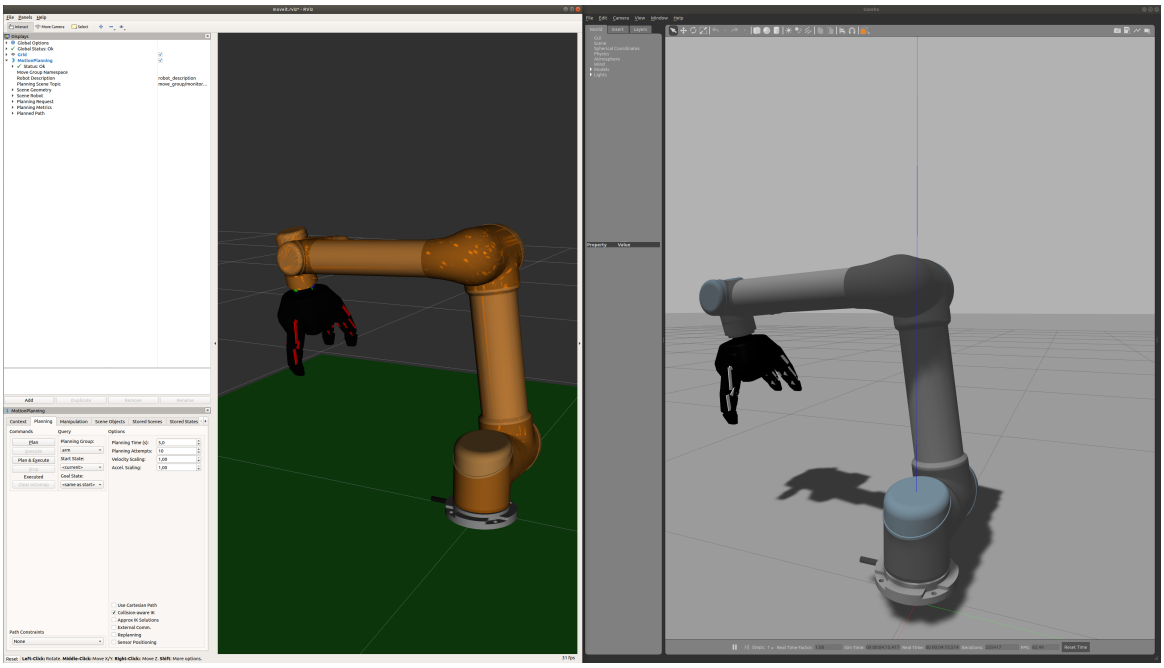
- Prvi terminal:

  `$ roslaunch ur5_3f custom_gazebo.launch`

- Drugi terminal:

  `$ rosrun robotiq_3f_gripper_control Robotiq3FGripperSimpleController.py`

Prvi terminal otvara RViz i Gazebo. Unutar RViz-a, u kartici `Planning` mogu se zadati trajektorije za robota, kojim se provjerava uspješan rad dodatka MoveIt na robotu. Unutar drugog terminala može upravljati robotskom šakom, te je tako ostvareno upravljanje robotskom rukom. Prikaz robota unutar programa RViz i Gazebo nalazi se na slici 5.1.



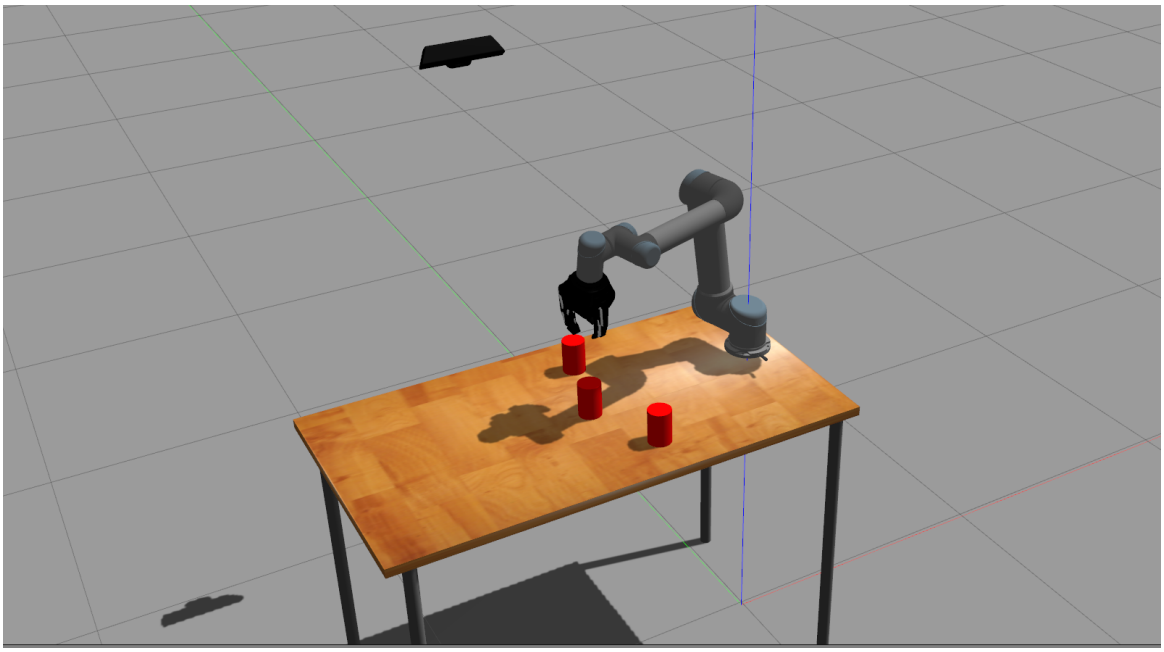
Sl. 5.1: *Prikaz robota u programu RViz i Gazebo*

Unutar *Gazebo* simulacije može se dodati radno okruženje robota, kako bi se dijelom testirao rad nekih algoritama. Primjerice, može se testirati rad algoritma za detekciju objekata u oblaku točaka, ako se u simulaciju dovede neka ploha i nekoliko cilindara. Unutar simulacije teško je predvidjeti ponašanje fizike pa je hvatanje robota otežano bez pisanja nekog koda koji će simulirati hvatanje objekata tako se da objekt spoji s robotom ili slično. Osim toga, detekciju objekata, izračun pozicije objekta preko matrice transformacija i navigaciju robota iznad detektiranog objekta moguće je testirati unutar simulacije.

Objekti se dovode unutar simulacije tako da se opišu unutar *.world* datoteke, koja se dalje uvodi u simulaciju preko *.launch* datoteke. Radno okruženje ove simulacije opisano je u

datoteci *custom.world* te se ona nalazi na primjeru koda 5.5. Svakom svijetu, tj. okruženju robota potrebno je dati ime. Ime se navodi prilikom otvaranja elementa `world` unutar njegovog argumenta `name`. Unutar elementa world, tj. između oznaka `<world>` i `</world>`, dodaju se elementi `include`. Unutar svakog `include` elementa opisuje se jedan model unutar okruženja robota, do kojih se upisuje putanja unutar elementa `uri`. Putanja do modela definira se kao: `model://ime_modela`, gdje je `ime_modela` na primjer, `ground_plane`. Uz putanju do modela, može se definirati ime modela unutar elementa `name` i pozicija modela u simulaciji unutar elementa `pose`. Mogući su i drugi elementi, ali se ne koriste u primjeru koda 5.5. Tako se radno okruženje robota unutar simulacije prikazano na slici 5.2 sastoji od sljedećih modela:

- `ground_plane`: ploha koja predstavlja tlo svijeta. Uglavnom se postavi da svaka točka na ovoj plohi ima vrijednost z koordinate jednaku 0.
- `sun`: objekt koji oponaša sunce te predstavlja izvor svjetlosti.
- `table`: stol nad kojim je robot i objekti koji se hvataju.
- `kinect_custom`: RGBD kamera po uzoru na *Kinect*. Objavljuje podatke na *ROS* teme.
- `cyl_can`: prisutno ih je tri na sceni. Objekti su s kojima se radi i na kojima se vrši detekcija i manipulacija.



Sl. 5.2: *Gazebo simulacija radnog okruženja robota*

Primjer koda 5.5: *custom.world datoteka za opisivanje okruženja robota*

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="custom">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://table</uri>
      <name>Table</name>
      <pose>-0.6 0.0 0.0 0.0 0.0 0.0</pose>
    </include>
    <include>
      <uri>model://kinect_custom</uri>
      <name>kinect_custom</name>
      <pose>-0.6 0.72 1.90 0.0 0.92 -1.570796</pose>
    </include>
    <include>
      <uri>model://cyl_can</uri>
      <name>cyl_can1</name>
      <pose>-0.5 0.23 1.08 0.0 0.0 0.0</pose>
    </include>
    <include>
      <uri>model://cyl_can</uri>
      <name>cyl_can2</name>
      <pose>-0.57 -0.02 1.08 0.0 0.0</pose>
    </include>
    <include>
      <uri>model://cyl_can</uri>
      <name>cyl_can3</name>
      <pose>-0.46 -0.25 1.08 0.0 0.0 0.0</pose>
    </include>
  </world>
</sdf>
```

5.2. Komunikacija sa sklopovljem

Komunikacija s robotskom rukom UR5 i robotskom šakom Robotiq 3f temelji se na uspješnoj konfiguraciji robota za simulaciju u Gazebo. Iako se Gazebo ne koristi kod upravljanja rukom i šakom, bitno je da robot radi uspješno unutar simulacije prije nego se pokuša raditi s pravim sklopovljem.

Dodatno je potrebno preuzeti repozitorij *Universal_Robots_ROS_Driver* koji sadržava upravljačke programe (engl. *drivers*) koji pomažu pri ostvarivanju upravljanja robotom. Dalje, kad se uspješno izgradi radno okruženje potrebno je pozvati sljedeće naredbe u terminalima:

- Svaki prozor Terminala inicijalizirati sa:
  `$ source devel/setup.bash`
- Pokrenuti skriptu za povezivanje s robotskom rukom:
  `$ roslaunch ur_robot_driver ur5_bringup.launch robot_ip:=<robot_ip>`
  `kinematics_config:=<putanja_do_kalibracijske_datoteke>`
- Pokrenuti planer putanje MoveIt:
  `$ roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch`
  `limited:=true`
- Pokrenuti RViz:
  `$ roslaunch ur5_moveit_config moveit_rviz.launch config:=true`
- Upravljanje robotske ruke sad je moguće preko dodatka *Motion Planner* unutar RViz.
- Pokrenuti upravljački program za robotsku šaku:
  `$ rosrun robotiq_3f_gripper_control Robotiq3FGripperTcpNode.py`
  `<ip_adresa_robotske_šake>`
- Pokrenuti čvor za ručno upravljanje robotskom šakom (slika 5.3). Zanemariti ako se robotskom šakom upravlja iz programa:
  `$ rosrun robotiq_3f_gripper_control Robotiq3FGripperSimpleController.py`
- Pokretanje *ROS* čvora za rad s *Intel RealSense* kamerom:
  `roslaunch realsense2_camera rs_camera.launch enable_pointcloud:=true`
  `depth_width:=640 depth_height:=480 depth_fps:=15 color_width:=640`
  `color_height:=480 color_fps:=15 align_depth:=true`

- Pokretanje programa za detekciju arUco markera *aruco_detect*:
  `roslaunch aruco_detect aruco_detect.launch`
- Dalje po potrebi pokretati odgovarajuće skripte.

```
Current command: rACT = 1, rMOD = 0, rGTO = 1, rATR = 0, rPRA = 255, rSPA = 255, rFRA = 150
-----
Available commands

r: Reset
a: Activate
c: Close
o: Open
b: Basic mode
p: Pinch mode
w: Wide mode
s: Scissor mode
(0-255): Go to that position
f: Faster
l: Slower
i: Increase force
d: Decrease force
-->c
```

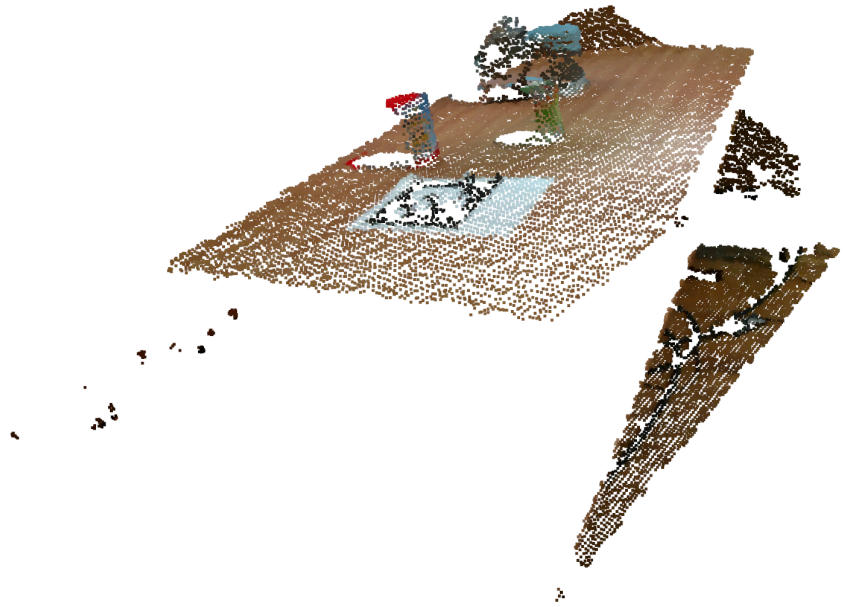
Sl. 5.3: *Prikaz upravljanja robotskom šakom*

5.3. Detekcija objekata u oblaku točaka

Intel RealSense, LIDAR kamera koja se koristi u ovome radu, kao izlaz daje RGBD sliku. RGBD slika sadrži, uz boju, informaciju o dubini svakog piksela. Ako je dostupna intrinzična matrica kamere, može se iz RGBD slike generirati oblak točaka. Oblak točaka generiran pomoću biblioteke Open3D nalazi se na slici 5.4.

Sl. 5.4: *Generirani oblak točaka*

Oblak točaka na slici 5.4 sadrži 831332 točaka, što je dalje poželjno smanjiti koliko je moguće, jer rad s velikim oblacima točaka može znatno usporiti algoritam. Smanjivanje je potrebno odraditi tako da informacija o objektima unutar oblaka točaka ostane velikim dijelom sačuvana. Odabrana metoda je voksel smanjivanje s veličinom voksela od jednog centimetara, uz eliminaciju statističkih *outliera*. Dobiveni oblak točaka nakon smanjivanja sadrži 26224, a nakon statističke eliminacije 22660. Smanjeni oblak točaka nalazi se na slici 5.5.



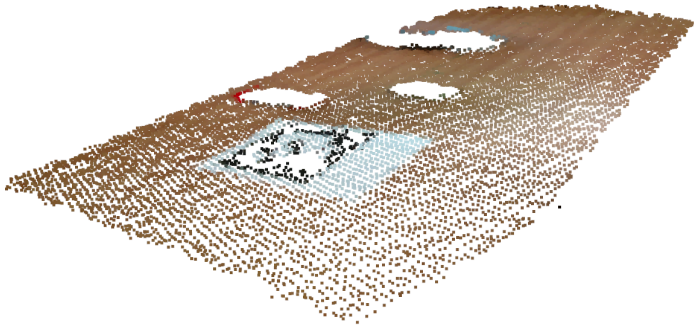
Sl. 5.5: *Oblak točaka nakon smanjivanja*

Sljedeći korak je pronalazak i eliminacija dominantne plohe, tj. plohe stola. Biblioteka Open3D sadrži funkciju koja vraća parametre ravnine i indekse točaka sadržane u dominantnoj ravnini. Dominantna ravnina nalazi se na slici 5.6, a oblak točaka nakon eliminacije dominantne ravnine nalazi se na slici 5.7.

Oblak točaka dominantne ravnine (slika 5.6) koristi se kasnije pri vizualizaciji rješenja detekcije objekta, dok se oblak točaka s eliminiranom dominantnom ravninom (slika 5.7) dalje obrađuje kako bi se detektirali traženi objekti, tj. valjci. Sljedeća operacija nad slikom je eliminiranje točaka koje se nalaze iza ravnine stola, jer ta informacija nije potrebna za lociranje valjaka na površini stola. Oblak točaka nakon eliminacije udaljenih točaka nalazi se na slici 5.8

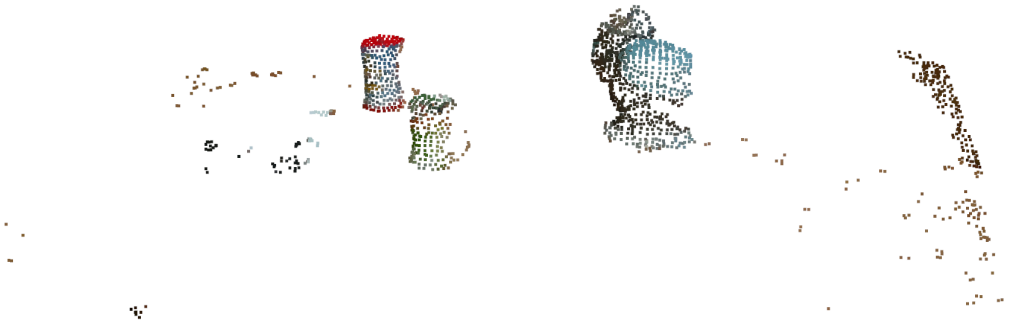
Eliminacijom statističkih *outliera* otklanjaju se točke koje ne nose korisnu informaciju i liče na šum. Operacijom eliminacije statističkih *outliera* otklanjaju se izolirane točke u oblaku točaka, tj. one koje s brojem okolnih susjeda ne zadovoljavaju kriterij koji je postavljen parametrom predanim funkciji. Oblak točaka nakon eliminacije statističkih *outliera* nalazi se na slici 5.9

Oblak točaka sa slike 5.9 jasno prikazuje nakupine točaka. Koristeći DBSCAN implemen-

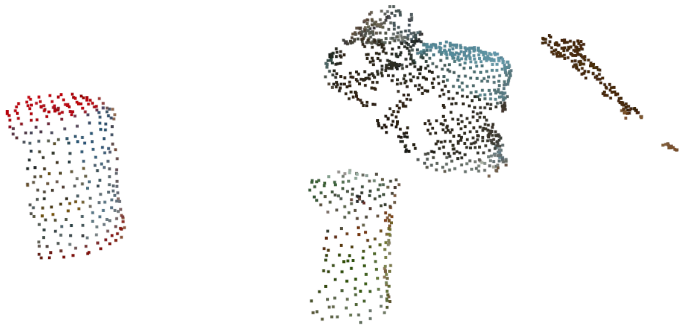


Sl. 5.6: *Dominantna ravnina*

Sl. 5.7: *Oblak točaka nakon eliminacije dominantne ravnine*



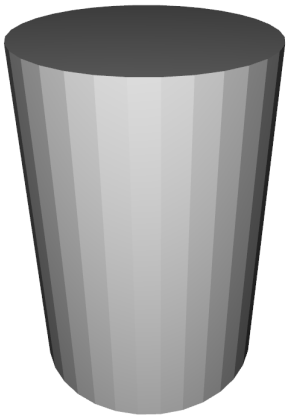
Sl. 5.8: *Oblak točaka nakon eliminacije udaljenih točaka*



Sl. 5.9: *Oblak točaka nakon eliminacije izoliranih točaka*

tiran unutar biblioteke Open3D oblak točaka dijeli se u grupe. Dobivene grupe predstavljaju objekte koje je potrebno dalje testirati i usporediti s modelom traženog objekta. Podijeljeni oblak točaka nalazi se na slici 5.10, a model objekta nalazi se na slici 5.11.

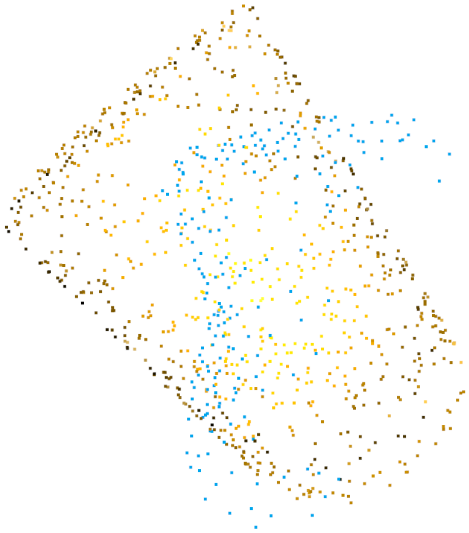
Sl. 5.10: *Oblak točaka podijeljen u grupe*



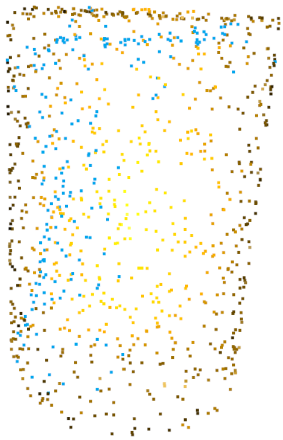
Sl. 5.11: *Model objekta*

Sredina oblaka točaka može se dobiti tako da se izračunaju srednje vrijednosti **x**, **y** i **z** koordinate svih točaka unutar oblaka točaka. Dobivena pozicija koristi se kao početna pozicija kod ICP algoritma. Slika 5.12 prikazuje oblak točaka modela valjka uz oblak točaka objekta koji je mogući kandidat za traženi objekt. Dva oblaka točaka su na približnoj poziciji jedno od

drugoga, ali se trebaju bolje preklopiti kako bi se bolje saznala orijentacija objekta na sceni. ICP algoritam iterativno dolazi do najboljeg rješenja, koje se nalazi na slici 5.13, gdje se vidi da su dva oblaka točaka dobro preklopljeni. ICP algoritam tako daje transformaciju objekta na sceni s obzirom na poziciju koordinatnog sustava kamere.

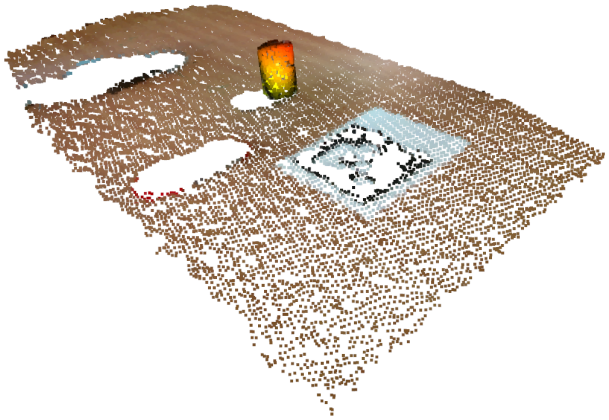


Sl. 5.12: *Oblak točaka modela valjka (narančasto) i oblak točaka objekta na sceni (plavo)*



Sl. 5.13: *Oblak točaka modela valjka (narančasto) i oblak točaka objekta na sceni (plavo) nakon preklapanja ICP-om*

Koristeći dobivenu transformaciju valjka s obzirom na koordinatni sustav kamere, model valjka može se plasirati na lokaciju detektiranog objekta na sceni. Ovaj korak služi samo za vizualizaciju koliko je precizno valjak lociran, te se može vidjeti na slici 5.14.



Sl. 5.14: *Model valjka na lokaciji gdje je valjak detektiran*

Sljedeće su priloženi primjeri koda 5.6, 5.7 i 5.8, koji zajedno čine skriptu *find_cyl_func.py*. Skripta *find_cyl_func.py* je odgovorna za generiranje oblaka točaka, obradu i pronalazak željenih objekata na oblaku točaka. U primjeru koda 5.6, nakon učitavanja potrebnih biblioteka, definira se funkcija `find_cyl` koja prima tri parametra: RGB sliku `color_img`, dubinsku sliku `depth_img` i putanju do modela `model_path`. U varijablu `mesh` učitava se model funkcijom `read_triangle_mesh` iz `o3d.io`, te se dalje jednolikim uzorkovanjem iz modela `mesh` generira oblak točaka `pc_model`. RGB i dubinska slika se dalje instanciraju kao objekti klase `o3d.geometry.Image` koji se dalje spajaju u jednu RGBD sliku koristeći metodu `create_from_color_and_depth` klase `o3d.geometry.RGBDImage`. Za stvaranje oblaka točaka iz RGBD slike potrebno je dodatno definirati intrinzične parametre kamere. Intrinzični parametri definiraju se unutar klase `o3d.camera.PinholeCameraIntrinsic` dodjeljujući vrijednosti njenim atributima `intrinsic_matrix`, `height` i `width`. Oblak točaka se na kraju generira koristeći metodu `create_from_rgbd_image` klase `o3d.geometry.PointCloud` kojoj se predaje RGBD slika `rgbd_img` i parametri kamere `cam_pam`.

Nakon generiranja oblaka točaka slijedi njegova obrada. Primjer koda 5.7 obrađuje oblak točaka kako bi on bio pogodan za segmentaciju i daljnju detekciju koristeći ICP algoritam. Unutar koda smanjenje oblaka točaka i uklanjanje netipičnih vrijednosti odrađuje se dva puta, na početku i na kraju. Smanjenje oblaka točaka odrađuje se metodom `voxel_down_sample` objekta klase `PointCloud`, kojoj se predaje veličina voksela. Uklanjanje netipičnih vrijednosti odrađuje se pozivanjem metode `remove_statistical_outlier` istog objekta. Nad točkama iz oblaka točaka radi se filtriranje po vrijednosti `z` koordinate. Filtriranje se izvodi koristeći

Primjer koda 5.6: *find_cyl_func.py*

```
import copy
import numpy as np
import matplotlib.pyplot as plt
import open3d as o3d


def find_cyl(color_img, depth_img, model_path):
    # LOAD MODEL TO LOOK FOR IN POINTCLOUD
    mesh = o3d.io.read_triangle_mesh(model_path)
    pc_model = mesh.sample_points_uniformly(number_of_points=800)

    # LOAD COLOR AND DEPTH IMAGE, CREATE RGBD IMAGE
    color_raw = o3d.geometry.Image(color_img)
    depth_raw = o3d.geometry.Image(depth_img)
    rgbd_img = o3d.geometry.RGBDImage.create_from_color_and_depth(color=color_raw, depth
    ↪ =depth_raw, convert_rgb_to_intensity=False)

    # DEFINE CAMERA PARAMETERS
    cam_pam = o3d.camera.PinholeCameraIntrinsic()
    cam_pam.intrinsic_matrix = [[597.9033203125, 0.0, 323.8436584472656],
                                [0.0, 598.47998046875, 236.32774353027344],
                                [0.0, 0.0, 1.0]]
    cam_pam.height = 320
    cam_pam.width = 240

    # CREATE POINTCLOUD FROM RGBD AND CAMERA PARAMETERS
    pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_img, cam_pam)
```

funkcionalnosti paketa *numpy*, pa se tako oblak točaka mora prebaciti u oblik *numpy* niza funkcijom `np.asarray`, i kasnije opet vratiti u oblik *Open3D* oblaka točaka instanciranjem točaka kao objekt klase `o3d.utility.Vector3dVector`. Dalje se metodom `segment_plane` objekta klase `PointCloud` pronalazi dominantna ploha unutar oblaka točaka. Metoda vraća parametre ravnine i indekse točke koje pripadaju toj ravnini. Koristeći metodu `select_by_index` iste klase, oblak točaka može se podijeliti na točke koje pripadaju i koje ne pripadaju dominantnoj ravnini. Obrada se dalje nastavlja s točkama koje ne pripadaju dominantnoj ravnini.

Na kraju, primjer koda 5.8 odrađuje segmentaciju oblaka točaka i ICP registraciju na svakom pojedinom segmentu. Metodom `cluster_dbscan` objekta klase `PointCloud` oblak točaka se dijeli na segmente. Podjela na segmente se podešava parametrima `eps` i `min_points`. Dalje se izvodi *for* petlja za `max_label + 1` iteracija, što odgovara broju pronađenih segmenta. Funkcijom `np.where` izdvoje se indeksi svakog pojedinog segmenta, pa se dalje metodom `select_by_index` objekta klase `PointCloud` izdvoje točke s navedenim indeksima. Tako se izdvoji pojedini segment iz oblaka točaka. Dalje se predviđa središte objekta tako da se uzme prosječna vrijednost x, y i z koordinate svake točke koja pripada jednom segmentu. Takvo dobiveno središte objekta često neće biti točno zbog neravnomjerne raspodjele točaka kod oblaka točaka generiranog iz RGBD slike snimljene iz jednog kuta. Bez obzira na to, takvo središte je dovoljno dobro kao početna točka za ICP algoritam. Tako se translacijski dio matrice transfor-

Primjer koda 5.7: *find_cyl_func.py*

```
    # POINTCLOUD DOWNSAMPLE, FIRST REMOVAL OF STATISTICAL OUTLIERS TO REDUCE NOISE
    # DOWNSAMPLING IS PRIMARILY TO REDUCE SIZE
    pcd = pcd.voxel_down_sample(voxel_size=0.005)
    pcd, ind = pcd.remove_statistical_outlier(nb_neighbors=10, std_ratio=0.1)

    # PASSTHROUGH FILTER, REMOVE POINTS WHERE z > some_value
    point = np.asarray(pcd.points)
    colors = np.asarray(pcd.colors)
    filter_z = 0.8
    filt = np.where(point[:,2] < filter_z)
    points_filtered = point[filt[0], :]
    colors_filtered = colors[filt[0], :]
    pc = o3d.geometry.PointCloud()
    pc.points = o3d.utility.Vector3dVector(points_filtered)
    pc.colors = o3d.utility.Vector3dVector(colors_filtered)

    # FIND TABLE PLANE ON POINTCLOUD
    plane_model, inliers = pc.segment_plane(distance_threshold=0.01, ransac_n=3,
    ↪ num_iterations=1000)
    [a, b, c, d] = plane_model
    inlier_cloud = pc.select_by_index(inliers)
    outlier_cloud = pc.select_by_index(inliers, invert=True)

    # SECOND REMOVAL OF STATISTICAL OUTLIERS, TO REDUCE NOISE AND PREPARE FOR CLUSTERING
    outlier_cloud = outlier_cloud.voxel_down_sample(voxel_size=0.01)
    pc, ind = outlier_cloud.remove_statistical_outlier(nb_neighbors=10, std_ratio=0.1)
```

macije `trans_init` inicijalizira s x, y i z koordinatama predviđenog središta objekta `obj_mean`. Parametar `threshold` postavlja se na vrijednost `0.017`. Vrijednost je proizvoljna, a parametar `threshold` se kasnije predaje ICP algoritmu i definira najveću udaljenost u metrima unutar koje se jedna točka može spariti s drugom točkom. Dalje se inicijalizira ICP algoritam funkcijom `o3d.pipelines.registration.registration_icp`, kojoj se predaju parametri:

- Oblak točaka modela objekta `pc_model`.
- Oblak točaka jednog segmenta `pc_obj`.
- Parametar `threshold`, koji definira najveću udaljenost u metrima unutar koje se jedna točka može spariti s drugom točkom.
- Početna matrica transformacije `trans_init`.
- Kriterijska funkcija `TransformationEstimationPointToPoint` iz `o3d.pipelines.registration`.
- Dodavanje kriterija za završetak rada algoritma `ICPConvergenceCriteria` iz `o3d.pipelines.registration`. Ovdje je jedini kriteriji da se dosegne 1000 iteracija.

Po potrebi se registracija evaluira funkcijom `evaluate_registration` iz `o3d.pipelines.registration`. Oblak točaka modela objekta `pc_model` dalje se rotira i tran-

slatira dobivenom matricom transformacija. Dobiveni oblak točaka naziva se `pc_model_transformed` te se dalje uspoređuje s oblakom točaka jednog segmenta `pc_obj`. Za ta dva oblaka točaka izračunava se *Chamfer* udaljenost. Ako je *Chamfer* udaljenost veća od empirijski određene vrijednosti `0.7`, onda se matrica transformacije pronađena ICP-om za taj objekt dodaje u listu `cyl_transforms`. Lista `cyl_transforms` s izlaskom iz *for* petlje sadrži matricu transformacija za svaki pronađeni objekt na sceni koji je sličan danom modelu objekta.

Primjer koda 5.8: *find_cyl_func.py*

```
    # CLUSTERING
    labels = np.array(pc.cluster_dbscan(eps=0.04, min_points=20, print_progress=True))
    max_label = labels.max()

    # ICP FOR EVERY CLUSTER
    cyl_transforms = []
    for i in range(max_label+1):
        cluster = np.where(labels == i)[0]
        cluster_pc = pc.select_by_index(cluster, invert=False)
        pc_obj = copy.deepcopy(cluster_pc)
        obj_points = np.asarray(pc_obj.points)
        obj_mean = np.mean(obj_points, axis=0)
        threshold = 0.017 # parameter
        trans_init = np.asarray([[1, 0, 0, obj_mean[0]],
                                 [0, 1, 0, obj_mean[1]],
                                 [0, 0, 1, obj_mean[2]],
                                 [0.0, 0.0, 0.0, 1.0]])

        reg_p2p = o3d.pipelines.registration.registration_icp(
            pc_model, pc_obj, threshold, trans_init,
            o3d.pipelines.registration.TransformationEstimationPointToPoint(),
            o3d.pipelines.registration.ICPConvergenceCriteria(max_iteration=1000))
        evaluation = o3d.pipelines.registration.evaluate_registration(pc_model, pc_obj,
    ↪ threshold, reg_p2p.transformation)
        pc_model_transformed = copy.deepcopy(pc_model)
        pc_model_transformed.transform(reg_p2p.transformation)

        # CHAMFER DISTANCE
        dist_a = pc_obj.compute_point_cloud_distance(pc_model_transformed)
        dist_a = np.asarray(dist_a)
        dist_a = np.square(dist_a)
        dist_a = np.sum(dist_a)
        dist_b = pc_model_transformed.compute_point_cloud_distance(pc_obj)
        dist_b = np.asarray(dist_b)
        dist_b = np.square(dist_b)
        dist_b = np.sum(dist_b)
        chamfer_dist = dist_a + dist_b
        if chamfer_dist < 0.7:
            cyl_transforms.append(reg_p2p.transformation)

    return cyl_transforms
```

5.4. Komunikacija između ROS i Open3D

Glavni dio programa koji je unutar ROS radnog okruženja napisan je za programski jezik Python inačice 2.7. Python 2.7 je preferirana inačica Pythona za korišteni ROS-melodic. S druge strane, dio programa za detekciju objekata u oblaku točaka koristi i zahtjeva inačicu Pythona 3.6. Komunikacija između različitih inačica Pythona je moguća i ostvariva na više

načina, ali ovdje su odabrane metode komunikacija preko ROS tema i preko lokalnih datoteka. Na Python 3.6 mogu se instalirati paketi `python3-catkin-pkg-modules` i `python3-roskpg-modules` koji omogućuju pristup nekim funkcionalnostima ROS-a, kao što je pristup biblioteci `rospy` i pristup bibliotekama ROS poruka `sensor_msgs` i `std_msgs`.

Iz glavnog programa koji je na inačici Pythona 2.7 poziva se funkcija `start_pc_capture_process()` na liniji 43 unutar primjera koda 5.9 koja priprema i šalje ROS poruku `msg` tipa *Bool* na ROS temu pod nazivom `'/custom_topic/pointcloud/start_ping'` preko metode objavitelja `ping.publish()`. Sadržaj poruke nije bitan, već se poruka koristi kao startni signal programu za detekciju objekta. Skripta *tf_cyl.py* se pretplaćuje na temu `'/custom_topic/pointcloud/end_ping'`, od koje očekuje poruku tipa `Bool`, koja također služi kao signal. Signal dolazi nakon što završi dio koda za generiranje i obradu oblaka točaka. Taj dio koda sprema podatke u datoteku *cyls.txt* te se njen sadržaj čita u povratnoj funkciji (engl. *Callback function*) `status_callback`. Datoteka *cyls.txt* sadrži matrice transformacija detektiranih objekata u odnosu na koordinatni sustav kamere. Matrica transformacije u datoteku je zapisana kao vektor sa 16 elemenata te ju je potrebno prebaciti u oblik matrice s 4 stupca i 4 retka. Iz matrice transformacije moguće je dobiti translacijski vektor i kvaternion rotacije. Translacijski vektor nalazi se u zadnjem stupcu matrice transformacija, a kvaternion se dobiva tako da se funkciji `quaternion_from_matrix()` preda matrica transformacije.

Sljedeće, potrebno je inicijalizirati poruku `tf` tipa `geometry_msgs.msg.TransformStamped` kojoj se dodaje trenutno vrijeme, roditeljski okvir *camera_optic_frame*, koji odgovara okviru kamere, i dijete-okvir, koji odgovara jednom od detektiranih objekta. Poruci `tf` sljedeće se dodaju podatci o translaciji i rotaciji, te se ona doda u listu transformacija objekata na sceni `tf_list`. Transformacije se objavljuju u stablo okvira koristeći klasu `TransformBroadcaster` iz biblioteke *tf2_ros* i njenu metodu `sendTransform()`.

Dio koda koji prima signal i pokreće postupak generiranja oblaka točaka prikazan je na primjeru koda 5.10. Skripta na primjeru koda 5.10 preplaćuje se na tri teme. Prva tema, `/camera/aligned_depth_to_color/image_raw`, sadrži podatke o dubinskoj slici s kamere. Druga, `/camera/color/image_raw`, sadrži podatke o RGB slici s kamere. Treća i zadnja tema na koju se skripta preplaćuje je `/custom_topic/pointcloud/start_ping`, na kojoj dolazi signal za početak rada. Podatci s tema dolaze u određene povratne funkcije. Povratna funkcija `image_callback` dodjeljuje vrijednost varijabli `color`. Povratna funkcija `img_callback` dodjeljuje vrijednost varijabli `depth`, dok povratna funkcija `status_callback` dodjeljuje varijabli `begin` vrijednost `True` kad se pozove. Skripta objavljuje signal na temu `/custom_topic/pointcloud/end_ping` nakon što završi s radom. Kada varijabla `begin` poprimi vrijednost `True` ulazi se u glavni dio koda. Prvo funkcijom `frombuffer()` iz biblioteke Numpy i naknadnim pozi-

Primjer koda 5.9: *tf_cyl.py*

```
import numpy as np
import rospy
from std_msgs.msg import Bool
import os
import tf2_ros
import geometry_msgs.msg
from tf.transformations import quaternion_from_matrix


def status_callback(data):
    with open(os.path.join(pointcloud_gen_path, save_file), "r") as f:
        data = f.readlines()
    data = np.loadtxt(data, delimiter=', ')
    tf_list = []
    if data.ndim is 1 and data.size > 0:
        data = np.expand_dims(data, axis=0)

    for i in range(len(data)):
        temp = data[i,:].reshape((4,-1))
        q = quaternion_from_matrix(temp)
        x = temp[0,3]
        y = temp[1,3]
        z = temp[2,3]

        tf = geometry_msgs.msg.TransformStamped()
        tf.header.stamp = rospy.Time.now()
        tf.header.frame_id = "camera_optic_frame"
        tf.child_frame_id = "cyl_{}".format(i)
        tf.transform.translation.x = x
        tf.transform.translation.y = y
        tf.transform.translation.z = z
        tf.transform.rotation.x = q[0]
        tf.transform.rotation.y = q[1]
        tf.transform.rotation.z = q[2]
        tf.transform.rotation.w = q[3]

        tf_list.append(tf)
    br = tf2_ros.TransformBroadcaster()
    br.sendTransform(tf_list)
    pass


def start_pc_capture_process():
    msg = Bool()
    msg.data = True
    ping.publish(msg)
    pass

pointcloud_gen_path = "../pointcloud_gen/"
save_file = "cyls.txt"

ping = rospy.Publisher('/custom_topic/pointcloud/start_ping', Bool, queue_size=10)
rospy.Subscriber('/custom_topic/pointcloud/end_ping', Bool, status_callback)
rospy.sleep(1)
```

vom metode `reshape()` preoblikujemo podatke iz RGB i dubinske slike u odgovarajući format. RGB i dubinska slika, zajedno s putanjom do modela objekta, predaju se funkciji za detekciju objekta. Funkcija za detekciju objekta generira iz RGB i dubinske slike oblak točaka, te na njemu pronalazi objekte koji su slični predanom modelu. Funkcija za detekciju vraća matricu transformacija svakog pronađenog objekta u odnosu na koordinatni sustav kamere. Dobivene matrice transformacija potrebno je pripremiti za spremanje u tekstualnu datoteku `cyls.txt` tako da se iz matrice preoblikuje u vektor. Podatci se u tekstualnu datoteku spremaju koristeći

Primjer koda 5.10: *main_gen.py*

```
import numpy as np
import rospy
from sensor_msgs.msg import Image
from std_msgs.msg import Bool
import numpy as np
import os
from find_cyl_func import find_cyl


def img_callback(data):
    global color
    color = data
    pass

def depth_callback(data):
    global depth
    depth = data
    pass

def status_callback(data):
    global begin
    begin = True
    pass


color = Image()
depth = Image()
begin = False

rospy.init_node('generate_pointcloud')
rospy.Subscriber('/camera/aligned_depth_to_color/image_raw', Image, depth_callback)
rospy.Subscriber('/camera/color/image_raw', Image, img_callback)
rospy.Subscriber('/custom_topic/pointcloud/start_ping', Bool, status_callback)

ping = rospy.Publisher('/custom_topic/pointcloud/end_ping', Bool, queue_size=10)

rospy.sleep(0.5)
save_file = "cyls.txt"
if os.path.exists(save_file):
    os.remove(save_file)
mesh_path = "data/modeli/cyl/cyl_grah.ply"

while not rospy.is_shutdown():
    if begin:
        cv_color = np.frombuffer(color.data, dtype=np.uint8).reshape(color.height, color
    ↪ .width, -1)
        cv_depth = np.frombuffer(depth.data, dtype=np.uint16).reshape(depth.height,
    ↪ depth.width, 1)

        cyl_tfs = find_cyl(cv_color, cv_depth, mesh_path)

        # flatten transfer matrices for file saving
        tf_msgs = []
        for i in range(len(cyl_tfs)):
            tf_msgs.append(cyl_tfs[i].flatten())

        # We write data to file, another script will read this data
        # this way we transfer data between different python versions
        np.savetxt(save_file, tf_msgs, fmt="%.6f", delimiter=', ')
        begin = False

        # Simple bool messages are used as pings to signal an event
        msg = Bool()
        msg.data = True
        ping.publish(msg)
    rospy.sleep(0.1)
print("end")
```

funkciju `savetext()` iz biblioteke Numpy. Nakon spremanja podataka signalizira se nazad da je skripta gotova s radom i da se podatci o transformacijama pronađenih objekata mogu pročitati iz datoteke.

Signal o završetku rada onda prima povratna funkcija `status_callback` u kodu 5.9, koja onda čita podatke iz datoteke i objavljuje ih na stablo okvira, kao što je objašnjeno.

5.5. Ostali potporni kod

Uz programski kod za detekciju objekta u oblaku točaka, i kod za komunikaciju između različitih dijelova koda, potreban je kod koji će davati naredbe robotskom manipulatoru. Tako se unutar skripte *move_robot.py* nalazi programski kod 5.13, koji je odgovoran za postavljanje robotske ruke u položaj u kojemu se s kamerom dobro vidi scena, kako bi se mogla dobiti RGB i dubinska slika scene za daljnje generiranje oblaka točaka. Također, unutar skripte nalazi se programski kod 5.14, koji je odgovoran za postavljanje vrha robotske ruke o određen položaj. Tu je i kod za provjeru pozicije robota, otvaranje robotske šake i zatvaranje robotske šake. Programski kod za zatvaranje robotske šake nalazi se na primjeru koda 5.15.

Unutar primjera koda 5.11 nalaze se deklaracije metoda klase `RobotControl`, koja je odgovorna za upravljanje robotom. Za početak tu je metoda za inicijalizaciju objekta klase `__init__(self, group_name)`. Njoj se predaje parametar `group_name`, koji nosi naziv grupe zglobova kojim će se upravljati. Metoda `__init__` inicijalizira sve potrebne varijable i objekte potrebne za rad klase, te se ona nalazi unutar primjera koda 5.12. Prvo se u varijablu `self.robot` sprema instanca klase `RobotCommander` iz paketa *moveit_commander*. Klasa `RobotCommander` inicijalizira sučelje prema robotu. U varijablu `self.scene` sprema se instanca klase `PlanningSceneInterface` iz istog paketa te ona predstavlja sučelje prema okolini robota. Unutar varijable `self.move_group` sprema se instanca klase `MoveGroupCommander` kojoj se predaje spomenuti parametar `group_name`. Klasa `MoveGroupCommander` tako predstavlja sučelje prema grupi zglobova koji se pokreću te se putem nje može dohvatiti i postaviti vrijednosti zglobova grupe. Sljedeće se inicijalizira objavitelj na temu `Robotiq3FGripperRobotOutput`. S te teme robotska šaka sluša naredbe te ih izvodi. Dalje, u varijablu `self.ik` sprema se objekt klase `IK` iz paketa *trac_ik_python.trac_ik* [38]. Klasa `IK` rješava problem inverzne kinematike te se kao parametar njoj predaju imena početnog i završnog članka u lancu. Na kraju, u varijablu `self.cam_pose` sprema se vrijednost svakog zgloba u radijanima. Te vrijednosti odgovaraju vrijednostima zglobova u poziciji robota za snimanje kamerom.

Sljedeća metoda unutar klase `RobotControl` je metoda `go_to_camera_pose(self)` (primjer koda 5.13). Unutar te metode čitaju se trenutne vrijednosti svih zglobova iz grupe zglo-

Primjer koda 5.11: *move_robot.py pregled strukture*

```
import moveit_commander, geometry_msgs.msg, moveit_msgs.msg, rospy, sys
import roslib
from time import time
from trac_ik_python.trac_ik import IK;
import math
roslib.load_manifest('robotiq_3f_gripper_control')
from robotiq_3f_gripper_articulated_msgs.msg import Robotiq3FGripperRobotOutput


class RobotControl:
    def __init__(self, group_name):
        pass
    def go_to_camera_pose(self):
        pass
    def check_if_at_cam_pose(self, tolerance):
        pass
    def setPosition(self, x, y, z):
        pass
    def openGripper(self):
        pass
    def closeGripper(self):
        pass
    def executeTrajectory(self, plan):
        pass
```

Primjer koda 5.12: *move_robot.py inicijalizacija klase*

```
    def __init__(self, group_name):
        self.robot = moveit_commander.RobotCommander()
        self.scene = moveit_commander.PlanningSceneInterface()
        self.group_name = group_name
        self.move_group = moveit_commander.MoveGroupCommander(self.group_name)
        self.pub = rospy.Publisher('Robotiq3FGripperRobotOutput',
    ↪ Robotiq3FGripperRobotOutput, queue_size=10)
        self.ik = IK('base_link', 'tool0', solve_type="Distance")
        self.cam_pose = [math.radians(276.13), math.radians(-11.92), math.radians(-
    ↪ 126.7), math.radians(-95.4), math.radians(93.8), math.radians(182.7)]
```

bova, te se te vrijednosti zamjenjuju vrijednostima zglobova u poziciji za snimanje kamerom. Pozivanjem metode `go(joint_goal, wait=True)` objekta `self.move_group` robotu se daje naredba da promjeni stanje svojih zglobova. Pozivanjem metode `stop()` nakon što robot dovrši svoje kretnje osigurava se da robot ostane na mjestu. Metodi `check_if_at_cam_pose(self, tolerance)` predajemo kao argument toleranciju, odnosno najveću moguću razliku stvarne i željene vrijednosti zglobova robota. Metoda `check_if_at_cam_pose(self, tolerance)` provjerava za svaki zglob grupe je li njegova vrijednost unutar tolerancije, te ako je vraća `True`, a ako nije vraća `False`.

Dalje, metoda `setPosition(self, x, y, z)` unutar primjera koda 5.14 prima x, y i z koordinate te postavlja zadnji članak robotske ruke na to mjesto. Zadnji članak ruke je zadnji zglob u grupi `self.move_group`. Koordinate x, y i z potrebno je pretvoriti u float vrijednosti. Potrebno je definirati rotaciju zadnjeg članka robotske ruke kvaternionom koji se sastoji od komponenata `[x,y,z,w] = [0,-1,0,0]`, što odgovara rotaciji oko x i y-osi za 180

Primjer koda 5.13: *move_robot.py metoda za dolazak u poziciju za snimanje*

```
    def go_to_camera_pose(self):
        joint_goal = self.move_group.get_current_joint_values()
        joint_goal[0] = self.cam_pose[0]
        joint_goal[1] = self.cam_pose[1]
        joint_goal[2] = self.cam_pose[2]
        joint_goal[3] = self.cam_pose[3]
        joint_goal[4] = self.cam_pose[4]
        joint_goal[5] = self.cam_pose[5]
        self.move_group.go(joint_goal, wait=True)
        self.move_group.stop()
```

stupnjeva. Rotacija zadnjeg zgloba robotske ruke definira rotaciju robotske šake, a ujedno i poziciju kamere na robotu. Varijabli `current_joints` dodjeljuje se trenutna vrijednost zglobova grupe `move_group`. Trenutne vrijednosti zglobova `current_joints`, zajedno s vrijednostima željene translacije i rotacije `[x,y,z,qx,qy,qz,qw]` predaju se metodi `get_ik` objekta `self.ik`. Povratna vrijednost metode `get_ik` sprema se u varijablu `goal_joints`. Trenutne vrijednosti zglobova grupe `move_group` onda se zamjenjuju vrijednostima iz varijable `goal_joints`, nakon čega se poziva metoda `go(joint_goal, wait=True)` koja daje naredbu robotu da promijeni svoj položaj.

Primjer koda 5.14: *move_robot.py metoda postavljanje pozicije završnog članka robota*

```
    def setPosition(self, x, y, z):
        x = float(x)
        y = float(y)
        z = float(z)
        #gripper faces down / rotated by z 180
        qx = 0
        qy = -1
        qz = 0
        qw = 0

        curr = self.move_group.get_current_pose()
        current_joints = self.move_group.get_current_joint_values()
        goal_joints = self.ik.get_ik(current_joints, x, y, z, qx, qy, qz, qw)
        if goal_joints is None:
            print("Solution not found")
            return
        print("Inverse kin:")
        print(goal_joints)

        joint_goal = self.move_group.get_current_joint_values()
        joint_goal[0] = goal_joints[0]
        joint_goal[1] = goal_joints[1]
        joint_goal[2] = goal_joints[2]
        joint_goal[3] = goal_joints[3]
        joint_goal[4] = goal_joints[4]
        joint_goal[5] = goal_joints[5]

        self.move_group.go(joint_goal, wait=True)
        self.move_group.stop()
```

Za upravljanje robotskom šakom Robotiq 3-Finger mogu se koristit metode `openGripper` i `closeGripper`. Metode se jedna od druge razlikuju samo u vrijednosti koje se objavljuju na temu, pa je zbog toga unutar rada priložena samo metoda `closeGripper` unutar primjera

koda 5.15. U varijablu `command` inicijalizira se objekt klase `Robotiq3FGripperRobotOutput`. Objekt `command` klase `Robotiq3FGripperRobotOutput` sadrži razne zastavice i postavke kojima se mijenja željeno ponašanje robotske šake. Objašnjenja za svaku zastavicu nalaze se unutar službenog priručnika za Robotiq 3-Finger robotsku šaku [35], a ukratko ona su:

- `rACT` - (engl. *Action Request)*: Zastavica za aktivaciju robotske šake. Moguće vrijednosti su 0 ili 1. Vrijednost 1 govori da se robotska šaka aktivira.
- `rGTO` - (engl. *Go To*): Zastavica za omogućavanje kretnje prstiju robotske šake. Moguće vrijednosti su 0 ili 1. Vrijednost 1 omogućuje kretnju.
- `rSPA` - (engl. *Speed (finger A)*): Podešavanje brzine kretnje prstiju robotske šake. Raspon je od 0 do 255. Vrijednost 255 označuje najveću brzinu kretnje.
- `rFRA` - (engl. *Force (finger A)*): Podešavanje snage prstiju robotske šake. Raspon je od 0 do 255. Vrijednost 150 daje naredbu da se koristi 58.82% snage.
- `rATR` - (engl. *Automatic Release*): Koristi se za odvajanje prstiju od objekta u slučaju neočekivanog prestanka rada robota. Moguće vrijednosti su 0 ili 1. Vrijednost 0 označava normalni rad.
- `rMOD` - (engl. *Gripper Mode*): Služi za podešavanje načina hvatanja robotske šake. Vrijednosti su od 0 do 3. Vrijednost od 1 označuje da se koristi način hvatanja "štipanje".
- `rPRA` - (engl. *Position Request (Finger A)*): Služi za pomicanje prstiju robotske šake. Raspon vrijednosti je od 0 do 255. Vrijednost od 120 znači da šaka nije zatvorena do kraja, ali je dovoljno zatvorena da se korišteni objekt uhvati. Kod metode `openGripper` ova vrijednost se postavlja na 23, što znači da je šaka otvorena, ali ne do kraja. Vrijednosti su proizvoljne.

Nakon što su opcije unutar varijable `command` postavljene na željene vrijednosti, koristi se metoda `publish(command)` objavitelja `self.pub` za objavljivanje naredbi na temu za upravljanje robotskom šakom. Robotskoj šaci daje se 3 sekunde da odradi kretnje. Tijekom te 3 sekunde naredbe se šalju na temu svakih 0.1 sekundi, te se s istekom tri sekunde slanje prekida.

Uz programski kod odgovoran za upravljanje robotskom rukom i robotskom šakom, još je potrebno nekoliko skripti koje imaju zadaću objavljivati prave transformacije na stablo transformacija, kako bi se detektirani objekt mogao uspješno prebaciti iz koordinatnog sustava kamere do koordinatnog sustava robota. Jedna od takvih skripti je *tf_aruco.py*, koja nije priložena u radu zbog sličnosti s primjerom koda 5.9. Razlika je što skripta *tf_aruco.py* ne čita podatke

Primjer koda 5.15: *move_robot.py metoda za zatvaranje robotske šake*

```
    def closeGripper(self):
        command = Robotiq3FGripperRobotOutput()
        command.rACT = 1
        command.rGTO = 1
        command.rSPA = 255
        command.rFRA = 150
        command.rATR = 0
        command.rMOD = 1
        command.rPRA = 120

        # Delay od 3 sekunde da se saka zatvori
        start_time = time()
        while True:
            self.pub.publish(command)
            rospy.sleep(0.1)
            end_time = time()
            if float(end_time - start_time) >= 3.0:
                break
```

iz datoteke, već se pretplaćuje na temu */fiducial_transforms*, odakle uzima podatke o transformaciji arUco markera pronađenih na sceni. Detekcija arUco markera obavlja se koristeći paket *aruco_detect* [39]. Taj se paket pretplaćuje na temu s RGB slikom kamere i na temu na kojoj se objavljuju parametri kamere. Koristeći te podatke locira arUco markere na sceni i njihove transformacije objavljuje na spomenutu temu */fiducial_transforms*. Skripta *tf_aruco.py* onda čita podatke s te teme, te ih objavljuje na stablo okvira kao transformacija između okvira *camera_optic_frame* i *aruco_n*, gdje je n redni broj arUco markera.

Uz skriptu *tf_aruco.py* i *tf_cyl.py* (primjer koda 5.9), još postoji skripta za objavljivanje statičnih transformacija u stablo okvira, naziva *tf_static_camera.py*, koja se nalazi na primjeru koda 5.16. Skripta *tf_static_camera.py* objavljuje transformaciju između koordinatnog sustava kamere i koordinatnog sustava vrha robotske ruke. Ta se transformacija ne mijenja tijekom rada robota jer je kamera fiksirana za zadnji članak robota. Iz matrice transformacija `TCA` dobiva se kvaternion kao povratna vrijednost funkcije `quaternion_from_matrix(TCA)` iz biblioteke *tf.transformations*. Komponente translacijskog vektora `[x,y,z]` dio su transformacijske matrice te je samo potrebno pristupiti odgovarajućim elementima matrice. Nakon toga se generira ROS-poruka tipa `TransformStamped()`, koja će definirati transformaciju između okvira *wrist_3_link* i *camera_link*. Transformacija se objavljuje kao statična transformacija koristeći metodu `sendTransform(tfs)` klase `StaticTransformBroadcaster` iz paketa *tf2_ros*.

Primjer koda 5.16: *tf_static_camera.py*

```
import numpy as np
import rospy
import tf2_ros
import geometry_msgs.msg
from tf.transformations import quaternion_from_matrix


rospy.init_node('tf_static_camera')
tfs = []

TCA = np.array([[-0.99499865, 0.0, -0.09988833, 0.0],[0.0, -1.0, 0.0, 0.19],[-
    ↪ 0.09988833, 0.0, 0.99499865,  -0.015],[ 0.0, 0.0, 0.0, 1.0]])

q = quaternion_from_matrix(TCA)
x = TCA[0,3]
y = TCA[1,3]
z = TCA[2,3]

tf = geometry_msgs.msg.TransformStamped()
tf.header.stamp = rospy.Time.now()
tf.header.frame_id = "wrist_3_link"
tf.child_frame_id = "camera_link"
tf.transform.translation.x = x
tf.transform.translation.y = y
tf.transform.translation.z = z
tf.transform.rotation.x = q[0]
tf.transform.rotation.y = q[1]
tf.transform.rotation.z = q[2]
tf.transform.rotation.w = q[3]

tfs.append(tf)

br = tf2_ros.StaticTransformBroadcaster()
br.sendTransform(tfs)
rospy.spin()
```

5.6. Glavni upravljački kod

Glavni upravljački kod ima ulogu spojiti pojedine dijelove koda u neku cjelinu. Poziva funkcije i metode iz različitih dijelova koda, dobiva potrebne informacije kao što je pozicija objekta u odnosu na robota te poziva funkcije za pokretanje robota kako bi se dovela robotska ruka do prikladne pozicije za hvatanje objekta i spuštanje na željenom mjestu. U sklopu ovog rada razvilo se više različitih glavnih upravljačkih kodova, svaki za provođenje različitog pokusa. Skripta *exp_3.py* glavni je upravljački kod namijenjen za izvršavanje zadnjeg pokusa, gdje se udružuju svi dijelovi koda te se ispituje kompletna funkcionalnost koda. Ostale skripte, *exp_1.py* i *exp_2.py*, ispituju samo nekoliko dijelova koda. Zbog toga se daje uvid u skriptu *exp_3.py*.

Na primjeru koda 5.17, poslije učitavanja potrebnih biblioteka, klasa i funkcija, definira se funkcija `search_frames(regex)`. Funkcija `search_frames` prima parametar `regex` koji predstavlja regularni izraz (engl. *Regular Expression*) za ime okvira iz stabla transformacija. Unutar varijable `frames` sprema se informacija o svim okvirima iz stabla transformacija kao niz znakova (engl. *String*). Dobivena informacija učitava se kao yaml objekt `data` koristeći funkciju `load(frames)` iz paketa *yaml*. Unutar linije koda 14, za svaku stavku unutar *yaml* teksta, tj.

za svaki okvir unutar stabla transformacija provjerava se funkcijom `search(regex, item)` iz paketa *re* zadovoljava li ime tog okvira regularni izraz unutar varijable `regex`. Okviri unutar stabla transformacija koji zadovoljavaju zadani regularni izraz spremaju se u listu `hits` te se vraćaju kao povratna vrijednost funkcije `search_frames`.

Unutar dijela koda koji se izvršava ako se skripta pokreće kao glavna (engl. *Main*) skripta prvo se pokreće *ROS* čvor kako bi se omogućile funkcionalnosti *ROS*-a dalje u kodu. Za dohvaćanje transformacija sa stabla transformacija koristi se paket *tf_ros* i njegove klase `Buffer` i `TransformListener`. Za upravljanje robotom instancira se objekt `RC` klase `RobotControl`. Dodatno se postavljaju potrebne vrijednosti za rad koda: `pose_check_tolerance`, `moved`, `enable_movement` i `grab_z_level`.

Primjer koda 5.17: *exp_3.py - učitavanje biblioteka, definiranje funkcija te inicijalizacija objekata i konstanti*

```
1  import rospy
2  import geometry_msgs.msg
3  import tf2_ros
4  import yaml
5  import re
6  from tf_cyl import start_pc_capture_process
7  from move_robot import RobotControl
8
9  def search_frames(regex):
10     frames = tfBuffer.all_frames_as_yaml()
11     #print(yaml.dump(yaml.load(frames)))
12     data = yaml.load(frames)
13     hits = []
14     for item in data:
15         x = re.search(regex, item)
16         if x is not None:
17             hits.append(x.string)
18     return hits
19
20
21 if __name__ == '__main__':
22     rospy.init_node('robot_manipulation_script')
23     tfBuffer = tf2_ros.Buffer()
24     listener = tf2_ros.TransformListener(tfBuffer)
25     RC = RobotControl("manipulator")
26
27     pose_check_tolerance = 0.01
28     moved = False
29     enable_movement = True
30     grab_z_level = 0.23
31
32     # ...
```

Dalje se u kodu nalazi `while` petlja koja traje sve dok je glavni čvor *ROS*-a aktivan. Struktura petlje prikazana je u primjeru koda 5.18, gdje su veći dijelovi koda izrezani i prikazani samostalno kao primjeri koda 5.19 i 5.20. Prvo se u petlji provjerava je li robot trenutno u poziciji za snimanje kamerom. Provjerava se pozivom metode `RC.check_if_at_cam_pose(pose_check_tolerance)`, te se rezultat tipa *Bool* sprema u varijablu `status`. Parametar `pose_check_tolerance` definira se ranije u kodu te predstavlja toleranciju između trenutne i

očekivane pozicije u metrima. Ako je vrijednost varijable `status` `False`, ulazi se u dio pod *if* naredbom te se poziva metoda `RC.go_to_camera_pose()` unutar linije 4. Metoda `RC.go_to_camera_pose()` daje naredbu robotu da se pomakne u poziciju za snimanje kamerom. Metoda `RC.openGripper()` daje naredbu robotskoj šaci da se otvori, a nakon 4 sekunde čekanja zbog poziva `rospy.sleep(4.0)` opet se provjerava je li robot na poziciji za snimanje kamerom. Ako je vrijednost varijable `status` opet `False`, onda se program vraća nazad na početak petlje zbog naredbe `continue`. Dalje, komentari na linijama 16 do 18 zamjena su za kod za dobivanje transformacija arUco markera sa stabla transformacija. Dobivene transformacije spremaju se u listu `Pose` poruka `aruco_pose_msgs`. Struktura koda identična je primjeru koda 5.19, ali su drugačija imena određenih varijabli i vrijednost regularnog izraza. Nakon što se dobiju transformacije tj. pozicije arUco markera u koordinatnom sustavu robota, funkcija `start_pc_capture_process()` iz skripte *tf_cyl.py* pokreće dio koda za slanje signala za snimanje kamerom, za generiranje oblaka točaka i njegovu obradu. Kad se izvršio dio koda za detekciju i kada se objave transformacije u stablo transformacija, pokreće se dio koda koji dohvaća transformaciju objekta iz koordinatnog sustava kamere u koordinatni sustav robota. Taj dio koda je zamijenjen komentarima u linijama 23 do 25 te se nalazi na primjeru koda 5.19. Nakon izvršenja koda, u varijablu `cyl_pose_msgs` sprema se lista *ROS*-poruka tipa `Pose`. Ako broj elemenata liste arUco markera i broj elemenata liste objekata na sceni nisu isti, program se vraća na početak petlje naredbom `continue`. Ako je broj elemenata isti, onda se nastavlja dalje u dio koda za hvatanje i premještanje objekata, pod uvjetom da je vrijednost varijable `enable_movement` `True`. Komentari na linijama 34 i 35 zamjenjuju primjer koda 5.20 te je taj kod niz naredbi koje robot izvršava kako bi se objekt uhvatio i ostavio na drugoj lokaciji. Ako su se ikakve kretnje izvršile, robot se postavlja nazad u poziciju za snimanje kamerom, te se izlazi iz glavne *while* petlje.

Dio koda koji dohvaća transformacije iz stabla transformacija nalazi se na primjeru koda 5.19. Pozivanjem funkcije `search_frames("cyl_.{1}")` s regularnim izrazom kao argumentom dohvaćaju se okviri iz stabla transformacija s imenom koji zadovoljavaju taj regularni izraz. Povratna vrijednost funkcije sprema se u varijablu `cyl_names`, te se dalje u *for* petlji za svaki dohvaća odgovarajuća transformacija. Svaka pojedina transformacija dohvaća se metodom `tfBuffer.lookup_transform(A,B,C)`, gdje je `A` ime početnog članka, `B` je ime završnog članka, a `C` je vrijeme u kojoj se transformacija traži. Dobivena transformacija dodaje se u listu `cyl_trans`. Ako dohvaćanje transformacije digne iznimku (engl. *Exception*) onda se preskoči na sljedeću iteraciju naredbom `continue`. Svaka transformacija u listi `cyl_trans` u sljedećoj *for* petlji na liniji 13 prebacuje se u oblik *ROS* poruke tipa `Pose` i dodaje u listu `cyl_pose_msgs`.

Na kraju, u primjeru koda 5.20 priložen je dio koda odgovaran za kretnje robota. Za svaki

Primjer koda 5.18: *exp_3.py - struktura glavne petlje*

```
    while not rospy.is_shutdown():
        status = RC.check_if_at_cam_pose(pose_check_tolerance)
        if not status:
            RC.go_to_camera_pose()

        RC.openGripper()
        rospy.sleep(4.0)
        status = RC.check_if_at_cam_pose(pose_check_tolerance)
        if status:
            print("AT CAMERA POSE GOAL")
        else:
            continue

        rospy.sleep(0.5)

        # Dobivanje transformacija aruco markera
        # ...
        # spremanje u aruco_pose_msgs

        start_pc_capture_process()
        rospy.sleep(2.0)

        # Dobivanje transformacija objekta na sceni
        # ...
        # spremanje u cyl_pose_msgs

        if not (len(aruco_pose_msgs) == len(cyl_pose_msgs)):
            print("NUMBER OF ARUCO AND CYL NOT SAME, CHECKING AGAIN")
            continue

        for i in range(len(cyl_pose_msgs)):
            rospy.sleep(3.0)
            if enable_movement:
                # Odradi hvatanje objekta i stavljanje na drugu lokaciju
                # ...
        if moved:
            RC.go_to_camera_pose()
            rospy.sleep(4.0)
            break
        rospy.sleep(2.0)
```

Primjer koda 5.19: *exp_3.py - kod za dobivanje transformacije*

```
        start_pc_capture_process()
        rospy.sleep(3.0)
        cyl_names = search_frames("cyl_.{1}")
        cyl_trans = []
        for i in range(len(cyl_names)):
            try:
                trans = tfBuffer.lookup_transform('base_link', cyl_names[i], rospy.Time
    ↪ ())
                cyl_trans.append(trans)
            except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.
    ↪ ExtrapolationException):
                rospy.sleep(0.1)
                continue
        cyl_pose_msgs = []
        for tran in cyl_trans:
            pose_msg = geometry_msgs.msg.Pose()
            pose_msg.position = tran.transform.translation
            pose_msg.orientation = tran.transform.rotation
            cyl_pose_msgs.append(pose_msg)
```

detektirani objekt na sceni odrađuje se točno definirani redoslijed kretnji:

- Dolazak iznad objekta na određenoj visini `h`, koja odgovara zbroju vrijednosti konstante `grab_z_level`, konstante `0.14`, što je visina za koliko se podigne objekt od površine, i iznosu z koordinate detektiranog objekta. Iznos konstante `grab_z_level` je 0.23. Visina je u metrima.
- Otvaranje robotske šake kao priprema za hvatanje objekta.
- Dolazak na poziciju za hvatanje objekta.
- Hvatanje objekta zatvaranjem robotske šake.
- Podizanje objekta i robota na visinu `h` iznad izvorne pozicije objekta.
- Dolazak objekta i robota na visinu `h` iznad lokacije detektiranog arUco markera.
- Spuštanje objekta i robota na visinu koja je za `0.0025` veća od visine na kojoj se objekt uhvatio, iznad arUco markera. Visina je u metrima.
- Otpuštanje objekta otvaranjem robotske šake. Objekt prilikom otpuštanja iz robotske šake ima pad od 2.5 mm.
- Dolazak robota na visinu `h` iznad pozicije arUco markera.

Ovaj redoslijed se izvodi pozivanjem metoda `RC.setPosition(x,y,z)`, `RC.openGripper()` i `RC.closeGripper()` iz klase `RobotControl`. Metode se pozivaju više puta s različitim argumentima te se tako izvode odgovarajuće kretnje. Na početku svake iteracije provjerava se vrijednost varijable `enable_movement`. Ako je vrijednost `True`, može se nastaviti s kretnjama, ali samo ako se zadovoljava kriterij da je visina detektiranog objekta nužno između 0.05 i 0.12 metara. Ta provjera je sigurnosna i služi za izbjegavanje netočnih vrijednosti, ako je objekt loše detektiran, ili je transformacija loše izračunata.

Primjer koda 5.20: *exp_3.py - kod za pokretanje robota*

```
        for i in range(len(cyl_pose_msgs)):
            rospy.sleep(3.0)
            if enable_movement:
                if cyl_pose_msgs[i].position.z < 0.05 or cyl_pose_msgs[i].position.z >
    ↪ 0.12:
                    break
                RC.setPosition(cyl_pose_msgs[i].position.x, cyl_pose_msgs[i].position.y,
    ↪  grab_z_level + 0.14 + cyl_pose_msgs[i].position.z)
                RC.openGripper()
                RC.setPosition(cyl_pose_msgs[i].position.x, cyl_pose_msgs[i].position.y,
    ↪  grab_z_level + cyl_pose_msgs[i].position.z)
                RC.closeGripper()
                RC.setPosition(cyl_pose_msgs[i].position.x, cyl_pose_msgs[i].position.y,
    ↪  grab_z_level + 0.14 + cyl_pose_msgs[i].position.z)

                RC.setPosition(aruco_pose_msgs[i].position.x, aruco_pose_msgs[i].
    ↪ position.y, grab_z_level + 0.14 + cyl_pose_msgs[i].position.z)
                RC.setPosition(aruco_pose_msgs[i].position.x, aruco_pose_msgs[i].
    ↪ position.y, grab_z_level + 0.0025 + cyl_pose_msgs[i].position.z)
                RC.openGripper()
                RC.setPosition(aruco_pose_msgs[i].position.x, aruco_pose_msgs[i].
    ↪ position.y, grab_z_level + 0.14 + cyl_pose_msgs[i].position.z)
                moved = True
            if moved:
                RC.go_to_camera_pose()
                rospy.sleep(4.0)
                break
            rospy.sleep(2.0)
```

6. EKSPERIMENTALNA EVALUACIJA I REZULTATI

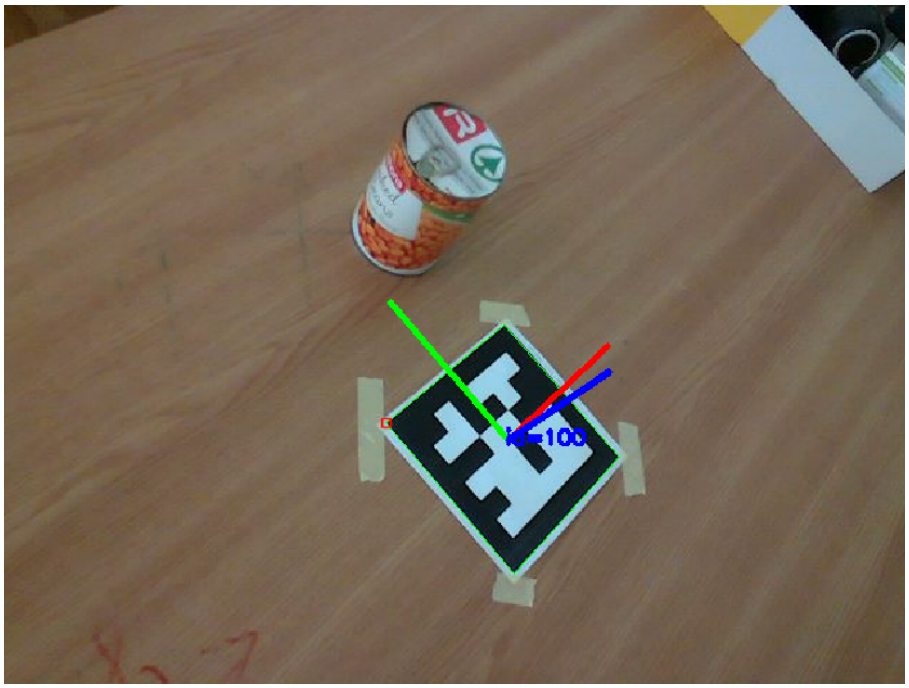
6.1. Evaluacija detekcije objekta u jednoj poziciji

Prvi pokus je namijenjen evaluaciji preciznosti lociranja objekata u oblaku točaka. Cilj je odrediti odstupanje dobivene pozicije od one stvarne. Kod usporedbe pozicije gledat će se samo X i Y osi. Z os nije bitna kod ovog pokusa jer su svi objekti na radnoj površini. Na radnu površinu je postavljen arUco marker, kao što je prikazano na slici 6.1.

U ovome pokusu arUco marker potreban je za dobivanje stvarne pozicije objekta. Stvarnu poziciju moguće je dobiti i mjerenjem udaljenosti od središta baze robota, metrom ili sličnim napravama. Ovisno o mjernom uređaju, ta metoda mjerenja može biti različitog stupnja pouzdanosti. Korištenje arUco markera za dobivanje udaljenosti neke točke radne površine u odnosu na koordinatni sistem baze robota je jednostavnija i možda pouzdanija metoda. Pouzdanost opet ovisi o sposobnosti algoritama da detektira arUco marker. Za detekciju arUco markera korišten je **aruco_detect** ROS paket. Paket **aruco_detect** pokreće se kao ROS čvor koji se pretplaćuje na teme **/camera/color/image_raw/compressed** i **/camera/color/camera_info**. Izlazne vrijednosti, tj. informaciju o detektiranim arUco markerima objavljuje na teme **/fiducial_vertices** i **fiducial_transforms**. Dodatno, na temu **/fiducial_image** objavljuje prizor s kamere na kojem je iscrtan koordinatni sustav arUco markera i njegov obrub, što se

Sl. 6.1: *ArUco marker na radnoj površini*

može vidjeti na slici 6.2.



Sl. 6.2: *Prepoznati arUco marker i njegove koordinatne osi*

Paket **aruco_detect** daje poziciju arUco markera s obzirom na koordinatni sustav kamere. Potrebno je dodati tu transformaciju u stablo transformacija paketa **tf2**, te se pomoću njega dobije pozicija arUco markera s obzirom na bazu robotskog manipulatora. Nakon što se pozicija arUco markera odredi, objekt se stavi točno na središte arUco markera, kako je prikazano na slici 6.3.

Sl. 6.3: *Objekt postavljen nad arUco markerom*

Kada je objekt postavljen točno u središte arUco markera, pokreće se algoritam za detekciju objekata. Pozicija dobivena od algoritma za detekciju se tada može usporediti s pozicijom arUco markera. One bi trebale biti jednake, te ih je zbog toga potrebno usporediti kako bi se dobilo moguće odstupanje i pogreška algoritma za detekciju objekata. Rezultati su prikazani u tablici 6.1.

Kao rezultati unutar tablice 6.1 priloženo je, za arUco i za objekt, broj mjerenja, prosječna vrijednost mjerenja, medijan mjerenja i standardna devijacija mjerenja. Razlike između pozicija arUco markera i objekta dobivaju se iz mjerenja. Dalje se izračuna prosječna vrijednost razlika mjerenja i medijan razlike mjerenja te razlika prosječnih vrijednosti i razlika medijana. Razlika između prosječne vrijednosti razlike mjerenja i razlike prosječnih vrijednosti je u redoslijedu operacija. U prvome slučaju su izračunate razlike svakog mjerenja te je poslije uzet njihov prosjek, a u drugome slučaju su izračunate prosječne vrijednosti pa je izračunata njihova razlika. Kod prvog slučaja je ograničavajući faktor to što je nužno da broj mjerenja bude isti, ili da se dio mjerenja zanemari kako bi broj mjerenja bio isti. Zbog toga je praktičnije uzeti razliku prosječnih vrijednosti i razliku medijana kao mjerodavne veličine.

Može se primijetiti da je iznos standardne devijacije kod mjerenja pozicije arUco markera `(0.001134, 0.00382227)`, dok je iznos standardne devijacije kod mjerenja pozicije objekta `(0.0005608, 0.00064557)`. Iznos standardne devijacije za mjerenje pozicije arUco markera je tako za `2.02` puta veći za x koordinatu, a `5.92` puta veći za y koordinatu. Veći iznos standardne devijacije znači da je varijacija među rješenjima veća. Tako je zaključak da algoritam detekcije

Tab. 6.1: *Mjerenja pozicije arUco markera i objekta*

Mjerenja pozicije arUco markera		
	x	y
Broj mjerenja	26	
Prosječna vrijednost $[m]$	0.05071032	0.59687478
Medijan $[m]$	0.05008759	0.59490615
Standardna devijacija $[m]$	0.001134	0.00382227
Mjerenja pozicije objekta		
	x	y
Broj mjerenja	24	
Prosječna vrijednost $[m]$	0.04796478	0.57369887
Medijan $[m]$	0.04793079	0.57363573
Standardna devijacija $[m]$	0.0005608	0.00064557
Razlike u vrijednostima		
	x	y
Prosječna vrijednost razlike mjerenja $[m]$	0.00269609	0.02310026
Medijan razlika mjerenja $[m]$	0.00232681	0.02173264
Razlika prosječnih vrijednosti $[m]$	**0.00274554**	**0.0231759**
Razlika medijana $[m]$	**0.0021568**	**0.02127042**

daje konzistentnija rješenja kad se usporedi s radom algoritma za detekciju arUco markera.

Dobivena razlika prosječnih vrijednosti iznosi `(0.00274554, 0.0231759)`, što je otprilike jednako odstupanju od 2.7 mm po x osi i 23.1 mm po y osi. S obzirom na to da je veličina objekta koji se detektira 72 mm, ovo odstupanje smatra se značajnim.

6.2. Evaluacija detekcije objekta na više pozicija

Drugi pokus mjeri točnost algoritma za različite pozicije objekta na radnoj površini. Objekt se pomiče za točno određenu udaljenost od ishodišne točke. Algoritam bi idealno trebao pokazati tu udaljenost kao euklidsku udaljenost između ishodišta i pozicije na koju je objekt naknadno pomaknut. Cilj pokusa je zabilježiti koliko se razlikuje stvarni pomak objekta od izračunate euklidske udaljenosti te na temelju rezultata doći do određenih zaključaka.

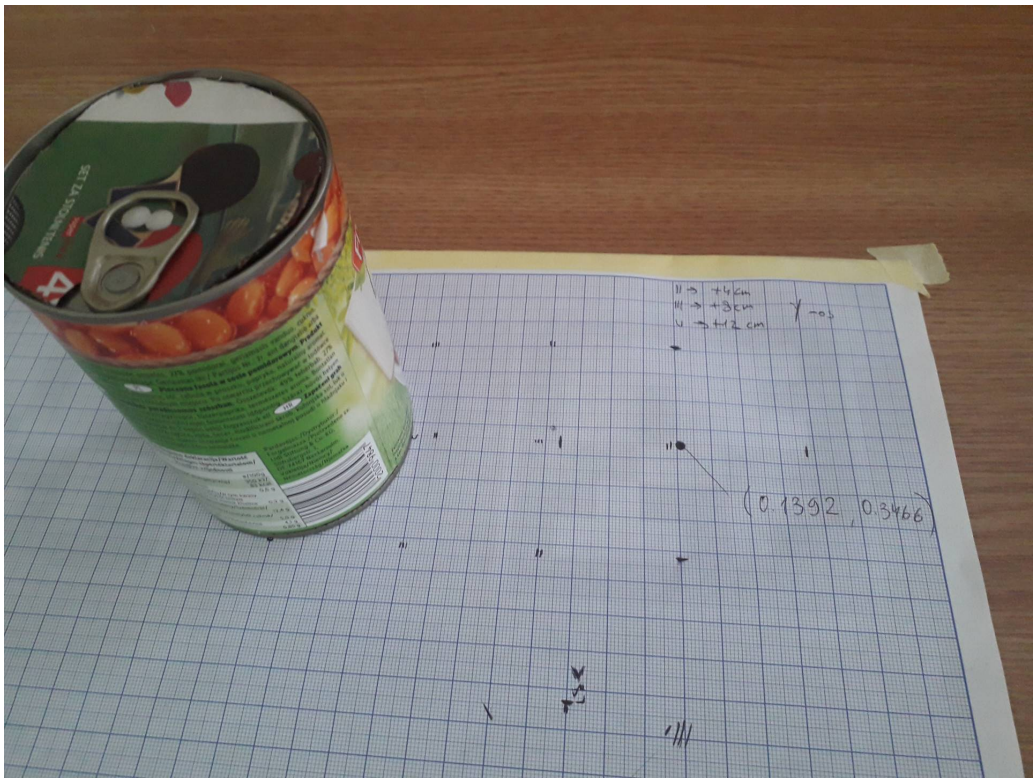
Pokus se tako sastoji od sljedećih koraka:

- Definirati ishodište za pokus. Ovdje je uzeto središte arUco markera, čija je pozicija dobivena preko paketa `aruco_detect` te iznosi `(0.1392, 0.3466)`. Poznavanje stvarne pozicije ishodišta nije nužno za pokus, jer se u ovome pokusu prate razlike između mjerenja. Ovdje je praktički nemoguće objekt postaviti na ishodište bez dodatnih oznaka, jer se postavljanjem objekta na ishodište isto ishodište prekriva. Kao pomoć se označe četiri točke oko objekta, te se svaka od te četiri točke translatira za određenu duljinu kako bi se dobio sljedeći položaj u pokusu. Primjer postavljanja objekta u pokusu može se vidjeti na slici 6.4.
- Objekt se postavi između četiri početne točke. Pokreće se algoritam detekcije i bilježi se više mjerenja pozicije objekta.
- Četiri početne točke translatiraju se za četiri centimetra u jednom smjeru. Objekt se postavlja između nove četiri točke, pokreće se algoritam detekcije i bilježi se više mjerenja pozicije objekta.
- Prethodni korak ponavlja se za udaljenost od osam centimetra od ishodišta.
- Prethodni korak ponavlja se za udaljenost od dvanaest centimetra od ishodišta.
- Provodi se statistička analiza za sva mjerenja.

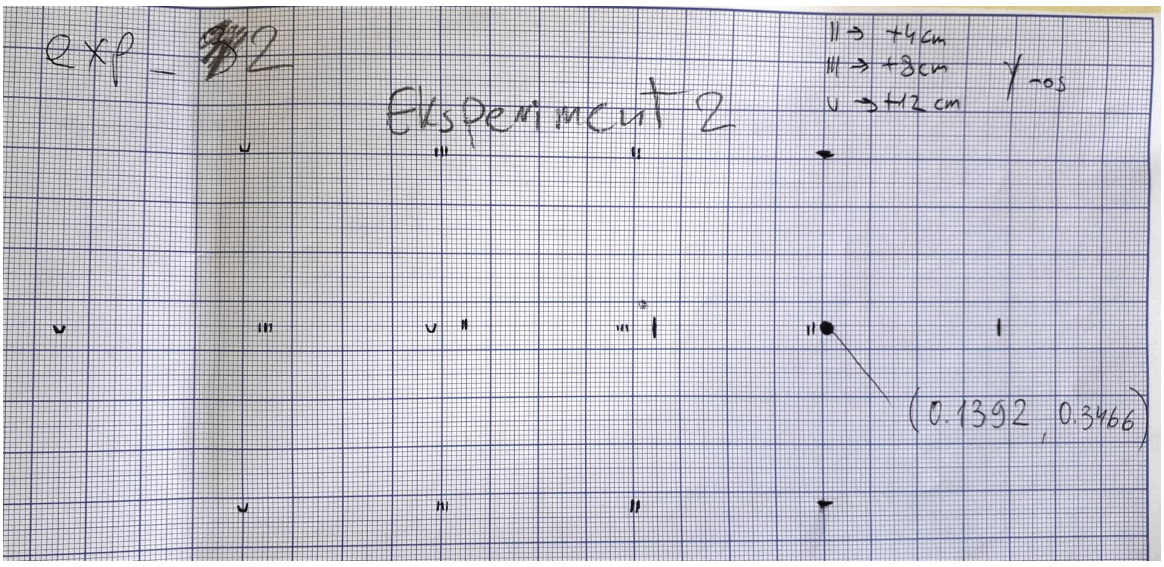
Zbog većeg broja točaka i zbog bilježenja rezultata na papiru, točke na različitim udaljenostima od ishodišta su različito označene, što se vidi na slici 6.5, Oznake su sljedeće:

- Oznaka `-`: Točke ishodišta

- Oznaka **II**: Točke udaljene od ishodišta za 4 cm
- Oznaka **III**: Točke udaljene od ishodišta za 8 cm
- Oznaka **u**: Točke udaljene od ishodišta za 12 cm



Sl. 6.4: *Izgled pokusa. Objekt je postavljen na mjesto koje je od ishodišta udaljeno 12 cm*



Sl. 6.5: *Pregled točaka definiranih radi izvedbe pokusa*

Rezultati zabilježeni tokom pokusa prikazani su u tablicama 6.2, 6.3 i 6.4. Unutar tablice prikazane su prosječne i medijan vrijednosti za svaku točku u pokusu: 0 cm, +4 cm, +8 cm i

+12 cm. Broj mjerenja pozicija za svaku točku je od 26 do 33, što se smatra dovoljnim brojem mjerenja za ovakav pokus. Između dvije pozicije računa se za svaku koordinatu razlika prosječnih vrijednosti te koordinate i ta se razlika označava s **A**, dok se razlika medijana označava s **B**. Tako se za razliku **A** računa euklidska udaljenost **A**, a za razliku **B** računa se euklidska udaljenost **B**. Sljedeće se računa apsolutna i postotna pogreška **A**, što je pogreška između euklidske udaljenosti **A** i stvarne udaljenosti. Tako se računa i pogreška **B**, kao pogreška između euklidske udaljenosti **B** i stvarne udaljenosti.

Tab. 6.2: *Mjerenja pozicije za pomak objekta od 4 cm*

Ishodišna pozicija			
	x	y	z
Broj mjerenja	30		
Prosječna vrijednost $[m]$	0.142439	0.35992	0.066973
Medijan $[m]$	0.141875	0.358887	0.062649
Mjerenja pozicije objekta na 4 cm			
	x	y	z
Broj mjerenja	28		
Prosječna vrijednost $[m]$	0.143671	0.397373	0.063207
Medijan $[m]$	0.143492	0.397104	0.063205
Razlike u vrijednostima			
	x	y	z
Razlika prosječnih vrijednosti (A) $[m]$	0.001232	0.037453	0.003765
Razlika medijana (B) $[m]$	0.001817	0.038217	0.000556

Euklidska udaljenost A $[m]$	0.037662	
Euklidska udaljenost B $[m]$	0.038264	
Stvarna udaljenost $[m]$	0.04	
Pogreška A	0.002337 m	5.84 %
Pogreška B	0.001736 m	4.33 %

Dobiveni rezultati iz tablica 6.2, 6.3 i 6.4 mogu se svesti na vrijednosti pogreška **A** i **B**. Odnos između vrijednosti pogrešaka za različite udaljenosti od ishodišta predočen je u tablici 6.5. Može se primijetiti da iznos pogreške raste s povećanjem stvarne udaljenosti objekta od ishodišta. Iznos postotne pogreške se međutim ili smanjuje ili ostaje približno isti, ali nema dovoljno mjerenja da se donese kvalitetan zaključak iz izračunatih postotnih pogreški.

Tab. 6.3: *Mjerenja pozicije za pomak objekta od 8 cm*

Ishodišna pozicija			
	x	y	z
Broj mjerenja	30		
Prosječna vrijednost $[m]$	0.142439	0.35992	0.066973
Medijan $[m]$	0.141875	0.358887	0.062649
Mjerenja pozicije objekta na 8 cm			
	x	y	z
Broj mjerenja	26		
Prosječna vrijednost $[m]$	0.144097	0.436568	0.066973
Medijan $[m]$	0.144119	0.436387	0.06469
Razlike u vrijednostima			
	x	y	z
Razlika prosječnih vrijednosti (A $[m]$	0.001658	0.076647	0.002282
Razlika medijana (B) $[m]$	0.002243	0.077501	0.00198

Euklidska udaljenost A $[m]$	0.076699	
Euklidska udaljenost B $[m]$	0.077559	
Stvarna udaljenost $[m]$	0.08	
Pogreška A	0.0033 m	4.12 %
Pogreška B	0.002441 m	3.05 %

Tab. 6.4: *Mjerenja pozicije za pomak objekta od 12 cm*

Ishodišna pozicija			
	x	y	z
Broj mjerenja	30		
Prosječna vrijednost [m]	0.142439	0.35992	0.066973
Medijan [m]	0.141875	0.358887	0.062649
Mjerenja pozicije objekta na 12 cm			
	x	y	z
Broj mjerenja	33		
Prosječna vrijednost [m]	0.145996	0.474589	0.065488
Medijan [m]	0.146	0.47449	0.065673
Razlike u vrijednostima			
	x	y	z
Razlika prosječnih vrijednosti (A) [m]	0.003556	0.114669	0.001485
Razlika medijana (B) [m]	0.004125	0.115604	0.003023

Euklidska udaljenost A [m]	0.114734	
Euklidska udaljenost B [m]	0.115716	
Stvarna udaljenost [m]	0.12	
Pogreška A	0.005266 m	4.39 %
Pogreška B	0.004283 m	3.57 %

Tab. 6.5: *Vrijednosti pogrešaka* ***A*** *i* ***B***

Ishodišna pozicija			
Stvarna udaljenost [m]	0.04	0.08	0.12
Pogreška A [m]	0.002337	0.0033	0.005266
Pogreška B [m]	0.001736	0.002441	0.004283
Pogreška A [%]	5.84	4.12	4.39
Pogreška B [%]	4.33	3.05	3.57

6.3. Evaluacija točnosti i ponovljivosti robota

Treći pokus ispituje točnost algoritma detekcije arUco markera i ponovljivost samog robotskog manipulatora. Paket *aruco_detect* i dalje se koristi za detektiranje arUco markera, kako bi se pronašle koordinate dvije pozicije na radnoj površini.

Pokus se sastoji od dvije pozicije čije su koordinate dobivene arUco markerom. Koordinate predstavljaju koordinate objekta unutar koordinatnog sustava baze objekta. Dvije definirane pozicije nazivaju se ishodište i odredište, ili polazište i destinacija. Koordinate ishodišta i odredišta mjere se jednome, na početku pokusa. Nakon mjerenja koordinata ishodišta i odredišta, preko njih je postavljen milimetarski papir. Na milimetarski papir označena su središta ishodišta i odredišta, kao i pomočne crtice između koji se objekt mora postaviti. Tijekom izvedbe pokusa ta koordinate pozicija ostaju nepromijenjena, kako bi se dobila kvalitetnija informacija o ponovljivosti robotske ruke.

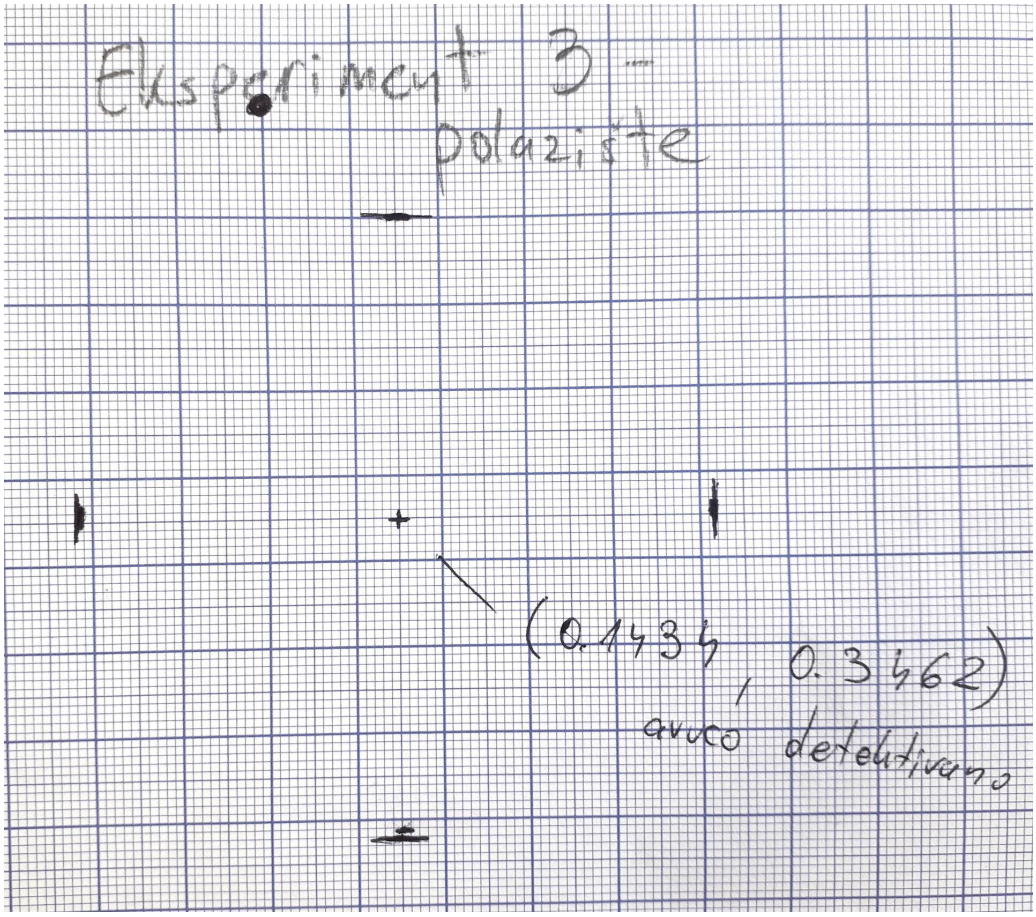
Objekt se stavlja na polazišnu točku koja je prikazana na slici 6.6. Objekt se ručno postavlja između četiri crte koje označuju gdje rub objekta mora biti. Činjenica da se objekt ručno postavlja na polazište dodatni je izvor pogreške te tako nedostatak pokusa. Kada se objekt postavi na predviđeno mjesto na polazištu, pokreće se program za hvatanje objekta te robot prenosi objekt na odredište vidljivo na slici 6.7. Koordinate polazišta u ovome pokusu su `(0.1434, 0.3462)`, a koordinate odredišta su `(0.1537, 0.5499)`.

Rezultati ovog pokusa nalaze se na slici 6.7 u obliku grupacije križića unutar crvene elipse, dok je zelenom točkom na slici označeno središte odredišta.

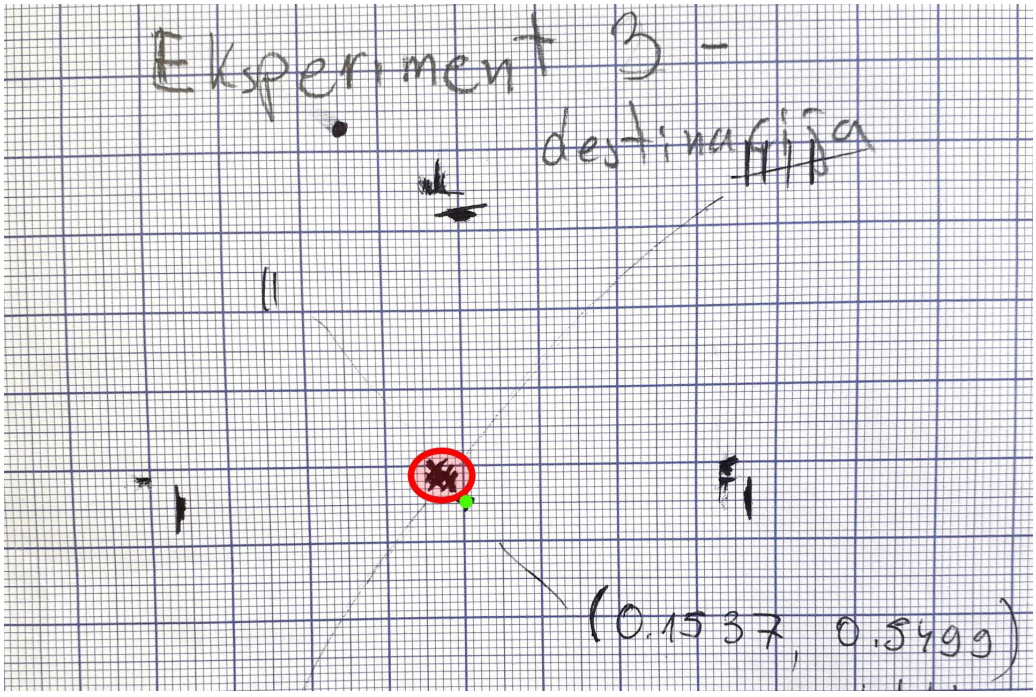
Grupacije unutar ovog rada će se opisati pomoću sljedećih svojstava:

- Broj mjerenja: Broj mjerenja, tj. točaka unutar grupacije.
- Širina grupacije $[mm]$: Horizontalna duljina između krajnjih točaka grupacije.
- Visina grupacije $[mm]$: Vertikalna duljina između krajnjih točaka grupacije.
- Površina grupacije $[mm^2]$: Površina pravokutnika definiranog širinom i visinom grupacije
- Gustoća grupacije $[mm^{-2}]$: Omjer broja točaka i površine grupacije.
- Udaljenost predviđenog središta grupacije od središnje točke $[mm]$.

Svojstva dobivene grupacije u ovom pokusu prikazana su u tablici 6.6.



Sl. 6.6: *Ishodišna točka na poziciji* `(0.1434, 0.3462)`



Sl. 6.7: *Odredišna točka na poziciji* `(0.1537, 0.5499)` *i grupacija rezultata. Crvena elipsa označuje grupaciju točaka, dok zelena točka označuje središte odredišta.*

Tab. 6.6: *Izmjerena svojstva grupacije za treći pokus*

Broj mjerenja	10
Širina grupacije $[mm]$	3
Visina grupacije $[mm]$	2.5
Površina grupacije $[mm^2]$	7.5
Gustoća grupacije $[mm^{-2}]$	1.3333
Udaljenost od središta $[mm]$	4.5

U ovome pokusu širina i visina grupacije vrlo su male što znači da su sve točke blizu međusobno. S time je površina grupacije manja, a gustoća grupacije veća. Mjerenje također pokazuje da je udaljenost središta grupacije od odredišta veća od širine ili visine same grupacije. Iz ovih rezultata mogu se dovesti sljedeći zaključci:

- Širina i visina, a s time i površina grupacije, male su. Znači da je ponovljivost robotskog manipulatora zadovoljavajuća. Budući da su dostupne specifikacije UR5 u kojima se može pronaći preciznost i ponovljivost ovog robota, robotska ruka vrlo vjerojatno nije uzrok ovih odstupanja. Najvjerojatniji uzrok odstupanja je pogreška prilikom postavljanja objekta na ishodište, jer se postavljanje objekta radi ručno.

- Širina ili visina grupacije manja je od udaljenosti središta grupacije od odredišta. Takav rezultat ukazuje na sistematsku pogrešku koja nije nastala pogreškom robotskog manipulatora, a ni zbog ručnog namještanja objekta na ishodište. Najvjerojatniji uzrok ovoj pogrešci je odstupanje prilikom detektiranja arUco markera. Pokazalo se da algoritam za detekciju arUco markera ima sistematsku pogrešku u usporedbi s algoritmom detekcije objekata unutar prvog pokusa (potpoglavlje 6.1). pa je moguće da je dijelom kriv algoritam za detekciju arUco markera.

- Udaljenost između ishodišta i odredišta iznosi 20.4 cm. Prosječna pogreška u postavljanju objekta s ishodišta na odredište u iznosu od 4.5 mm. Takva pogreška, uz relativno veliku udaljenost između ishodišta i odredišta, govori da algoritam detekcije arUco markera ima visoku točnost.

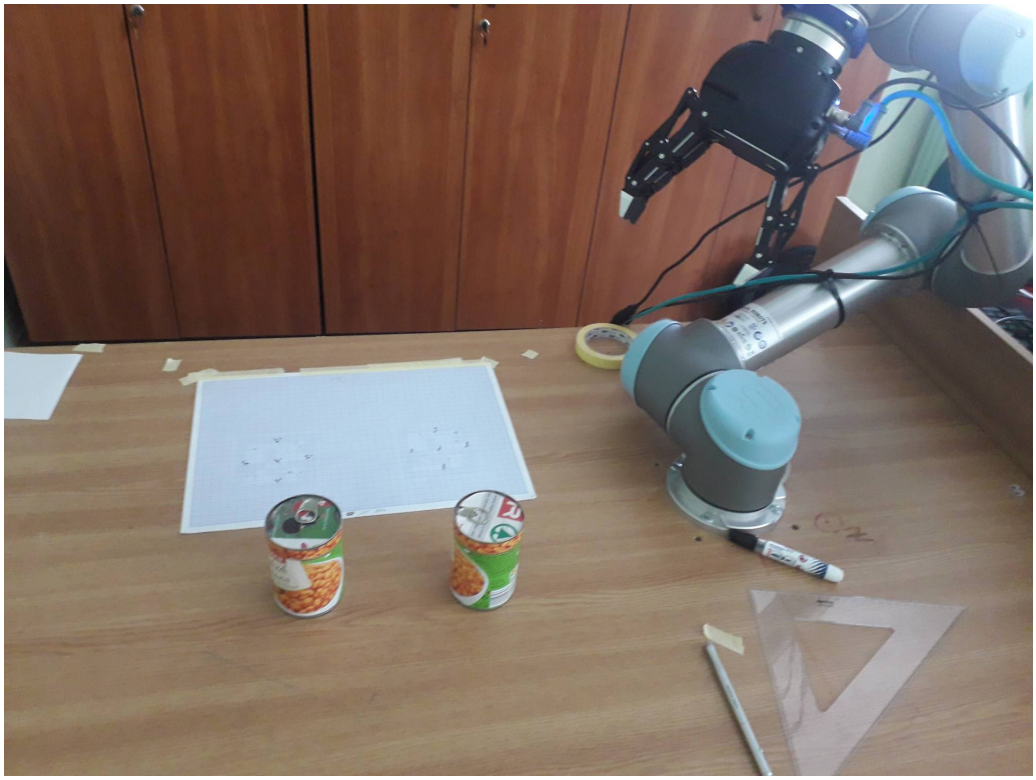
6.4. Evaluacija rada kompletnog algoritma

Četvrti i zadnji pokus evaluira rad kompletnog algoritma napravljenog u sklopu ovog rada. Pokus se sastoji od dva cilindrična objekta za hvatanje, dva arUco markera koji služe kao odredište za objekte te milimetarskog papira za bilježenje pozicije objekata spuštenih nad arUco markere. Izgled radnog okruženja prilikom izvođenja ovog pokusa može se vidjeti na slikama 6.8 i 6.9. Algoritmi korišteni u sklopu ovog pokusa opisani su u radu. Skripta *exp_3.py*, koja se koristi za ovaj pokus, opisana je u potpoglavlju 5.6. Cilj pokusa je zabilježiti središta objekata spuštenih na odredišta te analizirati grupaciju koju te središta tvore.

Tako ovaj pokus uključuje algoritme za generiranje oblaka točaka, algoritme za detekciju objekata u oblaku točaka i za detekciju arUco markera, izračun pozicije objekta koristeći matrice transformacija te upravljanje robotskom rukom i robotskom šakom. Tijek pokusa je sljedeći:

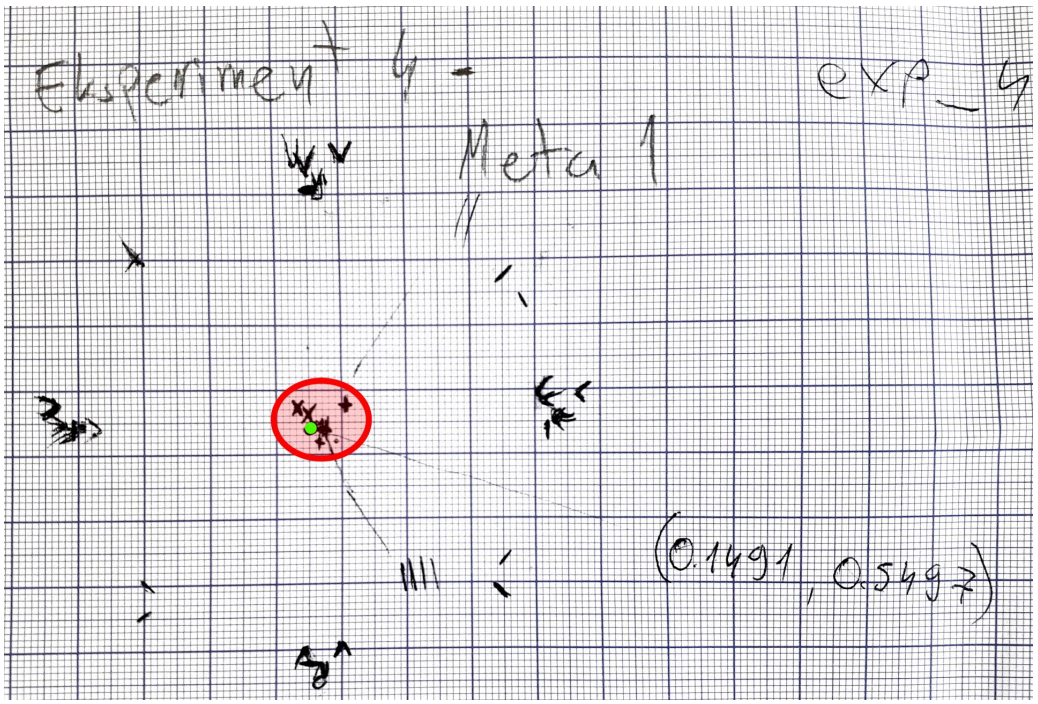
- Pokretanje skripte *exp_3.py* i ostalog potpornog koda. S time se pokreće kompletni algoritam za upravljanje s robotom.
- Robot dolazi na poziciju za snimanje kamerom, ako nije u toj poziciji.
- Podizanje milimetarskog papira kako bi kamera mogla detektirati arUco markere ispod.
- Algoritmi za detekciju arUco markera i za detekciju odabranih objekata vraćaju poziciju objekata unutar koordinatnog sustava kamere. Koristeći objavljene matrice transformacije izračuna se pozicija objekata i arUco markera u koordinatnom sustavu robota.
- Ako je broj detektiranih arUco markera i odabranih objekata isti, pokreće se hvatanje i spuštanje prvog objekta na arUco marker.
- Milimetarski papir je potrebno spustiti preko arUco markera kada se primijeti kretnja robota prema objektu.
- Pokreće se hvatanje i spuštanje drugog objekta na arUco marker.
- Nakon postavljanja objekata na arUco markere bilježi se njihova pozicija nad markerima.
- Postupak se ponavlja deset puta.

Prikaz rezultata pokusa nalazi se na slikama 6.10 i 6.11. Rezultati su u obliku grupacije zabilježenih točaka na kojima je objekt postavljen tijekom pokusa. Grupacija točaka označava se crvenom elipsom, a središte odredišta označava se zelenom točkom. Opis svojstva grupacije dan je u potpoglavlju 6.3 u kojemu je opisan treći pokus. Svojstva grupacija izmjerena su i priložena u tablici 6.7.

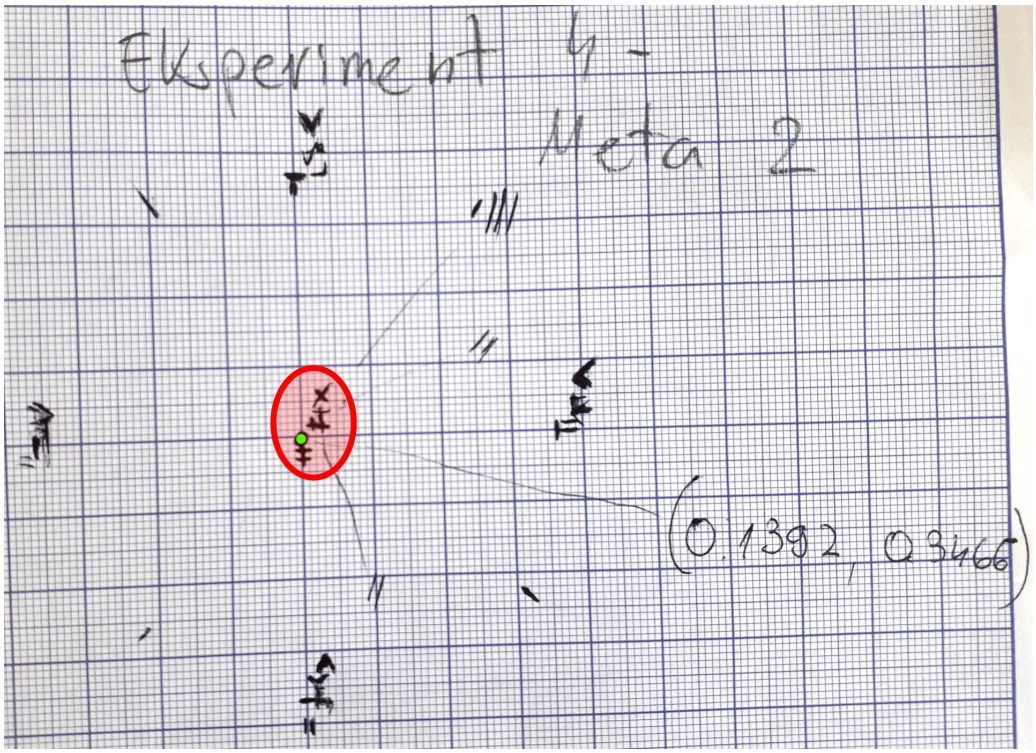


Sl. 6.8: *Izgled radnog okruženja s milimetarskim papirom preko arUco markera*

Sl. 6.9: *Izgled radnog okruženja s podignutim milimetarskim papirom*



Sl. 6.10: *Prva odredišna točka na poziciji* `(0.1491, 0.5497)` *i grupacija rezultata. Crvena elipsa označuje grupaciju točaka, dok zelena točka označuje središte odredišta.*



Sl. 6.11: *Druga odredišna točka na poziciji* `(0.1392, 0.3466)` *i grupacija rezultata. Crvena elipsa označuje grupaciju točaka, dok zelena točka označuje središte odredišta.*

Tab. 6.7: *Izmjerena svojstva grupacije za četvrti pokus*

	Grupacija 1	Grupacija 2
Broj mjerenja	10	10
Širina grupacije $[mm]$	8	3.5
Visina grupacije $[mm]$	7	10
Površina grupacije $[mm^2]$	56	35
Gustoća grupacije $[mm^{-2}]$	0.1786	0.2857
Udaljenost od središta $[mm]$	3	6

Na kraju, rezultati četvrtog pokusa pokazuju kakva grupacija nastaje kada svi algoritmi rade istovremeno. Za razliku od trećeg pokusa u potpoglavlju 6.3 gdje je grupacija vrlo kompaktna, ovdje su grupacije nad oba odredišta raširenije, s manjom gustoćom točaka. Također, gdje je kod grupacije u trećem pokusu sistemska pogreška jasno vidljiva, ovdje je na grupacijama oba odredišta takva sistemska pogreška manje izražena. Grupacije oba odredišta ipak čini se imaju određenu sistemsku pogrešku, ali u smjeru "gore-desno" na priloženim slikama 6.10 i 6.11. Sistemska pogreška u trećem pokusu bila je u smjeru "gore-lijevo", na priloženoj slici 6.7. Sve to ukazuje na sljedeće zaključke:

- Varijacija u poziciji točaka grupacije ovoga pokusa veća je od one u trećem pokusu. Razlog tome je što se u ovome pokusu koristi algoritam detekcije objekta, a algoritam za detekciju arUco markera detektira arUco markere za svako novo mjerenje. U trećem pokusu arUco marker je korišten za dobivanje pozicije točke na radnoj površini, ali je onda ta pozicija ostala konstantna. Korišteni algoritmi za detekciju pokazali su u prvome pokusu u potpoglavlju 6.1 da imaju određenu varijaciju u svojim povratnim vrijednostima.

- U prvom i drugom pokusu pokazala se određena sistemska pogreška u rezultatima. Isto tako, isti uzroci možda su doprinijeli vidljivoj sistemskoj pogrešci u ovome pokusu.

7. ZAKLJUČAK

U ovome radu predstavljen je sustav za robotsku paletizaciju primjenom algoritama računalnog vida. Rad je nastao u suradnji s tvrtkom Protostar Labs d.o.o. Korišteni robotski sustav sastoji se od robotske ruke UR5, od robotske šake Robotiq 3-Finger te od LIDAR kamere Intel RealSense. Programska podrška koja je nužna ovom robotskom sustavu sastoji se od ROS-a i njegovih paketa, programa Gazebo u kojemu se ostvaruje simulacija i od biblioteke Open3D koja se koristi za obradu oblaka točaka i za detekciju objekata u oblaku točaka. Osim korištenja inverzne kinematike i rada s matricama transformacija za potrebe pokretanja robota, algoritam za potrebe detekcije objekata u oblaku točaka koristi algoritme kao što je RANSAC, DBSCAN za segmentiranje oblaka točaka, ICP za registraciju dva oblaka točaka i Chamfer udaljenost za evaluaciju ICP registracije. Zadaća ovog robotskog sustava je detektirati nadolazeće objekte na radnoj površini, izračunati njihovu poziciju, uhvatiti ih i odložiti na za to predviđeno mjesto. Prednost ovakvog sustava robotske paletizacije jest veća tolerancija na lokaciju nadolazećih objekta. S ovakvim pametnijim sustavom objekt može biti bilo gdje u radnom dosegu robota, dok za tradicionalnije sustave robotske paletizacije objekt mora biti na strogo definiranom mjestu. Ovakav pametniji sustav je kompleksniji od tradicionalnijih, što je uglavnom mana u industriji, gdje se teži robusnim rješenjima. Sustavi u industriji moraju zadovoljavati određene kriterije. Loša detekcija, neuspjelo hvatanje ili odlaganje objekta mogu uzrokovati velike gubitke, pa je pouzdanost takvog sustava vrlo bitna. Provedeni pokusi o ovome radu pokazali su da ovaj sustav robotske paletizacije nije adekvatan za uporabu u industriji. Dodatan problem ovog sustava je nedovoljno kvalitetan i sofisticiran stalak za kameru. Kamera se na stalku vrlo lagano pomakne, što onda znači da je sad izračunata matrica transformacija za kameru, koju je samu po sebi teško dobiti, neodgovarajuća. Prvi pokus je ukazao na varijaciju među mjerenjima pozicije arUco markera do nekoliko milimetara. Takva varijacija je nepoželjna, ali se može tolerirati, ovisno o području primjene sustava. Prvi i drugi pokus pokazali su veliku i konzistentnu razliku u poziciji koje ova dva algoritma detekcije daju, s razlikom mjerenja za isto mjesto na radnoj površini koje u prosjeku iznosi 23 milimetra. Usporedbom rezultata trećeg i četvrtog pokusa dolazi se do zaključka da su algoritmi detekcije glavni uzročnici pogreške u radu ovog sustava. Ipak, rezultati četvrtog pokusa pokazali su da je najveća pogreška prilikom postavljanja objekta na odredište 8 milimetara, što upućuje na to da se pogreška kompenzira negdje tijekom rada. Algoritmi detekcije su svakako neadekvatni za uporabu unutar industrije. Kako bi se ovaj sustav robotske paletizacije približio zahtjevima industrijske primjene, potrebno je prvo pronaći alternativu za korišteni algoritam detekcije arUco markera. Dalje, moguće je bolje evaluirati rad algoritma za detekciju objekata u oblaku točaka. Moguće je bolje optimizirati

preklapanje ICP-om, ili postaviti bolje uvjete prije korištenja ICP algoritma. Primjerice, oblak točaka može se generirati koristeći slike iz više kuteva, ili smanjiti šum tako da se uzima više slika tijekom nekog vremena. Nužna nadogradnja za korišteni robotski sustav je kvalitetniji stalak za kameru. Robotska ruka UR5, robotska šaka Robotiq 3-Finger i pravilno kalibrirana LIDAR kamera L515 su se pokazali dovoljno dobrim te nisu znatno pridonijeli pogrešci koja je u ovome radu zabilježena.

LITERATURA

[1] Witold Czajewski and Krzysztof Kołomyjec. A linear approach to matching cuboids in rgbd images. CVPR, 2013.

[2] Shuran Song and Jianxiong Xiao. Sliding shapes for 3d object detection in depth images. 2014.

[3] Witold Czajewski and Krzysztof Kołomyjec. 3d object detection and recognition for robotic grasping based on rgb-d images and global features. *Foundations of Computing and Decision Sciences*, 42, 2017.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: transformers for image recognition at scale. ICLR, June 2021.

[5] Saurabh Gupta, Pablo Arbelaez, Ross Girshick, and Jitendra Malik. Aligning 3d models to rgb-d images of cluttered scenes. 2015.

[6] Shuran Song, Fisher Yu, Andy Zeng, Angel X. Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. CVPR, 2017.

[7] Jintai Chen, Biwen Lei, Quingyu Song, Haochao Ying, Danny Z. Chen, and Jian Wu. A hierarchical graph network for 3d object detection on point clouds. CVPR, 2020.

[8] Danila Rukhovich, Anna Vorontsova, and Anton Konushin. Fcaf3d: Fully convolutional anchor-free 3d object detection. 2020.

[9] Charles R. Qi, Or Litany, Kaiming He, and Leonidas J. Guibas. Deep hough voting for 3d object detection in point clouds. ICCV, 2020.

[10] *Pointcloud outlier removal.* http://www.open3d.org/docs/release/tutorial/geometry/pointcloud_outlier_removal.html. Pristup: 12.7.2022.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI, 1996.

[12] *DBSCAN.* https://en.wikipedia.org/wiki/DBSCAN. Pristup: 6.7.2022.

[13] Chire. *DBSCAN-Illustration*. https://commons.wikimedia.org/wiki/File:DBSCAN-Illustration.svg, Pristup: 6.7.2022.

[14] *Iterative Closest Point*. https://en.wikipedia.org/wiki/Iterative_closest_point. Pristup: 30.06.2022.

[15] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. 2001.

[16] *ICP registration*. http://www.open3d.org/docs/latest/tutorial/Basic/icp_registration.html. Pristup: 30.06.2022.

[17] *Chamfer Distance*. https://pdal.io/apps/chamfer.html. Pristup: 12.7.2022.

[18] *ROS Introduction*. https://wiki.ros.org/ROS/Introduction. Pristup: 2.03.2022.

[19] *Catkin Workspaces*. http://wiki.ros.org/catkin/workspaces. Pristup: 16.03.2022.

[20] *PEP 128*. https://www.ros.org/reps/rep-0128.html. Pristup: 16.03.2022.

[21] *URDF*. http://wiki.ros.org/urdf. Pristup: 17.03.2022.

[22] *URDF XML Specifications*. http://wiki.ros.org/urdf/XML. Pristup: 17.03.2022.

[23] rospy. http://wiki.ros.org/rospy. Pristup: 17.03.2022.

[24] *Gazebo Components*. http://gazebosim.org/tutorials?tut=components&cat=get_started. Pristup: 18.03.2022.

[25] *RViz user guide*. http://wiki.ros.org/rviz/UserGuide. Pristup: 23.03.2022.

[26] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correl. *Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study*. PhD thesis, Sveučilište Colorado u Boulder-u, 2014.

[27] *ROS joint*. http://wiki.ros.org/urdf/XML/joint. Pristup: 24.03.2022.

[28] *ROS Control*. https://www.rosroboticslearning.com/ros-control. Pristup: 24.03.2022.

[29] *tf2*. http://wiki.ros.org/tf2. Pristup: 28.06.2022.

[30] *Writing a tf2 listener (Python)*. http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20listener%20%28Python%29. Pristup: 28.06.2022.

[31] *Writing a tf2 broadcaster (Python).* http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20broadcaster%20%28Python%29. Pristup: 28.06.2022.

[32] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

[33] Universal Robots. *UR5 Technical specifications.* https://www.universal-robots.com/media/50588/ur5_en.pdf, Pristup: 25.6.2022.

[34] *FANUC R-2000.* https://www.fanuc.eu/hr/en/robots/robot-filter-page/r-2000-series/r-2000ic-165f. Pristup: 12.7.2022.

[35] Robotiq. *Robotiq 3-F Adaptive Robot Gripper Instruction Manual.* https://assets.robotiq.com/website-assets/support_documents/document/3-Finger_PDF_20190221.pdf, Pristup: 27.6.2022.

[36] Intel. *Intel RealSense LiDAR Camera L515 Datasheet.* https://www.intelrealsense.com/lidar-camera-l515/, Pristup: 28.6.2022.

[37] *MoveIT.* https://moveit.ros.org/. Pristup: 21.03.2022.

[38] *trac ik.* http://wiki.ros.org/trac_ik. Pristup: 15.7.2022.

[39] *aruco detect.* http://wiki.ros.org/aruco_detect. Pristup: 15.7.2022.

SAŽETAK

U ovome radu predstavljen je sustav za robotsku paletizaciju primjenom algoritama računalnog vida. Rad je nastao u suradnji s tvrtkom Protostar Labs d.o.o. Korišteni robotski sustav sastoji se od robotske ruke UR5, od robotske šake Robotiq 3-Finger, od LIDAR kamere Intel RealSense, od ROS-a, simulatora Gazebo i od biblioteke Open3D za rad s oblacima točaka. Za rad sustava koriste se algoritmi RANSAC, DBSCAN i ICP. Zadaća je ovog robotskog sustava detektirati nadolazeće objekte na radnoj površini, izračunati njihovu poziciju, uhvatiti ih i odložiti na za to predviđeno mjesto. Provedeni pokusi o ovome radu pokazali su da ovaj sustav robotske paletizacije nije adekvatan za uporabu u industriji. Iako je u četvrtom pokusu pokazano da je najveća pogreška sustava 8 mm, prisutna je sistemska pogreška između dva algoritma detekcije u prosjeku od 23 mm. Robotska ruka UR5, robotska šaka Robotiq 3-Finger i pravilno kalibrirana LIDAR kamera L515 su se pokazali dovoljno dobrim te nisu znatno pridonijeli pogrešci koja je u ovome radu zabilježena.

Ključne riječi: ICP, oblak točaka, računalni vid, robotski manipulator, ROS

ABSTRACT

Title: Robotic palletization using computer vision

In this thesis, a system for robot palletization using computer vision algorithms is presented. The thesis was written in cooperation with the company Protostar Labs d.o.o. The robotic system used for this thesis consists of the UR5 robot arm, the Robotiq 3-Finger gripper, the Intel RealSense LiDAR camera, ROS, Gazebo simulator, and the Open3D library used for working with point clouds. Algorithms RANSAC, DBSCAN, and ICP are used in this system. The purpose of this robotic system is to detect incoming objects in the work area, calculate their position, grab them, and release them at the specified location. The experiments conducted in this thesis have shown that this robotic palletization system is inadequate for industrial use. Although the results of the fourth experiment showed that the largest error in object placement was only eight millimeters, there is a consistent difference between the two detection algorithms which averages 23 millimeters. The UR5 robotic arm, the Robotiq 3-Finger gripper, and the properly calibrated L515 LiDAR camera proved to be good enough and haven't contributed in any meaningful way to the error in object placement noted in this thesis.

Keywords: ICP, pointcloud, computer vision, robotic manipulator, ROS

ŽIVOTOPIS

Tomislav Rekić rođen je 01.06.1997. godine u gradu Osijeku u Hrvatskoj. Pohađao je osnovnu školu Fran Krsto Frankopan u Osijeku, nakon čega upisuje Elektrotehničku i prometnu školu Osijek, smjer elektrotehnika. Godine 2016. upisuje preddiplomski studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Nakon završetka prve godine preddiplomskog studija prelazi na smjer računarstvo. Nakon završetka preddiplomskog studija upisuje se na diplomski studij računarstva, smjer Robotika i umjetna inteligencija. U suradnji s tvrtkom Protostar Labs d.o.o. na međunarodnoj konferenciji za pametne sustave i tehnologije SST 2022 ima prihvaćen rad pod nazivom *Visual quality inspection of rotors and stators*.