Programiranje robotske ruke u okolini ROS

Šterpin, Mauro

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

PROGRAMIRANJE ROBOTSKE RUKE U OKRUŽENJU ROS

Završni rad

Mentor: Nikola Tanković, Sven Maričić

Predmet: Programsko inženjerstvo

Student: Mauro Šterpin

Pula, srpanj 2023.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Mauro Štergar, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

Mauro Štergar

U Puli, 28. rujna 2023.

IZJAVA O KORIŠTENJU AUTORSKOG DJELA

Ja, Mauro Štergin dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj Završni rad pod nazivom Programiranje robotske ruke u okolini ROS

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 28. rujna 2023.

Potpis

Mauro Štergin

Sadržaj

1. Uvod

U okviru ovog završnog rada konstruirana je robotska ruka sposobna za prepoznavanje i prihvat pravilnih predmeta putem glasovnih komandi. Primarni cilj rada jest detaljna analizu tehnologija implementiranih u kreiranju ovog sustava, te dublje sagledavanje funkcionalnih aspekata korištenih u ostvarivanju zadatka. Posebna pažnja dana je na integraciju platforme Robot Operating System (ROS), pri čemu će se analizirati nužni koraci za efikasno prepoznavanje boje predmeta, lokalizaciju njihovih pozicija te adekvatno izvršenje manipulativnih akcija.

U uvodu rada će se istražiti osnovni pojmovi i koncepti iz područja robotske tehnologije, te će se pružiti pregled ključnih tehnologija koje su korištene u ovom istraživanju. Također, razmotrit će se važnost integracije ROS-a i kako je ta platforma omogućila uspješno povezivanje komponenata sustava. Kroz analizu ovih tehnologija, bolje ćemo razumjeti načine na koje su pridonijele razvoju inteligentnog sustava za hvatanje kockica na temelju glasovnih uputa.

U nastavku rada bit će detaljno opisana metodologija i koraci koji su poduzeti kako bi se ovaj robotski sustav uspješno implementirao. Bit će istražene specifičnosti algoritama za prepoznavanje boje, kinematička analiza robotske ruke i integracija ROS komponenata. Kroz ovaj proces, bit će jasno kako su tehnologije surađivale kako bi se ostvarila konačna funkcionalnost sustava.

Na kraju, rad će biti zaključen sumiranjem postignuća ovog rada, te ukazivanjem na moguće smjernice za daljnji razvoj i primjenu ovakvih robotskih sustava. Kroz detaljan opis tehnologija i procesa, ovaj rad će pružiti vrijedan uvid u korake potrebne za stvaranje inteligentnog robota sposobnog za hvatanje objekata temeljem glasovnih naredbi.

Programska podrška razvijena za potrebe ovog rada nalazi se na:
https://github.com/maurosterpin/ros-robot-arm

2. Pregled korištenih tehnologija

ROS

ROS, skraćeno od Robotics Operating System, predstavlja softversku platformu koja, unatoč nazivu, nije klasičan operativni sustav. ROS se može opisati kao programski alat koji se instalira na računalima i simulira funkcionalnosti tipične za konvencionalne operativne sustave, no prilagođene su za potrebe robotskih aplikacija. Sličnost između tradicionalnog operativnog sustava na računalu, koji upravlja hardverom i softverom, i ROS-a ogleda se u činjenici da ROS preuzima upravljanje hardverom i softverom robota. Analogija s konvencionalnim operativnim sustavom, kao što je Windows, ilustrira da se softveri poput Google Chrome-a mogu instalirati bez obzira na konfiguraciju računala, bilo da se radi o Intel ili AMD procesoru, ili ima 8, 16 ili 32 gigabajta RAM memorije. Iako će bolji hardver pružiti brže i bolje performanse, osnovne funkcionalnosti softvera ostaju nepromijenjene, eliminirajući potrebu za prilagodbom radi ispravnog funkcioniranja u ROS okruženju. Softverski paketi koji implementiraju određene robotske funkcionalnosti mogu se instalirati i koristiti neovisno o specifičnom hardveru.

Među značajkama ROS-a koje ga definiraju kao operativni sustav, ubrajaju se apstrakcija hardvera, kontrola uređaja na niskoj razini, komunikacija između procesa i upravljanje paketima.

Apstrakcija hardvera

Ako računalo ima Intelov procesor, to ne utječe kako on komunicira s memorijom ili drugim komponentama. Operativni sustav će to učiniti umjesto korisnika. Na isti način, operativni sustav također nudi standardno sučelje za ostale aplikacije koje su instalirane na računalu kako bi koristile njegove resurse.

Ova maskiranje hardverskih razlika ima nekoliko prednosti. Prvo, olakšava korisnicima korištenje računala. Drugo, omogućuje softverskim inženjerima da razvijaju općeniti softver koji će raditi na bilo kojem računalu. Treće, pomaže u osiguravanju kompatibilnosti između različitih hardverskih komponenti.

Operativni sustav igra važnu ulogu u maskiranju hardverskih razlika. To omogućuje korisnicima da koriste računalo bez brige o specifičnostima hardvera na kojem se pokreće. Također omogućuje softverskim inženjerima da razvijaju općeniti softver koji će raditi na bilo kojem računalu.

Kontrola uređaja na niskoj razini

ROS se može smatrati operativnim sustavom za robote. Na isti način na koji operativni sustav omogućuje korisnicima da koriste računalo bez brige o specifičnostima hardvera, ROS omogućuje robotičarima da koriste različite hardverske komponente robota bez brige o specifičnostima njihove komunikacije i logike.

To je omogućeno kroz uporabu upravljačkih programa (eng. drivers). Upravljački program je softver koji omogućuje komunikaciju između operativnog sustava i hardverske komponente. ROS nudi standardno sučelje za upravljačke programe, što znači da robotičari mogu razvijati svoje aplikacije bez brige o specifičnostima hardvera koji se koristi u robotu. Ako robotičar želi koristiti kameru u svom robotu, prvo mora instalirati upravljački program za tu kameru. Upravljački program će pružiti ROS-u standardno sučelje za komunikaciju s kamerom. Robotičar onda može razvijati svoju aplikaciju koja će koristiti kameru, bez brige o specifičnostima njene komunikacije.

Upravljačke programe za ROS obično razvijaju proizvođači hardverskih komponenti za robote. Međutim, postoje i upravljački programi koje razvijaju zajedničke grupe robotičara. ROS-ov sustav upravljanja upravljačkima programima (eng. driver manager) odgovoran je za pokretanje i upravljanje upravljačkim programima. Sustav upravljanja upravljačkim programima također osigurati da svi upravljački programi komuniciraju međusobno na ispravan način. Analogija između ROS-a i operativnog sustava je vrlo korisna za razumijevanje kako ROS funkcionira. Zahvaljujući ROS-u, robotičari se mogu fokusirati na razvijanje svojih aplikacija, bez brige o specifičnostima hardvera koji se koristi u robotu.

Komunikacija između procesa

Jedna od najvažnijih funkcionalnosti koju ROS implementira jest upravljanje komunikacijom između procesa. ROS čvorovi (engl. nodes) mogu komunicirati međusobno koristeći komunikacijski protokol izdavač-pretplatnik (engl. publishersubscriber). Ovaj protokol omogućuje čvorovima da šalju i primaju poruke na određenim kanalima koji se nazivaju teme (engl. topics). ROS master je centralna komponenta koja upravlja komunikacijom između čvorova. Master registrira teme i čvorove, te održava tablicu rutiranja koja osigurava da poruke budu proslijeđene pravim čvorovima.

Čvor koji želi poslati poruku na određenu temu prvo mora objaviti tu temu. Nakon što tema bude objavljena, čvor može poslati poruke na tu temu koristeći funkciju za izdavanje (engl. publish). Čvor koji želi primati poruke s određene teme prvo mora pretplatiti se na tu temu. Nakon što se pretplati, čvor će primati sve poruke koje su poslane na tu temu koristeći funkciju za pretplatu (engl. subscribe).

ROS komunikacijski protokol je asinkroni, što znači da čvor koji šalje poruku ne mora čekati da čvor koji prima poruku odgovori. Ovo omogućuje čvorovima da komuniciraju međusobno na učinkovit način, čak i ako su vrlo različitih performansi. Također je skalabilan, što znači da može podržati veliki broj čvorova koji komuniciraju međusobno. Ovo je postignuto korištenjem ROS mastera za upravljanje komunikacijom između čvorova, te omogućuje čvorovima da komuniciraju međusobno na učinkovit i skalabilan način, bez brige o specifičnostima hardvera na kojem se izvode.

Upravljanje paketima

ROS omogućuje enkapsulaciju robotske funkcionalnosti u pakete. Paketi su kolekcije datoteka koje sadrže čvorove, konfiguracijske datoteke, biblioteke i druge resurse potrebne za implementaciju robotske funkcionalnosti. Paketi imaju zajedničko sučelje i zajedničku strukturu koja je lako ponovno koristiti u bilo kojoj drugoj robotskoj aplikaciji.

Upravljanje paketima u ROS-u omogućuje korisnicima da ponovno koriste robotsku funkcionalnost koju su razvili drugi korisnici, ubrzaju razvoj svojih robotskih aplikacija i fokusiraju se na razvoj novih značajki za svoje robotske aplikacije.

ROS arhitektura

ROS aplikacija je sastavljena od tri glavne komponente: radnog prostora (engl. workspace), paketa (engl. package) i čvora (engl. node). Radni prostor je direktorij koji sadrži sve ROS pakete koje koristi robotska aplikacija. Paket je kolekcija datoteka koje sadrže čvorove, konfiguracijske datoteke, biblioteke i druge resurse potrebne za implementaciju određene robotske funkcionalnosti. Čvor je najmanja komponenta robotske aplikacije i obično je to dio koda koji implementira samo jednu funkcionalnost aplikacije.

ROS struktura aplikacije omogućuje modularan razvoj robotskih aplikacija i omogućuje ponovno korištenje robotskih paketa u različitim robotskim aplikacijama.

ROS aplikacija

Kreiranje ROS aplikacije započinje kreiranjem radnog okruženja. To se radi kroz sljedeće korake:

Komande:

1. mkdir my_robot

2. cd my_robot

3. mkdir src

4. cd src

5. catkin_init_workspace

Sada je radno okruženje inicijalizirano. Za izgradnju radnog okruženja, potrebno je izaći iz foldera src i pokrenuti komandu catkin_make.

Komande:

1. cd ..

2. catkin_make

Nakon toga, ROS radno okruženje će se sastojati od foldera build, devel i src.

Svaki put kada se otvori terminal u svrhu pokretanja skripti ROS paketa, bitno je sourcati setup.bash file: **source devel/setup.bash**

Ako se ne sourca setup.bash ROS neće moći pronaći i pokrenuti ROS pakete.

Za kreirati ROS modul, pozicionira se u src folder te se pozove komandu **catkin_create_pkg** zatim se doda naziv modula/paketa **robot_test** te se doda postojeće ros pakete koji su potrebni.

Primjer:

1. cd ..

2. catkin_create_pkg robot_test roscpp std_msgs

ROS publisher

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>


int main(int argc, char **argv)
{
    // Inicijaliziraj ROS čvor nazvan talker
    ros::init(argc, argv, "simple_publisher_cpp");
    ros::NodeHandle n;

    // registriraj publisher na temi /chatter koji će objavljivati poruke tipa String
    ros::Publisher pub = n.advertise<std_msgs::String>("chatter", 10);

    // Definiraj frekvenciju objavljivanja poruka
    // Stopa je izražena u hercima
    ros::Rate rate(10); // Hz


    int counter = 0;
    // Nastavi objavljivati poruke dok je komunikacija s ROS-om aktivna
    while (ros::ok())
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "Hello world " << counter;
        msg.data = ss.str();
        pub.publish(msg);

        ros::spinOnce();

        // pričekaj željenu stopu prije objavljivanja sljedeće poruke
        rate.sleep();
        ++counter;
    }


  return 0;
}
```

Slika 1. ROS publisher C++

Slika 1. prikazuje kod koji koristi ROS biblioteke ros/ros.h i std_msgs/String.h kako bi omogućio komunikaciju i manipulaciju s porukama. Glavni cilj ovog koda je stvaranje ROS čvora koji će objavljivati poruke na određenoj temi. Kod počinje inicijalizacijom ROS čvora koristeći ros::init funkciju. Naziv ovog čvora postavljen je na `simple_publisher_cpp`. Nakon inicijalizacije čvora, stvara se ros::NodeHandle objekt koji predstavlja rukovatelj za komunikaciju s ROS Masterom. Sljedeći korak je registracija publishera na temu `chatter` koristeći ros::Publisher. Publisher se povezuje s određenom temom i šalje poruke na tu temu. Nakon toga, definira se frekvencija objavljivanja poruka koristeći ros::Rate objekt. Stopa je izražena u hercima. U ovom primjeru, poruke će se objavljivati s frekvencijom od 10 herca, što znači da će se jedna poruka objaviti svake sekunde. U glavnoj funkciji main, ROS čvor se pokreće dok je komunikacija s ROS-om aktivna. U petlji, stvara se nova poruka tipa std_msgs::String. U sadržaj poruke se upisuje poruka "Hello world" zajedno s brojem ponavljanja. Poruka se zatim objavljuje na temu `chatter`. Naposljetku, ROS čvor se čeka određeni

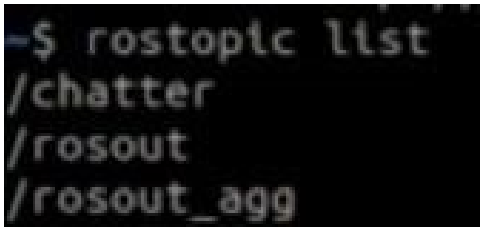
broj sekundi prije objavljivanja sljedeće poruke. Ovaj jednostavan kod prikazuje osnovni koncept objavljivanja poruka unutar ROS okruženja. Pomoću objavljivanja poruka, ROS čvorovi mogu razmjenjivati informacije, omogućavajući tako robotskim sustavima učinkovitu komunikaciju i suradnju.

```
add_executable(simple_cpp_publisher src/simple_publisher.cpp)
target_link_libraries(simple_cpp_publisher ${catkin_LIBRARIES})
```

Slika 2. CMakeLists publisher

CMakeLists datoteka je tekstualna datoteka koja sadrži upute za CMake sustav za gradnju programske aplikacije. U kontekstu ROS-a, CMakeLists datoteka se koristi za upravljanje kompilacijom ROS paketa, uključujući i izgradnju izvršnih datoteka. Kada se doda izvršnu datoteku za ROS paket u CMakeLists datoteku, obavještava se CMake kako da tu skriptu pretvori u izvršnu datoteku. Ovo je važno jer CMake mora znati koji izvorni kod treba kompilirati i kako povezati biblioteke kako bi se stvorila izvršna datoteka. Specifično, kroz sljedeći redak koda: add_executable(simple_cpp_publisher src/simple_publisher.cpp) dodaje se izvršnu datoteku simple_cpp_publisher koja će biti stvorena iz izvorne datoteke simple_publisher.cpp smještene u direktoriju src. Nakon toga, koristi se sljedeći redak koda: target_link_libraries(simple_cpp_publisher ${catkin_LIBRARIES}) da bi se definiralo potrebne biblioteke koje će biti povezane s ovom izvršnom datotekom prilikom kompilacije. Varijabla catkin_LIBRARIES sadrži sve biblioteke koje su potrebne za kompilaciju ROS paketa. Dodavanje izvršne datoteke u CMakeLists osigurava da CMake zna kako kompilirati i povezati ROS skriptu u izvršnu datoteku, što omogućava ROS Masteru da je prepozna i izvrši kao čvor unutar ROS sustava. CMakeLists datoteka je važan dio ROS sustava i omogućuje nam da izgradimo naše ROS pakete i izvršne datoteke na učinkovit način.

Sada se može testirati publisher. U terminalu se prvo pokrene komanda **roscore** koja pokreće ros master koji omogućuje komunikaciju ROS modula. Zatim se pokreće publisher s komandom **rosrun robot_test simple_cpp_publisher**. Zatim u novom terminalu se pokrene komanda **rostopic list** te se prikažu aktivni topics.



Slika 3. ROS topics

ROS subscriber

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"


// Ova funkcija se poziva svaki put kada se objavi nova poruka na temi /chatter
// Poruka koja je objavljena se zatim prosljeđuje kao ulaz u ovu funkciju
void msgCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: %s", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Inicijaliziraj ROS čvor nazvan listener
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;

    // registriraj pretplatnika na temi /chatter koji će slušati poruke tipa String
    // kada se primi nova poruka, pokreće se izvršavanje povratne funkcije
    ros::Subscriber sub = n.subscribe("chatter", 1000, msgCallback);

    // održava čvor aktivnim
    ros::spin();
    return 0;
}
```

Slika 4. ROS Subscriber C++

Ovaj kod koristi ROS biblioteke ros/ros.h i std_msgs/String.h kako bi omogućio komunikaciju i manipulaciju s porukama. Funkcija msgCallback se poziva svaki put kada stigne nova poruka na temu /chatter. Funkcija prima argument msg, koji je tipa std_msgs::String::ConstPtr, što označava da je to konstantni pokazivač na primljenu poruku. U ovom slučaju, funkcija jednostavno ispisuje primljenu poruku koristeći ROS_INFO makro, pri čemu koristi metodu c_str() da bi dobila sadržaj poruke kao običan C-string. U glavnoj funkciji, najprije se inicijalizira ROS čvor sa specifičnim imenom `listener`. Nakon toga, stvara se NodeHandle objekt (n), koji je potreban za komunikaciju unutar ROS sustava. Sljedeća linija koda registrira pretplatnik na temu `chatter` koji će slušati poruke tipa String. Argument 1000 označava veličinu reda poruka koji se koristi za pohranu primljenih poruka prije nego što ih pretplatnik obradi. Treći argument je povratna funkcija msgCallback, koja će se izvršiti svaki put kad stigne nova poruka. Linija koda ros::spin() započinje petlju obrade događaja (event loop) u kojoj čvor aktivno čeka i obrađuje događaje dok ne završi izvođenje programa. Na kraju, funkcija se završava s povratnom vrijednosti 0. Ukratko, ovaj kod prikazuje primjer ROS čvora koji se pretplati na temu /chatter i sluša poruke tipa String koje su objavljene na toj temi. Svaka primljena poruka pokreće izvođenje funkcije msgCallback, koja jednostavno ispisuje sadržaj primljene poruke.

```
add_executable(simple_cpp_publisher src/simple_publisher.cpp)
target_link_libraries(simple_cpp_publisher ${catkin_LIBRARIES})

add_executable(simple_cpp_subscriber src/simple_subscriber.cpp)
target_link_libraries(simple_cpp_subscriber ${catkin_LIBRARIES})
```

Slika 5. CmakeLists Subscriber

Zatim se dodaje subscriber u CmakeList. Sve skripte paketa dodaju se u njihov odgovarajući CmakeLists, stoga od sada na dalje ovaj korak neće biti spomenut već će se podrazumijevati.

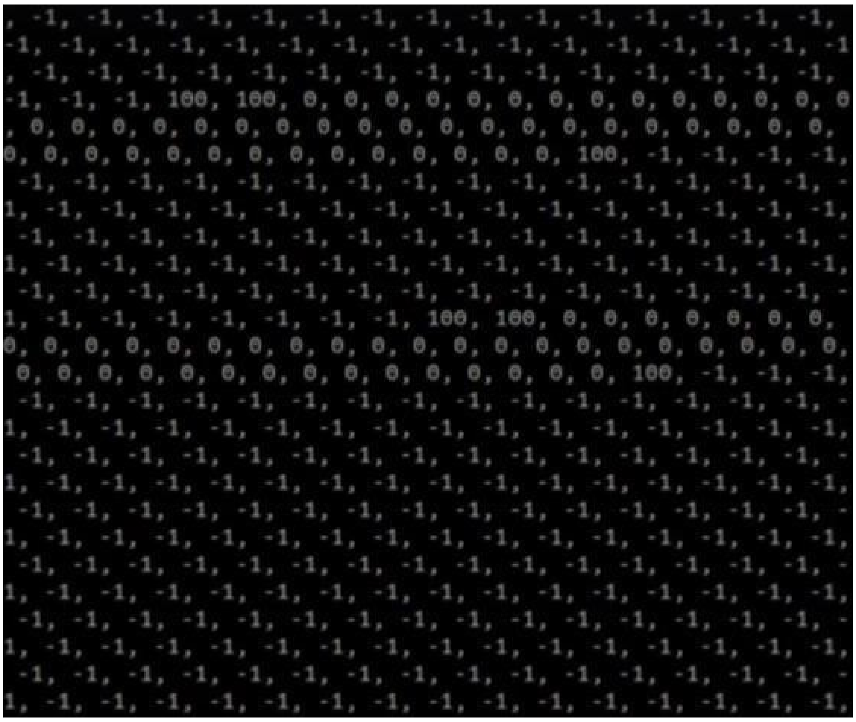
Digitalni dvojnik

Razvoj robota je kompleksan proces koji zahtijeva širok spektar znanja i vještina iz različitih područja, kao što su elektrotehnika, mehatronika, računarstvo i informatika. Jedna od najvećih razlika između razvoja robota i razvoja softwarea za druge platforme, kao što su web ili mobilne aplikacije, je potreba za testiranjem i ispravljanjem koda na pravom robotu. To može biti značajan trošak, kako s gledišta vremena uloženog u izvođenje testova, tako i s gledišta ekonomskih resursa, jer je potreban pristup pravom robotu s pravim hardverom. Da bi se ubrzali proces razvoja i ispravljanja koda, dostupni su alati za simulaciju robota (digital twins). Digitalni dvojnik je virtualni model robota koji omogućava da se testira i vizualizira ponašanje robota u simuliranom okruženju, bez potrebe za korištenjem pravog robota. Digitalni dvojnik može simulirati fizičke sile poput gravitacije, trenje i okretni moment koji se prenosi svakim motorom. Također, može simulirati gotovo sve senzore, poput laserskih skenera, sonara i kamera. Koristi korištenja digitalnih dvojnika u razvoju robota su brojne. Digitalni dvojnik omogućuje rano otkrivanje i ispravljanje pogrešaka u kodu. Testiranje koda u različitim scenarijima, bez potrebe za korištenjem pravog robota. Vizualizaciju ponašanja robota, što može biti korisno za razumijevanje i optimizaciju njegove kinematike i dinamike. Obuku operatora robota, bez potrebe za korištenjem pravog robota. Razvoj softvera za robote sastoji se od tri dijela. Razvoj i izgradnja koda na vašem računalu kako biste riješili rane probleme pri kompilaciji, odnosno te probleme u sintaksi programskog jezika. Testiranje izvršivog koda generiranog u prethodnoj fazi sa simuliranim robotom u simuliranom okruženju na vašem računalu. Testiranje i izvođenje vašeg koda na pravom robotu s pravim hardverom. Iako su simulatori dobar alat za rano ispravljanje koda, oni su samo pojednostavljivanje stvarnog svijeta i vrlo često ono što radi u simulaciji ne radi u stvarnom svijetu ili ne daje iste rezultate. To može biti uzrokovano nekoliko razloga, kao što su bukom u očitanjima senzora ili bukom u kretanju robota koja utječe na stvarni svijet. Razvoj robota je kompleksan proces koji zahtijeva testiranje i

ispravljanje koda na pravom robotu. Digitalni dvojnik je alat koji omogućava da se testira i vizualizira ponašanje robota u simuliranom okruženju, bez potrebe za korištenjem pravog robota. Korištenje digitalnih dvojnika u razvoju robota može značajno ubrzati proces razvoja i ispravljanja koda, te smanjiti troškove razvoja.

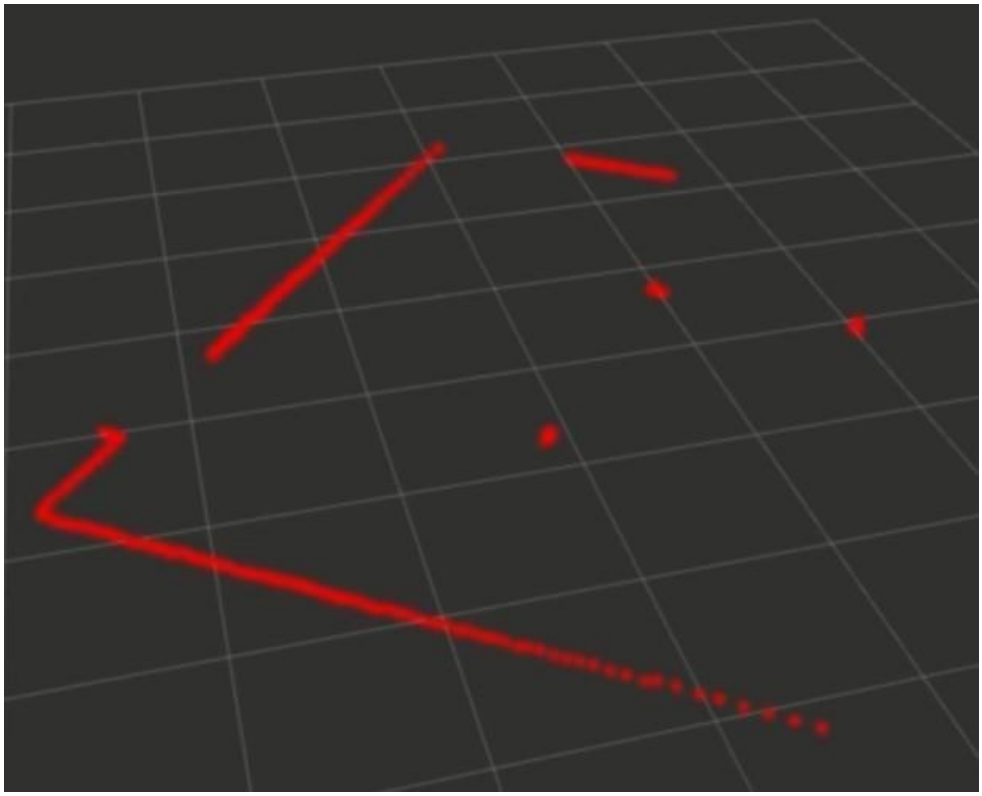
Rviz

RViz je ROS alat za vizualizaciju koji korisniku omogućuje da vidi sadržaj ROS poruka koje su objavljene na određenom topicu. RViz je ROS čvor koji se pretplati na jedan ili više topica i prikazuje primljene poruke na korisnički prijateljski način unutar svog grafičkog sučelja. Na primjer, RViz može vizualizirati mapu zgrade kojom se robot mora kretati, očitanja senzora poput laserskih skenera koji mjere udaljenost od prepreka, ili čak slike s kamera koje mogu biti montirane na robotu. RViz koristi standardne poruke na topicima za vizualizaciju različitih tipova podataka. Često su poruke koje prolaze kroz topice vrlo kompleksne i gotovo nemoguće razumjeti samo čitanjem njihovog sadržaja u terminalu. Na primjer, poruka koja predstavlja mapu naziva se zauzeti grid (occupancy grid) i sadrži vrlo dugačak vektor pun brojeva. Svaki od njih predstavlja jednu ćeliju, jedan piksel mape, a broj koji je s njim povezan označava je li područje mape potpuno zauzeto preprekom ili nepoznato, još neistraženo. RViz je koristan alat za razvoj robota jer omogućuje da se vizualizira ponašanje robota u realnom vremenu. To može biti korisno za otkrivanje pogrešaka u kodu, optimizaciju algoritama i razumijevanje interakcije robota s okolinom. Rviz u razvoju robota se koristi na nekoliko načina. Vizualizacija mape okruženja i položaja robota na mapi. Vizualizacija očitanja senzora, poput laserskih skenera, sonara i kamera. Vizualizacija putanje kojom se robot kreće. Vizualizacija izlaza algoritama za upravljanje robotom, poput algoritama za izbjegavanje prepreka i navigaciju. RViz je moćan alat koji može značajno pomoći razvoju robota. Njegova jednostavnost upotrebe i široka podrška za različite tipove podataka čini ga idealnim alatom za vizualizaciju robota u realnom vremenu.



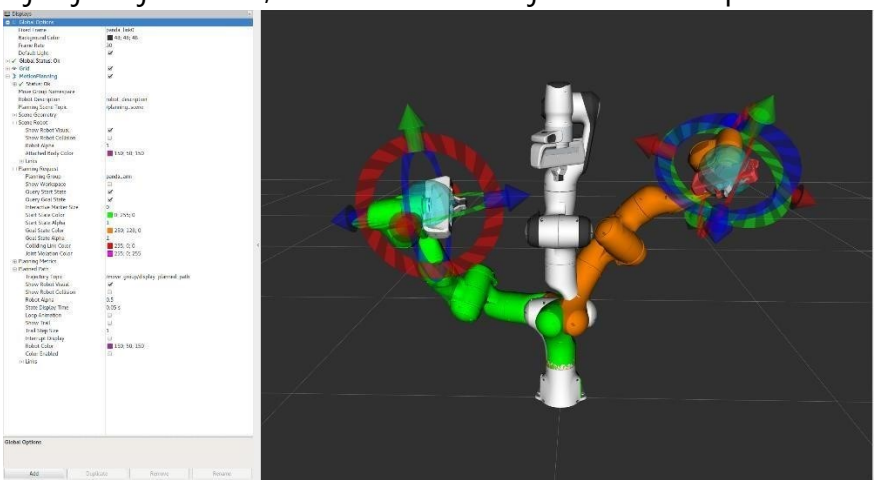
Slika 6. Occupancy grid vektor

ROS čvor koji se pretplati na topic na kojem se objavljuje poruka mape (occupancy grid) može koristiti RViz kako bi vizualizirao mapu na korisnički prijateljski način. To omogućuje da se lako vidi jesu li prepreke dodane na ispravnom mjestu, bez potrebe za čitanjem vrijednosti svakog piksela mape u terminalu. Slično tome, RViz se može koristiti za vizualizaciju očitanja senzora, kao što su laserski skeneri. U ovom slučaju, RViz može prikazati udaljenosti od prepreka u svim smjerovima detektiranim skenerima, što je mnogo korisnije od čitanja sadržaja poruke u konzoli. RViz je moćan alat za vizualizaciju koji može značajno olakšati razvoj robota. Omogućuje da se vizualizira ponašanje robota u realnom vremenu, što može biti korisno za otkrivanje pogrešaka u kodu, optimizaciju algoritama i razumijevanje interakcije robota s okolinom.



Slika 7. Očitanja laserskog skenera u RViz-u

RViz ne samo da omogućuje vizualizaciju svih teško razumljivih poruka, već i omogućava to kada se te poruke objavljuju s visokom frekvencijom na topiku. Na primjer, u slučaju strujanja videozapisa poruka je teško razumljiva jer sadrži RGB boju svakog piksela slike i zato što prikazuje strujanje videozapisa. Ovo se postiže konstantnim objavljivanjem slika, visokom frekvencijom unutar topika.



Slika 8. Robotska ruka u Rviz-u

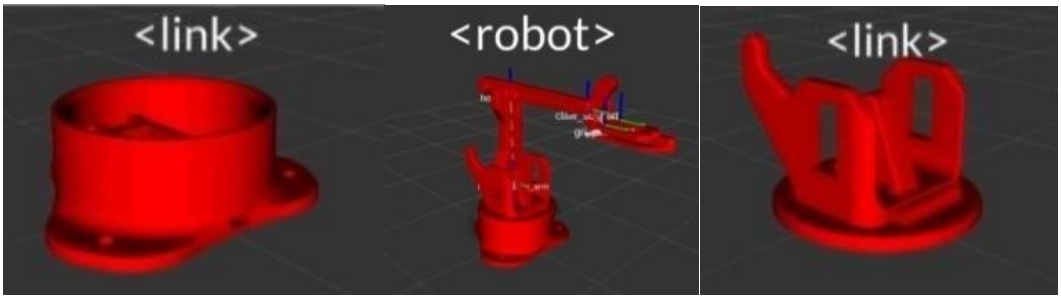
URDF

URDF (Unified Robot Description Format) je XML format za opis strukture robota. URDF model robota je XML datoteka u kojoj je svaka komponenta robota definirana

pomoću XML oznake. Komponenta robota može biti fizički dio robota, poput ruke (označena pomoću oznake <link>), ili spoj između dva fizička dijela robota (označen pomoću oznake <joint>). URDF model robota se koristi u ROS-u za vizualizaciju, simulaciju i upravljanje robotom. Na primjer, URDF model robota se može koristiti u RVizu za vizualizaciju robota i njegove okoline. URDF model robota se također može koristiti u GAZEBU, simulacijskom okruženju za robote, za simulaciju robota i njegove interakcije s okolinom. Na kraju, URDF model robota se može koristiti u ROS-ovim čvorovima koji upravljaju robotom, poput čvorova za upravljanje kretanjem robota ili upravljanje njegovim manipulatorom.

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">

    <!-- Links -->

    <link name="world"/>

    <link name="base_link">
        <visual>
            <origin rpy="0 0 0" xyz="-0.5 -0.5 0"/>
            <geometry>
                <mesh filename="package://arduinobot_description/mesh/basement.STL"/>
            </geometry>
        </visual>
        <collision>
            <origin rpy="0 0 0" xyz="-0.5 -0.5 0"/>
            <geometry>
                <mesh filename="package://arduinobot_description/mesh/basement.STL"/>
            </geometry>
        </collision>
    </link>

    <!-- Joints -->
    <joint name="virtual_joint" type="fixed">
        <parent link="world"/>
        <child link="base_link"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </joint>

</robot>
```

Slika 8. URDF datoteka



Slika 9. <robot> i <link> tagovi

ROS Parameter Server

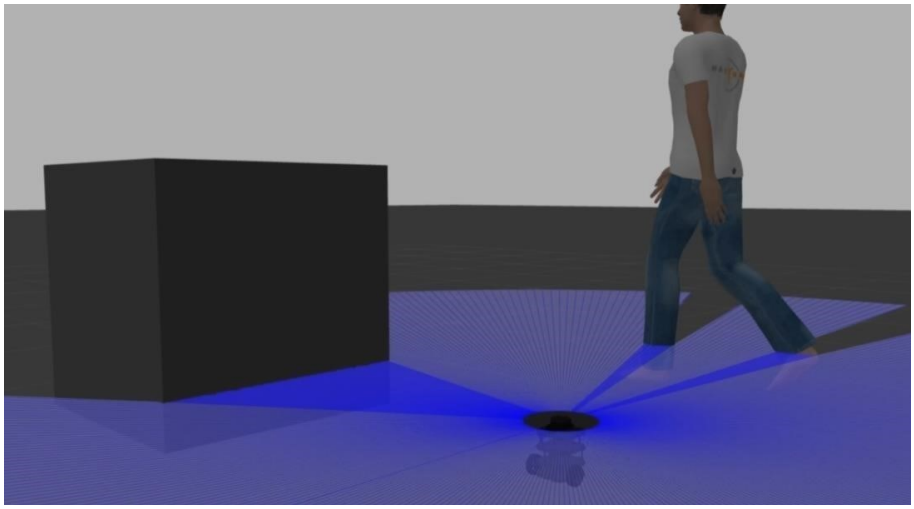
ROS parametarski poslužitelj je alat koji omogućuje ROS čvorovima da dijele varijable, poznate kao parametri, međusobno. Parametri su identificirani određenim imenom i sadržavaju određenu vrijednost. Parametarski poslužitelj je automatski pokrenut s ROS glavnim čvorom, koji se pokreće naredbom roscore. Svaki ROS čvor može učitati

parametar na parametarski poslužitelj. Parametar je pohranjen na poslužitelju i dostupan je svim drugim ROS čvorovima. Čak i ako čvor koji je učitao parametar zaustavi svoje izvršavanje, parametar će i dalje biti dostupan drugim čvorovima. Parametarski poslužitelj se koristi u slučajevima poput dijeljenja konfiguracijskih parametara robota koji se ne mijenjaju često. Dijeljenje podataka koji se ne mijenjaju često, kao što su mape ili raspored senzora. Dijeljenje podataka koje treba koristiti samo jednom, kao što je rezultat obrade slike. Parametarski poslužitelj je asinkroni alat, što znači da čvorovi koji dijele parametre ne moraju biti aktivni u isto vrijeme. Parametarski poslužitelj se automatski uništava kada se zaustavi ROS Master. Stoga, svi parametri koji su pohranjeni na poslužitelju su izgubljeni kada se zaustavi ROS sesija. Ključne značajke Omogućuje ROS čvorovima da dijele varijable međusobno. Parametri su identificirani određenim imenom i sadržavaju određenu vrijednost. Parametarski poslužitelj je automatski pokrenut s ROS glavnim čvorom. Parametri su dostupni svim ROS čvorovima, čak i ako čvor koji ih je učitao zaustavi svoje izvršavanje. Parametarski poslužitelj je asinkroni alat. Parametarski poslužitelj se automatski uništava kada se zaustavi ROS Master. ROS parametarski poslužitelj je moćan alat koji može pomoći u pojednostavljenju komunikacije između ROS čvorova.

Gazebo

Gazebo je simulator robota koji omogućuje simuliranje robota, njegovih senzora i motora, kao i njihovu interakciju s okolnim okruženjem. Gazebo simulira fizički model robota koristeći parametre poput mase, brzine, trenja i slično. Na temelju tih informacija, Gazebo može simulirati ponašanje robota podvrgnuto vanjskim silama poput gravitacije ili težine. Gazebo se može koristiti za razvijanje i testiranje robotičkih algoritama prije njihova implementiranja na stvarnom robotu. Gazebo omogućuje simuliranje različitih scenarija i okruženja, što može pomoći u otkrivanju pogrešaka i optimizaciji algoritama. Gazebo se također može koristiti za edukaciju o robotici. Gazebo omogućava studentima da eksperimentiraju s različitim robotičkim konceptima i da razviju svoje razumijevanje ponašanja robota. Gazebo služi za ključne korake u izradi robota poput simuliranje robota, njegovih senzora i motora, simuliranje interakcije robota s okolnim okruženjem, simuliranje različitih scenarija i okruženja, mogućnost dodavanja fizičkih svojstava vezama i zglobovima robota i integraciju s ROS-om. Gazebo se koristi u slučajevima poput razvoja i testiranja robotičkih algoritama prije njihova implementiranja na stvarnom robotu. Simuliranje različitih scenarija i okruženja za otkrivanje pogrešaka i optimizaciju algoritama. Edukacija o robotici i eksperimentiranje s različitim robotičkim konceptima. Gazebo je moćan simulator robota koji se široko koristi u razvoju, testiranju i edukaciji o robotici. Gazebo omogućuje simuliranje robota, njegovih senzora i motora, kao i

njihovu interakciju s okolnim okruženjem. Gazebo je integriran s ROS-om, što omogućuje simuliranje robota koji koriste ROS čvorove.



Slika 10. Gazebo simulacija

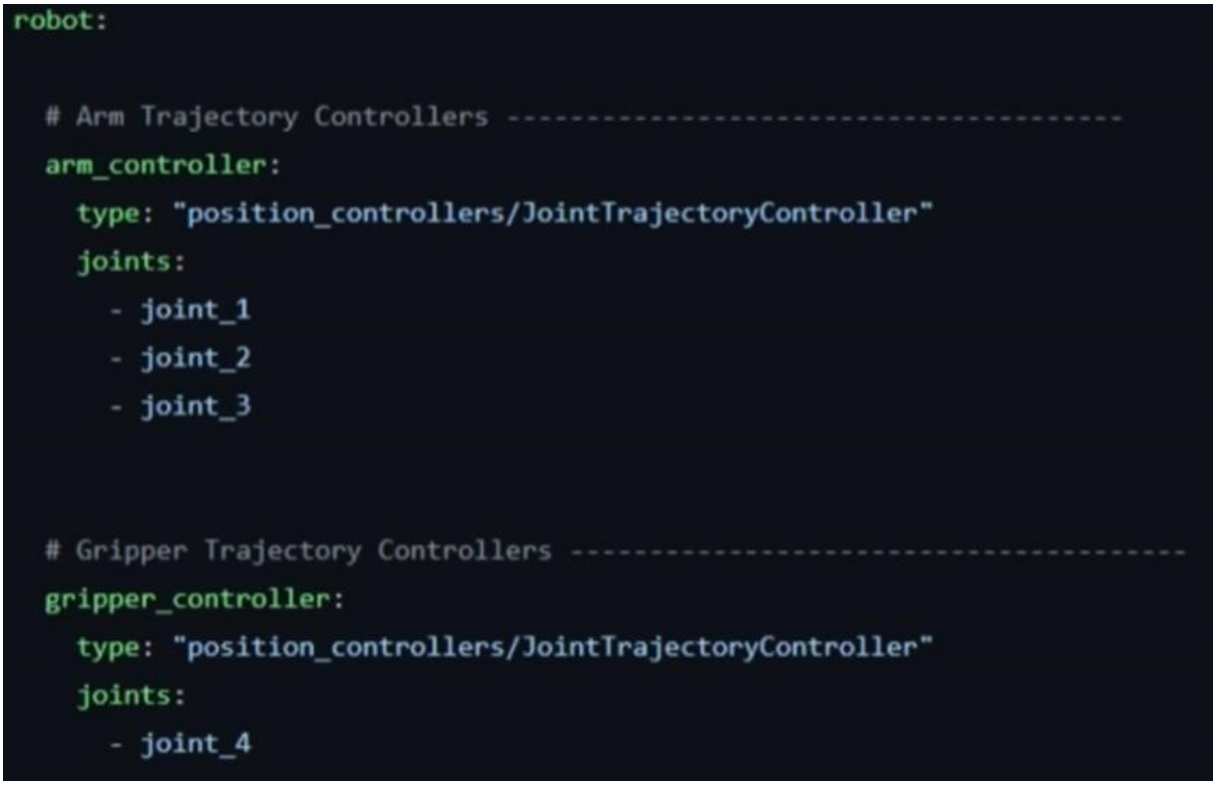
ROS kontrola

Nakon što se razvije digitalni dvojnik robota, moguće je nastaviti s razvojem funkcionalnosti robota. Čak i ako još nije dostupan stvarni robot, moguće je vidjeti kako funkcionalnosti funkcioniraju u simuliranom okruženju. U ovom dijelu, predstavit će se prva i vjerojatno najosnovnija funkcionalnost koju svaki robot treba implementirati, a to je pokretanje motora. Ova funkcionalnost, iako najosnovnija, također je ključna za sve ostale module i čvorove koji čine softver robota i koji pokreću robota određenom logikom. Kada se razvije ova osnovna funkcionalnost, moći će se slati naredbe zglobovima, ili bolje rečeno motorima koji pokreću zglobove, i dobiti kretanje robota kao posljedicu poslane naredbe. Akcija slanja naredbe motoru i praćenje njegovog kretanja kako bi se poklopilo s željenim naziva se upravljanje. Osnovno, ovo je polje proučavanja koje se bavi matematikom koja opisuje ponašanje komponente nazvane kontroler. Svrha ovog objekta je jednostavna i sastoji se od proučavanja tehnika koje nam omogućuju definiranje varijable nazvane naredba, koja se koristi kao ulaz za sustav gdje, primjerice, sustav može biti motor. U praksi, želimo poslati naredbu, cilj sustavu za upravljanje. Na primjer, želimo da se osi motora okreću brzinom od jednog radijana u sekundi, a sustav za upravljanje će se pobrinuti za premještanje motora prema potrebi. Ovaj cilj koji šaljemo sustavu za upravljanje naziva se ulaz i varijabla je koja sadrži željenu vrijednost za kretanje sustava. Ovaj ulaz zatim se uspoređuje s trenutnim statusom sustava. To znači s trenutnim statusom

motora, na primjer. Razlika između trenutnog statusa motora, koji je varijabla nazvana izlaz, i cilja postavljenog za kretanje robota, što je varijabla nazvana ulaz, proizvodi novu varijablu nazvanu greška. Ova varijabla ukazuje točno koliko je trenutni status motora, dakle izlazna varijabla, udaljen od svoje željene vrijednosti. Dakle, od ulazne varijable. Svrha stvaranja sustava za upravljanje je razvoj modula nazvanog Kontroler koji prima kao ulaz ovu varijablu greške i pretvara je u ulaz za sustav, u ovom slučaju za motor. Ovaj izračun se provodi s ciljem minimiziranja varijable greške kako bi njena vrijednost postajala sve bliža nuli i to u što kraćem vremenu. Da bismo razvili sustav za upravljanje u ROS-u, koristi se biblioteka nazvana ROS Control koju možete koristiti za razvoj vlastitog kontrolera ili također za ponovnu upotrebu nekih kontrolera koji su već implementirani. Upravljanje ROS kontrolerima sastoji se od dvije glavne komponente koje su upravitelj kontrolera i sučelje s hardverom. Sučelje s hardverom, kao što i samo ime kaže, odgovorno je za interakciju s hardverom robota, bilo da je riječ o stvarnom robotu ili simuliranom u Gazebo-u.

Konfiguracija

ROS robotske aplikacije se razvijaju tako da se mogu izvoditi na različitim vrstama robota. Međutim, da bi se to postiglo, aplikacija se razvija na način da je općenita i da se treba konfigurirati prema specifičnim karakteristikama robota na kojem se koristi. To se postiže u ROS-u putem YAML konfiguracijskih datoteka. YAML konfiguracijske datoteke definirane su na jednostavan i sveobuhvatan način i sadrže sve konfiguracijske parametre čvora. Ovi parametri se učitavaju u ROS parametarski poslužitelj u jednom koraku putem jedne naredbe, čime su svi dostupni u ROS-u. Parametar u YAML datoteci sastoji se od dvije komponente: imena parametra i njegove vrijednosti.



Slika 11. YAML konfiguracija

ROS Timer

ROS-u je dostupan ugrađeni tajmer koji označava kretanje vremena i pruža ovu informaciju svim ostalim čvorovima. Poznavanje kretanja vremena je važno u robotskim aplikacijama iz nekoliko razloga. Jedan od razloga je potreba za sinkronizacijom između različitih senzora i aktuatora robota. Na primjer, robotska aplikacija za autonomnu navigaciju može koristiti podatke s kamere i laserskog skenera za generiranje mape okruženja i planiranje putanje. Ako ovi senzori nisu sinkronizirani, podaci s kamere i laserskog skenera mogu biti nekonzistentni, što može dovesti do pogrešnog generiranja mape i planiranja putanje. Drugi razlog zašto je važno znati kretanje vremena je potreba za detektiranjem promjena u okruženju. Na primjer, robotska aplikacija za hvatanje objekta može koristiti podatke s kamere za detekciju objekta koji treba uhvatiti. Međutim, ako robotska aplikacija ne zna koliko vremena je prošlo od trenutka kada je kamera snimila sliku do trenutka kada je aplikacija obradila sliku, robotska aplikacija neće moći znati je li se položaj objekta promijenio u tom vremenu. ROS koristi vrijeme koje pruža ugrađeni tajmer za sinkronizaciju između različitih čvorova i detekciju promjena u okruženju. Jedna od mogućnosti za korištenje kretanja vremena u ROS-u je korištenje alata nazvanog Timer. Timer radi na isti način kao tajmer na vašem pametnom telefonu. Možete postaviti trajanje za tajmer i kad pokrenete tajmer, on će odbrojavati unatrag do kraja

trajanja. Kada tajmer istekne, on će pozvati i izvršiti funkciju koju ste definirali. Ova funkcija može biti bilo koja funkcija koja izvodi bilo koju akciju ili operaciju u ROS-u. Timer se može koristiti za izvođenje funkcionalnosti koja treba biti izvršena redovito i s određenom frekvencijom. Na primjer, timer se može koristiti za redovito provjeravanje je li robot dosegnuo svoju ciljanu poziciju ili koja redovito provjerava prelaze li motori robota temperaturne granice.

ROS Servisi

ROS usluge su jedan od mehanizama komunikacije između čvorova u ROS-u. ROS usluga se sastoji od dvije komponente: uslužni server (service server) i uslužni klijent (service client). Uslužni server je čvor koji nudi određenu funkcionalnost drugim čvorovima, a uslužni klijent je čvor koji koristi funkcionalnosti koje nudi uslužni server. Komunikacija između uslužnog klijenta i uslužnog servera odvija se putem protokola "zahtjev-odgovor" (request-response). Kada uslužni klijent želi koristiti funkcionalnost uslužnog servera, on šalje zahtjev uslužnom serveru. Zahtjev može sadržavati argumente koji su potrebni uslužnom serveru za izvršenje funkcionalnosti. Uslužni server zatim izvršava svoju logiku i generira izlaz. Izlaz uslužnog servera sadržan je u drugoj poruci, odgovornoj poruci (response), koju server šalje klijentu na kraju svoje izvedbe. ROS usluge se koriste za razvijanje modularnih i ponovno koristivih robotskih aplikacija. Na primjer, uslužni server može implementirati funkcionalnost prepoznavanja lica. Uslužni klijenti tada mogu koristiti ovu funkcionalnost za prepoznavanje lica u svojim slikama. To omogućuje razvoj različitih robotskih aplikacija, kao što su robotski manipulatori koji prate lica ljudi ili roboti koji govore kada se detektira lice osobe u njihovoj okolini.

Robotska kinematika

Kinematika je grana matematike koja proučava kretanje krutog tijela u prostoru te izračunava njegovu poziciju, brzinu i ubrzanje u određenom referentnom okviru. U robotici se kinematika koristi za upravljanje robotima pomoću pomicanja njihovih hvataljki u kartezijanskom prostoru. Postoje dva glavna slučaja u kojima se kinematika koristi u robotici, direktna kinematika i inverzna kinematika. Direktna kinematika koristi se za određivanje pozicije hvataljke robota znajući kuteve njegovih zglobova. Inverzna kinematika koristi se za određivanje kutova zglobova robota znajući poziciju njegove hvataljke. Direktna kinematika definira skup jednadžbi koje povezuju poziciju

hvataljke, koja je nepoznata, s kutovima zglobova, koji su poznati parametri u jednadžbama. Inverzna kinematika izvodi skup jednadžbi u kojima je ovaj put pozicija hvataljke u prostoru poznata i želimo umjesto toga izračunati kut svakog zgloba koji će hvataljku postaviti u određenu poziciju i orijentaciju. Kada se kinematika implementira u robotu, njime se može upravljati na jednostavniji i intuitivniji način, direktnim pomicanjem hvataljke u kartezijanskom prostoru. Kinematički algoritmi koji rade u pozadini pretvaraju koordinata hvataljke u pokrete zglobova. Kinematika je važan alat u robotici koji omogućuje precizno upravljanje robotima. Koristi se u širokom spektru robotskih aplikacija, od industrijskih robota do autonomnih vozila. Primjer korištenja kinematike u robotici je robotska ruka koja se sastoji od rotacijskih zglobova. Robotska ruka može se pomicati u kartezijanskom prostoru pomicanjem svojih zglobova. Kinematika se može koristiti za određivanje kutova zglobova robotske ruke koji su potrebni za pomicanje hvataljke na određenu poziciju u prostoru. Kinematika je moćan alat koji omogućuje razvijanje sofisticiranih robotskih aplikacija. Koristi se u širokom spektru industrija, uključujući proizvodnju, logistiku i zdravstvo.

2D Translacija

2D translacija je pomak točke ili tijela u dvije dimenzije. To se može predstaviti vektorom translacije, koji ima magnitudu i smjer. Magnituda vektora translacije predstavlja udaljenost koju je točka ili tijelo prešlo, a smjer vektora translacije predstavlja smjer u kojem se točka ili tijelo premjestilo.

2D translacija se može opisati pomoću dviju jednadžbi:

x' = x + Tx y'

= y + Ty

gdje je:

(x, y) koordinata točke ili tijela prije translacije

(x', y') koordinata točke ili tijela nakon translacije

Tx i Ty su komponente vektora translacije u x i y smjeru, respektivno

2D translacija je linearna transformacija, što znači da ako pomičemo točku ili tijelo dva puta, konačni rezultat je isti kao da smo točku ili tijelo pomaknuli jednom za udaljenost koja je jednaka zbroju dviju udaljenosti translacije.

2D translacija se koristi u mnogim različitim primjenama, kao što su:

Robotika: za upravljanje kretanjima robota

Animacija: za animaciju likova i objekata

Računalna grafika: za transformaciju objekata u 3D prostoru

Primjer 2D translacije je kretanje auta po cesti. Auto se može predstaviti točkom u 2D prostoru, a njegovo kretanje se može predstaviti vektorom translacije. Magnituda vektora translacije predstavlja udaljenost koju je auto prešao, a smjer vektora translacije predstavlja smjer u kojem se auto premjestilo. 2D translacija se može koristiti i za opisivanje kretanja složenijih sustava, kao što su npr. ljudi i životinje. Na primjer, kretanje čovjeka pri hodanju se može predstaviti kao niz 2D translacija. 2D translacija je važan koncept u kinematici, koji se koristi za opisivanje kretanja točaka i tijela u prostoru.

2D Rotacija

2D rotacija je rotacija točke ili tijela oko osi u dvije dimenzije. To se može predstaviti maticom rotacije, koja ima dimenzije 2x2. Matica rotacije ima sljedeću formu:

R = [[cos(theta), -sin(theta)],

[sin(theta), cos(theta)]] gdje je theta kut

rotacije u radijanima.

2D rotacija se može opisati pomoću sljedećih jednadžbi:

x' = Rx y'

= Ry

gdje je:

- (x, y) koordinata točke ili tijela prije rotacije
- (x', y') koordinata točke ili tijela nakon rotacije
- R je matica rotacije

2D rotacija je linearna transformacija, što znači da ako rotiramo točku ili tijelo dva puta, konačni rezultat je isti kao da smo točku ili tijelo rotirali jednom za udaljenost koja je jednaka zbroju dviju udaljenosti rotacije.

2D rotacija se koristi u mnogim različitim primjenama, kao što su:

- Robotika: za upravljanje kretanjima robota
- Animacija: za animaciju likova i objekata
- Računalna grafika: za transformaciju objekata u 3D prostoru

Primjer 2D rotacije je kretanje auta oko zavoja. Auto se može predstaviti točkom u 2D prostoru, a njegovo kretanje se može predstaviti maticom rotacije. Magnituda rotacije predstavlja kut za koji se auto rotiralo, a smjer rotacije predstavlja smjer u kojem se auto rotiralo.

2D rotacija se može koristiti i za opisivanje kretanja složenijih sustava, kao što su npr. ljudi i životinje. Na primjer, kretanje čovjeka prilikom okretaja oko sebe se može predstaviti kao niz 2D rotacija.

2D rotacija je važan koncept u kinematici i robotici, koji se koristi za opisivanje kretanja točaka i tijela u prostoru.

U robotici se 2D rotacija koristi za upravljanje kretanjima robota. Na primjer, robot koji ima kinematičku strukturu sličnu ljudskoj ruci može koristiti 2D rotaciju za upravljanje kretanjima svojih zglobova.

2D RotoTranslacija

D rototranslacija je kombinacija rotacije i translacije u dvije dimenzije. To se može predstaviti matricom rototranslacije, koja ima dimenzije 3x3. Matrica rototranslacije ima sljedeću formu:

```
T = [[cos(theta), -sin(theta), Tx],
     [sin(theta), cos(theta), Ty],
     [0, 0, 1]] gdje
```

je:

- theta kut rotacije u radijanima
- Tx i Ty su komponente vektora translacije u x i y smjeru, respektivno

2D rototranslacija se može opisati pomoću sljedećih jednadžbi:

x' = Tx + cos(theta) * x - sin(theta) * y y'

= Ty + sin(theta) * x + cos(theta) * y

gdje je:

- (x, y) koordinata točke ili tijela prije rototranslacije
- (x', y') koordinata točke ili tijela nakon rototranslacije

2D rototranslacija je linearna transformacija, što znači da ako se točku ili tijelo pomakne dva puta, konačni rezultat je isti kao da se točku ili tijelo pomaknulo jednom za udaljenost koja je jednaka zbroju dviju udaljenosti rototranslacije.

2D rototranslacija se koristi u mnogim različitim primjenama, kao što su:

- Robotika: za upravljanje kretanjima robota
- Animacija: za animaciju likova i objekata
- Računalna grafika: za transformaciju objekata u 3D prostoru

Primjer 2D rototranslacije je kretanje auta oko zavoja s promjenom brzine. Auto se može predstaviti točkom u 2D prostoru, a njegovo kretanje se može predstaviti matricom rototranslacije. Magnituda rotacije predstavlja kut za koji se auto rotiralo, a smjer rotacije predstavlja smjer u kojem se auto rotiralo. Magnituda translacije predstavlja udaljenost koju je auto prešlo, a smjer translacije predstavlja smjer u kojem se auto premjestilo.

2D rototranslacija se može koristiti i za opisivanje kretanja složenijih sustava, kao što su npr. ljudi i životinje. Na primjer, kretanje čovjeka prilikom hodanja, trčanja ili skakanja se može predstaviti kao niz 2D rototranslacija.

2D rototranslacija je važan koncept u kinematici i robotici, koji se koristi za opisivanje kretanja točaka i tijela u prostoru.

U robotici se 2D rototranslacija koristi za upravljanje kretanjima robota. Na primjer, robot koji ima kinematičku strukturu sličnu ljudskoj ruci može koristiti 2D rototranslaciju za upravljanje kretanjima svojih zglobova.

3D Translacija

Translacija u trodimenzionalnom prostoru je kretanje objekta u prostoru bez promjene orijentacije.

Translacija u trodimenzionalnom prostoru se može matematički predstaviti kao vektor translacije. Vektor translacije je vektor koji povezuje središte referentnog okvira iz kojeg se objekt giba sa središtem referentnog okvira u koji se objekt giba.

Razmotrimo robotsku ruku koja se sastoji od dvije poveznice. Prva poveznica je vezana za bazu robota, a druga poveznica je vezana za prvu poveznicu.

Ako želimo definirati položaj hvataljke robota, moramo definirati položaj točke P na drugoj poveznici.

Pozicija točke P u referentnom okviru L može se izračunati pomoću sljedeće formule:

x_P = L * cos(alfa)

y_P = L * sin(alfa)

z_P = 0 gdje je:

- x_P je koordinata točke P duž X osi u referentnom okviru L
- y_P je koordinata točke P duž Y osi u referentnom okviru L
- z_P je koordinata točke P duž Z osi u referentnom okviru L
- L je duljina druge poveznice
- alfa je kut između prve i druge poveznice

Ako transliramo referentni okvir L za vektor translacije T, koordinate točke P u referentnom okviru W mogu se izračunati pomoću sljedeće formule:

x_P = x_P + T_x y_P

= y_P + T_y z_P =

z_P + T_z gdje je:

- x_P je koordinata točke P duž X osi u referentnom okviru W
- y_P je koordinata točke P duž Y osi u referentnom okviru W
- z_P je koordinata točke P duž Z osi u referentnom okviru W
- T_x je komponenta vektora translacije T duž X osi
- T_y je komponenta vektora translacije T duž Y osi
- T_z je komponenta vektora translacije T duž Z osi

3D Rotacija

Rotacija u trodimenzionalnom prostoru je kretanje objekta oko jedne osi ili više osi. Rotacija određuje orijentaciju objekta u prostoru.

Rotacija u trodimenzionalnom prostoru se može matematički predstaviti kao matrica rotacije. Matrica rotacije je matrica koja povezuje koordinate točke u jednom referentnom okviru s koordinatama iste točke u rotiranom referentnom okviru.

Elementarna rotacija je rotacija oko jedne osi. U trodimenzionalnom prostoru postoje tri elementarne rotacije: rotacija oko X-osi, rotacija oko Y-osi i rotacija oko Z-osi.

Matrice rotacije za elementarne rotacije su sljedeće:

- Matrica rotacije oko X-osi za kut alfa:
- [1 0 0]
- [0 cos(alfa) -sin(alfa)]
- [0 sin(alfa) cos(alfa)]
- Matrica rotacije oko Y-osi za kut data:
- [cos(data) 0 sin(data)]
- [0 1 0]
- [-sin(data) 0 cos(data)]
- Matrica rotacije oko Z-osi za kut gama:
- [cos(gama) -sin(gama) 0]
- [sin(gama) cos(gama) 0]
- [0 0 1]

Opća rotacija u trodimenzionalnom prostoru je kombinacija elementarnih rotacija. Matrica rotacije za opću rotaciju se može dobiti kompozicijom matrica rotacije za elementarne rotacije.

Razmotrimo robotsku ruku koja se sastoji od dvije poveznice. Prva poveznica je vezana za bazu robota, a druga poveznica je vezana za prvu poveznicu.

Ako se želi definirati orijentaciju hvataljke robota, mora se definirati rotaciju druge poveznice u odnosu na bazu robota.

Orijentaciju druge poveznice se definira pomoću matrice rotacije. Matrica rotacije druge poveznice u odnosu na bazu robota povezuje koordinate točke na drugoj poveznici u referentnom okviru druge poveznice s koordinatama iste točke na drugoj poveznici u referentnom okviru baze robota.

3D RotoTranslacija

Translacijsko kretanje u trodimenzionalnom prostoru je kretanje objekta bez promjene orijentacije. Ono se može definirati kao kompozicija rotacijskog kretanja i translacijskog kretanja u istoj matrici nazvanoj matricom transformacije.

Translacijsko kretanje u trodimenzionalnom prostoru se može matematički predstaviti matricom transformacije. Matrica transformacije je matrica koja povezuje koordinate točke u jednom referentnom okviru s koordinatama iste točke u drugom referentnom okviru.

Matrica transformacije za translacijsko kretanje u trodimenzionalnom prostoru se može izračunati sljedećim postupkom:

1. Izračunati rotacijsku matricu koja povezuje prvi referentni okvir s drugim referentnim okvirom.
2. Izračunati vektor translacije koji povezuje središte prvog referentnog okvira sa središtem drugog referentnog okvira.
3. Konstruirati matricu transformacije sljedećim načinom:

[R | T] [0

0 0 1]

gdje je:

- R rotacijska matrica
- T vektor translacije

Razmotrimo robotsku ruku koja se sastoji od dvije poveznice. Prva poveznica je vezana za bazu robota, a druga poveznica je vezana za prvu poveznicu.

Ako želimo izračunati položaj hvataljke robota u referentnom okviru svijeta, moramo izračunati matricu transformacije koja povezuje referentni okvir hvataljke s referentnim okvirom svijeta.

Matrica transformacije se može izračunati pomoću rotacijske matrice druge poveznice u odnosu na bazu robota i vektora translacije koji povezuje središte referentnog okvira druge poveznice sa središtem referentnog okvira svijeta.

Direktna kinematika

Direktna kinematika je problem određivanja položaja i orijentacije krajnjeg efektora robota, znajući kuteve njegovih zglobova. Ovaj problem se može riješiti koristeći matrice transformacije.

Matrice transformacije su matrice koje povezuju koordinate točke u jednom referentnom okviru s koordinatama iste točke u drugom referentnom okviru. Matrica transformacije se sastoji od rotacijske matrice i translacijskog vektora. Rotacijska matrica predstavlja orijentaciju jednog referentnog okvira u odnosu na drugi referentni okvir, a translacijski vektor predstavlja relativni položaj središta dvaju referentnih okvira.

Za rješavanje problema direktne kinematike, potrebno je izračunati matricu transformacije koja povezuje krajnji efektor robota sa svjetskim referentnim okvirom. Ova matrica se može izračunati množeći matrice transformacije za sve zglobove robota, počevši od krajnjeg efektora i krećući se prema bazi robota.

Sljedeći primjer pokazuje kako izračunati matricu transformacije za krajnji efektor robota s tri zgloba:

T_EE_W = T_3_2 * T_2_1 * T_1_W gdje

je:

- T_EE_W matrica transformacije koja povezuje krajnji efektor robota sa glavnim referentnim okvirom
- T_3_2 matrica transformacije koja povezuje treći zglob robota s drugim zglobom robota
- T_2_1 matrica transformacije koja povezuje drugi zglob robota s prvim zglobom robota
- T_1_W matrica transformacije koja povezuje prvi zglob robota sa glavnim referentnim okvirom

Matrice transformacije za pojedine zglobove robota se mogu izračunati koristeći informacije o duljini veze robota i kutu zgloba.

Kada se izračuna matrica transformacije koja povezuje krajnji efektor robota sa glavnim referentnim okvirom, položaj i orijentaciju krajnjeg efektora robota se mogu odrediti koristeći sljedeće jednadžbe:

p_EE_W = T_EE_W(1:3, 4)

R_EE_W = T_EE_W(1:3, 1:3) gdje

je:

- p_EE_W vektor položaja krajnjeg efektora robota u glavnom referentnom okviru
- R_EE_W rotacijska matrica koja predstavlja orijentaciju krajnjeg efektora robota u odnosu na glavni referentni okvir

Direktna kinematika je važan koncept u robotici, jer se koristi za upravljanje robotskim manipulatorima. Korištenjem direktne kinematike, mogu se izračunati kutevi zglobova robota potrebni za premještanje krajnjeg efektora robota u željenu poziciju i orijentaciju.

Inverzna kinematika

Inverzna kinematika je problem određivanja pokreta zglobova robota kako bi se dobio željeni položaj i orijentacija efektora robota. To je suprotan problem od

direktne kinematike, koja se bavi određivanjem položaja i orijentacije efektora robota na temelju pokreta njegovih zglobova.

Inverzna kinematika je kompleks problem koji nema uvijek rješenja. Na primjer, robot s tri ili više zglobova može imati više različitih kombinacija pokreta zglobova koji rezultiraju istim položajem i orijentacijom efektora robota. Ovaj se problem naziva višestruko rješenje inverzne kinematike.

Postoji nekoliko različitih pristupa rješavanju problema inverzne kinematike. Jedan od najčešćih pristupa je korištenje analitičkih rješenja. Analitička rješenja su matematička rješenja koja se mogu izračunati u realnom vremenu. Međutim, analitička rješenja nisu uvijek dostupna za sve robote.

Drugi pristup rješavanju problema inverzne kinematike je korištenje numeričkih rješenja. Numerička rješenja su rješenja koja se aproksimiraju koristeći numeričke metode. Numerička rješenja su dostupna za sve robote, ali mogu biti usporenija od analitičkih rješenja.

Inverzna kinematika je važan problem u robotici, jer se koristi za upravljanje kretanjima robota. Na primjer, robot koji ima kinematičku strukturu sličnu ljudskoj ruci može koristiti inverznu kinematiku za upravljanje kretanjima svojih zglobova kako bi dobio željeni položaj i orijentaciju ruke.

MoveIt

MOVE IT je paket za planiranje putanja i izbjegavanje prepreka za robotske ruke. Glavna komponenta MOVE IT-a je čvor MOVE grupe koji upravlja svim funkcionalnostima i sučeljima redova. Svrha ovog čvora je omogućiti komunikaciju između svih modula MOVE IT-a i dopustiti razmjenu poruka između svih ovih funkcionalnosti te ih učiniti dostupnima korisniku putem grafičkog korisničkog sučelja, ali i putem programskog sučelja ili API-ja.

Jedna od značajki dostupnih u MOVE IT-u je inverzna kinematika. Inverzna kinematika je problem određivanja kuteva zglobova robota potrebnih za premještanje krajnjeg efektora robota u željenu poziciju i orijentaciju. MOVE IT koristi općeniti rješavač za rješavanje problema inverzne kinematike za generičku konfiguraciju generičke robotske ruke. Kasnije se može prilagoditi i koristiti rješavače koji su implementirani za specifičnu strukturu robota.

MOVE IT također ima mogućnost upravljanja planskom scenom. Planska scena je virtualni prikaz okolnog okruženja robota, uključujući prepreke, zidove, predmete za hvatanje i slično. MOVE IT koristi plansku scenu tijekom planiranja trajektorija kako bi izbjegao prepreke i planirao trajektorije bez sudara. Kako biste omogućili i koristili ovu funkcionalnost, trebate robotu pružiti senzor koji mu omogućava da zna kakvo je okolno okruženje. To može biti, primjerice, laserski senzor, kamera i slično.

Kada dva modula za planiranje i izbjegavanje prepreka završe izvođenje, MOVE IT također sadrži modul za implementaciju i provjeru trajekcije. Tijekom izvođenja, ovaj je modul koji ćemo povezati s našim upravljačkim sustavom kako bismo poslali trajekciju svakom zglobu robota i omogućili motorima, bilo stvarnim ili simuliranim, da slijede traženu trajekciju.

Konačno, kako bismo iskoristili sve ove komponente, MOVE IT nudi dvije vrste sučelja korisniku. Grafičko sučelje za upravljanje robotom u usluzi i također softversko sučelje putem API-ja koje omogućava korištenje svih MOVE IT funkcionalnosti u drugim aplikacijama. Primjerice, implementiranje ROS čvora koji pomiče hvataljku prema određenoj logici koristeći sve MOVE IT funkcionalnosti za amplifikaciju trajekcije i inverzne kinematike.

Robot aplikacije

Robot aplikacije su softverski programi koji se koriste za upravljanje i kontroliranje robota. Ove aplikacije mogu se koristiti za razne svrhe, kao što su proizvodnja, logistika, usluge i zabava.

Jedan od najpopularnijih okvira za razvoj robot aplikacija je ROS (Robot Operating System). ROS je otvoreni softverski okvir koji pruža niz alata i biblioteka za razvoj robotskih aplikacija. ROS je modularan i skalabilni okvir koji se može koristiti za razvoj robot aplikacija za širok spektar robota, od jednostavnih robotskih ruku do složenih humanoidnih robota.

MOVE IT je jedan od najpopularnijih paketa u ROS-u. MOVE IT pruža niz funkcionalnosti za upravljanje robotskim manipulatorima, kao što su inverzna kinematika, planiranje trajektorija i izbjegavanje prepreka.

U ovom tekstu, opisat ćemo nekoliko primjena robot aplikacija koje se mogu razviti koristeći ROS i MOVE IT.

Ručno upravljanje robotskom rukom

Jedan od najjednostavnijih primjera robot aplikacije je aplikacija koja se koristi za ručno upravljanje robotskom rukom. Ova aplikacija omogućuje korisniku da odredi željeni položaj i orijentaciju krajnjeg efektora robotske ruke. MOVE IT se zatim koristi za planiranje i izvršavanje trajektorije krajnjeg efektora do željene pozicije i orijentacije.

Upravljanje robotskom rukom pomoću glasovnih naredbi

Druga primjena robot aplikacije je aplikacija koja se koristi za upravljanje robotskom rukom pomoću glasovnih naredbi. Ova aplikacija koristi glasovni asistent, kao što je Amazon Alexa, za prevođenje glasovnih naredbi korisnika u ciljeve za robotsku ruku. MOVE IT se zatim koristi za planiranje i izvršavanje trajektorije krajnjeg efektora do željene pozicije i orijentacije.

Razvijanje neuronskih mreža za prepoznavanje objekata i upravljanje robotom

MOVE IT se također može koristiti za razvijanje neuronskih mreža za prepoznavanje objekata i upravljanje robotom. Na primjer, neuronska mreža se može obučiti da prepoznaje objekte koje treba uhvatiti. Neuronska mreža zatim može generirati željene položaje i orijentacije krajnjeg efektora robotske ruke za hvatanje objekata. MOVE IT se zatim koristi za planiranje i izvršavanje trajektorije krajnjeg efektora do željene pozicije i orijentacije.

ROS akcije

ROS akcije su komunikacijski protokol između čvorova u ROS-u koji se koristi za pokretanje dugotrajnog zadatka na drugom čvoru, bez blokiranja klijentskog čvora. Ovaj protokol koristi dvije vrste poruka: zahtjev i odgovor. Kada klijent želi pokrenuti zadatak na poslužiteljskom čvoru, šalje zahtjevnu poruku poslužiteljskom čvoru. Poslužiteljski čvor zatim počinje izvoditi zadatak i šalje povratne poruke klijentskom čvoru kako bi ga informirao o statusu izvođenja. Kada poslužiteljski čvor završi izvođenje zadatka, šalje završnu poruku klijentskom čvoru s rezultatom zadatka.

ROS akcije su korisne za zadatke koji traju dulje vrijeme, kao što je premještanje robota na određeni položaj ili prepoznavanje objekta na slici. Korištenjem ROS akcija, klijentski čvor može pokrenuti zadatak na poslužiteljskom čvoru i nastaviti s izvođenjem svojih funkcionalnosti, bez čekanja na završetak zadatka. Ovo omogućuje klijentskom čvoru da bude više responzivan i da izvodi više zadataka istovremeno.

ActionLib

ROS akcije su komunikacijski protokol između čvorova u ROS-u koji se koristi za pokretanje dugotrajnog zadatka na drugom čvoru, bez blokiranja klijentskog čvora. Implementacija ovog protokola je definirana u ROS knjižnici nazvanoj ActionLib. Glavno sučelje između akcijskog servera i klijenta sastoji se od četiri poruke:

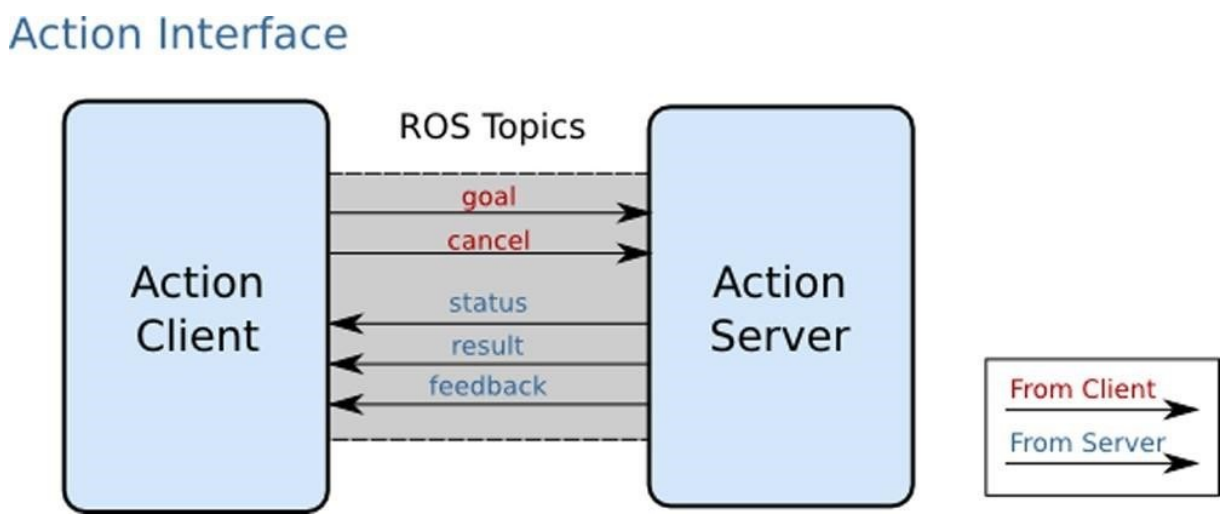
Cilj: Klijent šalje cilj serveru. Cilj može biti bilo koji podatak koji je potreban serveru za izvođenje zadatka.

Povratna poruka: Server šalje povratne poruke klijentu kako bi ga informirao o statusu izvođenja zadatka. Povratne poruke mogu sadržavati informacije o napretku izvođenja, kao i bilo kakve druge informacije koje su relevantne za klijenta.

Rezultat: Server šalje rezultat klijentu nakon što završi izvođenje zadatka. Rezultat može sadržavati informacije o uspješnosti izvođenja, kao i bilo kakve druge informacije koje su relevantne za klijenta.

Otkazivanje: Klijent može poslati poruku otkazivanja serveru kako bi zaustavio izvođenje zadatka. Server je zatim obvezan da zaustavi izvođenje zadatka i pošalje poruku s rezultatom klijentu.

ActionLib pruža niz funkcionalnosti koje čine razvoj akcijskih servisa i klijenata jednostavnijim i robusnijim. Na primjer, ActionLib automatski upravlja distribuiranom stanom zadatka i osigurava da klijent primi poruku s rezultatom, čak i ako server neočekivano padne. Primjer korištenja ActionLib-a je implementacija MoveIt! akcijskog servera. MoveIt! je ROS okvir za planiranje trajektorija i izbjegavanje prepreka za robotske ruke. MoveIt! akcijski server omogućuje klijentima da zahtijevaju od MoveIt-a da planira i izvrši trajektoriju za robotsku ruku. Klijent ne mora da blokira dok MoveIt! izvršava trajektoriju, već može nastaviti s izvođenjem svojih funkcionalnosti. ActionLib je moćan alat za razvijanje aplikacija koje koriste razne usluge na robotima. Korištenjem ActionLib-a, mogu se razviti aplikacije koje su više responzivne i mogu izvoditi više zadataka istovremeno.



Slika 12. ROS akcije

Action Server

```
class FibonacciActionServer
{
public:
  ros::NodeHandle nh_;
  actionlib::SimpleActionServer<robot_test::FibonacciAction> as_;
  std::string action_name_;
  // kreiraj poruke koje su koriste za published feedback/result
  robot_test::FibonacciFeedback feedback_;
  robot_test::FibonacciResult result_;

  FibonacciActionServer(std::string name) :
    as_(nh_, name, boost::bind(&FibonacciActionServer::executeCB, this, _1), false),
    action_name_(name)
  {
    as_.start();
    ROS_INFO("Simple Action Server Started");
  }

  void executeCB(const robot_test::FibonacciGoalConstPtr &goal)
  {
    ROS_INFO("Goal Received %i", goal->order);

    // pomočne varijable
    ros::Rate r(1);
    bool success = true;

    // push_back početak fibonacci sekvence
    feedback_.sequence.clear();
    feedback_.sequence.push_back(1);
    feedback_.sequence.push_back(1);

    // pokreni akciju
    for(int i=1; i<=(goal->order-1); i++)
    {
      // check that preempt has not been requested by the client
      if (as_.isPreemptRequested() || !ros::ok())
      {
        ROS_INFO("%s: Preempted", action_name_.c_str());
        // postavi stanje akcije na preempted
        as_.setPreempted();
        success = false;
        break;
      }
      feedback_.sequence.push_back(feedback_.sequence[i] + feedback_.sequence[i-1]);
      // publishaj feedback
      as_.publishFeedback(feedback_);
      r.sleep();
    }
```

Slika 20. ROS akcijski server 1. dio

```
    if(success)
    {
      result_.sequence = feedback_.sequence;
      ROS_INFO("%s: Succeeded", action_name_.c_str());
      // postavi stanje akcije na succeeded
      as_.setSucceeded(result_);
    }
  }
};


int main(int argc, char** argv)
{
  ros::init(argc, argv, "fibonacci");

  FibonacciActionServer fibonacci("fibonacci");
  ros::spin();

  return 0;
}
```

Slika 21. ROS akcijski server 2. dio

U trenutku kada je moguće koristiti radnje iz praktične perspektive, koristit će se ovaj alat kako bi se dodalo nove funkcionalnosti robotu. Primjer akcijskog servera na 20. i 21. slici računata brojeve koji pripadaju Fibonacci nizu do reda naznačenog u ciljnoj poruci. Fibonacci niz je skup cjelobrojnih brojeva koji započinje s dva broja jedan, i nastavlja se dodavanjem novog elementa ovom nizu. To je zbroj prethodna dva broja. Broj elemenata koji se pojavljuju u ovom nizu naziva se redom u Fibonacci nizu. Svrha akcijskog poslužitelja iz primjera je izdvojiti sve brojeve iz Fibonacci niza do određenog reda koji je definiran u ciljnoj poruci akcije. Prvo, stvara se instancu klase jednostavnog akcijskog poslužitelja koji sadrži sve korisne funkcije za stvaranje akcijskog poslužitelja. Iz biblioteke ActionLib, koristi se definicija jednostavnog akcijskog poslužitelja. U uglatim zagradama, mora se specificirati koja je poruka sučelje koje ovaj akcijski poslužitelj mora koristiti. Također, dodaje se atribut za string u klasu, koji će sadržavati ime akcijskog poslužitelja. Dodaje se još dva atributa, a to su poruka povratnih informacija akcije i poruka rezultata. Stvara se instancu povratne informacije Fibonacci zatim se stvara i instancu rezultata poruke. Stvara se instancu Fibonacci rezultata, zatim se izgradi konstruktor klase, koji je `FibonacciActionServer`. U suštini to je funkcija sa istim nazivom kao klasa i prima samo jedan ulaz, string. Naziva se name, to će odrediti ime akcijskog poslužitelja i to je konstruktor klase. Ovo će automatski biti pozvano kada se stvori novi objekt klase `FibonacciActionServer`. Kada se to dogodi, inicira se jednostavni akcijski poslužitelj iz knjižnice ActionLib. Tako da varijablu koja se naziva `as` prima kao ulazni čvor i varijablu koja se naziva `nh` također prima ime akcijskog poslužitelja koje je

definirano u varijabli name. Također, klasa jednostavnog akcijskog poslužitelja također prima kao ulaz ime funkcije koja se mora izvesti svaki put kad akcijski poslužitelj primi novi cilj. To se definira u klasi `FibonacciActionServer`, budući da se ova funkciju definira unutar klase `FibonacciActionServer`, prenosi se i pokazivač na trenutni objekt. Dakle, `this`. Budući da se ova funkcija poziva samo kad se primi novi cilj, prima samo ulazni parametar. Točno ciljna poruka koja je primljena. Koristi se zamjenski znak kako bi se naznačilo da ova funkcija treba samo jedan parametar. Zatim pruža se još jedan posljednji parametar konstruktoru jednostavnog akcijskog poslužitelja, a to je boolean koji se može postaviti na `false`. To sprečava da akcijski poslužitelj već bude pokrenut, jer prvo treba završiti inicijalizaciju ostalih atributa klase i zatim se može ručno pokrenuti. Dopušteno je također da se pridijeli vrijednost atributu koji se zove `action_name_` i postavlja se na `name` varijablu koja je primljena kao ulaz u konstruktor `FibonacciActionServer` klase. Sada se može ručno pokrenuti izvršenje akcijskog poslužitelja pozivom funkcije `start`. Dakle, na objektu koji se zove `action_server`, koristi se funkciju `start`, a zatim u tijelu konstruktora također ispisuje se poruku dnevnika koja obavještava korisnika da je akcijski poslužitelj ispravno pokrenut, da čeka poruke cilja i to se učiniti s "ROS info". To je sve što je potrebno za konstruktor klase `FibonacciActionServer` i za kreirati akcijski server. On sada sluša nove poruke i kad to učini, izvršava funkciju. Dakle, sada se može definirati ponašanje callback funkcije. Funkcija prima ulaznu poruku cilja akcije koju je upravo primio akcijski poslužitelj. Kako je definirali, poruka sadržava samo redoslijed Fibonacci niza koji poslužitelj mora izračunati. Prva stvar koju će se učiniti je ispisati poruku koja obavještava korisnika da je primljen novi cilj s "Ros.Info". Ispisat će se poruku "Cilj primljen" i ispisat će se redoslijed poruke koji je primljen. Dakle, ispisat će se cijeli broj u ovoj poruci i prikazat će se varijablu `sequence` koja je u poruci. Onda još uvijek unutar ove funkcije, stvorimo varijablu koja se zove `error` tipa ROS brzina. ROS brzina, naziva se varijablu `error` i inicijalizira ju se na 1. Ovo će odrediti frekvenciju kojom će se dodavati nove elemente u Fibonacci niz. Tako će to raditi jednom u sekundi. Zatim stvara se još jednu varijablu koja je boolean, `success`, inicijalizirana je na `true` i ova varijabla će ostati `true` osim ako akcijski poslužitelj ne primi zahtjev za otkazivanjem tijekom izvršenja akcije. Nastavlja se s inicijalizacijom poruke povratnih informacija. Tako da prvo, poništava ju se na prazan vektor. Uzima se poruku povratnih informacija i pristupa se vektoru koji se zove `sequence` i čisti se taj vektor. Također, već se može započeti dodjeljivati prvih dvaju brojeva koji uvijek pripadaju Fibonacci nizu, a oba su jedinica. Sada u petlji se dodaje jedan po jedan sve brojeve koji pripadaju Fibonacci nizu do željenog reda. Dakle, stvara se petlju i stvara se brojač koji ide od 1 do ciljnog reda minus 1. Budući da su već umetnuti prva dva elementa u niz i povećava se brojač za jedan svaki put kad se petlja izvrši, tako se dodaje novi element u niz. Budući da je novi element dodan u niz, sada se objavljuje poruka povratnih informacija s parcijalnim nizom koji je izračunat do sada. Kako bi se to moglo učiniti, koristi se funkcija objavi povratne informacije na objektu

jednostavnog akcijskog poslužitelja. To znači da se funkcija poziva na objektu akcijski poslužitelj i prenosi joj se poruka povratnih informacija koju je ispunjena. Poruka povratnih informacija je spremljena u varijabli povratne informacije. Zatim, prije nego što se pokrene nova petlja i doda novi element Fibonacci nizu, čeka se trenutak kako bi se imalo vremena provjeriti rezultate u terminalu tijekom izvršenja. To se radi tako da se doda odmor s funkcijom spavanje. Funkcija spavanje poziva se na objektu stopa i tako će se petlja izvršavati jednom po sekundi. Također, unutar ove petlje, prije nego što se doda bilo koji novi element u niz, prvo se provjerava je li akcijski poslužitelj primio poruku o otkazivanju od klijenta. To se provjerava funkcijom is prempt requested. Funkcija is prempt requested poziva se na objektu akcijski poslužitelj i zamata se u if izjavu. Također se provjerava je li ROS master i dalje u izvršenju. To znači da se u ovu if izjavu unosi sljedeće: ako je akcijski poslužitelj primio zahtjev za otkazivanjem ili ako je ROS master prestao s izvršenjem. Tako, ako se to dogodi, ako akcijski poslužitelj, na primjer, primi zahtjev za otkazivanjem, prvo se ispisuje poruka. Poruka se ispisuje s funkcijom ROS info. Ispisuje se da je akcija ime akcije prekinuta. Ime akcije je spremljeno u varijabli ime akcije. Ispisuje se kao string. Zatim također se mijenja status akcije u prekinut. To znači da se varijabla status akcije postavlja na prekinut. Također se postavlja vrijednost varijable succeded na false jer je akcija zaustavljena. Također, ako se to dogodi, dakle, ako je akcija otkazana, također se zaustavlja izvođenje petlje koja dodaje novi element u Fibonacci nizu s uputstvom break. Zatim, još uvijek unutar funkcije izvrši povratni poziv, nakon petlje, bilo da je akcijski poslužitelj završio bez problema ili ga je otkazano, vraća se poruka rezultata. To znači da ako je akcija uspješna, ako je succeded istinit, stvara se poruka rezultata. Poruka rezultata će sadržavati samo niz svih brojeva koji pripadaju Fibonacci poslužitelju. Budući da u ovom trenutku skripta već sadrži sve elemente Fibonacci niza, jednostavno se podaci rezultata postavljaju na one koji su u poruci povratnih informacija. To znači da se niz rezultata rezultat postavlja na niz niza povratne informacije. Zatim također se ispisuje još jedna poruka dnevnika u konzoli. To znači da se funkcijom ROS info ispisuje da je akcija ime akcije uspješna. Ime akcije je spremljeno u varijabli ime akcije. Također se postavlja status akcijskog poslužitelja na uspjelo pomoću funkcije set succeded. Funkcija set succeded poziva se na objektu akcijski poslužitelj i prenosi joj se poruka rezultata.

Action Client

```
int main (int argc, char **argv)
{
  ros::init(argc, argv, "fibonacci_client");

  // kreiranje akcijskog klijenta
  // true uzrokuje da klijent pokreće vlastiti thread
  actionlib::SimpleActionClient<robot_test::FibonacciAction> ac("fibonacci");

  ROS_INFO("Waiting for action server to start.");
  // čekanje pokretanja akcijskog servera
  ac.waitForServer(); //will wait for infinite time

  ROS_INFO("Action server started, sending goal.");
  // pošalji cilj akciji
  robot_test::FibonacciGoal goal;
  goal.order = 20;
  ac.sendGoal(goal);

  // čekaj da se akcija returna
  ac.waitForResult(ros::Duration(30.0));

  robot_test::FibonacciResultConstPtr result = ac.getResult();
  ROS_INFO("Action finished:");
  for (int i : result->sequence)
  {
    ROS_INFO("%d", i);
  }

  // završi
  return 0;
}
```

Slika 22. ROS akcijski klijent

ROS akcijski klijent započinje s inicijalizacijom čvora pomoću funkcije `init` iz ROS biblioteke. Ova funkcija kao i obično prima argumente `argc` i `argv`, te naziv koji se predaje čvoru Kako bi klijent komunicirao s akcijskim poslužiteljem, potrebno je kreirati instancu klase `SimpleActionClient`. Ova klasa je definirana u akcijskoj biblioteci. Tako će se koristiti klasu `SimpleActionClient` i nazvat će se novu varijablu `ac`, skraćeno od `action client`. U uglastim zagradama treba se specificirati koju vrstu sučelja poruke se želi koristiti za komunikaciju s akcijskim poslužiteljem. To je sučelje koje je definirano u `FibonacciAction`. Koristi se sučelje `FibonacciAction` umjesto u konstruktoru objekta `SimpleActionClient`. Također mora se specificirati ime poslužitelja s kojim se želi komunicirati, to je `Fibonacci` poslužitelj. Također, u glavnoj funkciji, ispisuje se poruku u konzoli kako bi se obavijestilo korisnika da je čvor klijenta pokrenut i čeka da akcijski poslužitelj započne s radom. Poruka se

ispisuje pomoću "ROS_INFO" i ispisuje tekst "Waiting for action server to start." Prije nego što se zaista pošalje cilj poslužitelju, prvo je potrebno da je poslužitelj online i dostupan za primanje novih ciljeva. To se može učiniti pomoću funkcije `wait_for_server`. Zatim se koristi objekt `ac`, tj. `action_client`, i koristi se funkciju `wait_for_server`. Ova funkcija osigurava da C++ skripta zaustavi svoje izvršavanje i čeka dok je Fibonacci poslužitelj dostupan u ROS-u. Kada je to osigurano, može se stvoriti i poslati poruku poslužitelju. Koristi se "ROS_INFO" kako bi se obavjestilo korisnika da će se poslati cilj akcijskom poslužitelju. Ispisuje se poruku "Action server started, sending a goal. Ciljna poruka je definirana i dalje u sučelju `FibonacciAction`. Stvara se instancu `FibonacciGoal` koja se naziva `goal`. Ciljna poruka sadrži samo jednu varijablu, jednu vrijednost koja je cijeli broj i koja je nazvana `order` te je u ciljnoj poruci postavljena na 20. Sada naposljetku može se poslati ovu poruku poslužitelju pomoću funkcije `send_goal`. Dalje na objektu `ac`, tj. `action_client`, koristi se funkciju `sendGoal` i prosljeđuje joj se ciljnu poruku koja je upravo stvorena. Dakle, onu koja se zove `goal`. Sada klijent je poslao cilj akcijskom poslužitelju i zapravo će ovaj čvor biti slobodan za izvođenje bilo koje druge funkcionalnosti i bit će obaviješten od strane poslužitelja kada postoje ažuriranja s povratnim porukama. Kada je klijent poslao cilj poslužitelju ovaj će biti slobodan za izvođenje drugih funkcionalnosti i bit će obaviješten od strane poslužitelja kada, primjerice, postoje ažuriranja s povratnim porukama ili kada se izvođenje ovog poslužitelja završi s rezultatom. Međutim, budući da u ovoj jednostavnoj skripti ne mora se izvoditi nikakvu drugu funkcionalnost, čekati će se da akcijski poslužitelj završi s izvođenjem pomoću funkcije `waitForResult`. Na objektu akcijskog klijenta koristi se funkciju `waitForResult`. Ova funkcija zahtijeva maksimalno trajanje koje klijent treba čekati rezultat poslužitelja. U slučaju da se postavi čekanje na 30 sekundi pomoću `ros::Duration`, još jednom funkcija `waitForResult` zaustavlja izvršavanje skripte klijenta i blokira ga dok poslužitelj Fibonacci ne završi izvođenje i ne vrati poruku. Kada se to dogodi, tada klijent može nastaviti s daljnjom naredbom. Tako, budući da je u tom trenutku skripti poslužitelj već završio izvođenje, može se dohvatiti i poruku koja je izlaz poslužitelja. Da bi se to učinilo, koristi se funkciju `getResult` i dalje na objektu akcijskog klijenta i može se spremiti rezultat u drugu varijablu. Druga varijabla se naziva `result` i koristi tip `auto` kako bi C++ automatski mogao prepoznati tip varijable, zatim može se ispisati poruku u terminalu koja informira korisnika da je akcijski poslužitelj završio s izvođenjem. Tako `ros::info` ispisuje poruku "Action finished." Sada ispisuju se jedan po jedan svi elementi koji su u poruci. Budući da poruka sadrži niz, vektor svih brojeva Fibonacci niza, ispišu se svi elementi u petlji "for". Tako se stvara indeks koji prolazi kroz sve elemente u nizu i tako se dobije rezultat, zatim pristupi se nizu koji je u poruci i ispisuje se jedan po jedan sve poruke u konzoli pomoću naredbe "ROS_INFO", i samo se ispisuju brojevi koji pripadaju ovom nizu. S tim je ovaj čvor završen i može ga se izgraditi i pokrenuti. Dakle, kako bi ga se izgradilo, otvara se "CMakeLists.txt". Ovdje su već definirana

sučelja akcijske poruk. Sada se moramo kompajlirati C++ skriptu za akcijskog klijenta. Da bi se to učinilo, kopira se i zalijepi istu naredbu koja se koristila za poslužitelja i promijenimo se ime u `simple_cpp_action_client`. Kako bi se izgradio novi čvor, koristi se naredbu `catkin_make`. Nakon toga prvo se pokrene `roscore` pomoću naredbe `roscore`, a zatim se pokrene čvor akcijskog klijenta. Zatim se sourcamo workspace. Tj. sourca se datoteku "setup.bash" i pokrene se čvor klijenta s naredbom `rosrun` te se pokrene čvor `simple_cpp_action_client` . Tako, pri pokretanju, ovaj čvor obavještava da akcijski klijent započinje. Ali zatim se ništa ne događa jer još uvijek čeka da akcijski poslužitelj također započne s radom. To je zato što u kodu koristi se funkciju `waitForServer` a još nema Fibonacci poslužitelja, čvor klijenta je blokiran i čeka da se taj poslužitelj pojavi. Ako se pokrene poslužiteljski čvor i zatim u novom terminalu također se sourcamo workspace. Tj. sourca se datoteku `setup.bash` i pokrene poslužiteljski čvor s naredbom `rosrun` i `simple_cpp_action_server`. Čim se akcijski poslužitelj pokrene, klijent odmah prepoznaje poslužitelj Fibonacci među svim ostalim dostupnim čvorovima u ROS-u i konačno mu može poslati ciljnu poruku. Također počinje izračunavati niz Fibonaccijevih brojeva do reda koji je naznačen, a to je 20. U međuvremenu, akcijski klijent je blokiran, ne radi ništa i jednostavno čeka da poslužitelj završi s izvođenjem. Kada se to dogodi, akcijski klijent ispisuje rezultat poslužitelja s cijelim nizom Fibonacci brojeva do reda 20.

OpenCV

Računalni vid je disciplina koja se bavi razvojem računalnih sustava koji mogu analizirati i razumjeti slike i video materijale. OpenCV je open-source biblioteka koja pruža širok spektar alata i algoritama za razvoj aplikacija računalnog vida. OpenCV je posebno popularan u robotici, jer omogućava robotima da percipiraju svoju okolinu i donose odluke na temelju vizualnih podataka.

Postoji mnogo primjena OpenCV-a u robotici. Autonomni roboti mogu koristiti OpenCV za prepoznavanje prepreka i znakova te za sigurno kretanje kroz nepoznate prostorije ili ceste. Roboti manipulatori mogu koristiti OpenCV za prepoznavanje i praćenje objekata koje manipuliraju. Roboti koji interagiraju s ljudima mogu koristiti OpenCV za prepoznavanje lica i gesta. Roboti koji se koriste za nadzor i sigurnost mogu koristiti OpenCV za praćenje kretanja objekata u stvarnom vremenu.

Alexa

Alexa je virtualni asistent razvijen od strane Amazona koji pruža širok spektar funkcionalnosti, uključujući odgovaranje na pitanja, puštanje glazbe i upravljanje uređajima. Alexa se također može integrirati s robotima, što omogućava korisnicima da upravljaju robotima putem glasovnih naredbi.

Kada je Alexa integrirana s robotom, korisnici mogu koristiti glasovne naredbe za upravljanje robotom. Na primjer, korisnici mogu narediti robotu da se pomakne naprijed, zaustavi, snimi video ili paljenje svjetla. Korisnici mogu također postavljati pitanja o okolini robota, kao što su "Što vidi robot?" ili "Ima li prepreka ispred robota?"

Integracija Alexe s robotima ima mnogo prednosti. Glasovno upravljanje robotom je jednostavno i intuitivno, što omogućuje korisnicima da upravljaju robotima bez potrebe za korištenjem kompleksnih interfejsa. Osim toga, integracija Alexe omogućuje robotima da izvršavaju širi spektar zadataka, jer korisnici mogu koristiti glasovne naredbe za pristupanje širokom spektru Alexa vještina.

Kako se tehnologija razvija, očekuje se da će Alexa i robotika zajedno otvarati nove mogućnosti za inovacije i praktične primjene u različitim industrijama. Na primjer, Alexa se može koristiti za upravljanje robotima u logistici, proizvodnji, zdravstvu i drugim industrijama. Alexa se također može koristiti za upravljanje robotima u domovima, što može omogućiti starijim osobama i osobama s invaliditetom da žive samostalnije.

3. Implementacija rješenja

Aplikacijski Action Server

Ovaj alat će se upotrijebiti kako bi se kreirao akcijski server zadataka za robota.

```
class TaskServer(object):
    # kreiranje poruka koje se koriste za objavu povratnih informacija/rezultata
    result_ = ArduinobotTaskResult()
    arm_goal_ = []
    gripper_goal_ = []
    # x = 0
    # y = 0

    def __init__(self, name):
        # Konstruktor
        # funkcija koja inicijalizira klasu ArduinobotTaskAction i stvara
        # jednostavan akcijski poslužitelj iz actionlib biblioteke
        # Konstruktor koji se poziva kad se stvori nova instanca ove klase
        # u osnovi inicijalizira MoveIt! API koji će se koristiti kroz skriptu
        # inicijalizacija ROS sučelja s robotom putem MoveIt
        moveit_commander.roscpp_initialize(sys.argv)

        self.robot = moveit_commander.RobotCommander()

        self.scene = moveit_commander.PlanningSceneInterface()

        # stvaranje objekta komandne grupe za kretanje ruke robota
        self.arm_move_group_ = moveit_commander.MoveGroupCommander('arduinobot_arm')

        self.display_trajectory_publisher = rospy.Publisher(
            "/move_group/display_planned_path",
            moveit_msgs.msg.DisplayTrajectory,
            queue_size=20,
        )

        # stvaranje objekta komandne grupe za hvataljku
        self.gripper_move_group_ = moveit_commander.MoveGroupCommander('arduinobot_hand')

        self.scene = moveit_commander.PlanningSceneInterface()

        self.display_trajectory_publisher = rospy.Publisher(
            "/move_group/display_planned_path",
            moveit_msgs.msg.DisplayTrajectory,
            queue_size=20,
        )

        self.action_name_ = name
        self.as_ = actionlib.SimpleActionServer(self.action_name_, ArduinobotTaskAction, execute_cb=self.execute_cb, auto_start = False)
        self.as_.start()
```

Slika 23. ROS aplikacijski akcijski klijent 1.dio

```
def execute_cb(self, goal):

    success = True

    # početak izvođenja akcije
    # na temelju primljenog ID-a zadatka, šalje se različit cilj
    # robotu
    if goal.task_number == 0:
        self.arm_goal_ = [0.0,0.0,0.0]
        self.gripper_goal_ = [-0.7, 0.7]
        self.arm_move_group_.go(self.arm_goal_, wait=True)
        self.gripper_move_group_.go(self.gripper_goal_, wait=True)
        self.arm_move_group_.stop()
        self.gripper_move_group_.stop()
    elif goal.task_number == 1:
        # We can get the name of the reference frame for this robot:
        planning_frame = self.arm_move_group_.get_planning_frame()
        print("============ Planning frame: %s" % planning_frame)

        # We can also print the name of the end-effector link for this group:
        eef_link = self.arm_move_group_.get_end_effector_link()
        print("============ End effector link: %s" % eef_link)

        # We can get a list of all the groups in the robot:
        group_names = self.robot.get_group_names()
        print("============ Available Planning Groups:", self.robot.get_group_names())

        # Sometimes for debugging it is useful to print the entire state of the
        # robot:
        print("============ Printing robot state")
        print(self.robot.get_current_state())
        print("")
        x = rospy.get_param("x")
        y = rospy.get_param("y")
        print("x: ", x)
        print("y: ", y)
        waypoints = []
```

Slika 24. ROS aplikacijski akcijski klijent 2.dio

```
        wpose = self.arm_move_group_.get_current_pose().pose
        if x < 0:
            wpose.position.x = x + 0.05 # First move up (x)
        else:
            wpose.position.x = x - 0.05 # First move up (x)
        if y < 0:
            wpose.position.y = y + 0.25 # First move up (y)
        else:
            wpose.position.y = y - 0.25 # First move up (y)
        wpose.position.z = 2.35 - y # Second move forward/backwards in (z)
        if(wpose.position.z > 0.7):
            wpose.position.z = 0.6

        waypoints.append(copy.deepcopy(wpose))

        # Želimo da se kartezijanska putanja interpolira s razlučivošću od 1 cm
        # stoga ćemo postaviti eef_step na 0.01 pri izračunu kartezijanske putanje.
        # Preskočit ćemo prag za skakanje postavljanjem ga na 0.0
        (plan) = self.arm_move_group_.compute_cartesian_path(
            waypoints, 0.01, 0.0  # waypoints to follow  # eef_step
        )  # jump_threshold
        display_trajectory = moveit_msgs.msg.DisplayTrajectory()
        display_trajectory.trajectory_start = self.robot.get_current_state()
        display_trajectory.trajectory.append(plan)
        # Publish
        self.display_trajectory_publisher.publish(display_trajectory)
        self.arm_move_group_.execute(plan, wait=True)
        self.gripper_goal_ = [-0.12, 0.12]
        self.gripper_move_group_.go(self.gripper_goal_, wait=True)
        self.gripper_move_group_.stop()
        self.arm_goal_ = [0.0,0.0,0.0]
        self.gripper_goal_ = [-0.7, 0.7]
        self.arm_move_group_.go(self.arm_goal_, wait=True)
        self.gripper_move_group_.go(self.gripper_goal_, wait=True)
    elif goal.task_number == 2:
        self.arm_goal_ = [-1.57,0.0,-1.0]
        self.gripper_goal_ = [0.0, 0.0]
        self.arm_move_group_.go(self.arm_goal_, wait=True)
        self.gripper_move_group_.go(self.gripper_goal_, wait=True)
        self.arm_move_group_.stop()
        self.gripper_move_group_.stop()
    else:
        rospy.logerr('Invalid goal')
        return
```

Slika 24. ROS aplikacijski akcijski klijent 3.dio

```
        # Šalje cilj MoveIt API-ju
        # self.grupa_kretanja_ruke_.go(self.cilj_ruke_, wait=True)
        # self.grupa_kretanja_hvataljke_.go(self.cilj_stisak_, wait=True)

        # Pazite da nema preostalog kretanja
        self.arm_move_group_.stop()
        self.gripper_move_group_.stop()

        # provjerava je li klijent zatražio prekid
        if self.as_.is_preempt_requested():
            rospy.loginfo('%s: Preempted' % self.action_name_)
            self.as_.set_preempted()
            success = False

        # provjerava je li zahtjev za ciljem izvršen ispravno
        if success:
            self.result_.success = True
            rospy.loginfo('%s: Succeeded' % self.action_name_)
            self.as_.set_succeeded(self.result_)


if __name__ == '__main__':

    # Inicijalizacija ROS čvora nazvanog task_server
    rospy.init_node('task_server')

    server = TaskServer(rospy.get_name())

    # održava čvor aktivnim
    rospy.spin()
```

Slika 25. ROS aplikacijski akcijski klijent 4.dio

Potrebno je stvoriti akcijski poslužitelj koji je povezan s API-jem za moveit i pruža popis zadataka koje robot može izvršiti, a koje drugi čvor može pokrenuti koristeći sučelje akcijskog poslužitelja. Isti će poslužitelj zadataka biti sučelje za Alexu. To će koristiti kako bi pokrenulo izvršavanje zadatka i pokretalo robota. S obzirom na to da ovaj poslužitelj zadataka sadrži potpuno novu funkcionalnost, dodaje ga se novom paketu. Stoga unutar terminala, potrebno je navigirati workspace projekta, zatim mapu source gdje se nalaze paketi robota. Na toj lokaciji stvara se novi paket koristeći naredbu "catkin_create_pkg" te ga se naziva "robot_remote". Budući da će ovaj paket koristiti Python skripte, dodaju se biblioteke ROS i SPI. Također, budući da će paket koristiti definiciju nekih osnovnih ROS poruka, dodaje se ovisnost i o standardnim porukama. Sada je potrebno ponovno izgraditi workspace koristeći

naredbu "catkin_make" kako bi prepoznao novi paket. Prva stvar koja se treba definirati, budući da će se razvijati akcijski poslužitelj, jest definirati sučelje poruke koje će se koristiti za komunikaciju. Unutar paketa "robot_remote", stvara se mapu "action", a unutar nje se stvara novu datoteku naziva "RobotTask.action". Unutar ovog sučelja treba se definirati tri poruke. Prva je poruka cilja ("goal"), koja sadrži samo cijeli broj koji označava ID zadatka koji poslužitelj treba izvršiti među svim dostupnim zadacima. Dakle, to je cijeli broj, naziva ga se "task_number". Zatim se može definirati poruku rezultata ("result"). Poruka rezultata će sadržavati samo varijablu tipa "boolean" koja označava je li akcija uspješno završena ili ne. Naziva se `success`. Isto tako, definira se poruka povratne informacije ("feedback"). Poruka povratnih informacija može sadržavati postotak izvršenja zadatka. Dakle, definira se varijablu `percentage` koja je tipa cijelog broja ("integer"). Nakon što je sučelje komunikacije s akcijskim poslužiteljem izgrađeni, treba se stvoriti stvarnu logiku poslužitelja. To će se implementirati unutar nove C++ skripte. Unutar mape "src", stvara se novu datoteku nazvanu "TaskServer.cpp". Kao i obično, započinje se uključivanjem odgovarajućih biblioteka. Uključuje se "ros/ros.h" i definiciju "SimpleActionServer" iz biblioteke "actionlib" za akcijske poslužitelje. Također, treba se uključiti definiciju sučelja poruka koje su upravo stvorene. Tako da se uključi "robot_remote/RobotTask.h". Također, budući da će se koristiti MOVE IT API u C++ skripti, potrebno je uključiti MOVE IT API biblioteku za planiranje trajektorija i upravljanje robotom. Također, budući da će se koristiti definicije nekih osnovnih poruka iz ROS-a, potrebno je uključiti odgovarajuće biblioteke. Uključuje se "moveit/move_group_interface/move_group_interface.h" za MOVE IT API i "std_msgs/Bool.h" za standardne poruke. Nakon što su uključene potrebne biblioteke, sada se implementira svu logiku akcijskog poslužitelja unutar nove C++ skripte. Unutar klase `TaskServer` definira se privatne atribute koji će se koristiti za komunikaciju s ROS-om i MOVE IT API-jem, kao i za praćenje stanja izvršavanja. Unutar ove klase definira se konstruktor koji će se koristiti za inicijalizaciju atributa i postavljanje akcijskog poslužitelja. Kroz konstruktor se također podesi način rada poslužitelja i njegova funkcija za obradu poruka. Isto tako, unutar klase `TaskServer` definira se funkciju `executeCallback` koja će se pozivati svaki put kad poslužitelj primi novu poruku. Ova funkcija će planirati i izvršiti zadani zadatak koristeći MOVE IT API. Nakon izvođenja zadatka, poslužitelj će poslati povratnu informaciju i rezultat izvršavanja. Osim toga, potrebna je i glavna funkcija (main) koja će pokrenuti ROS čvor i startati akcijski poslužitelj. Ova funkcija će također održavati akcijski poslužitelj dostupnim za primanje poruka. Nakon što je implementirana C++ skriptu, potrebno je pravilno konfigurirati "CMakeLists.txt" datoteku za prepoznavanje novih datoteka i biblioteka. Također, potrebno je izvršiti izgradnju koda koristeći "catkin_make" kako bi se stvorila izvršna datoteka. Nakon je sve izgrađeno, potrebno je pokrenuti simulaciju robota u Gazebo okruženju i pokrenuti kontrolne sustave.

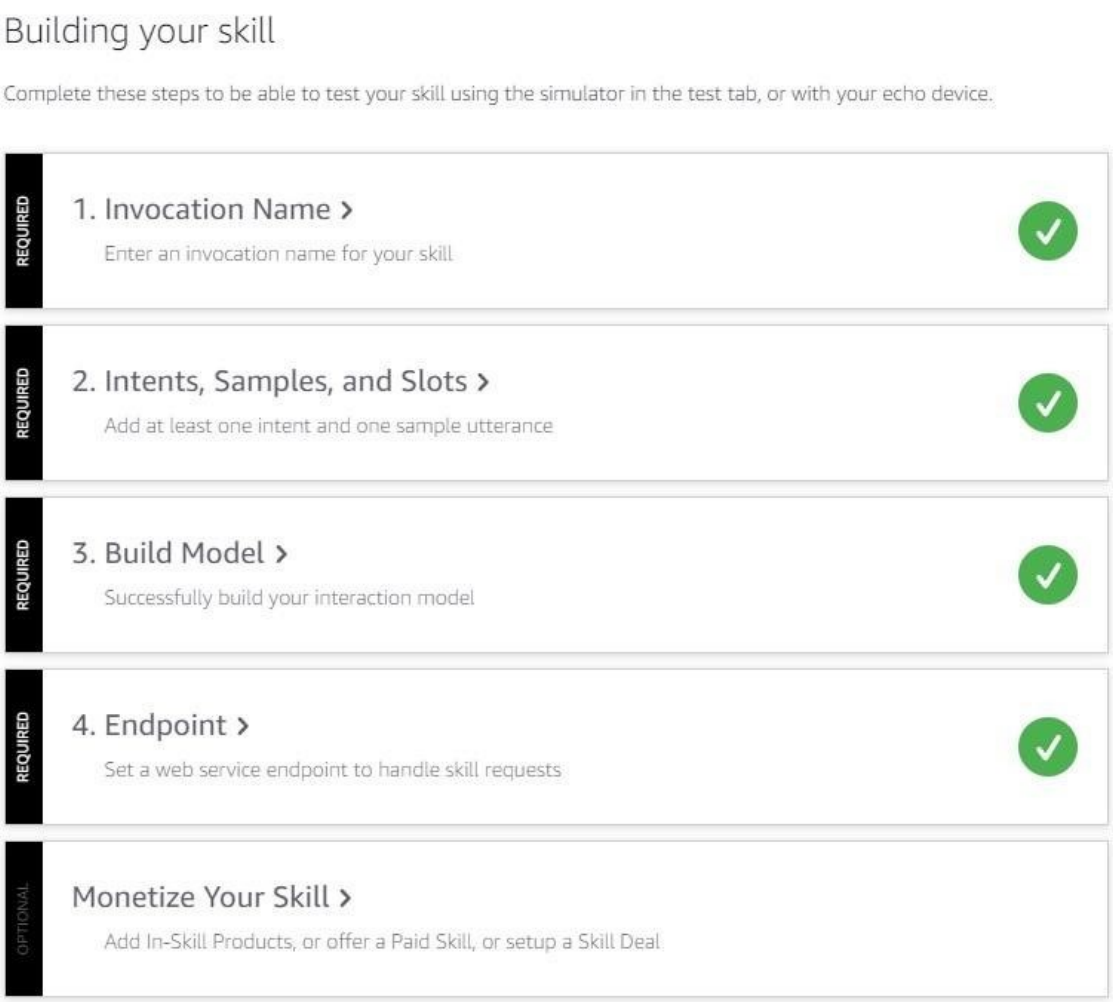
Nakon što je sve pokrenuto, može se koristiti alate za akcijske poslužitelje kako bi se slalo zadatke robotu i pratilo njihovu izvedbu.

Alexa integracija

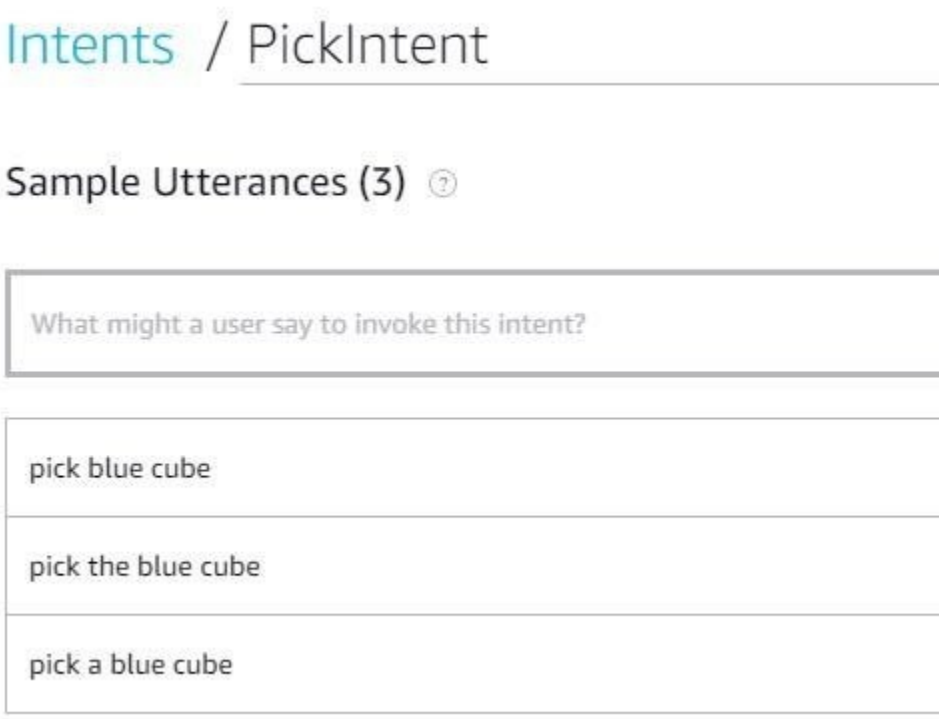
Ako smo do sada razvijali sve funkcionalnosti robota lokalno, na našem računalu, sada je vrijeme da povežemo robota s internetom i oslobodimo njegov potencijal. Ovo je naš temeljni korak, jer potencijalno omogućava neograničen broj primjena. To su prava inovacija, prava revolucija u području robotike. Jer robot koji je povezan s internetom više nije ograničen snagom računala i računalnim resursima koje ima lokalno, već potencijalno može biti povezan s oblakom i tako pristupiti našem nizu resursa i aplikacija koji poboljšavaju sve sposobnosti robota. Među ovim aplikacijama, jedna od najzanimljivijih za robotiku je ona pomoću glasovnih asistenata. Ako razmislimo o tvornicama i radionicama gdje smo češće navikli vidjeti robote, oni se često kontroliraju i pomiču pomoću velikih i kompliciranih džojstika, posebno tijekom učenja pokreta. Ovaj način kretanja robota prikladan je za industrijsko okruženje gdje prašina, voda i stroge sigurnosne mjere zahtijevaju da se robot kreće pomoću ovakvih uređaja. Međutim, svjedočimo prijelazu s industrijske robotike na kućnu robotiku, i u ovom scenariju, roboti prelaze iz sigurnog i zaštićenog prostora rezerviranog samo za njihovo angažiranje u dijeljenje prostora s ljudima. Kao što možete zamisliti, to izaziva nove aspekte interakcije između robota i ljudi. To je dovelo do nastanka novih tema kao što je interakcija između ljudi i robota ili HRT. U ovom novom scenariju gdje roboti interagiraju s ljudima, što je bolji alat, ako ne glas, koji nam omogućuje komunikaciju s drugim ljudima od našeg pojavljivanja na Zemlji. Također, kako bismo komunicirali s robotima putem glasa. Možemo komunicirati naše namjere, ili možemo zatražiti izvršenje nekih zadataka i također dobiti povratne informacije od robota. To povećava našu svijest o pokretima koje će robot izvršiti. Među svim dostupnim glasovnim asistentima, odlučio sam koristiti Amazon Alexu jer nudi mogućnost dodavanja novih funkcionalnosti i ponašanja, nazvanih vještine asistenta. Također, jer Amazon za razvoj takvih funkcionalnosti nudi vrlo intuitivno API sučelje i grafičko korisničko sučelje za testiranje ovih funkcionalnosti. Također, ne trebate imati fizičkog glasovnog asistenta u svojoj kući kako biste izveli ovu akciju. U stvarnosti, nećete trebati kupiti glasovnog asistenta, već ćemo pristupiti razvojnoj web stranici Amazon Alexe i tamo ćemo kreirati nove vještine, nova ponašanja asistenta, i koristiti ih kako bismo pokrenuli izvršenje zadatka robota putem glasa. Međutim, prije nego što to učinimo, prije nego što zaista možemo slati naredbe robotu, prvo ćemo morati povezati robota s internetom kako bi ga glasovni asistent mogao doseći. Da bismo to učinili, koristit ćemo vrlo jednostavan i koristan alat nazvan "ngrok". Ovaj alat nam omogućava otvaranje vrata i HTTP porta našeg računala prema internetu. Također će

pružiti javnu adresu na koju se može dosegnuti naše računalo, odnosno gdje je naše računalo dostupno. Na ovaj način, računalo, pa tako i glasovni asistent Amazon Alexa koji je povezan s internetom, moći će komunicirati s našim računalom, odnosno s robotom.

Alexa Skills



Slika 26. Alexa skills



Slika 27. Alexa intent

Alexa skills su funkcionalnosti koje omogućuju Alexi da obavlja različite zadatke i pruža informacije na temelju glasovnih naredbi korisnika. To su kao digitalni alati koji proširuju sposobnosti Alexe daleko izvan osnovnih funkcija. Vještine mogu pružiti informacije, reproducirati medij, kontrolirati pametne uređaje u domaćinstvu, pružiti zabavne igre i još mnogo toga, sve to putem jednostavnih razgovora s Alexom. Proces stvaranja Alexa vještina zahtijeva nekoliko koraka. Prvi korak je odabir koncepta vještine. To uključuje definiranje svrhe vještine i ciljne publike. Nakon toga, developer treba osmisliti korisničko iskustvo, odnosno kako će korisnici komunicirati s vještinom putem glasovnih naredbi. Slijedi korak programiranja vještine koristeći Alexa Skills Kit (ASK), alat za razvoj vještina koji pruža Amazon. ASK omogućuje developeru definiranje vokabulara, naredbi i odgovora koje će vještina razumjeti i izvoditi.

Ngrok

Ovaj alat djeluje kao poslužitelj koji se izvršava na lokalnom računalu te će primati JSON poruke od Alexa skills-a, razumjeti i obraditi zahtjev, potom će odgovoriti Alexa skills-u drugom JSON porukom. Link koji ngrok generira predaje se Alexi kako bi se

ostvarila veza između robot servisa i Alexe. Kako pokrenuti ngrok i gdje staviti link koji generira biti će objašnjeno pod Simulacija robota na stranici 55.

Računalni vid

Računalni vid, kao grana računalne znanosti, postao je ključna tehnologija koja ima široku primjenu u mnogim industrijama. Jedno od područja gdje je računalni vid postigao izvanredne rezultate je robotika. Integracija računalnog vida u robotičke sustave omogućila je stvaranje inteligentnih i autonomnih robota koji su sposobni interagirati s okolinom i obavljati složene zadatke. Računalni vid u robotici omogućuje robotima da percipiraju svoje okruženje, identificiraju objekte, prepoznaju obrasce, te donose informirane odluke na temelju prikupljenih vizualnih podataka. To omogućuje robotima da se prilagode različitim situacijama i radnim uvjetima, što je ključno u različitim primjenama, od industrijske automatizacije do usluga u kućanstvu. Primjena računalnog vida u robotici je široka i raznovrsna. U industrijskoj automatizaciji, roboti opremljeni kamerama mogu identificirati i praćenje objekte na proizvodnoj liniji, što povećava efikasnost i preciznost proizvodnog procesa. U skladištima, roboti se mogu koristiti za autonomno sortiranje i pretraživanje proizvoda koristeći vizualno prepoznavanje. U robotskoj kirurgiji, računalni vid omogućuje kirurzima precizno upravljanje robotima tijekom operacija, minimizirajući rizik i povećavajući točnost. U uslugama dostave, autonomni roboti koriste računalni vid za navigaciju i izbjegavanje prepreka kako bi sigurno dostavili pakete. Još jedna primjena računalnog vida u robotici je u razvoju autonomnih vozila. Vizualni senzori, kao što su kamere i lidari, omogućuju vozilima da prepoznaju ceste, prometne znakove, semafore i pješake. Ova tehnologija je ključna za razvoj sigurnih i samostalnih vozila koja mogu komunicirati s okolinom i donositi brze i precizne odluke na cesti. Uz sve navedene primjene, treba istaknuti da računalni vid u robotici donosi i izazove. Promjenjivi uvjeti osvjetljenja, različite perspektive i složeni obrasci mogu predstavljati poteškoće u prepoznavanju i interpretaciji podataka. Osim toga, potrebno je razviti algoritme za brzu i efikasnu obradu slika kako bi roboti mogli djelovati u stvarnom vremenu. U zaključku, računalni vid je postao neizostavna komponenta u robotici koja omogućuje robotima da percipiraju i razumiju svoje okruženje. Njegova primjena seže od industrije do medicine, od autonomnih vozila do robotske kirurgije. Kroz integraciju računalnog vida, robotika ostvaruje značajan napredak prema stvaranju inteligentnih, prilagodljivih i autonomnih robota koji će oblikovati budućnost tehnološkog napretka.

OpenCV

Računalni vid je postao neodvojiva komponenta današnjeg tehnološkog okruženja, omogućujući računalima da interpretiraju i analiziraju vizualne informacije. U ovom kontekstu, OpenCV (Open Source Computer Vision Library) izdvaja se kao izuzetno moćan alat koji omogućuje razvoj raznovrsnih aplikacija u području računalnog vida. OpenCV je otvorena izvornog koda računalnog vida i strojnog učenja, stvorena kako bi omogućila razvoj visoko optimiziranih aplikacija za obradu slika i video sadržaja. Svojim bogatim skupom algoritama, funkcija i alata, OpenCV pruža korisnicima raznolike mogućnosti, od jednostavnih manipulacija slika do složenih analiza obrazaca i prepoznavanja objekata. Jedna od ključnih prednosti OpenCV-a je njegova široka podrška za različite platforme i programski jezici. OpenCV se može koristiti na raznim operativnim sustavima poput Windows-a, Linux-a, MacOS-a i drugih, uz podršku za programiranje u C++, Python-u, Java-i i drugim jezicima. To čini OpenCV pristupačnim i prilagodljivim za različite razvojne timove i projekte. Primjene OpenCV-a su bezgranične. U računalnom vidu, koristi se za osnovne operacije poput filtriranja, segmentacije i detekcije rubova, kao i za složenije zadatke poput prepoznavanja lica, objekata i gestikulacija. Medicinske primjene obuhvaćaju analizu medicinskih slika, dijagnostiku i planiranje tretmana. Industrija pametnih vozila koristi OpenCV za detekciju znakova, pješaka i drugih vozila kako bi omogućila autonomno upravljanje. Nadalje, OpenCV ima značajnu ulogu u robotskoj percepciji. Roboti opremljeni kamerama mogu koristiti OpenCV za navigaciju, prepoznavanje okoline i interakciju s objektima. Ovo je ključno za razvoj inteligentnih i autonomnih robota koji mogu obavljati različite zadatke u raznim okruženjima. Važno je istaknuti da OpenCV podržava i strojno učenje. Ovo omogućuje razvoj dubokih neuronskih mreža i modela koji omogućuju složene analize i prepoznavanje uzoraka. Kroz ovu integraciju, OpenCV postaje još snažniji alat za rješavanje složenih problema u računalnom vidu. Unatoč svojim brojnim prednostima, OpenCV nije bez izazova. Složenost algoritama i potreba za prilagodbom za specifične primjene mogu zahtijevati dublje razumijevanje računalnog vida i programiranja. Također, brzina izvršavanja algoritama može biti izazov u aplikacijama koje zahtijevaju obradu u stvarnom vremenu. U zaključku, OpenCV predstavlja ključni alat u razvoju računalnog vida i njegove primjene. Njegova fleksibilnost, bogatstvo funkcionalnosti i podrška za različite platforme čine ga nezaobilaznim za razvojne timove i istraživače u području računalnog vida. Kroz kontinuiranu podršku zajednice i napredak u strojnom učenju, OpenCV ostaje središnji igrač u stvaranju inovativnih tehnoloških rješenja koja koriste moć vizualnih podataka.

Implementacija

Integracija OpenCV-a (Open Source Computer Vision Library) s ROS-om (Robot Operating System) predstavlja snažan spoj tehnologija koji otvara vrata za raznolike primjene u robotici, omogućujući robotima da interpretiraju i reagiraju na vizualne informacije iz svog okruženja. Ova kombinacija donosi mogućnosti za napredne zadatke percepcije, navigacije i interakcije s objektima, čime se podiže razina autonomnosti i inteligencije robota. ROS, popularna platforma za razvoj robotskih aplikacija, pruža okruženje za komunikaciju između različitih komponenti robota, poput senzora, aktuatora i algoritama. OpenCV se ističe kao ključni alat za obradu vizualnih podataka, omogućujući robotima da razumiju svijet oko sebe. Kombinacija ovih tehnologija otvara vrata za širok spektar aplikacija u robotici. Jedna od ključnih primjena integracije OpenCV-a s ROS-om je vizualna percepcija. Roboti opremljeni kamerama mogu koristiti OpenCV za prepoznavanje objekata, prepreka, ljudi i drugih entiteta u svom okruženju. Ovo je ključno za navigaciju robota, jer im omogućuje da planiraju putanje i izbjegavaju kolizije. Na primjer, robot može koristiti detekciju prepreka pomoću OpenCV-a kako bi se izbjegla sudaranja tijekom kretanja. Također, OpenCV se koristi za analizu slika i video sadržaja u stvarnom vremenu. Roboti mogu koristiti OpenCV za prepoznavanje lica, gestikulacija i izražavanje emocionalnih stanja ljudi. Ovo otvara mogućnosti za interakciju robota s ljudima na prirodniji način, pružajući temelj za razvoj robotske asistencije u domaćinstvima i industriji. Integracija OpenCV-a i ROS-a omogućuje i razvoj autonomnih vozila. Vizualna percepcija je ključna za autonomnu vožnju, jer vozila moraju prepoznati znakove, pješake, bicikliste i druge sudionike u prometu. OpenCV se koristi za detekciju i praćenje objekata na cesti te za donošenje brzih i preciznih odluka tijekom vožnje. Kroz podršku za strojno učenje, OpenCV se integrira i u razvoj dubokih neuronskih mreža i modela za prepoznavanje oblika i uzoraka. Ovo je korisno za zadatke poput prepoznavanja složenih objekata ili analize medicinskih slika. Kombinacija ROS-a i OpenCV-a omogućuje istraživačima i inženjerima da brzo testiraju i primijene nove algoritme za strojno učenje u robotskim aplikacijama. Unatoč prednostima, integracija OpenCV-a i ROS-a također nosi izazove. Složenost algoritama i zahtjevi za brzim izvođenjem mogu zahtijevati optimizaciju kako bi se osigurala izvedivost u stvarnom vremenu. Također, postavljanje i konfiguracija ROS čvorova za komunikaciju s OpenCV-om zahtijeva razumijevanje ROS-ove arhitekture. U zaključku, integracija OpenCV-a s ROS-om predstavlja snažan alat za razvoj robotskih aplikacija koje koriste vizualne podatke. Ova kombinacija otvara vrata za napredne zadatke percepcije, prepoznavanja, navigacije i interakcije. Kroz primjene u autonomnoj vožnji, robotskoj asistenciji, industriji i mnogim drugim područjima, ROS i OpenCV zajedno čine temelj za inovaciju u robotici i tehnološkom napretku.

```
bridge = CvBridge()

def callback(img_msg):
    try:
        cv_image = bridge.imgmsg_to_cv2(img_msg, desired_encoding="bgr8")
        height, width = cv_image.shape

        hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)

        mask = cv2.inRange(hsv,lower_color_value, upper_color_value)

        m = cv2.moments(mask, False)

        try:
            calc_y, calc_x = m['m10']/m['m00'], m['m01']/m['m00']
            calc_x = SCREEN_WIDTH - calc_x
            calc_y = SCREEN_HEIGHT - calc_y
            calc_x = (X_UNIT * calc_x) - ARM_MAX_MOVEMENT_X
            calc_y = (Y_UNIT * calc_y) + ARM_MAX_MOVEMENT_Y

            rospy.set_param('x', calc_x)
            rospy.set_param('y', calc_y)

        except ZeroDivisionError:
            calc_x, calc_y = height/2, width/2


    except CvBridgeError as e:
        rospy.logerr("CvBridge Error: {0}".format(e))


if __name__ == '__main__':

    rospy.init_node('image_listener', anonymous=True)

    rospy.Subscriber("/image_raw", Image, callback)

    r = rospy.Rate(1)

    rospy.spin()
```

Slika 27. Kod za detektiranje kockica

Integracija OpenCV-a i ROS-a omogućuje razvoj inovativnih robotskih aplikacija, a jedna od primjena je praćenje i lociranje obojenih kockica pomoću robotske ruke i kamere. Ova tehnika omogućuje robotu da prepozna crvene, zelene ili plave kockice, te da ih locira u prostoru koristeći proračun centroida kockice. Prvo, potrebno je uspostaviti vezu između ROS-a i OpenCV-a pomoću CvBridge-a. Ovaj most omogućuje konverziju između ROS-ovih poruka slike i OpenCV formata. Nakon toga,

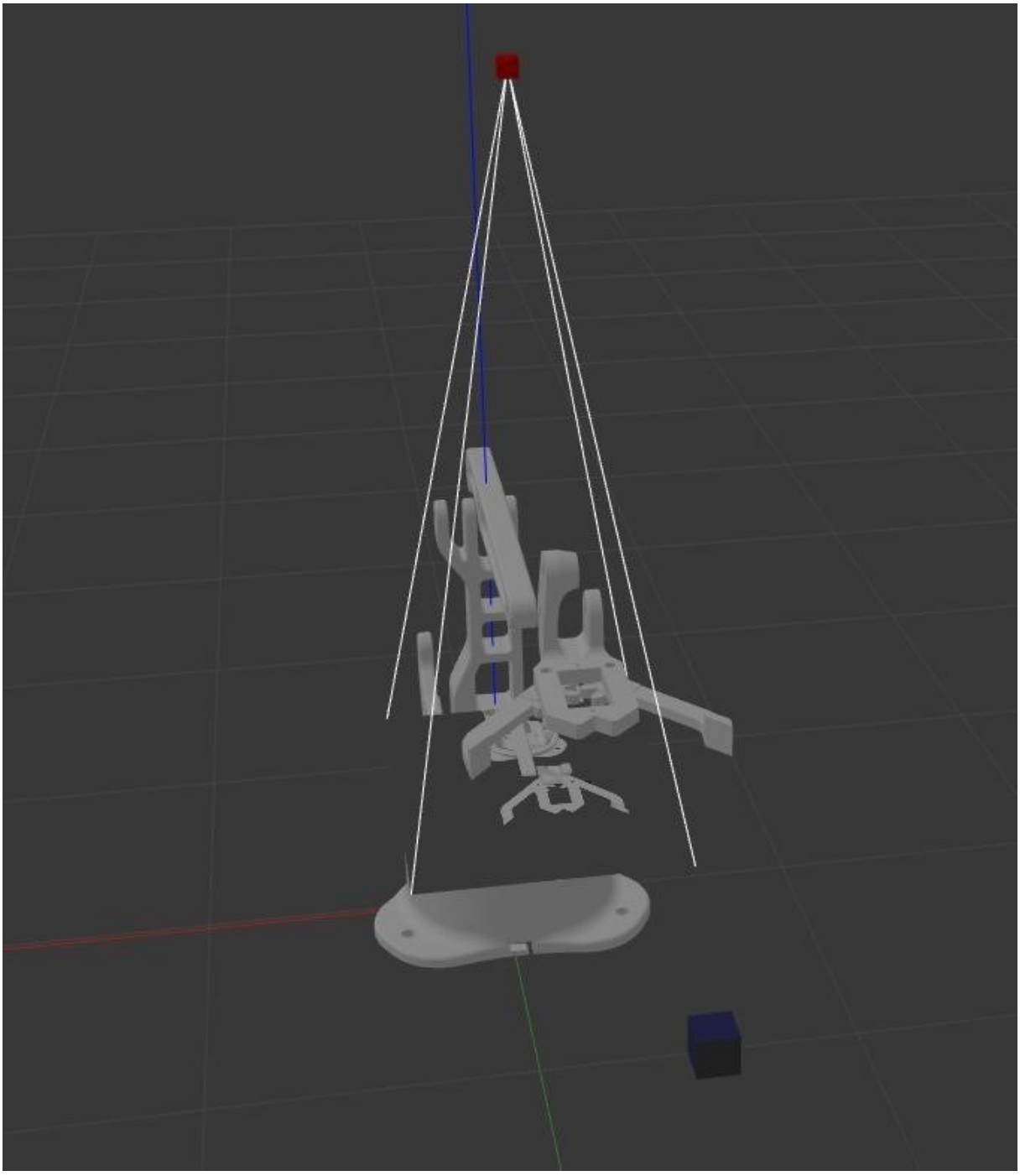
definira se funkcija callback koja se aktivira svaki put kad se primi slika od kamere. Slika se konvertira u OpenCV format i zatim se pretvara u HSV (Hue, Saturation, Value) prostor boja. Ovdje se definiraju pragovi za detekciju određene boje - u ovom slučaju, plave. Izrađuje se maska koja prikazuje regije na slici koje sadrže plavu boju. Nakon toga, koriste se momenti maskirane slike da bi se izračunao centroid kockice. Centroid je težište površine kockice, odnosno središnja točka. Ova točka se dalje koristi za izračunavanje stvarne lokacije kockice u prostoru. Da bi se izračunale koordinate kockice u stvarnom svijetu, koriste se matematičke transformacije i kalibracije. Ako je položaj kamere i robotske ruke fiksan, može se koristiti pristup kao što je opisan u kodu. Koristeći matematičke izračune, centroid kockice se pretvara u stvarne koordinate u prostoru. To omogućuje precizno lociranje kockice bez potrebe za složenim senzorima. U kodu, vidimo da su x i y koordinate kockice postavljene kao parametri ROS-a. To omogućuje drugim čvorovima da pristupe ovim vrijednostima i koriste ih za daljnje upravljanje robotom ili izvođenje drugih zadataka. Na primjer, robot bi mogao koristiti dobivene koordinate kako bi pravilno pozicionirao hvataljku i uhvatio kockicu. Ova metoda ima mnoge prednosti. Korištenjem jeftine kamere i jednostavne robotske ruke, moguće je implementirati ovu funkcionalnost u stvarnom svijetu. Također, pristup kalibracije i izračuna centroida omogućuje precizno, pouzdano i brzo lociranje objekata. U zaključku, korištenje OpenCV-a s ROS-om za praćenje i lociranje obojenih kockica pruža napredne mogućnosti percepcije i interakcije robota s okruženjem. Ova tehnika pokazuje kako se jednostavni alati i pristupi mogu iskoristiti za razvoj sofisticiranih robotskih funkcionalnosti. Kroz ovu integraciju, moguće je postići autonomnost i inteligenciju u robotskim aplikacijama, što otvara vrata za napredak u robotici i automatizaciji.

Za moći koristiti ovu skriptu u simulaciji kreira se URDF file kamere koju se stavi u prostor gazebo simulacije, te ju se pozicionira iznad robotse ruke.

> FIPU files > 5. i 6. semestar > Završni rad > Pick-And-Place-Practice > src >

```
<?xml version='1.0'?>
<sdf version='1.7'>
  <model name='camera'>
    <link name='link'>
      <inertial>
        <mass>0.1</mass>
        <inertia>
          <ixx>0.000166667</ixx>
          <iyy>0.000166667</iyy>
          <izz>0.000166667</izz>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyz>0</iyz>
        </inertia>
        <pose>0 0 0 0 -0 0</pose>
      </inertial>
      <sensor name='camera' type='camera'>
        <camera>
          <horizontal_fov>1.047</horizontal_fov>
          <image>
            <width>320</width>
            <height>240</height>
          </image>
          <clip>
            <near>0.1</near>
            <far>1000</far>
          </clip>
        </camera>
        <always_on>1</always_on>
        <update_rate>30</update_rate>
        <visualize>1</visualize>
      </sensor>
```

Slika 28. URDF file kamere

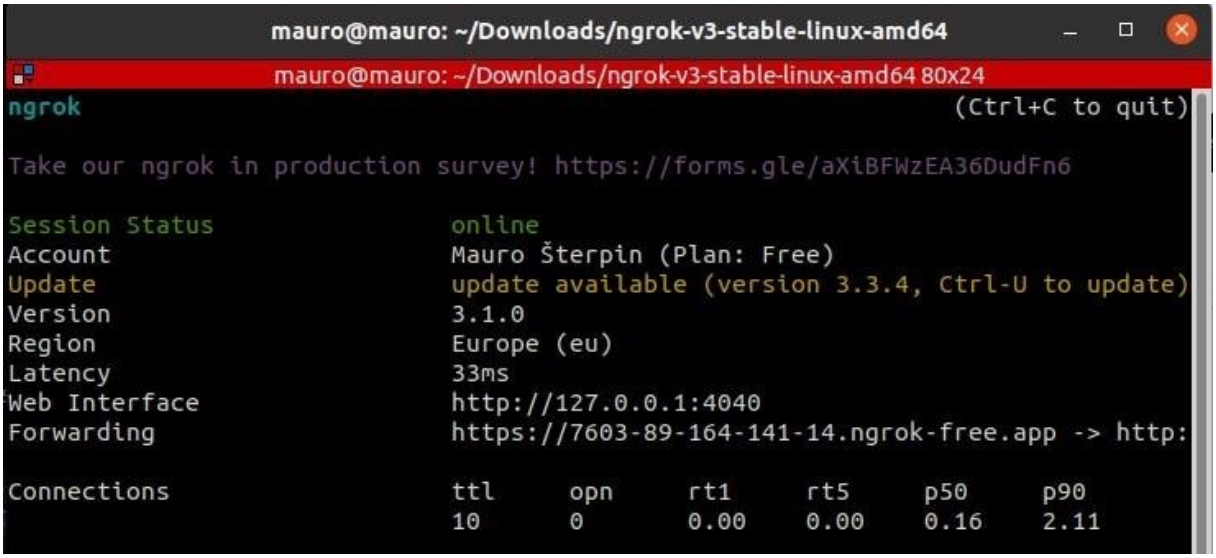


Slika 29. Gazebo simulacija s kamerom iznad robotske ruke

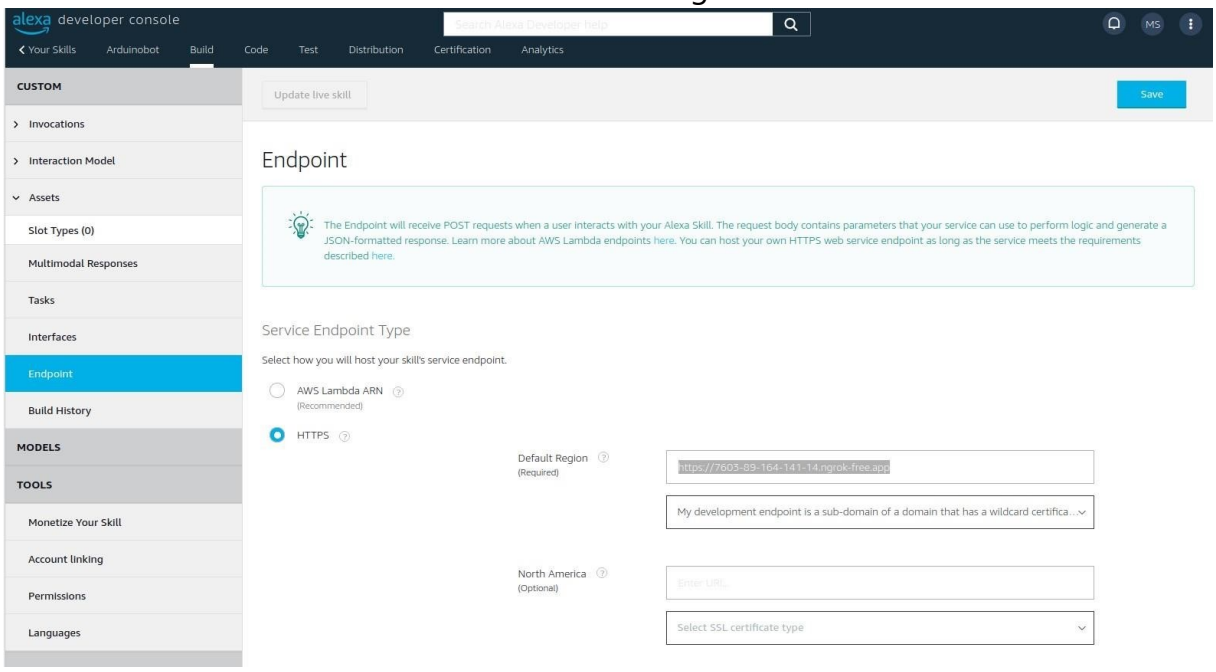
4. Rezultati Simulacije Robotske Ruke

Nakon što se napravi sve module robotske ruke vrijeme je za pokrenuti simulaciju. Za pokrenuti simulaciju potrebno je pokrenuti ngrok s komandom ./ngrok 5000. Ngrok

pruža link s kojim se može vanjsko računalo povezati s lokalnim računalom na portu 5000. Forward link se daje Alexi kako bi Alexa servis znao s kojim računalom treba komunicirati.



Slika 31. Pokrenuti ngrok



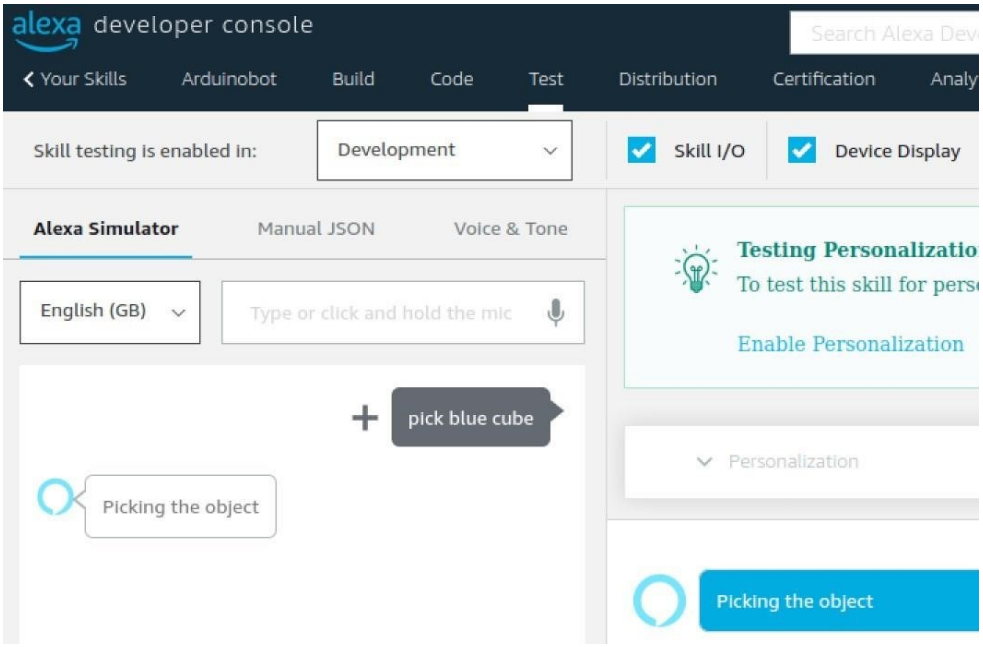
Slika 32. Dodavanje linka na alexu

Zatim u drugom terminalu pokreče se jednostavnu skriptu koja služi kao launch file. Ona pokreče sve module, stoga ih se ne treba sve manualno pokretati.

```
<launch>

  <!-- Launch the Gazebo simulation of the robot arm -->
  <arg name="model" default="$(find robot_description)/urdf/robot.urdf.xacro"/>

  <include file="$(find robot_description)/launch/gazebo.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Launch the controllers -->
  <include file="$(find robot_controller)/launch/controller.launch">
    <arg name="is_sim" value="true"/>
  </include>

  <!-- Launch moveit move_group -->
  <include file="$(find robot_moveit)/launch/move_group.launch"/>

  <!-- Launch moveit rviz gui -->
  <include file="$(find robot_moveit)/launch/moveit_rviz.launch">
    <arg name="rviz_config" value="$(find robot_moveit)/launch/moveit.rviz"/>
  </include>

  <!-- Launch the interface with alexa -->
  <include file="$(find robot_remote)/launch/remote_interface.launch"/>

  <!-- Launch object detection -->
  <include file="$(find vision)/launch/publisher.launch"/>

</launch>
```

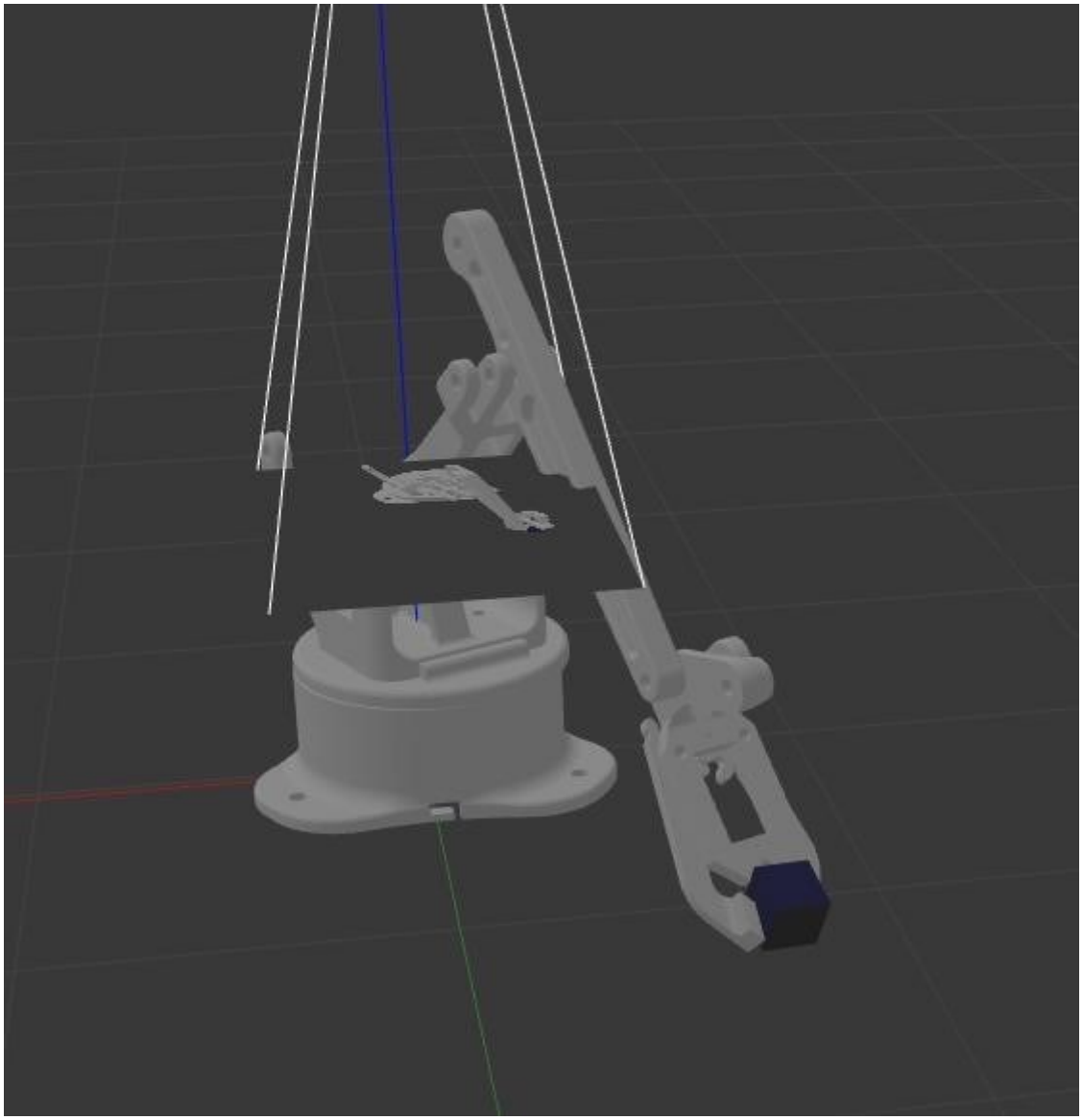
Slika 33. Launch file

Putem Alexa sučelja robotu se šalje naredbe.



Slika 34. Slanje komande robotu putem Alexa sučelja.

Zatim se u gazebo simulaciji može vidjeti robota u akciji.



Slika 35. Robot u Gazebo simulaciji

5. Zaključak

Kroz ovaj rad istraženo je područje robotike, dizajna, simulacije i integracije robotskih sustava. Korištenje jednostavnog modela robotske ruke, 3D ispis i simulacija u Gazebo okruženju pružaju razumijevanje osnova robotskog kretanja, detekcije objekata i interakcije s okolinom. Integracija s Alexa sustavom dodaje dodatnu dimenziju komunikacije i interakcije, što je korisno za razumijevanje mogućnosti integriranja

robotskih sustava s drugim tehnologijama. Kroz ovaj rad usvojene su vještine u radu s robotskim simulacijama, programiranjem, komunikacijom s vanjskim uređajima i problemima integracije. Ovo iskustvo pruža mogućnost učenja i stvara osnovu za buduće istraživanje i daljnji razvoj u području robotike. Ovaj rad nije nužno o revolucionarnim inovacijama, već o koraku prema razumijevanju i dijeljenju znanja koje će pomoći drugima da se bolje upoznaju s robotikom i njezinim potencijalom.

Literatura

"Programming Robots with ROS: A Practical Introduction to the Robot Operating System" by Morgan Quigley, Brian Gerkey, William D. Smart

"ROS Robotics By Example" by Carol Fairchild

ROS wiki - http://wiki.ros.org/

MoveIt! Tutorials - https://ros-planning.github.io/moveit_tutorials/

ROS dokumentacija - http://docs.ros.org/en/

Robot Operating System (ROS) - Complete Developer's Guide – Udemy - https://www.udemy.com/course/ros-essentials/

Robotics Specialization – Coursera - https://www.coursera.org/specializations/robotics

Gazebo dokumentacija - https://gazebosim.org/docs/

Robotics and ROS - Towards Data Science - https://towardsdatascience.com/roboticsand-ros-8de5b00e46f9

Robot Grasping and Manipulation - A comprehensive review of robot grasping and manipulation techniques. - http://robotics.stanford.edu/~ang/papers/grasp-it.pdf

ROS-Industrial - https://github.com/ros-industrial

MoveIt! - https://moveit.ros.org/